

ApacheCon
Europe

Sinsheim, Germany 5th-8th November 2012

cloudera

HBASE SCHEMA DESIGN

and Cluster Sizing Notes

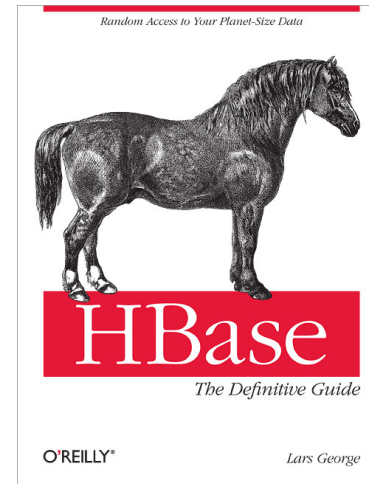
ApacheCon Europe, November 2012

Lars George

Director EMEA Services

About Me

- Director EMEA Services @ Cloudera
 - Consulting on Hadoop projects (everywhere)
- Apache Committer
 - HBase and Whirr
- O'Reilly Author
 - HBase – The Definitive Guide
 - Now in Japanese!
- Contact
 - lars@cloudera.com
 - [@larsgeorge](https://twitter.com/larsgeorge)



日本語版も出ました!

Agenda

- HBase Architecture
- Schema Design
- Cluster Sizing Notes

HBASE ARCHITECTURE



HBase Tables

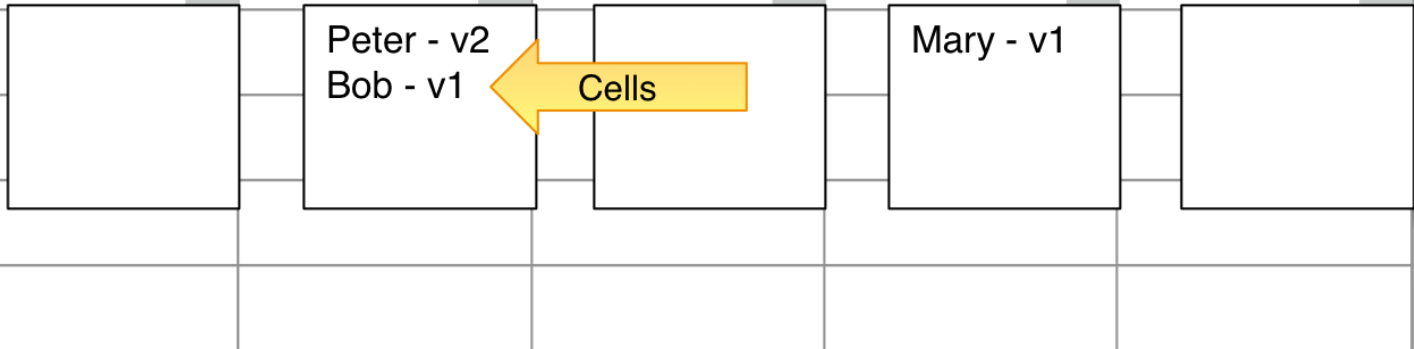
Row Keys

Column Names, aka Column Qualifiers, aka Column Keys

	col-A	col-B	col-Foo	col-XYZ	foobar
row-1					
row-10					
row-18					
row-2					
row-5					
row-6					
row-7					

HBase Tables

	col-A	col-B	col-Foo	col-XYZ	foobar
row-1					
row-10					
row-18	A18 - v1 ▼	B18 - v3 ▼	Foo18 - v1 ▼	XYZ18 - v2 ▼	foobar18 - v1 ▼
row-2		Peter - v2 Bob - v1		Mary - v1	
row-5					
row-6					
row-7					



Coordinates for a Cell: *Row Key* → *Column Name* → *Version*

HBase Tables

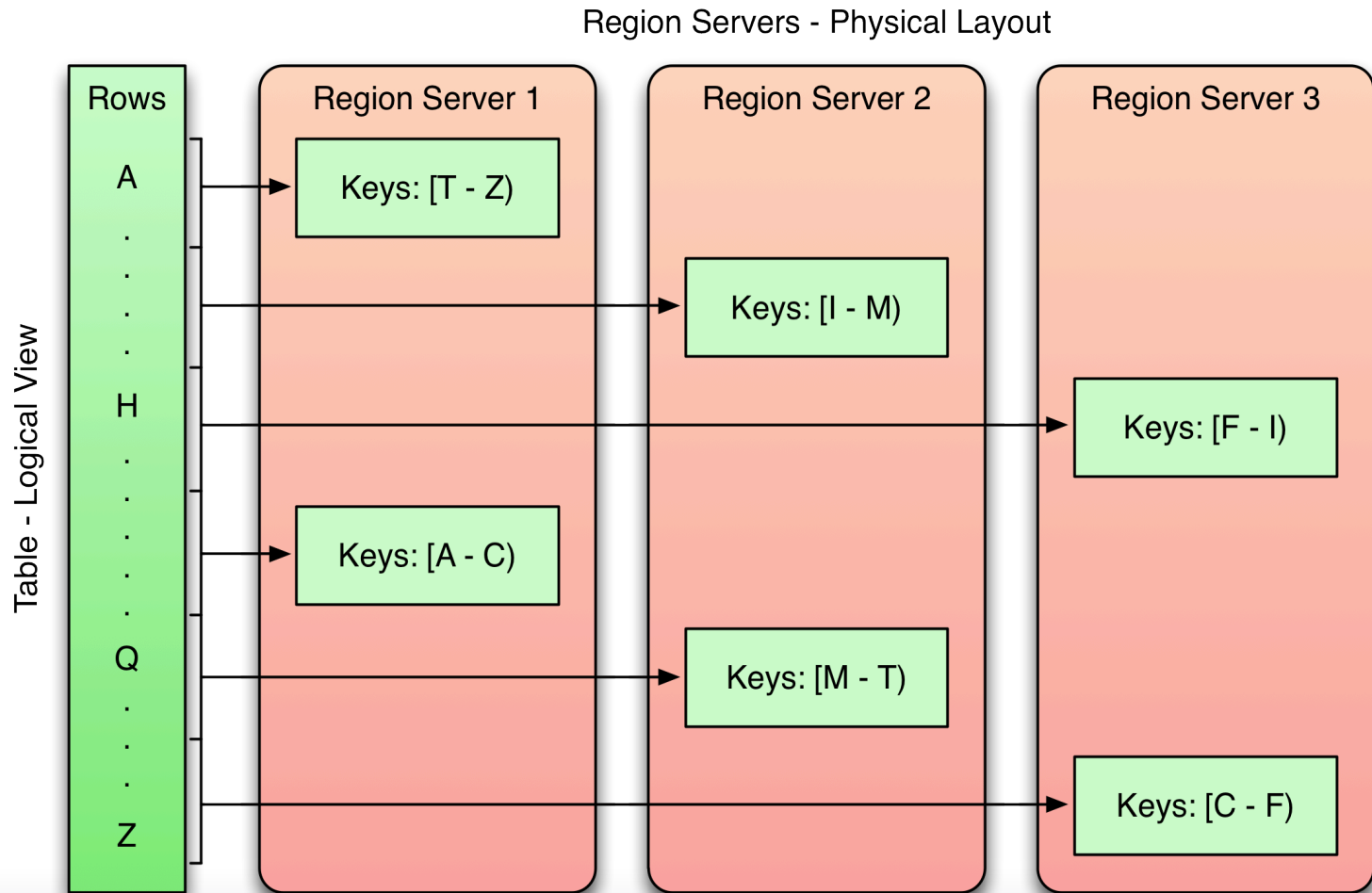
		Column Family 1		Column Family 2		
		cf1:col-A	cf1:col-B	cf2:col-Foo	cf2:col-XYZ	cf2:foobar
Region 1	row-1					
	row-10					
	row-18	A18 - v1 ▼	B18 - v3 ▼	Foo18 - v1 ▼	XYZ18 - v2 ▼	foobar18 - v1 ▼
Region 2	row-2					
	row-5					
	row-6					
	row-7					

Physical Coordinates for a Cell: *Region Directory* → *Column Family Directory*
→ *Row Key* → *Column Family Name* → *Column Qualifier* → *Version*

HBase Tables and Regions

- Table is made up of any number of regions
- Region is specified by its startKey and endKey
 - Empty table: (Table, NULL, NULL)
 - Two-region table: (Table, NULL, “com.cloudera.www”) and (Table, “com.cloudera.www”, NULL)
- Each region may live on a different node and is made up of several HDFS files and blocks, each of which is replicated by Hadoop

Distribution

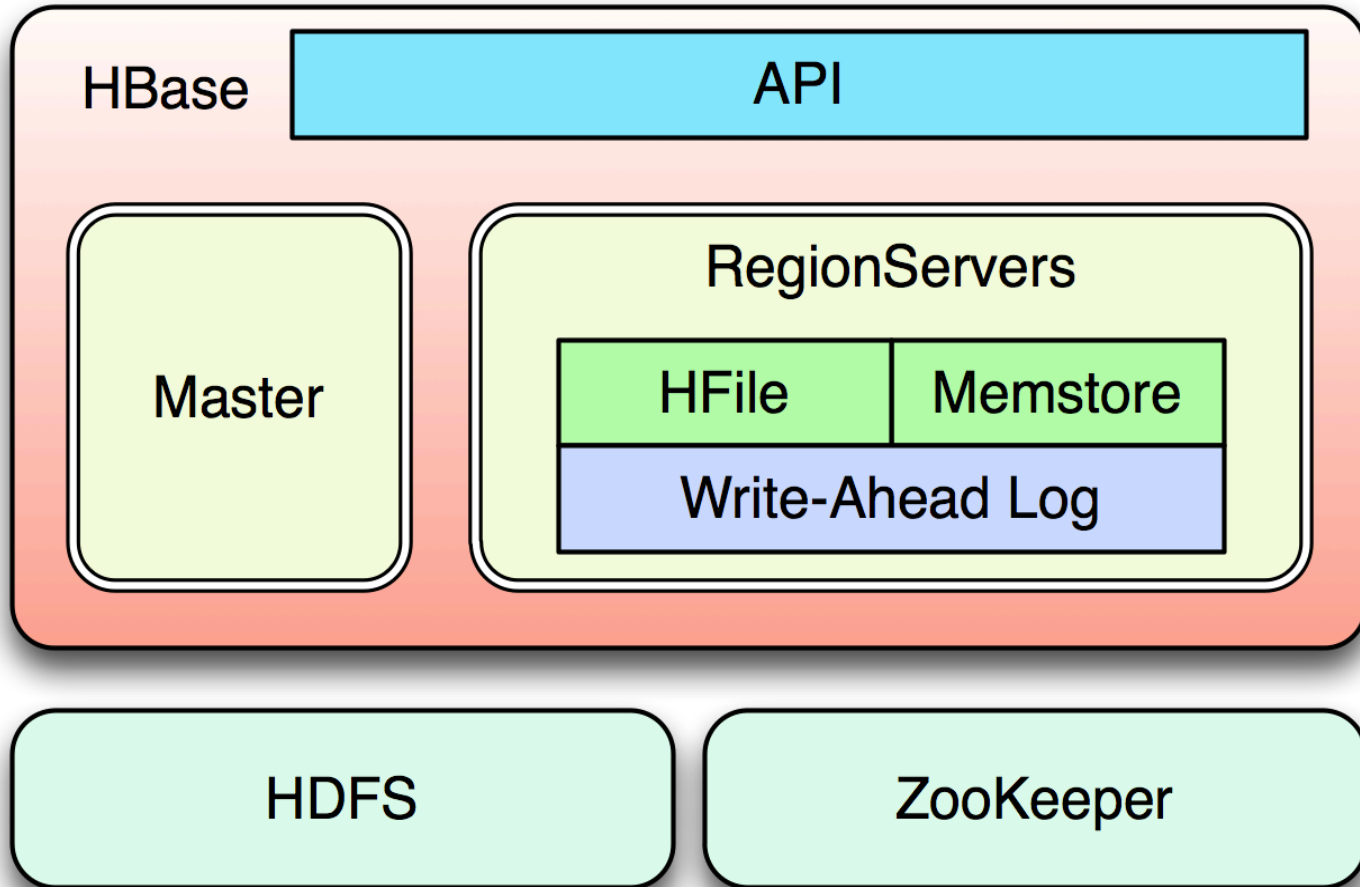


HBase Tables

- Tables are sorted by Row in lexicographical order
- Table schema only defines its column families
 - Each family consists of any number of columns
 - Each column consists of any number of versions
 - Columns only exist when inserted, NULLs are free
 - Columns within a family are sorted and stored together
 - Everything except table names are byte[]

(Table, Row, Family:Column, Timestamp) -> Value

HBase Architecture



HBase Architecture (cont.)

- HBase uses HDFS (or similar) as its reliable storage layer
 - Handles checksums, replication, failover
- Native Java API, Gateway for REST, Thrift, Avro
- Master manages cluster
- RegionServer manage data
- ZooKeeper is used the “neural network”
 - Crucial for HBase
 - Bootstraps and coordinates cluster

HBase Architecture (cont.)

- Based on Log-Structured Merge-Trees (LSM-Trees)
- Inserts are done in write-ahead log first
- Data is stored in memory and flushed to disk on regular intervals or based on size
- Small flushes are merged in the background to keep number of files small
- Reads read memory stores first and then disk based files second
- Deletes are handled with “tombstone” markers
- Atomicity on row level no matter how many columns
 - keeps locking model easy

MemStores

- After data is written to the WAL the RegionServer saves KeyValues in **memory store**
- Flush to disk based on size, see *hbase.hregion.memstore.flush.size*
- Default size is **64MB**
- Uses **snapshot** mechanism to write flush to disk while still serving from it and accepting new data at the same time
- Snapshots are released when flush has succeeded

Compactions

- General Concepts
 - Two types: **Minor** and **Major** Compactions
 - Asynchronous and transparent to client
 - Manage file bloat from MemStore flushes
- Minor Compactions
 - Combine last “few” flushes
 - Triggered by number of storage files
- Major Compactions
 - Rewrite **all** storage files
 - Drop deleted data and those values exceeding TTL and/or number of versions
 - Triggered by time threshold
 - Cannot be scheduled automatically starting at a specific time (bummer!)
 - May (most definitely) tax overall HDFS IO performance

Tip: Disable major compactions and schedule to run manually (e.g. cron) at off-peak times

Block Cache

- Acts as very large, in-memory **distributed cache**
- Assigned a large part of the JVM **heap** in the RegionServer process, see *hfile.block.cache.size*
- Optimizes **reads** on subsequent columns and rows
- Has **priority** to keep “in-memory” column families in cache

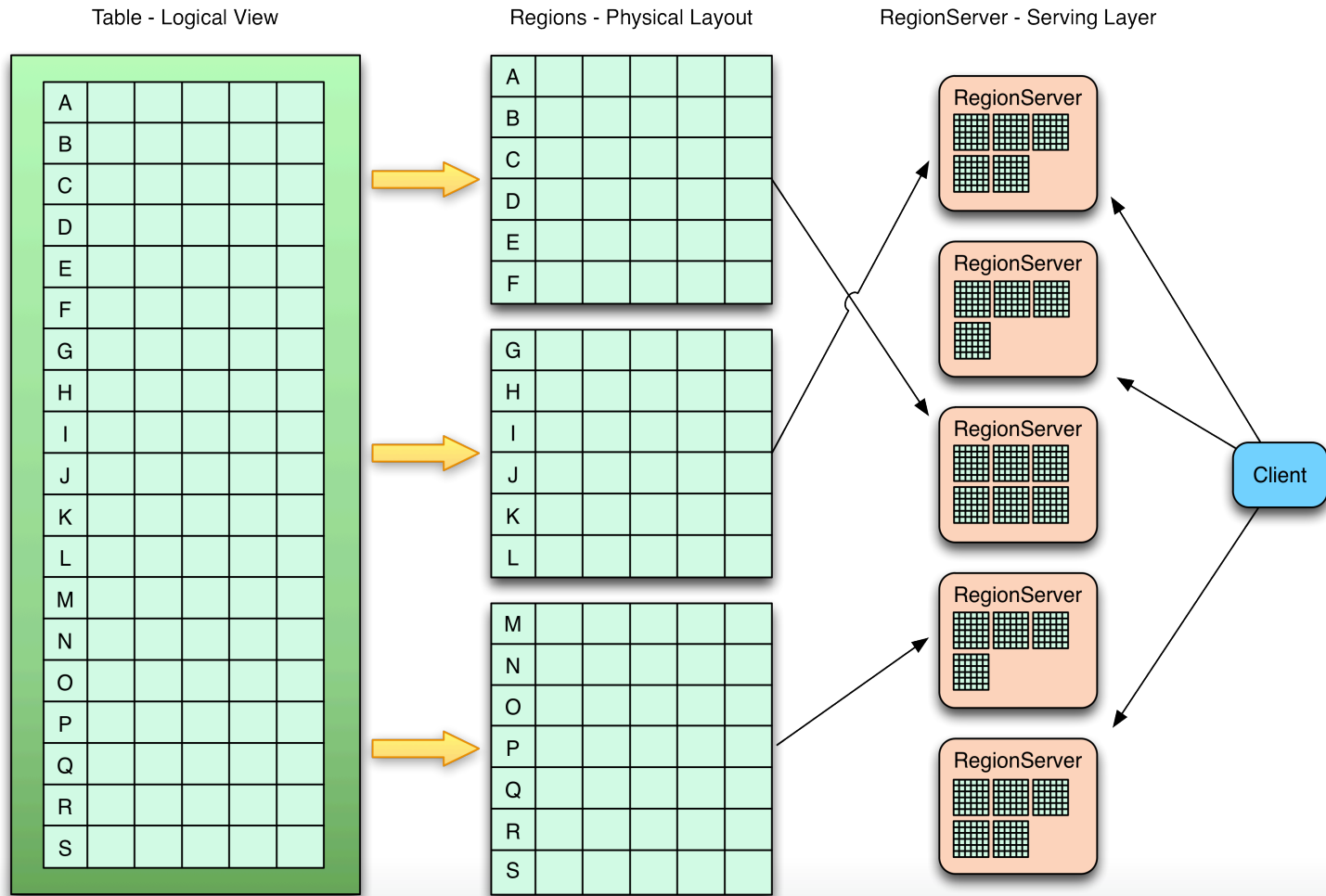
```
if(inMemory) {
    this.priority = BlockPriority.MEMORY;
} else {
    this.priority = BlockPriority.SINGLE;
}
```

- Cache needs to be used properly to get best read performance
 - Turn off block cache on operations that cause large churn
 - Store related data “close” to each other
- Uses **LRU** cache with threaded (asynchronous) evictions based on priorities

Region Splits

- Triggered by configured maximum file size of any store file
 - This is checked directly **after** the compaction call to ensure store files are actually approaching the threshold
- Runs as **asynchronous** thread on RegionServer
- Splits are **fast** and nearly instant
 - Reference files point to original region files and represent each half of the split
- Compactions take care of splitting original files into new region directories

Auto Sharding



Auto Sharding and Distribution

- Unit of scalability in HBase is the *Region*
- Sorted, contiguous range of rows
- Spread “randomly” across RegionServer
- Moved around for load balancing and failover
- Split automatically or manually to scale with growing data
- Capacity is solely a factor of cluster nodes vs. regions per node

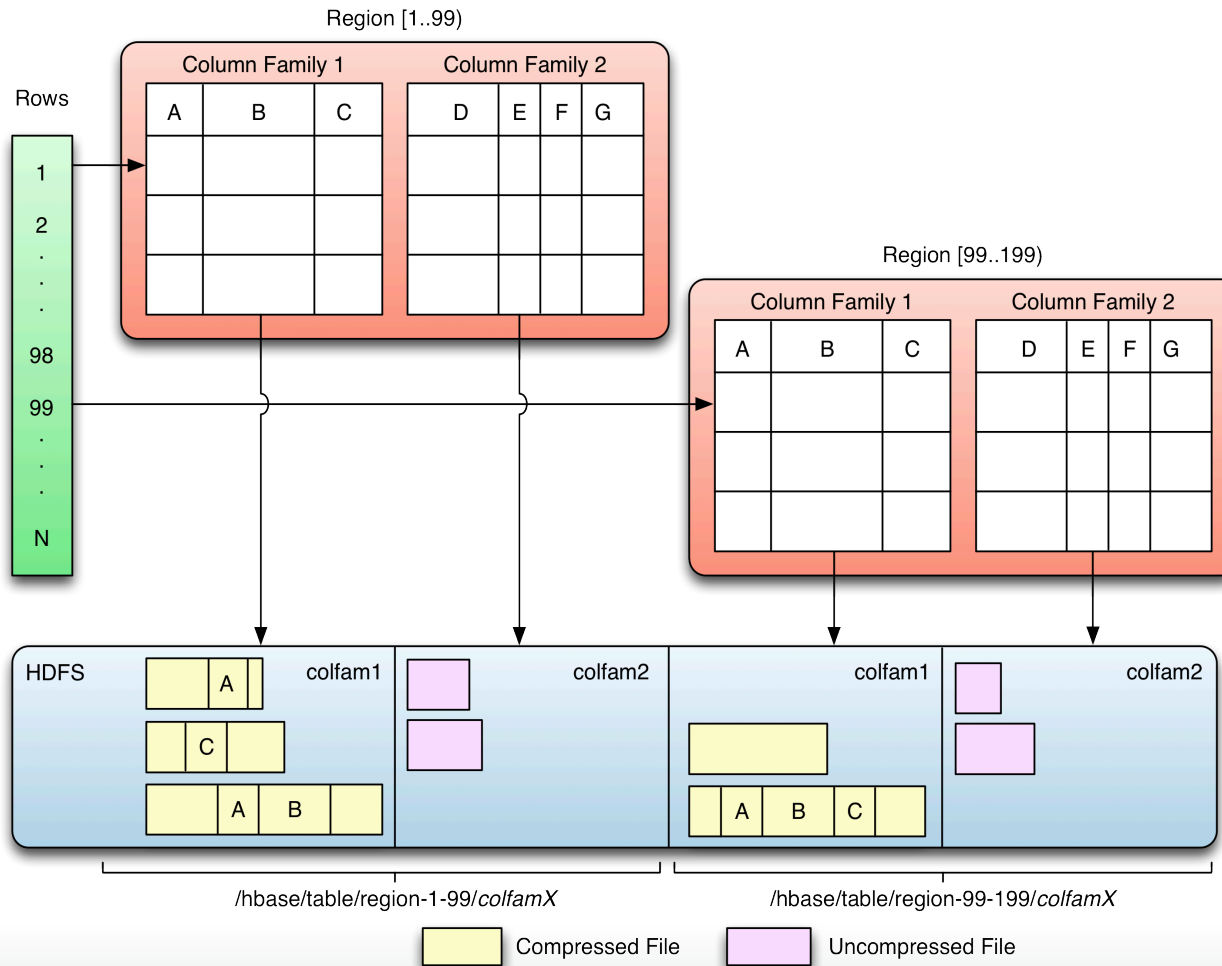
Column Family vs. Column

- Use only a few column families
 - Causes many files that need to stay open per region plus class overhead per family
- Best used when logical separation between data and meta columns
- Sorting per family can be used to convey application logic or access pattern

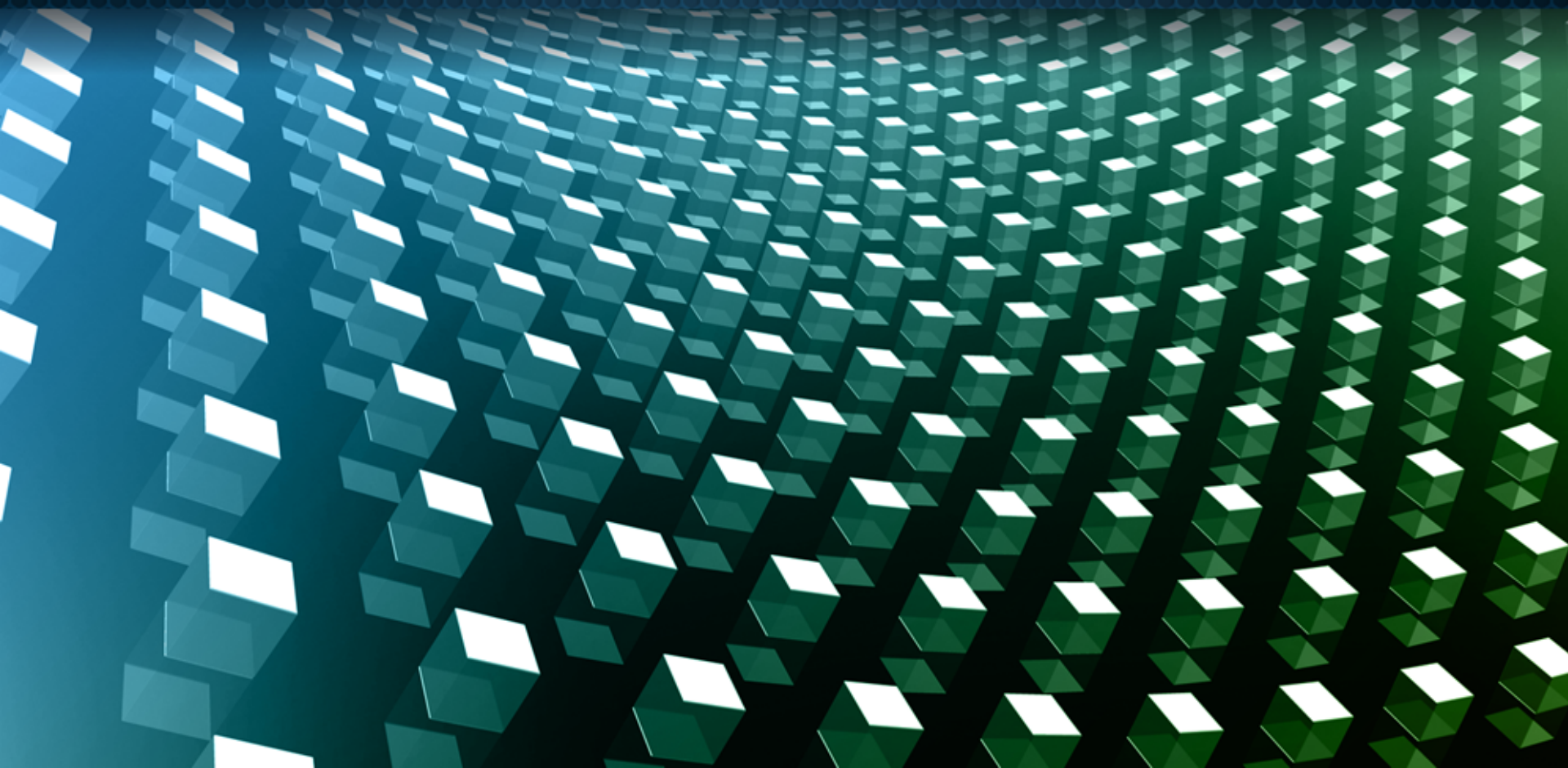
Storage Separation

- Column Families allow for separation of data
 - Used by Columnar Databases for fast analytical queries, but on column level only
 - Allows different or no compression depending on the content type
- Segregate information based on access pattern
- Data is stored in one or more storage file, called HFiles

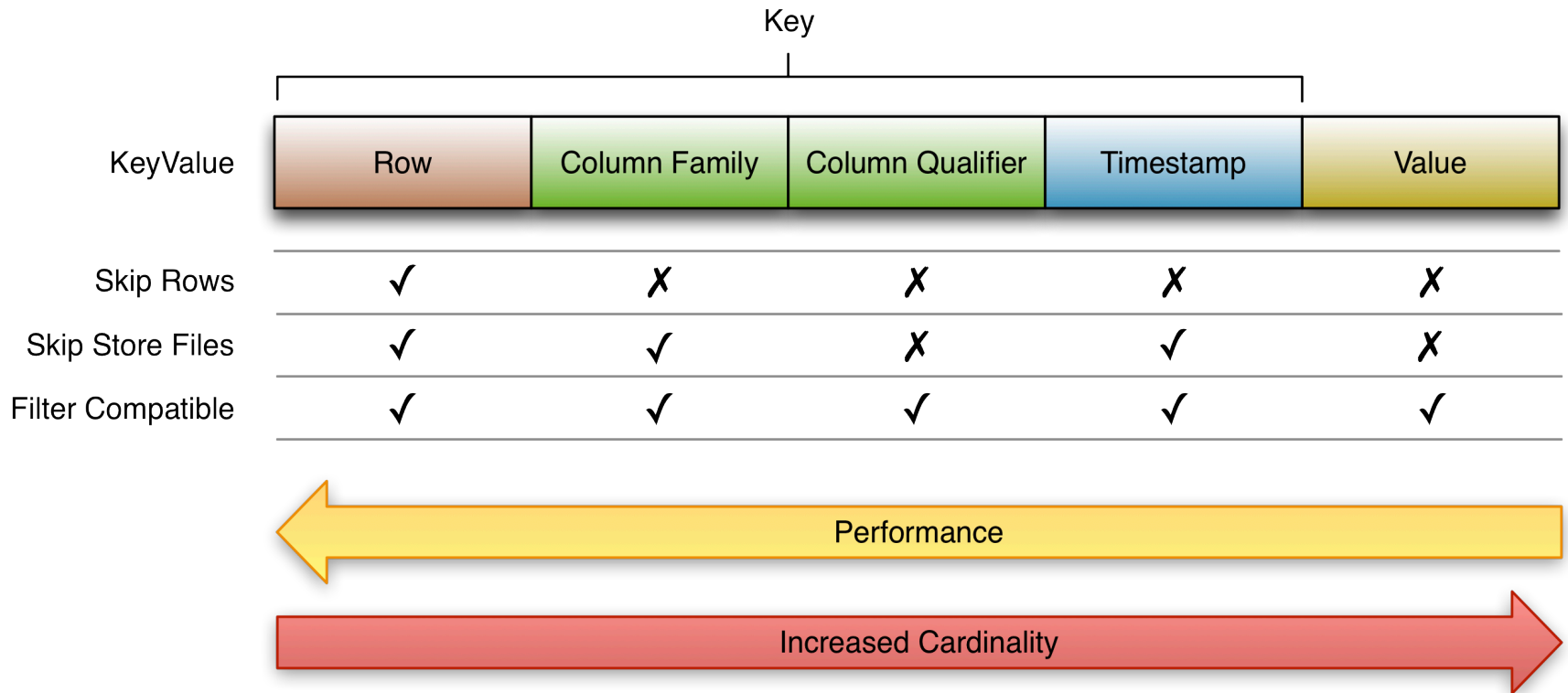
Column Families



SCHEMA DESIGN



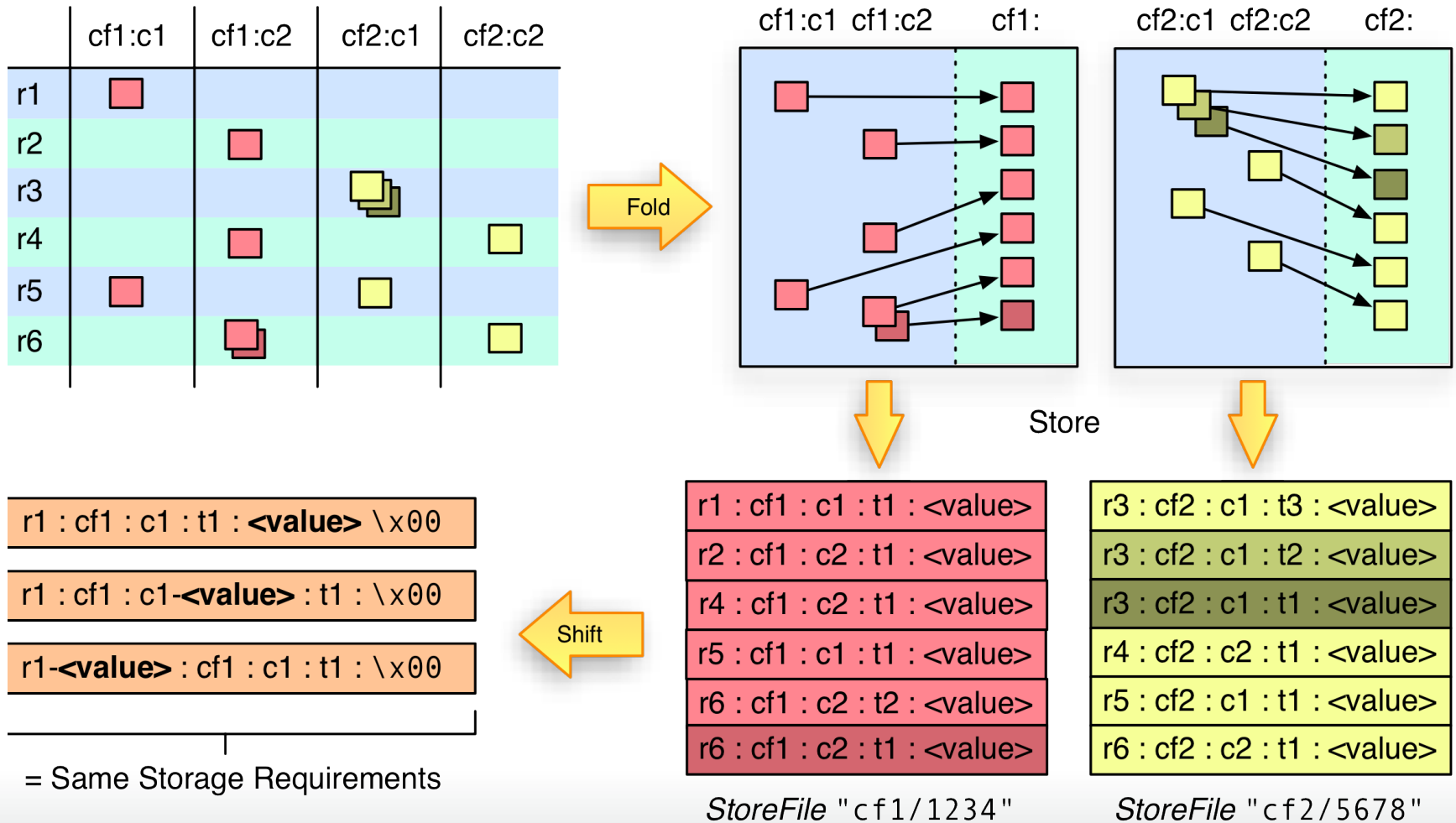
Key Cardinality



Key Cardinality

- The best performance is gained from using row keys
- Time range bound reads can skip store files
 - So can Bloom Filters
- Selecting column families reduces the amount of data to be scanned
- Pure value based filtering is a full table scan
 - Filters often are too, but reduce network traffic

Fold, Store, and Shift



Fold, Store, and Shift

- Logical layout does not match physical one
- All values are stored with the full coordinates, including: Row Key, Column Family, Column Qualifier, and Timestamp
- Folds columns into “row per column”
- NULLs are cost free as nothing is stored
- Versions are multiple “rows” in folded table

Key/Table Design

- Crucial to gain best performance
 - Why do I need to know? Well, you also need to know that RDBMS is only working well when columns are indexed and query plan is OK
- Absence of secondary indexes forces use of *row key* or *column name* sorting
- Transfer multiple indexes into one
 - Generate large table -> Good since fits architecture and spreads across cluster

DDI

- Stands for Denormalization, Duplication and Intelligent Keys
- Needed to overcome shortcomings of architecture
- Denormalization -> Replacement for JOINS
- Duplication -> Design for reads
- Intelligent Keys -> Implement indexing and sorting, optimize reads

Pre-materialize Everything

- Achieve one read per customer request if possible
- Otherwise keep at lowest number
- Reads between 10ms (cache miss) and 1ms (cache hit)
- Use MapReduce to compute exacts in batch
- Store and merge updates live
- Use incrementColumnValue

Motto: “Design for **Reads**”

Tall-Narrow vs. Flat-Wide Tables

- Rows do not split
 - Might end up with one row per region
- Same storage footprint
- Put more details into the row key
 - Sometimes *dummy* column only
 - Make use of partial key scans
- Tall with Scans, Wide with Gets
 - Atomicity only on row level
- Example: Large graphs, stored as adjacency matrix

Example: Mail Inbox

```
<userId> : <colfam> : <messageId> : <timestamp> : <email-message>
```

```
12345 : data : 5fc38314-e290-ae5da5fc375d : 1307097848 : "Hi Lars, ..."  
12345 : data : 725aae5f-d72e-f90f3f070419 : 1307099848 : "Welcome, and ..."  
12345 : data : cc6775b3-f249-c6dd2b1a7467 : 1307101848 : "To Whom It ..."  
12345 : data : dcbee495-6d5e-6ed48124632c : 1307103848 : "Hi, how are ..."
```

or

```
12345-5fc38314-e290-ae5da5fc375d : data : : 1307097848 : "Hi Lars, ..."  
12345-725aae5f-d72e-f90f3f070419 : data : : 1307099848 : "Welcome, and ..."  
12345-cc6775b3-f249-c6dd2b1a7467 : data : : 1307101848 : "To Whom It ..."  
12345-dcbee495-6d5e-6ed48124632c : data : : 1307103848 : "Hi, how are ..."
```

→ Same Storage Requirements

Partial Key Scans

Key	Description
<code><userId></code>	Scan over all messages for a given user ID
<code><userId>-<date></code>	Scan over all messages on a given date for the given user ID
<code><userId>-<date>-<messageId></code>	Scan over all parts of a message for a given user ID and date
<code><userId>-<date>-<messageId>-<attachmentId></code>	Scan over all attachments of a message for a given user ID and date

Sequential Keys

`<timestamp><more key>: {CF: {CQ: {TS : Val}}}`

- Hotspotting on Regions: **bad!**
- Instead do one of the following:
 - Salting
 - Prefix `<timestamp>` with distributed value
 - Binning or bucketing rows across regions
 - Key field swap/promotion
 - Move `<more key>` before the timestamp (see OpenTSDB later)
 - Randomization
 - Move `<timestamp>` out of key

Salting

- Prefix row keys to gain spread
- Use well known or numbered prefixes
- Use modulo to spread across servers
- Enforce common data stay close to each other for subsequent scanning or MapReduce processing

```
0_rowkey1, 1_rowkey2, 2_rowkey3  
0_rowkey4, 1_rowkey5, 2_rowkey6
```

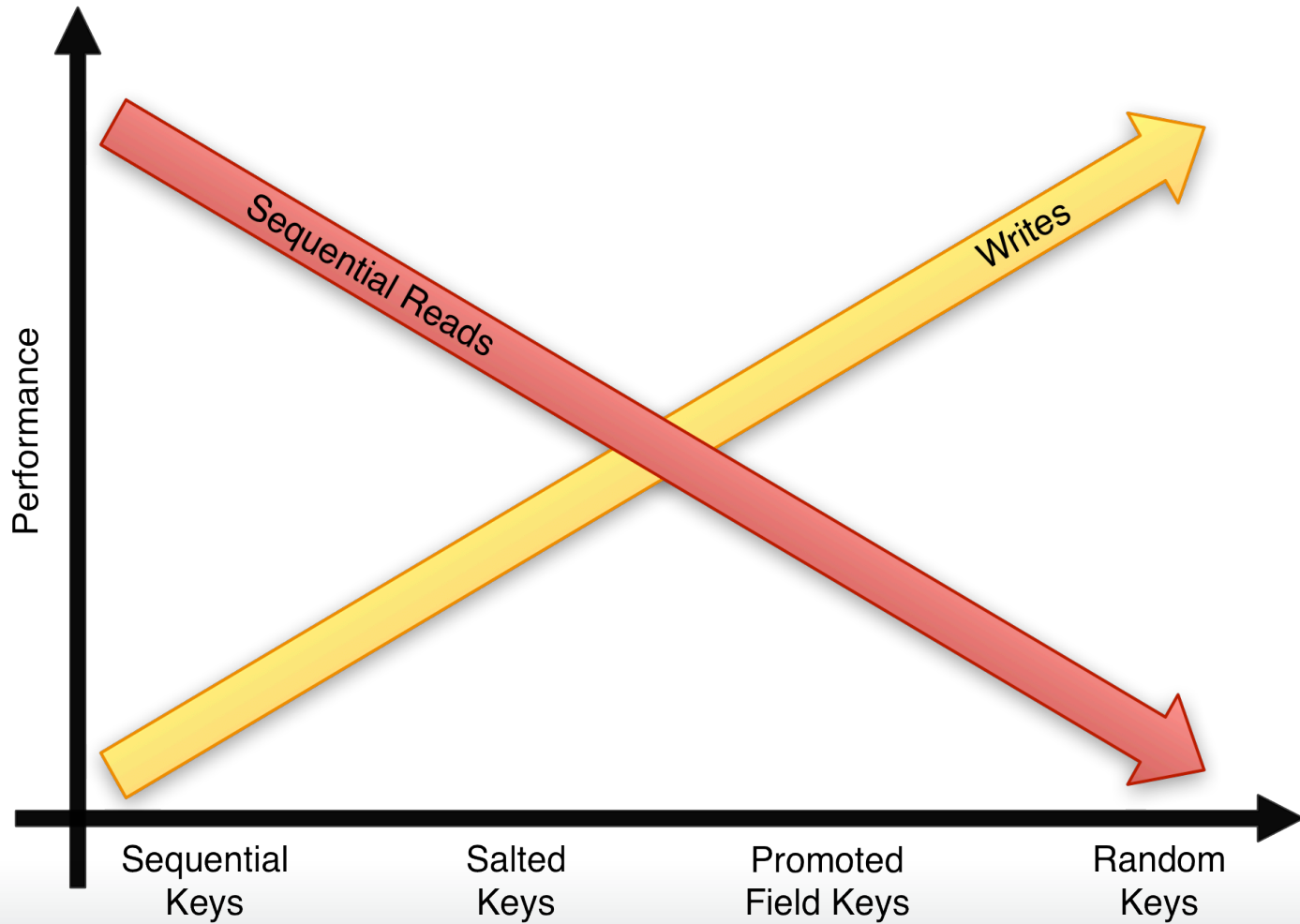
- Sorted by prefix first

```
0_rowkey1  
0_rowkey4  
1_rowkey2  
1_rowkey5  
...
```

Hashing vs. Sequential Keys

- Uses hashes for best spread
 - Use for example MD5 to be able to recreate key
 - Key = MD5(customerID)
 - Counter productive for range scans
- Use sequential keys for locality
 - Makes use of block caches
 - May tax one server overly, may be avoided by salting or splitting regions while keeping them small

Key Design



Key Design Summary

- Based on access pattern, either use sequential or random keys
- Often a combination of both is needed
 - Overcome architectural limitations
- Neither is necessarily bad
 - Use bulk import for sequential keys and reads
 - Random keys are good for random access patterns

Example: Facebook Insights

- > 20B Events per Day
- 1M Counter Updates per Second
 - 100 Nodes Cluster
 - 10K OPS per Node
- "Like" button triggers AJAX request
- Event written to log file
- 30mins current for website owner

Web → Scribe → Ptail → Puma → HBase

HBase Counters

- Store counters per Domain and per URL
 - Leverage HBase *increment* (atomic read-modify-write) feature
- Each row is one specific Domain or URL
- The columns are the counters for specific metrics
- Column families are used to group counters by time range
 - Set time-to-live on CF level to auto-expire counters by age to save space, e.g., 2 weeks on “Daily Counters” family

Key Design

- **Reversed Domains**

- Examples: “com.cloudera.www”, “com.cloudera.blog”
- Helps keeping pages *per site* close, as HBase efficiently scans blocks of sorted keys

- **Domain Row Key =**

MD5(Reversed Domain) + Reversed Domain

- Leading MD5 hash spreads keys randomly across all regions for load balancing reasons
- Only hashing the domain groups per site (and per subdomain if needed)

- **URL Row Key =**

MD5(Reversed Domain) + Reversed Domain + URL ID

- Unique ID per URL already available, make use of it

Insights Schema

Row Key: *Domain Row Key*

Columns:

Hourly Counters CF					Daily Counters CF					Lifetime Counters CF				
6pm Total	6pm Male	6pm US	7pm	1/1 Total	1/1 Male	1/1 US	2/1	Total	Male	Female	US	...
100	50	92	45		1000	320	670	990		10000	6780	3220	9900	

Row Key: *URL Row Key*

Columns:

Hourly Counters CF					Daily Counters CF					Lifetime Counters CF				
6pm Total	6pm Male	6pm US	7pm	1/1 Total	1/1 Male	1/1 US	2/1	Total	Male	Female	US	...
10	5	9	4		100	20	70	99		100	8	92	100	

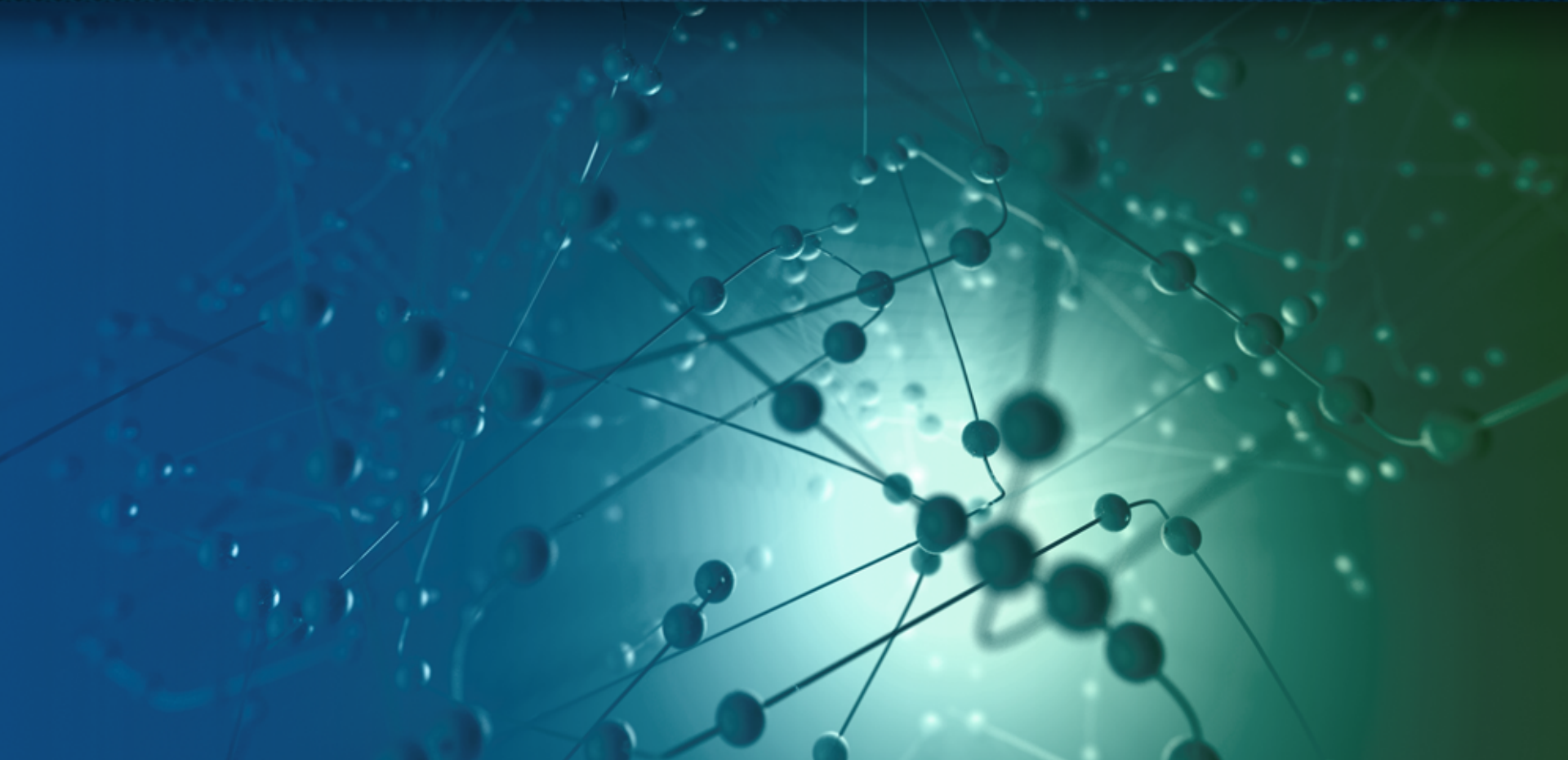
Summary

- Design for Use-Case
 - Read, Write, or Both?
- Avoid Hotspotting
- Consider using IDs instead of full text
- Leverage Column Family to HFile relation
- Shift details to appropriate position
 - Composite Keys
 - Column Qualifiers

Summary (cont.)

- Schema design is a combination of
 - Designing the keys (row and column)
 - Segregate data into column families
 - Choose compression and block sizes
- Similar techniques are needed to scale most systems
 - Add indexes, partition data, consistent hashing
- Denormalization, Duplication, and Intelligent Keys (DDI)

CLUSTER SIZING



Competing Resources

- Reads and Writes compete for the same low-level resources
 - Disk (HDFS) and Network I/O
 - RPC Handlers and Threads
- Otherwise they do exercise completely separate code paths

Memory Sharing

- By default every region server is dividing its memory (i.e. given maximum heap) into
 - 40% for in-memory stores (write ops)
 - 20% for block caching (reads ops)
 - remaining space (here 40%) go towards usual Java heap usage (objects etc.)
- Share of memory needs to be tweaked

Reads

- Locate and route request to appropriate region server
 - Client caches information for faster lookups → consider prefetching option for fast warmups
- Eliminate store files if possible using time ranges or Bloom filter
- Try **block cache**, if block is missing then load from disk

Block Cache

- Use exported metrics to see effectiveness of block cache
 - Check fill and eviction rate, as well as hit ratios → random reads are not ideal
- Tweak up or down as needed, but watch overall heap usage
- You absolutely need the block cache
 - Set to 10% at least for short term benefits

Writes

- The cluster size is often determined by the write performance
- Log structured merge trees like
 - Store mutation in **in-memory store** and **write-ahead log**
 - Flush out aggregated, sorted maps at specified threshold - or - when under pressure
 - Discard logs with no pending edits
 - Perform regular compactions of store files

Write Performance

- There are many factors to the overall write performance of a cluster
 - Key Distribution → Avoid region hotspot
 - Handlers → Do not pile up too early
 - Write-ahead log → Bottleneck #1
 - Compactions → Badly tuned can cause ever increasing background noise

Write-Ahead Log

- Currently only one per region server
 - Shared across all stores (i.e. column families)
 - Synchronized on file append calls
- Work being done on mitigating this
 - WAL Compression
 - Multiple WAL's per region server → Start more than one region server per node?

Write-Ahead Log (cont.)

- Size set to 95% of default block size
 - 64MB or 128MB, but check config!
- Keep number low to reduce recovery time
 - Limit set to 32, but can be increased
- Increase size of logs - and/or - increase the number of logs before blocking
- Compute number based on fill distribution and flush frequencies

Write-Ahead Log (cont.)

- Writes are synchronized across all stores
 - A large cell in one family can stop all writes of another
 - In this case the RPC handlers go binary, i.e. either work or all block
- *Can* be bypassed on writes, but means no *real* durability and no replication
 - Maybe use coprocessor to restore dependent data sets (preWALRestore)

Flushes

- Every mutation call (put, delete etc.) causes a check for a flush
- If threshold is met, flush file to disk and schedule a compaction
 - Try to compact newly flushed files quickly
- The compaction returns - if necessary - where a region should be split

Compaction Storms

- Premature flushing because of # of logs or memory pressure
 - Files will be smaller than the configured flush size
- The background compactions are hard at work merging small flush files into the existing, larger store files
 - Rewrite hundreds of MB over and over

Dependencies

- Flushes happen across all stores/column families, even if just one triggers it
- The flush size is compared to the size of all stores combined
 - Many column families dilute the size
 - Example: 55MB + 5MB + 4MB

Some Numbers

- Typical write performance of HDFS is 35-50MB/s

Cell Size	OPS
0.5MB	70-100
100KB	350-500
10KB	3500-5000 ??
1KB	35000-50000 ????

This is way to high in practice - Contention!

Some More Numbers

- Under real world conditions the rate is less, more like 15MB/s or less
- Thread contention is cause for massive slow down

Cell Size	OPS
0.5MB	10
100KB	100
10KB	800
1KB	6000

Notes

- Compute memstore sizes based on number of regions x flush size
- Compute number of logs to keep based on fill and flush rate
- Ultimately the capacity is driven by
 - Java Heap
 - Region Count and Size
 - Key Distribution

Cheat Sheet #1

- Ensure you have enough or large enough write-ahead logs
- Ensure you do not oversubscribe available memstore space
- Ensure to set flush size large enough but not too large
- Check write-ahead log usage carefully

Cheat Sheet #2

- Enable compression to store more data per node
- Tweak compaction algorithm to peg background I/O at some level
- Consider putting uneven column families in separate tables
- Check metrics carefully for block cache, memstore, and all queues

Example

- Java Xmx heap at 10GB
- Memstore share at 40% (default)
 - $10\text{GB Heap} \times 0.4 = 4\text{GB}$
- Desired flush size at 128MB
 - $4\text{GB} / 128\text{MB} = 32$ regions max!
- For WAL size of 128MB x 0.95%
 - $4\text{GB} / (128\text{MB} \times 0.95) = \sim 33$ partially uncommitted logs to keep around
- Region size at 20GB
 - $20\text{GB} \times 32$ regions = 640GB raw storage used

A scenic landscape photograph featuring a large mountain in the background with the sun setting behind its peak, creating a bright lens flare. In the foreground, a river flows through a lush green field. The word "Questions?" is overlaid in white text in the center of the image.

Questions?