Microsoft

Getting Started with HBase on Azure

Architecture Guidelines and Implementation Best Practices

Prepared by

DM Jumpstart Engineering Team (askdmjfordmtools@microsoft.com)

Note: The detail provided in this document has been harvested as part of a customer engagement sponsored through the Azure Data Services Jumpstart Program.

Table of Contents

Table of Figures

List of Tables

1 Introduction

HBase is a distributed No-SQL (Not-only-SQL) data store purposed to handle big data workloads using low-latent ingress-egress techniques in a scalable and fault-tolerant manner. The core data model supported by HBase can be considered as a distributed sparse matrix, where it can be scaled to accommodate billions of rows with millions of attributes per logical row. HBase supports schema-on-read principle, allowing you to persist data as key-value pairs. The key values will help establish a logical record structure at the time of read. Using read optimized immutable data structures (HFile format), HBase offers a low-latent access to scale writes and reads. It is also important to note that HBase is a compute-intensive framework (CPU, Memory & Network); followed by corresponding storage demands. It depends on a HDFS (Hadoop Distributed File System) compliant storage backend, for example – HDFS, WASB (Azure Storage Blob) or ADLS (Azure Data Lake Store) to meet at-rest persistence and data-replication responsibilities. Specific to the context of Azure HDInsight, recommend considering ADLS Gen2 or Gen1 as preferred choices. You can learn more about ADLS here.

This document covers HBase essentials with an intent to offer the readers with a jump-start approach to setup, configure and manage a healthy HBase based data foundation, with focus on following areas:

- Basic data model support, including thinking beyond SQL where terms like cardinality, referential integrity, etc., are well noted terms.
- Key system inter-dependencies that highlight critical connected components such as Zookeeper, HBase Master, Region Server and the HDFS compliant storage layers (HDFS, WASB and ADLS) influence your read-write paths i.e., I/O and overall achievable throughput.
- First steps towards strategizing for capacity i.e., knowing how much to build for and how to determine points of scale. For example, scaling out versus scaling in.
- Key configuration parameters to derive an implementation baseline that then allows you to meet your workload demands.
- Wrap-up notes i.e., items to watch for, when considering HBase as your data foundation.

2 Usage Guideline

HBase is a data-driven tool. For example, data patterns associated with your use cases i.e., workloads will exert greater influence on its runtime. The document strives to strike a balance between different levels of users i.e., seasonal users to those with hands-on exposure to the tool. The primary objective is to serve as a jump-start guide and a quick reference to essential tuning aspects.

Please refer to the following guidance to help you in leveraging the content, there-by becoming productive in addressing your goals for HBase:

- If you consider yourself as a starter and like to gain a deeper understanding about HBase strengths and how to play it well in your data foundation strategies, recommend to follow through the sections in the order they are represented.
- If you are a seasoned developer with hands-on skill using HBase, you may jump straight into HFile formats and the I/O paths, as you begin with your functional data modeling exercises.
- If you are the primary stakeholder and intend to manage the conceptualization through implementation steps, consider focusing on the strategy section. It offers recommendations that are helpful and critical in successful HBase adoption as a data foundation platform.
- Please note, this does not intend to be an all-in-all enumerator of all concepts associated with HBase's feature implementations. Its intent is to give you enough detail so that you can spring-board your implementations, learn and optimize from there-on!
- All references to Azure, including HDInsight cluster components or the storage layer options such as WASB and ADLS are made within an intent to explain how, when and where HBase can be further pushed to scale or to know where any such elastic-capabilities do become a constant. That is, you cannot achieve more per node in a HBase cluster and should look out for adding more nodes.
- However, if you are some one in rush and like to know the starting point to initiate a capacity sizing conversation, look for the section on "Rough-scale math for sizing ingress". On a purpose this does not go explicitly into egress. Such detail is implicitly captured in the way we derive ingress. Please feel free to reach out to us at feedback and suggestions alias shared towards end of this document.

3 Architecture

3.1 HBase Data Model

Let us look at the basic data model framework supported by HBase. It will help data modelers, analysts, architects or developers to realize the tool from its application relevance and derive appropriate solutions by working through the limitations. Will help in strategizing for better data foundations and data orchestration workflows.

3.1.1 Logical View

Following visual provides basic building blocks at a very high-level:

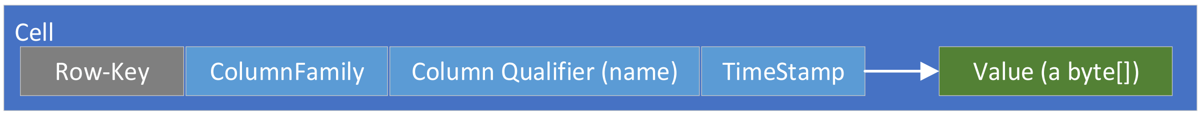


Figure 1: HBase Logical Data Model

Let's briefly understand these terminologies:

- *Cell* – least possible atomic data point that can be stored in the data store.
- *Row-key* – a unique identifier of the entry; cells with common row key form a logical row.
- *Column-family* – a logical group of cells that often are stored together. Such logical grouping can be derived from the underlying functional use case.
- *Column Qualifier (a.k.a., column name)* – name of a cell or column. Example – `uid` that represents a user id in a User table.
- *Timestamp* – a user provided timestamp value (a long value) associated with the cell. If timestamp is not provided, HBase would apply system time. This value is used to track versions. ***Note*** - HBase retains the most recent value if user tries to persist a key-value pair that has same row-key, column family, column-qualifier, and timestamp, but different values.
- *Value* – HBase persists user data as a byte array collection. No other type information is retained.

When you portray across the universe of all {key → value} pairs in your domain entity (i.e., table), the physical structure will look somewhat like the following empirical structure:

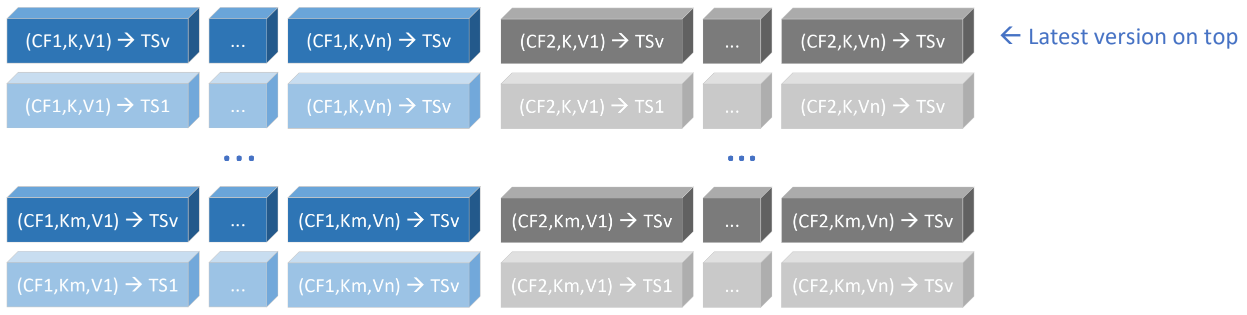


Figure 2: Grouping and Partitioning Approach of the Data Model

The physical layout takes this logical structure, packages keys that belong to a common column family into one columnar organization, partition such data further by key-ranges and stores them in a distributed fashion across the available region servers. Each RegionServer is then responsible to manage the range assigned to it.

3.1.2 HFiles – Physical View

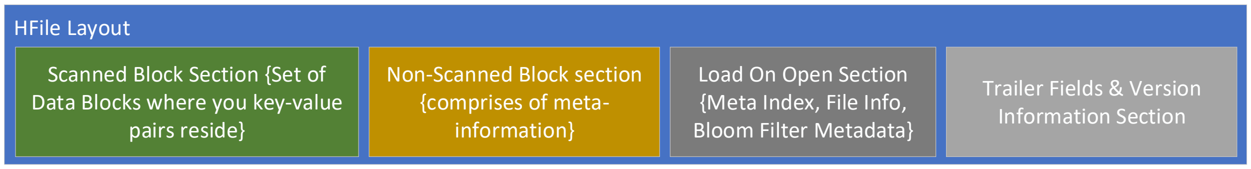

HFile Layout
Scanned Block Section {Set of Data Blocks where you key-value pairs reside}
Non-Scanned Block section {comprises of meta-information}
Load On Open Section {Meta Index, File Info, Bloom Filter Metadata}
Trailer Fields & Version Information Section

Figure 3: HFile - Empirical view of physical data model

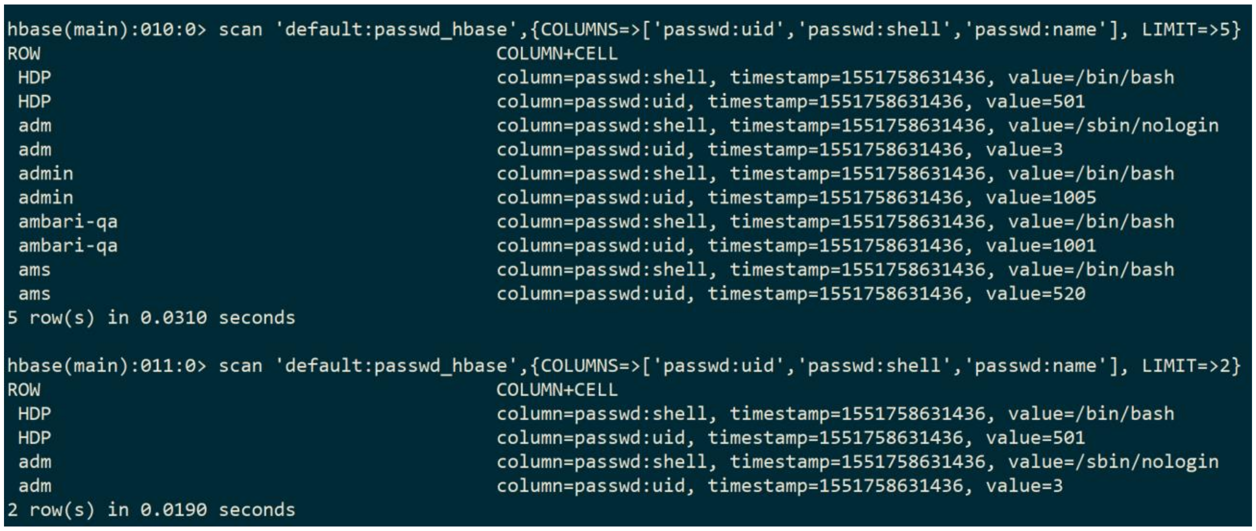
User data is packaged in store files using the HBase native file format called – HFile. The format has evolved over a period, with the latest being version 3. This is introduced as part of 0.98 release. It bears similarity in layout with its predecessor i.e., version 2, with only one exception that it supports ability to store cell-level additional meta-data called tags. These tags are user definable, in the sense one can include tags as part of individual put requests as part of your key-value pairs. For example, consider a measurement dataset, where in based on certain value thresholds per logical row, you would want to add additional qualifier, example low, moderate and high. This additional dimension can be captured as a tag on the put request. A put is a collection of one more attribute that form a logical role. Another example is use of tagging feature by HBase internals to support cell-level ACL.

HFiles are physically persisted as HDFS blocks, where the recommended configuration is to have a HFile size under a physical HDFS block. This is to avoid overhead of reading more blocks per read. Idea here is to achieve read optimization. These two - HFile and HDFS block, are conceptually two different topics. A block should represent grouping of similar data (in this case by key-order).

Once persisted a HFile remains immutable. HBase uses an internal process called compaction cycles to organize data, sorted in lexicographical order and over a period generate better organized data sets and corresponding HFiles. Once merged the older HFiles are decoupled from the meta-store information and purged. This is completely internal to HBase. Considering the out-of-the-box sort order, it is very important that table modelers pay a close attention to their row-key design. See section on *Data Modeling Quick Tips* below for additional detail.

Note: This layout is consistent with both the version 2 and version 3 formats, with exception in case of version 3, the File Info section will have details about tags (hfile.MAX_TAGS_LEN & hfile.TAGS_COMPRESSED). More about HFile format can be found in the HBase book section on HFile Version 3.

3.2 Physical Layout Sample

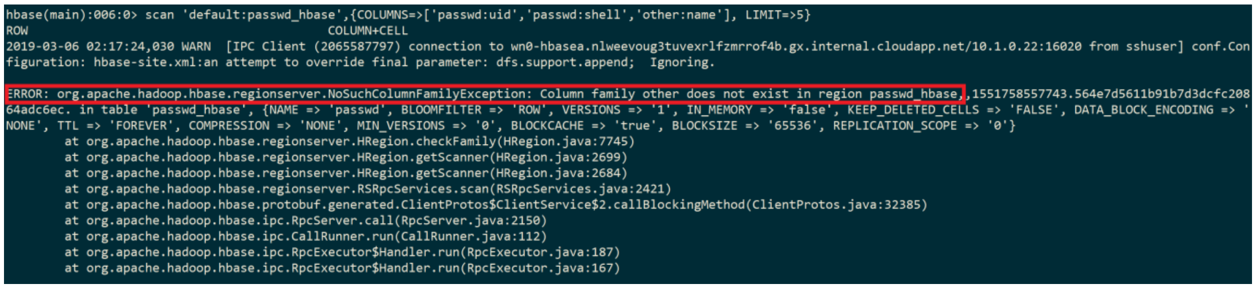


```
hbase(main):010:0> scan 'default:passwd_hbase',{COLUMNS=>['passwd:uid','passwd:shell','passwd:name'], LIMIT=>5}
ROW                                          COLUMN+CELL
 HDP                                         column=passwd:shell, timestamp=1551758631436, value=/bin/bash
 HDP                                         column=passwd:uid, timestamp=1551758631436, value=501
 adm                                         column=passwd:shell, timestamp=1551758631436, value=/sbin/nologin
 adm                                         column=passwd:uid, timestamp=1551758631436, value=3
 admin                                       column=passwd:shell, timestamp=1551758631436, value=/bin/bash
 admin                                       column=passwd:uid, timestamp=1551758631436, value=1005
 ambari-qa                                   column=passwd:shell, timestamp=1551758631436, value=/bin/bash
 ambari-qa                                   column=passwd:uid, timestamp=1551758631436, value=1001
 ams                                         column=passwd:shell, timestamp=1551758631436, value=/bin/bash
 ams                                         column=passwd:uid, timestamp=1551758631436, value=520
5 row(s) in 0.0310 seconds

hbase(main):011:0> scan 'default:passwd_hbase',{COLUMNS=>['passwd:uid','passwd:shell','passwd:name'], LIMIT=>2}
ROW                                          COLUMN+CELL
 HDP                                         column=passwd:shell, timestamp=1551758631436, value=/bin/bash
 HDP                                         column=passwd:uid, timestamp=1551758631436, value=501
 adm                                         column=passwd:shell, timestamp=1551758631436, value=/sbin/nologin
 adm                                         column=passwd:uid, timestamp=1551758631436, value=3
2 row(s) in 0.0190 seconds
```

Figure 4: Physical Layout Data Sample

As you observe in the above visual, data is retrieved in the column-order format as also stored in the at-rest layer i.e., storage. Interesting note here is the output from the second query i.e., scan request. Though the request specifies it is looking for data for column qualifier `name`, at the time of response, the query engine does not throw an error, instead simply ignores it. Reason, if a key-value pair qualified by that column name exists, it will be included in the response as well.

Let's look another variation here, when a client asks for key-value pairs that belong to a non-existent column family:



```
hbase(main):006:0> scan 'default:passwd_hbase',{COLUMNS=>['passwd:uid','passwd:shell','other:name'], LIMIT=>5}
ROW                                    COLUMN+CELL
2019-03-06 02:17:24,030 WARN  [IPC Client (2065587797) connection to wn0-hbasea.nlweevoug3tuvexrlfzmrrof4b.gx.internal.cloudapp.net/10.1.0.22:16020 from sshuser] conf.Con
figuration: hbase-site.xml:an attempt to override final parameter: dfs.support.append;  Ignoring.

ERROR: org.apache.hadoop.hbase.regionserver.NoSuchColumnFamilyException: Column family other does not exist in region passwd_hbase,,1551758557743.564e7d5611b91b7d3dcfc208
64adc6ec. in table 'passwd_hbase', {NAME => 'passwd', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false', KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => '
NONE', TTL => 'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}
        at org.apache.hadoop.hbase.regionserver.HRegion.checkFamily(HRegion.java:7745)
        at org.apache.hadoop.hbase.regionserver.HRegion.getScanner(HRegion.java:2699)
        at org.apache.hadoop.hbase.regionserver.HRegion.getScanner(HRegion.java:2684)
        at org.apache.hadoop.hbase.regionserver.RSRpcServices.scan(RSRpcServices.java:2421)
        at org.apache.hadoop.hbase.protobuf.generated.ClientProtos$ClientService$2.callBlockingMethod(ClientProtos.java:32385)
        at org.apache.hadoop.hbase.ipc.RpcServer.call(RpcServer.java:2150)
        at org.apache.hadoop.hbase.ipc.CallRunner.run(CallRunner.java:112)
        at org.apache.hadoop.hbase.ipc.RpcExecutor$Handler.run(RpcExecutor.java:187)
        at org.apache.hadoop.hbase.ipc.RpcExecutor$Handler.run(RpcExecutor.java:167)
```

Figure 5: Physical Data Layout - Column Family Assertion sample

3.3 No-SQL Support

Let us quickly check some important No-SQL aspects of the HBase architecture:

- HBase primary design goal is to support large data sets in the order of billions of rows by millions of columns, at the same time where low-latent accessibility is assured.
- It supports schema-on-read principle, where-in at the time writes your data is captured and maintained as a sparse collection of key-value pairs.
- The tool offers basic query language constructs, including critical DDL capabilities. Examples include functions such as create, disable, enable, scan, get and put.
- Key composition defines write and read efficiency, where columns that have alike keys are packed together. Hence it is very important to get your row-key composition right!
- Out-of-the-box, HBase maintains key uniqueness and offers sorted data sets. Data is sorted in lexicographical order.
- Framework offers extensible support to apply push-down predicates, that allow you to optimize volume of data in-flight to limit extracting just about much as needed by your query criteria.
- Choices include pre-defined set of filters and co-processors. The later are more like SQL-triggers applied within the query engine.
- DDL operations are handled by the HBase master, while DML is the responsibility of HBase RegionServers. Clients coordinate with zookeeper to realize cluster meta-state and can influence execution or runtime process of DML. Please review the section on Service Discovery that provides more detailed explanation of these different flow paths is provided.
- ACID (Atomicity, Consistency, Isolation and Durability) semantics are available in limited capacity around partitions. Transactions for example can be implemented with row-level locks.

- Tables are grouped by namespaces, and each table has one or more column families. Besides, HBase does not maintain meta-information about columns. It however supports a physical layout adhering to the data model described earlier.
- Schema is enforced, validated and applied during read phase.
- Higher-order frameworks such as Hive-HBase integration and Apache Phoenix offer SQL-semantics and capabilities to work with HBase using SQL interfaces.
    - When you intend to use Phoenix, it is recommended to approach DDL using Phoenix DDL constructs.
    - This will help the query engine to take advantage of provisions, such as Phoenix co-processor framework and the additional column family to hold meta-information that phoenix query engine depends on.
    - For example – HBase does not support secondary indices out of the box. Phoenix uses the additional column family that maps relevant keys to offer SQL capabilities that depend on secondary indices.

3.4 Data Modeling Quick Tips

Following are few technical recommendations i.e., tips to aid your data modeling effort:

- Row-key composition defines data distribution across participating region servers in a HBase cluster.
    - As a best practice it is recommended to come up with a composite defined based on attributes from the source data set.
    - For example, in case of an IoT data set maintained as a time series collection, consider a row-key of the format – `YYYYMMDDHHGUID` that potentially give you the essential randomization of write operations and also allow you maintain low-latent access to data.
    - For any given combination, it is recommended to play it by data, balancing between write and read requirements.
- It is recommended to always play-out a sample data set to validate, even a well-defined row-key before you can bring your actual workload. It will help isolate data-driven issues from that of infrastructure provisioning aspects.
- As you have noticed by now, data is stored in key → value pairs. The value portion is just a collection of bytes persisted as a byte-array. Types are applied in the compute layer i.e., write or read layers and not at storage.
- Data that is qualified under a common column-family is packed together in a columnar-fashion and partitioned by key-ranges. HBase ensures to create partitions at key boundaries and stores each partition segment as a block. These blocks form the composite parts of a HFile.

- HFile is the unit level of parallelism and replication. Replication is handled by HDFS-compliant storage layer.
- Be economical on the length of row-key. Keys are hashed and stored in the indices to help faster lookups. Also, at the time of DDL execution, you can recommend the type of encoding scheme that HBase should follow for a given column family.
    - For long keys if you cannot avoid them due to functional constraints, ensure your encoding is set to FAST_DIFF scheme.
    - Also, consider the ordering of composite parts based on your read preferences, besides writes alone.
- Key splits can be pre-defined and allocated as part of your DDL request. This will allow HBase to assign these split ranges across each available HBase RegionServer.
- Each RegionServer must be organized to hold certain maximum number of splits or regions. This is to optimize compute resource utilization – memory and CPU.
- At the time of DDL submission, you can also include NUMREGIONS parameter. Infer the appropriate number based on your existing workloads that your cluster is currently supporting.
    - Revisit this on a periodic basis to achieve right balance such that you can ensure desired high-availability, example – four-9's or five-9's.
- To optimize your tables for scans, also consider utilizing the push-down predicate features – filters and co-processors. Co-processors must be leveraged with at-most caution. They are part of region server's runtime and are initialized at the time of your region initialization.
- Compactions do not influence your active writes or reads, to the point the outcome of compactions is registered in the cluster's meta-data. However, note that your I/O can still suffer from lack of compute resources to handle your requests.

Note: Recommend checking some more recommendations as noted in the following sections:

- Approaching table designs (i.e., Physical Models)
- Strategizing for Capacity – Quantitative References

3.5 Connected Components

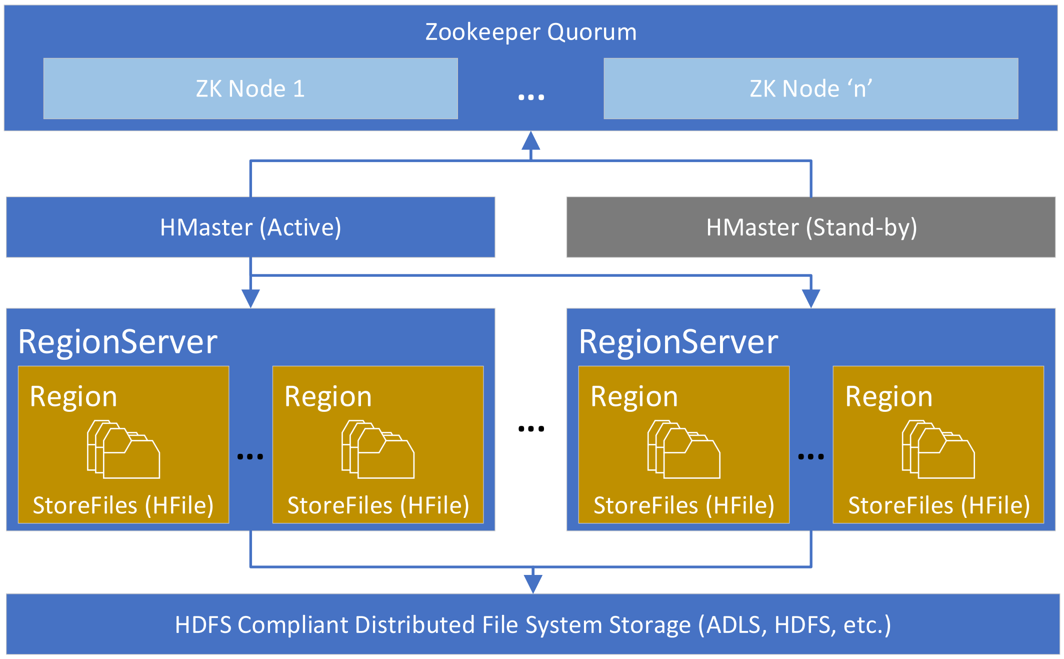


Figure 6: HBase Architecture - Connected Components

The above visual offers an empirical view of the connected components in a typical HBase setup. For brevity, it highlights only the critical inter-dependencies between such components, such as dependency on Zookeeper, a typical HA-setup, etc. The idea here is to visually demonstrate how these different pieces come together to serve a scalable, distributed and highly available data foundation.

3.6 Service Discovery

HBase is a distributed data store. Hence, it is important to understand how services are discovered. Following visual provides an empirical view, considering each connected component:

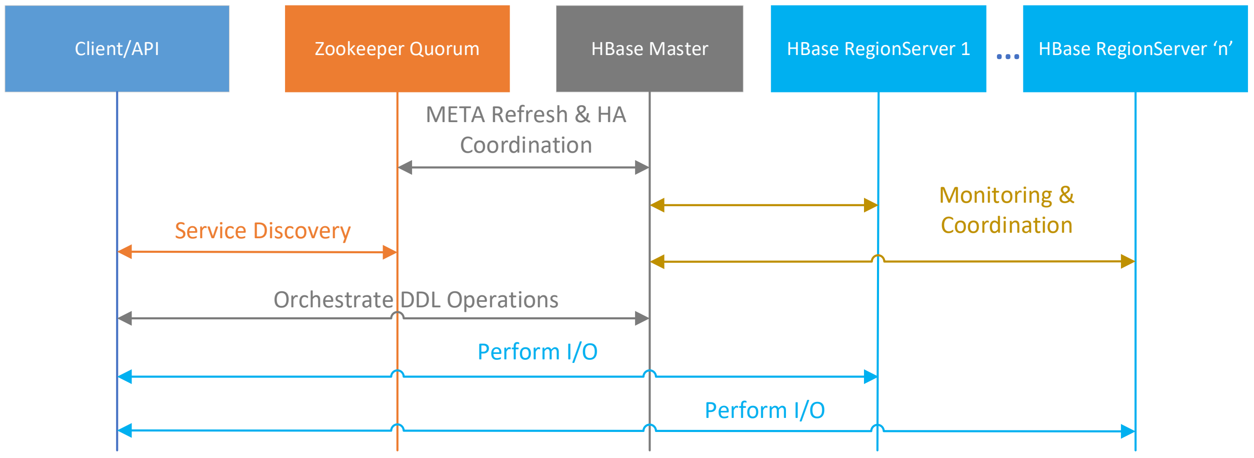


Figure 7: Service Discovery

Briefly about various interactions:

- Any interaction with and between HBase cluster starts with a service discovery call to Zookeeper.
- Each of the paths – DDL or DML begin with Zookeeper, and further fork-out into an RPC call with HBase Master or the appropriate RegionServer.
- HBase Master is never in the path of an RPC call path that involves writing or reading data. Calls to initiate such activities may leverage HBase master and Zookeeper for any meta-data assertions and meta-data gathering, before they initiate actual I/O.

3.7 Zookeeper Dependency

Zookeeper dependency is briefly highlighted in the previous section. A failure within zookeeper ensemble that HBase service depends on will immediately cascade into an offline HBase service. It may not hamper active I/O phase for clients already in active pursuit with connected region servers, but new requests will go un-answered. Hence planning for resiliency is essential in your cluster sizing, especially for Zookeeper!

HBase uses Zookeeper for coordination and meta-data persistence and discovery. The appendix – Zookeeper Quick Preview provides a comprehensive insight to Zookeeper ensemble runtime. Data inside Zookeeper is preserved in a hierarchical fashion with each node in the hierarchy called as a ZNode. The name of the root folder that HBase uses to maintain its meta-data is either /hbase/unsecure (in case of a unsecure cluster setup) or /hbase/secure! Under these folders you can find information related to regions in transition (i.e., those that are in split process), replication, master HA, etc.

3.8 I/O Paths

HBase scales very well at random writes and reads. That is, when your data set can achieve optimal physical distribution across the available region servers. This is the basis for its low-latent I/O guarantee. In this section will understand the I/O paths through which data movement occurs, from the client to HBase API and within HBase sub-features i.e., between region servers and between region server to the master and vice-versa. The basic goal is to learn enough to gather pointers that will help realize the thresholds of runtime operations.

3.8.1 Write – Streaming feed of edits

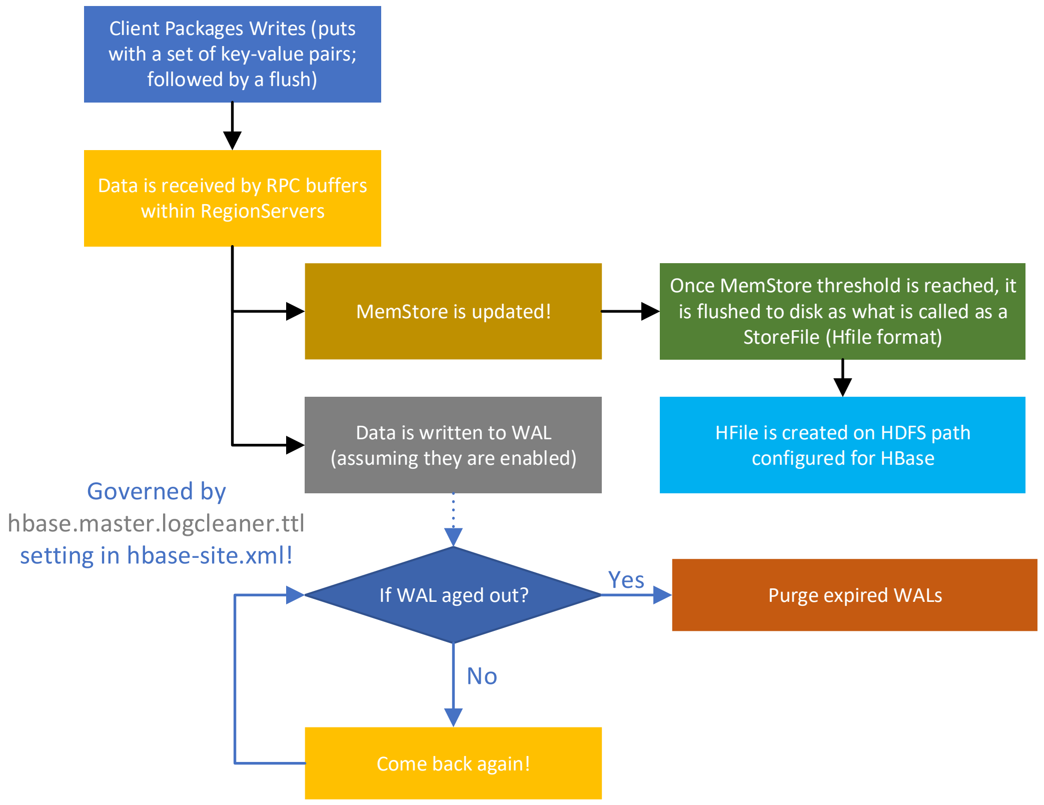

Client Packages Writes (puts with a set of key-value pairs; followed by a flush)
Data is received by RPC buffers within RegionServers
MemStore is updated!
Once MemStore threshold is reached, it is flushed to disk as what is called as a StoreFile (Hfile format)
HFile is created on HDFS path configured for HBase
Data is written to WAL (assuming they are enabled)
Governed by hbase.master.logcleaner.ttl setting in hbase-site.xml!
If WAL aged out?
Yes
Purge expired WALs
No
Come back again!

Figure 8: Write - Streaming Feeds (regular path)

Above visual provides a typical flow of events that happen when a connected client attempts write data (i.e., key-value pairs packaged as puts) to HBase cluster.

Please review following essential points that apply to this approach, where a write is managed via *RPC Handler* → *Memstore* → *At-rest*, where edits coming into a memstore are also registered with configured WAL (write ahead logs) if enabled:

- WAL management is an independent process and is a forked-out path from the handler that is servicing the in-bound edits.

- Once a memstore threshold is reached, RegionServer initiates a flush operation that generates a HFile on the underlying HDFS-compliant storage layer.
    - If your primary storage layer choice is WASB, then the appropriate file reference is a storage blob, within your cluster's assigned blob container.
    - If your primary storage layer choice is either HDFS or ADLS, then the file is split into a sequence of blocks.
    - Data that belongs to similar keys i.e., keys with a common prefix is usually found in sequential blocks or blob entries depending on your storage choice.
    - This will help better reads for closely associated data. On a wider range the reads however are split between all participating region servers (assuming you have a good data distribution)

3.8.2 Write – Bulk load process

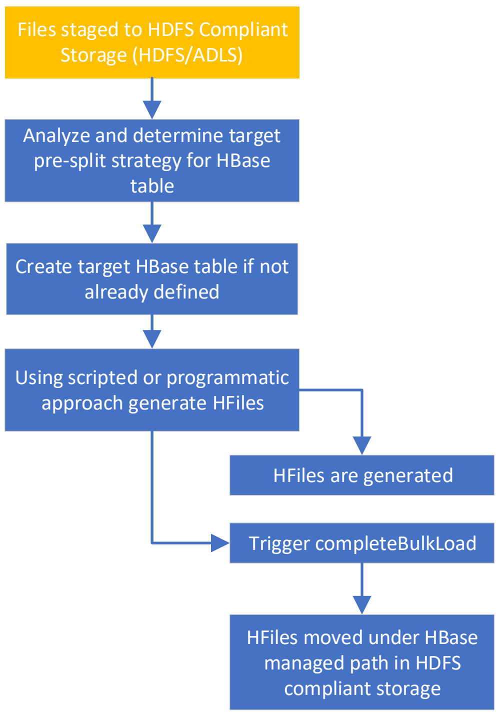

Files staged to HDFS Compliant Storage (HDFS/ADLS)
Analyze and determine target pre-split strategy for HBase table
Create target HBase table if not already defined
Using scripted or programmatic approach generate HFiles
HFiles are generated
Trigger completeBulkLoad
HFiles moved under HBase managed path in HDFS compliant storage

Figure 9: Write Path - Using bulk-loader

Few points to review:

- As you see, the bulk load process completely by-passes the CPU & Memory bound write path that is the default approach to stream continuous edits. Such edits could represent contiguous data points or completely random sequence of key-value pairs.
- Bulk-load process is employed to push large volumes of data efficiently and in short amount of time, without having to get constrained by memory and CPU allocations on your region servers. Note – at the time of HFile generation process.

- Bulk load is a two-step process - generate HFiles, then move them under HDFS path that is managed or monitored by HBase.
- It is a recommended that you disable major compactions on your cluster, or that fall back to a default disabled mode and manage compactions on individual tables following your workload patterns.
    - The later will ensure any cascading negative effects from compactions does not cluster outage.
    - Note – do not leave your cluster for ever in a non-compacted mode. This will lead to region server hotspots and eventual failures taking few or all your regions offline.
- For additional information check-out about the manual HBase shell command - \`major_compact\`.
- Note about using ADLS and assuming you can pack more storage per region server. While it is technically possible to pack as much data by region server at the time of write, specially with WASB or ADLS as your storage choice, configurations that effect the read-path can suffer from too much data per available compute resource that your region server can be allocated.
    - See the section "*Rough scale math for sizing ingress!*", for additional information.
- The final step in the process \`*completeBulkLoad*\` is all about moving the generated HFiles from your temporary workspace used by the generation process, to under the path that is managed by HBase.
    - If your generation process is on the same cluster's edge node, the files will be moved to the HBase monitored path.
    - However, if you run your generation process from another cluster's edge node or compute nodes, the \`*completeBulkLoad*\` will copy the files to the primary storage path that supports your target HBase cluster. Ensure you clean up these files, or have a clean-up frequency, for example – clean-up once your cluster is stabilized. Else, you will have to replay the whole sequence starting with your source.

3.8.3 Read Path

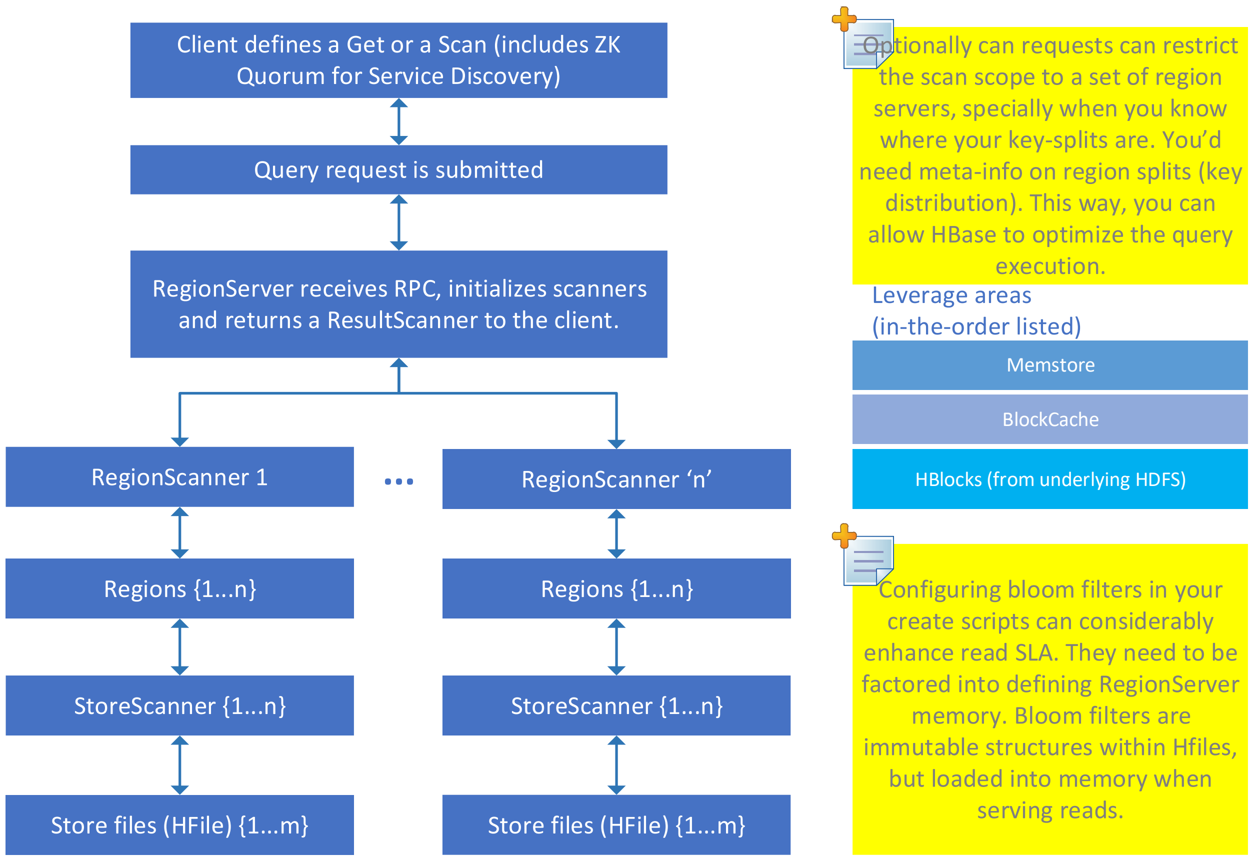

Client defines a Get or a Scan (includes ZK Quorum for Service Discovery)
Query request is submitted
RegionServer receives RPC, initializes scanners and returns a ResultScanner to the client.
RegionScanner 1
...
RegionScanner 'n'
Regions {1...n}
Regions {1...n}
StoreScanner {1...n}
StoreScanner {1...n}
Store files (HFile) {1...m}
Store files (HFile) {1...m}
Optionally can requests can restrict the scan scope to a set of region servers, specially when you know where your key-splits are. You'd need meta-info on region splits (key distribution). This way, you can allow HBase to optimize the query execution.
Leverage areas (in-the-order listed)
Memstore
BlockCache
HBlocks (from underlying HDFS)
Configuring bloom filters in your create scripts can considerably enhance read SLA. They need to be factored into defining RegionServer memory. Bloom filters are immutable structures within Hfiles, but loaded into memory when serving reads.

Figure 10: Read Path

Read path significantly differs from the write path implementations. The only dependency that it will have is the shared set of available compute and memory resources, besides the storage layer. Hence, it is important to ensure you have enough percentage of compute and memory spared to address read demands. Specially when you have a hybrid scenario – write and read heavy workloads.

Few recommendations when framing your queries:

- Ensure you always set a limit, specially if you are trying to assert basic conditions that have potential for triggering a full-table scan.
    - A full-table scan will delegate the scan task to all the available region servers that are currently hosting regions that belong to your table in the query statement.
- Consider having query predicate that can help narrow down the data volume that gets loaded and shipped to your client.
    - Leverage the filters and co-processors wherever applicable. It will help optimize inflight data volume and optimize your scan performance.

- If you most of your queries focus on data recency and do not require older data, then you can balance memstore size allocations to retain most recent edits for the duration that satisfies your recency bound queries.
    - This will help avoid costly disk-read.
    - Also, enable bloom-filters at the time of DDL execution.
    - The read path will consider keys from memstore, cached store-file indices (leverages bloom-filters if configured).

4 Strategizing for Capacity

Just to refresh, HBase is a very good big data store choice when you have random (i.e., ad-hoc) workloads, where consistency is your first preference and you have high-volume data sets to support i.e., in the orders of billions of rows and millions of columns. It relies on HDFS compliant data storage layers such as HDFS, WASB or ADLS. The eco-system is matured enough to support standard SQL capabilities. Also, we reviewed the significance a row-key design can have on your implementation and overall performance yield.

In this section will cover essential planning aspects to help you strategize for an optimal cluster capacity.

4.1 Quantitative References

Quantitative references offer key insights about your workload demands as they exist in the problem domain or functional context. For example - data-patterns to support, duration to maintain data in the cluster, concurrent workloads to support, etc. Following are few recommendations to gather such references:

- *Workload Name* – canonical name given to a data flow use case. Example – *Daily Financials*.
- *Type of workload* – write-heavy, read-heavy or a mixed workload
- *Ingress - Volumetric Information* - depending on highest volume sizes you have, this can be in the order of MB, GB, TB or PB. Consider raw size of a single copy.
    - *One-time Initial Load* – the very first snapshot of data from your source that will be migrated to HBase. Example, 100TB.
    - *Incremental Volume* – size of data set that each incremental load will add to existing data set in your target HBase. Example, 5TB daily incremental load.
    - *Time-to-live (retention period)* – duration for which the data set will remain active your data store, before it can either be archived or purged post its time to live setting. Unit of measure is milliseconds. For example, you can come up with a number like 604800000 milliseconds, which equates to a duration of 7 days. Post this time your data becomes eligible for removal.
    - *Write Concurrency* – this metric should represent just-in-time, random-write scenarios where clients are streaming writes to your cluster. Note - You can discount bulk-load job concurrency from this number. It is best captured along with incremental volumetric information.
- *Egress – Volumetric Information* – this gives a picture of data serving demands the targeted cluster is expected to satisfy.

    - *Expected Recency Duration* – How far into history does your queries hit the data? For example, will your most frequent or top queries hit data set from last hour, etc.
    - *Data Volume (guestimate) to satisfy recency* – Based on your recency duration, come up with a number that represents data volume size that ideally should be less than the overall data foot-print. For example, you can come up with accrued data size of 128MB of most recent data volume. At-rest data can still run into other higher-order numbers such as 10s or 100s of GB or more. Or, even in terms of TB or PB.
    - *Concurrency factor* – this represents approximate count of distinct clients that connect to your cluster to run some form of query. Such clients can be your downstream transformation jobs, example Spark or an API back-end that offers data as a service.
- *Data Fail-over strategy* – instantaneous i.e., faster, or something that can lend some leeway in terms of recovery time.
- *Data Model Insights*
    - *Key cardinality* – a list of keys to # of records mapped to that key. Will help understand if your source data has fair distribution of data across keys or does it follow a long-tail situation, where most percentage of data is concentrated around few keys, while most keys have very feeble count.
    - *Key composition* – keys must be composed with an attempt to achieve more projection and avoid creating redundant copies of the same dataset for a different key composition. Unless, your use cases are drastically different and need some physical isolation. Key composition must be reviewed alongside key cardinality.
- *Query Predicates and expected SLA* – We want to serve data faster, by ensuring it is consistent. For each combination you can come up with expected SLA that satisfies baseline needs for both speed and consistency.

4.2 Approaching Table Designs (i.e., physical models)

Data modeling focuses on following key aspects of your table design:

- *Row-key* – key to access a logical record from HBase.
- *Column Families* – logical grouping of your table attributes.
    - Derive column families based on your access patterns.
    - Do not define column families for each of pattern, that will lead to inappropriate data footprint as well have your reads suffer.
    - Common access patterns across your attributes will help you in defining optimal number of column families.
    - In most scenarios, having a single column family is good enough. Unless, your access patterns are distinct in their own sense with no common attribute between them.

- *Versions* – HBase allows you to version your data, at row-key level.
    - As you recall from the HBase data model, each key-value pair is associated with a timestamp. The field also signifies version of the entry in the table. You can configure your table semantics to have min/max versions at column family level.
    - From a storage standpoint, key-value pairs with common row-keys are packed together, including different versions of the same row-key.
        - This can bloat your store files at rest (i.e., on DFS). If percentage of queries that focus on versions is low, it is recommended to not expand on this versioning feature.
        - Instead, if a date predicate is involved in such queries, you can suffix such value into your row-key itself and use a wild-card scan.
    - Timestamp values can be leveraged to implement time range queries without having to define additional columns. At the time of defining your puts (i.e., write) you can configure appropriate timestamp value from your source.
    - Only caveat to remember here is not all HBase integrations yet support mapping to HBase timestamp value. For example, Hive-HBase integration. Check this JIRA – HIVE-2306.
- *Schema-on-Read* – HBase meta store does not maintain or either provide assertions for column names.
    - A schema is established only at the time of write.
    - The meta-information that is preserved with your table definition is limited to the namespace that your table belongs to, list of column families and configurations specific each of those column families.
    - At the time of processing your query request, HBase query engine will simply ignore columns that do not exist in first place.
    - Also, it is not necessary that columns in your functional data model (i.e., record) must exist on each row.
    - Other SQL capabilities such as referential integrity is not supported out of the box.
    - You can check SQL semantics provided by framework that work with HBase as their underlying data store and try to extend such capabilities. Example, Phoenix or Hive integrations.
    - HBase is a not an ACID compliant datastore, although it does guarantee certain specific properties. Check this link for a quick insight.
- *Randomized writes* – HBase is a distributed data store that scales efficiently when you can spread your I/O across. Doesn't necessarily mean spread thin. It is important to validate your row-key against kind of data distribution you can achieve, with enough keys mapped to the splits that you can define.
    - Watch for monotonicity in your keys.

        - That is keys with increasing order of their value, appearing to close to each other in your insertion order.
        - This can quickly develop region hotspots.
        - Another term to indicate you have long tail distributions that cause few regions to get hammered with too much of I/O or compute, while others tend to remain under-utilized.
- Scaling Out
    - A quick note on urge scale-out immediately by volume – when you need to scale-out or provision for certain volume of data all three factors are key – CPU, Memory and Storage.
    - Observe the growth of number of regions your tables have generated and see if there can be more to be done to adjust load between existing set of regions.
    - One more recommendation before you consider scaling out is to look at legitimacy of all the data you currently have. Instead if there can be some data cleanup to be done, take your table offline i.e., into maintenance mode and apply major compaction. Ensure your cluster's CPU and Memory have enough head room, as you may have other kinds of tables and workloads.
    - As a remediation attempt, you can also consider re-splitting existing table regions.
    - ADLS scales alongside your storage requirements. However, you must balance with fewer regions per region server i.e., allowing more keys per region split to stay easy on memory constraint.
    - Newly added regions will help capture future edits for the keys that they represent. This does not necessarily rebalance current at-rest footprint. That requires rebalancing measures such as region re-splitting and running major compactions.
    - When you run major compactions, ensure your table is taken offline for maintenance such that it will not impact external I/O performance.
        - That may complicate things depending on available headroom on both CPU and memory.
- Logical Data Patterns to observe
    - HBase is good at managing billions of rows by millions of columns as a sparse matrix. There are tools that extend this capability to offer SQL semantics.
    - Identify the primary objective of your HBase Cluster. For example, is it setup to satisfy Data Warehouse and Business Intelligence, exploratory tasks like Data Science, or to offer a data foundation that delivers Data as a Service via front-ending API.
    - This will help approaching your row-key design and application preferences of available HBase features such as filters, out-of-the-box sort orders, range queries, random queries that do not focus on row-key alone, implicit publication of generated insights to other down streams by application of co-processors, etc.

4.3 Configuration Essentials

Having realized the architecture and data modeling aspects, it is time to look at few configuration parameters for each of the constituents that you need to configure and manage for a healthy HBase cluster. This section will focus on these aspects. Consider them as starters. Once you get more familiar with your workloads and HBase in general, you can experiment with new set of permutations and combinations. This will help you to also derive baseline settings customized to your environment, workloads and usage. Will start with how to set expectations for any kind of cluster sizing that you'd like to begin with.

4.3.1 Balancing the architecture for optimal runtime!

Balancing the architecture for optimal throughput must be perceived from a data-driven angle and inching towards capacity modeling exercise. While data tuning or model refinement is an iterative process with dependency on domain knowledge, from a systems perspective you can start by factoring three key parameters – Consistency, Availability and Partition tolerance, considering it's a distributed system setup.

The classic CAP theorem (also known as the Brewer's theorem) states that any given distribution system can meet at-most two of these criteria, but not all. HBase's basic design principle prioritizes on consistency and partition-tolerance, but compromises on availability. This does not mean to say you cannot access your data. Data is safe, as all writes are consistent. In rare situations when a region server goes offline before it replicates writes that it received, one need to wait while the node is recovered, and edits get replicated in the process of recovery (handled by Region Server's internal processes). Hence, the recommendation to not disable WALs, instead tune their TTL (time to live).

Before we move further, let us understand each of the CAP aspects in principle and as they are applied in case of HBase:

- *Consistency* – All writes to a HBase cluster are consistent, ensuring your data is safe (replicated by configured replication factor). This allows your clients to have consistent view over your data sets. Dirty writes are never exposed to client.
- *Partition-tolerance* – this is about ability to sustain network or hardware failures. Unless there is a total outage, when partial failures effect certain nodes, other nodes can continue to serve data or read requests.
    - Note that because consistency is a guaranteed to preserve copies of your data, partition tolerance is automagically met by HBase.
    - One must watch for the extent of these failures and tune replication factor, such that you have enough copies on remaining nodes to satisfy your reads.

- *Availability* – If a receiving node fails before you receive an ack on your request, most likely it did not also complete its write path sequence. Such as flushing MemStore entries to the underlying storage (HDFS/WASB/ADLS). This is where enabling WAL feature becomes a critical choice. If the failure is limited to system process and that you could revive the failed node, you can replay these WALs and allow HBase to stabilize the data footprint. The stabilization comes through replication that occurs when WAL replays kick-in.

4.3.2 Resource Allocations – Quick Tips

You need to balance between CPU, Memory and Storage for each of the constituent services – Zookeeper, HBase Master, HBase RegionServers, YARN and HDFS (including HDFS compliant DFS services such as WASB or ADLS). To continue to serve I/O in a very low-latent manner, HBase hungers for more compute power – CPU and Memory. With one important note – increasing memory is not the first remediation step. It would only complicate things further triggering events like long-GC pauses more often than needed.

Employing choices like ADLS as your primary storage option will ease concerns about scaling storage as a factored value. However, as your storage continues to increase per mapped Region Server, your region server will tend to increase its demand for more compute power. To refresh our understanding, data in HBase is mapped by key-ranges (a.k.a., splits) where each range mapped to a region, which internally mapped to one or more store files. These store files are further mapped to individual HDFS blocks.

Consider a write path scenario – a write is first persisted to the MemStore(s). RegionServer will

Solving a write scenario can have adverse effects on your read path scenarios. Note, the same CPU and Memory are also shared by your read path. Like a MemStore (which is useful if your query predicates point to more recent data), HBase RegionServer has another similar cache or buffer zone to support reads – BlockCache. BlockCache is used by RegionServer to load store-file indices, bloom-filters, etc. Unlike MemStore, HBase allows you to configure memory requirements for BlockCache using off-heap memory space. Though this space is not accounted from your JVM heap, it will occupy the same VM's memory resource. So, you need to balance these allocations as an all-inclusive sum-total requirement rather than treating them as mutually exclusive items.

Azure stack offers choices like ADLS to support primary storage dependencies for your HDInsight clusters. Check about ADLS and HDInsight cluster pricing details for more specific inputs. Also, checkout some specifics in the CPU and Memory section below, that is part of the Key Configuration Parameters section.

5 Key Configuration Parameters

It is important to note that each constituent service component is designed and implemented for scale, hence comes along with a wide number of configuration parameter choices. This section will focus on first few starter parameters. Whenever there is a parameter change, specially a change to the memory split percentage or increase in number of RPC listeners/handlers, it is always recommended to try out the parameters using sample workloads, and if you intend to do so in production, factor in current cluster workload. Making random changes to production environments can cause negative impact that can quickly cascade all-over. For example, if for a small memory tweak, it may cascade into buildup of too many store files. Usually minor compactions are good at handling such situations, but in rare scenarios they can also morph into major compactions. These are very resource hungry processes.

Usually, the default out-of-the-box settings are a good start with the exception they may not match the volume expectations, but should give you a good initial run to try your workload patterns. This will be a handy tip to consider, especially in scenarios where you do not have near accurate i.e., ballpark range of your workload numbers.

Let’s check some specifics now!

5.1 CPU and Memory

Azure stack for HDInsight offers a range of infrastructure choices. A quick note on memory available versus that you can actually allocate – you can start with 2GB and upwards on head nodes (i.e., HBase Master). Try not to go beyond 8GB as more heap essentially refers to long GC-pauses by JVM. On worker nodes you can go higher than 8GB but consider using the increased reservation for off-heap allocations. Read structures such as BlockCache can benefit from this extra space and you can avoid GC influence on such space utilization. It means your reads do not need to suffer GC pause cycles as much it can with heap-based caching. As you tend to exhaust on memory, you’d experience scenarios like OOME or more frequent GC cycles. This should be considered as a wake-up call to consider cluster scale-out.

Note – at any point you consider scaling out i.e., adding more region servers, pre-empt any data-driven challenges first. Example, re-splitting existing regions, checking your row-key composition for any adverse impact on data distribution, running major compactions on specific tables, etc.

Following table provides suggested baseline choices that you can consider for each kind of service (said that, you can explore other available options as well and experiment with them):

Service	Node Type Recommendation	Baseline Recommendation
Zookeeper	Memory Optimized Nodes for HDInsight	E4 V3 – 4 vCPU, 32GB RAM
HBase Master (HA-enabled)	Memory Optimized Nodes for HDInsight	E4 V3 – 4 vCPU, 32 GB RAM
HBase RegionServers	Memory Optimized Nodes for HDInsight	E8 V3 – 8 vCPU, 64 GB RAM

Table 1: Compute Resource Recommendations

5.2 JVM Essentials

JVM (Java Virtual Machine) configuration is focused on two key aspects – heap settings and settings to tune GC (Garbage Collection) activity. Heap settings include minimum heap to begin with and limits to which JVM can eventually increase heap space (i.e., maximum heap). Garbage Collection settings start by identifying the kind of GC-collector algorithm the JVM must implement for your runtime and all other associated tuning parameters. The out-of-the-box configuration comes with Concurrent-Mark-Sweep collector.

CMS is a generational collector that attempts to perform frequent cleanup by running one or more threads to clean-up invalidated references parallel to your application threads, without causing impact to your overall throughput. When a JVM is initialized with CMS as the chosen GC collector, it organizes heap space into buckets – young generation, tenured spaces (2 of them) and old generation. As objects are initialized, they are allocated space in young generation. As they age, the objects are moved to tenured space (1st one and then to the 2nd one) if they are actively referenced in your application threads. Further they are moved into old generation.

Garbage collection occurs in two cycles – minor collections and major collections. GC is triggered in two scenario – at a configured occupancy threshold or when the thresholds on individual buckets are reached. Reason we have GC in first place, is to allow JVM to create space for new objects such that your applications can continue to perform. For a healthy application it is hence necessary to pay attention to life-time value of your objects and preferably have them created to thread-local such that your use of allocated memory is efficient. This best practice is followed by these frameworks like HBase, as a first order design principle. Things can still happen based on your current data patterns. Which is where this step of tuning GC comes in for any active application instance.

Following is an example process information (for HBase Master) for quick reference:

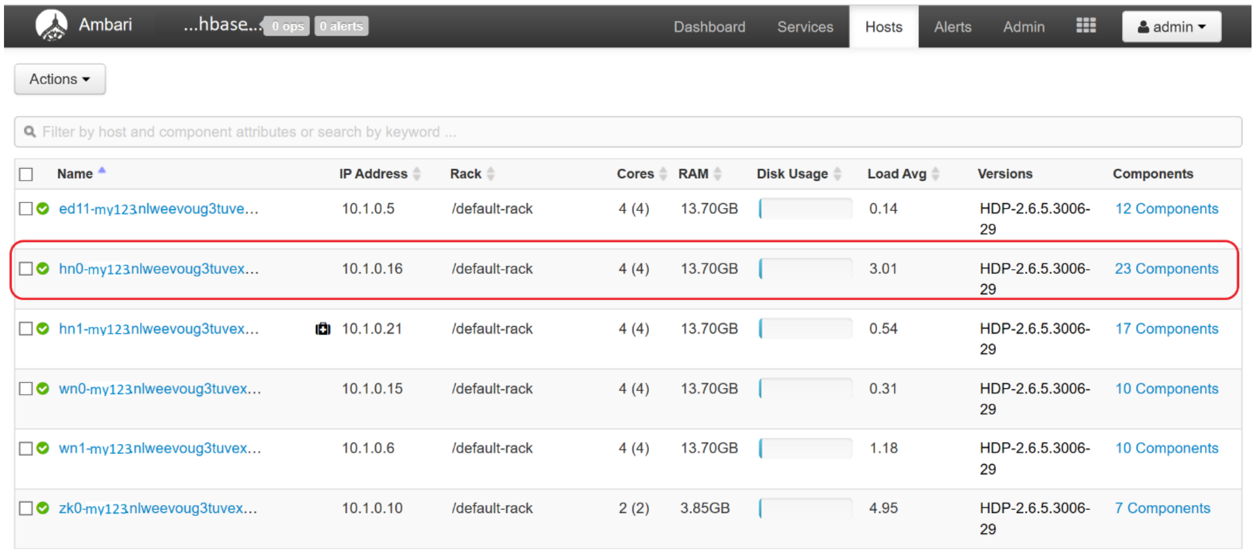
```
ams      16139 16125  3 06:33 ?        00:34:48 /usr/lib/jvm/java-8-openjdk-amd64/bin/java -
Dproc_master -XX:OnOutOfMemoryError=kill -9 %p -XX:+UseConcMarkSweepGC -
XX:ErrorFile=/var/log/ambari-metrics-collector/hs_err_pid%p.log -Djava.io.tmpdir=/var/lib/ambari-
metrics-collector/hbase-tmp -Djava.library.path=/usr/lib/ams-hbase/lib/hadoop-native/ -verbose:gc
-XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:/var/log/ambari-metrics-collector/gc.log-
201903120633 -Xms2048m -Xmx2048m -Xmn256m -XX:CMSInitiatingOccupancyFraction=70 -
XX:+UseCMSInitiatingOccupancyOnly -Dhbase.log.dir=/var/log/ambari-metrics-collector -
Dhbase.log.file=hbase-ams-master-hn0-customer.log -Dhbase.home.dir=/usr/lib/ams-hbase/bin/.. -
Dhbase.id.str=ams -Dhbase.root.logger=INFO,RFA,ETW,FilterLog -Dhbase.security.logger=INFO,RFAS
org.apache.hadoop.hbase.master.HMaster start
```

As you may observe in the above process extract, CMS is the configured collector algorithm on the JVM instance (for HBase master process). GC logs can be a bit verbose and can quickly fill-up the node storage. Consider turning these off if you don't need such verboseness. You can still accrue metrics that Ambari tends to collect from each of these managed services. Verbose GC logs are helpful to dig deeper into JVM internals such how much space certain GC cycle can collect or how many objects are aging and the rate of such build-up, etc.

Essentially time that your JVM invests in GC cycles indicates non-performing time when application processes are paused to allow for more heap space. Hence developers writing the code or administrators tuning GC shoot for shorter GC pause cycles that are spaced out farther than more often. Tuning GC can at times become a re-entrant cycle, hence start by checking few important aspects and come back to tune it as you gather more statistics about your runtime.

Here are few such parameters:

- Garbage Collection Algorithm that the JVM should use: -XX:+UseConcMarkSweepGC
  - Will launch your JVM process by defining your heap into generational spaces and assigning CMS as the GC collector.
- Slicing available memory
  - Let's consider an example – say you have 8GB of available memory and 4 cores.
  - Factor in other processes on the node that also are initialized along with either Master or RegionServer. You can refer the details from Ambari portal's Hosts tab, as shown in the following visual:



Name	IP Address	Rack	Cores	RAM	Disk Usage	Load Avg	Versions	Components
ed11-my123nlweevoug3tuve...	10.1.0.5	/default-rack	4 (4)	13.70GB		0.14	HDP-2.6.5.3006-29	12 Components
hn0-my123nlweevoug3tuvex...	10.1.0.16	/default-rack	4 (4)	13.70GB		3.01	HDP-2.6.5.3006-29	23 Components
hn1-my123nlweevoug3tuvex...	10.1.0.21	/default-rack	4 (4)	13.70GB		0.54	HDP-2.6.5.3006-29	17 Components
wn0-my123nlweevoug3tuvex...	10.1.0.15	/default-rack	4 (4)	13.70GB		0.31	HDP-2.6.5.3006-29	10 Components
wn1-my123nlweevoug3tuvex...	10.1.0.6	/default-rack	4 (4)	13.70GB		1.18	HDP-2.6.5.3006-29	10 Components
zk0-my123nlweevoug3tuvex...	10.1.0.10	/default-rack	2 (2)	3.85GB		4.95	HDP-2.6.5.3006-29	7 Components

Figure 11: Ambari - Host Configuration Sample

- Number of component services that share the same compute resource pool on this node include:

• App Timeline Server • HBase Client • HCat Client • HDFS Client • History Server • Hive Client • Hive Metastore	• HiveServer2 • MapReduce2 Client • Metrics Collector • Grafana • Metrics Monitor • NameNode • Oozie Client • Oozie Server	• Pig Client • ResourceManager • Sqoop Client • Tez Client • WebHCat Server • YARN Client • Zeppelin Notebook • ZKFailoverController

Table 2: Collocated Services Sample

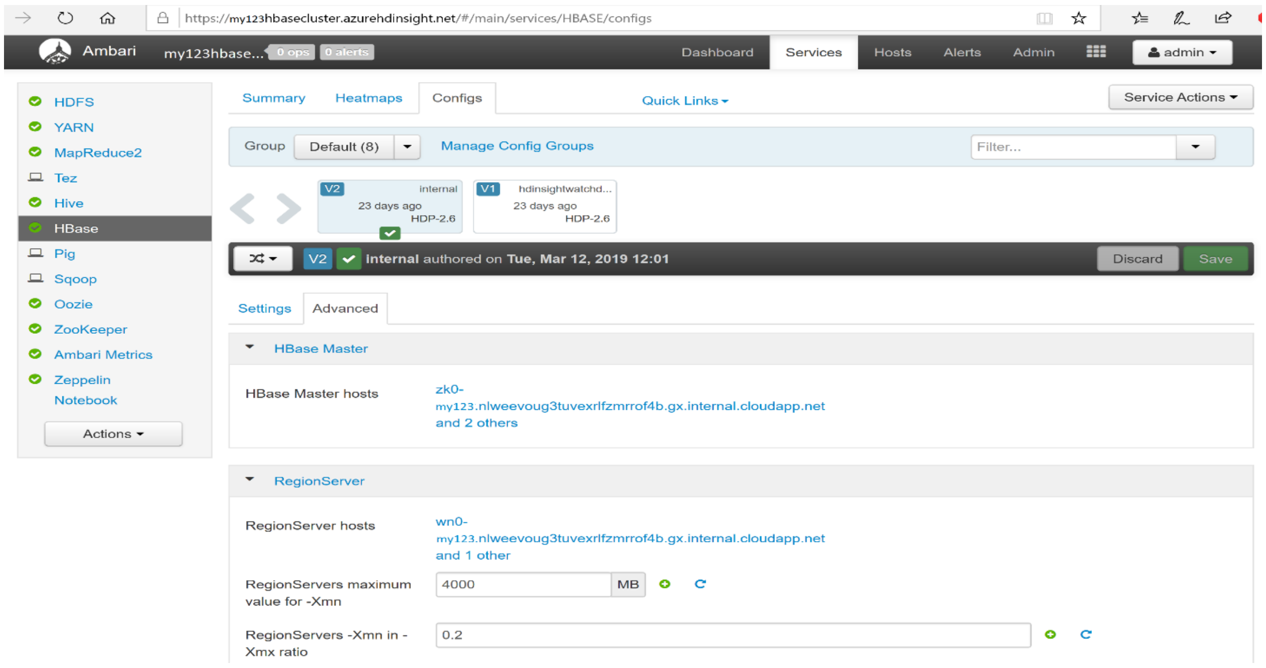
- Now, on a HBase Master node you do not need services like Oozie, Zeppelin and even services like Hive (if you do like to strictly restrict the node to HBase master and its immediate interdependencies). This will allow you ensure you are not in a run-away situation constrained by available memory on the node to satisfy and support HBase service.
  - It also depends on how much actual memory that these other services would use. Factor those numbers as well for subsequent steps below.
  - HDInsight packs these various services as part of its vanilla setup, which you can tweak according to your target architecture and workload patterns.
- Decide how much you can slice for HBase – say topside 6GB (with 2GB left for any spikes or headroom). Consider following combinations:

        - ***-Xms and -Xmx*** to 4GB (that is no intermediate pauses when your JVM needs to increase heap; it's given a decent heap size, unlike 2GB that you find in the above process example).
        - **-Xmn** to 256MB or 512MB; this allows JVM to slice this much of allocated heap to maintain young generation (again, its equivalent of stating min/max of young generation size).
            - You can increase this to a higher number (in context) specially if you have too many short-lived objects or application threads causing more objects to be initialized. This will avoid such objects to be moved to tenured spaces, by when you may not need them. Will help avoid some redundant moves.
        - **-XX:+UseConcMarkSweepGC with -XX:+CMSIncrementalMode** – initializes your JVM with CMS as the garbage collector, with incremental mode enabled. Together these flags will ensure shorter GC pauses allowing your application to achieve more throughput.
            - Incremental mode sets of what is called a duty cycle that is executed between compaction cycle. It allows to chunk the amount of work to be done in collection phases. It is very useful when you are constrained by CPU or hit your vCPU thresholds. Something to try before considering adding more vCPU cores.
        - **-XX:CMSInitiatingOccupancyFraction** – fraction of tenured spaces which when filled will trigger a compaction cycle. For example, if this is set to 70, a compaction cycle is triggered at 70 percent tenured space occupancy.
        - **-XX:+UseCMSInitiatingOccupancyOnly** – CMS collector frequently traverses the object graph in your heap and collects statistics. Then uses those heuristics to drive further compaction cycles. You can avoid application of heuristics and instead run compactions when tenured spaces reach the set occupancy fraction.
    - You can also try the following command to check default settings and an exhaustive list of flags supported by install base JVM on your:

```
java -XX:+AggressiveOpts -XX:+UnlockDiagnosticVMOptions -XX:+UnlockExperimentalVMOptions -
XX:+PrintFlagsFinal -XX:+PrintFlagsWithComments -version
```

5.3 How to tweak JVM Parameters? An Example!

Here are some quick visual clues of how to tweak these parameters using Ambari.



HBase Master hosts: zk0-my123.nlweevoug3tuvexrlfzmrrof4b.gx.internal.cloudapp.net and 2 others

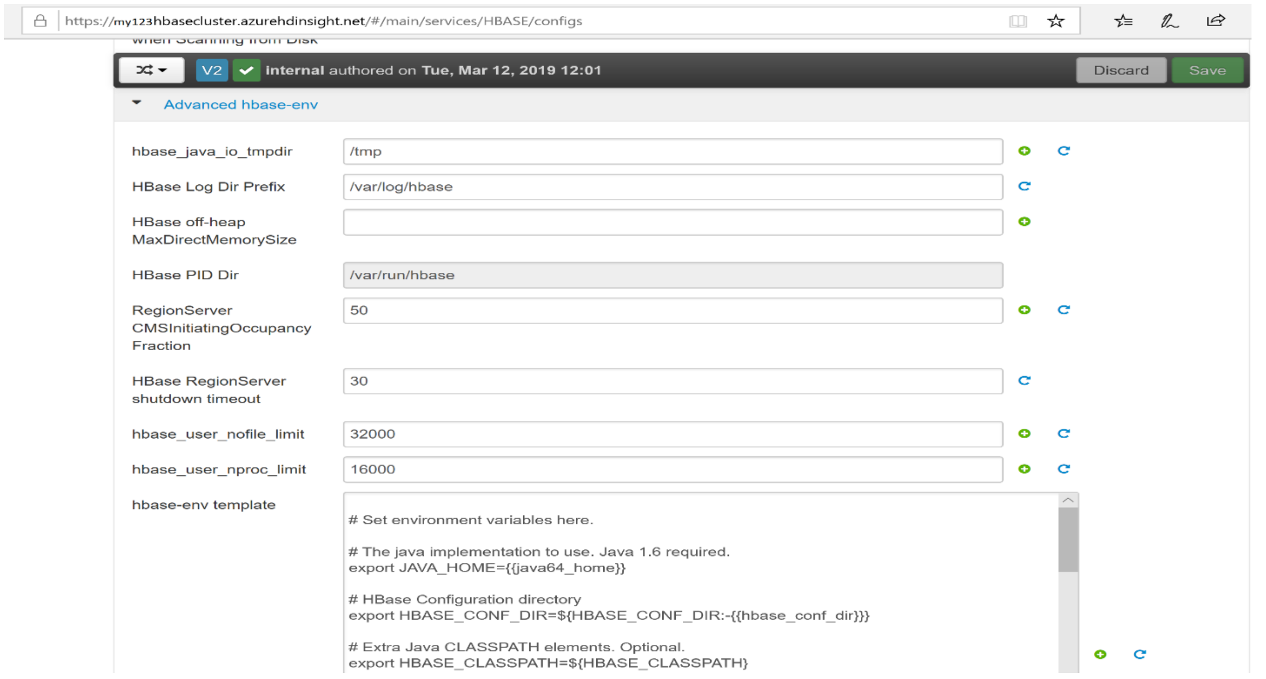
RegionServer hosts: wn0-my123.nlweevoug3tuvexrlfzmrrof4b.gx.internal.cloudapp.net and 1 other

RegionServers maximum value for -Xmn: 4000 MB

RegionServers -Xmn in -Xmx ratio: 0.2

Figure 12: Ambari HBase Master Configuration Tab

And,


https://my123hbasecluster.azurehdinsight.net/#/main/services/HBASE/configs
V2 internal authored on Tue, Mar 12, 2019 12:01
Discard
Save
Advanced hbase-env

hbase_java_io_tmpdir: /tmp

HBase Log Dir Prefix: /var/log/hbase

HBase off-heap MaxDirectMemorySize:

HBase PID Dir: /var/run/hbase

RegionServer CMSInitiatingOccupancy Fraction: 50

HBase RegionServer shutdown timeout: 30

hbase_user_nofile_limit: 32000

hbase_user_nproc_limit: 16000

hbase-env template:

```
# Set environment variables here.

# The java implementation to use. Java 1.6 required.
export JAVA_HOME={{java64_home}}

# HBase Configuration directory
export HBASE_CONF_DIR=${HBASE_CONF_DIR:-{{hbase_conf_dir}}}

# Extra Java CLASSPATH elements. Optional.
export HBASE_CLASSPATH=${HBASE_CLASSPATH}
```

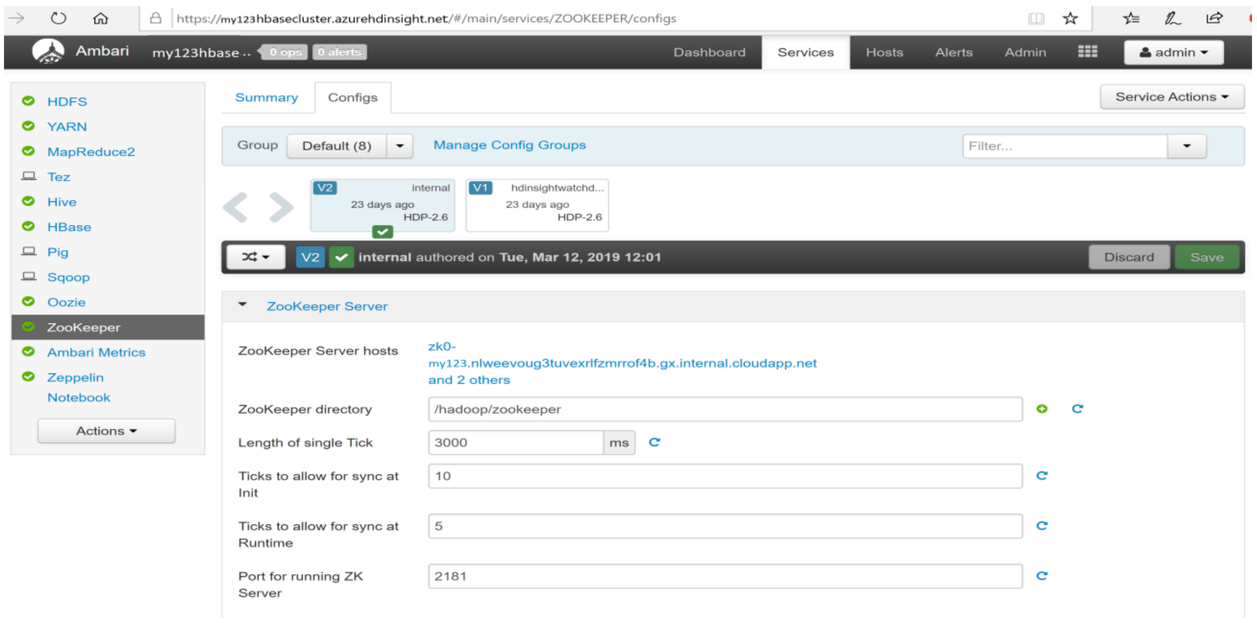
Figure 13: Ambari HBase-env.sh template

Extending further on hbase-env template:

```
# Set environment variables here.

# The java implementation to use. Java 1.6 required.
export JAVA_HOME={{java64_home}}

# HBase Configuration directory
export HBASE_CONF_DIR=${HBASE_CONF_DIR:-{{hbase_conf_dir}}}

# Extra Java CLASSPATH elements. Optional.
export HBASE_CLASSPATH=${HBASE_CLASSPATH}


# The maximum amount of heap to use, in MB. Default is 1000.
# export HBASE_HEAPSIZE=1000

# Extra Java runtime options.
# Below are what we set by default. May only work with SUN JVM.
# For more on why as well as other possible settings,
# see http://wiki.apache.org/hadoop/PerformanceTuning
export SERVER_GC_OPTS="-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps -
Xloggc:{{log_dir}}/gc.log-`date +'%Y%m%d%H%M'`"
# Uncomment below to enable java garbage collection logging.
# export HBASE_OPTS="$HBASE_OPTS -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps -
Xloggc:$HBASE_HOME/logs/gc-hbase.log"

# Uncomment and adjust to enable JMX exporting
# See jmxremote.password and jmxremote.access in $JRE_HOME/lib/management to configure remote
password access.
# More details at: http://java.sun.com/javase/6/docs/technotes/guides/management/agent.html
#
# export HBASE_JMX_BASE="-Dcom.sun.management.jmxremote.ssl=false -
Dcom.sun.management.jmxremote.authenticate=false"
# If you want to configure BucketCache, specify '-XX: MaxDirectMemorySize=' with proper direct
memory size
# export HBASE_THRIFT_OPTS="$HBASE_JMX_BASE -Dcom.sun.management.jmxremote.port=10103"
# export HBASE_ZOOKEEPER_OPTS="$HBASE_JMX_BASE -Dcom.sun.management.jmxremote.port=10104"

# File naming hosts on which HRegionServers will run. $HBASE_HOME/conf/regionservers by default.
export HBASE_REGIONSERVERS=${HBASE_CONF_DIR}/regionservers

# Extra ssh options. Empty by default.
# export HBASE_SSH_OPTS="-o ConnectTimeout=1 -o SendEnv=HBASE_CONF_DIR"

# Where log files are stored. $HBASE_HOME/logs by default.
export HBASE_LOG_DIR={{log_dir}}

# A string representing this instance of hbase. $USER by default.
# export HBASE_IDENT_STRING=$USER

# The scheduling priority for daemon processes. See 'man nice'.
# export HBASE_NICENESS=10

# The directory where pid files are stored. /tmp by default.
export HBASE_PID_DIR={{pid_dir}}

# Seconds to sleep between slave commands. Unset by default. This
# can be useful in large clusters, where, e.g., slave rsyncs can
# otherwise arrive faster than the master can service them.
# export HBASE_SLAVE_SLEEP=0.1

# Tell HBase whether it should manage it's own instance of Zookeeper or not.
export HBASE_MANAGES_ZK=false

{% if security_enabled %}
export HBASE_OPTS="$HBASE_OPTS -XX:+UseConcMarkSweepGC -XX:ErrorFile={{log_dir}}/hs_err_pid%p.log
-Djava.security.auth.login.config={{client_jaas_config_file}}"
export HBASE_MASTER_OPTS="$HBASE_MASTER_OPTS -Xmx{{master_heapsize}} -
Djava.security.auth.login.config={{master_jaas_config_file}} -Dwhitelist.filename=core-
whitelist.res,hbase-whitelist.res -Dcomponent=master"
export HBASE_REGIONSERVER_OPTS="$HBASE_REGIONSERVER_OPTS -Xmn{{regionserver_xmn_size}} -
XX:CMSInitiatingOccupancyFraction=70  -Xms{{regionserver_heapsize}} -Xmx{{regionserver_heapsize}}
{% if hbase_max_direct_memory_size %} -XX:MaxDirectMemorySize={{hbase_max_direct_memory_size}}m
{% endif %} -Djava.security.auth.login.config={{regionserver_jaas_config_file}} -
Dwhitelist.filename=core-whitelist.res,hbase-whitelist.res -Dcomponent=regionserver"
{% else %}
export HBASE_OPTS="$HBASE_OPTS -XX:+UseConcMarkSweepGC -
XX:ErrorFile={{log_dir}}/hs_err_pid%p.log"
export HBASE_MASTER_OPTS="$HBASE_MASTER_OPTS -Xmx{{master_heapsize}} -Dwhitelist.filename=core-
whitelist.res,hbase-whitelist.res -Dcomponent=master"
export HBASE_REGIONSERVER_OPTS="$HBASE_REGIONSERVER_OPTS -Xmn{{regionserver_xmn_size}} -
XX:CMSInitiatingOccupancyFraction=70  -Xms{{regionserver_heapsize}} -Xmx{{regionserver_heapsize}}
{% if hbase_max_direct_memory_size %} -XX:MaxDirectMemorySize={{hbase_max_direct_memory_size}}m
{% endif %} -Dwhitelist.filename=core-whitelist.res,hbase-whitelist.res -Dcomponent=regionserver"
{% endif %}
```

5.4 Zookeeper

Zookeeper depends on its configuration file `${install_base}/conf/zoo.cfg`. For example, the absolute path could be - /usr/hdp/current/zookeeper-server/conf/zoo.cfg. It is recommended to pursue configuration changes using Ambari. Following visual provides a sample interface and how to get there:



V2 internal authored on Tue, Mar 12, 2019 12:01

ZooKeeper Server hosts: zk0-my123.nlweevoug3tuvexrlfzmrrof4b.gx.internal.cloudapp.net and 2 others
ZooKeeper directory: /hadoop/zookeeper
Length of single Tick: 3000 ms
Ticks to allow for sync at Init: 10
Ticks to allow for sync at Runtime: 5
Port for running ZK Server: 2181

Figure 14: Ambari Zookeeper Configuration Tab

Here is a quick excerpt of entries as maintained in `zoo.cfg` file:

```
clientPort=2181
syncLimit=5
autopurge.purgeInterval=24
maxClientCnxns=300
dataDir=/hadoop/zookeeper
initLimit=10
maxSessionTimeout=120000
tickTime=3000
autopurge.snapRetainCount=30
server.1=zk0- my123 .nlweevoug3tuvexrlfzmrrof4b.gx.internal.cloudapp.net:2888:3888
server.2=zk1- my123 .nlweevoug3tuvexrlfzmrrof4b.gx.internal.cloudapp.net:2888:3888
server.3=zk2- my123 .nlweevoug3tuvexrlfzmrrof4b.gx.internal.cloudapp.net:2888:3888
```

Figure 15: Zoo.cfg Quick Excerpt

These are some of the typical configurations to be found in stock installations as well.

Note: For a complete list of configuration parameters recommend checking on official documentation here.

Following is a quick description of these common configuration parameters:

Parameter	Description
tickTime	Zookeeper depends on tickTime as its unit-of-time measure. This is derived from system timeclock represented in milliseconds. Example – 1000, 2000, etc. Other time related functions such as leader-follower synchronization, client time-outs, etc. are defined in tickTime units. Hence it is very important to have system clocks synchronized to NTP.
initLimit	During bootstrap, a zookeeper node must discover its leader or be elected as a leader and synchronize the state with the leader. As the data volume grows, this bootstrapping time may exceed the limit set here. Consider tweaking this in scenarios when you start to observe bootstrap failures. This parameter is essential in a distributed zookeeper setup.
syncLimit	Number of ticks that can go past before an acknowledgment to synchronize is received. This is an essential parameter in a distributed zookeeper setup. As you observe this value must be less than the \`initLimit\` setting. \`initLimit\` controls overall bootstrapping time, which also includes time allocation for the initial synchronization (which is governed by this value). You can check source code to see how the \`initLimit\` and \`syncLimit\` are validated.
autopurge.purgeInterval	Edits captured by Zookeeper leader are also preserved in transaction logs. Further once a transaction log threshold is reached, Zookeeper will take a snapshot and persist them. The purge interval will determine the frequency in hours that these logs can be purged.
autopurge.snapRetainCount	Indicates the number of older snapshots and corresponding transaction logs that must be preserved, from newest to oldest in the order they are created.
maxClientCnxns	Determines maximum number of connections a client can establish with a single member (i.e., zookeeper node) in the ensemble.
minSessionTimeOut	Clients when establishing a connect request to Zookeeper also indicate session timeout value. This requested time out value must be within the range of min/max set in Zookeeper configuration. This parameter determines the minimum session timeout value. If client specified value is lower than this, zookeeper server will override requested value and instead use the one from its configuration. See the source code here and here!

maxSessionTimeOut	This is the upper range of session timeout that a client can request. Must be considered along with minSessionTimeOut.
server.id	It is essential that all nodes in a zookeeper ensemble, know about all the other nodes in the ensemble. A list of such entries prefixed with server.<id> define the list of participating members in the ensemble. The value is represented in the format – host:listen_port:leader_election_port; the leader_election_port is required if electionAlg choice is either 1, 2 or 3. It is not required if electionAlg is set to 0.

Figure 16: Zookeeper Configurations

5.5 HBase in general

There are configuration parameters that apply common across master and region server nodes (VM's that host region server service).

Parameter	Description
hbase.tmp.dir	This can be considered as temporary work directory. Consider pointing to this a non-temporary location, if you'd like to preserve the entries that sustain a node restart. Remember when a node is restarted entries in /tmp are not retained. Default location is - /tmp/hbase-${user-name}, where user-name is set to hbase.
hbase.zookeeper.quorum	A comma-separated list of zookeeper servers that form a zookeeper ensemble (complete list) that HBase should depend on. Interactions will follow typical zookeeper service discovery process, i.e.., leader and to the followers. All edits go through leader and followers synchronize with the leader. Consider HBase yet another Zookeeper client.
zookeeper.recovery.retry.maxsleeptime	Maximum time in milliseconds to sleep, before retrying zookeeper operations.
hbase.local.dir	Local workspace that HBase can use. For example, when running compactions this space can be used to maintain temporary files.

zookeeper.znode.parent	The root path in Zookeeper where HBase initializes and manages its meta information. Example - /hbase-secure or /hbase-unsecure.
hbase.zookeeper.property.*	It is recommended to have a dedicated Zookeeper cluster in hybrid environments or in environments that experience heavy workloads. Besides, HBase can also host Zookeeper as a sub-process as part of its bootstrap sequence. These properties allow you pass custom parameters that map to zookeeper configurations. See hbase-default.xml for more info.

Table 3: HBase Commons Configurations

5.6 HBase Master

HBase master coordinate cluster house-keeping activities such as allocating pre-defined region pre-splits when defining a table, gathering heart-beat and meta-information from region servers, addressing fault-tolerant scenarios, etc.

Note: For brevity, ignored mention about ports, which are obvious to any distributed system. Please check the same from hbase-default.xml for additional detail.

Let us review few essential configuration parameters:

Parameter	Description
hbase.master.logcleaner.ttl	HBase during its runtime generates different kind of logs. HBase master maintains internal house-keeping routines to trigger log cleanup. When scheduled, the log cleaners check TTL expiry on such logs. One such logs is WAL or Write-ahead Logs that receive edits from clients (user-data). When a WAL entry lives past the TTL value configured here, it will be purged. Depending your scenario, you can extend or reduce the duration specified here. Unit of measure is milliseconds.
hbase.master.fileSplitTimeout	HBase master instructs region split tasks and works with RegionServers. This timeout indicates for how long the master would wait before throwing a region split exception and aborts the split process. Split process is non-blocking i.e., will not impact active writes. This is a background process.

Table 4: HBase Master Configurations

5.7 HBase RegionServer

Region servers are the work horses in a HBase cluster. Note - for brevity, ignored mention about ports, which are obvious to any distributed system. Please check the same from hbase-default.xml for additional detail. One rule of thumb to follow here is ensure you do not block writes or reads. At any time, you see contention in HBase performance, it is time to go to the drawing board, re-evaluate your workloads and data retention and determine scale-out factor. Subsequently you can try changing some of the parameters that directly influence such blocking aspects. Example, memstore configurations, etc.

Let's look at few essential configuration parameters that apply to region servers:

Parameter	Description
hbase.regionserver.handler.count	To support concurrency, each client's interaction is assigned a handler (i.e., RPC Listener). You can consider each of these handlers as threads. This value can be increased or decreased to rebalance load on your cluster. Note, a greater value here does not essentially mean increased throughput. At CPU level, thread mutexes will cause contention between more than one process, especially when load is high. Maintain this value as a multiple of CPU cores and observe overall CPU occupancy rate on region server nodes. Default is set to 30.
hbase.ipc.server.callqueue.*	Client requests are enqueued and processed in the order they are received (write or read). Call queue settings will influence how call queues are setup. For example, initialize RPC listener/handler specific call queue or share the queue across. Or to define some level of isolation between write queues and read queues. *Note*: • Do not start tweaking these parameters right away. • These are scenarios to be tried once you have level of quantitative understanding about your data flows and usage patterns. • Check description available in hbase-default.xml for more details on specific entries. Again, these are inter-dependent configuration parameters.

hbase.hregion.memstore.flush.size	Indicates the threshold at which a memstore flush occurs. This must be considered along with hbase.regionserver.global.memstore.* parameters. Say you have 128MB of configured memstore space and that you have writes coming in for two column families by a RegionServer. You can go up-to 64MB of flush size between two memstores, however note, if the global limit (i.e., aggregate is reached) then a flush is enforced on each of the memstores. Also note, the numbers are a hypothetical scenario of even distribution of capacity. This need not be a true possible picture and is more driven by your data semantics.
hbase.regionserver.global.memstore.size	Maximum size of all memstores in a region server before updates are blocked and flushes are forced. Writes are resumed until the overall size of memstores in a region server is reduced to hbase.regionserver.global.memstore.size.lower.limit. See explanation for the hbase.regionserver.memstore.flush.size parameter above.
hbase.regionserver.regionSplitLimit	As the number of regions grow per RegionServer, it puts pressure on memory allocations. This usually driven by write and read patterns. Note data is partitioned by at column family and allocated across regions, where regions are in turn distributed across available RegionServers. This is a *hint* to the RegionServers indicating that they should further region splits at this threshold. Note – region split tasks are initiated by master but delegated to the region servers that own the regions.

Table 5: HBase RegionServer Configurations

5.8 HBase Client

Like server-side configurations, one can also tune parameters that effect client experience. Here are few suggested parameters to review:

Parameter	Description
hbase.**client**.keyvalue.maxsize	Limits maximum size of individual key-value pair that can be submitted to HBase. This limit is overridden by likewise server-side parameter mentioned below. Default is set to 10mb
hbase.***server***.keyvalue.maxsize	This is very much like the client-side configuration setting to limit maximum size of a key-value that can be packed for write. Default is set to 10MB. Choose a value that meets the upper limit of your use cases or certain governing policy based max limit on size of values a client can submit to HBase. Setting to 0 will skip checking size of input key-value data. Size is a combined size value of both key and the value. Note – this parameter should be set on server-side hbase-site.xml for it to be honored.
hbase.client.write.buffer	To achieve more per call, client can buffer a batch of puts (i.e., a set of key-values). Size your data per logical record to avoid increased constraints on memory. Both on the client side as well on the server side. Server side consumption can be calculated using hbase.client.write.buffer * hbase.regionserver.handler.count. When buffering more per flush on client side, ensure your application logic can handle fail-over scenarios without having to lose the data.
hbase.**client**.scanner.max.result.size	Tune this parameter to adjust volume of data that your client intends to fetch per call to \`next()\` on your result scanner. Default set here is 2MB. Remember higher volume of data in-flight raises expectation for appropriate fault-tolerance measures on the client side (example, ability to efficiently process the results before fetching more).
hbase.***server***.scanner.max.result.size	Like the above setting, this setting controls the maximum that will be allowed by a Region Server.

	Useful setting to avoid potential out of memory exceptions. Ensure this setting is part your hbase-site.xml for it to be honored by the region servers.

Table 6: HBase Client Configurations

5.9 HDFS-complaint DFS

HDInsight distribution offers Azure Storage, Azure Data Lake Storage Gen1 and Azure Data Lake Storage Gen2 as storage options. All these choices offer level of parallelism and distributed behaviors expected from a HDFS compliant data store. Some notable differences can be observed in their cost models and the technical composition of integral components that carry out write and read operations.

It is important to understand the physical break-down i.e., at-rest representation of a HBase table. Following visual offers an empirical view of such break-down mapping:

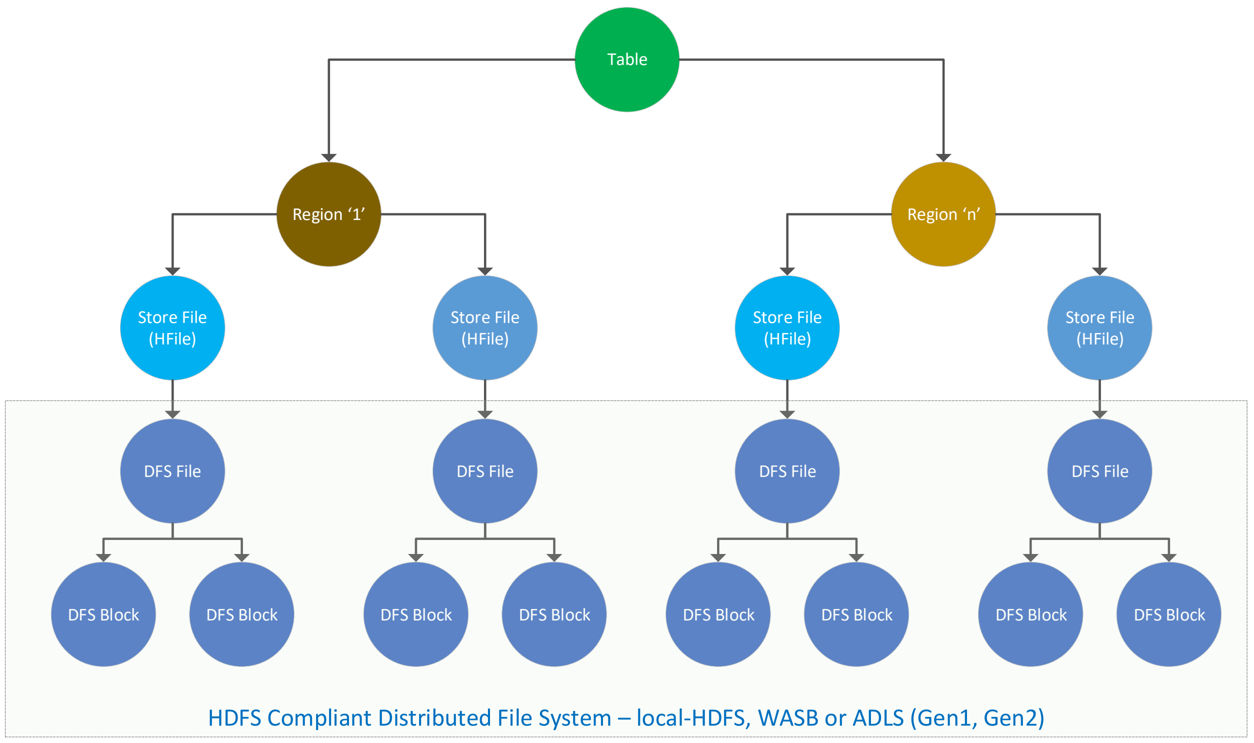


Figure 17: HBase-HDFS Dependency

Intent of this section is not to provide technical insight of these file systems, but to highlight critical paths that cross system memory & compute boundaries and spill into at-rest layers and vice-versa. The conversation with WASB or ADLS (Gen1 or Gen2) is not about maximum required physical storage space per node, rather, how much of it can be achieved by factoring certain memory and CPU capacity a selected worker node in your cluster can support. Let's infer the above visual further in detail:

- A table in HBase can be defined by a fixed set of region pre-splits (if you have a good handle on your data distribution and cardinality information) or by specifying maximum number of regions that a table can have. These are once again, hints to HBase.
- Either you define regions by pre-splits or by specifying a number, each region that represents certain key-range is assigned to one of the RegionServers as a primary owner. The replication factor will allow other RegionServers to also maintain copies of the data and aid in read performance. Writes however go to the primary owner of the region or the key-space that your inbound would map to.
    - So, the logic here more keys per key space. Hence, do not define too many region pre-splits or specify a greater number for number of regions (NUMREGIONS parameter in your DDL).
- A region is mapped to a store, which again maps to a HFile. The HFile is written in blocks (HDFS or HDFS-compliant blocks).
    - In case of a local storage mapped DFS implementation, you can have limit on maximum number of blocks that a single data node can accommodate.
    - With WASB or ADLS choice you do not need to be concerned about this limit.
    - Instead, you can choose to pack more data per region i.e., HFile.
    - Blocks are where you gain parallelism and replication.
    - Note that when you write data a memstore flush creates an HFile.
- Let's do some math. By default, memstore occupies 40% of your allocated heap. Check the configuration parameter – "hbase.regionserver.global.memstore.size", in your hbase-site.xml.
    - Memstore is per column family and this 40% is across all memstore instances per region server.
    - For example, consider you have 12GB of heap allocated to a Region Server. 40% of this amounts to ~4.8GB for memstores.
    - When this threshold is reached, a flush happens resulting a HFile creation. Assuming you have a single workload writing to a single column family, you could have a new HFile of size ~4.8GB on your file system, further split up into DFS blocks in sequence and persisted to the at-rest layer. Now this is without compression enabled.
    - But if you have configured your table with compression (recommend always) say, Snappy (that prioritizes compression speed over compression ratio) and assume you get about 40 to 50% of compression, the data footprint before it gets written to HDFS will come down to ~2.25GB (ball-park). And if you have your DFS block size configured to say 512MB you can expect roughly about ~5blocks.

Let's look at few configuration parameters (check hdfs-default.xml for the exhaustive set of parameters supported by HDFS) that matter the most:

Parameter	Description
dfs.datanode.handler.count	Number of server threads for data node. Tweak this number considering amount of available memory and CPU. Specially if you have other collocated services on the same node that also hosts a Datanode service.
dfs.replication	Replication factor to be maintained and applied for files that are created and managed on Hadoop DFS. Recommend minimum 3.
dfs.blocksize	Size of HDFS block. Consider your typical data volumes and patterns. Default is 128MB. You can even start from 256 as default and revise further.

Table 7: Key HDFS Configurations

6 Rough-scale math for sizing ingress!

Note – sections above argued to model workloads based on worst-case scenario. Reason is to emphasize upon on data patterns. In this section will take a best-case scenario to describe how numbers that you derive and goals to achieve even distribution can pan out in a cluster of your chosen capacity or that the capacity that you can aim for.

So, where do we stand considering a terabyte of ingress volume, with three region servers? Let's make some assumptions and do some math for a ball-park guestimate for an SLA i.e., time it will take a data ingress job to ingest a terabyte of data into HBase:

- Cluster capacity:
  - HBase Master HA enabled with a 3-quorum Zookeeper ensemble
  - 3 region server nodes with 8 cores and 12GB of memory allocation (you can go higher, but for now we'd like to keep GC or long-GC pauses to minimal). These are guestimate capacities on what is available when you choose a region server node when setting up a cluster.
  - Primary storage of choice – WASB or ADLS (recommend ADLS)
- Ingress-data volume

  - Uncompressed total data volume – 1 TB or 1099511627775 bytes.
  - Assuming even distribution, each node can go up to ~341GB or 366503875925 bytes.
- Let's walk through the break-down for the write-path scenario!
  - MemStore (hbase.regionserver.global.memstore.size) – 60% assuming you are setting up a write-heavy cluster. Intent here is more per write! There will still be reads all by themselves or in parallel.
  - Assume that attributes or columns in our source are being written to a common column family. Unit of abstraction that you can aim for. If you have more column families, then you'd multiply on memstores per column family, regions per column family and corresponding HFile numbers as well. So, let's go with 1 column family, hence one memstore that can occupy the whole of 60% heap allocation for memstore.
  - 60% of heap will be 12GB * 0.6 ~= 7.2GB; also represents a maximum size of a chunk of data that can be written to HBase before a memstore flush occurs.
  - Number of chunks that we can expect per configured region server can be ~341GB divided by 7.2GB or 47.36 or 48 chunks (remember we still need to push the tail-end smaller chunk).
  - Each chunk will create 7.2GB (uncompressed) or 2.88GB (snappy compressed assuming you'd get 40% of compression ratio). Note, data is sorted and compressed (if compression is enabled on the table) before it is persisted to HDFS or HDFS-compliant DFS.
  - Assume your DFS block size i.e., setting for "dfs.blocksize" is set to 128MB or 134217728bytes.
  - Each flushed HFile of 2.88GB is now split into ~23 blocks (2.88GB divided by 128MB).
  - Let us now assume a guestimate for throughput to write one chunk of user data to be 3 minutes, roughly speaking you are able to achieve 1GB/minute network I/O from your source to target. Give and take other dependent effects within cluster runtime.
  - For 48 chunks (that amount to ~341gb of our per-node distribution target), we are looking at 3 * 48 = 144minutes or ***2 hours and 40*** minutes per 1 terabyte of data ingestion using a 3 region nodes based HBase cluster. A quick word of caution, these numbers are hypothetical as mentioned in the note above. It is only to provide a context of how different pieces come together.
  - ***Tweaking these numbers*** – available memory, CPU speed, memory allocation for memstore, tuning on memstore flushes, etc., your actual numbers may be different. *Hence once you setup your cluster you can take some sample workloads*

that represent your typical spread, you can tune HBase as well set achievable goals on SLA and share a ball-park to involved stakeholders.

    - ***Note*** – more memory does not mean more throughput. More memory is good, when you consider using a good slice of it for off-heap block caches. This will certainly enhance your read throughput. Again, do not forget data distribution impacts.

- The approach here considered a typical path of a connected client trying to push data into HBase using the regular write path. The goal however is not about regular path or bulk load, but to emphasize the point that tuning HBase should always begin by factoring in CPU and Memory, rather storage as your first choice.

5 Wrapping up!

This document is quite verbose and having started with more theory, then walking through some key configuration choices and some math at the end, here is a quick set of things to revise as you strive towards an optimal HBase cluster setup:

- Always begin mapping your problem domain, gathering quantitative numbers that represent your data patterns, data, and workload demands.
- Capacity starts from memory & CPU then to storage. Not the other way around. Example, assuming ADLS elasticity you may quickly end up in a situation where your region servers are constrained by available memory and CPU.
- If constrained by CPU, most likely your throughput numbers tend to drop considerably. You may have chosen a higher count on RPC listeners i.e., concurrency per region server, but each of those listeners are now in contention for same compute power.
- Understanding nature of your workloads between read-heavy, write-heavy or mixed workload kinds.
- Retention policies – you can continue to achieve same initial numbers as your cluster tends to gather more at-rest footprint. It will become evident in your read performance numbers.
- Compress always, and never try the whole data set at once. Identify a logical sample size, try with that first and then expand to the rest. This will help you avoid any data issues that may have morphed into infrastructure issues.
- Too many regions exert pressure on memory resources. When adding new tables, consider implicit impact on existing tables and spread. This is one of the key reasons to think large across workloads.

- Finally, get your row-key right and remember to test for fail-over before considering your production is truly in production mode and capable of offering five-nine's availability.

6 Appendices

6.4 Appendix - Zookeeper – Quick Preview

As covered in the Zookeeper dependency section above, HBase maintains deep integration with Zookeeper. Zookeeper as you may know is a distributed coordination framework using which implementations can satisfy needs to address coordination, meta-information management, distributed locks, etc. This section intends to provide an overview of Zookeeper framework and its runtime. It will help in tuning Zookeeper, which becomes an implicit step towards tuning HBase.

6.4.1 Node Roles

Each node in a Zookeeper ensemble can take on two roles – *Leader* and/or *Follower*. Primary role of the leader is to serve coordination and write requests, the followers can share the responsibility to satisfy read requests. In case a client connects to a follower to write data to zookeeper, the write responsibility is delegated to the current leader. To enable high-availability in write-heavy situations, usually the quorum size is extended by +1 (besides 2n+1) and limit such additional node to deal with cluster coordination responsibilities. The node will not be available to satisfy client requests.

6.4.2 Quorum Resiliency

A replicated set of servers that are part of a cluster is referred to as a Quorum. The recommended starter size of a zookeeper ensemble (quorum) is a three-node cluster. This will help Zookeeper to remain available even in case of failures. One can determine the size of a cluster i.e., number of nodes to form a cluster, by applying the formula – 2n+1. Here, n is the max number of node failures the cluster can sustain and yet remain available to its clients. For example, a three-node cluster can sustain at-most one node failure before cascading affects take your cluster offline. At least, the potential for cascading failures becomes more evident in such scenarios.

Reason the cluster sizes are computed to 2n+1 is to allow other nodes to re-distribute and share the workloads (client-side fault-tolerance is expected however to be able to re-connect and continue). This will give some leeway for you to attempt node recovery and hopefully for you to re-instate a healthy cluster.

Quick note about leader re-election – Zookeeper ensemble cannot function without an active leader. Hence in case of node failures the remaining nodes participate in a consensus exercise to re-elect a new leader amongst the remaining nodes. Also, it is important to understand the failed node, if recovered does not take on the leader role immediately.

6.4.3 Data storage inside Zookeeper

Zookeeper maintains a hierarchical layout of client data, starting with the root node `/` (added and managed by zookeeper internally). Though this layout bears similarity with a typical file system, there is one exception to such rule. Even parent nodes i.e., folders can hold data and at the same time have child nodes under them. A node in Zookeeper terminology is referred to as a *ZNode*. Here's the empirical view of a sample structure:

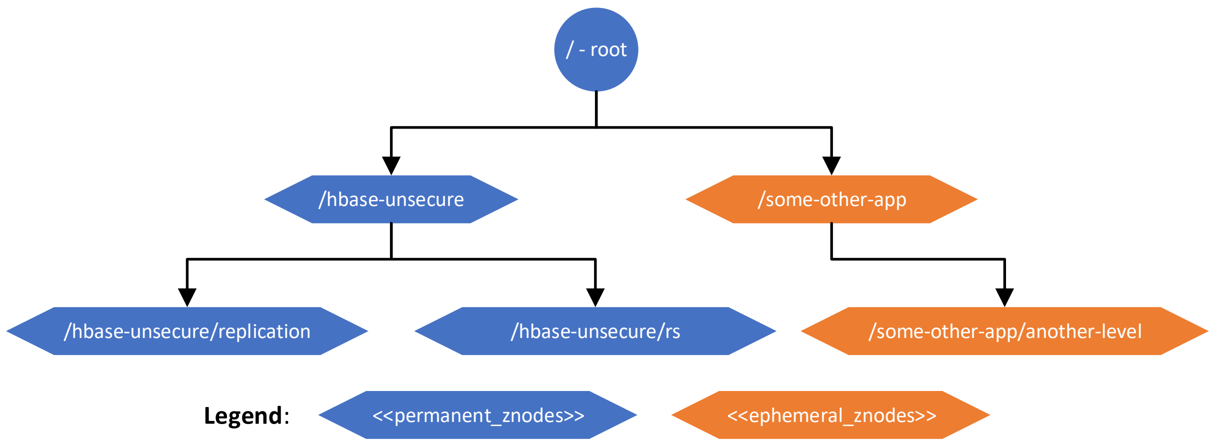


Figure 18: ZNode Sample Hierarchy

As you see from the visual there are two types of znodes – permanent and ephemeral. Permanent nodes are persisted to disk besides being maintained in in-memory data structures. The persistence layer is a collection of transaction logs and snapshots. Ephemeral nodes are removed from in-memory at the end of a client session, when the client disconnects, and the session is invalidated.

6.5 Appendix - Known Limitations

- Hive-HBase integration
    - Open JIRA issues; check specifically on ability to load data by creating region pre-splits (HIVE-13584). Current work around is to try creating the table first in HBase, then define an external table using Hive-HBase integration approach.

6.6 Appendix - External References

- HBase Reference Guide
- Azure HDInsight Pricing
- Azure Data Lake Storage Gen1 Pricing
- Quick Reference to HBase Shell Commands
- HBase and HDFS: Understanding file system usage
- HBase Default configuration XML
- HDFS Default configuration XML
- Apache HBase in HDInsight: Overview
- Apache Hadoop Component versions in HDInsight
- HFile Format Git Resource & Book Reference
- HBase Tags (JIRA HBase-8496)
- HBase Architecture 101 (Old article but useful reference)
- HBase zookeeper znodes explained
- HDInsight HBase: 9 things you must do to get great HBase Performance
- HDInsight – How to perform Bulk Load with Phoenix?
- About HBase flushes and compactions
- Apache HBase and Apache Phoenix, more on block encoding and compression
- Offheap Read-Path in Production – The Alibaba Story
- How are bloom-filters used in HBase
- How to list regions and their sizes for a HBase table
- HBase and HDFS: Understanding Filesystem Usage in HBase
- Azure Data Lake Store: a hyperscale distributed file service for big data analytics

Note – this is a bit old but provides relevant background.

7 Feedback and suggestions

If you have feedback or suggestions for improving this data migration asset, please contact the Data Migration Jumpstart Team (askdmjfordmtools@microsoft.com). Thanks for your support!

Note: For additional information about migrating various source databases to Azure, see the Azure Database Migration Guide.