

# Gemini HBASE Workshop

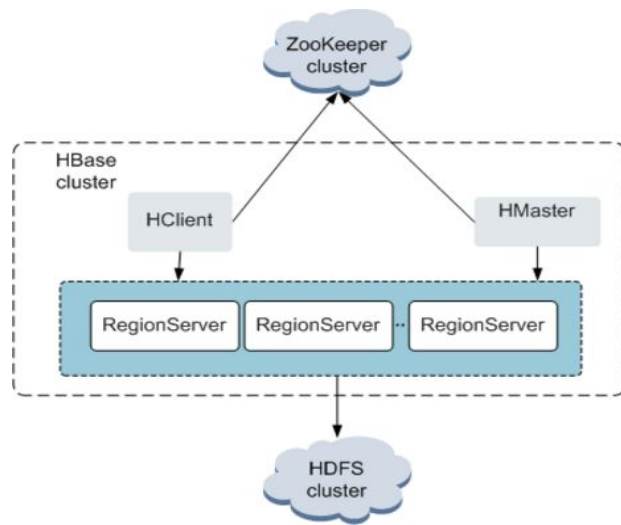
02-02-2016

# Agenda

- HBASE Fundamentals & Architecture
- HBASE Client API
- HBASE Schema Design Patterns
- HBASE Map/Reduce Integration
- Use Cases
  - COW Domain Model
  - Extracts - Query Index Tables, Offer Index tables (GPA)
  - Budget HBase tables
  - Yamas Metrics Tables
- HBASE Shell Ruby scripting

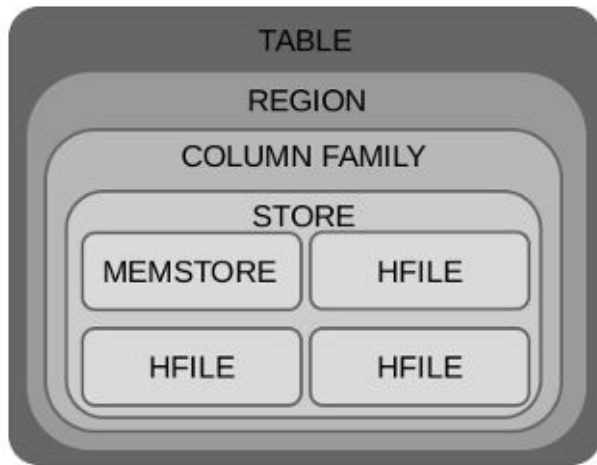
# HBASE

- Column-Oriented Data Store
- Supports random real-time CRUD operations
- Distributed - Designed to serve large tables
- Horizontally scalable
- Automatic fail-over
- Simple JAVA API (Written in Java)
- Implemented on top of HDFS
- Great for single random selects and range scans by key
- Great for variable schema
  - If schema has many columns and most of them are null

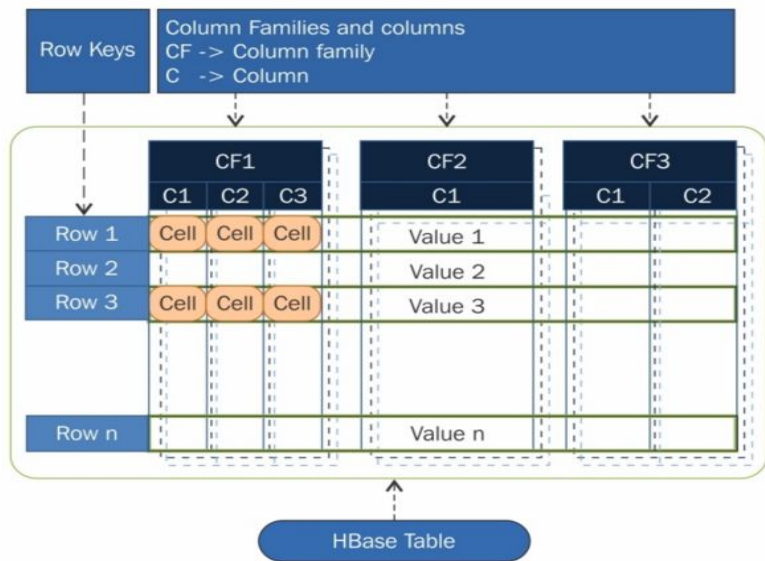


# Data Model

- Data is stored in tables
  - Tables are grouped into namespaces
- Tables contain rows
  - Rows are referenced by unique key
  - Key is an array of bytes
- Rows made of columns which are grouped in column families
  - A way to organize data
  - Various features are applied to families (compression, stored together)
  - Family definitions are static
  - Limited to small number of **column families**
- Data is stored in cells
  - Identified by row x and column-family y column
  - Cell's content is also an array of bytes



# HBASE Data Model - Example



- Column
  - Exists only when inserted
  - Can have multiple versions
  - Each row can have different sets of columns
  - All columns in a column family are stored together in the same low-level storage files, called **HFile**
- Row Key
  - Used for storing ordered rows
  - The sorting of rows done lexicographically by their key

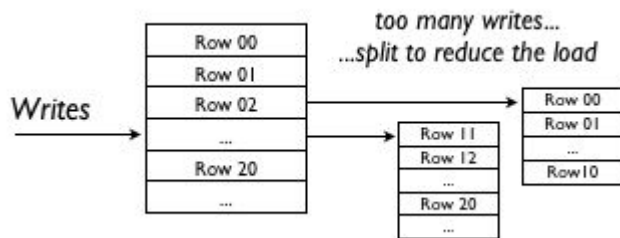
# Column Versions

- Every cell is associated with timestamp
- Can be used to save multiple versions of a value as it changes over time
- User can specify how many versions of a value should be kept
- HBase, is a sparse, distributed, persistent, multidimensional map, which is indexed by row key, column key, and a timestamp

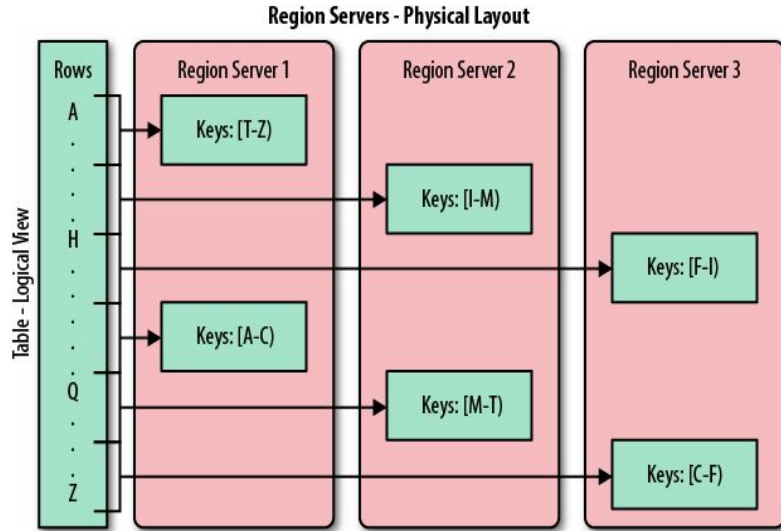
**(Table, RowKey, Family, Column, Timestamp) → Value**

# Auto Sharding

- Basic unit of scalability and load balancing in HBase -> **Region**
- Regions are essentially contiguous ranges of rows stored together
- Dynamically split when regions become big
- Initially there is only one region for table, splits into 2 regions by middle key when table becomes too large
- Each region is exactly served by one **region server**



# Auto Sharding - Region Servers



- Each Region Server is responsible to serve a set of regions
- One Region (i.e. range of rows) can be served only by one Region Server



# Gemini Campaign Tables - Luxblue HBASE

- <http://luxblue-hb.blue.ygrid.yahoo.com:50500/master-status>

Table	Number of Regions	Table Description
<a href="#">cow_prod:PROD_COW_AD</a>	561	'cow_prod:PROD_COW_AD', {NAME => 'd', VERSIONS => '10', COMPRESSION => 'GZ', TTL => '604800 SECONDS (7 DAYS)', MIN_VERSIONS => '2'}
<a href="#">cow_prod:PROD_COW_ADEXTENSION</a>	83	cow_prod:PROD_COW_ADEXTENSION', {NAME => 'd', VERSIONS => '10', COMPRESSION => 'GZ', TTL => '604800 SECONDS (7 DAYS)', MIN_VERSIONS => '2'}
<a href="#">cow_prod:PROD_COW_ADGROUP</a>	197	
<a href="#">cow_prod:PROD_COW_ADVERTISER</a>	7	
<a href="#">cow_prod:PROD_COW_CAMPAIGN</a>	14	
<a href="#">cow_prod:PROD_COW_DOMAINOBJECTS</a>	22	
<a href="#">cow_prod:PROD_COW_PARENTMAPPING</a>	704	
<a href="#">cow_prod:PROD_COW_TARGETINGATTRIB</a>	2372	
<a href="#">cow_prod:PROD_COW_OFFER</a>	214	

# Explore cow\_prod:PROD\_COW\_ADVERTISER

- [http://luxblue-hb.blue.ygrid.yahoo.com:50500/table.jsp?name=cow\\_prod:PROD\\_COW\\_ADVERTISER](http://luxblue-hb.blue.ygrid.yahoo.com:50500/table.jsp?name=cow_prod:PROD_COW_ADVERTISER)

## Table Regions

Name	Region Server	Start Key	End Key
cow_prod:PROD_COW_ADVERTISER,,1425422180311.4e07a6ff55c79955bf31343743f4c731.	<a href="#">gsbl877n03.blue.ygrid.yahoo.com:50511</a>		1025301
cow_prod:PROD_COW_ADVERTISER,1025301,1425422180311.873c9a5fbb76a473b018b70455e34a75.	<a href="#">gsbl840n05.blue.ygrid.yahoo.com:50511</a>	1025301	36578
cow_prod:PROD_COW_ADVERTISER,36578,1425422180311.d8e0bbbc1f7d309fe267d0f253293717.	<a href="#">gsbl840n19.blue.ygrid.yahoo.com:50511</a>	36578	65986
cow_prod:PROD_COW_ADVERTISER,65986,1425422180311.1125c72c81855a9b0dea8827b73c35d1.	<a href="#">gsbl840n14.blue.ygrid.yahoo.com:50511</a>	65986	914954
cow_prod:PROD_COW_ADVERTISER,914954,1425422180311.089c2c97bd9c91a7ce329ba73bf397e7.	<a href="#">gsbl841n21.blue.ygrid.yahoo.com:50511</a>	914954	952259
cow_prod:PROD_COW_ADVERTISER,952259,1425422180311.31e98fcb6be6817851cf497a7540726.	<a href="#">gsbl841n10.blue.ygrid.yahoo.com:50511</a>	952259	999995
cow_prod:PROD_COW_ADVERTISER,999995,1425422180311.0a6e14ab1a2f78026ecbcc27e8c321f1.	<a href="#">gsbl840n17.blue.ygrid.yahoo.com:50511</a>	999995	

# Explore table schema

```
[srinathm@gwbl395n06 ~]$ hbase shell
```

```
hbase(main):003:0> list_namespace_tables 'cow_prod'
```

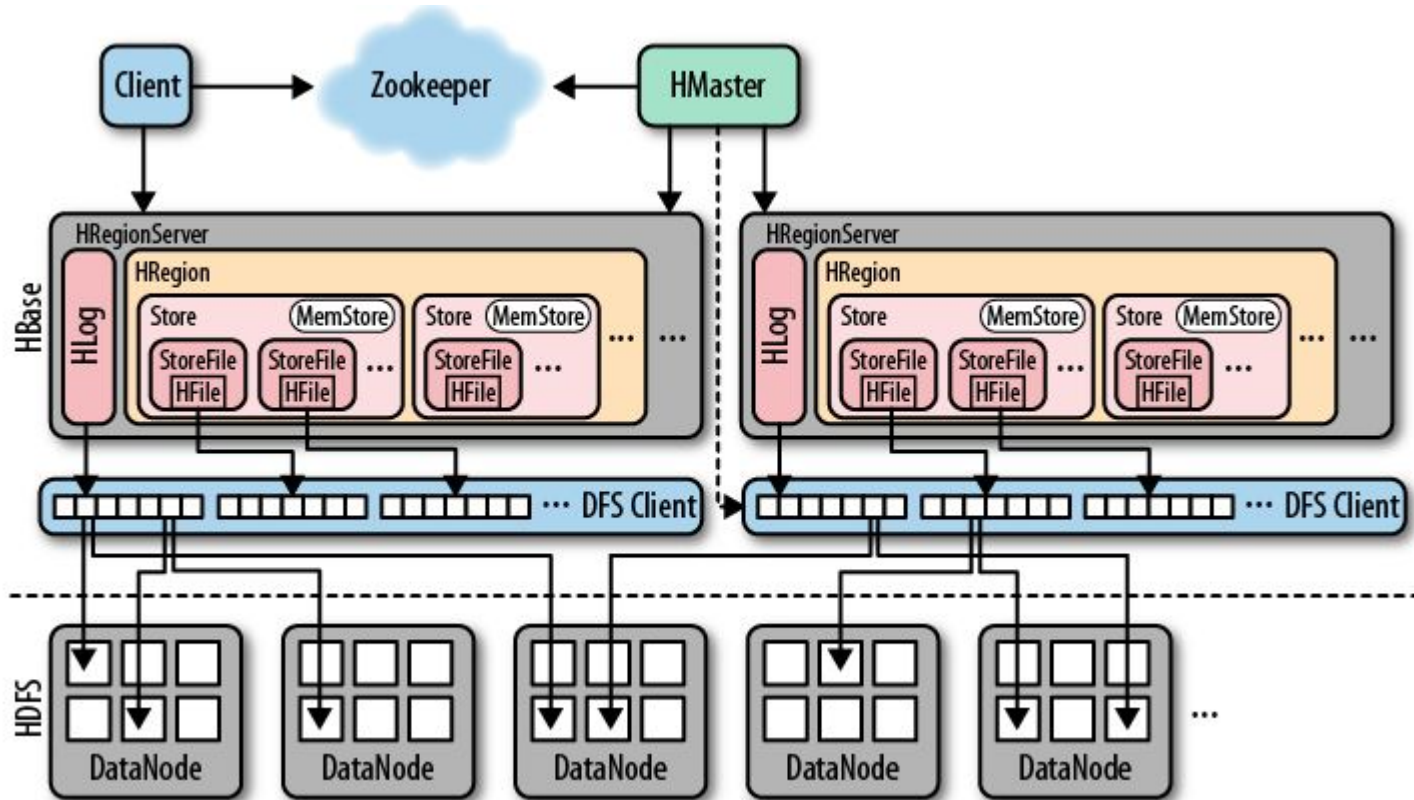
```
hbase(main):001:0> describe 'cow_prod:PROD_COW_ADVERTISER'
```

```
'cow_prod:PROD_COW_ADVERTISER', {NAME => 'd', VERSIONS => '10', COMPRESSION => 'GZ', TTL => '604800 SECONDS (7 DAYS)', MIN_VERSIONS => '2'}
```

- One column family - (NAME => 'd')
- Maximum and minimum number of versions to keep for a given column (VERSIONS => '10', MIN\_VERSIONS => '2')
- Time-to-live (TTL) - the number of seconds before a version is deleted (TTL => '604800 SECONDS (7 DAYS))
- HBase sorts the versions of a cell from newest to oldest, by sorting the timestamps lexicographically
- If MIN\_VERSIONS and TTL conflict, MIN\_VERSIONS takes precedence.
- How can I retrieve multiple versions of a particular column of a particular row?

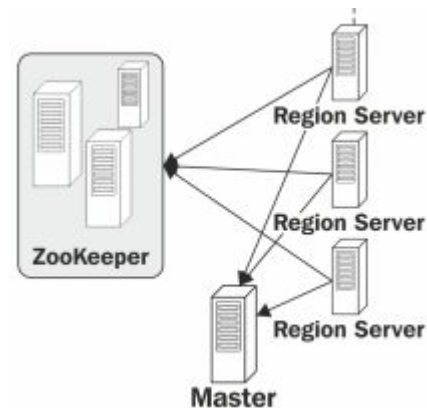
```
○ hbase(main):011:0> get 'cow_prod:PROD_COW_ADVERTISER', 1000117, {COLUMN => 'd:m', VERSIONS=>2}
```

# HBASE Storage Architecture



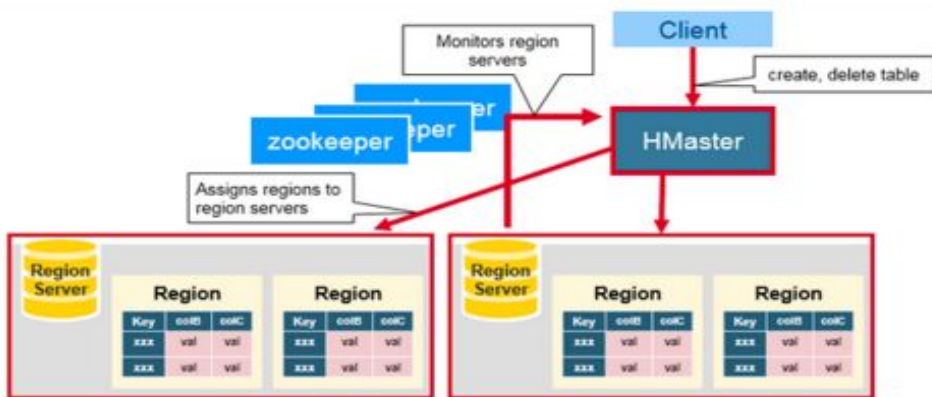
# HMaster

- One Master Server and a number of Region Servers
  - HMaster monitors regions servers which store and manage regions
  - Regions are contiguous ranges of rows stored together
  - Data is stored in persistent storage files called HFiles
- Zookeeper
  - HBase uses ZooKeeper for distributed coordination
  - Each region server created one ephemeral node in zookeeper
  - HMaster discovers available region servers through zookeeper
  - HBase also uses zookeeper to make sure that only one HMaster running

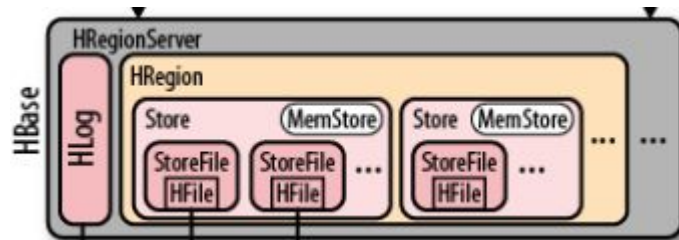


# HMaster Responsibilities

- Co-ordinating Region Servers
  - Assigning regions on startup
  - Re-assigning regions for recovery or load balancing
  - Monitoring all RegionServer instances in the cluster
- Admin Functions
  - Interface for creating, deleting, updating tables

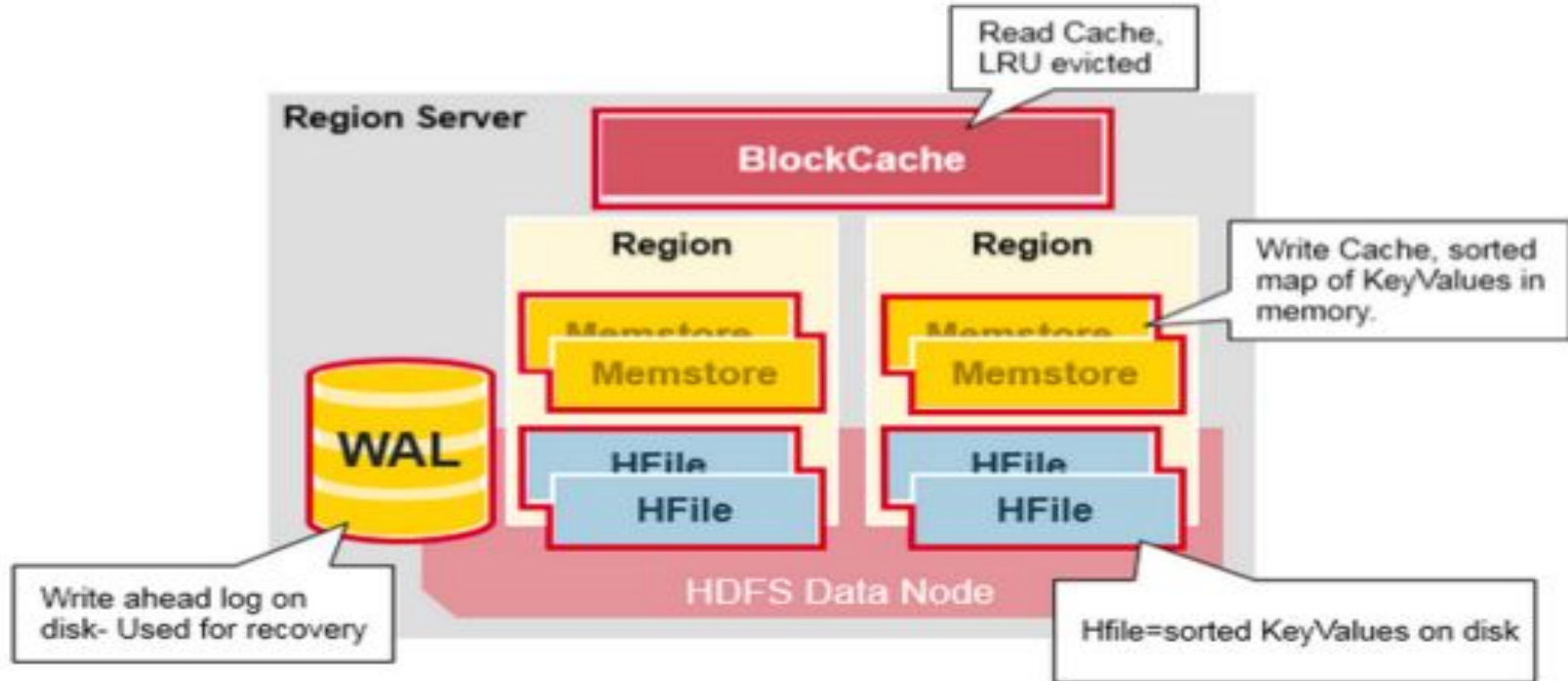


# Components of HRegionServer



- HRegionServer hosts a region by creating corresponding HRegion object
- Multiple regions are hosted by a single HRegionServer
- Sets up a Store instance for each column family in a Region
  - Each table can have multiple column families
- Each Store has in-memory data store - MemStore - Write cache.
  - Stores new data which has not yet been written to disk.
  - Sorted before writing to disk.
  - One MemStore per column family per region
- Hfiles store the rows as sorted KeyValues on disk
  - Multiple HFiles can exist for one column family - (Compaction later)
- HLog - Write Ahead Log (WAL) is a file on HDFS
  - Stores new data that hasn't yet been persisted to permanent storage
  - Used for recovery in the case of failure.

# Region Server - Summary





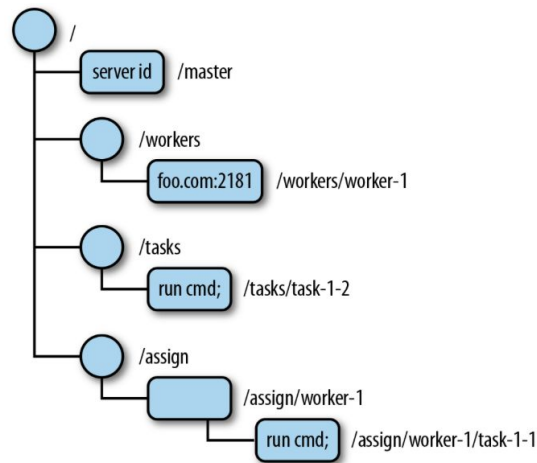
# HBase ZNodes

- Zookeeper

- Client/Server system for distributed coordination
- Exposes interface similar to file system
- Each node called **znode** may contain data and a set of children
- Each znode has a name and identified using file system like path (/root/sub/znode\_1)

- Zookeeper in HBase

- Zookeeper coordinates, communicates and shares state between Master and Region Servers  
(Stores only transient data)

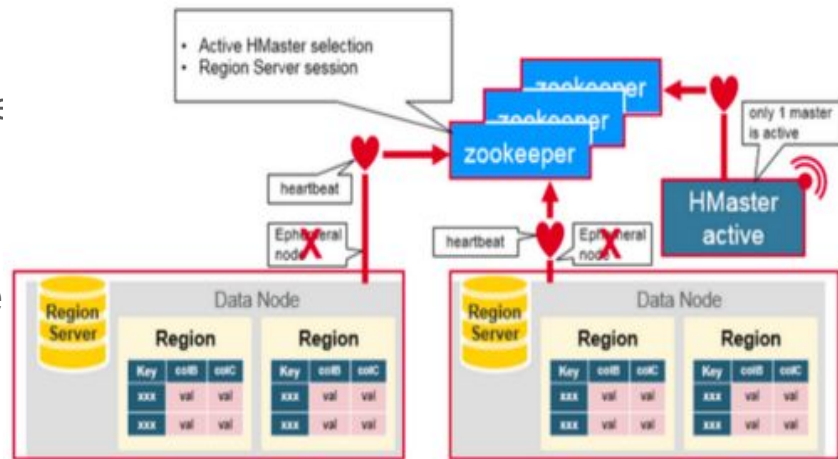


# HBase Zookeeper Shell

- Check hbase-site.xml for zookeeper configuration (/home/gs/conf/hbase/hbase-site.xml)
  - hbase.zookeeper.quorum
  - zookeeper.znode.parent - (/hbase/luxblue-hb1) - Root node for HBASE in Zookeeper
  - zookeeper.znode.rootserver - (root-region-server)
- hbase zkcli - Interactive shell with zookeeper
  - get /hbase/luxblue-hb1 - Contents of parent z node (numChildren = 4)
  - ls /hbase/luxblue-hb1 - [meta-region-server, hbaseid, table, master]
  - get /hbase/luxblue-hb1/meta-region-server
    - Contains the location of server hosting META (later)
  - get /hbase/luxblue-hb1/master
    - The “active” master will register its own address in this znode at startup, making this znode the source of truth for identifying which server is the Master.

# Master/Region Servers - Zookeeper

- Master and RegionServers register themselves with Zookeeper
  - master - Current master server
  - rs - dir contains one znode per region server
  - Each region server creates an ephemeral node
- Master monitors RS ephemeral nodes to discover available region servers
- Master also tracks these nodes for server failures
- Uses zookeeper to make sure that only one master is registered



# ROOT And META Tables

- HBase has 2 special tables: **'root' And 'meta'**
  - META holds the location of the regions of all the tables.
  - ROOT then holds the location of .META.
  - HBASE stores the location of ROOT in a znode in a ZooKeeper.
  - It's the first place a client application looks when it needs to locate data stored in HBase

```
hbase(main):002:0> scan 'hbase:root'
ROW                                COLUMN+CELL
hbase:meta,,1  column=info:regioninfo, timestamp=1416817905424, value={ENCODED => 1588230740, NAME => 'hbase:meta,,1', STARTKEY => '', ENDKEY => ''}
hbase:meta,,1  column=info:seqnumDuringOpen, timestamp=1453324613717, value=\x00\x00\x00\x04N\x81b\x7F
hbase:meta,,1  column=info:server, timestamp=1453324613717, value=gsbl197n34.blue.ygrid.yahoo.com:50511
hbase:meta,,1  column=info:serverstartcode, timestamp=1453324613717, value=1453324610796
hbase:meta,,1  column=info:sn, timestamp=1453324613311, value=gsbl197n34.blue.ygrid.yahoo.com,50511,1453324610796
hbase:meta,,1  column=info:state, timestamp=1453324613717, value=OPEN
hbase:meta,,1  column=info:v, timestamp=1416817905424, value=\x00\x01
```

# HBASE Meta Table

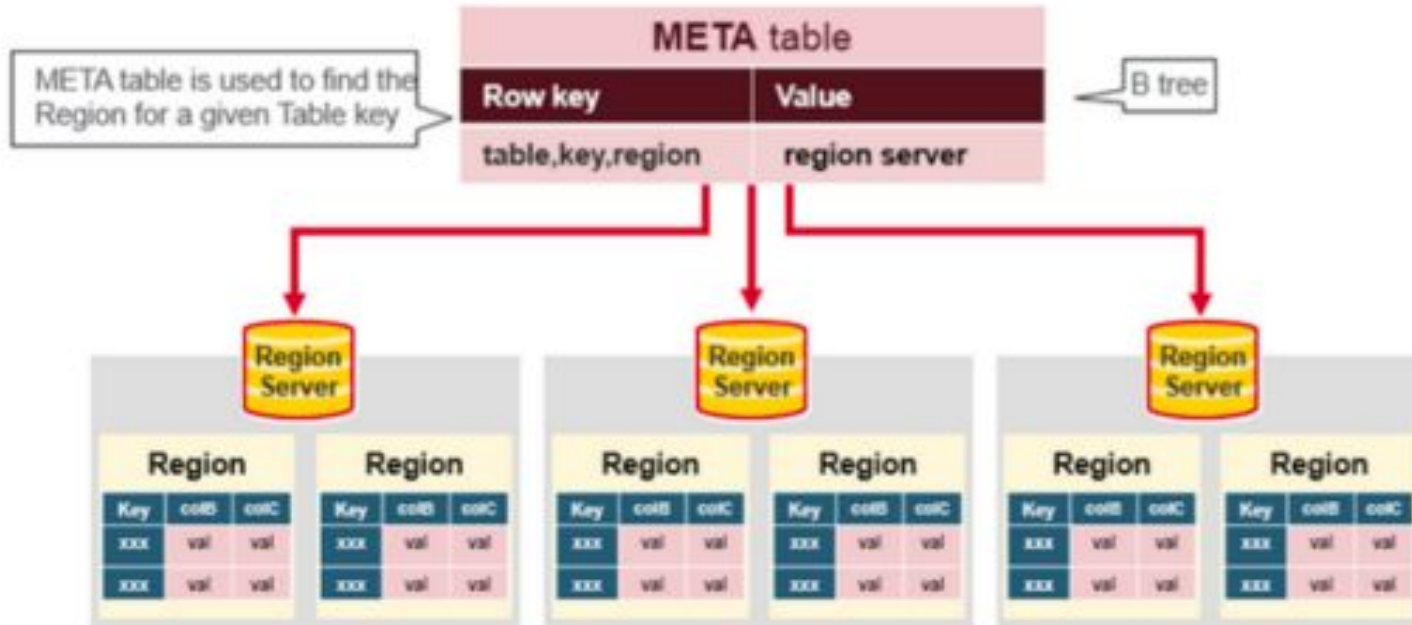
- META keeps track of the current location of each region, for each table
- A row in .META. corresponds to a single region.
- Region information includes server info about the server hosting this region
- Let's look at region information for PROD\_COW\_ADVERTISER table

```
hbase(main):007:0> scan 'hbase:meta', {STARTROW=>'cow_prod:PROD_COW_ADVERTISER', STOPROW=>'cow_prod:PROD_COW_ADVERTISER.', COLUMNS=>['info:regioninfo', 'info:server']}
```

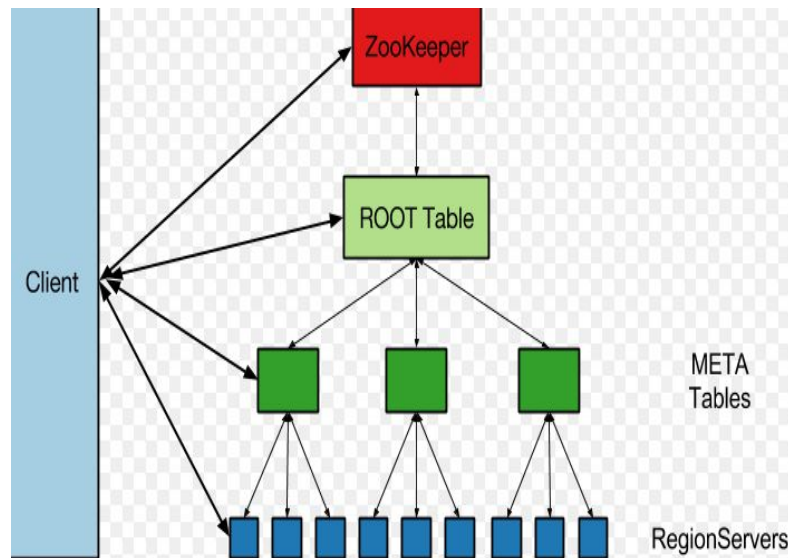
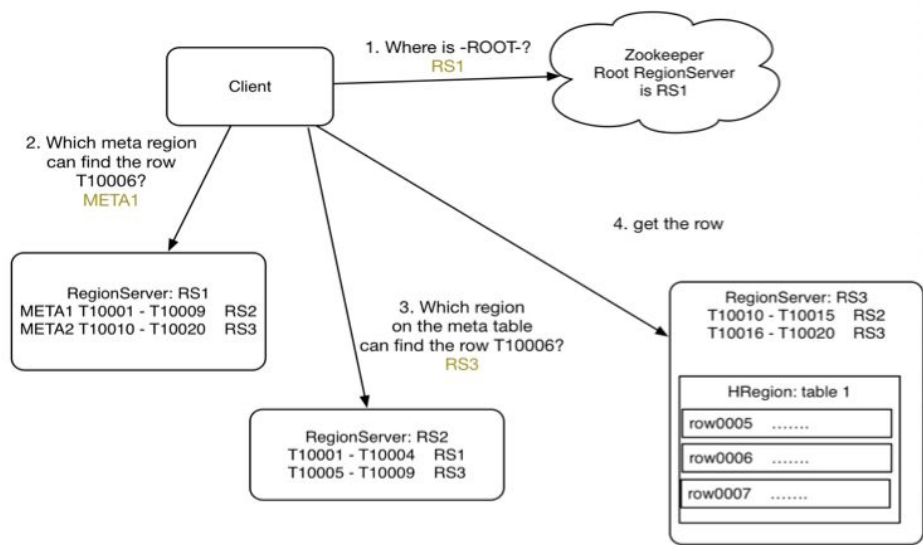
```
hbase(main):005:0> scan 'hbase:meta',{FILTER=>"PrefixFilter('cow_prod:PROD_COW_ADVERTISER')", COLUMNS=>['info:regioninfo']}
```

```
cow_prod:PROD_COW_ADVERTISER,,1425422180311.4e07a6f column=info:regioninfo, timestamp=1425422180560, value={ENCODED => 4e07a6ff55c79955bf31343743f4c731, NAME => 'cow_prod:PROD_COW_ADVERTISER,,1425422180311.4e07a6ff55c79955bf31343743f4c731.', STARTKEY => '', ENDKEY => '1025301'}
f55c79955bf31343743f4c731.
cow_prod:PROD_COW_ADVERTISER,,1425422180311.4e07a6f column=info:server, timestamp=1453441367387, value=gsbl877n03.blue.ygrid.yahoo.com:50511
f55c79955bf31343743f4c731.
cow_prod:PROD_COW_ADVERTISER,1025301,1425422180311. column=info:regioninfo, timestamp=1425422180560, value={ENCODED => 873c9a5fbb76a473b018b70455e34a75, NAME => 'cow_prod:PROD_COW_ADVERTISER,1025301,1425422180311.873c9a5fbb76a473b018b70455e34a75.', STARTKEY => '1025301', ENDKEY => '36578'}
873c9a5fbb76a473b018b70455e34a75.
cow_prod:PROD_COW_ADVERTISER,1025301,1425422180311. column=info:server, timestamp=1453326273233, value=gsbl840n05.blue.ygrid.yahoo.com:50511
873c9a5fbb76a473b018b70455e34a75.
cow_prod:PROD_COW_ADVERTISER,36578,1425422180311.d8 column=info:regioninfo, timestamp=1425422180560, value={ENCODED => d8e0bbbc1f7d309fe267d0f253293717, NAME => 'cow_prod:PROD_COW_ADVERTISER,36578,1425422180311.d8e0bbbc1f7d309fe267d0f253293717.', STARTKEY => '36578', ENDKEY => '65986'}
e0bbbc1f7d309fe267d0f253293717.
cow_prod:PROD_COW_ADVERTISER,36578,1425422180311.d8 column=info:server, timestamp=1453431722564, value=gsbl840n19.blue.ygrid.yahoo.com:50511
e0bbbc1f7d309fe267d0f253293717.
```

# HBASE META Table

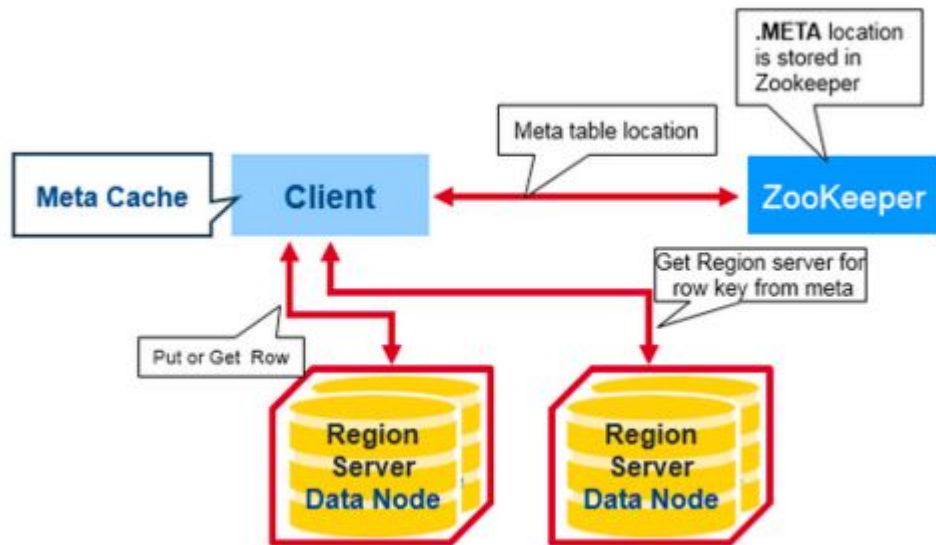


# Client Region Lookups (Previous 0.96 Version)



- Client caches this information along with the META table location

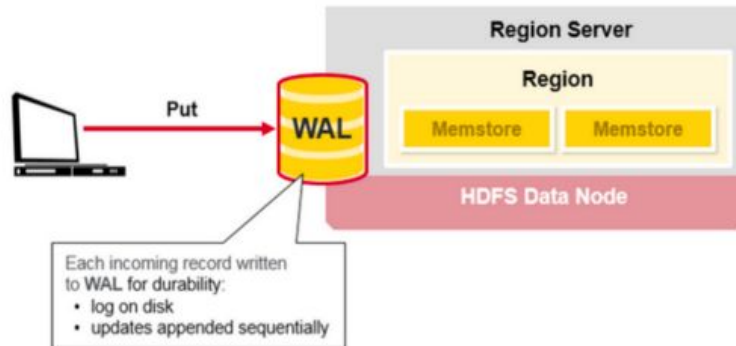
# Region Lookup



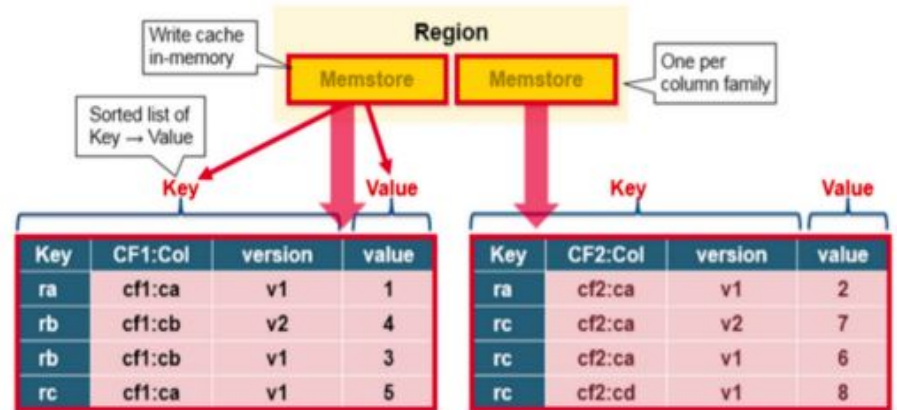
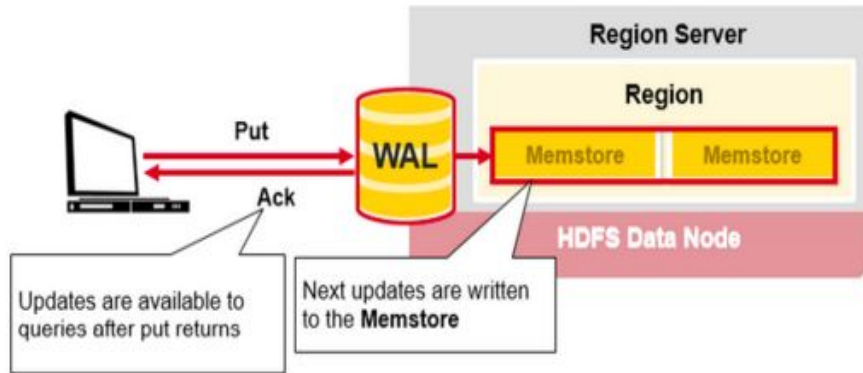


# HBASE Write Path (1)

- Write the data to the write-ahead log
- Edits are appended to the end of the WAL file that is stored on disk
- The WAL is used to recover not-yet-persisted data in case a server crashes.

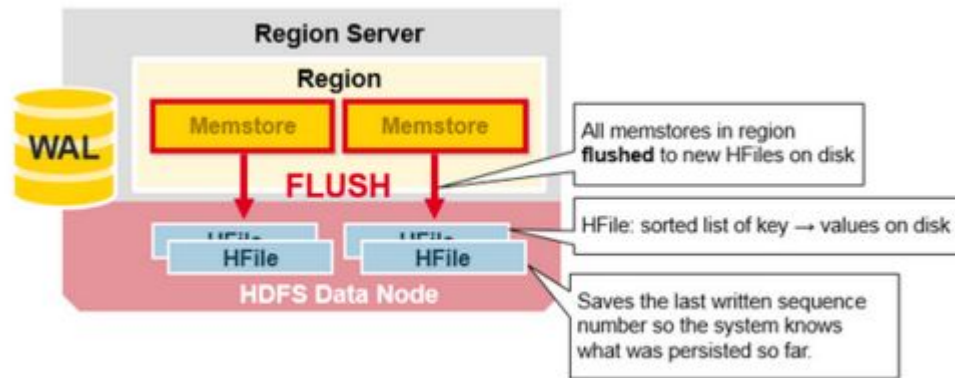


## HBASE Write Path (2)



- Once the data is written to the WAL, it is placed in the MemStore
- The MemStore stores updates in memory as sorted KeyValues
- There is one MemStore per column family
- The updates are sorted per column family

# HBASE Region Flush



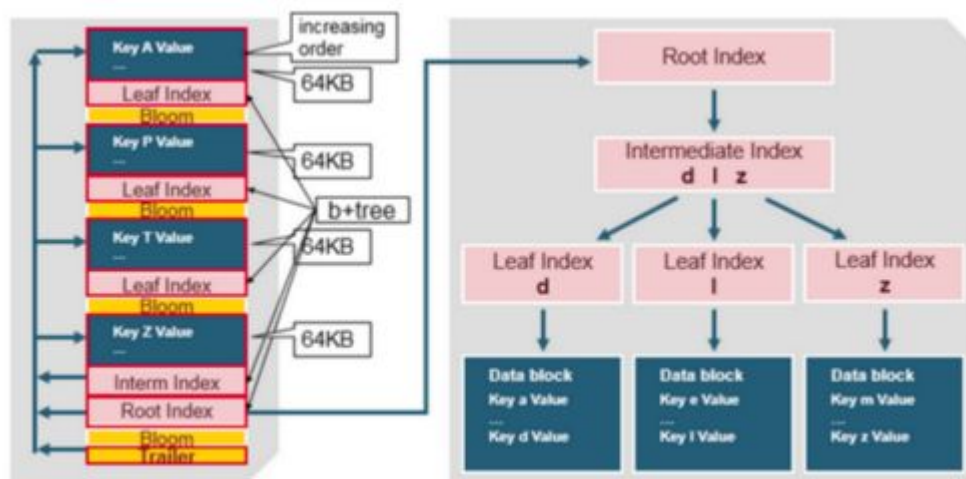
- When the MemStore accumulates enough data, the entire sorted set is written to a new HFile in HDFS
- HBase uses multiple HFiles per column family
  - Contains actual key-value instances
  - Created over time as KeyValue edits sorted in the MemStores are flushed as files to disk
- Records from a single column family might be split across multiple HFiles

# HBASE HFile Structure

- The HBase Key consists of
  - the row key
  - column family
  - column qualifier
  - timestamp and a type
- The data blocks contain the actual key/values in sorted key order

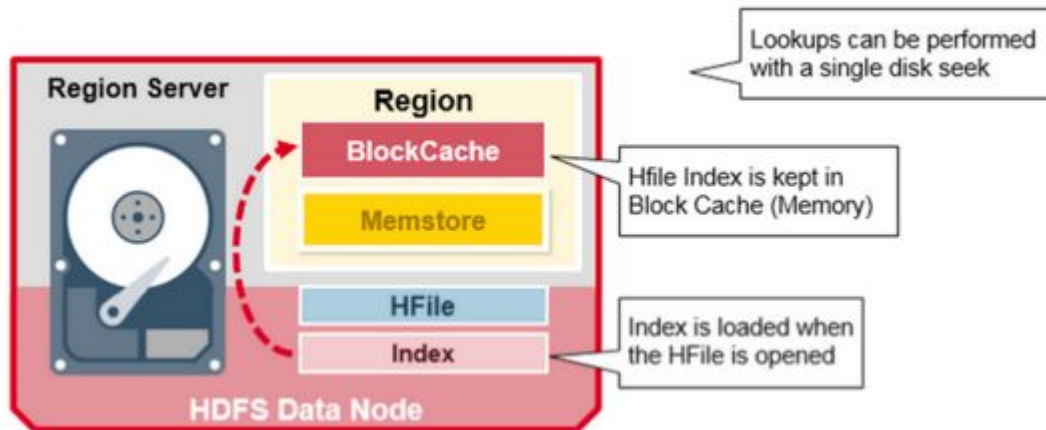
Row Length <i>short</i>	Row Key <i>byte[]</i>	Family Length <i>byte</i>	Column Family <i>byte[]</i>	Column Qualifier <i>byte[]</i>	Timestamp <i>long</i>	Key Type <i>byte</i>
----------------------------	--------------------------	------------------------------	--------------------------------	-----------------------------------	--------------------------	-------------------------

- Contains multi-level index
  - Indexes point by row key to the key value data in 64KB “blocks”
  - Each block has its own leaf-index
  - The last key of each block is put in the intermediate index
  - The root index points to the intermediate index



# Block Cache

- LRU cache for reads
  - Keeps the frequently accessed data from the HFiles in the memory
  - Whenever a read request arrives, the block cache is first checked for the relevant row
  - If it is not found, the HFiles on the disk are then checked for the same



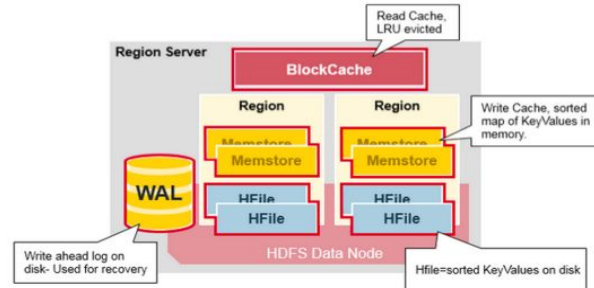
# MemStore Vs BlockCache

- **MemStore**
  - Accumulates data edits as they received, buffering them in memory
  - Important for accessing recent edits, otherwise they have to read from WAL
- **Block Cache**
  - Keeps data blocks resident in memory after they are read
  - When data block is read from HDFS, it is cached in block cache
  - 4 types of blocks - DATA, META, INDEX and BLOOM
  - Data Blocks - Contains data
  - Index Blocks - Provide an index over the cells contained in DATA blocks
  - BLOOM blocks contain a bloom filter over same data (more on bloom filters later)
- **HFile Format (Recap)**



# More on Block Cache

- A single instance for region server
  - All regions hosted by region server shares same Block Cache
  - Multiple implementations - LruBlockCache, SlabCache, BucketCache etc
- LruBlockCache
  - Data blocks are cached in JVM heap using this implementation
  - Subdivided into three areas: single-access (25%), multi-access (50%), and in-memory (25%)
  - A block initially read from HDFS is populated in the single-access area
  - Consecutive accesses promote that block into the multi-access area
  - The in-memory area is reserved for blocks loaded from column families flagged “inmemory”
  - Old blocks are evicted to make room for new blocks using LRU algorithm
    - Background eviction thread



```
hbase> alter 'myTable', 'myCF', CONFIGURATION => {IN_MEMORY => 'true'}
```

# Block Cache Stats

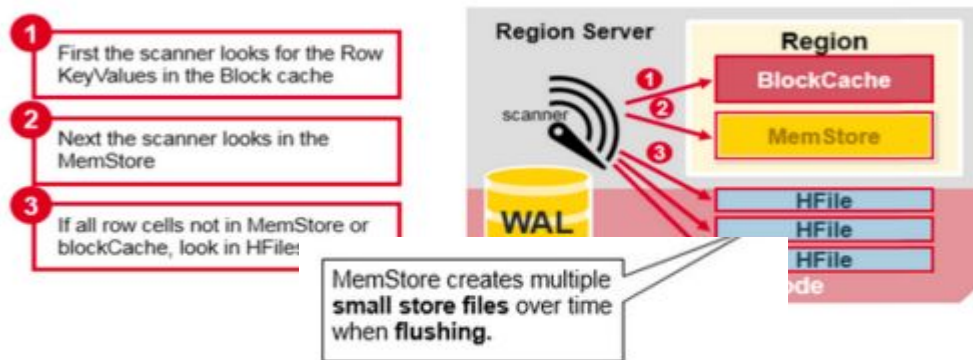
- Example: [http://gsbl877n03.blue.ygrid.yahoo.com:50501/rs-status#bc\\_stats](http://gsbl877n03.blue.ygrid.yahoo.com:50501/rs-status#bc_stats)

<div><div>APACHE HBASE</div><div><a href="#">Home</a><a href="#">Local Logs</a><a href="#">Log Level</a><a href="#">Debug Dump</a><a href="#">Metrics Dump</a><a href="#">HBase Configuration</a></div></div>		
<h2>Block Cache</h2> <div><a href="#">Base Info</a><a href="#">Config</a><a href="#">Stats</a><a href="#">L1</a><a href="#">L2</a></div>		
Attribute	Value	Description
Size	28.3 G	Total size of Block Cache (bytes)
Free	1.4 G	Free space in Block Cache (bytes)
Count	428,788	Number of blocks in Block Cache
Evicted	3,757,716	Number of blocks evicted
Evictions	55,627	Number of times an eviction occurred
Hits	804,523,601	Number requests that were cache hits
Hits Caching	777,819,069	Cache hit block requests but only requests set to use Block Cache
Misses	486,122,967	Number of requests that were cache misses
Misses Caching	486,122,967	Block requests that were cache misses but only requests set to use Block Cache
Hit Ratio	62.33%	Hit Count divided by total requests count



# HBASE Read Merge

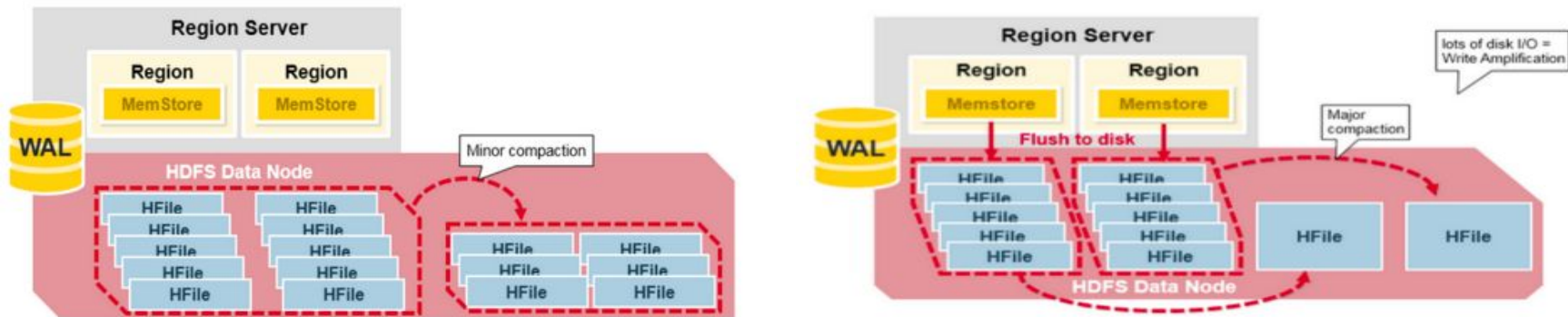
- Read merges Key Values from the Block Cache, MemStore, and HFiles in the following steps



- If the scanner does not find all of the row cells in the MemStore and Block Cache, then HBase will use the Block Cache indexes and bloom filters to load HFiles into memory, which may contain the target row cells.
- There may be many HFiles per MemStore, which means for a read, multiple files may have to be examined, which can affect the performance

# HBASE Compaction

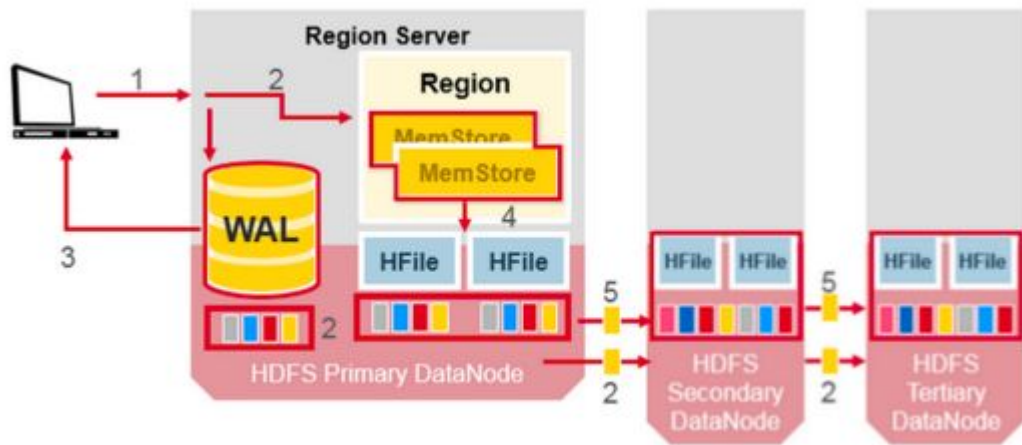
- Minor compaction reduces the number of storage files by rewriting smaller files into fewer but larger ones, performing a merge sort



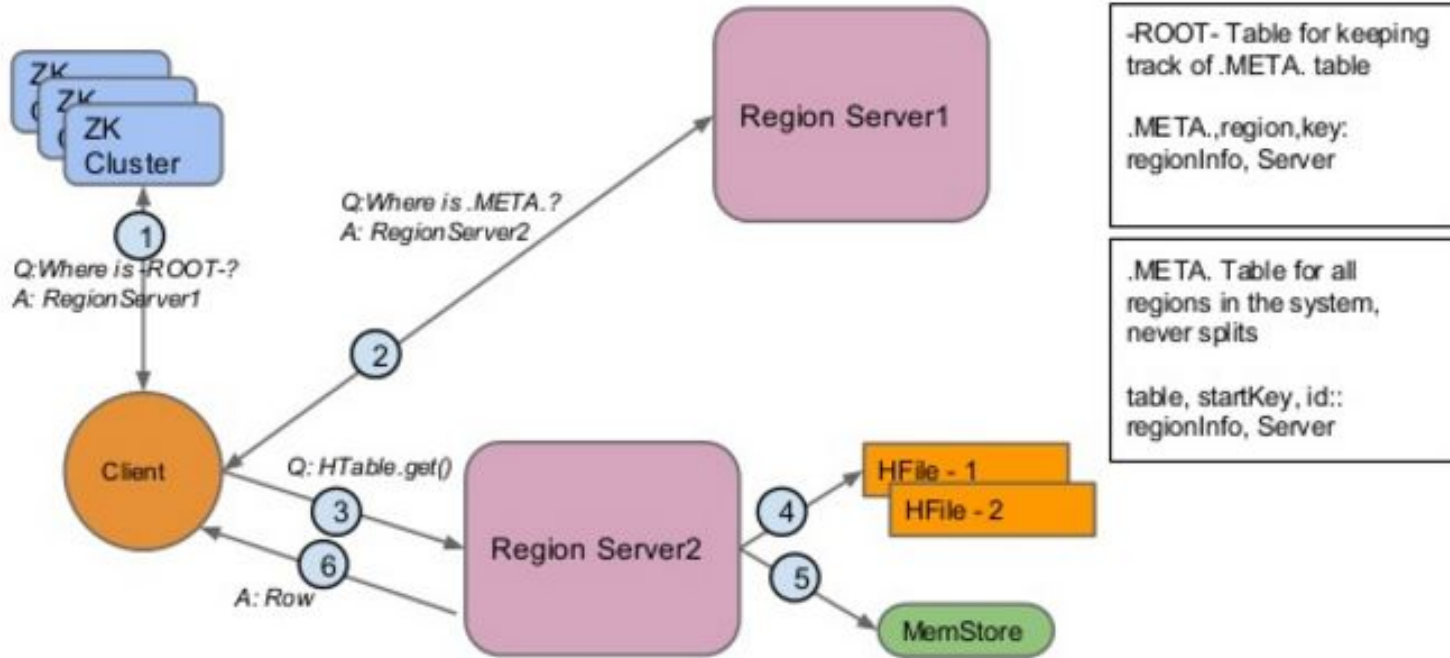
- Major compaction merges and rewrites all the HFiles in a region to one HFile per column family, and in the process, drops deleted or expired cells.

# Data Replication

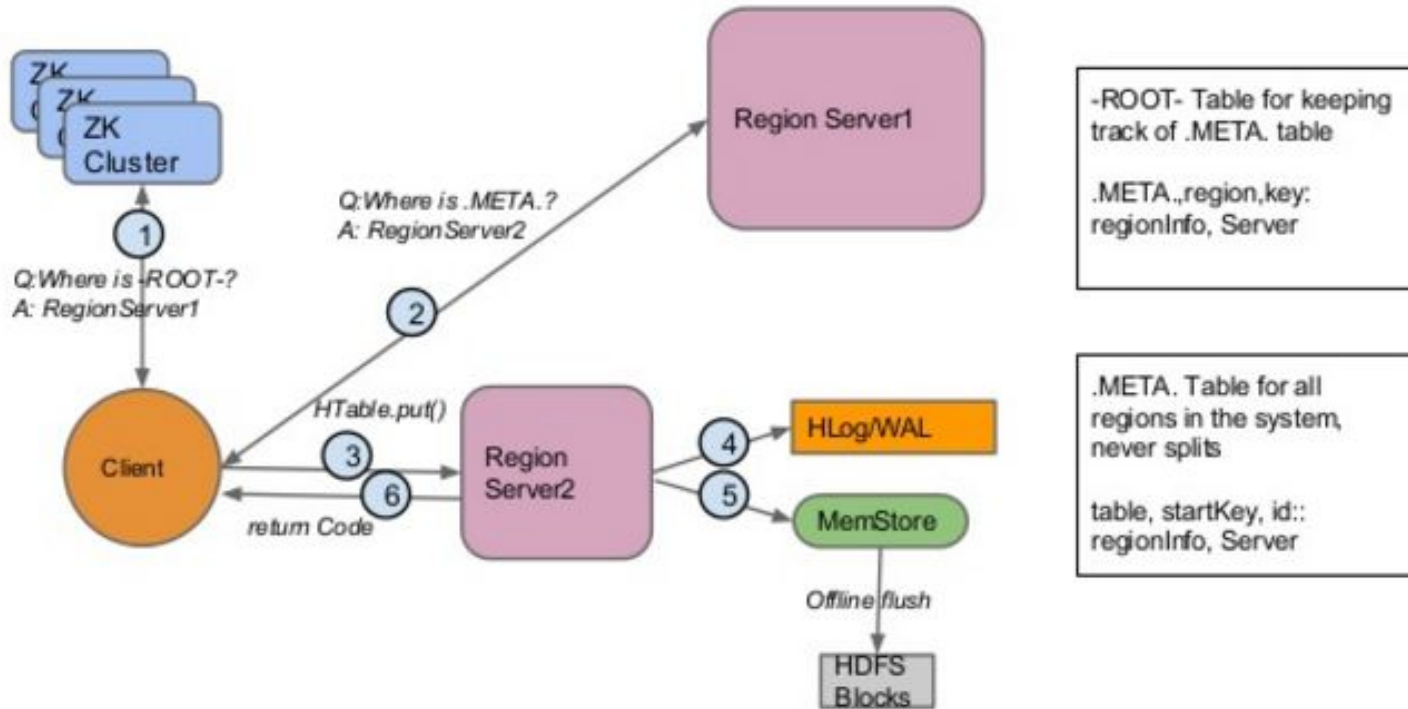
- HBASE relies on HDFS to replicate the WAL and HFile blocks



# Read Path - Putting All things together



# Write Path - Putting All Things Together



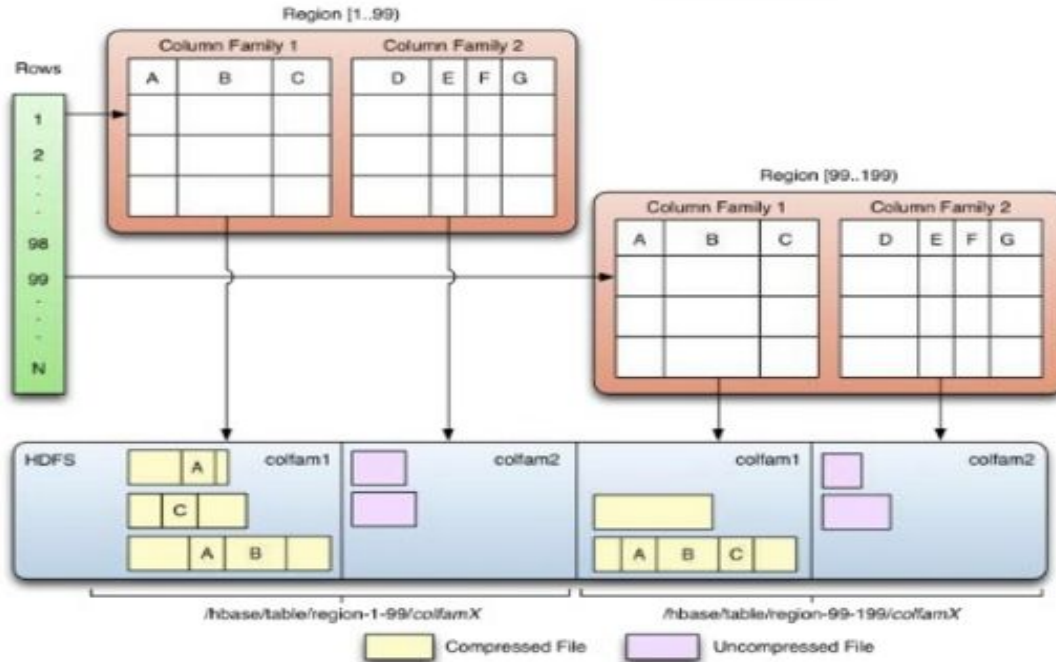
# Region Assignment

- Every region from every table is assigned to a RegionServer
- The Master is responsible for assignment
  - Noticing if region servers go down
  - When machines fail, move regions from affected machines to others
  - When regions split, move regions to balance cluster

# Stores And StoreFiles

- Stores - Total number of stores - One store for column family
  - Each region server may be hosting multiple regions (belonging to multiple tables)
  - Each table can contain multiple column families
- StoreFiles - Total number of Store Files
  - There can be multiple HFiles per store (multiple flushes from memstore)
- Large ratio between number of stores and store files may indicate it's time to compact more frequently - (Read Merge)
- All hbase files are stored in hbase.rootdir
  - Separate directories for each table
  - Each table contains multiple regions ...

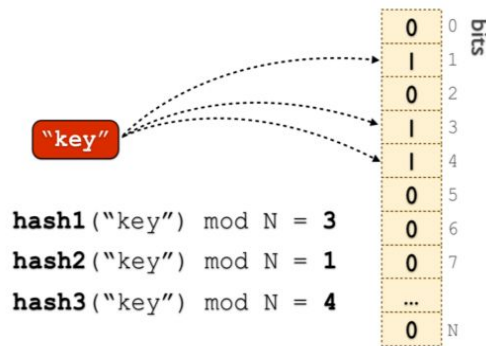
# HBase data locations





# Bloom Filters

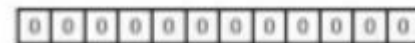
- Stored in the metadata of each HFile
  - Bloom Filters are generated when HFile is persisted
  - Bloom Filter is loaded into memory to determine if a given key is in that store file
  - HBase only has a block index - coarse grained (start and end key range)
  - If the key is actually present can only be determined by loading that block and scanning it
  - Bloom Filters allows check on row or row+column level
  - Can skip files not containing requested details
- Time range bound reads can skip store files
- What is Bloom Filter?
  - Space efficient probabilistic data structure
  - Used to test whether an element is member of a set or not
  - Can return false positive but definitely not a false negative



# Bloom Filter - Crash Course

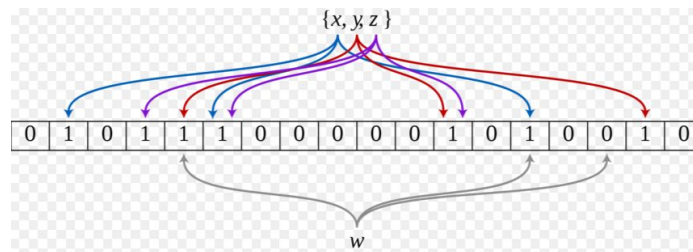
- Data Structure

- initialization: a bit array of  $m$  bits, all set to 0
- Add an element
  - $K$  hash functions to get  $K$  array positions
  - set the bits at all these positions to 1



- Query an element (test whether it's in the set)

- $K$  hash function to get  $K$  array positions
- if any position are zero, not in the set
- if all are 1, it may exist (can be false positive)



# HBASE Shell

- Shell is based on JRuby
  - Interactive Ruby Shell (IRB) - Just a ruby script
  - `${HBASE_HOME}/bin/hirb.rb` passed to JRuby class `org.jruby.Main` (see `${HBASE_HOME}/bin/shell` )
  - Enter Ruby commands and get an immediate response
  - HBase ships with Ruby scripts that extend the IRB with specific commands
- Ruby hashes per properties
  - `{'key1' => 'value1', 'key2' => 'value2', ...}`
  - `hbase(main):001:0> create 'testtable', { NAME => 'colfam1', VERSIONS => 1, TTL => 2592000, BLOCKCACHE => true }`
- Data Manipulation Commands
  - put, get, delete, scan, count, truncate ...
- Data Definition Commands
  - create, alter, describe, disable, enable, drop etc ...
- Various other command groups (namespace, security ...)

# Client API

- Create a Configuration object
  - Add HBASE specific properties
  - HBaseConfiguration extends Hadoop's configuration class
  - Loads hbase-default.xml and hbase-site.xml config files
- Construct HTable
  - Provide Configuration object
  - Provide Table Name
- Perform operations
  - Put, get, scan, delete etc ...
- Close HTable instance
  - Flushes all internal buffers
  - Releases all the resources

# HTable - org.apache.hadoop.hbase.client.HTable

- Client interface to a single HBase Table
- Exposes CRUD operations
- Operations that change data are atomic on per-row-basis
  - No transaction for multiple rows or tables
  - 100% consistency per row - client will either read/write the entire row OR have to wait
  - scans catalog hbase:meta table

# Put Operation

- Put is a save operation for a single row
- Must provide a row id to the constructor
  - Row id is raw bytes
  - Convert id to bytes
  - `Put put1 = new Put(Bytes.toBytes("row1"))`
  - Optionally can provide cell's timestamp
- Add Columns to save to Put instance
  - `put.add(family, column, value)`
  - `put.add(family, column, timestamp, value)`
  - Family, column and value are in raw binary
- It is client's responsibility to convert to binary data
- Provide initialized Put object to HTable

# Put Example

```
Configuration conf = HBaseConfiguration.create();  
  
HTable hTable = new HTable(conf, "Advertiser");  
  
Put put = new Put(toBytes("advertiser_1"));  
  
put.add(toBytes("meta"), toBytes("status"), toBytes("on"));  
  
hTable.put(put);  
  
hTable.close();
```

# Retrieving Data

- API
  - Get a single row by id
  - Get a set of rows by a set of row ids
  - Scan an entire table or a sub set of rows
    - To scan a portion of table provide start and stop row ids
    - row ids are ordered by raw byte comparison
- Only retrieve the data that you need
  - If not specified then an entire row is retrieved
  - Can narrow down by family, columns, time range, max versions ...
  - Family and column name parameters are in raw bytes
- Result class
  - Allows to access everything returned
  - `Result.getRow()` - get row's id
  - `Result.getFamily(family, column)` - get a value for chosen cell etc



# Get Operation

```
Configuration conf = HBaseConfiguration.create();
```

```
HTable hTable = new HTable(conf, "Advertiser");
```

```
Get get = new Get(toBytes("advertiser_1");
```

```
Result result = hTable.get(get);
```

```
print(result);
```

```
get.addColumn(toBytes("meta"), toBytes("status"));
```

```
result = hTable.get(get); print(result);
```

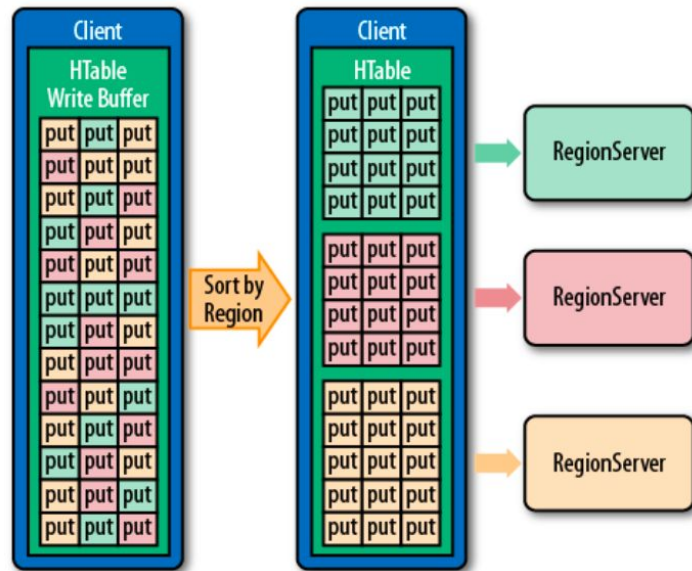
```
hTable.close();
```

# Delete Operation

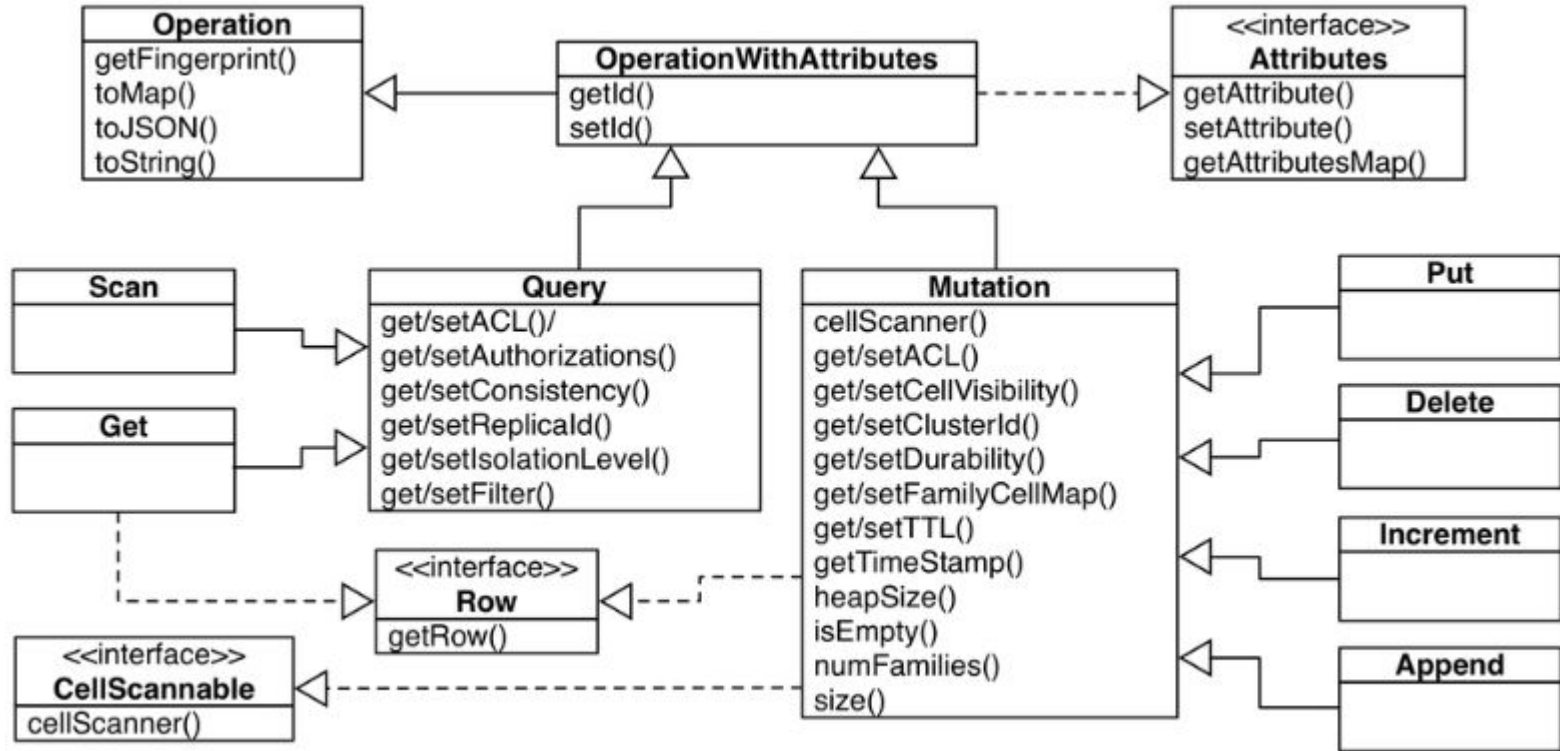
- Construct a Delete instance
  - Delete(byte[] row) - Provide a row id to delete/modify
  - Optionally narrow down deletes
    - Delete a subset of a row by narrowing down
    - Just delete a particular column (deleteColumns)
    - Entire column family (deleteFamily)
  - Most of methods are also overloaded to take timestamp
    - Deletes everything on or before the timestamp provided

# Client side write buffer

- Put operation is RPC transferring data from client to server
- Client side write buffer to batch all put operations
  - They are sent in one RPC call
  - Enable buffering by `hTable.setAutoFlush(false)`
  - Finally flush commits by `flushCommits()`
  - Buffered put instances can spawn multiple rows
  - Client is smart enough to batch these updates accordingly and send them to appropriate region servers



# Class Hierarchy (HBASE 1.0)



# Scans

- Similar to familiar cursors in other database systems
  - `Scan(byte[] startRow, byte[] stopRow)`
  - Startrow is inclusive and stoprow is exclusive `[startrow, stoprow)`
  - Can narrow down data adding column family and columns
  - Can also specify time range and number of versions to limit data
  - Once scan instance is configured, call `getScanner` on `HTable` instance
  - Remember - Row ids are stored in sequence
- **ResultScanner**
  - Wrapping the `Result` instance for each row into an iterator
  - `Result next()`, `Result[] next(int nbRows)` etc.

# Scan - Example

```
Scan scan1 = new Scan(); ❶
ResultScanner scanner1 = table.getScanner(scan1); ❷
for (Result res : scanner1) {
    System.out.println(res); ❸
}
scanner1.close(); ❹

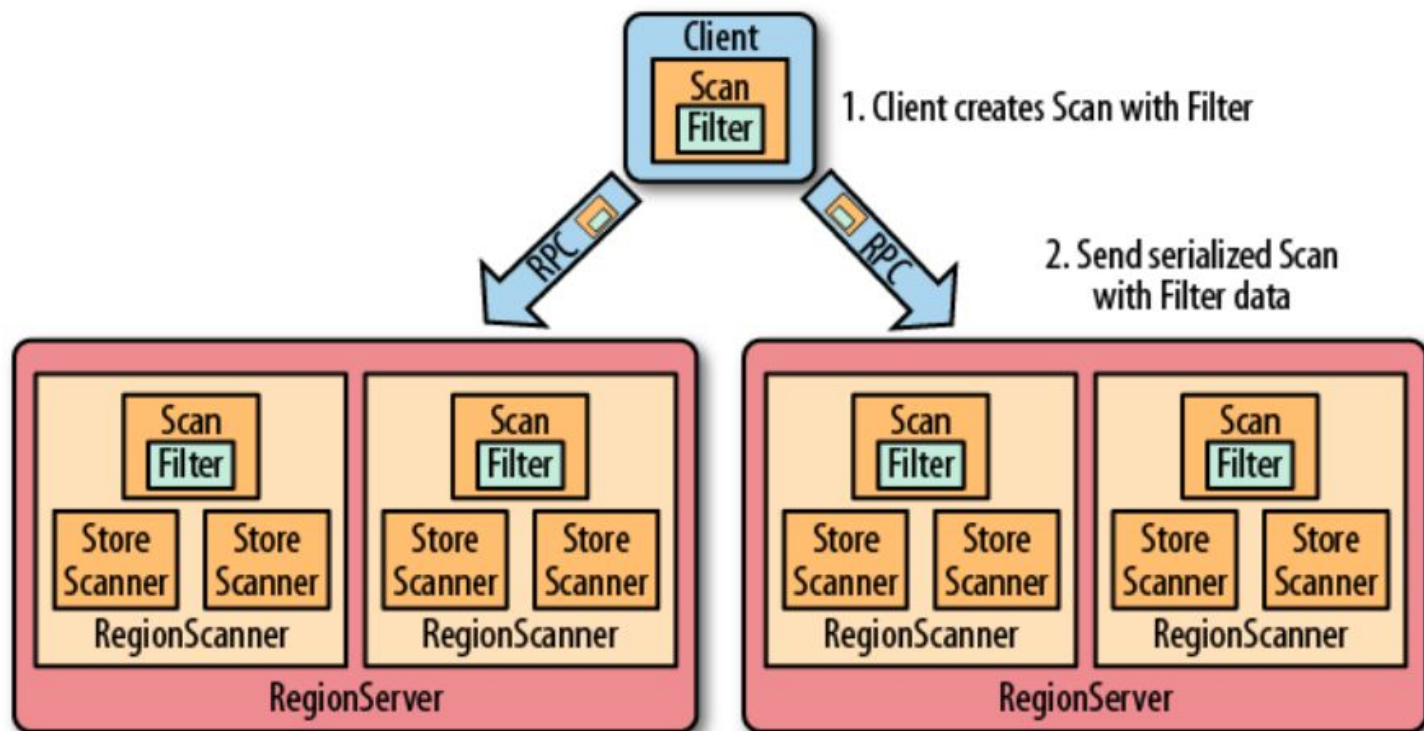
Scan scan2 = new Scan();
scan2.addFamily(Bytes.toBytes("colfam1")); ❺
ResultScanner scanner2 = table.getScanner(scan2);
for (Result res : scanner2) {
    System.out.println(res);
}
scanner2.close();

Scan scan3 = new Scan();
scan3.addColumn(Bytes.toBytes("colfam1"), Bytes.toBytes("col-5")).
    addColumn(Bytes.toBytes("colfam2"), Bytes.toBytes("col-33")); ❻
scan3.setStartRow(Bytes.toBytes("row-10"));
scan3.setStopRow(Bytes.toBytes("row-20"));
ResultScanner scanner3 = table.getScanner(scan3);
for (Result res : scanner3) {
    System.out.println(res);
}
scanner3.close();
```

# Scanner Caching And Batching

- Use scanner caching to fetch more than a single row per RPC
- Each call to next() would be a separate RPC for every row
  - If not configured correctly
  - Get more rows per RPC call instead of one row
  - Cluster wide configuration - `hbase.client.scanner.caching`
  - Set caching at scanner instance level (like fetching 1000 rows - `setCaching(1000)`)
  - Can set large value but make sure that client won't run out of heap memory
- Batching
  - A single row with lots of columns may not fit in memory
  - Batching allows to page through columns on per row basis
  - To limit the number of columns if rows have a large number of columns
  - `setBatch()` helps to set the number of columns to be returned in one call

# Filters





## Filters (2)

- Filters are actually applied on server side - Predicate push down
- Moving filtering from client to server to reduce network traffic
- Can specify Filter in Get and Scan to specify predicates on data
- More fine grained control over what is returned to the client
- Many filters are based on row key, column family or column qualifier
- Example:
  - RowFilter - ability to filter data based on row keys
  - `Filter filter1 = new RowFilter(CompareFilter.CompareOp.LESS_OR_EQUAL, new BinaryComparator(Bytes.toBytes("row-22")));`
  - `scan.setFilter(filter1);`

# Comparison Filters

- Based on CompareFilter Class
- Takes operator that defines how the comparison is performed
  - LESS, LESS\_OR\_EQUAL, EQUAL ...
- Need a comparator to do actual check
  - Binary Comparator, BinaryPrefixComparator, SubStringComparator etc
- Available Comparison Filters
  - Row Filter - Based on row keys comparisons
  - Family Filter - Based on Column Family names
  - Qualifier Filter - Based on Column names
  - Value Filter - Based on actual value of a column

# Example Filters - HBase Shell

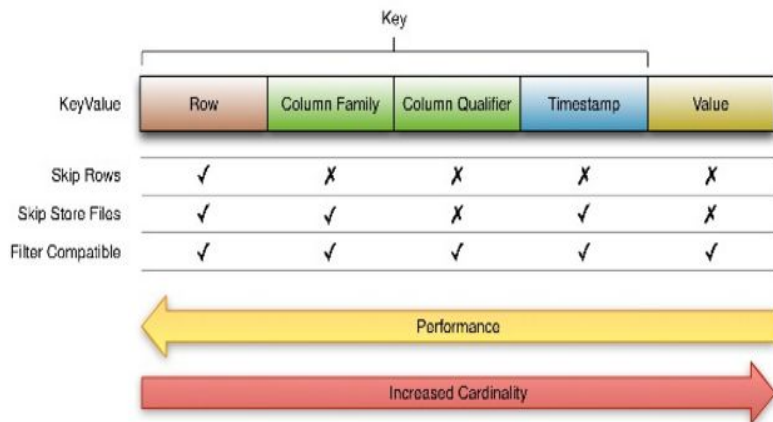
```
hbase(main):001:0> show_filters
Documentation on filters mentioned b
DependentColumnFilter
KeyOnlyFilter
ColumnCountGetFilter
SingleColumnValueFilter
PrefixFilter
SingleColumnValueExcludeFilter
FirstKeyOnlyFilter
ColumnRangeFilter
TimestampsFilter
FamilyFilter
QualifierFilter
ColumnPrefixFilter
RowFilter
MultipleColumnPrefixFilter
InclusiveStopFilter
PageFilter
ValueFilter
ColumnPaginationFilter
```

- hbase(main):011:0> scan 'cow\_prod:PROD\_COW\_ADVERTISER', {FILTER=>"PrefixFilter('10')", LIMIT=>5}
- hbase(main):015:0> scan 'cow\_prod:PROD\_COW\_ADVERTISER', {FILTER=>"KeyOnlyFilter()", LIMIT=>5}
- hbase(main):016:0> scan 'cow\_prod:PROD\_COW\_CAMPAIGN', {FILTER=>"PrefixFilter('10,')", LIMIT=>5}
- hbase(main):018:0> scan 'cow\_prod:PROD\_COW\_CAMPAIGN', {FILTER=>"ValueFilter(=, 'substring:VISIT\_WEB')", LIMIT=>5}

# KeyValue Class

- KeyValue class is the heart of data storage in HBase
  - KeyValue wraps a byte array
  - KeyValue Format - [keylength, valuelength, key, value]
- Key is further decomposed as:
  - rowlength
  - row (i.e., the rowkey)
  - columnfamilylength
  - columnfamily
  - columnqualifier
  - timestamp
  - keytype (e.g., Put, Delete, DeleteColumn, DeleteFamily)

# Key Cardinality



- Time range bound reads can skip store files
  - Bloom Filters can also help
- Selecting column families reduces the amount of data to be scanned
- Pure value based filtering is a full table scan
- Partial key scans - Examples
  - Campaigns for a given advertiser
  - Adgroups for a given advertiser, campaign

# HBASE Key Points

- Inserts are done in write-ahead log first
- Data is stored in memory and flushed to disk on regular intervals (or size)
- Small flushes are merged in the background to keep number of files small
- Reads memory stores first and then disk based files second
- Deletes are handled with “tombstone” markers
- Atomicity on row level no matter how many columns
- In-memory block cache to optimize reads on subsequent columns and rows
- Region splits - triggered by configured max file size of any store file

# HBASE Schema Design

- HBASE is a sparse, distributed, persistent multidimensional sorted map, which is indexed by a row key, column key, and a timestamp (verbose :-))
- Using row key (or key ranges) is the most efficient way to retrieve data
- HBase stores cells individually - Great for sparse data
  - Row key, Column Family (CF) name, Column Qualifier (CQ) stored with each cell
  - Keep CF and CQ names short
  - Serialize and store many values into single cell
- Column Families allow for separation of data
  - Segregate information based on access pattern
  - Use only a few column families (stored in separate HFiles)

# Tall vs Wide Tables

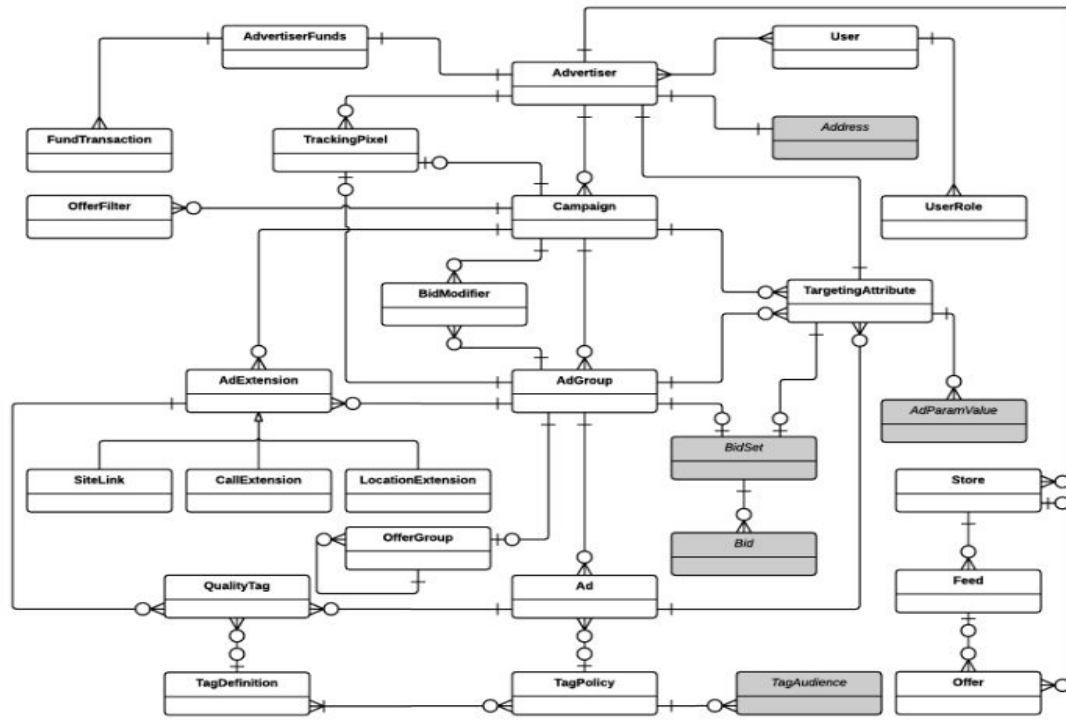
- Wide Tables - Each row has a lots of columns
  - Allows atomicity at row level
- Tall Tables - Put more details into the row key
  - Make use of partial key scans
  - Tall tables can potentially allow you faster and simpler operations
  - Trade off atomicity



# COW Schema

- COW - (Curveball Object Warehouse)
- Maintains Gemini (Native and Search Demand) in HBASE for GRID jobs
- [https://docs.google.com/document/d/1GZuNr3Sb2Q66DCwrc2SJLOuLJ7alyhX\\_qZoM77QSIhE/edit#](https://docs.google.com/document/d/1GZuNr3Sb2Q66DCwrc2SJLOuLJ7alyhX_qZoM77QSIhE/edit#)

# COW - Quick View



hbase(main):001:0>

list\_namespace\_tables 'cow\_prod'

- PROD\_COW\_ADVERTISER
- PROD\_COW\_CAMPAIGN
- PROD\_COW\_ADGROUP
- PROD\_COW\_AD
- PROD\_COW\_ADEXTENSION
- PROD\_COW\_TARGETING\_ATTRIBUTE
- PROD\_COW\_OFFER
- PROD\_COW\_PARENTMAPPING

# COW Tables - Key Design

PROD_COW_ADVERTISER	Advertiser ID		
PROD_COW_CAMPAIGN	Advertiser ID, Campaign ID		
PROD_COW_ADGROUP	Advertiser ID, Campaign ID, AdGroup ID		
PROD_COW_AD	Advertiser ID, Campaign ID, AdGroup ID, Ad ID		
PROD_COW_ADEXTENSION	Advertiser ID, [campaign adgroup], Parent ID, [AdExtensionType], AdExtension ID		
PROD_COW_TARGETING_ATTRIBUTE	Advertiser ID, [campaign adgroup], Parent ID, [Targeting Type], [Include Exclude], Targeting Attribute ID		
PROD_COW_OFFER	Advertiser ID, Offer ID		
PROD_COW_DOMAINOBJECTS	[Domain Object Type], Object ID		
PROD_COW_PARENTMAPPING	[Domain Object Type], Object ID		Index

# COW Tables - Key Design

- Hierarchical Schema
  - Child table must have a foreign key to its parent table as a prefix of its primary key
  - All child table rows corresponding to a root row are clustered together
  - Make use of partial key scans (prefix key scans)
- Only one column family and one column qualifier (d:m)
  - Entire object is encoded in JSON format (verbose, not space efficient)

```
column=d:m, timestamp=1425447797805, value={"domainObjectType":"campaign","domainObject":{"language":"en", "priority":null,"id":1,"startTime":1367294400000,"endTime":1372651199000,"status":"REJECT_INTERNAL","advertiserId":1,"lastUpdatedByUser":"escobedf","sourceType":null,"createdDate":1364774400000,"budgetType":"UNLIMITED","spendCapType":"DAILY","budget":100,"trackingPixelId":null,"lastUpdateDate":1399616805522,"objective":"VISIT_WEB","mobileAppParam":null,"campaignName":"Centerfield Media Holdings, LLC2","createdByUser":"SY STEM","startDateStr":"2013-04-30","endDateStr":"2013-06-30","platform":null,"rmxLineId":null,"rmxIOId":null,"spendCap":20,"effectivePacingType":null,"effectiveDailySpendCap":null,"effectiveStatus":"REJECT_INTERNAL","budgetExpiredOn":null,"rmxPixelId":null,"adProducts":[{"productName":"StreamAd","status":null}], "liveDate":null,"defaultLandingUrl":null,"cpi":null,"pacingType":"DEFAULT","internalStatus":null,"lastSyncedDate":null,"sourceID":null,"cpc":0.05},"sequenceId":4507962874,"domainObjectId":1}
```

# Motivation - F1: A Distributed SQL Database that scales (Google Paper)

	Traditional Relational	Clustered Hierarchical
Logical Schema	<p>Customer(<u>CustomerId</u>, ...)</p> <p>Campaign(<u>CampaignId</u>, CustomerId, ...)</p> <p>AdGroup(<u>AdGroupId</u>, CampaignId, ...)</p> <p>Foreign key references only the parent record.</p>	<p>Customer(<u>CustomerId</u>, ...)</p> <p>↳ Campaign(<u>CustomerId</u>, <u>CampaignId</u>, ...)</p> <p>↳ AdGroup(<u>CustomerId</u>, <u>CampaignId</u>, <u>AdGroupId</u>, ...)</p> <p>Primary key includes foreign keys that reference all ancestor rows.</p>
Physical Layout	<p>Joining related data often requires reads spanning multiple machines.</p> <div> <div>Customer(1,...)</div> <div>Customer(2,...)</div> </div> <div> <div>Campaign(3,1,...)</div> <div>Campaign(4,1,...)</div> <div>Campaign(5,2,...)</div> </div> <div> <div>AdGroup(6,3,...)</div> <div>AdGroup(7,3,...)</div> <div>AdGroup(8,4,...)</div> <div>AdGroup(9,5,...)</div> </div>	<div> <div>Customer(1,...)</div> <div>Campaign(1,3,...)</div> <div>AdGroup (1,3,6,...)</div> <div>AdGroup (1,3,7,...)</div> <div>Campaign(1,4,...)</div> <div>AdGroup (1,4,8,...)</div> </div> <p>Physical data partition boundaries occur between root rows.</p> <div> <div>Customer(2,...)</div> <div>Campaign(2,5,...)</div> <div>AdGroup (2,5,9,...)</div> </div> <p>Related data is clustered for fast common-case join processing.</p>

# PARENT MAPPING TABLE

- Given a [domain object type, ID], parent mapping table provides hierarchy of keys from root to domain object - which can be used to retrieve object from corresponding table. Here are examples:

ad, [ad ID]	Advertiser ID, Campaign ID, AdGroup ID, Ad ID
campaign, [campaign ID]	Advertiser ID, Campaign ID
adGroup, [adGroup ID]	Advertiser ID, Campaign ID, AdGroup ID
targetingAttribute, [TA ID]	Advertiser ID, [campaign/adGroup], parent id, [Targeting Type], [Include/Exclude], Targeting Attribute ID

# Extracts - TS range scans

- Find explicitly changed domain objects between timestamp ranges
  - Domain Objects - Campaigns, AdGroups, Ads, TargetingAttributes
  - Extracts Delta Workflow, Bootstrap (with mintimestamp as 0)

```
DomainObjectKeyVal = LOAD 'hbase://$table' USING org.apache.pig.backend.hadoop.hbase.HBaseStorage('d:m', '-loadKey true -minTimestamp $minHWM -maxTimestamp $maxHWM -caching 1000');
```

- Time range bound reads can skip store files
  - Remember about ReadMerge
  - Read merges Key Values from the Block Cache, MemStore, and HFiles
  - Key may be existing in multiple HFiles (due to multiple changes)

# Extracts - Advertiser Fanouts

- Advertiser Fanout
  - When an advertiser status changes from “OFF” to “ON”, push all domain objects associated with advertiser to serving DataStores.
  - Domain Objects include - Campaigns, AdGroups, Ads, Targeting
    - FetchAdvertiserCamapigns - Prefix key scan from PROD\_COW\_CAMPAIGN)
    - FetchAdvertiserTargeting - Campaign/AdGroup Level (Geo, Device)
      - Prefix key scan PROD\_COW\_TARGETING\_ATTRIBUTE
    - FetchAdvertiserAdgroups - Prefix key scan from PROD\_COW\_ADGROUP
    - FetchAdvertiserAds - Prefix key scan from PROD\_COW\_AD
- [https://git.corp.yahoo.com/Curveball/falcon\\_extracts/blob/master/src/main/scripts/falcon/AdvertiserDataMacros.pig#L49](https://git.corp.yahoo.com/Curveball/falcon_extracts/blob/master/src/main/scripts/falcon/AdvertiserDataMacros.pig#L49)



# Extracts - Nested Lists

- Data Stores maintains a list of items in certain domain objects
  - For a given adgroup, it maintains the list of active ads in that adgroup

# Extracts - Query Server Inverted Index

- Query Server maintains inverted index of eligible candidates for a given bidded term
  - Bidded Term Text ==> List [Advertiser ID, Campaign ID, AdGroup ID, Bidded Term ID, Match Type]
  - Example: Car Insurance => All Bidded Terms bidding on “Car Insurance”
  - When new bidded term is added for “Car Insurance”, need to construct entire posting list containing previous bidded terms and new bidded term - no incremental updates or deletes
  - Data Stores can only accept full record - no updates
  - Extracts maintain serving index table - key based on hash(phrase\_canon\_text of bidded term)
- Wide or Tall table ?
  - Preferred tall table
  - Key contains additional information [Advertiser ID, Campaign ID, AdGroup ID, Bidded Term ID, Match Type]
  - Just prefix scan based on hash of bidded term text to get all bidded terms - very simple
  - If a new bidded term is added, just add another row for that bidded term text
  - If we were adopted wide table, some rows can be very large (deep market bidded terms)
    - multiple columns need to be added (one idea: bidded term id as column name)
    - may be a single column but contains protobuf/avro/json format of all bidded terms

# GPA Extracts - Offer Matching Index Tables

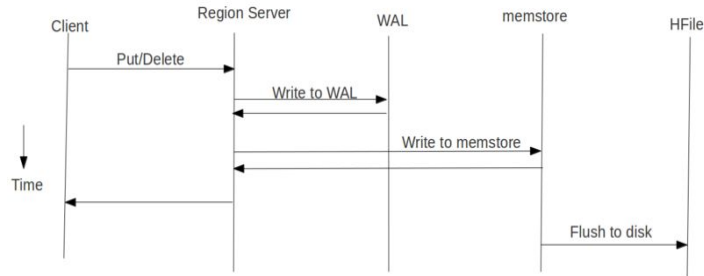
- Similar to Query Server inverted index tables
- All of these tables are tall tables
- `hbase(main):011:0> list_namespace_tables 'cb_gpa'`
  - `PROD_SERVING_OFFER_FILTER_INDEX`
    - Maintains offer to eligible campaigns based on offer attributes and campaign filters
  - `PROD_SERVING_OFFER_FILTER_REVERSE_INDEX`
    - Maintains campaign to eligible offers
  - `PROD_SERVING_OFFER_GROUP_INDEX`
    - Maintains offer group to eligible offers
  - `PROD_SERVING_OFFER_INDEX`
    - Maintains offer to eligible offer groups (adv id, campaign id, adgroup id, offer group id)
- Key contains all needed fields, no value - Just prefix key scans

# HBASE - ACID Semantics - Refresh

- HBASE honors ACID semantics per-row
  - Atomicity: All parts of transaction complete or none complete
  - Consistency: Only valid data written to database
  - Isolation: Parallel transactions do not impact each other's execution
  - Durability: Once transaction committed, it remains

# Locking in HBASE

- Write to Write-Ahead-Log (WAL)
- Update MemStore: Write each data cell [the (row, column) pair] to the memstore
- If there is no concurrency control over the writes, two writes to the same row can be inter-mixed (updating multiple columns)
  - results in in-consistent state of the row
- Simple solution could be
  - **Obtain Row Lock**
  - Write to Write-Ahead-Log
  - Update MemStore: write each cell to the memstore
  - **Release Row Lock**




# Row Atomicity

- Row is locked while memstore is getting updated
- Each row-update “creates” a new snapshot
- Each row-read sees exactly one snapshot

# Multi-Version Concurrency Control

- HBASE maintains ACID Semantics using MVCC
- Instead of overwriting state, create a new version of object with new timestamp (aka: memStoreTS)
- Reads never have to lock
- “memStoreTS” is not externally visible. It is different from external timestamp



memstoreTs	RowKey	fam1:col1	fam2:col2
t2	row1	val2	val2
t1	row1	val1	val1

# Read-Write Synchronization

- If no concurrency control for reads, read with writes can also result in inter-mixed results (multiple columns getting updated)
- Need some concurrency-control to deal with RW synchronization
- Simple solution would be obtain row lock similar to write path and release
  - Affects performance, both reads/writes have to obtain row locks
- HBASE uses Multiversion Concurrency Control (MVCC) to avoid requiring the reads to obtain row locks



# HBASE - Multiversion Concurrency Control

- Writes

- (w1) After acquiring the RowLock, each write operation is immediately assigned a write number
- (w2) Each data cell in the write stores its write number.
- (w3) A write operation completes by declaring it is finished with the write number.

- Reads

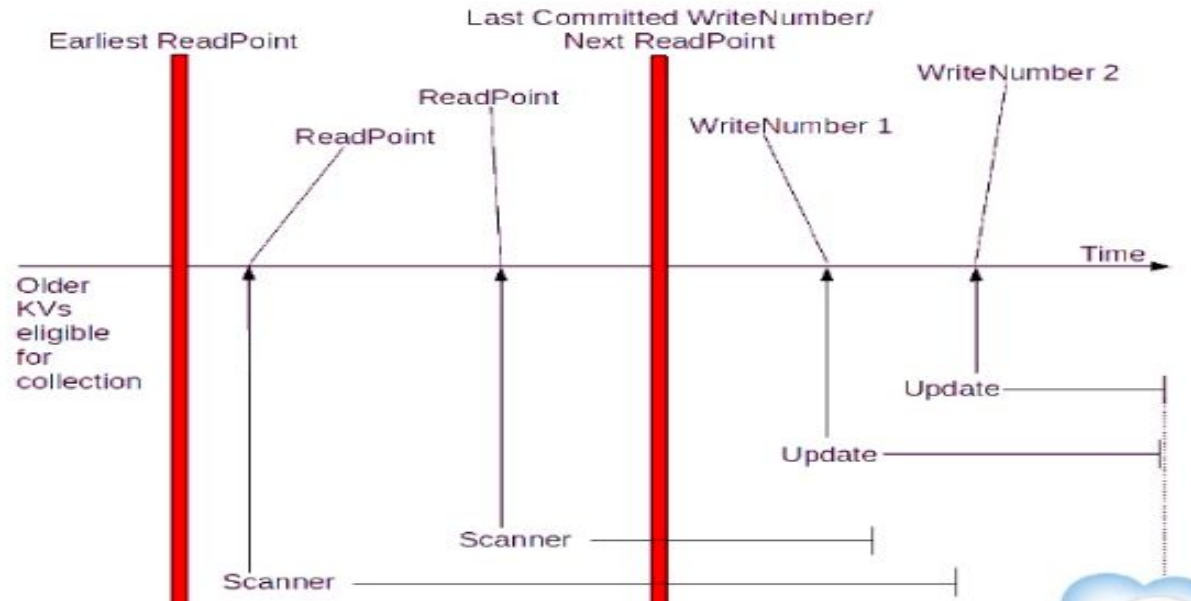
- (r1) Each read operation is first assigned a read timestamp, called a read point.
- (r2) The read point is assigned to be the highest integer such that all writes with write number  $\leq x$  have been completed.
- (r3) A read  $r$  for a certain (row, column) combination returns the data cell with the matching (row, column) whose write number is the largest value that is less than or equal to the read point of  $r$ .

- All transactions are committed serially

# Undo

- Undo happens in-memory only
- Changes are not visible until MVCC is rolled
- Changes are tagged with the write-number
- “Undo” removes KVs tagged with write-number

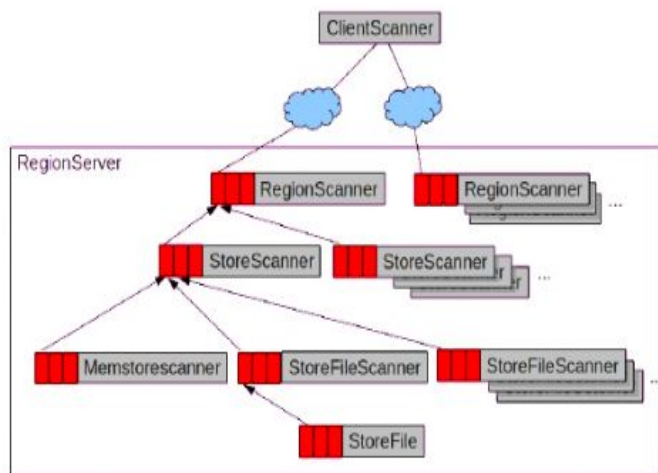
# Read - MVCC Example



# HBASE - Transactions Commit

- All transactions are committed serially
- HBase keeps a list of all unfinished transactions
- A transaction's commit is delayed until all prior transactions committed
- HBase can still make all changes immediately and concurrently, only the commits are serial
- Committing a transaction in HBase means setting the current ReadPoint to the transaction's WriteNumber, and hence make its changes visible to all new Scans

# Scanner - Internals



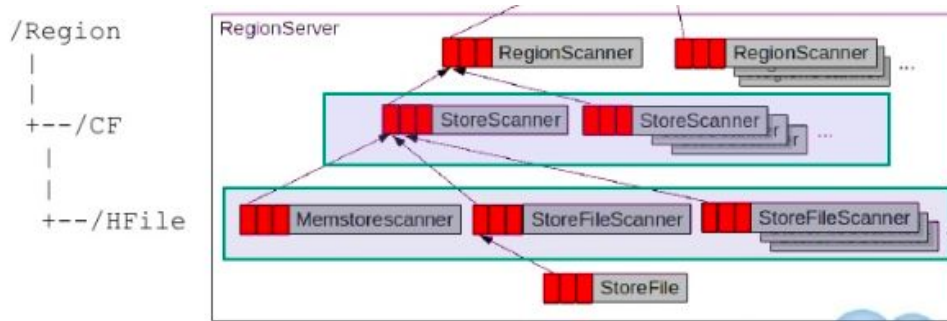
 = KeyValue

- HBase HDFS Directory hierarchy

```
/hbase
/-ROOT-
/.META.
/.logs
/<table1>
  /<region>
    [/.recovered_logs]
    /<column family>
      /<HFile>
      /...
      /...
    /<column family>
      /...
  /<table2>
  ...
```

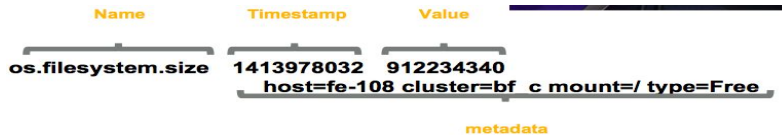
Storage is per CF

# Merging



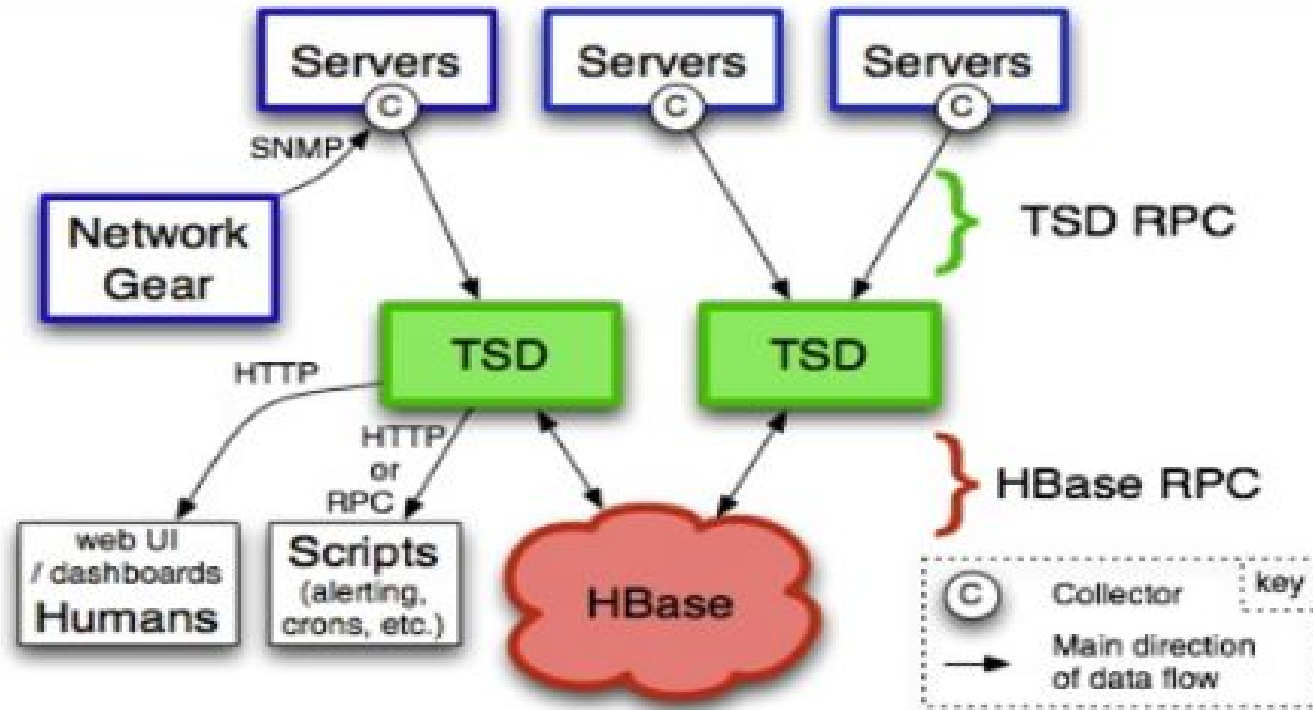
- Multiway Merge Sort

# Distributed Scalable Time Series Database



- OpenTSDB at Yahoo
  - Monitoring application performance and statistics
  - Time Series - Data points for a metric over time
  - Data Point - [Metric Name + TimeStamp + Any tags] => Value
- Key Design
  - Metric name first in row key for data locality + timestamp
    - Avoiding hot spots by keeping metric UID first in key
  - Store uniq ID for metric name instead of using metric name in the row key
    - Fixed width for key instead of variable length
  - Use a separate table to obtain metric name to ID mapping
  - Reduce number of rows by storing multiple consecutive data points in a single row
    - Wide table (Fewer rows, faster scan time)

# OpenTSDB Architecture





# YAMAS 2.0 Tables in HBASE

<http://luxblue-hb.blue.ygrid.yahoo.com:50500/master-status>

tsdb	Time series data - measurements and metadata	
tsdb-uid	stores UID mappings, both forward and reverse <ul style="list-style-type: none"><li>metrics for mapping metric names to UIDs</li><li>tagk for mapping tag names to UIDs</li><li>tagv for mapping tag values to UIDs</li></ul>	
tsdb-meta	Meta data for each time series	
tsdb-agg		

# TSDB - UID

- Each data point is associated with a metric and list of tag name/value pairs
- Here is an example
  - Curveball.CBUFE\_SSI\_REQUESTS.numRequests (metric)
  - tagk: device, tagv: [Mobile|Tablet|Desktop]
  - tagk: property, tagv: [Search]
  - tagk: partner, tagv: [ysearch|syndi|yhs|None]
- Each metric, tag name and tag value is assigned a unique identifier (UID)
  - Each Metric, TagK (tag name), TagV (tag value) are assigned unique UID in each category
- Each timestamp will be normalized to an hour value
- When data point is written to TSDB, key is formatted as follows
  - `<metric_UID><timestamp><tagk1_UID><tagv1_UID>[...<tagkN_UID><tagvN_UID>]`

# tsdb - schema

- A single metric row contains multiple columns
  - The qualifier is comprised of 2 or 4 bytes that encode an offset from the row's base time

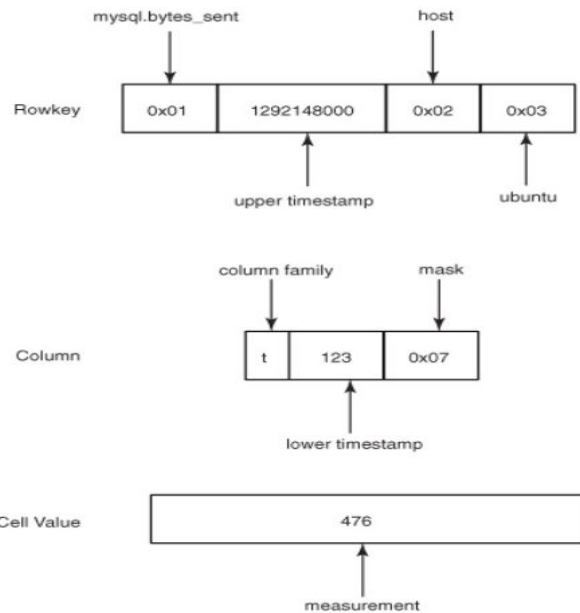


- flags to determine if the value is an integer or a decimal value
- 2 Bytes - offset in seconds, 4 Bytes - offset in milliseconds

- Column Values

- 1 to 8 bytes encoded as indicated by the qualifier flag

- Row Key Format



tsdb

# Inside HBase

Table: tsdb

Row Key	Column Family: t						
	+0	+15	+20	...	+1890	...	+3600
	0.69		0.51		0.42		
	0.99	0.72					

  
put proc.loadavg.1m 1234567890 0.42  
host=web42 pool=static

# UID Table Schema

- UID Table contains mappings between [metric|tagk|tagv] and UID (reverse also)
- Two column Families
  - id column family (name -> UID), Row Key is name, Column Qualifier: [metric/tagk/tagv]
  - name column family (UID -> name), Row Key is UID, Column Qualifier: [metric/tagk/tagv]
  - Special row with UID : 0, Column Qualifier: [metric/tagk/tagv]
    - Contains next UID for that [metric|tagk|tagv]
- Example:

```
hbase@ubuntu:~$ tsdb mkmetric mysql.bytes_sent mysql.bytes_received
metrics mysql.bytes_sent: [0, 0, 1]
metrics mysql.bytes_received: [0, 0, 2]
```

```
hbase@ubuntu:~$ hbase shell
hbase(main):001:0> scan 'tsdb-uid', {STARTROW => "\0\0\1"}
ROW                                COLUMN+CELL
 \x00\x00\x01                     column=name:metrics, value=mysql.bytes_sent
 \x00\x00\x02                     column=name:metrics, value=mysql.bytes_received
 mysql.bytes_received              column=id:metrics, value=\x00\x00\x02
 mysql.bytes_sent                 column=id:metrics, value=\x00\x00\x01
4 row(s) in 0.0460 seconds
hbase(main):002:0>
```

# tsdb-uid

## Inside HBase

Table: tsdb-uid

Row Key	Column Family: name			Column Family: id		
	metrics	tagk	tagv	metrics	tagk	tagv
0 0 1		host	static			
0 5 2	proc.loadavg.1m					
host					0 0 1	
proc.loadavg.1m				0 5 2		

0 5 2

put proc.loadavg.1m 1234567890 0.42

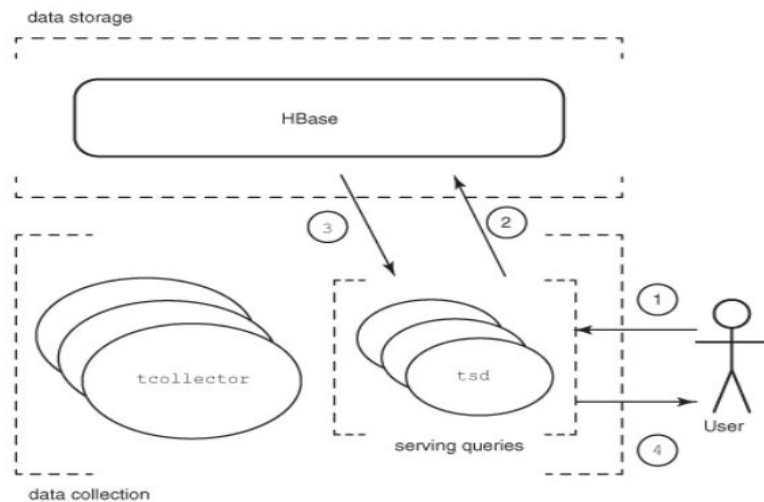
0 0 1 0 2 8 0 4 7 0 0 1

host=web42

pool=static

# Serving Queries

- User specifies query params
- tsd constructs filter and requests range scan
- HBASE scans key range
  - omitting filtered records
- Finally tsd returns results set



<https://github.com/OpenTSDB/opentsdb/blob/master/src/search/TimeSeriesLookup.java>

# Querying Open TSDB

```
sys.cpu.user host=webserver01,cpu=0 1356998400 1
sys.cpu.user host=webserver01,cpu=1 1356998400 4
sys.cpu.user host=webserver02,cpu=0 1356998400 2
sys.cpu.user host=webserver02,cpu=1 1356998400 1
```

- OpenTSDB includes a process called tsd for handling interactions with hbase
  - Exposes simple HTTP interface
- All time series for a metric
  - Scanner to fetch all data points for the metric UID 01 between [start ts, end ts]
- Filter on a tag
  - Time series that contain a specific tagk/tagv pair combination for a given metric
- Example Queries
  - `start=1356998400&m=sum:sys.cpu.user{host=webserver01,cpu=0}` => will return value 1
  - `start=1356998400&m=sum:sys.cpu.user` => 8 at 1356998400 that incorporates all 4 time series
  - `start=1356998400&m=sum:sys.cpu.user{host=webserver01}` => will return value of 5



# openTSDB Schema Implications

- All data points for a given metric next to each other
- All data points for a time range next to each other
- Tag filtering is pushed down to HBASE server
- Efficient range scans - Compact data

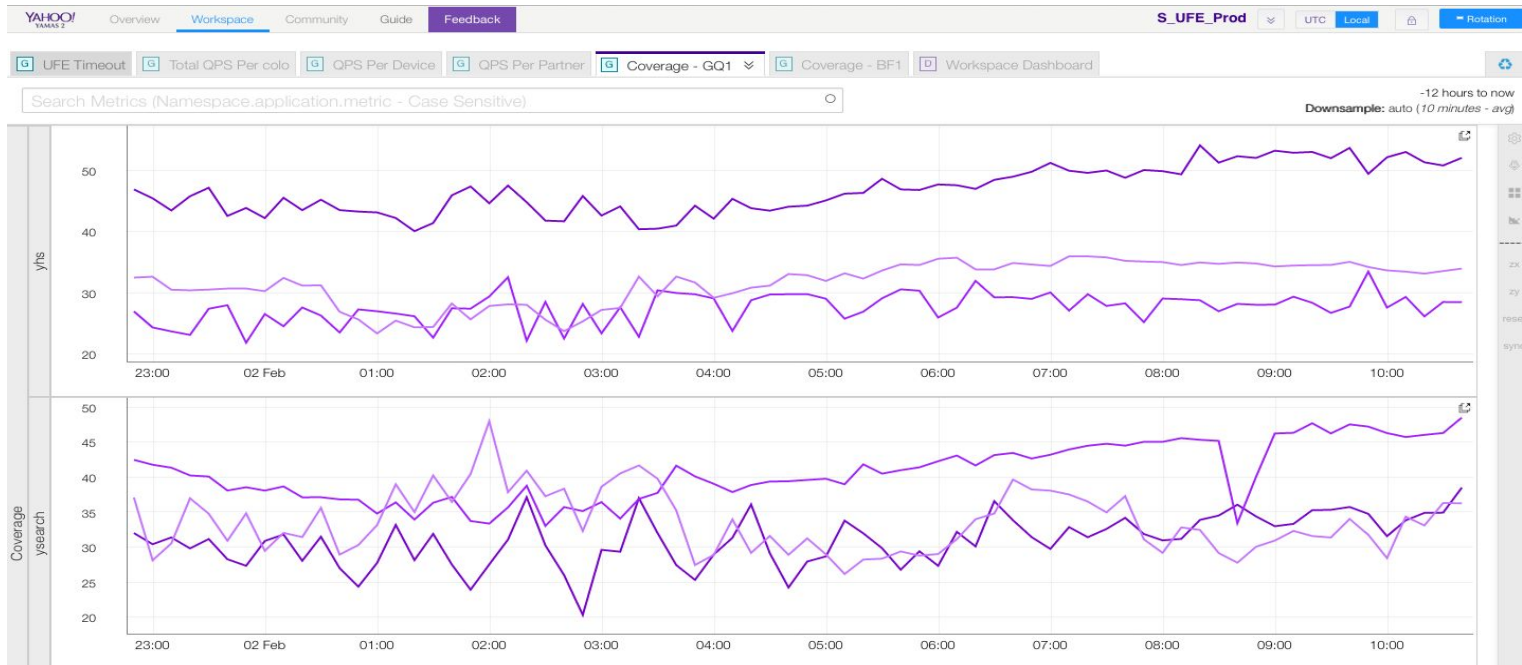
Row Key	Column Family: t						
	+0	+15	+20	...	+1890	...	+3600
	0.69		0.51		0.42		
	0.99	0.72					

# TSUID Index and Meta Table

- Time Series UID = data table row key without timestamp
  - Metric UID + TagK UID + TagV UID + ... (all TagKs are sorted)
- Use TSUID as the row key
  - Can use a row key regex filter to scan for metrics with a tag pair
  - Tags associated with a metric
- Atomic Increment for each data point
  - `ts_counter` - Tracks the number of data points written

# YAMAS - 2.0

Example: S\_UFE\_Prod <http://yamas.ops.yahoo.com:9999/#/ws/wjn462/uav>



# Region Splitting And Merging

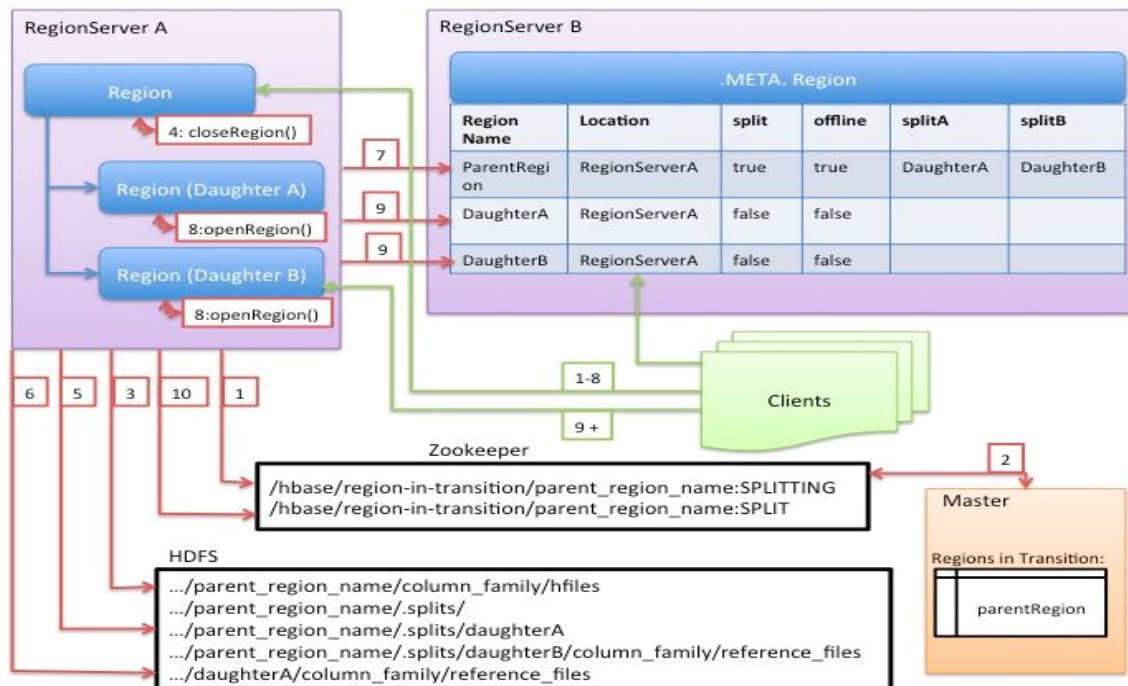
- Region is a continuous range within key space (start key, end key)
  - Regions are not overlapping
  - Served by only one region server at any point in time
  - Region consists of multiple many store files (one for column family)
  - Contains one memstore (write cache), block cache (read cache) and one or more HFiles
  - Regions are mechanism to distribute read/write loads across region servers
- Pre-splitting
  - Can create a table with many regions by supplying the split points at the table creation time
  - Need to know key distribution before hand (Be Aware of Data skewness, hotspots etc)
  - HBase supplies a few utilities to compute split points
    - `hbase org.apache.hadoop.hbase.util.RegionSplitter test_table HexStringSplit -c 10 -f f1 (10 regions)`
    - `create 'test_table', 'f1', SPLITS=> ['a', 'b', 'c']`
  - Need to monitor load on regions once rows are getting inserted into these regions

# Region Auto Splitting

- Once a region gets a certain limit, it automatically split into 2 regions
- Many region split policies exist in HBASE
  - ConstantSizeRegionSplitPolicy
    - Split when total data size for one of the stores > hbase.hregion.max.filesize (default 10GB)
  - IncreasingToUpperBoundRegionSplitPolicy
  - KeyPrefixRegionSplitPolicy
    - Configure the length of the prefix for row keys for grouping them
    - Regions are not split in the middle of a group of rows having the same prefix
    - Rows having the same rowkey prefix always end up in the same region

```
'yamas:tsdb', {TABLE_ATTRIBUTES => {DURABILITY => 'ASYNC_WAL', METADATA =>
{'KeyPrefixRegionSplitPolicy.prefix_length' => '8', 'SPLIT_POLICY' =>
'org.apache.hadoop.hbase.regionserver.KeyPrefixRegionSplitPolicy'}}, {NAME => 't',
COMPRESSION => 'LZO', TTL => '5356800 SECONDS (62 DAYS)', METADATA =>
{'ENCODE_ON_DISK' => 'true'}}
```

# Region Splitting



# PIG Integration

- Pig supports reading/writing to existing HBASE tables (Load/Store Funcs)
  - Maps table columns as Tuples
  - Include the row key as the first field for read operations
- One mapper per region
- Examples
  - DomainObjectKeyVal = LOAD 'hbase://\$table' USING `org.apache.pig.backend.hadoop.hbase.HBaseStorage`('d:m', '-loadKey true -minTimestamp \$minHWM -maxTimestamp \$maxHWM -caching 1000');
  - STORE T INTO 'excite' USING `org.apache.pig.backend.hadoop.hbase.HBaseStorage`('colfam1:query');

## Summary

***A sparse, consistent, distributed,  
multi-dimensional, persistent, sorted  
map***