# HBase介绍

英特尔软件部
邓　刚

# 议程

- What is HBase
- HBase Architecture
- HBase Basics

# 为什么需要NoSQL

传统关系型数据库的优点：

- 严格的一致性模型
- 易于管理和维护

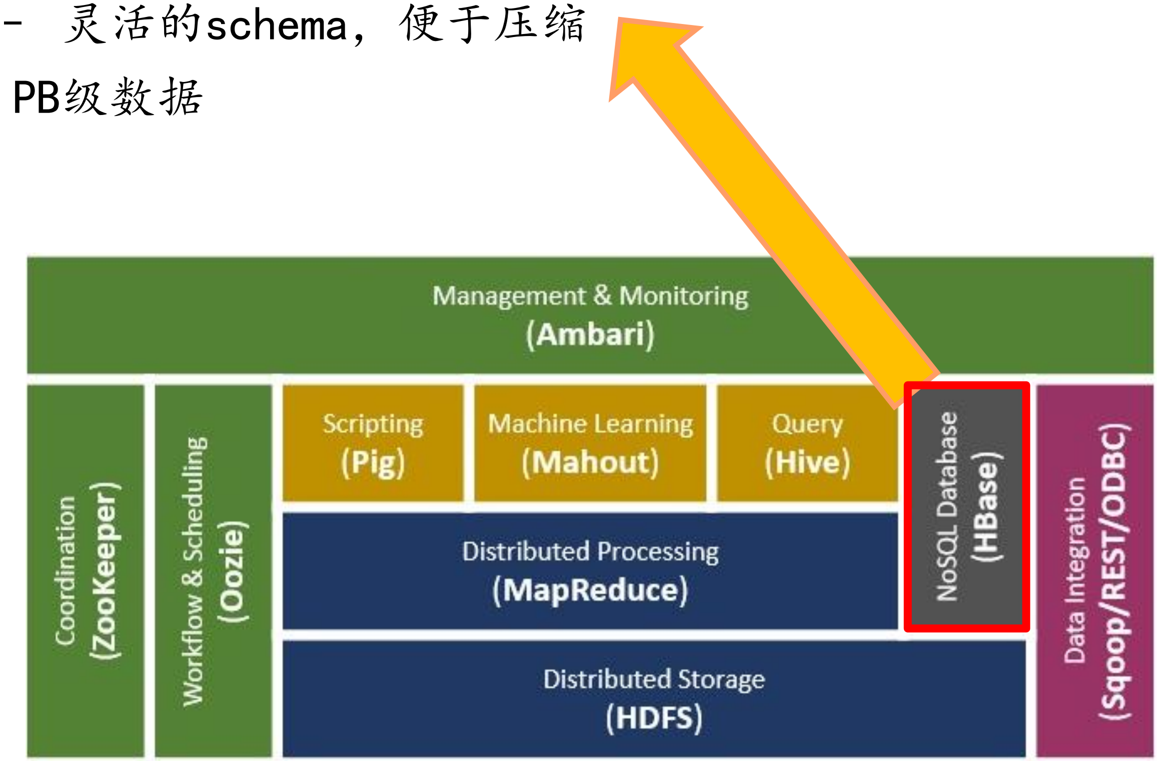
。。。

传统关系型数据库的挑战：

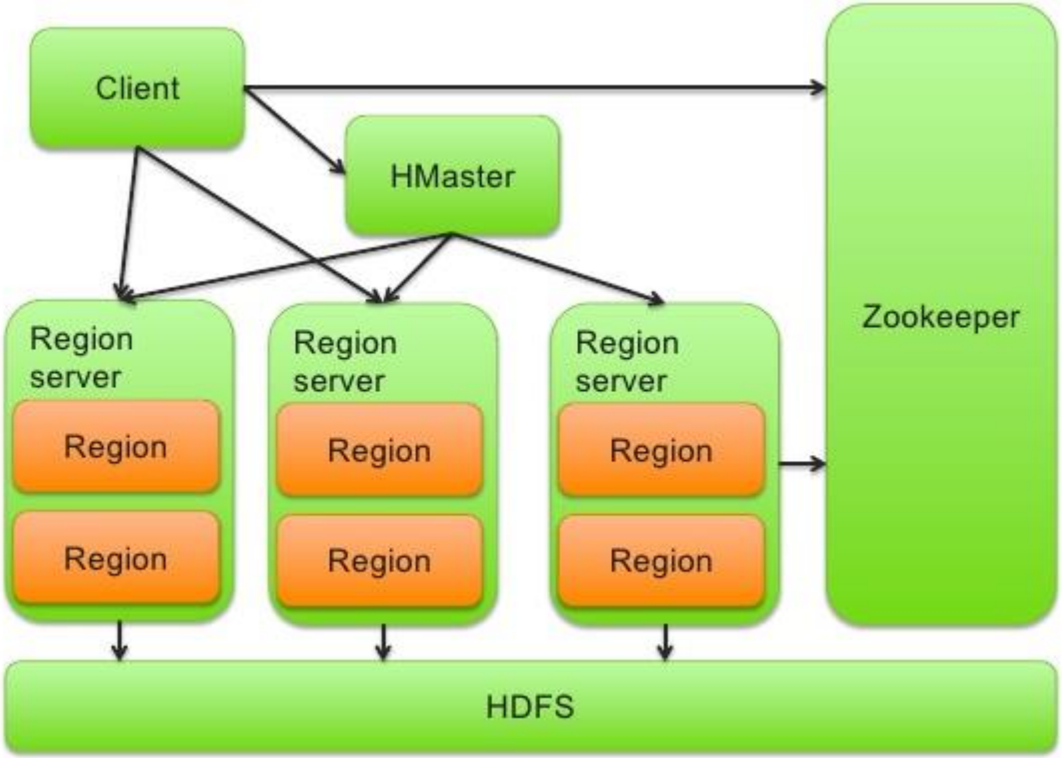
- 扩展瓶颈 – scale up vs. scale out
- 严苛的Schema

# What is HBase

- KV 模型
- 分布式 －可水平扩展到成百上千个节点
- 列式存储 － 灵活的schema，便于压缩
- 大容量 － PB级数据
- 高性能



# HBase Architecture



# Hbase Architecture (cont.)

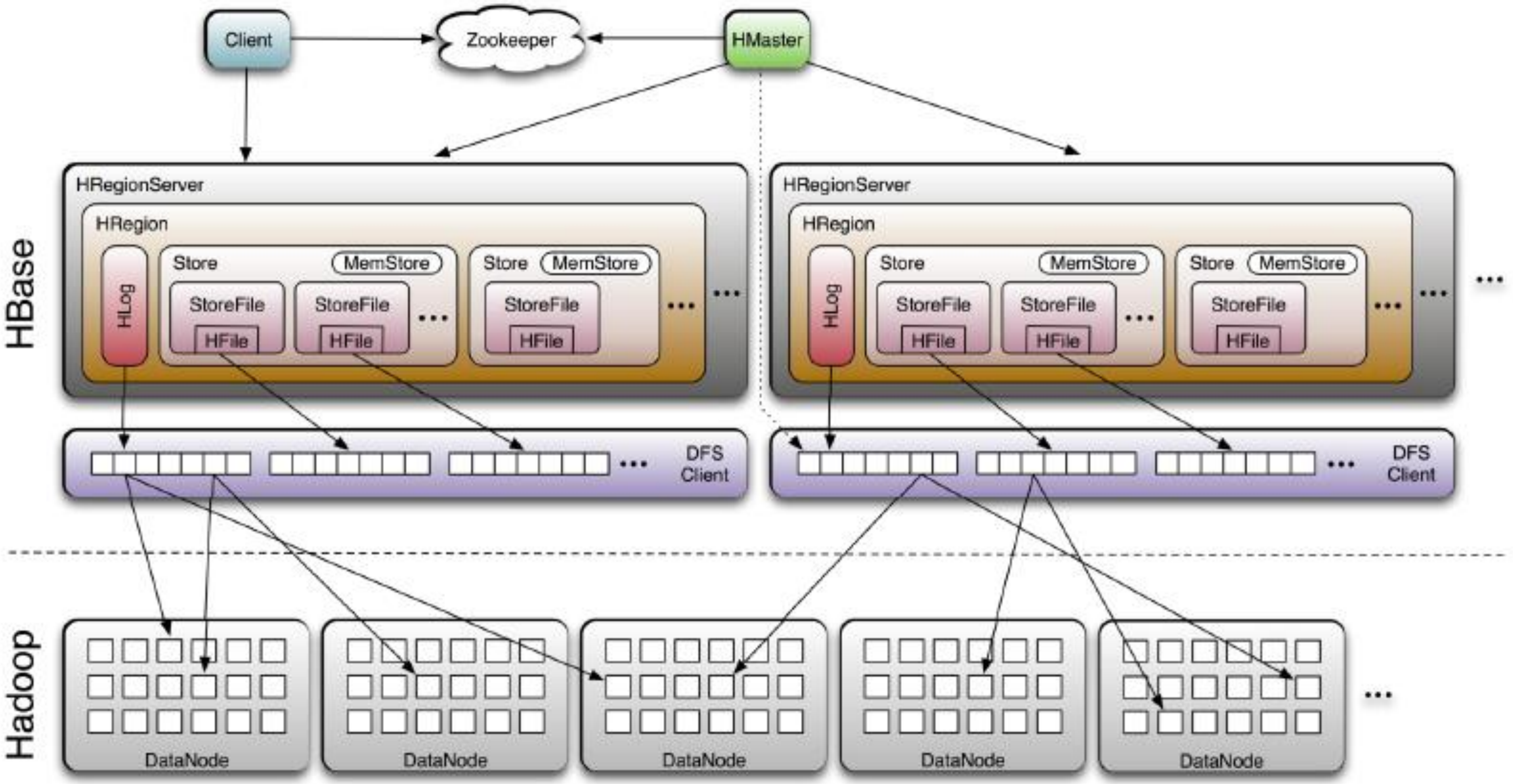
- Table is made up of any number if regions
- Region is specified by its startKey and endKey
  - Empty table: (Table, NULL, NULL)
  - Two-region table: (Table, NULL, “com.cloudera.www”) and (Table, “com.cloudera.www”, NULL)
- Each region may live on a different node and is made up of several HDFS files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

# Hbase Architecture (cont.)

- Two types of HBase nodes:

  **Master** and **RegionServer**
- Special tables -ROOT- and .META. store schema information and region locations
- Master server responsible for RegionServer monitoring as well as assignment and load balancing of regions
- Uses ZooKeeper as its distributed coordination service
  - Manages Master election and server availability

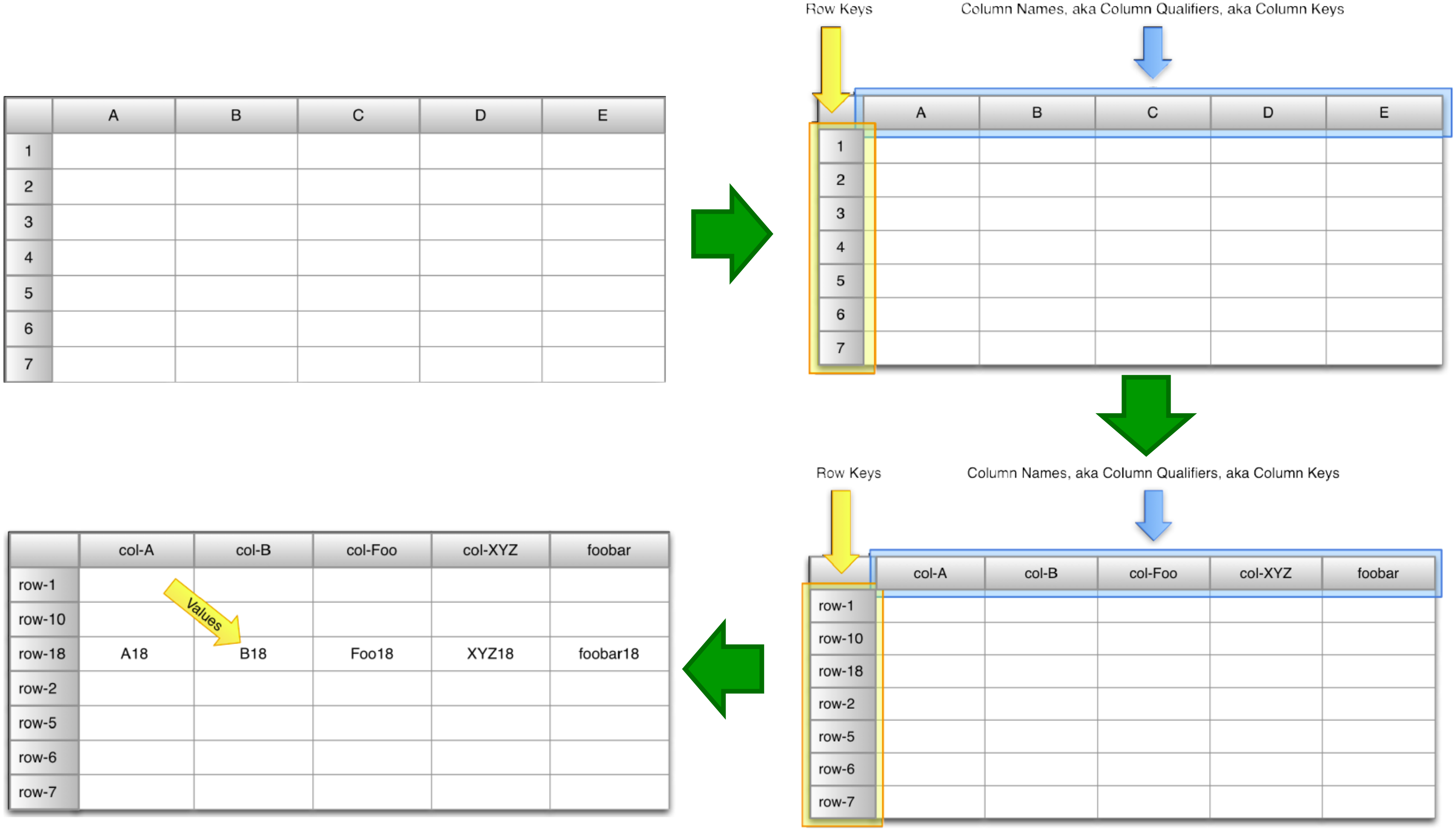
# Hbase Architecture (cont.)



# 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

# HBase Table



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

Row Keys

Column Names, aka Column Qualifiers, aka Column Keys

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

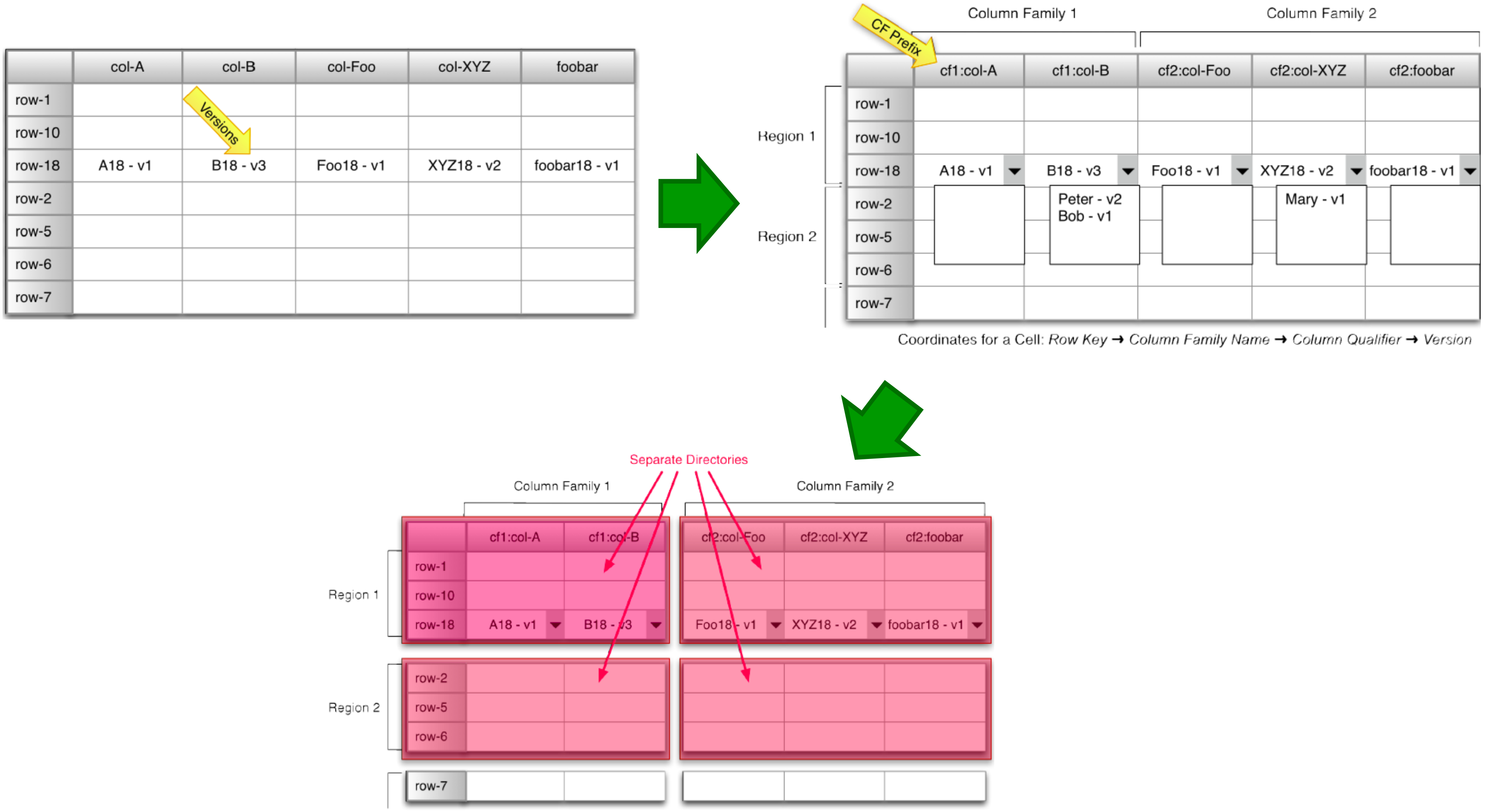
Row Keys

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 | | | | | |

| | col-A | col-B | col-Foo | col-XYZ | foobar |
|---|---|---|---|---|---|
| row-1 | | | | | |
| row-10 | | | | | |
| row-18 | A18 | B18 | Foo18 | XYZ18 | foobar18 |
| row-2 | | | | | |
| row-5 | | | | | |
| row-6 | | | | | |
| row-7 | | | | | |

# HBase Table (cont.)



| | 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 | | | | | |
| row-5 | | | | | |
| row-6 | | | | | |
| row-7 | | | | | |

| | Column Family 1 | | Column Family 2 | | |
|---|---|---|---|---|---|
| | cf1:col-A | cf1:col-B | cf2:col-Foo | cf2:col-XYZ | cf2:foobar |
| row-1 | | | | | |
| row-10 | | | | | |
| row-18 | A18 - v1 | B18 - v3 | Foo18 - v1 | XYZ18 - v2 | foobar18 - v1 |
| row-2 | | Peter - v2<br>Bob - v1 | | Mary - v1 | |
| row-5 | | | | | |
| row-6 | | | | | |
| row-7 | | | | | |

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

| | Column Family 1 | | Column Family 2 | | |
|---|---|---|---|---|---|
| | cf1:col-A | cf1:col-B | cf2:col-Foo | cf2:col-XYZ | cf2:foobar |
| row-1 | | | | | |
| row-10 | | | | | |
| row-18 | A18 - v1 | B18 - v3 | Foo18 - v1 | XYZ18 - v2 | foobar18 - v1 |
| 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 Table (cont.)

- Tables are sorted by the *Row Key* 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

# 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

# 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

# Block Cache

- 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

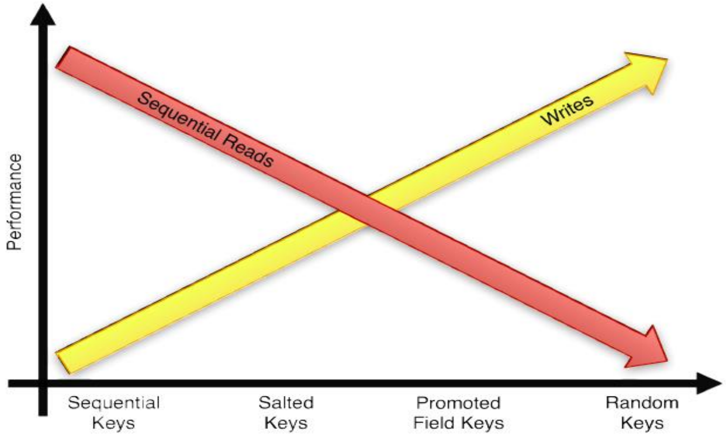
Tip: Disable major compactions and schedule to run manually (e.g. cron) at off-peak times

# Bloom Filters

- Defines a filter that allows to determine if a store file does **not** contain a row or column
- Error rate can control overhead but is usually very low, 1% or less
- Stored with each storage file on flush and compactions
- Good for large regions with many distinct row keys and many expected misses

# Key Design

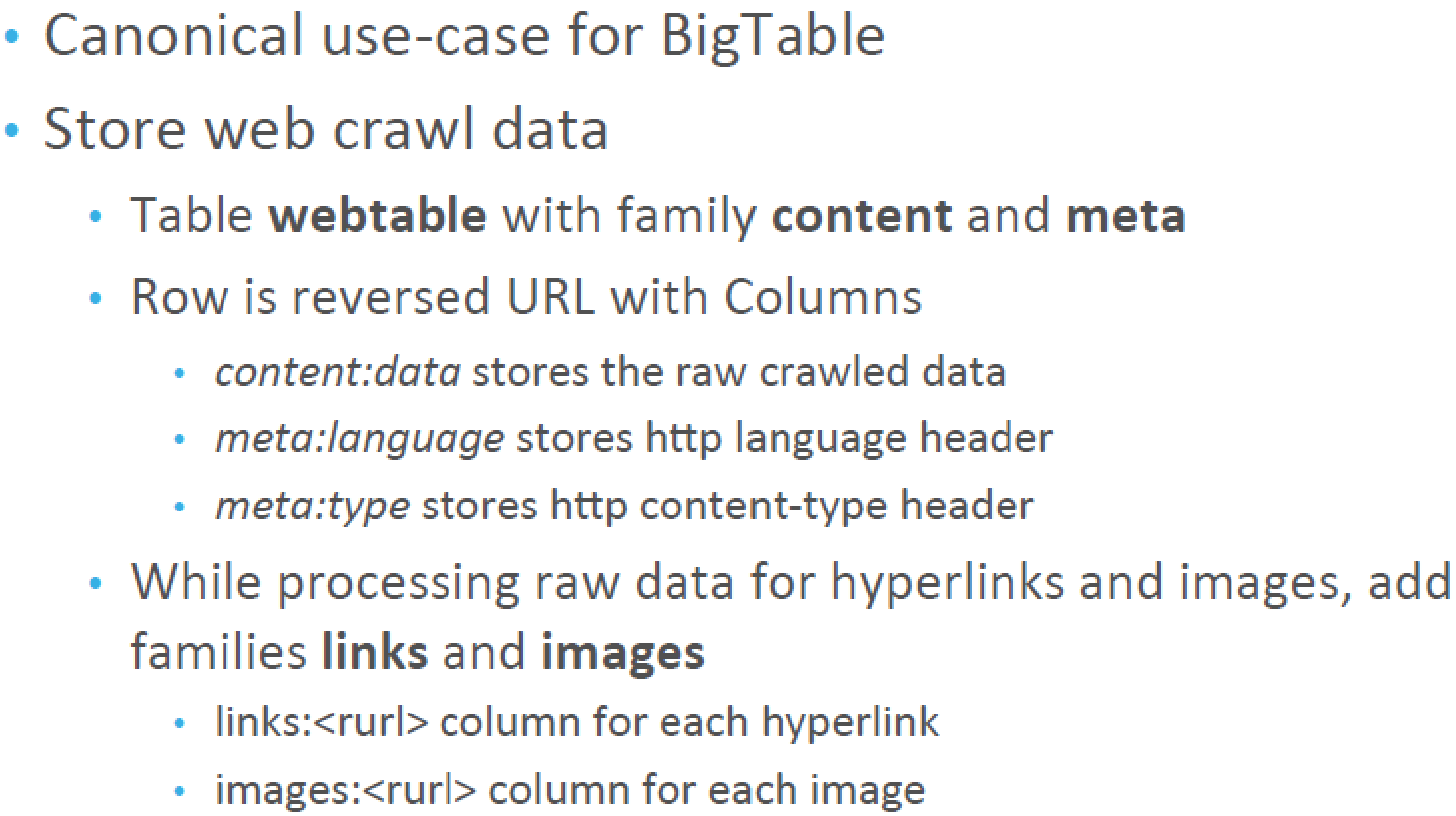
- Based on access pattern, either use sequential or random keys
- Often a combination of both is needed
  - Overcome architectural limitations
- Neither is necessarily bad
  - Use bulk import for sequential keys and reads
  - Random keys are good for random access patterns



# ColumnFamily 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
- Define compression or in-memory attributes to optimize access and performance

# Web Crawl Example



- Canonical use-case for BigTable
- Store web crawl data
  - Table **webtable** with family **content** and **meta**
  - Row is reversed URL with Columns
    - *content:data* stores the raw crawled data
    - *meta:language* stores http language header
    - *meta:type* stores http content-type header
  - While processing raw data for hyperlinks and images, add families **links** and **images**
    - links:<rurl> column for each hyperlink
    - images:<rurl> column for each image