Introduction to Apache HBase(part 2)

medium.com/@lagrang09/introduction-to-apache-hbase-part-2-ee00bd1ec9bb

This is second part of blog post dedicated to Apache HBase basics. First part can be found here.

This chapter will be dedicated to HBase administration topics, e.g. HBase cluster architecture, replication, data storage format, etc. It will be helpful for system administrators as well as developers which want to know how HBase works inside.

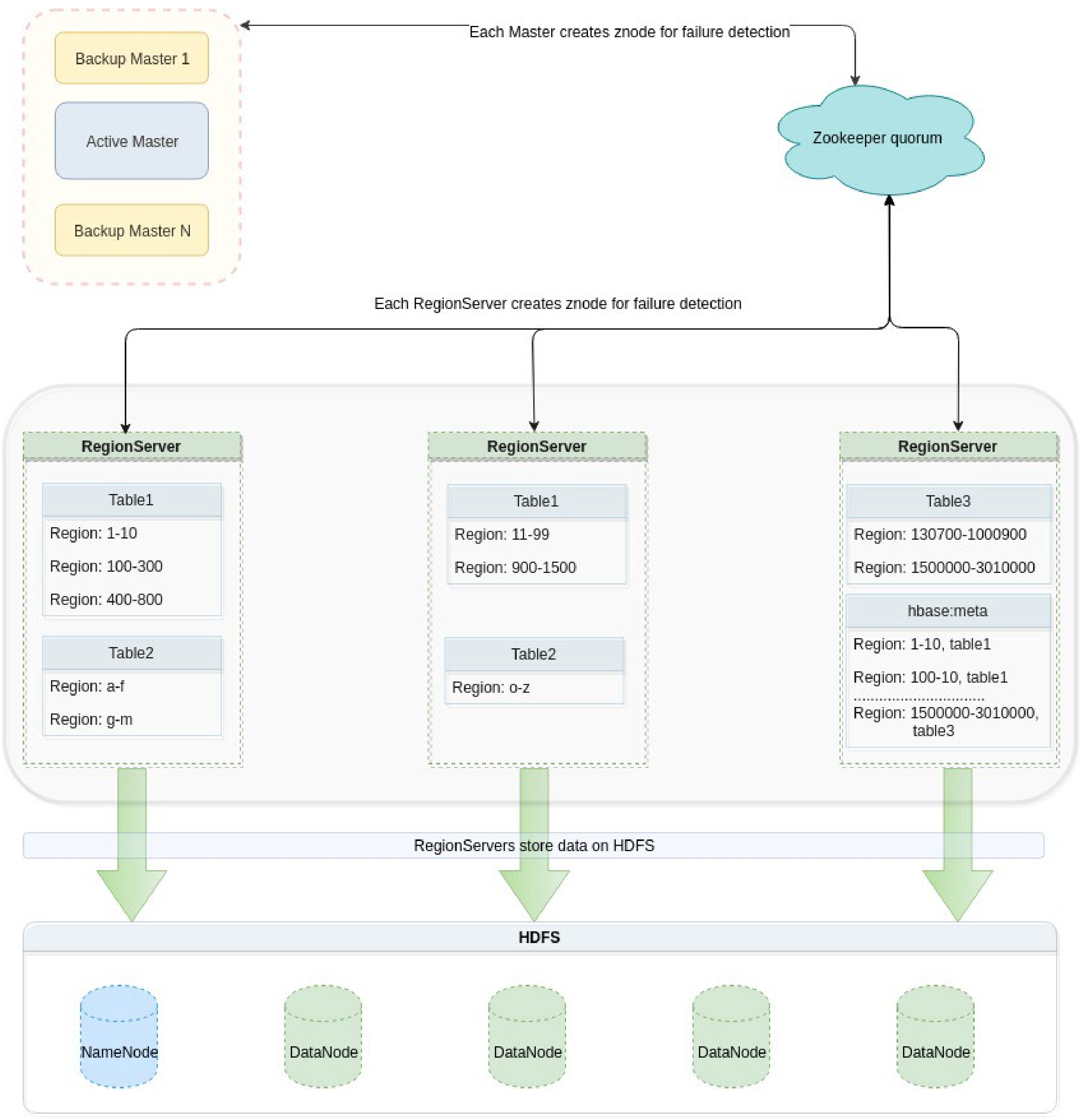
HBase architecture

We start from components which HBase cluster have under hood and how it interacts with each other.

HBase cluster consist of few Master servers and many RegionServers.

HBase runs on top of Apache Hadoop(it mostly requires only HDFS where it stores the data) and Apache Zookeeper. Apache Zookeeper cluster is used for failure detection of HBase nodes and stores distributed configuration of HBase cluster(more info in following sections).

Following diagram shows typical HBase cluster and how it components interacts with each other:



Master

Master is responsible for following tasks:

- monitoring RegionServers(detecting failures through Zookeeper)
- assigning regions to RegionServers
- region load balancing between RegionServers
- cluster metadata handling(for instance, table/CF creation/altering/removal, etc)

HBase cluster typically consists of multiple Masters, one of which is active and other are backup. When all master instances run, each start leader election(by using Zookeeper) to become an active master. Then some instance wins election, others switch to “observer” state and wait until active master will fail(and start new round of election).

RegionServer

Other component of HBase cluster is RegionServer. You can think about it as a “worker” node which is responsible for serving client requests and managing data regions.
Let’s talk about regions. As we already know, tables in HBase consist of rows which are identified by key. Rows are sorted according it’s key in data structures inside of HBase. Region is a group of continuous rows defined by start key and end key of rows which belong to it. RegionServer hosts multiple regions of different tables. It’s important to note that regions of the same table may be hosted on different servers, e.g. table data is distributed across cluster. But each region is managed by only one RegionServer at a time(this guarantees that row mutation is atomic, see ACID section in first chapter).

When RegionServer fails, Master reassign all regions to another RegionServers. Because all region’s data stored on HDFS, Master can safely assign region to any live server. Typically RegionServer and HDFS DataNode are collocated on the same host. But when region will be assigned, data belongs to this region will be “non-local” for responsible RegionServer, because collocated DataNode may not contains replica for region’s data. This can affect performance.

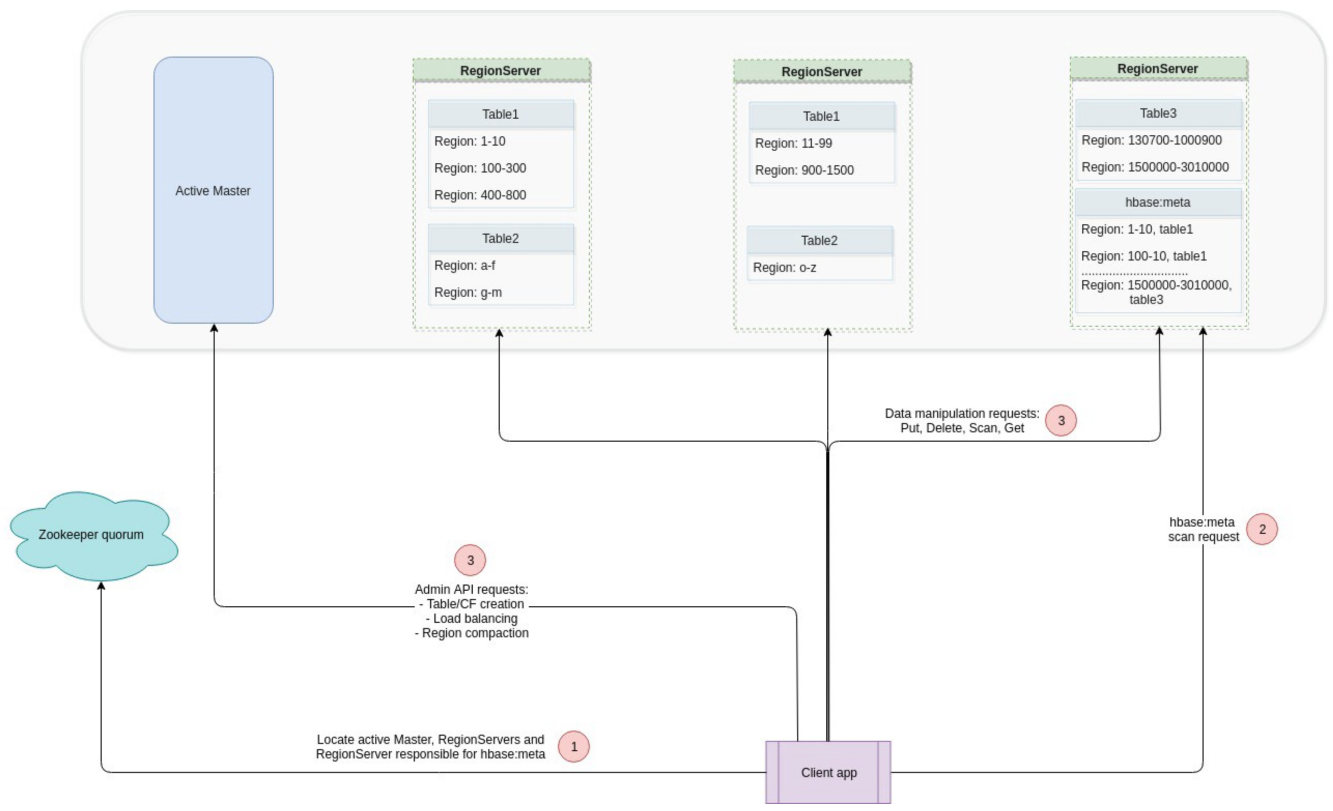
But situation not so bad. During the time, RegionServer will perform compaction of files and compacted files will be written on local DataNode: when RegionServer save data on HDFS it will write first replica on the same host as RegionServer, and other replicas on remote DataNodes. From this point, data will be local for RegionServer which will improve performance. More information about data locality see in docs.

Other notable RegionServer responsibilities are:

- split region into few smaller regions(see docs)
- region compaction(see docs)
- WAL splitting(see docs)

Client-cluster interaction

In this section we get high level overview of how clients connect to HBase cluster and interact with it. Following simplified diagram shows how client interacts with HBase cluster:



Client requires Zookeeper quorum connection string(which contains all Zookeeper quorum servers, e.g. “server1:port, ..., serverN:port”) and base znode which is used by HBase cluster(see `zookeeper.znode.parent` server property). It will be used to connect to Zookeeper quorum and read location of hbase:meta system table(which RegionServer manage it now). Then it connects to this RegionServer and read content of hbase:meta to cache region locations. hbase:meta table contains metadata of all regions of all tables managed by cluster. Using cached region metadata, client can find RegionServer which can handle request for particular row. But data in this cache can become invalid, for instance, when Master reassing regions between RegionServers. In this case, client will request RegionServer which already relinquish region serving and it responds

with "NotServingRegion" error. On receiving "NotServingRegion" error, client will invalidate hbase:meta cache and repeat request to new RegionServer.
As you can see, for typical data manipulation requests, client doesn't interact with Master and doesn't depend on it's availability. But administration API(table/CF/namespace creation, start load balancing,region compaction, etc) requests requires connection with active HBase Master.

On-disk data representation

In this section we discuss how HBase stores data on disk.
As we already know, RegionServer responsible for data managing in HBase cluster. For each column family in each region, RegionServer create so-called ***Store***. Store consists of MemStore and a collection of on-disk StoreFiles(HFiles).
MemStore is in-memory data structure implemented by skip list. It contains cells or key-values which are represent last data changes. All Put and Delete requests served by RegionServer applied to MemStore and also written into WAL for durability. MemStore have configurable max size which by default is 128MB. When MemStore will reach this limit, it will be flushed to disk(in HFile format) and RegionServer create new empty MemStore.
HFile is file format based on SSTables described in BigTable paper. HFile consist of sequence of blocks of different types with size of 64KB(configurable value). Blocks can have different types:

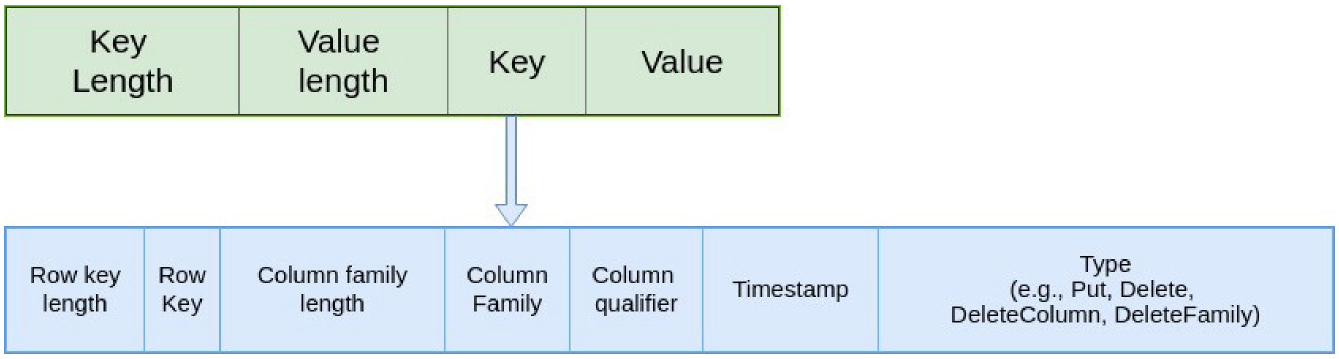
- data block(actually contains data cells)
- index block(have subtypes for data, meta and bloom)
- bloom filter block(contains Bloom filter)
- file info block(block with key-value map of metadata)
- meta block

In following sections, I will shortly describe each type of blocks. Detailed description of HFile format can be found in docs.

Data blocks

Each data block consists of KeyValue data structures. KeyValues inside of HFile are sorted according following rule: first by row, then by ColumnFamily, followed by column qualifier, and finally timestamp (sorted in reverse, so newest records are returned first).

Following picture illustrate on-disk representation of KeyValue(blue part is expanded view of key):



KeyValue never cross block boundaries, e.g. if it have size greater than block size, it will be written into one block.

As you already know, HBase row consist of many cells which are presented as KeyValues on disk. In common case, cells of the same row have many fields which contain same data(most frequent is a row key and column family). To reduce disk usage, HBase have a option to enable data encoding/compression. More information about which compression/encoding algorithm to choose, read the Compression and Data Block Encoding In HBase section in official docs.

Index blocks

Index blocks inside of HFile contains index structure. It provide quick binary search by key to find blocks which contains particular row.

Bloom filter blocks

Bloom filter blocks contain chucks of Bloom Filter. Bloom Filter is a data structure which is designed to predict whether a given element is a member of a set of data.
When HBase tries to execute Get request for row, it uses Bloom Filters to detect whether row present in this HFile. If not, then HBase skips entire HFile and keeps scanning other files. But it is important to note, that

Bloom Filters is a probabilistic structure which can get “false positives”, e.g. it can say that row contained in HFile, but actually it doesn’t. In that case HBase must perform additional reads of HFile to ensure that row present in file.

Today(versions before HBase 2.2), Bloom filter only used for Get operations and doesn’t support Scan. But recently, I found HBASE-20636 which will fill this gap. It will add support of prefix Bloom filters. When your start and stop keys of Scan have common prefix, scan will use Bloom filter to filter out files which not contain rows with this prefix. According to ticket, this feature planned in HBase 2.2.0.

Region replication

HBase provides strongly consistent reads and writes. Strong consistency achieved by fact that each region managed by one RegionServer. But in case of RegionServer failure, all it regions will be inaccessible during some time. This time defined by Zookeeper session timeout, by default, 90 sec(see docs). That timeout value can be decreased to reduce time to recovery(TTR). But this can lead to spurious failures caused by temporary network issues, Java GC, etc which can lead to excessive regions reassignment and consequently to improper balance of regions across RegionServers.
Feature known as region replication designed to partially overcome this limitation. By default, each region have only 1 replica. When replication factor increased to 2 or more, region will be assigned to several RegionServers. One of this replicas is **primary**, which accepts writes and reads to this region. Other replicas is a **secondary**, it can handle only read requests.
HBase replication is asynchronous process and can take some time to propagate new writes to secondary replicas. Because visibility of a changes can be delayed, client have two options:

- strong consistent read
- timeline consistency Strong consistent reads goes through primary replica and will see all changes. Timeline reads can see stale data in response. Timeline read starts by requesting primary replica, but

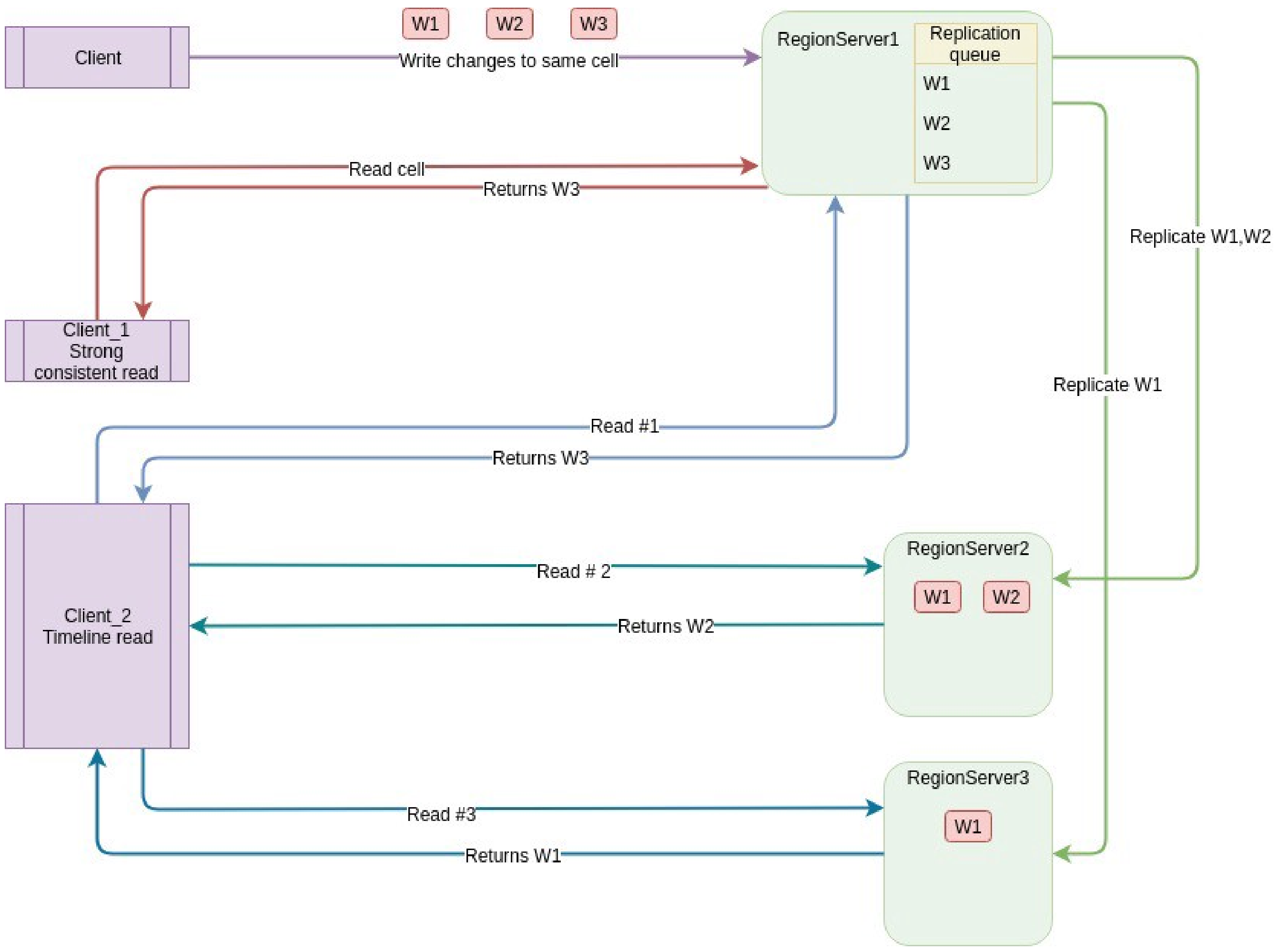
after short interval(configurable parameter) client will send next request to secondary replica. Client will see response which will contains data from replica respond first.

Timeline read example

Suppose that we have 3 RegionServer(RS1,RS2,RS3), 1 write-only client(CW1) and 2 read-only(CR1 and CR2). RS1 hosts primary replica, RS2 and RS3 hosts secondary(RS2, RS3 is a replication sinks). CW1 client execute 3 write operations W1, W2, W3 one by one with some time delays between. CR1 client read only from RS1 and CR2 use timeline reads and can read from any server.
As you can see on picture, CR1 always read last value W3 written by W1. CR2 execute 3 consequent reads with timeline consistency:

1. First read get response from primary replica and see last written value W3.
2. Second read goes to RS2 server which see W2 as last value.
3. And third read operation get response from RS3 which see W1 as last written value.

As you can see, 3 consequent read operation with timeline consistency can return different result during the time. Application should be ready for this behavior to handle stale reads.



Replication caveats

Replication have few caveats which you should prepare to handle.

- increased memory usage: Region replicas have it's own memstores.
- stale reads
- extra network traffic for data replication
- because implementation details, replicas can see partial updates(only for cross column family requests)

For more information about replication, see docs.

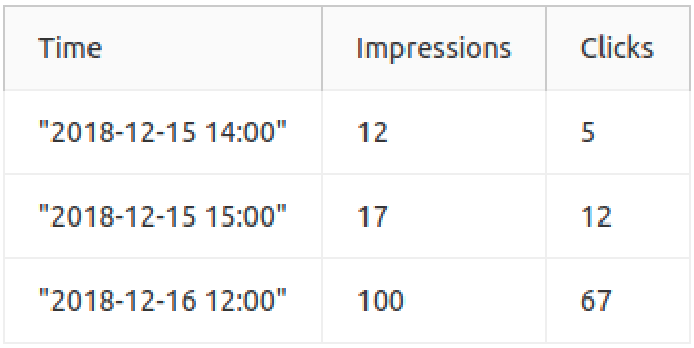
Use cases

This is last section in this post. At this point, we know how HBase works and ready to describe some possible use cases of HBase. Examples have domain-specific description and detailed explanation how we store data inside of HBase.

Example 1: Realtime data aggregation

Suppose, we have advertising platform which show ads on web sites and/or mobile apps. Advertisers start it campaigns and such platforms collect information about ad events, such as when ad was showed(impression event) or when user click on ad’s banner. Suppose, impression and click events written to Apache Kafka by service which catch it. And now, we as a team which prepare reports, need to collect all event into some aggregated form. This aggregations can be used as building blocks for time-based reports, where user want to get report with count of impressions and clicks some campaign in selected time range. Resulting data in report should be hour granularity, e.g. user want to see counter change each hour:

This task can be solved with help of HBase.



Time	Impressions	Clicks
"2018-12-15 14:00"	12	5
"2018-12-15 15:00"	17	12
"2018-12-16 12:00"	100	67

In fact, our report consists of two sub-reports: impressions count and click count. Let’s mark each subtype according to it event type: *EventType.IMPRESSION* and *EventType.CLICK*.
And now we have stream of events of different types which we poll from Kafka. Because we have large number of events, we decide to run few instances of our reporting service. Each service receive of batch of events(Kafka return bunch of records on each poll operation) and group it by [campaign ID, event type, timestamp] and compute count for each group. Timestamp in each group is a truncation of original event timestamp to beginning of hour, for instance, if ts=”2018–12–15 14:23:45" then truncated value will be “2018–12–15 14:00:00”. This truncation make possible to group same event which belong to same hour.
 Now we need to save new value of counter. We will use following schema to store data in HBase:

- Row key: `[CampaignID]_[TruncatedTimestamp]` with timestamp stored ad long value(epoch seconds).
- Columns: `[EventType]`

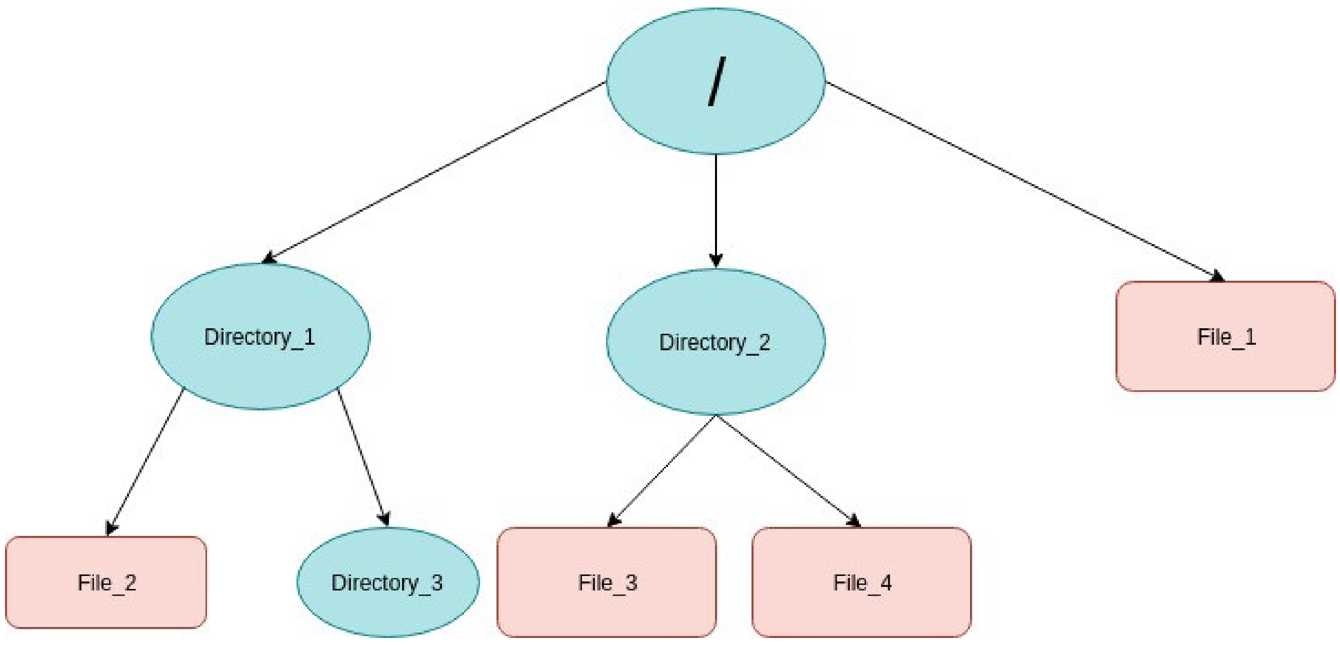
- Value: current event count value

Because we have few instances which can change counter value concurrently, we can’t simply put new counter value. We will use HBase `Increment` operation to atomically increment current counter value.
One interesting nuance on how we aggregate and write data to HBase. As you can see, we poll data from Kafka by batches and perform pre-aggregation by grouping events by [campaign ID, event type, timestamp]. Reader can propose other more simple solution which doesn't aggregate data on service side, but send bunch of increments by one for each [campaign ID, event type, timestamp]. This solution will work but have performance impact because increment operation use some sort of CAS on server side. When all service instances will send batch of increment operations, HBase will try to apply increments from few client requests to same cell. This will create contention on server side and affect request execution time. That's why we use pre-aggregation on service side and reduce count of increments executed on HBase side.

Example 2: File system image storage

Suppose, we try to implement high level document/file store for non-advanced users which will interact with it though web interface. First idea, that you can imagine, is to store file content inside HBase, because it can store raw binary data in cells. But, in reality, HBase not designed to storage big BLOBs.
In documentation we can find section with name of “Storing Medium-sized Objects(MOB)” which starts with following: “Data comes in many sizes, and saving all of your data in HBase, including binary data such as images and documents, is ideal”. Great, this can help us. But if we read whole section, we realize that this feature focusing on BLOBs of size between 100KB and 10MB. To simplify things and suppose we can’t have files greater that 10MB in our system.
Typical file system contains file system image(FS hierarchy, tree of directories and files) and actual content of files(binary data). Using MOB features of HBase, we can easily store file content inside cells. Now, we need to define how we will store FS image.

FS image, as well as binary data of files, must be durably stored to prevent data loss. Also, we need fast access to list directory content. And again, HBase meet this requirements, because it provide durable storage and fast key-value access.
FS image can be represent as tree with nodes of different types. For simplicity, let’s suppose that we should support only files and directories. As any typical file system, our FS also define root node, which we mark as “/”. This is special type of node and any other FS tree node is successor of it.



Now, we start to design how we will store FS image in HBase.
As we defined earlier, we have 2 type of nodes, directory and files and 1 special root node. Each node will be represented as row in HBase table. Row key will contains unique node ID(for instance, GUID), generated when node was created. We cannot use node name(which is file or directory name) because it’s not unique.
Each row in HBase will have 2 column families: one for metadata and one for file’s content(MOB enabled column family). Metadata will contain:

- node type(FILE or DIRECTORY)
- parent node(node ID of parent directory)
- node children list(only for directory nodes)
- other typical file system info(creation/modification timestamp, owner, access rights and so on)

We should define operations which will be supported by our file system:

1. Create file or directory
2. Read file content
3. Write file content
4. List content of directory

From here, we start some coding and demonstrate how each operation can be implemented using HBase client API. This examples is not written to be optimal at performance point of view, as well as doesn’t contain exhaustive checks to prevent all type of errors.

Create file with content

List content of directory

Summary

HBase is very mature open-source project with reach feature set. It has big community, strong committers list(Alibaba, Cloudera, Hortonworks, Salesforce, etc). Project consistently evolves and expanded by new functionality, such as, SQL by Apache Phoenix, distributed transactions by Apache Omid/Apache Tephra.
As we see, HBase has many applications in different areas: storing metrics(see OpenTSDB project), advertising data, store file system metadata and even more that we can explore.

References

1. https://hbase.apache.org/book.html
2. https://hadoop.apache.org/
3. https://zookeeper.apache.org
4. BigTable paper: https://ai.google/research/pubs/pub27898
5. https://hadmin.io