Advanced HBase Schema Design

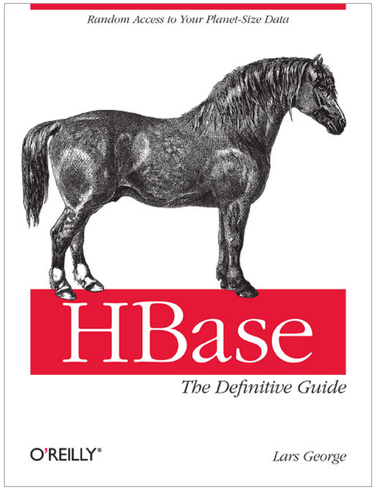
Berlin Buzzwords, June 2012

Lars George

lars@cloudera.com

About Me

- Solutions Architect @ Cloudera
- Apache HBase & Whirr Committer
- Author of
  HBase – The Definitive Guide
- Working with HBase since end of 2007
- Organizer of the Munich OpenHUG



Agenda

1 Overview of HBase

2 Schema Design

3 Examples

4 Wrap up

HBase Tables

	A	B	C	D	E
1					
2					
3					
4					
5					
6					
7					

HBase Tables

	A	B	C	D	E
1					
2					
3	A3	B3	C3	D3	E3
4					
5					
6					
7					

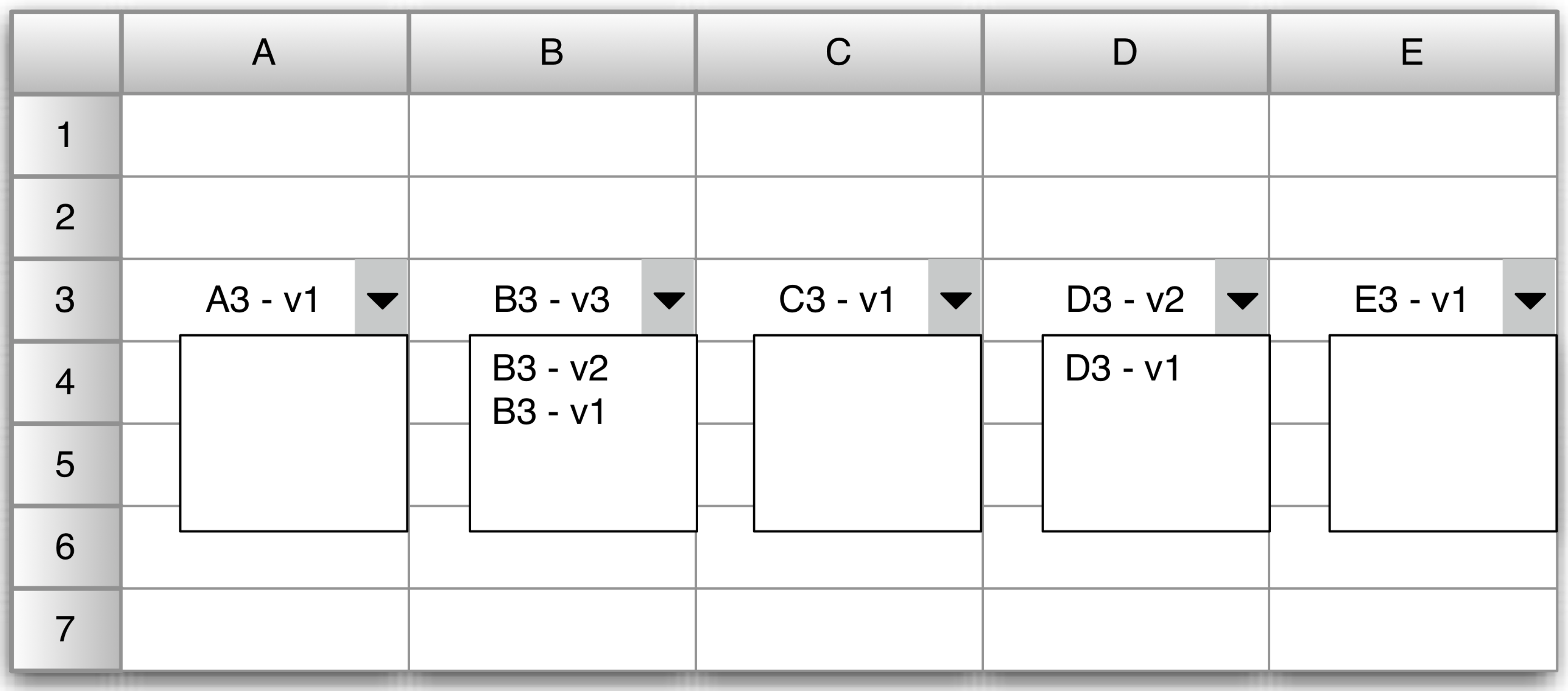
HBase Tables

	A	B	C	D	E
1					
2					
3	A3 - v1	B3 - v3	C3 - v1	D3 - v2	E3 - v1
4					
5					
6					
7					

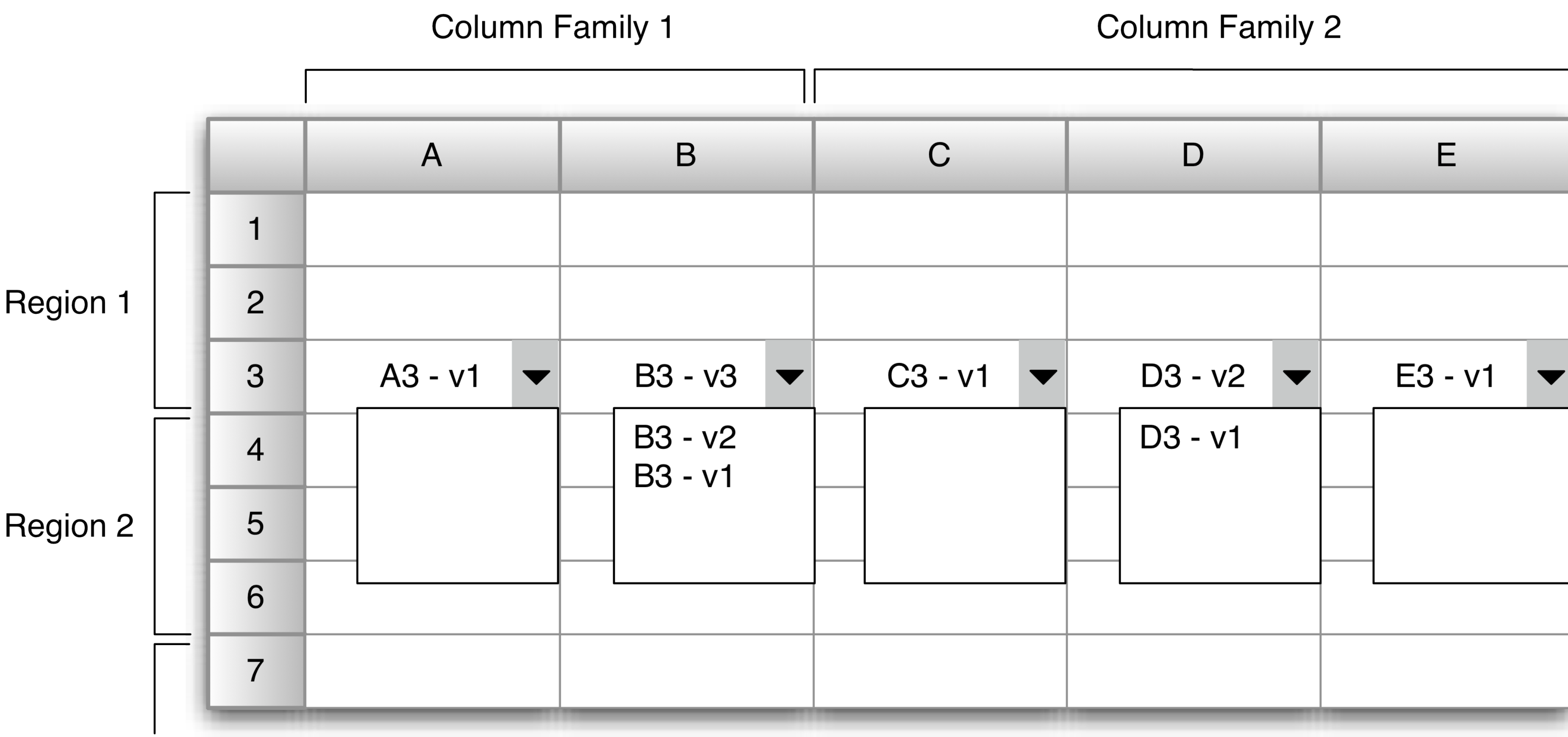
HBase Tables

	A	B	C	D	E
1					
2					
3	A3 - v1	B3 - v3	C3 - v1	D3 - v2	E3 - v1
4					
5					
6					
7					

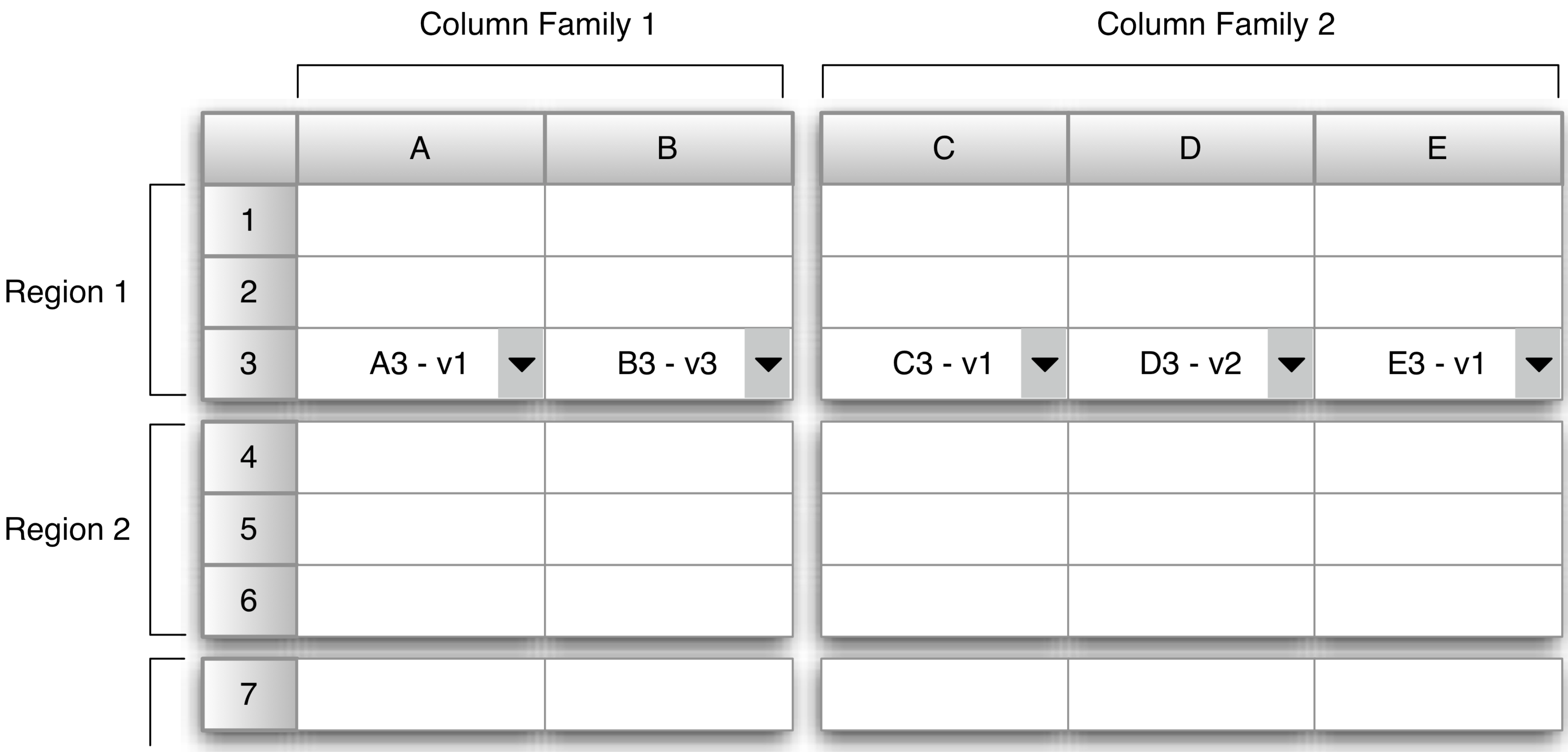
HBase Tables



HBase Tables



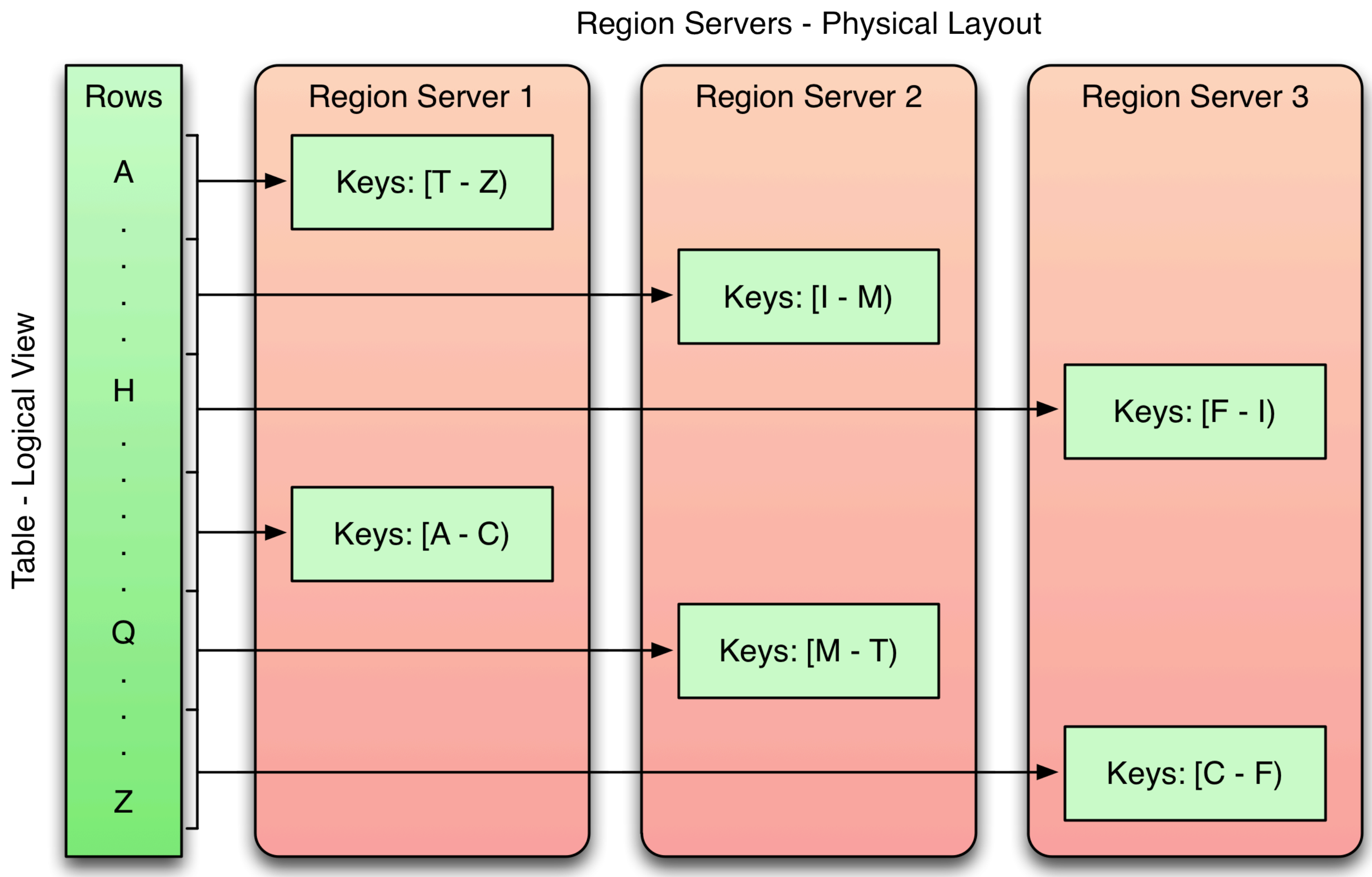
HBase Tables



Physical Model: HBase Tables and Regions

- Table is made up of any number if regions
- Region is specified by its startKey and endKey
- 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

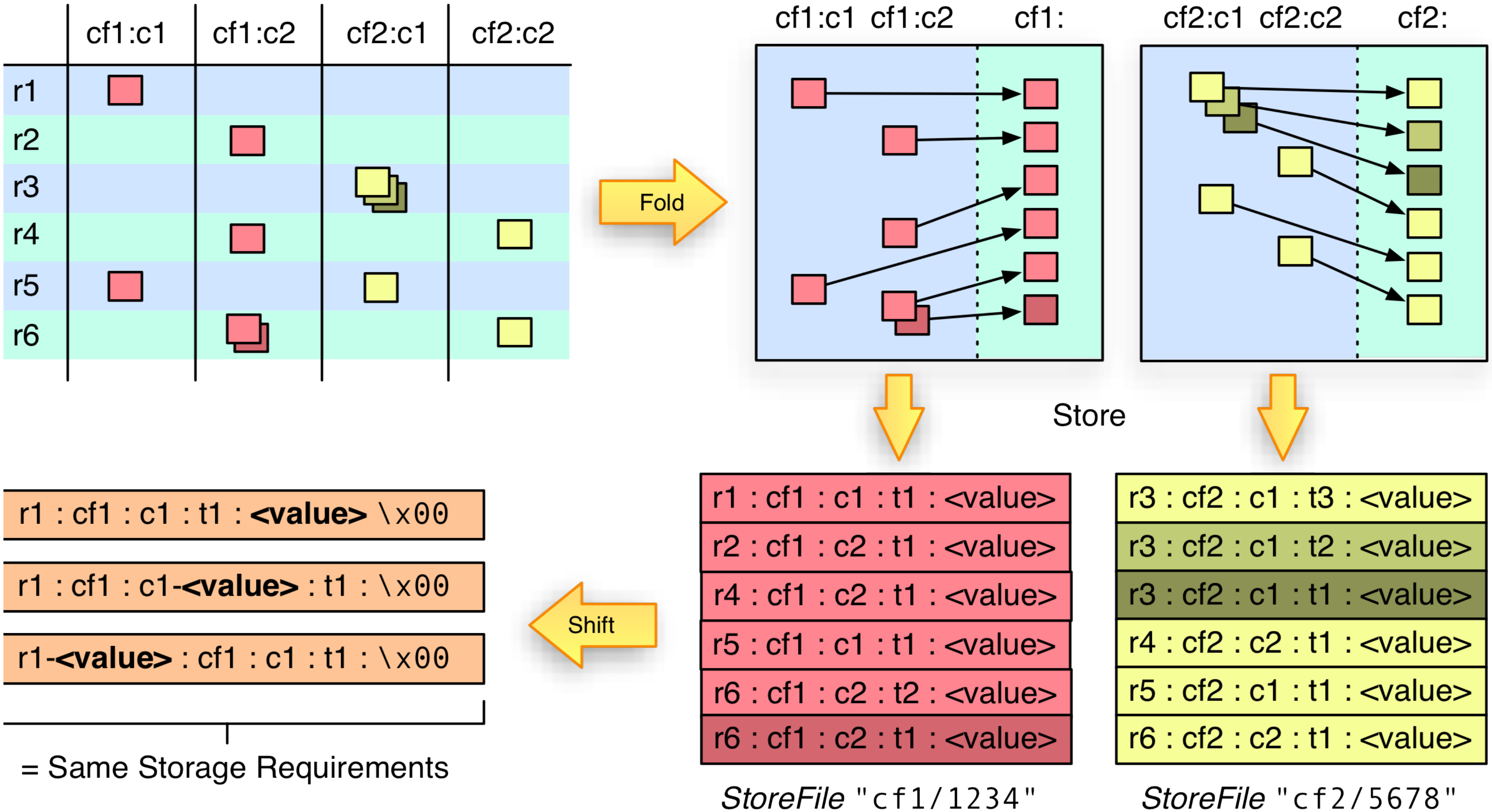
Distribution



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

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

Logical Model: 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[]

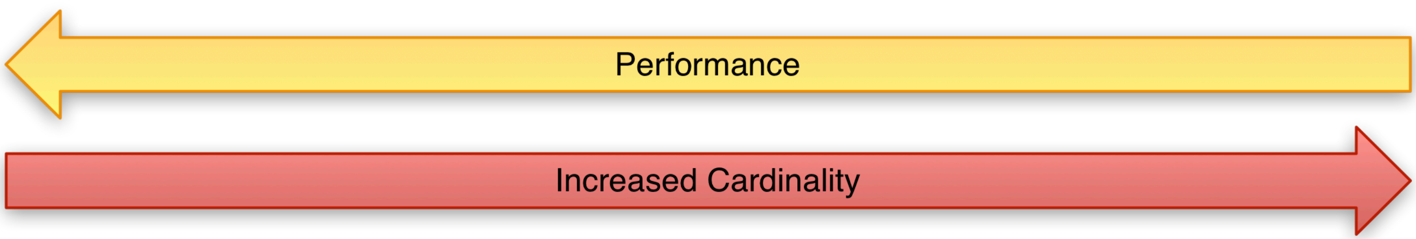
(Table, Row, Family:Column, Timestamp) -> Value

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

Key Cardinality

	Key				
KeyValue	Row	Column Family	Column Qualifier	Timestamp	Value
Skip Rows	✓	✗	✗	✗	✗
Skip Store Files	✓	✓	✗	✓	✗
Filter Compatible	✓	✓	✓	✓	✓



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

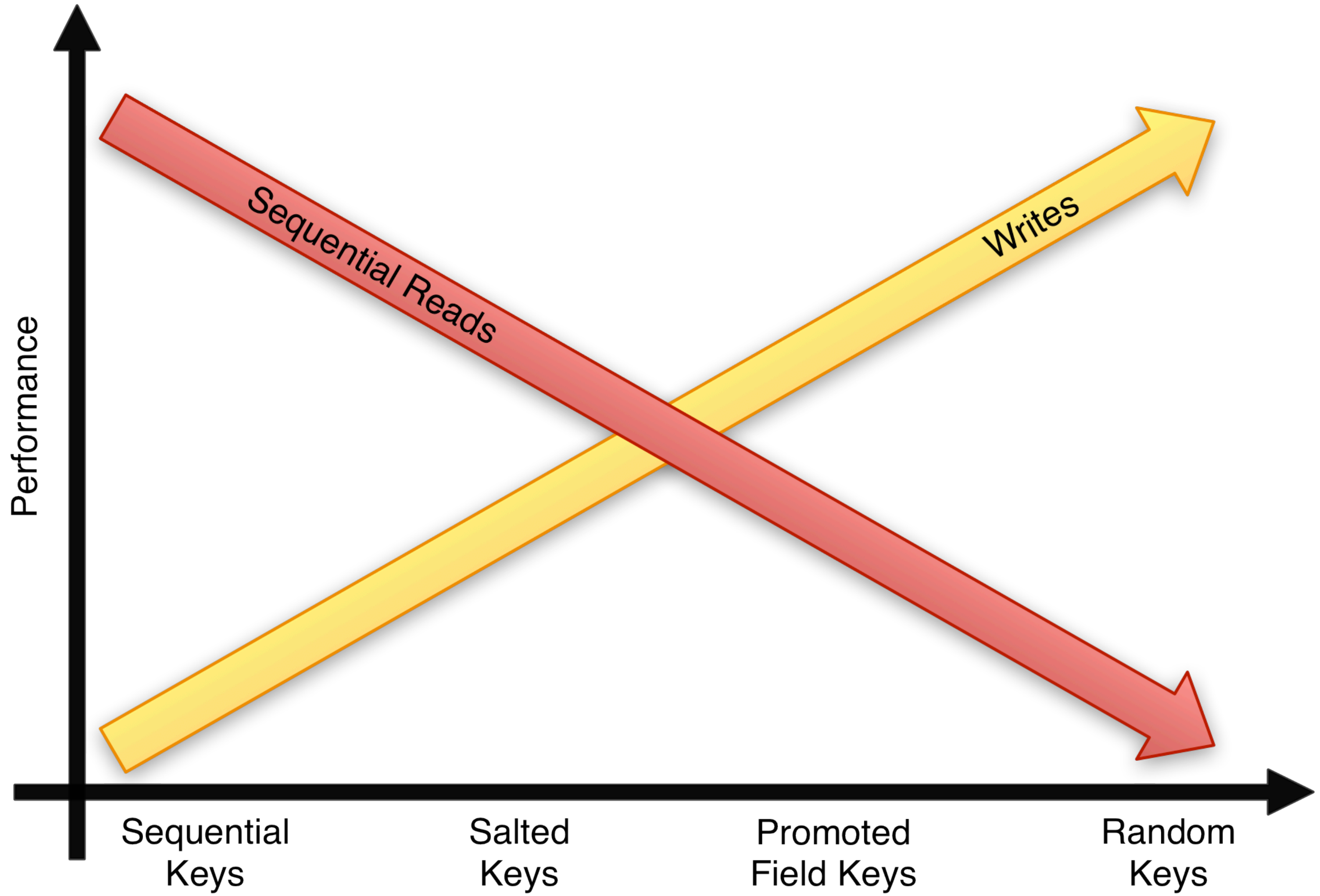
Key/Table Design

- 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

Key Design


Performance
Sequential Reads
Writes
Sequential Keys
Salted Keys
Promoted Field Keys
Random Keys

Key Design Summary

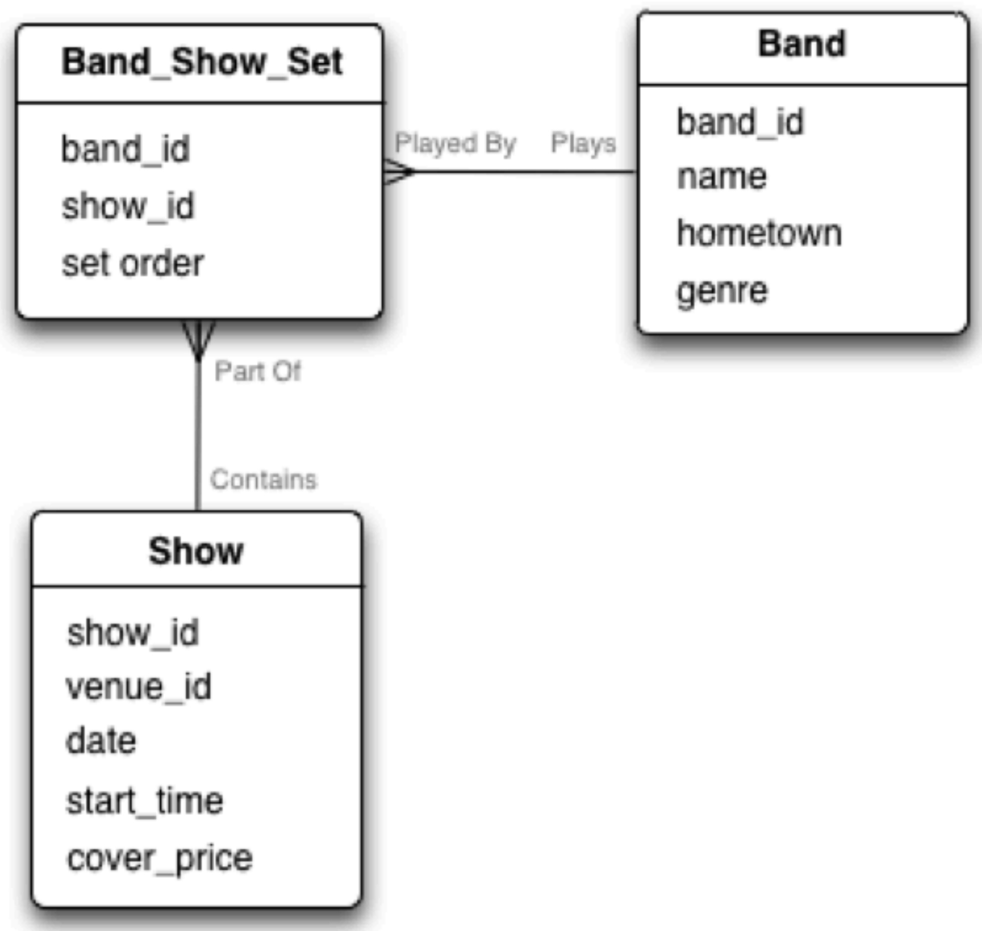
- Based on access pattern, either use sequential or random keys
- Often a combination of both is needed
  - Overcome architectural limitations
- Neither is necessarily bad
  - Use bulk import for sequential keys and reads
  - Random keys are good for random access patterns

Pre-materialize Everything

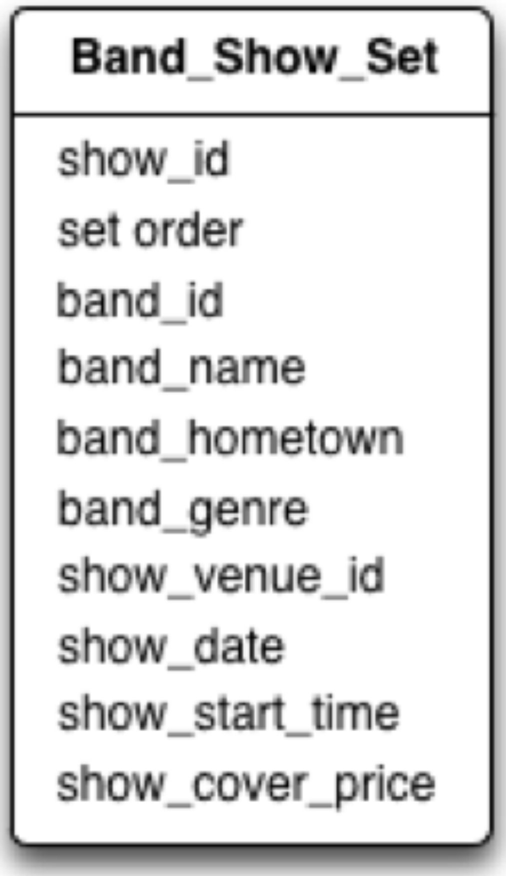
- Achieve one read per customer request if possible
- Otherwise keep at lowest number
- Reads between 10ms (cache miss) and 1ms (cache hit)
- Use MapReduce to compute exacts in batch
- Store and merge updates live
- Use incrementColumnValue

Motto: “Design for **Reads**”

Relational Model



Muddled Up!


Band_Show_Set
show_id
set order
band_id
band_name
band_hometown
band_genre
show_venue_id
show_date
show_start_time
show_cover_price

Remodeling

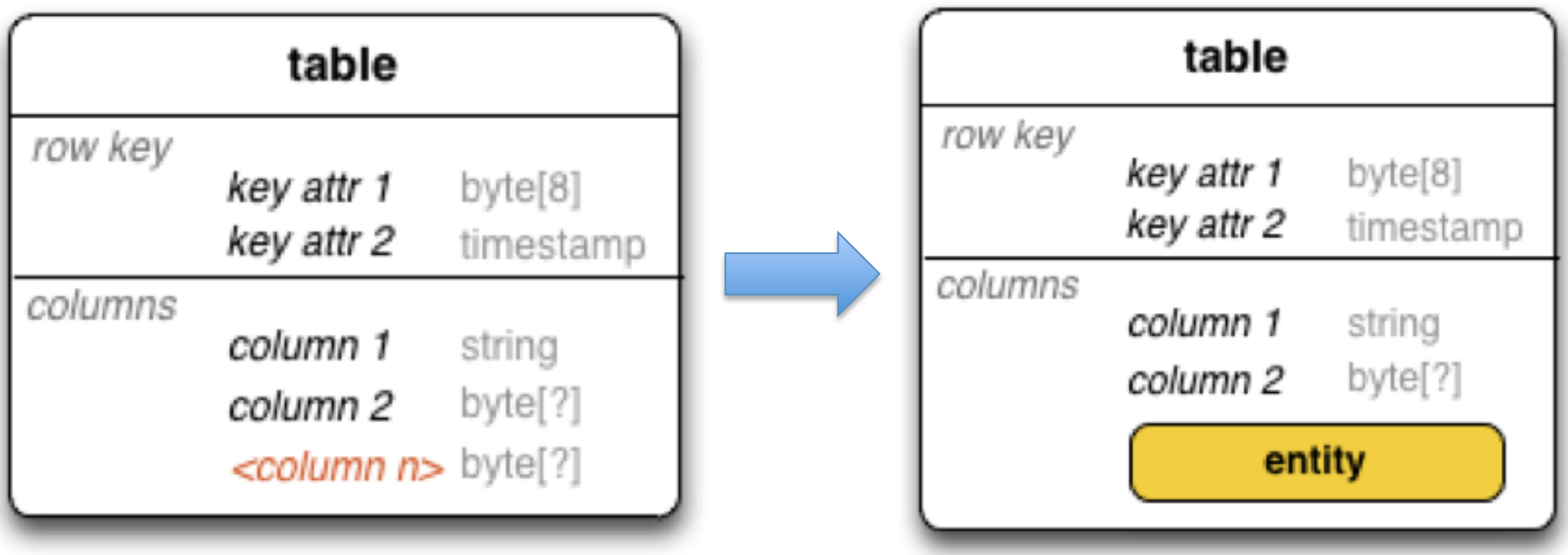


table
row key
key attr 1
byte[8]
key attr 2
timestamp
columns
column 1
string
column 2
byte[?]
<column n>
byte[?]
table
row key
key attr 1
byte[8]
key attr 2
timestamp
columns
column 1
string
column 2
byte[?]
entity

WTH?

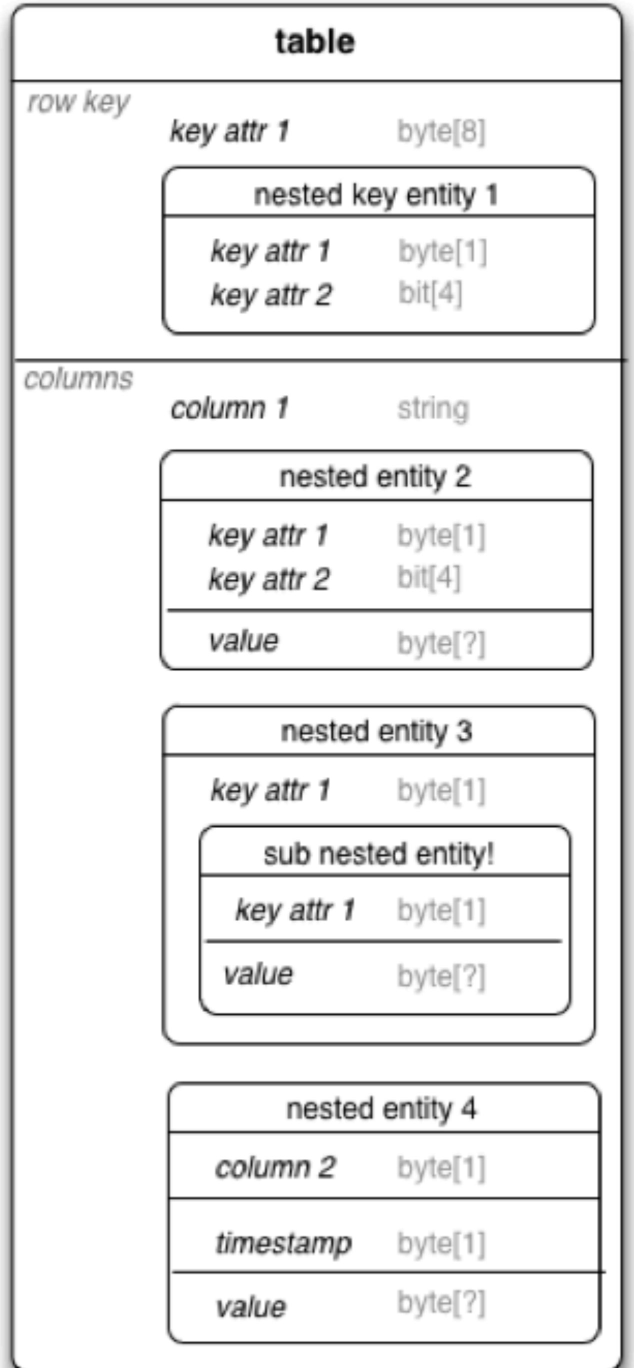
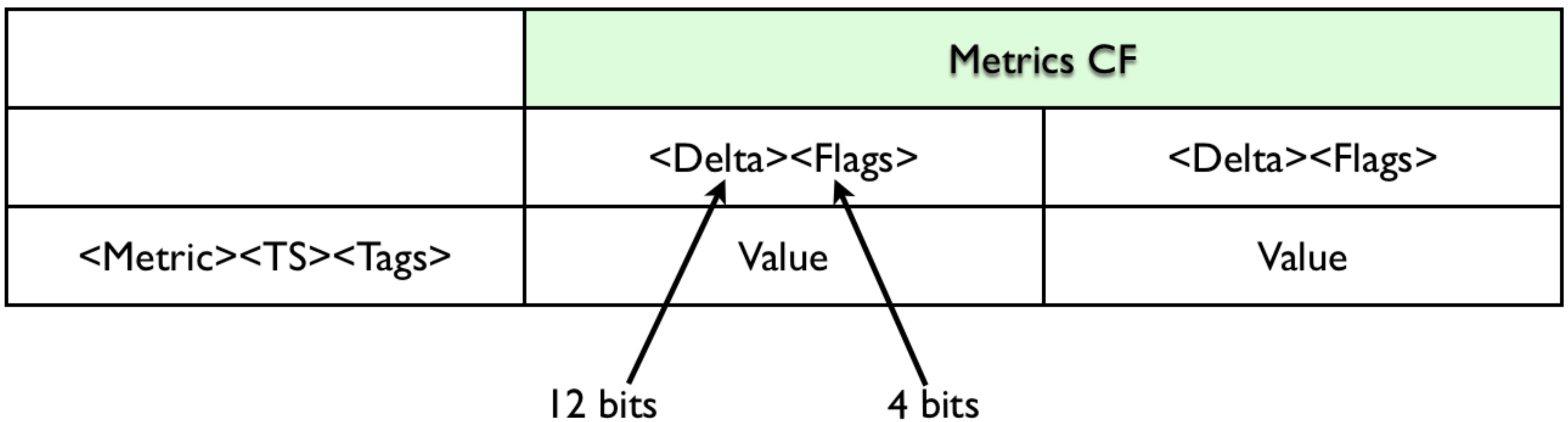


table
row key
key attr 1
byte[8]
nested key entity 1
key attr 1
byte[1]
key attr 2
bit[4]
columns
column 1
string
nested entity 2
key attr 1
byte[1]
key attr 2
bit[4]
value
byte[?]
nested entity 3
key attr 1
byte[1]
sub nested entity!
key attr 1
byte[1]
value
byte[?]
nested entity 4
column 2
byte[1]
timestamp
byte[1]
value
byte[?]

Example: OpenTSDB



- Metric Type, Tags are stored as IDs
- Periodically rolled up

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)


Questions?
Email: lars@cloudera.com
Twitter: @larsgeorge

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
`<userId>`	Scan over all messages for a given user ID
`<userId>-<date>`	Scan over all messages on a given date for the given user ID
`<userId>-<date>-<messageId>`	Scan over all parts of a message for a given user ID and date
`<userId>-<date>-<messageId>-<attachmentId>`	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
```

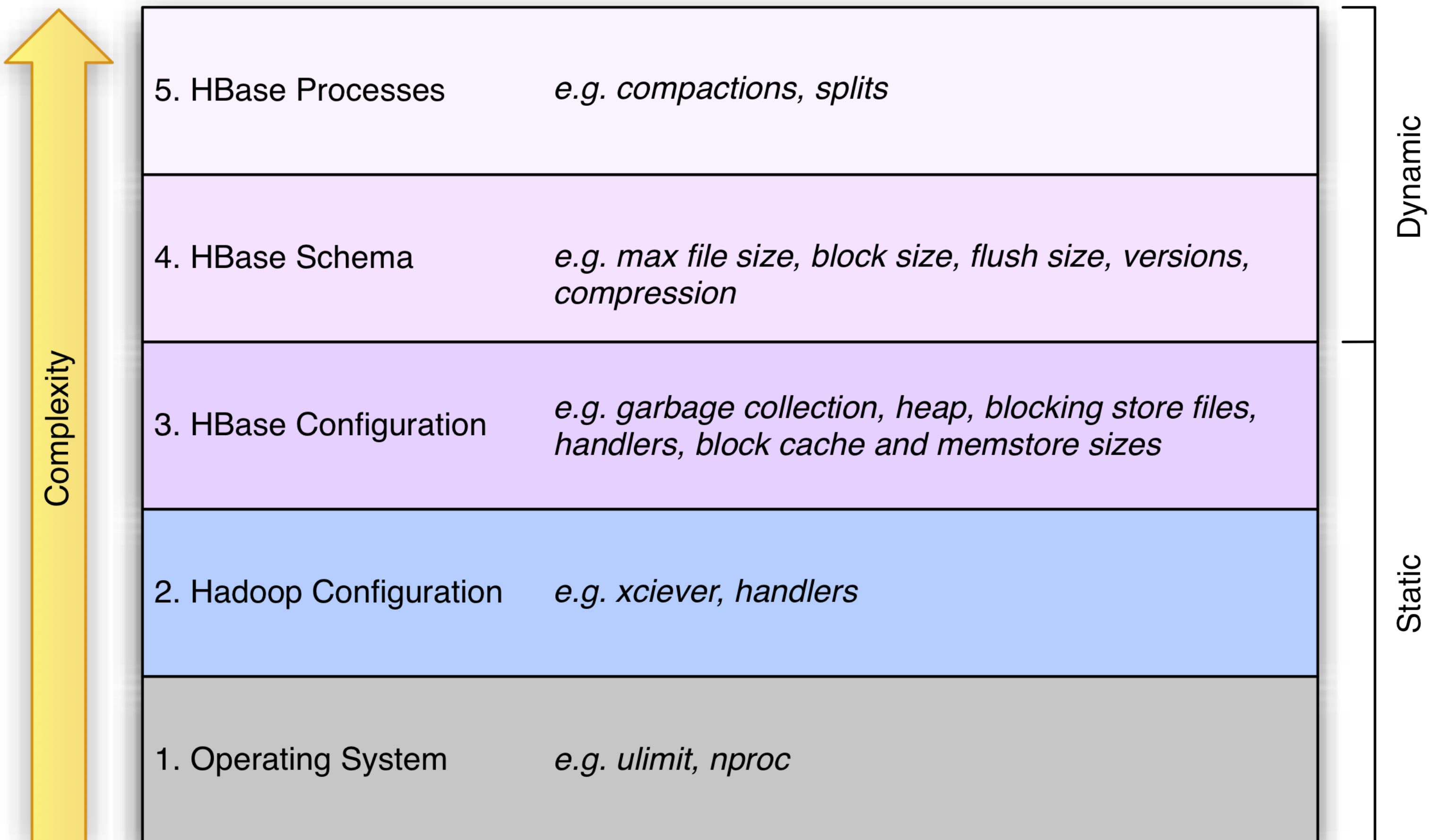
- Sorted by prefix first

```
0_rowkey1
0_rowkey4
1_rowkey2
1_rowkey5
…
```

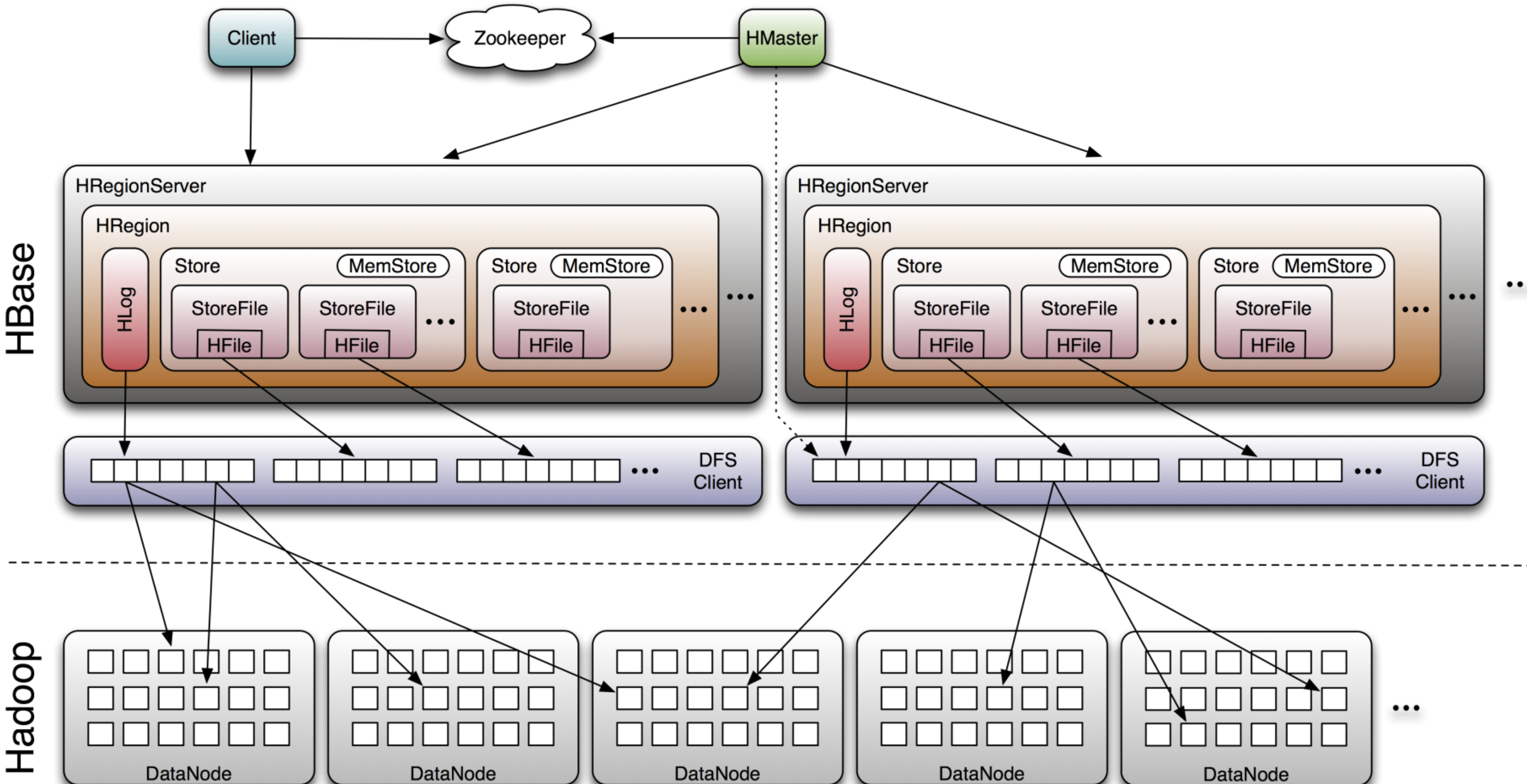
Hashing vs. Sequential Keys

- 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

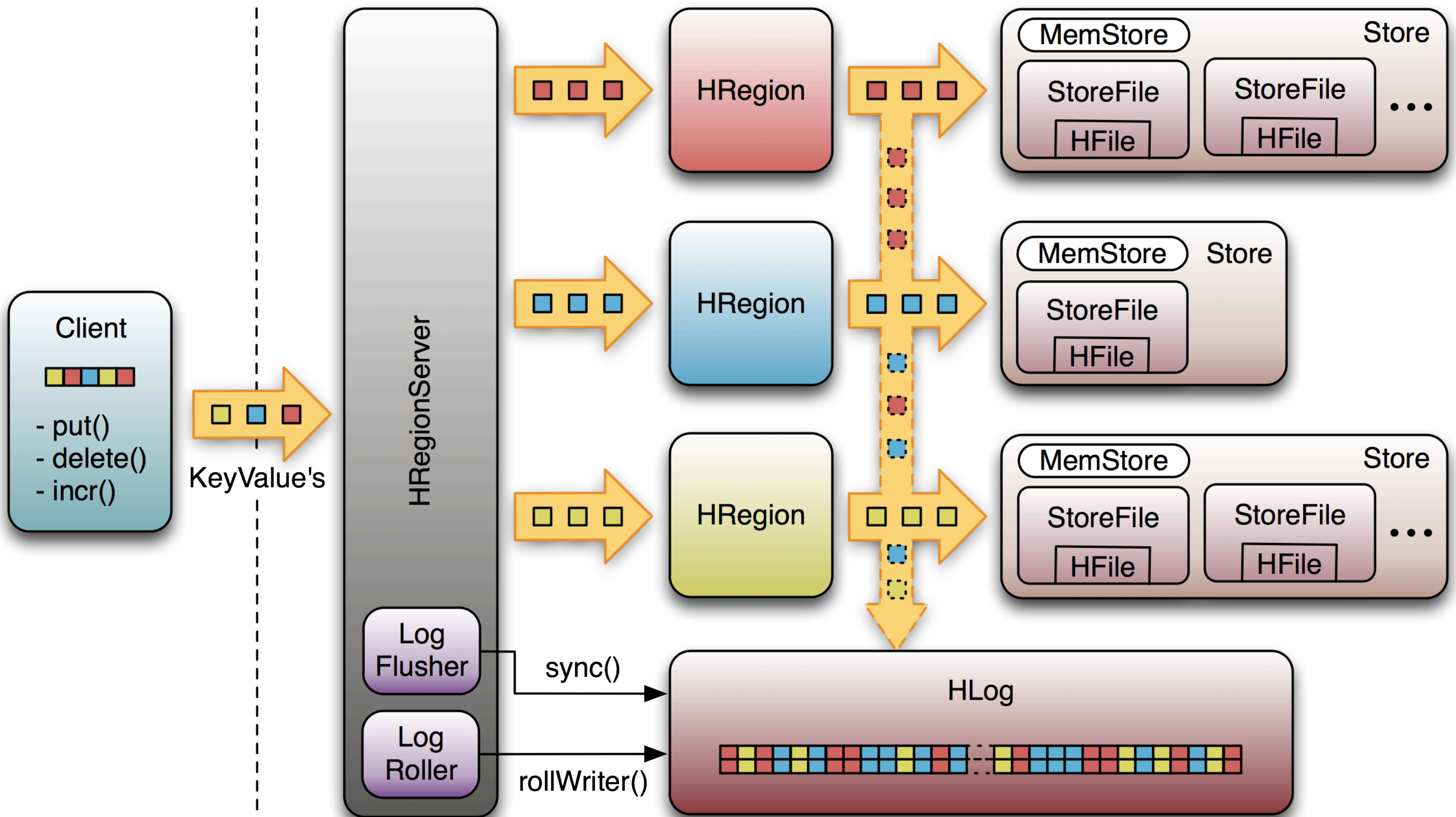
Configuration Layers (aka “OSI for HBase”)



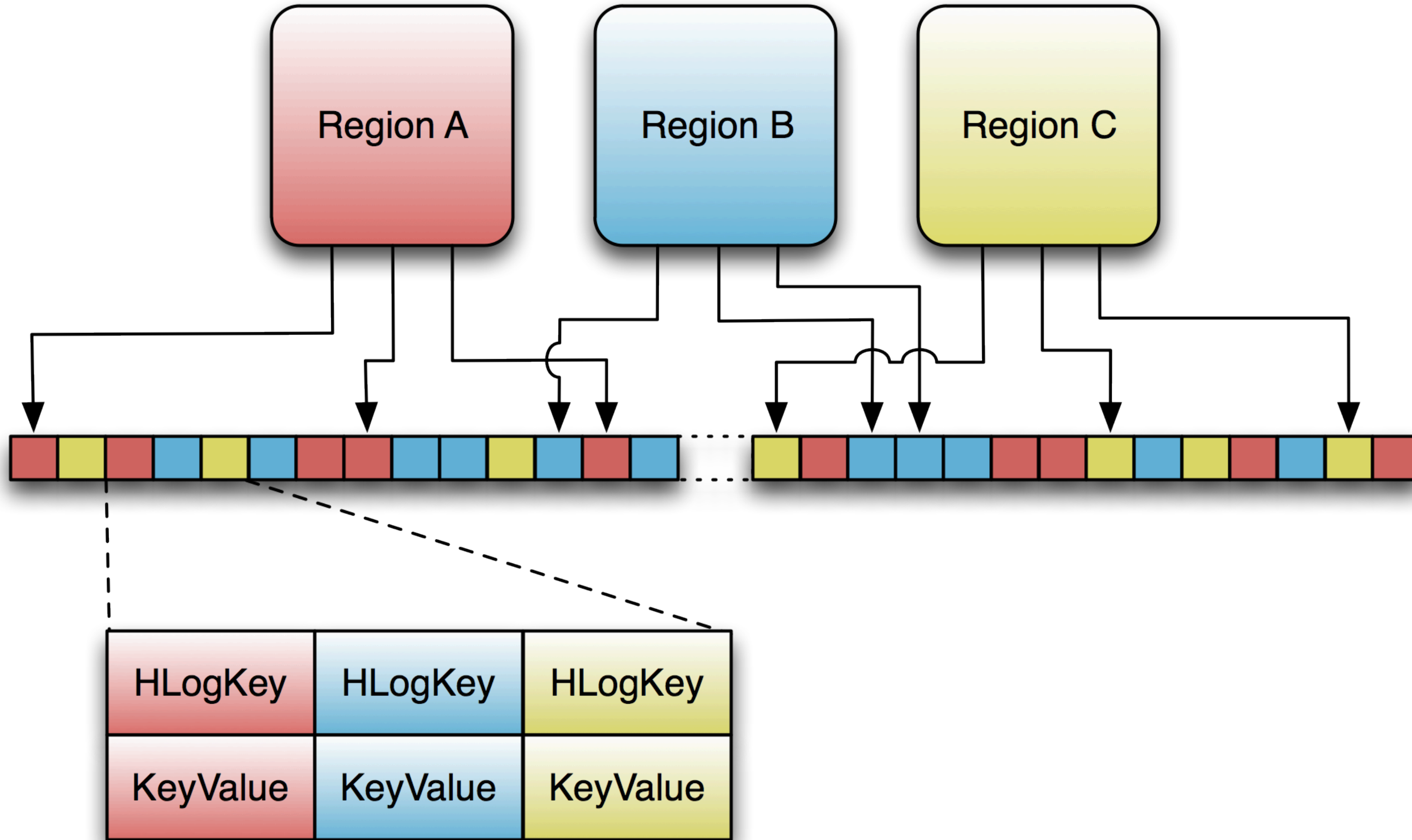
HBase Architecture (cont.)



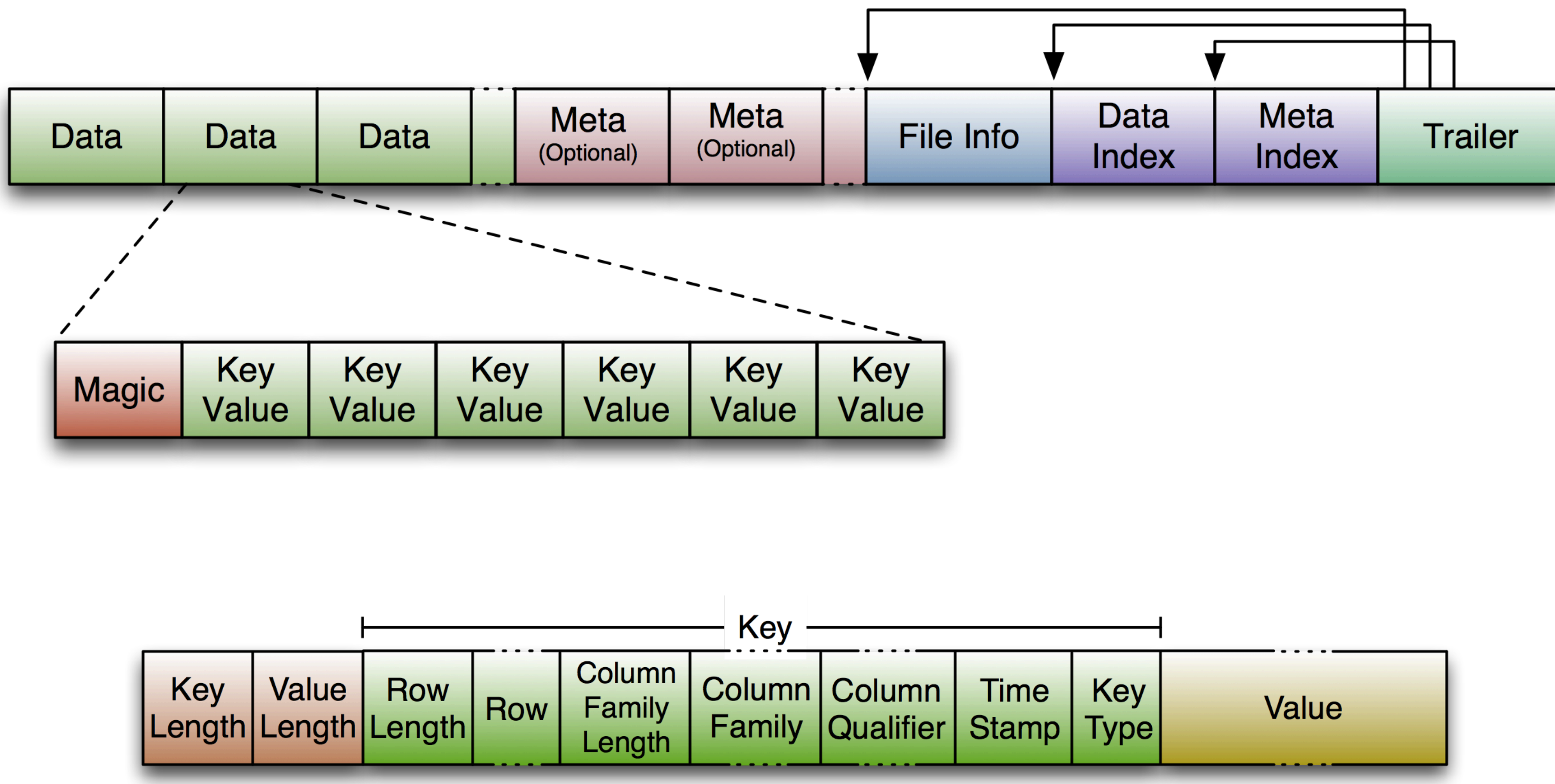
Write-Ahead-Log (WAL) Flow



Write-Ahead-Log (cont.)



HFile and KeyValue



Raw Data View

```
$ ./bin/hbase org.apache.hadoop.hbase.io.hfile.HFile -f file:///tmp/
   hbase-larsgeorge/hbase/testtable/272a63b23bdb5fae759be5192cabc0ce/
   f1/4992515006010131591 -p

K: row1/f1:/1290345071149/Put/vlen=6 V: value1
K: row2/f1:/1290345078351/Put/vlen=6 V: value2
K: row3/f1:/1290345089750/Put/vlen=6 V: value3
K: row4/f1:/1290345095724/Put/vlen=6 V: value4
K: row5/f1:c1/1290347447541/Put/vlen=6 V: value5
K: row6/f1:c2/1290347461068/Put/vlen=6 V: value6
K: row7/f1:c1/1290347581879/Put/vlen=7 V: value10
K: row7/f1:c1/1290347469553/Put/vlen=6 V: value7
K: row7/f1:c10/1290348157074/DeleteColumn/vlen=0 V:
K: row7/f1:c10/1290347625771/Put/vlen=7 V: value11
K: row7/f1:c11/1290347971849/Put/vlen=7 V: value14
K: row7/f1:c12/1290347979559/Put/vlen=7 V: value15
K: row7/f1:c13/1290347986384/Put/vlen=7 V: value16
K: row7/f1:c2/1290347569785/Put/vlen=6 V: value8
K: row7/f1:c3/1290347575521/Put/vlen=6 V: value9
K: row7/f1:c8/1290347638008/Put/vlen=7 V: value13
K: row7/f1:c9/1290347632777/Put/vlen=7 V: value12
```