HBase

Леонид Налчаджи
leonid.nalchadzhi@gmail.com

Яндекс

HBase

- Overview
- Table layout
- Architecture
- Client API
- Key design

Overview

Overview

- NoSQL
- Column oriented
- Versioned

Overview

- All rows ordered by row key
- All cells are uninterpreted arrays of bytes
- Auto-sharding

Overview

- Distributed
- HDFS based
- Highly fault tolerant
- CP/CAP

Overview

- Coordinates: <RowKey, CF:CN, Version>
- Data is grouped by column family
- Null values for free

Overview

- No actual deletes (tombstones)
- TTL for cells

Disadvantages

- No secondary indexes from the box
  - Available through additional table and coprocessors
- No transactions
  - CAS support, row locks, counters
- Not good for multiple data centers
  - Master-slave replication
- No complex query processing

Table layout

Table layout

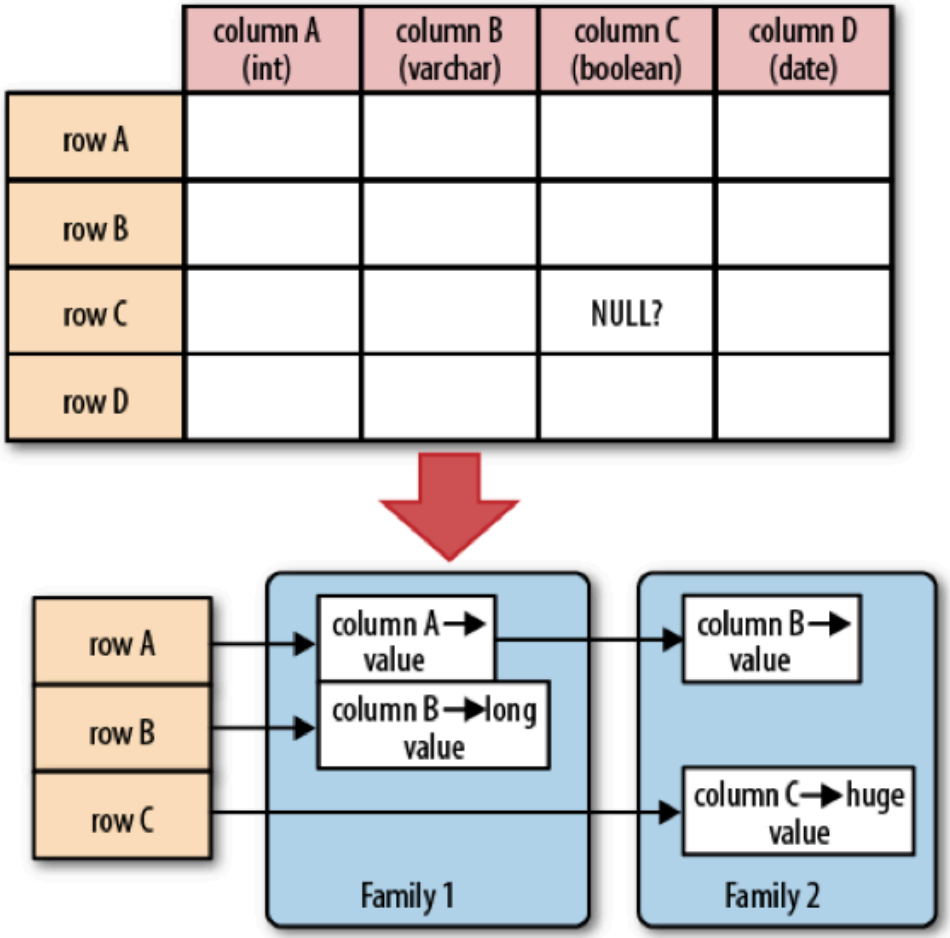


Table layout

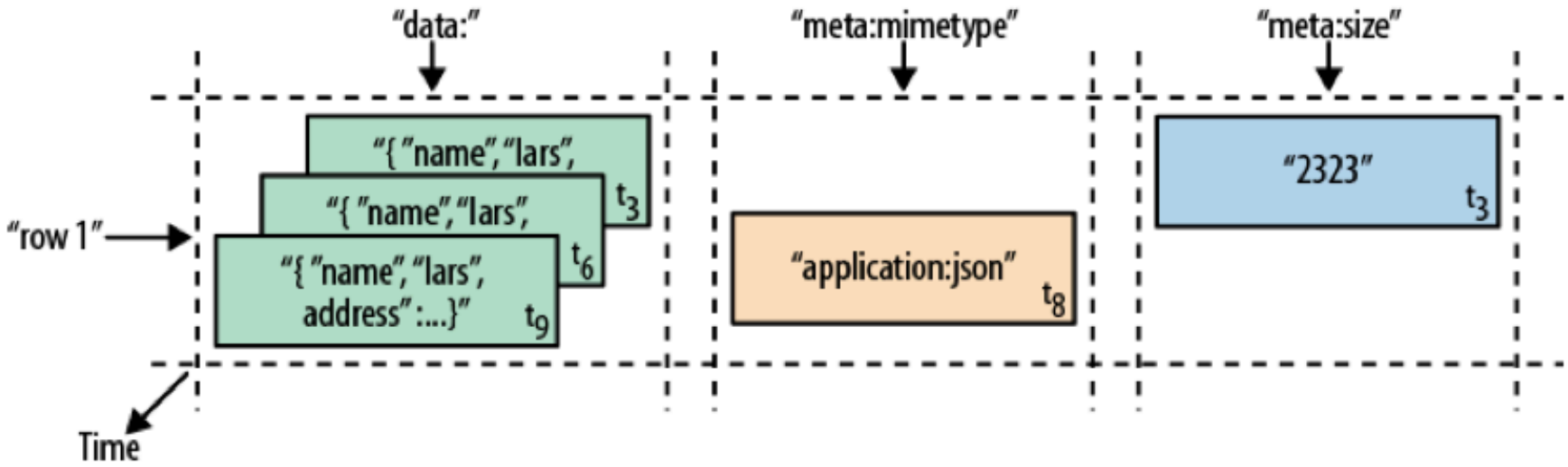


Table layout

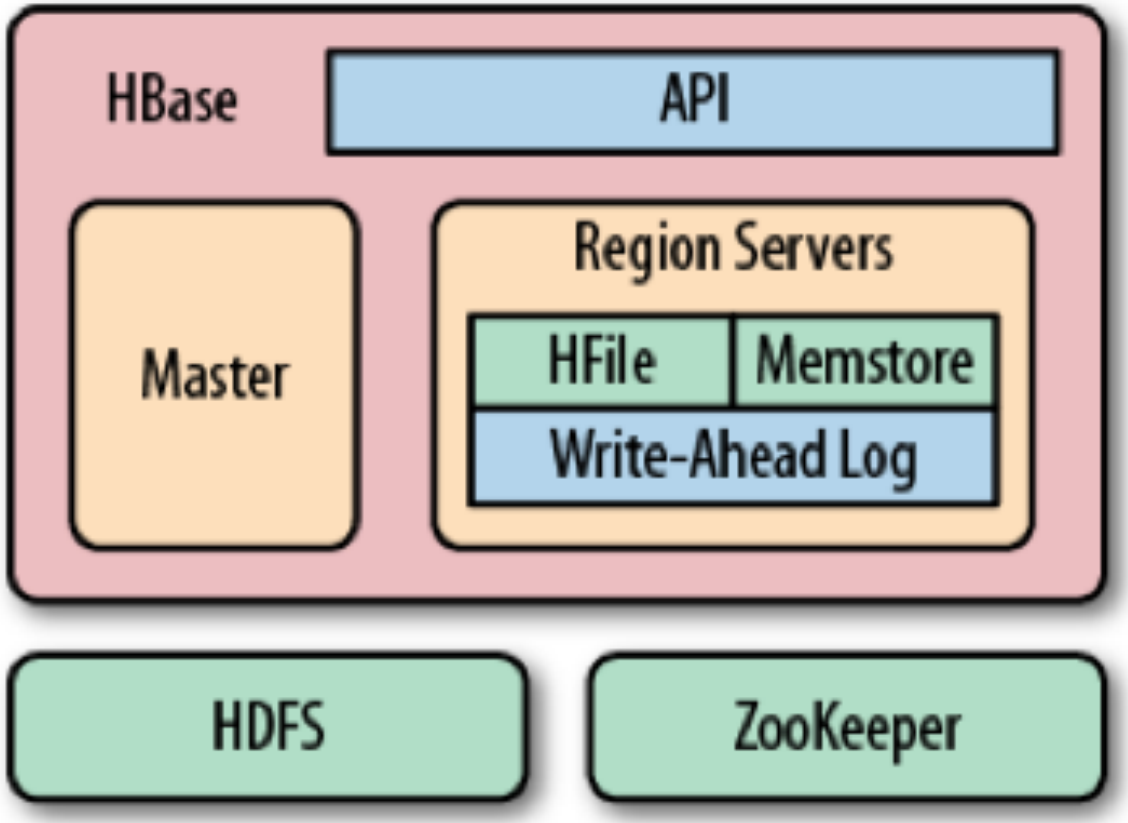
Row Key	Time Stamp	Column "data:"	Column "meta:" "mimetype"	Column "meta:" "size"	Column "counters:" "updates"
"row1"	t_3	"{ "name" : "lars", "address" : ...}"		"2323"	"1"
	t_6	"{ "name" : "lars", "address" : ...}"			"2"
	t_8		"application/json"		
	t_9	"{ "name" : "lars", "address" : ...}"			"3"

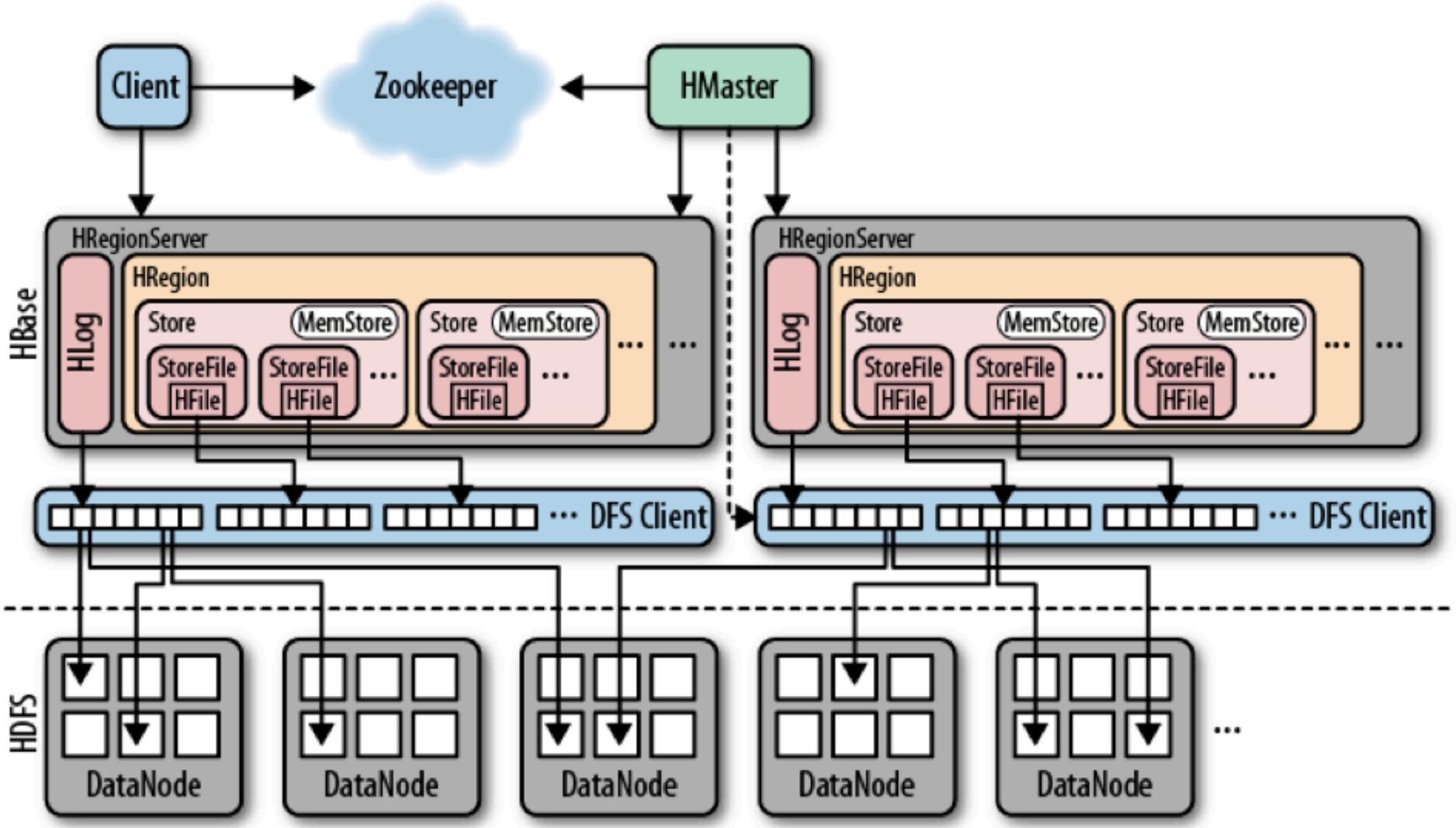
Architecture

Terms

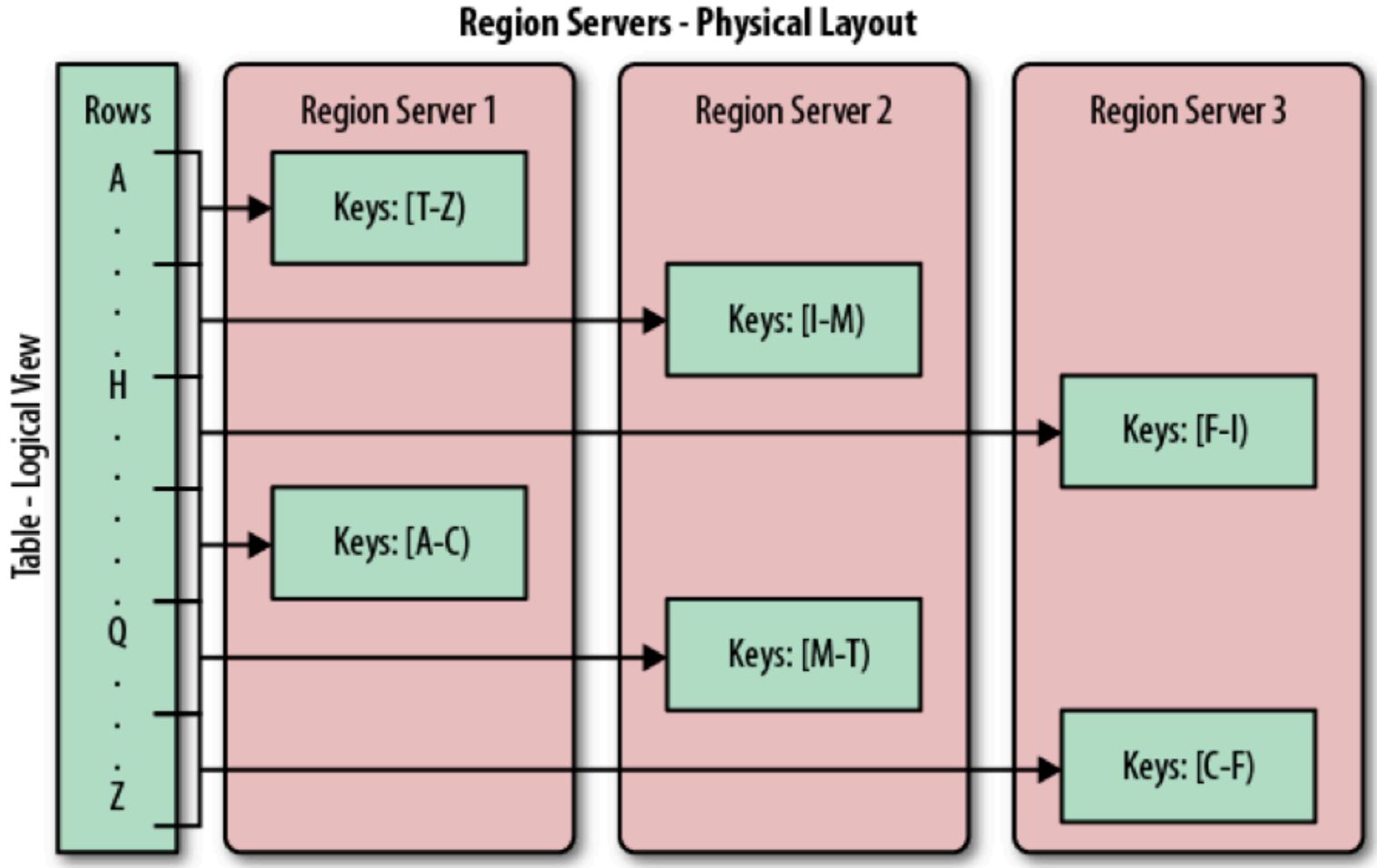
- Region
- Region server
- WAL
- Memstore
- StoreFile/HFile

Architecture




Client
Zookeeper
HMaster
HBase
HRegionServer
HLog
HRegion
Store
MemStore
StoreFile
HFile
DFS Client
HDFS
DataNode

Architecture



Architecture

- One master (with backup) many slaves (region servers)
- Very tight integration with ZooKeeper
- Client calls ZK, RS, Master

Master

- Handles schema changes
- Cluster housekeeping operations
- Not SPOF
- Check RS status through ZK

Region server

- Stores data in regions
- Region is a container of data
- Does actual data manipulation
- HDFS DataNode

ZooKeeper

- Distributed synchronization service
- FS-like tree structure contains keys
- Provides primitives to achieve a lot of goals

Storage

- Handled by region server
- Memstore: sorted write buffer
- 2 types of file
  - WAL log
  - Data storage

Client connection

- Ask ZK to get address -ROOT- region
- From -ROOT- gets the RS with .META.
- From .META. gets the address of RS
- Cache all these data

Write path

- Write to HLog (one per RS)
- Write to Memstore (one per CF)
- If Memstore is full, flush is requested
  - This request is processed async by another thread

On Memstore flush

- On flush Memstore is written to HDFS file
- The last operation sequence number is stored
- Memstore is cleared

Region splits

- Region is closed
- Creates directory structure for new regions (multithreaded)
- .META. is updated (parent and daughters)

Region splits

- Master is notified for load balancing
- All steps are tracked in ZK
- Parent cleaned up eventually

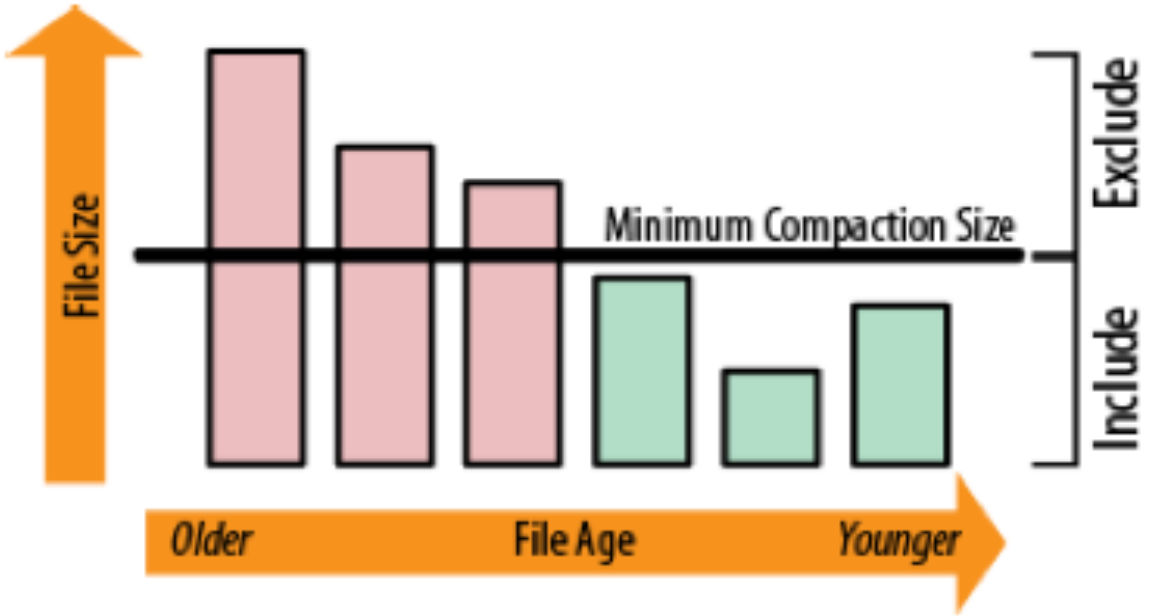
Compaction

- When Memstore is flushed new store file is created
- Compaction is a housekeeping mechanism which merges store files
- Come in two varieties: *minor* and *major*
- The type is determined when the compaction check is executed

Minor compaction

- Fast
- Selects files to include
- Params: max size and min number of files
- From the oldest to the newest

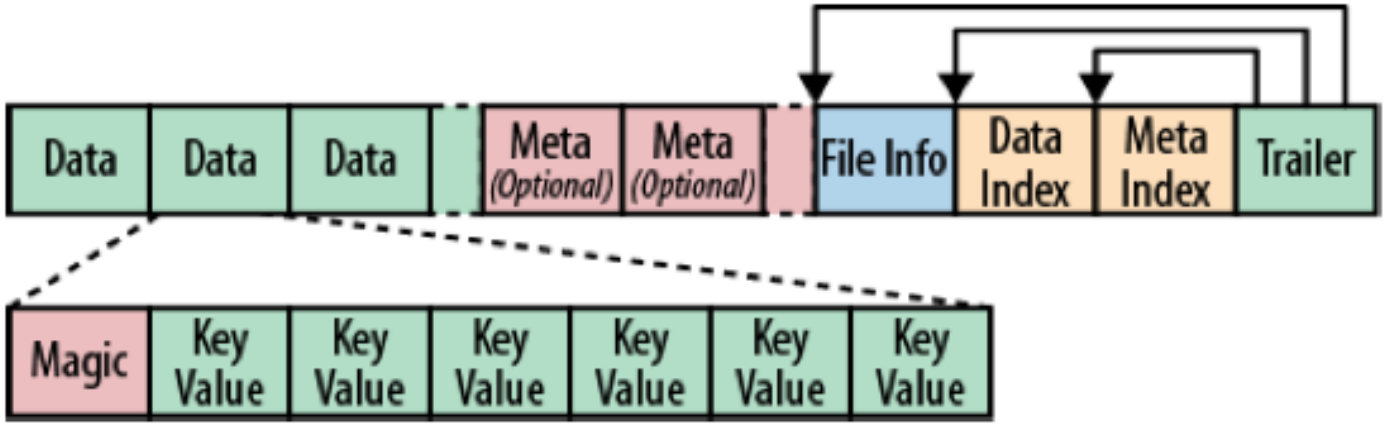
Minor compaction



Major compaction

- Compact all files into a single file
- Might be promoted from the minor compaction
- Starts periodically (each 24 hours by default)
- Checks tombstone markers and removes data

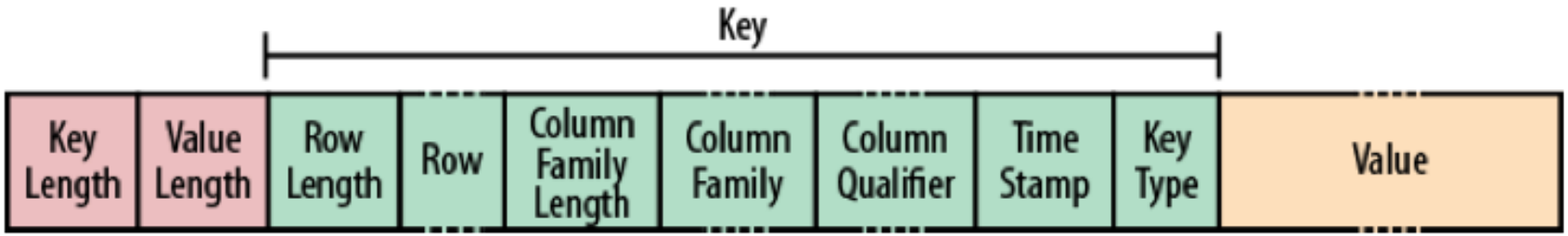
HFile format



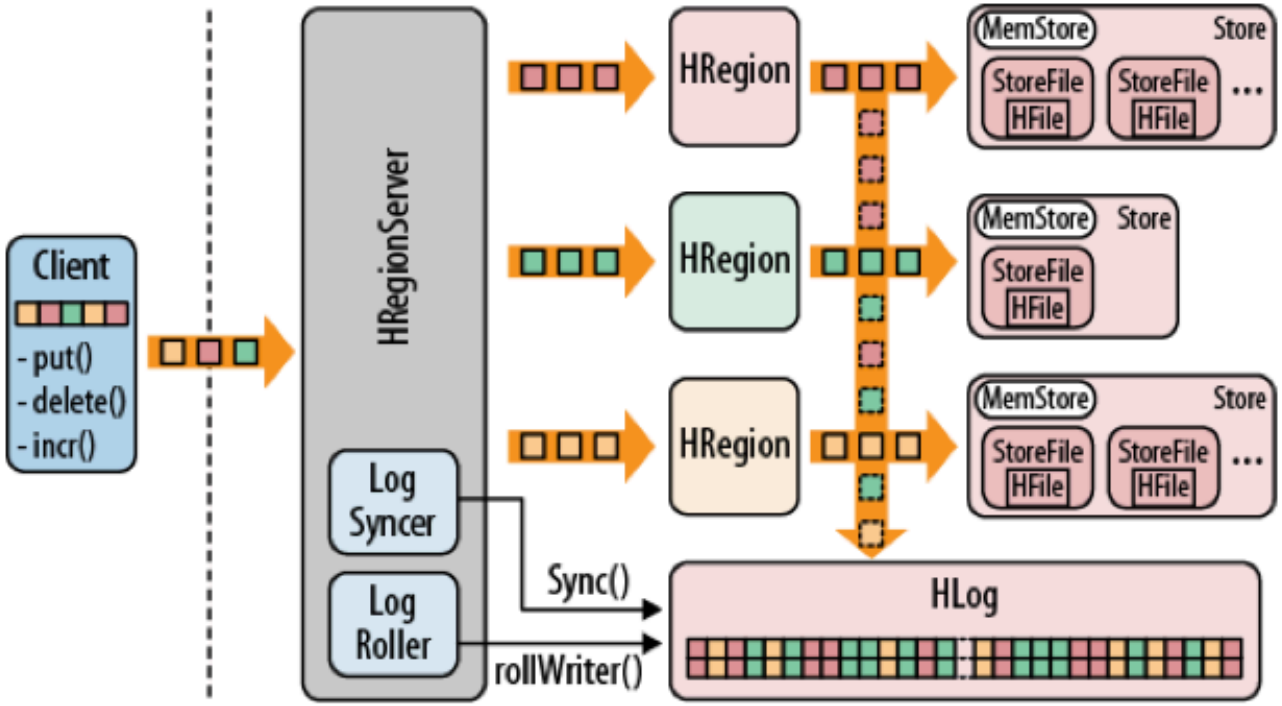
HFile format

- Immutable (trailer is written once)
- Trailer has pointers to other blocks
- Indexes store offsets to data, metadata
- Block size by default 64K
- Block contains magic and number of key values

KeyValue format



Write-Ahead log



Write-Ahead log

- One per region server
- Contains data from several regions
- Stores data sequentially

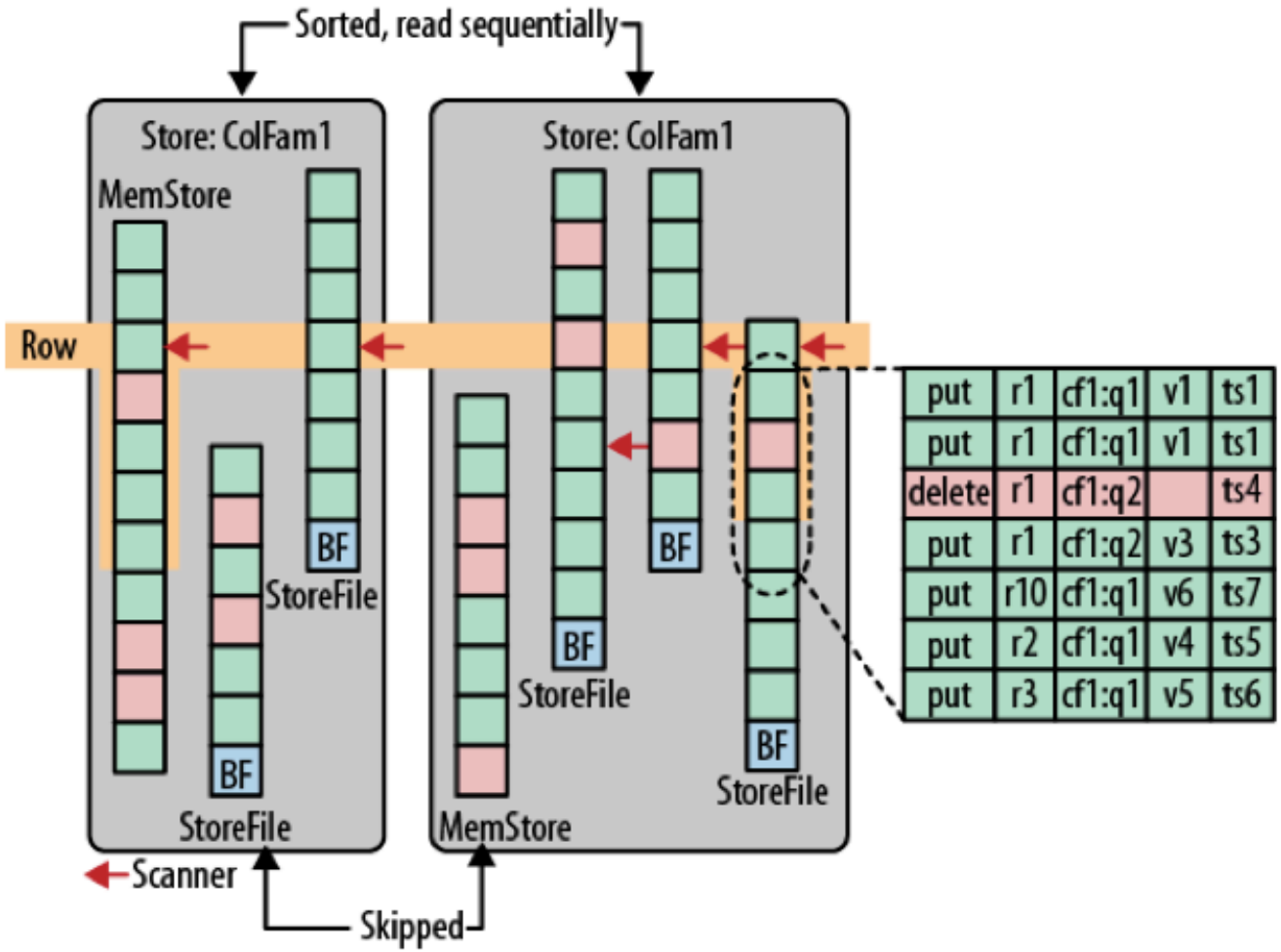
Write-Ahead log

- Rolled periodically (every hour by default)
- Applied on cluster restart and on server failure
- Split is distributed

Read path

- There is no single index with logical rows
- Data is stored in several files and Memstore per column family
- Gets are the same as scans

Read path


Sorted, read sequentially
Store: ColFam1
MemStore
Row
BF
StoreFile
BF
StoreFile
Scanner
Skipped
Store: ColFam1
BF
StoreFile
BF
StoreFile
MemStore
put r1 cf1:q1 v1 ts1
put r1 cf1:q1 v1 ts1
delete r1 cf1:q2 ts4
put r1 cf1:q2 v3 ts3
put r10 cf1:q1 v6 ts7
put r2 cf1:q1 v4 ts5
put r3 cf1:q1 v5 ts6

Block cache

- Each region has LRU block cache
- Based on priorities
  1. Single access
  2. Multi access priority
  3. Catalog CF: ROOT and META

Block cache

- Size of cache:
  - number of region servers * heap size * (hfile.block.cache.size) * 0.85
- 1 RS with 1 Gb RAM = 217 Mb
- 20 RS with 8 Gb RAM = 34 Gb
- 100 RS with 24 Gb RAM with 0.5 = 1 Tb

Block cache

- Always in cache:
  - Catalog tables
  - HFiles indexes
  - Keys (row keys, CF, TS)

Client API

CRUD

- All operations do through HTable object
- All operations per-row atomic
- Row lock is available
- Batch is available
- Looks like this:

```
HTable table = new HTable("table");

table.put(new Put(...));
```

Put

- Put(byte[] row)
- Put(byte[] row, RowLock rowLock)
- Put(byte[] row, long ts)
- Put(byte[] row, long ts, RowLock rowLock)

Put

- Put add(byte[] family, byte[] qualifier, byte[] value)
- Put add(byte[] family, byte[] qualifier, long ts, byte[] value)
- Put add(KeyValue kv)

Put

- setWriteToWAL()
- heapSize()

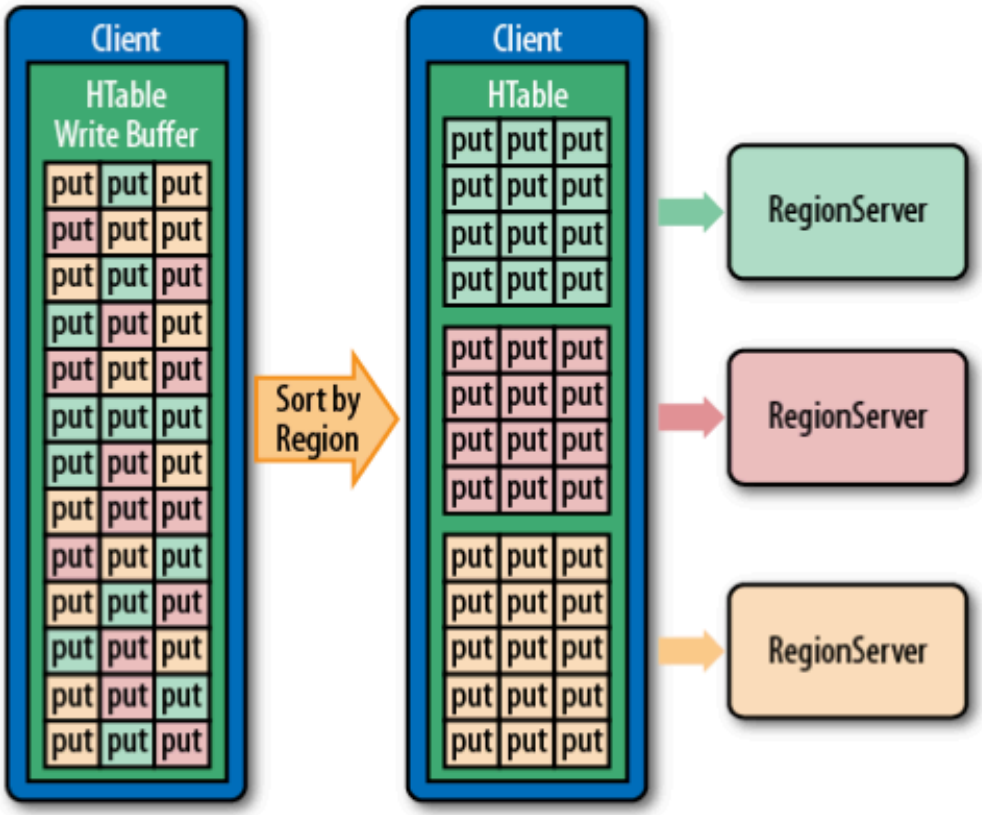
Write buffer

- There is a client-side write buffer
- Sorts data by Region Servers
- 2Mb by default
- Controlled by:

```
table.setAutoFlush(false);

table.flushCommits();
```

Write buffer



What else

- CAS:

```
boolean checkAndPut(byte[] row, byte[] family,
    byte[] qualifier, byte[] value, Put put) throws IOException
```

- Batch:

```
void put(List<Put> puts) throws IOException
```

Get

- Returns strictly one row

```
Result res = table.get(new Get(row));
```

- Can ask if row exists

```
boolean e = table.exist(new Get(row));
```

- Filter can be applied

Row lock

- Region server provide a row lock
- Client can ask for explicit lock:

```
RowLock lockRow(byte[] row) throws IOException;

void unlockRow(RowLock rl) throws IOException;
```

- Do *not* use if you do not have to

Scan

- Iterator over a number of rows
- Can be defined start and end rows
- Leased for amount of time
- Batching and caching is available

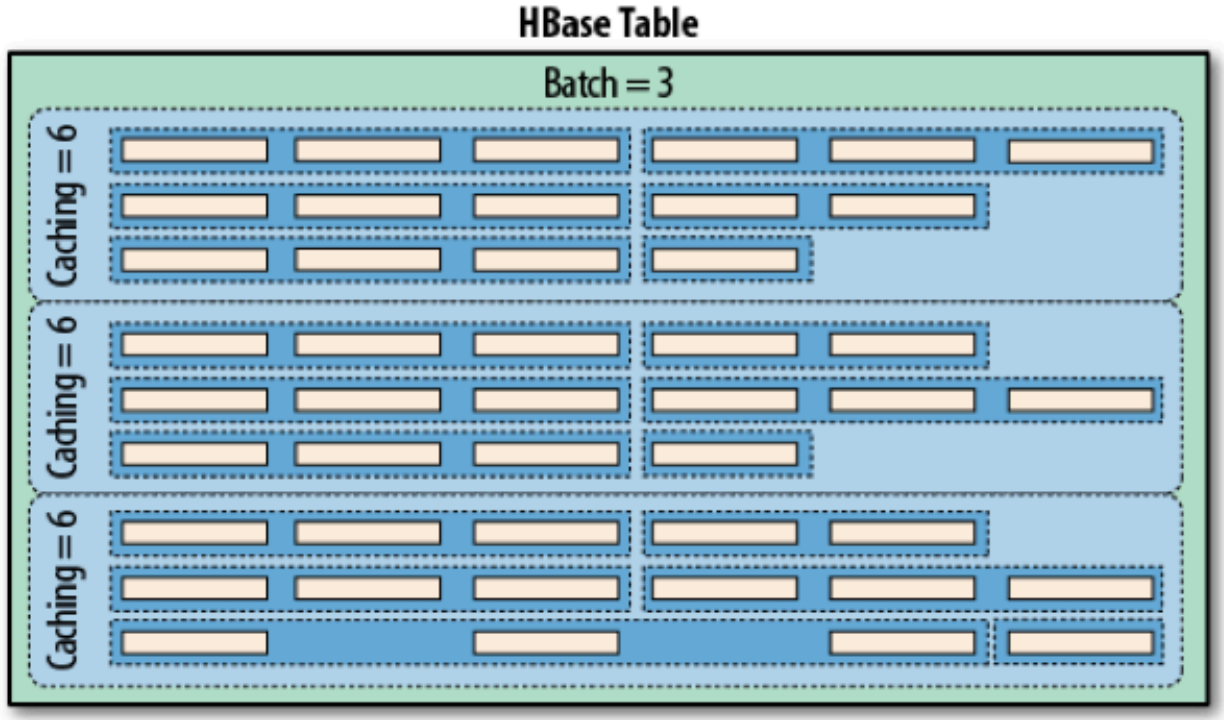
Scan

```
final Scan scan = new Scan();
scan.addFamily(Bytes.toBytes("colfam1"));
scan.setBatch(5);
scan.setCaching(5);
scan.setMaxVersions(1);
scan.setTimeStamp(0);
scan.setStartRow(Bytes.toBytes(0));
scan.setStartRow(Bytes.toBytes(1000));
ResultScanner scanner = table.getScanner(scan);
for (Result res : scanner) {
...
}
scanner.close();
```

Scan

- Caching is for rows
- Batching is for columns
- Result.next() will return next set of batched cols (more than one for row)

Scan



Counters

- Any column can be treated as counters
- Atomic increment without locking
- Can be applied for multiple rows
- Looks like this:

```
long cnt = table.incrementColumnValue(Bytes.toBytes("20131028"),
    Bytes.toBytes("daily"), Bytes.toBytes("hits"), 1);
```

HTable pooling

- Creating an HTable instance is expensive
- HTable is *not* thread safe
- Should be created on start up and closed on application exit

HTable pooling

- Provided pool: HTablePool
- Thread safe
- Shares common resources

HTable pooling

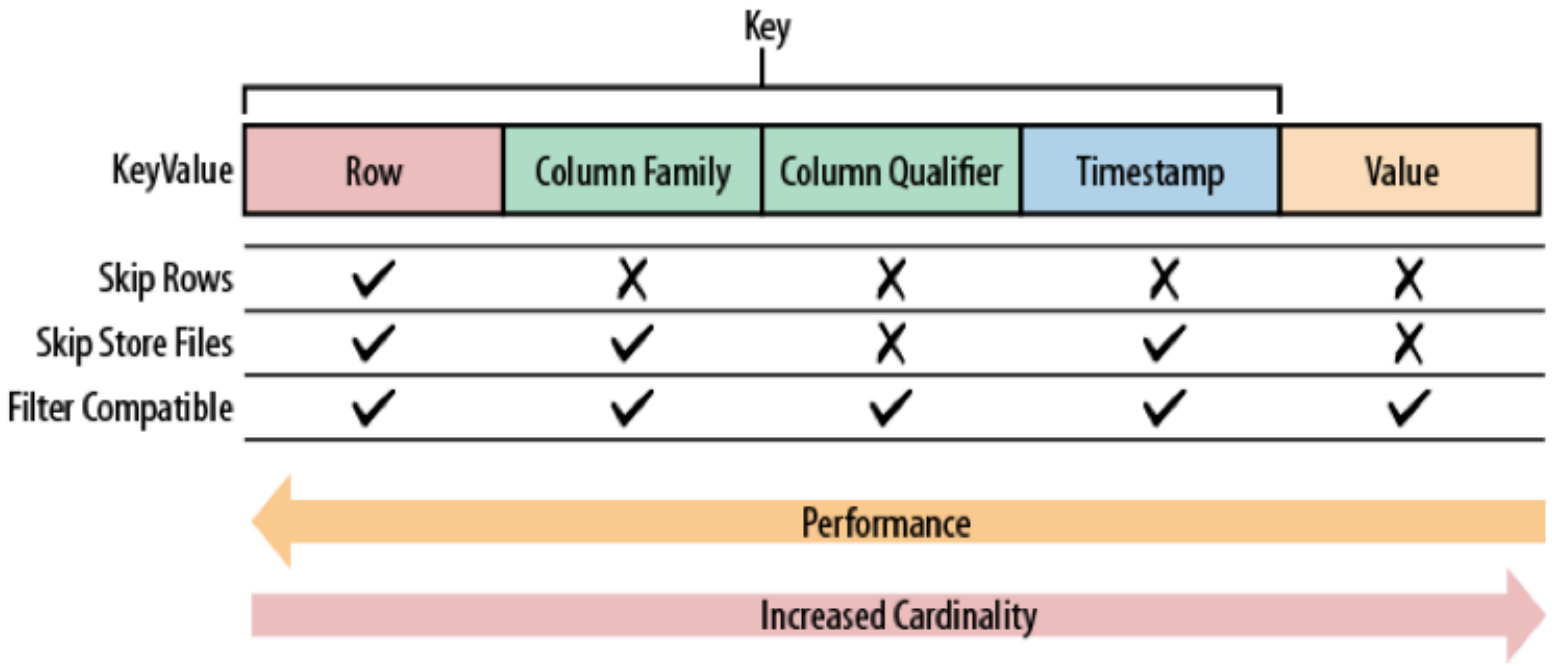
- HTablePool(Configuration config, int maxSize)
- HTableInterface getTable(String tableName)
- void putTable(HTableInterface table)
- void closeTablePool(String tableName)

Key design

Key design

- The only index is row key
- Row keys are sorted
- Key is uninterpreted array of bytes

Key design


Key
KeyValue
Row
Column Family
Column Qualifier
Timestamp
Value
Skip Rows
Skip Store Files
Filter Compatible
Performance
Increased Cardinality

Email example

Approach #1: user id is raw key, all messages in one cell

<userId> : <cf> : <col> : <ts>:<messages>

12345 : data : msgs : 1307097848 : ${msg1}, ${msg2}...

Email example

Approach #1: user id is raw key, all messages in one cell

<userId> : <cf> : <colval> : <messages>

- All in one request
- Hard to find anything
- Users with a lot of messages

Email example

Approach #2: user id is raw key, message id in column family

<userId> : <cf> : <msgId> :
<timestamp> : <email-message>

12345 : data : 5fc38314-e290-ae5da5fc375d : 1307097848 : "Hi, ..."

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 ..."

Email example

Approach #2: user id is raw key, message id in column family

<userId> : <cf> : <msgId> :
<timestamp> : <email-message>

- Can load data on demand
- Find anything only with filter
- Users with a lot of messages

Email example

Approach #3: raw id is a <userId> <msgId>

<userId> - <msgId>: <cf> : <qualifier> :
<timestamp> : <email-message>

12345-5fc38314-e290-ae5da5fc375d : data : : 1307097848 : "Hi, ..."

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 ..."

Email example

Approach #3: raw id is a <userId> <msgId>

<userId> - <msgId>: <cf> : <qualifier> :
<timestamp> : <email-message>

- Can load data on demand
- Find message with row key
- Users with a lot of messages

Email example

- Go further:

<userId>-<date>-<messageId>-<attachmentId>

Email example

- Go further:

  <userId>-<date>-<messageId>-<attachmentId>

  <userId>-(MAX_LONG - <date>)-<messageId>

  -<attachmentId>

Performance

Performance

- 10 Gb data, 10 parallel clients, 1kb per entry
- HDFs replication = 3
- 3 Region servers, 2CPU (24 cores), 48Gb RAM, 10GB for App

	nobuf, noWAL	buf:100, noWAL	buf:1000, noWAL	nobuf, WAL	buf:100, WAL	buf:1000, WAL
MB/s	12MB/s	53MB/s	48MB/s	5MB/s	11MB/s	31MB/s
puts/s	11k rows/s	50k rows/s	45k rows/s	4.7k rows/s	10k rows/s	30k rows/s

Sources

- HBase. The defenitive guide (O'Reilly)
- HBase in action (Manning)
- Official documentation

  http://hbase.apache.org/0.94/book.html

- Cloudera articles

  http://blog.cloudera.com/blog/category/hbase/

HBase

Леонид Налчаджи
leonid.nalchadzhi@gmail.com

Яндекс