

# HBase Schema Design

NoSQL Matters, Cologne, April 2013

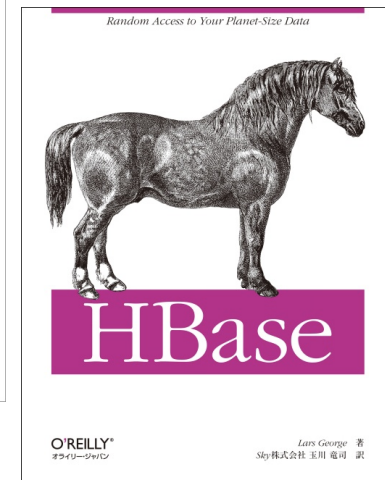
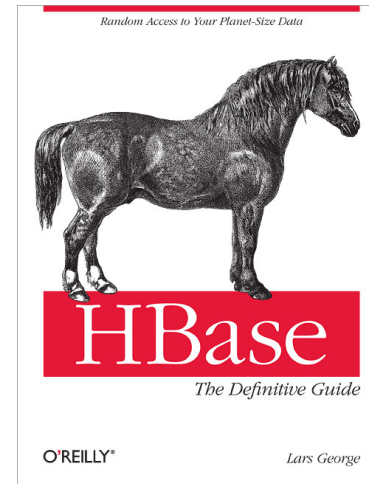
Lars George

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](mailto:lars@cloudera.com)
  - @larsgeorge



日本語版も出ました!

# Agenda

---

- HBase Architecture
- Schema Design
- Summary

# HBase Architecture

---

# HBase Tables

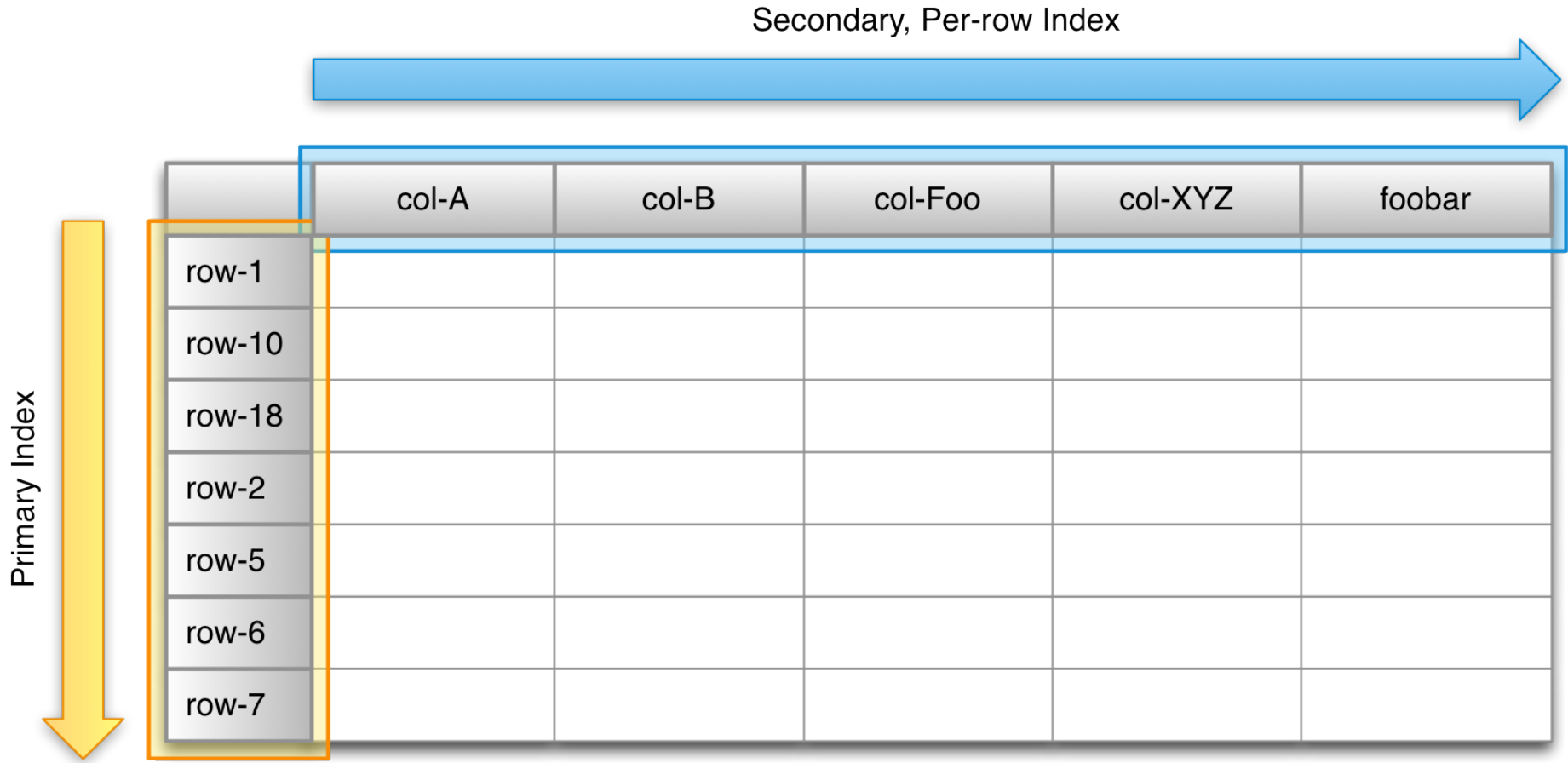
Row Keys

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

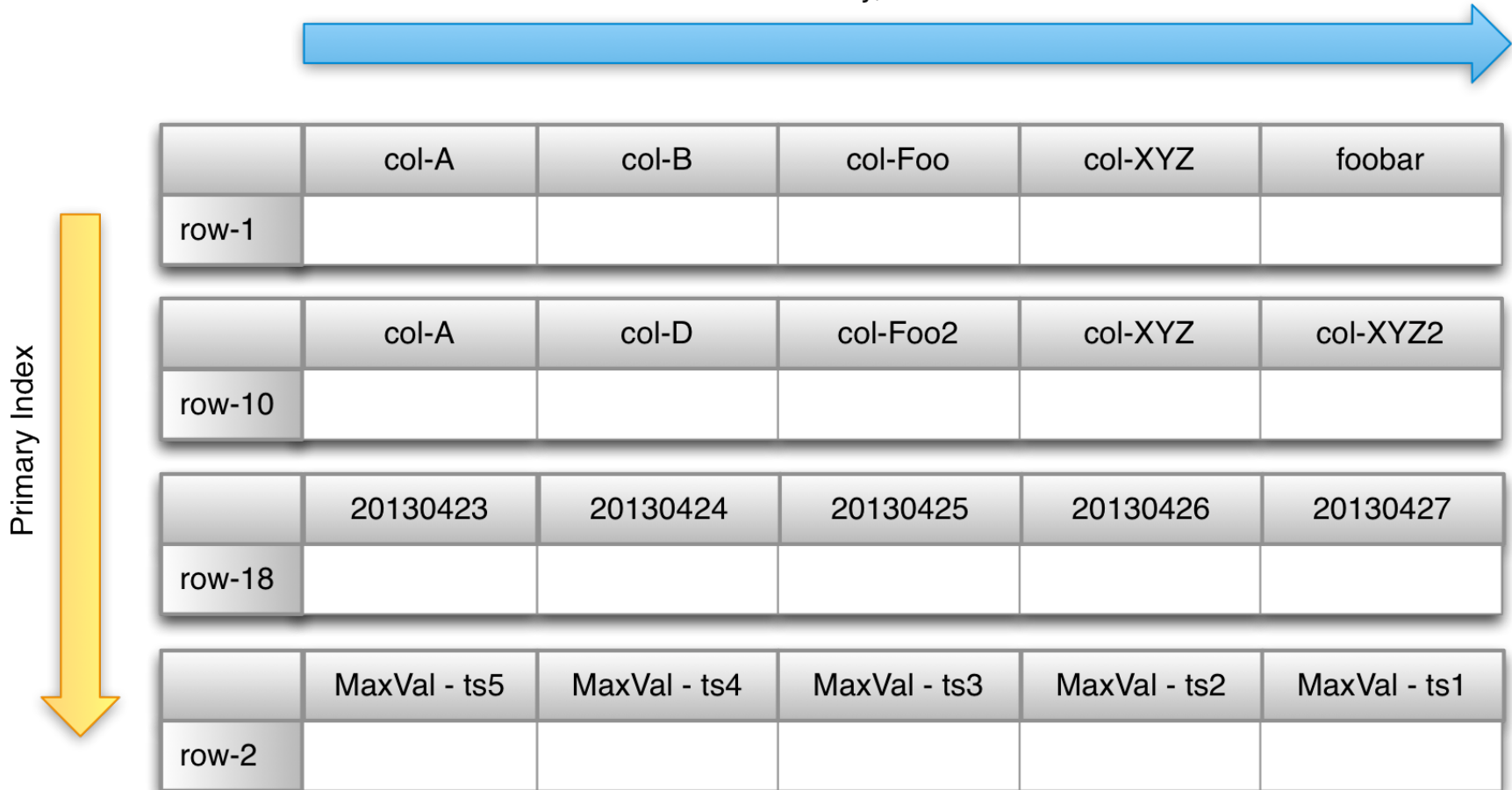
Ascending, Lexicographically Sorted Indexes



# HBase Tables

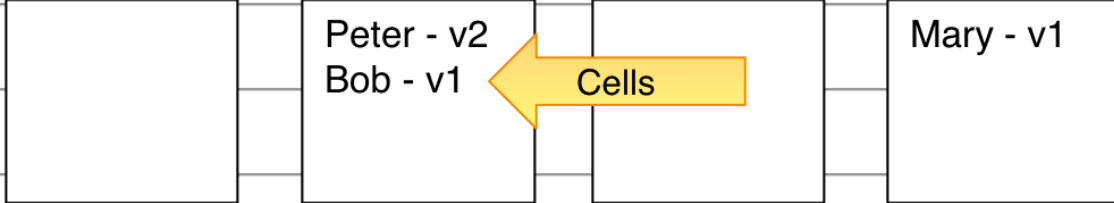
Ascending, Lexicographically Sorted Indexes

Secondary, Per-row Index



# HBase Tables

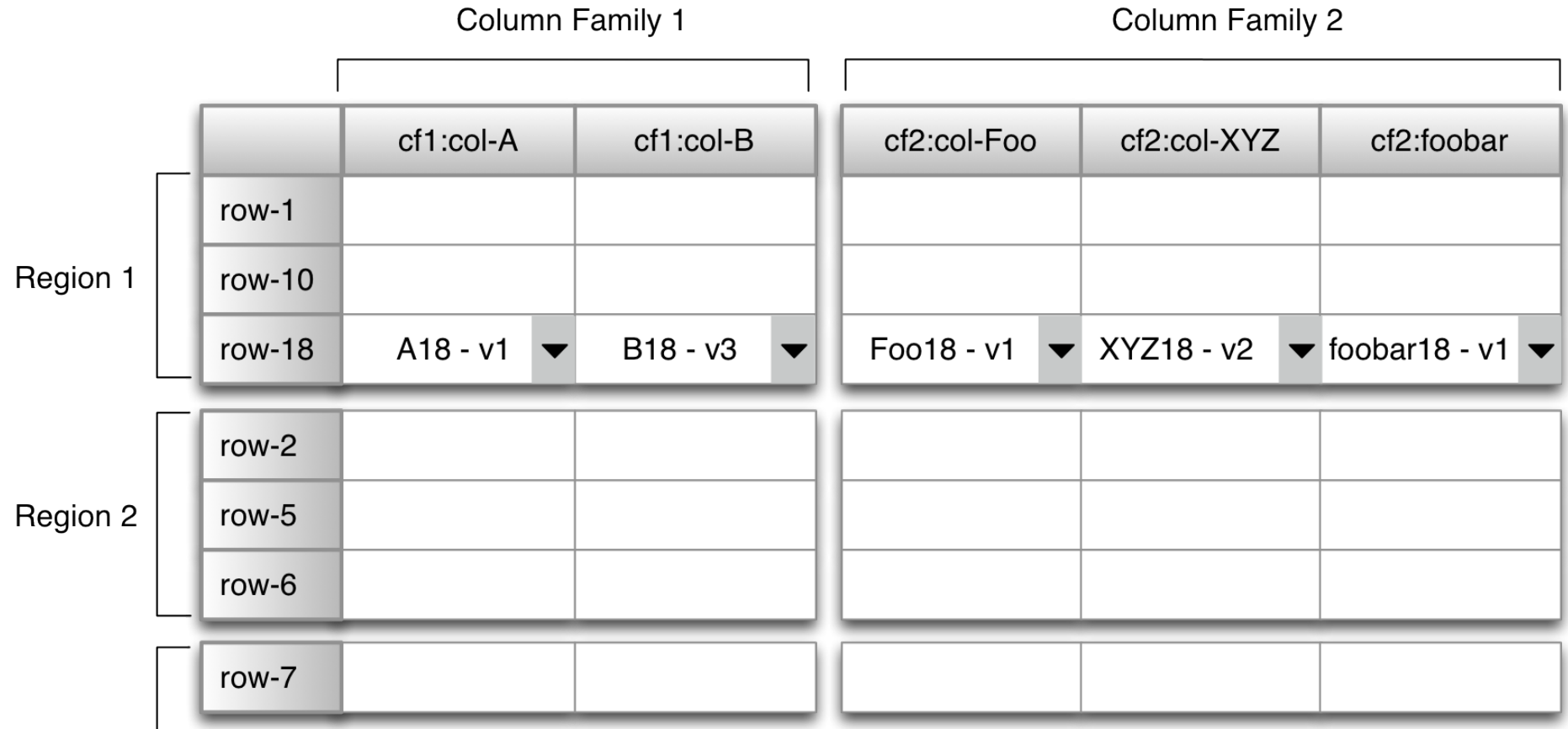
	col-A	col-B	col-Foo	col-XYZ	foobar
row-1					
row-10					
row-18	A18 - v1 ▼	B18 - v3 ▼	Foo18 - v1 ▼	XYZ18 - v2 ▼	foobar18 - v1 ▼
row-2		Peter - v2 Bob - v1		Mary - v1	
row-5					
row-6					
row-7					

A yellow arrow labeled "Cells" points to the cell at row-2, col-B. This cell contains two lines of text: "Peter - v2" and "Bob - v1".

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

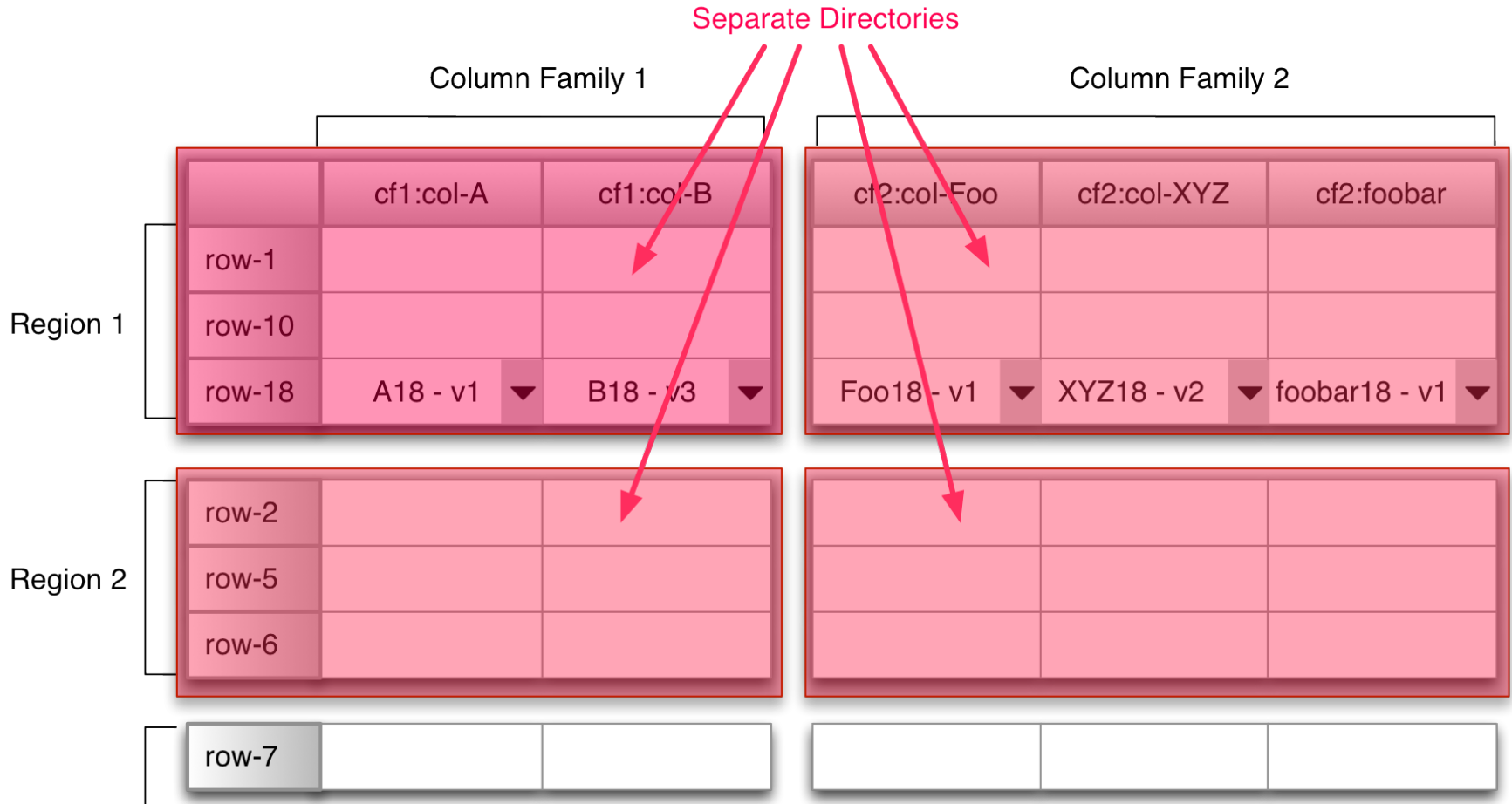


# HBase Tables



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

# HBase Tables



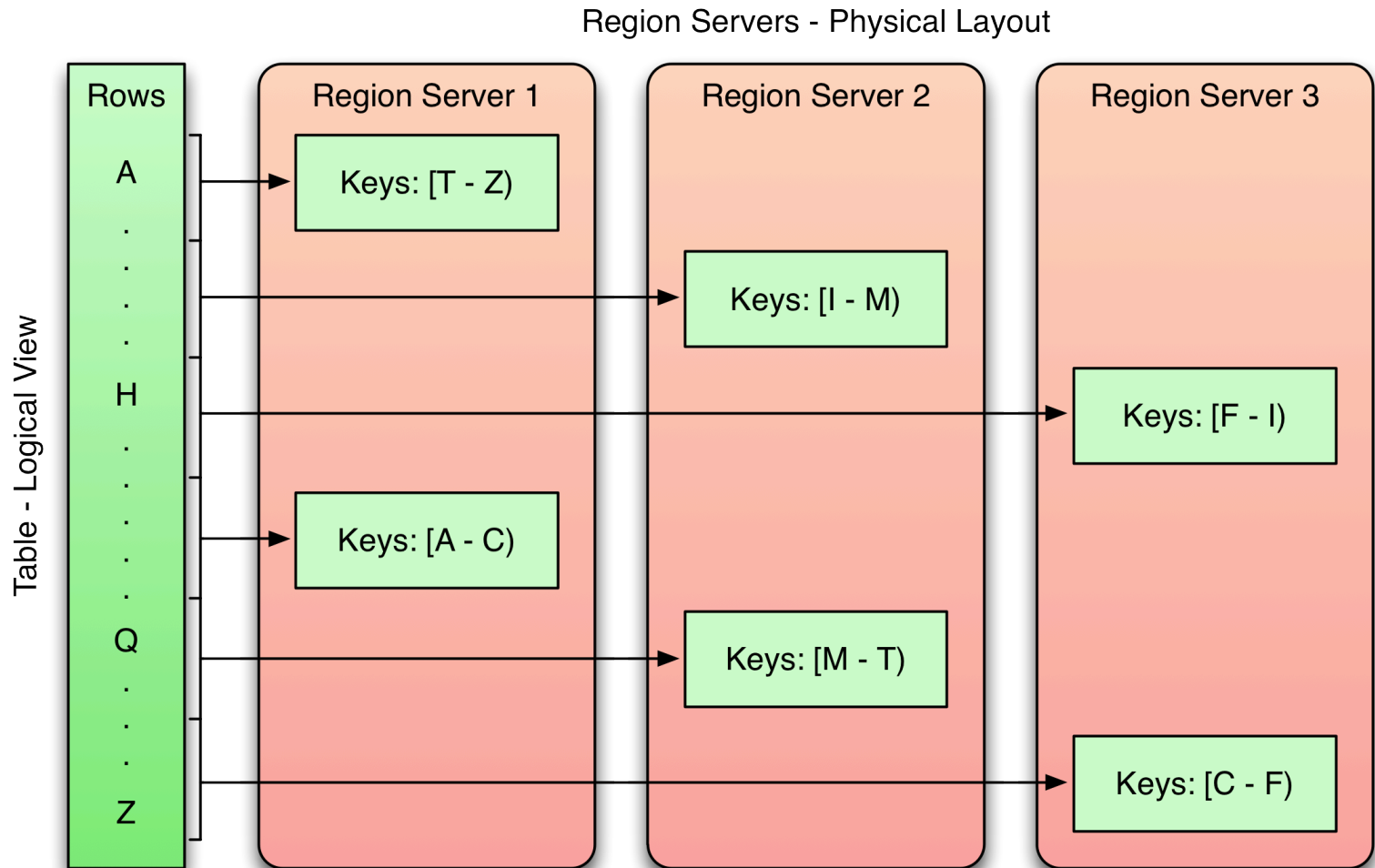
Physical Coordinates for a Cell: *Region Directory* → *Column Family Directory*  
→ *Row Key* → *Column Family Name* → *Column Qualifier* → *Version*

# 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



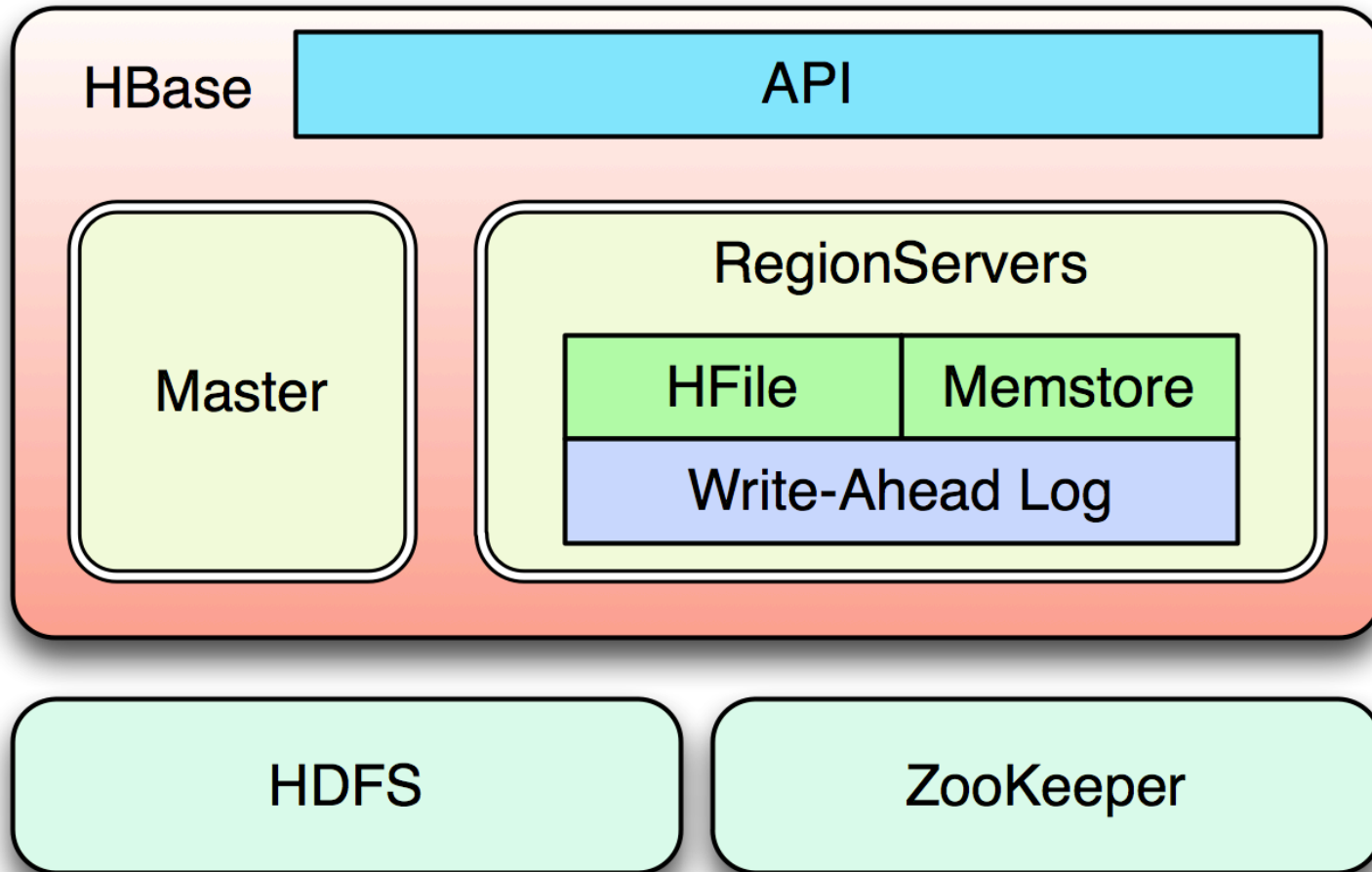
# HBase Tables

---

- 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 (MemStores) and flushed to disk on regular intervals or based on size
- Small flushes are merged in the background to keep number of files small (Compactions)
- 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



# 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

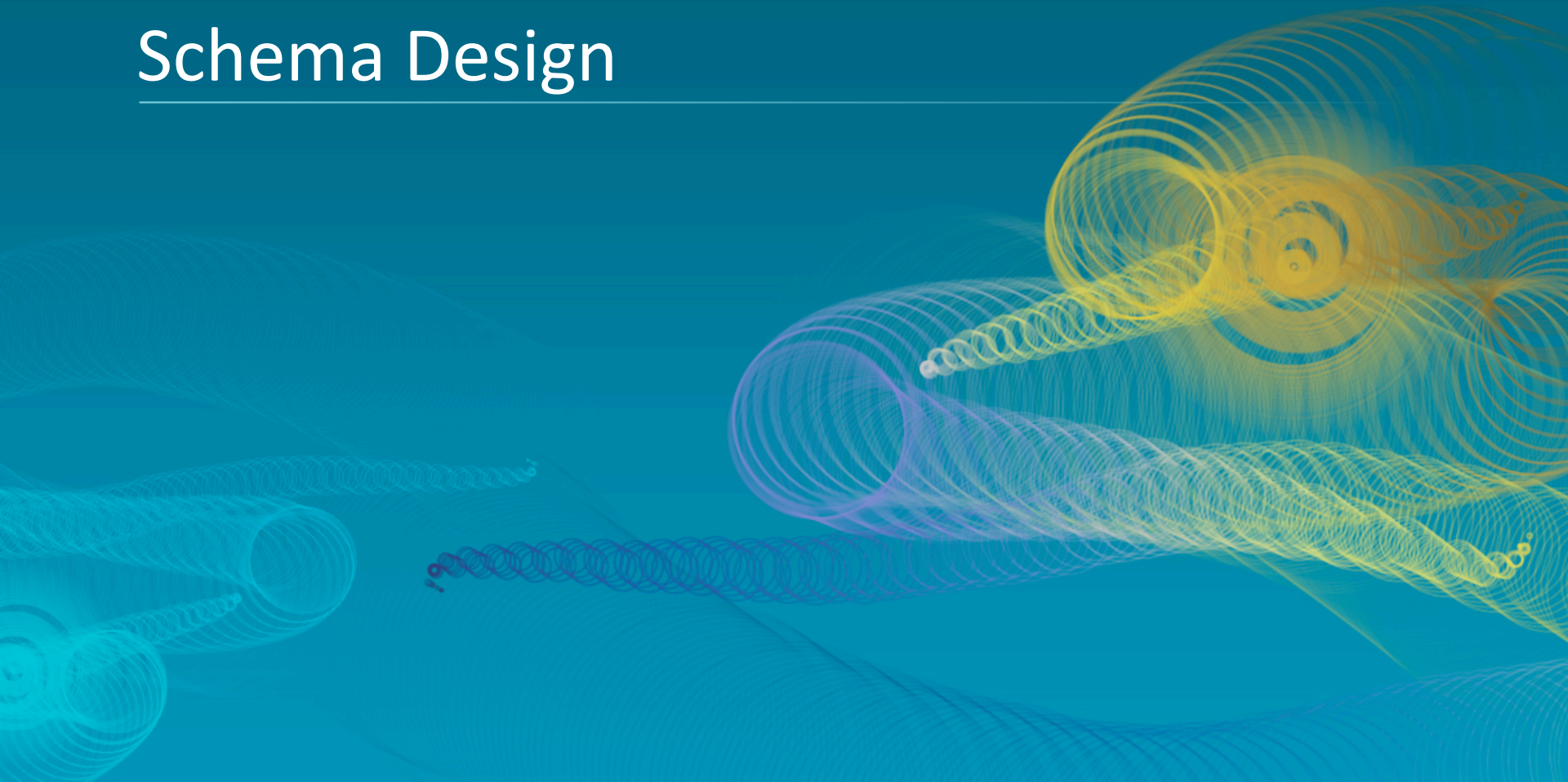
# Column Family vs. Column

---

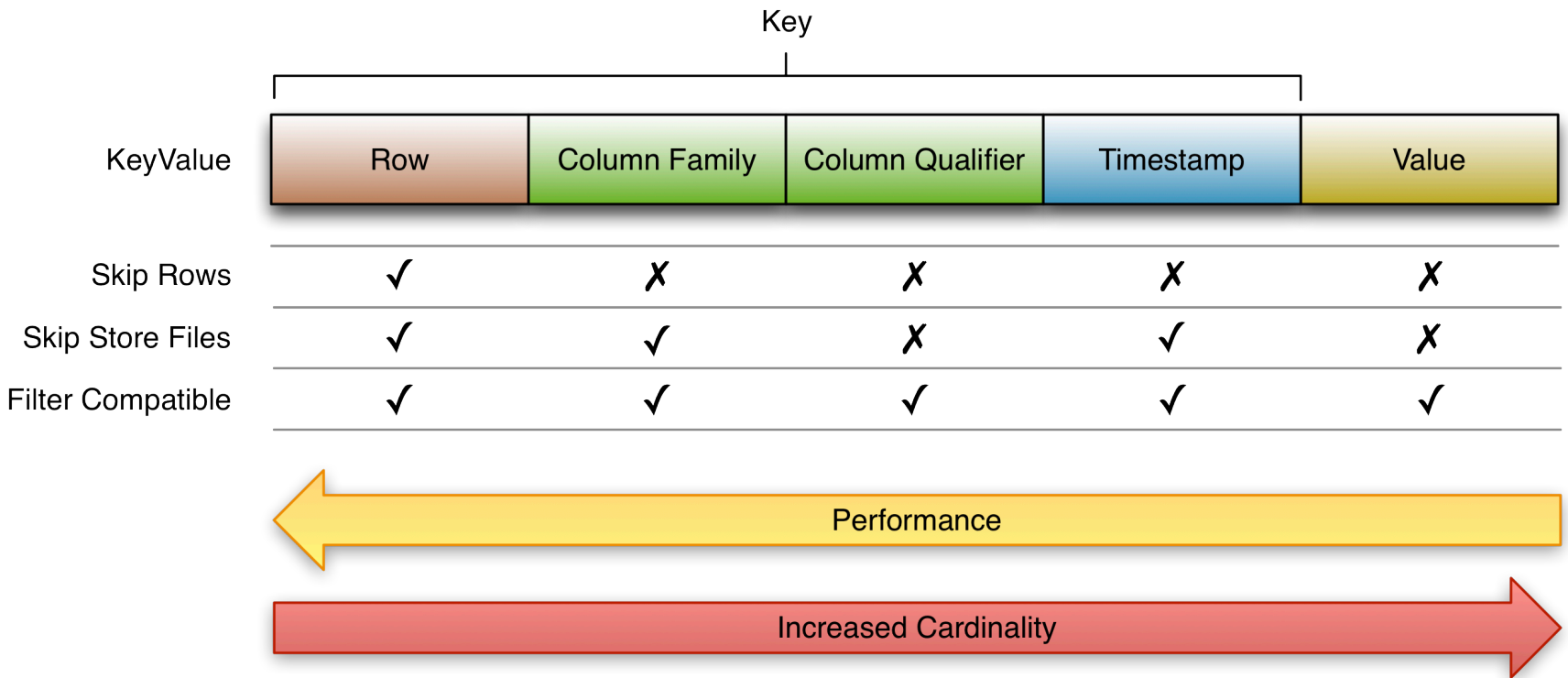
- Use only a few column families
  - Causes many files that need to stay open per region plus class overhead per family
  - Might trigger “compaction storms”
- Best used when logical separation between data and meta columns
- Sorting per family can be used to convey application logic or access pattern

# Schema Design

---



# Key Cardinality

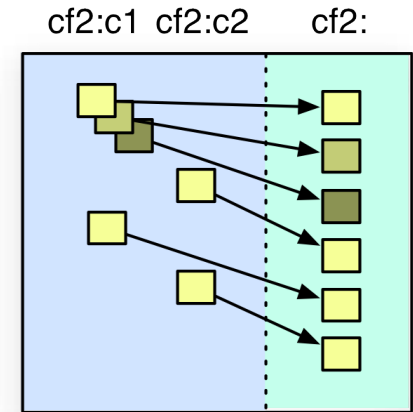
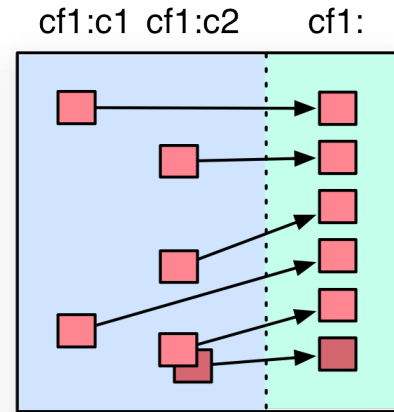
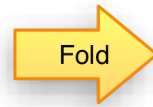
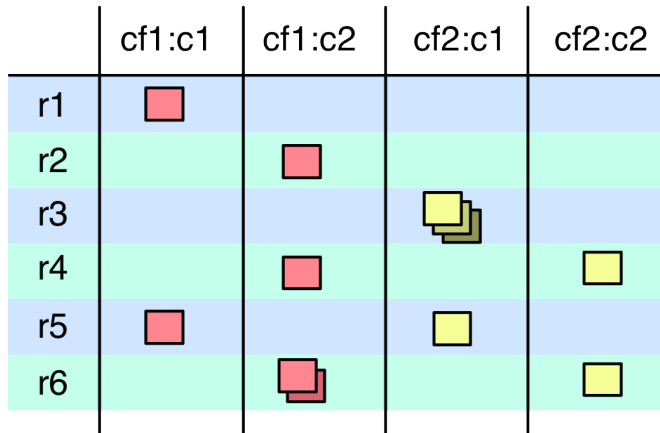


# Key Cardinality

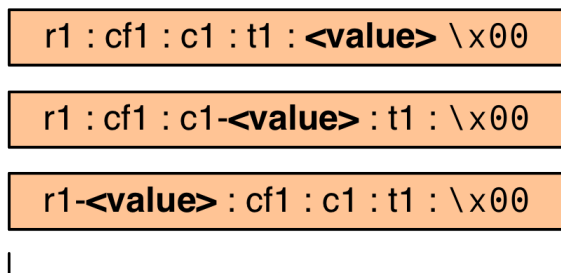
---

- 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

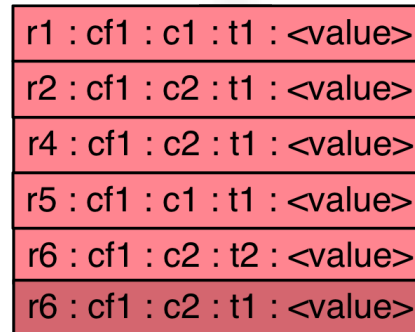
# Fold, Store, and Shift



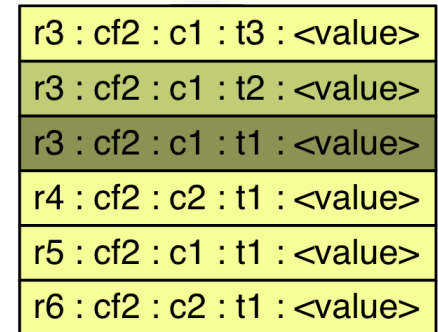
Store



= Same Storage Requirements



StoreFile "cf1/1234"



StoreFile "cf2/5678"

# 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>&lt;userId&gt;</code>	Scan over all messages for a given user ID
<code>&lt;userId&gt;-&lt;date&gt;</code>	Scan over all messages on a given date for the given user ID
<code>&lt;userId&gt;-&lt;date&gt;-&lt;messageId&gt;</code>	Scan over all parts of a message for a given user ID and date
<code>&lt;userId&gt;-&lt;date&gt;-&lt;messageId&gt;-&lt;attachmentId&gt;</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  
...
```

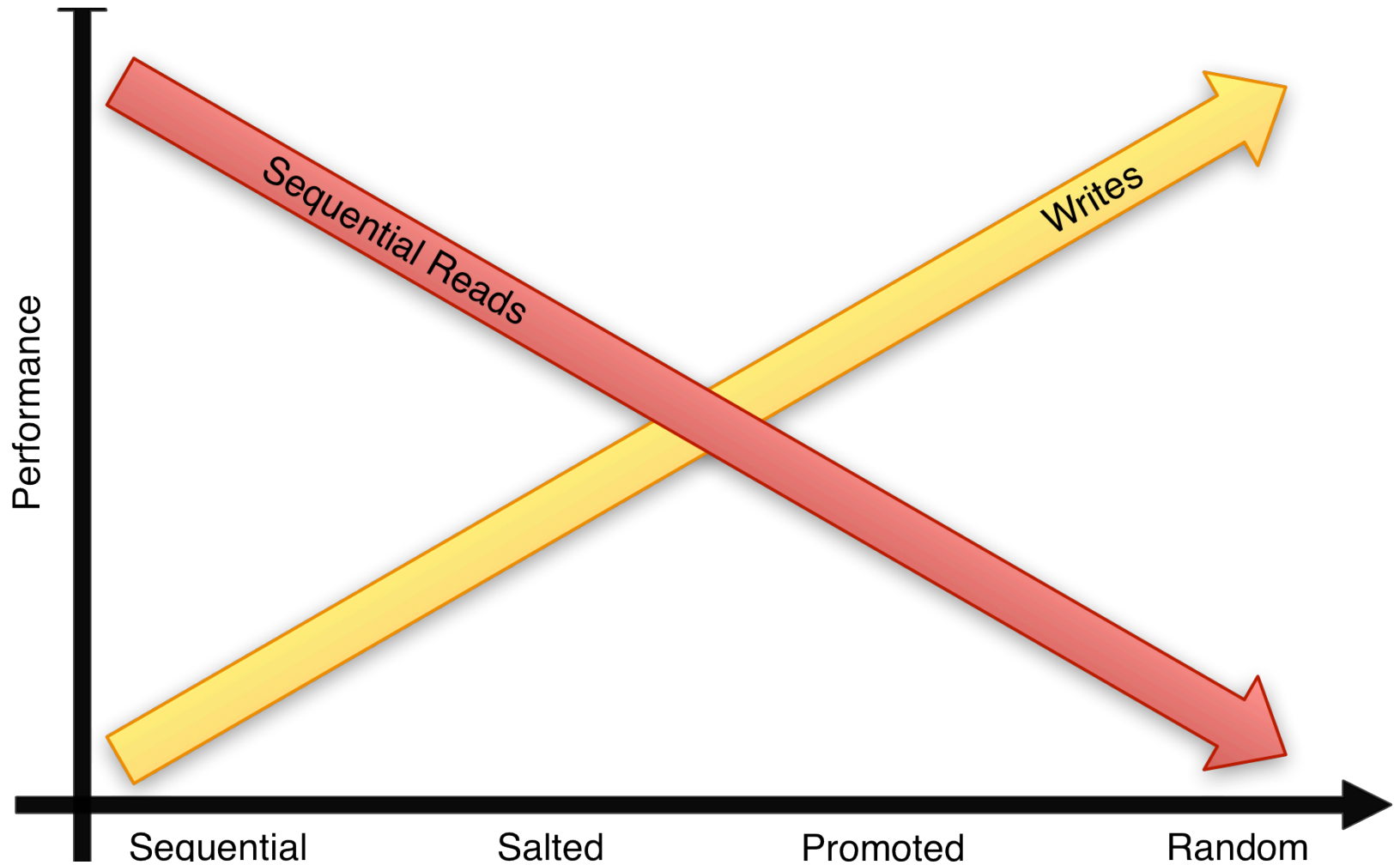
# Hashing vs. Sequential Keys

---

- 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

---

# 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)



**cloudera**<sup>®</sup>  
Ask Bigger Questions

**cloudera**<sup>®</sup>  
Ask Bigger Questions