Columnar Databases for Big Data: HBase

Humberto Silva Galiza de Freitas - D20124066
TU060 - M.Sc. in Computer Science/Advanced Software Development
DATA9810 - Advanced Databases (2021-22) - Prof. Brendan Tierney
Assignment A

Contents

Abstract

Big data belongs to a distinct group of data analytics buzzwords that are often overly explored across many marketing departments, offering different meanings for different people. Skipping the hype behind the term, the Merriam-Webster dictionary [1] brings a much better and precise definition of Big data as *"an accumulation of data that is too large and complex for processing by traditional database management tools"*. Besides the massive volume, the data in this context is often unstructured and unpredictable, creating a challenge for traditional databases. They expect a well-defined and fixed structure where the data in each table row is stored together and whose performance is optimal for single transactions but not for analytic queries performance. In this report, we present an overview of columnar databases. In this approach, data is stored by columns (vertically) rather than by rows (horizontally), speeding up the reading operations for big data. We review some market solutions but ultimately focus on HBase, a distributed column-oriented database based on Google's BigTable but built on top of the Hadoop. HBase has become one of the most popular columnar database engines nowadays for big data because it was designed to leverage the fault tolerance provided by the Hadoop Distributed File System (HDFS) to provide quick random read and write to massive amounts of semi-structured as well as structured data.

Keywords

Big data, Columnar Databases, BigTable, Hadoop, HBase.

I. Introduction

A. Background

Big data belongs to a distinct group of data analytics buzzwords that are often overly explored across many marketing departments, offering different meanings for different people. Skipping the hype behind the term, the Merriam-Webster dictionary [1] brings a much better and precise definition of Big data as *"an accumulation of data that is too large and complex for processing by traditional database management tools"*.

The work of [2] expands in the concept by providing a definition that is closer to the business world as follows *"Big data enables organizations to store, manage, and manipulate vast amounts of disparate data at the right speed and at the right time*. Still, the author argues that *"to gain the right insights, big data is typically broken down by three characteristics"*:

- **Volume**: How much data
- **Velocity**: How fast data is processed
- **Variety**: The various types of data

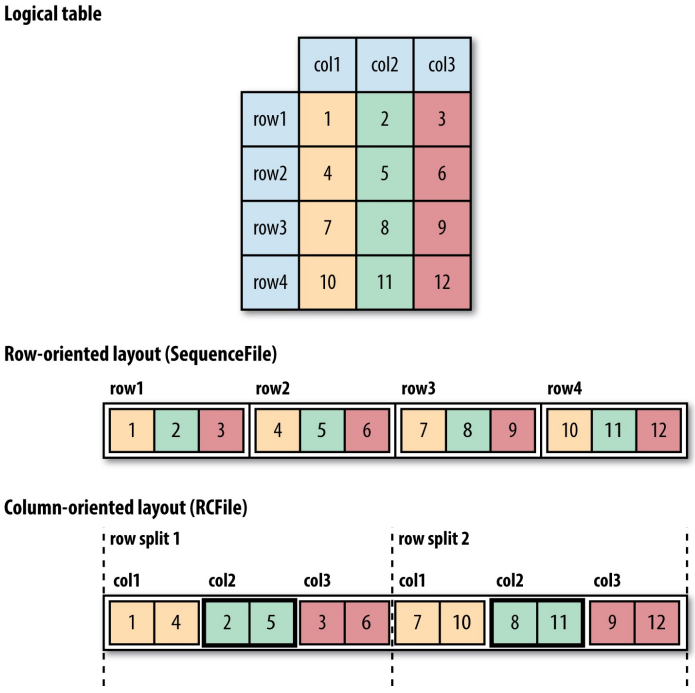
Besides the massive volume, the data in this context is often unstructured and unpredictable, creating a challenge for traditional databases. Relational Database Management Systems (RDBMS) often expect a well-defined and fixed structure where the data in each row of a table is stored together, and they are used optimized for Online Transaction Processing (OLTP) [3]. In short, it means that relational databases perform well for a single transaction like inserting, updating, or deleting relatively small amounts of data but not for analytic queries performance [4].

The need to handle such peculiar amounts of data led to developing an approach to store and retrieve data faster and more efficiently, allowing for instant analytical queries. Column-oriented databases are one approach that can be beneficial in a big data environment. As the name hints, data is stored by columns (vertically) rather than by rows (horizontally).

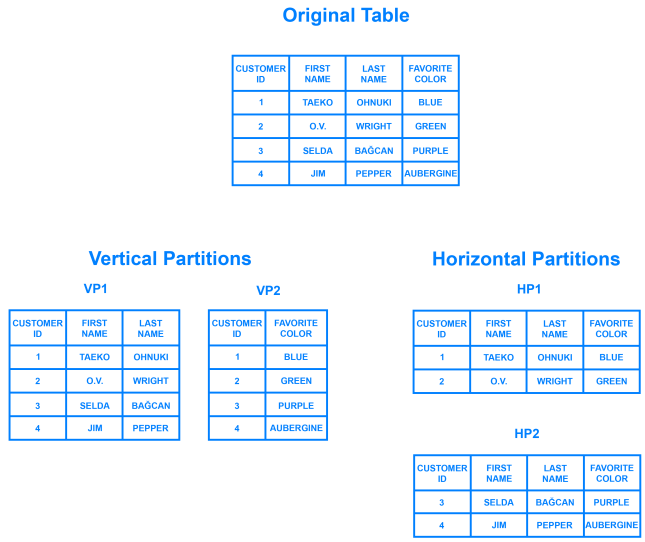
The main advantage of a columnar store is that partial reads are much more efficient because a lower volume of data is loaded due to reading only the relevant data instead of the whole record [5]. In addition, this approach helps to minimize resource usage related to queries on big data sets.

As presented by [6], a column-oriented layout permits columns that are not accessed in a query to be skipped, and this is a key point that differentiates it from a relational database. Columnar databases support document creation, retrieval via query, updating and editing, and deletion of information. Therefore, businesses interested in implementing a data warehousing and big data processing database may opt for a columnar database. Figure 1a helps to visualize how different rows are in a normal database as opposed to the column-oriented design [7].

Fig. 1: Row-oriented vs Column-oriented.



(a) Logical storage of rows and columns on the disk. Credits: [8].



(b) Database sharding: vertical vs horizontal partitioning. Credits: [9].

In general, column-oriented formats work well when queries access only a small number of columns in the table. Conversely, row-oriented formats are appropriate when many columns of a single row are needed for processing simultaneously. Figure 1a helps to illustrate and understand the differences between the two layouts from a data storage perspective [8].

In a column-oriented format, the rows in a file are broken up into row splits, and then each split is stored in a column-oriented fashion. Then, the values for each row in the first column are stored first, followed by the values for each row in the second column, and so on [6]. Thus, in a wide-column database, data is represented as a multidimensional map. The columns are grouped into column families (usually storing data of the same type), and inside each column, family data are stored row-wise. This layout is best for storing data retrieved by a key or a sequence of keys [10].

As [10] points out, *"column-oriented databases should not be mixed up with wide column stores (...)"*. In the simplest terms, a wide-column database is a type of columnar database that supports a column family stored together on disk, not just a single column [11]. Figure 2 illustrates one example of column family representation in HBase for a table called "Person" [12]. In this case, by querying the column family "personal_data" we will retrieve columns "Name" and "Address".

Sharding is a database architecture pattern related to horizontal partitioning, whereby each partition has the same schema and columns and entirely different rows [7]. The data held in each partition is unique and independent of the data held in other partitions. Like other NoSQL databases, column-oriented databases are designed to scale "out" (i.e., horizontally) using distributed clusters of low-cost hardware to increase throughput. In this case, the data stored in individual columns is sharded (or partitioned) across multiple servers [12]. Figure 1b compares how horizontal partitioning relates to vertical partitioning.

The data layout is just one of the steps in a series of possible optimizations that columnar databases bring accordingly to [10]. Reading multiple values for the same column in one run significantly improves cache utilization and computational efficiency. In this sense, the author generalizes that:

Reduced I/O is one of the primary reasons for this new layout, but it offers additional advantages playing

> *into the same category: since the values of one column are often very similar or even vary only slightly between logical rows, they are often much better suited for compression than the heterogeneous values of a row-oriented record structure; most compression algorithms only look at a finite window.* [10]

Columnar databases are typically better for Online Analytical Processing (OLAP) applications. Analytical applications often need aggregate data, where only a subset of a table's attributes are required. As already stated, column-oriented databases are partitioned vertically instead of storing the entire row. Hence the individual values are stored contiguously in storage by column. Therefore, columnar stores are suitable for computing trends and averages for trillions of rows and petabytes for data.

In addition to that, wide-column databases are ideal for use cases that require a large data set that can be distributed across multiple database nodes, especially when the columns are not always the same for every row. A few examples of typical use cases are log data (from applications, devices, servers, etc.), Internet of Things (IoT) sensor data, time-series data (such as temperature monitoring or financial trading data), attribute-based data (such as user preferences or equipment features), and general real-time analytics.

The main drawback of a column-oriented approach is that manipulating (e.g., lookup, update or delete) an entire given row is inefficient. However, the situation should rarely occur in databases for analytics (i.e., "data warehousing") provided that most operations are read-only, rarely read many attributes in the same table, and writes are only appended to the table.

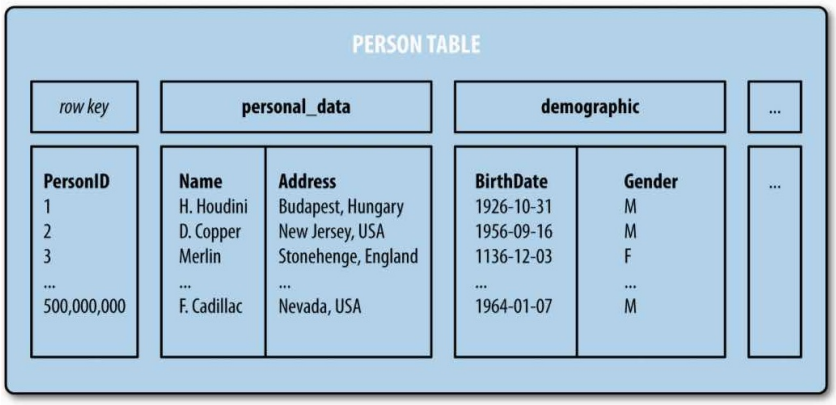


Fig. 2: HBase column family data representation example. Credits: [12].

Also, from a design perspective, columnar databases are more complex than relational table design, as data access patterns drive table design. Hence, tables should be structured to return data efficiently otherwise will sacrifice all the performance gains brought by the solution. In the box below, we briefly compiled some of the advantages and disadvantages of such an approach.

Advantages	Disadvantages
Whole columns can be skipped	Many algorithms will need all columns, and only record at a time (e.g. k-means) or may even need to compute a pairwise distance matrix
Run-length compression works better on columns (for certain data types; in particular with few distinct values)	compression techniques only work well on sparse data types and factors, but not well on double-valued continuous data
	Appends on column stores are expensive, so it is not ideal for streaming / changing data

In summary, as evidenced in the previous paragraphs, the data storage format in use by columnar databases makes them faster and more efficient for instant analytical queries, ideally for data warehouses to handle and process massive

volumes of data from multiple sources. Moreover, it currently serves as a basis for several business intelligence tools. In the following Subsection, we explore the landscape of market solutions of columnar databases, aiming to highlight the critical aspects of each of them, the commonalities, and the differences.

B. Columnar Databases Solutions Landscape

There is a multitude of columnar databases solutions available in the market nowadays. This section introduces a non-exhaustive list of such solutions, sorting them into three main categories: Free and Open-Source Software (FOSS), Platform-as-a-Service (PaaS), and Proprietary solutions.

We limited the analysis to cover only the Top 10 database engines for October 2021 in terms of popularity. To help with the selection criteria, we combined the "DB-Engines Ranking" website [13] ranking with the search statistics retrieved from Google Trends. To this end, we leveraged the "pytrends" Python library to build a script and gather the data from Google Trends. See Appendix B for more details about the implementation.

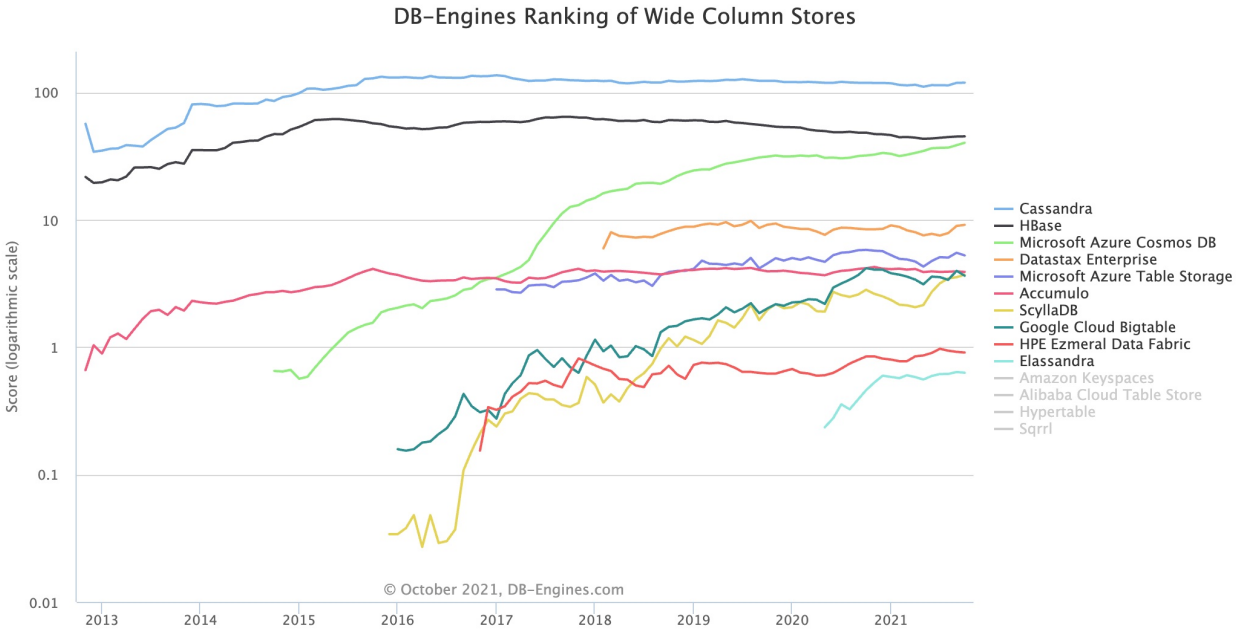


Fig. 3: Top 10 wide-column databases for October/2021. Credits: [13].

1) Free and Open-Source Software (FOSS):

1) **Apache Accumulo** is a highly scalable, distributed, open-source data store modeled after Google's Bigtable design. From their offical website, *Accumulo is built to store up to trillions of data elements and keeps them organized so that users can perform fast lookups*. This database supports flexible data schemas and scales horizontally across thousands of machines. Website: https://accumulo.apache.org/
2) **Apache Cassandra** is a wide-column store implemented in Java and based on a few ideas of BigTable and DynamoDB, supporting a wide range of programming languages. Apache Cassandra is currently the leading NoSQL distributed database management system suited for hybrid and multi-cloud environments. As described in their website, "*It drives many of today's modern business applications by offering continuous availability, high scalability and performance, strong security, and operational simplicity while lowering the overall cost of ownership*.". Website: https://cassandra.apache.org/
3) **Apache Elassandra** is a project that provides a forked version of ElasticSearch that works as a secondary index implementation for Cassandra. Website: https://www.elassandra.io/
4) **Apache HBase** is a wide-column store based on Apache Hadoop and, similarly to Apache Cassandra, based on the concepts of Google's BigTable. It is a non-relational NoSQL database that features BigTable capabilities

on top of Hadoop and HDFS, thus, ensuring similar scalability and fault tolerance. As briefly explained in the Section II this work focus on HBase and we provide in-depth details about Hbase in Section II. Website: https://hbase.apache.org/

5) **ScyllaDB** is specifically designed for real-time applications. This database is compatible with Apache Cassandra while achieving significantly higher throughputs and lower latencies. Website: https://www.scylladb.com/

2) Platform-as-a-Service (PaaS):

1) **Google Cloud BigTable** is the cloud version of the proprietary data storage system built on Google File System (GFS). Being the precursor of most of the actual columnar-datases, including the popular Cassandra and HBase, the solution is designed to scale a massive amount of data distributed across data centers globally [14]. Website: https://cloud.google.com/bigtable
2) **Amazon Redshift** is a column-oriented, fully managed, petabyte-scale data warehouse that makes it simple and cost-effective to analyze big data using existing business intelligence tools. Amazon Redshift achieves efficient storage and optimum query performance through a combination of massively parallel processing, columnar data storage, and very efficient, targeted data compression encoding schemes. Website: https://aws.amazon.com/redshift/
3) **Microsoft Azure Table Storage** is a cloud-based NoSQL datastore that can be used to store large amounts of structured non-relational data. Azure Table offers a schema-less design, enabling to keep a collection of entities in one table. Website: https://azure.microsoft.com/en-us/services/storage/tables/

3) Proprietary:

1) **DataStax Enterprise** is a product built on top of Apache Cassandra and designed for hybrid cloud. The solution integrates graph, search, analytics, administration, developer tooling, and monitoring into a unified platform into a unique venue. Another differential is the user support, provided by the company of same name. Website: https://www.datastax.com/products/datastax-enterprise
2) **HPE Ezmeral Data Fabric** was previously known as the MapR Data Platform for data lakes. It is an exabyte-scale repository for streaming and file data, a real-time database, a global namespace, and integrations with Hadoop, Spark, and Apache Drill for analytics and other applications. Website: https://www.hpe.com/ie/en/software/ezm data-fabric.html

It is also worth mentioning that there are other companies that are engaged directly with the Open Source community (e.g., providing patches and new functionalities or just maintaining the code base) for some of the FOSS products mentioned above, as well as providing professional services and support - similar to what DataStax does for Apache Cassandra. For instance, for Apache HBase, there is a "commercial" version supported by Cloudera [1].

The rest of this work is organized as follows. First, in Section II we shift the focus to HBase, one of the most popular columnar database engines nowadays, which is licensed under the Apache Software Foundation umbrella and that uses the Hadoop file system and MapReduce engine for its core data storage needs. Following in Section III we introduce a tutorial on how to use HBase. We start the guide with basic examples using a "CRUD" (Create, Read, Update, and Delete) exercise to explore the key features present in the solution. Then, we present a simulation to insert bulk big data into HBase, to demonstrate the performance of random write operations in batches using programmatic access. Finally, in Section IV we summarize this report and draw some conclusions after havign thoroughly reviewed the columnar databases as a potential solution for big data in the enterprise space.

II. HBase: a deep dive

A. What is HBase?

HBase is a column-oriented database that prides itself on its ability to provide consistency and scalability [15]. The design of HBase is modeled on Google's BigTable, a high-performance, proprietary database developed by Google and described in the 2006 white paper "Bigtable: A Distributed Storage System for Structured Data" [14].

HBase extends the BigTable model, where each row is indexed by a single row key (similar to a primary key in the RDBMS world), offering the server-side hooks to implement flexible secondary index solutions. In addition, it

[1]https://www.cloudera.com/products/open-source/apache-hadoop/apache-hbase.html

provides push-down predicates, that is, filters, reducing data transferred over the network. The table below flags some nomenclature differences between Google BigTable and HBase implementation [7].

HBase	BigTable
Region	Tablet
RegionServer	Tablet Server
Flush	Minor compaction
Minor compaction	Merging compaction
HDFS	GFS
Hadoop MapReduce	MapReduce
MemStore	memtable
HFile	SSTable
ZooKeeper	Chubby

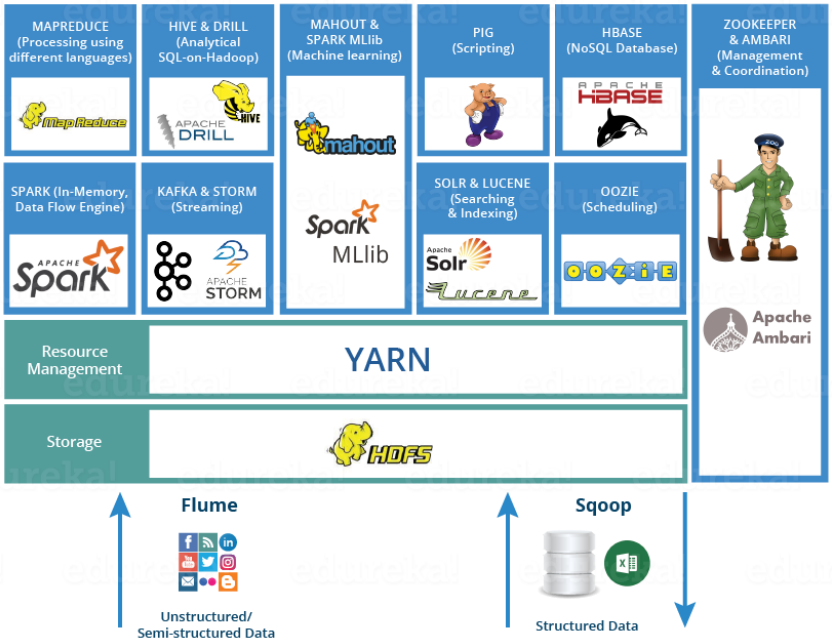


Fig. 4: HBase within the Hadoop Ecosystem. Credits: [16]

HBase is programmed with Java and is part of Apache's Hadoop MapReduce framework [7]. Simply put, HBase replaces Google File System (GFS) with Hadoop Distributed File System (HDFS). Figure 4 depicts the various layers of the Hadoop 2.0 ecosystem - Hbase located on the structured storage layer.

As outlined in [7], *"HBase writes updated contents into the RAM and regularly writes them into files in discs. The row operations are atomic operations, equipped with row-level locking and transaction processing. In addition, large-scope transaction processing is provided with optional support"*. Figure 5 illustrates data consumer reads/accesses of the data randomly in HDFS using HBase.

In addition to that, the work of [17] highlights some relevant differences between HDFS and HBase. The author sustains that:

> *"Partition and distribution are transparently operated, with client hash or a fixed secret key space. Unlike MapReduce's offline batch computing framework, HBase is random access storage and retrieval data platform, which makes up for the shortcomings of HDFS that cannot access data randomly".*

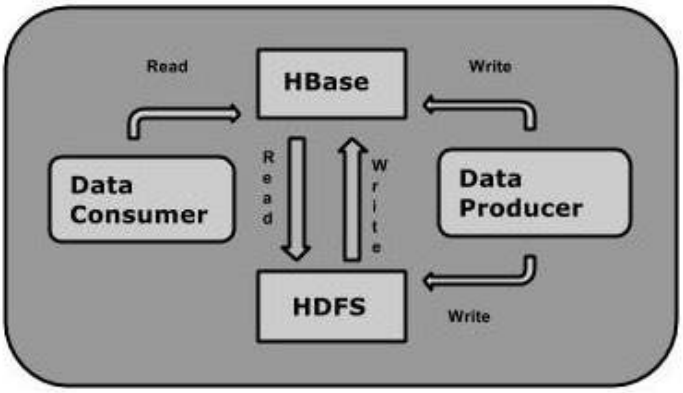


Fig. 5: Data consumer reads/accesses the data in HDFS randomly using HBase. Credits: [18]

Therefore, that is the gap that HBase fills. Unfortunately, HDFS is not built for fast record lookups in files. HBase instead creates indexed StoreFiles on HDFS that enable it to perform fast lookups by row keys. In the box below, we contrast the main differences between HDFS and HBase, which helps to understand the relevance of HBase as a compliment to the Hadoop ecosystem [18] [19].

HDFS	HBase
Java-based file system utilized for storing large data sets.	HBase is a Java-based, Not Only SQL database.
It has a rigid architecture that does not allow changes. It does not facilitate dynamic storage.	Allows for dynamic changes and can be utilized for standalone applications.
Ideally suited for write-once and read-many times use cases.	Ideally suited for random write and read of data that is stored in HDFS.
Distributed file system suitable for storing large files.	HBase is a database built on top of the HDFS.
Does not support fast individual record lookups.	HBase provides fast lookups for larger tables.
It provides high latency batch processing; no concept of batch processing.	It provides low latency access to single rows from billions of records (Random access).
It provides only sequential access to data.	Internally uses Hash tables and provides random access, and it stores the data in indexed HDFS files for faster lookups.

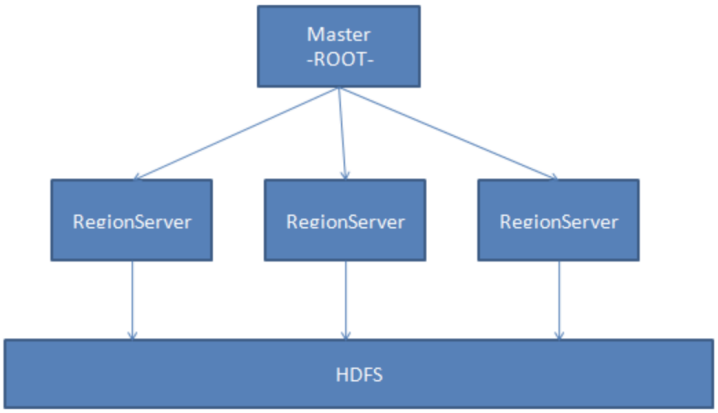
Another two relevant aspects of HBase are that there is no declarative query language as part of the core implementation, and it has limited support for transactions. Instead, row atomicity and read-modify-write operations make up for this in practice. As is highlighted in the work of [7]. The author also explains that:

> *"They cover most use cases and remove the wait or deadlock-related pauses experienced with other systems. HBase handles shifting load and failures gracefully and transparently to the clients. Scalability is built-in, and clusters can be grown or shrunk while the system is in production. Changing the cluster does not involve any complicated rebalancing or re-sharding procedure: it is completely automated."* [7].

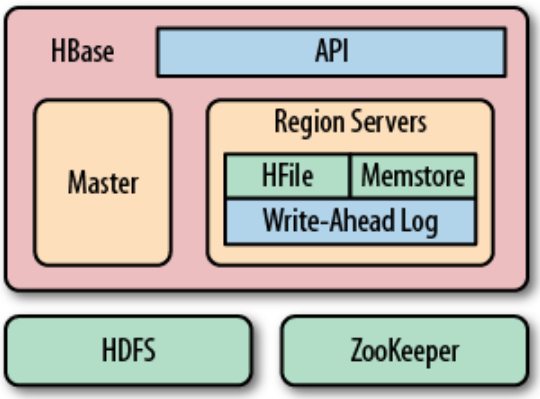
HBase has three major components: the client library, one master server, and many region servers. The master is responsible for assigning regions to region servers and uses Apache ZooKeeper, a reliable, highly available, persistent, and distributed coordination service, to facilitate that task [20]. The region servers can be added or removed while the system is up and running to accommodate changing workloads [18]. Figure 6b shows how the various components of HBase are orchestrated to make use of the existing system, like HDFS and ZooKeeper, but also add their layers to form a complete platform.

A RegionServer corresponds to one node in the cluster, and one RegionServer is responsible for managing multiple

Fig. 6: HBase major components.



(a) HBase components overview. Credits: [7].



(b) HBase using its own components while leveraging existing systems. Credits: [7].

Regions. In HBase, a table may require many Regions to store data, and the data in each Region is not disorganized as highlighted by [19]. The underlying architecture in high-level is shown in Figure 6a. We will explore the Hbase architecture in depth in Section II-E.

HBase works with sparse data in a highly fault-tolerant and resilient way. The way it can work on multiple types of data also makes it useful for varied business scenarios. For example, HBase is a good solution if the big data implementation requires random real-time read/write data access. The work of [10] also adds that *"their implementation is best suited for high-volume, incremental data gathering and processing, real-time information exchange, and frequently changing content serving"*.

In addition to that, [7] also points out some of the crucial characteristics of HBase:

- **Consistency**: Although not an "ACID" implementation similar to the traditional RDBMS, HBase offers strongly consistent reads and writes and is not based on an eventually compatible model. It can be used for high-speed requirements as long as there is no need to use the "extra features" offered by RDBMS, like full transaction support or typed columns.
- **Sharding**: Because the supporting file system distributes the data, HBase offers transparent, automatic splitting and redistribution of its content. The term sharding describes the logical separation of records into horizontal partitions. The idea is to spread data across multiple storage files—or servers—instead of having each stored contiguously. We explore HBase sharding with more details in the following Subsection.
- **High availability**: Through the implementation of region servers, HBase supports LAN and WAN failover and recovery. At the core, there is a master server responsible for monitoring the region servers and all metadata for the cluster.
- **Client API**: HBase offers programmatic access through a Java API (native). However, there are also two other options to access HBase without Java: Hbase Thrift interface (the more lightweight and hence faster of the two options) and the other is the REST interface (formally Stargate, a RESTful web service front end for HBase, which uses HTTP verbs to perform an action).

B. HBase Use Case

As highlighted by [21], in today's world, there is a variety of content that is available for consumption both by the users but also by applications clients (i.e., machine-to-machine traffic). This shift in the data generation and consumption patterns leads to an additional requirement where each client needs the same content in different formats. Users consume content and generate a variety of content in a large volume with a high velocity, such as Tweets, WhatsApp messages, Instagram images, or Linkedin posts.

All this content is a clear example of unstructured data, and HBase is the perfect choice as the backend of such applications. In this sense, many scalable content management solutions use HBase as their backend. The work of [21] brings several real-life and well-tested use cases for HBase. The author outlines that *"in many use cases, trickled data is added to a data store for further usages, such as analytics, processing, and serving. This trickled data could be coming from an advertisement's impressions, such as clickstreams and user interaction data, or it can be time-series data"*. HBase is used for storage in all such cases. For example, Open Time Series Database (OpenTSDB) [2] uses HBase for data storage and metrics generation. As described in their website [22]:

> *All OpenTSDB data points are stored in a single, massive table named TSDB by default. This is to take advantage of HBase's ordering and region distribution. All values are stored in the t column family.*

The work of [21] also lists some of the companies that are using HBase in their respective use cases are as follows:

- Facebook (www.facebook.com) is using HBase to power its message infrastructure. Facebook opted for HBase to scale from their old messages infrastructure, which handled over 350 million users, sending over 15 billion person-to-person messages per month. HBase was selected due to the excellent scalability and performance for big workloads, along with autoloading balancing and failover features, and so on. Facebook also uses HBase for counting and storing the "likes" contributed by users. More details are given at their Engineering Blog: Migrating Messenger storage to optimize performance.
- Meetup (www.meetup.com) uses HBase to power a site-wide, real-time activity feed system for all members and groups. In its architecture, group activity is written directly to HBase and indexed per member, with the member's custom feed served directly from HBase for incoming requests.
- Twitter (www.twitter.com) uses HBase to provide a distributed, read/write backup of all the transactional tables in Twitter's production backend. Later, this backup is used to run MapReduce jobs over the data. Additionally, its operations team uses HBase as a time-series database for cluster-wide monitoring/performance data.
- Yahoo (www.yahoo.com) uses HBase to store document fingerprints for detecting near-duplications. With millions of rows in the HBase table, Yahoo runs a query for finding duplicated documents with real-time traffic.

In addition to that, the Apache HBase project maintains a list of institutions and projects which are using HBase on this website: https://hbase.apache.org/poweredbyhbase.html

In summary, HBase should be used where the use case requires large (and potentially distributed) data sets (millions or billions of rows and columns), and the applications need fast, random, and real time, read and write access over the data.

C. HBase vs RDBMS

HBase is not a column-oriented database in the typical RDBMS sense but utilizes an on-disk column storage format. As pointed out by [7], *"unfortunately, this is also where the majority of similarities end"*. Although HBase stores data on disk in a column-oriented format, it is distinctly different from traditional columnar databases. Whereas columnar databases excel at providing real-time analytical access to data, HBase excels at providing key-based access to a specific cell of data or a sequential range of cells.

HBase	RDBMS
HBase is schema-less, it doesn't have the concept of fixed columns schema; defines only column families.	An RDBMS is governed by its schema, which describes the whole structure of tables.
It is built for wide tables. HBase is horizontally scalable.	It is thin and built for small tables. Hard to scale.
No transactions are there in HBase.	RDBMS is transactional.
It has de-normalized data.	It will have normalized data.
It is good for semi-structured as well as structured data.	It is good for structured data.

[2]http://opentsdb.net/

D. HBase Data Model

HBase data model stores semi-structured data having different data types, varying column size, and field size. The layout of the HBase data model eases data partitioning and distribution across the cluster. HBase data model consists of several logical components as detailed in the list below. The Row Key is used to identify the rows in HBase tables uniquely, which makes searches fast. Row key acts as a Primary key in HBase. Any access to HBase tables uses this Primary Key. Each column's name is known as its column qualifier. Column families in HBase are static, whereas the columns qualifiers are dynamic. Figure 7 details the HBase Table data model.

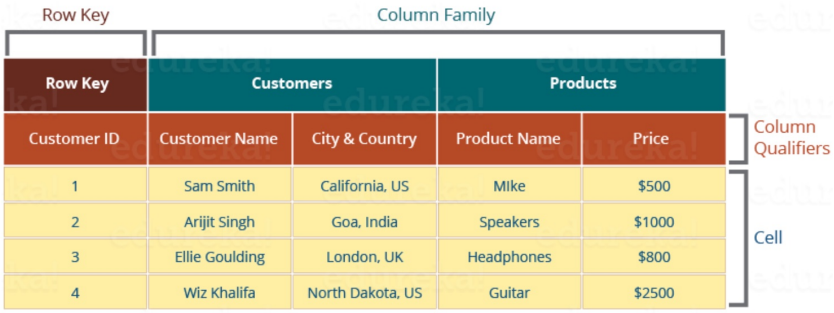


Fig. 7: HBase Table. Credits: [16]

- **HBase Tables**: Logical collection of rows stored in individual partitions known as Regions.
- **Row Key**: Every entry in an HBase table is identified and indexed by a Row Key.
- **Column Family**: Data in rows is grouped as column families, and all columns are stored together in a low-level storage file known as
- **Column Qualifiers**: Each column's name is known as its column qualifier.
- **Cell**: Data is stored in cells. The data is dumped into cells which are specifically identified by row key and column qualifiers.
- **HBase Row**: Instance of data in a table.
- **Timestamp**: Timestamp is a combination of date and time. Whenever data is stored, it is stored with its timestamp. This makes it easy to search for a particular version of data.

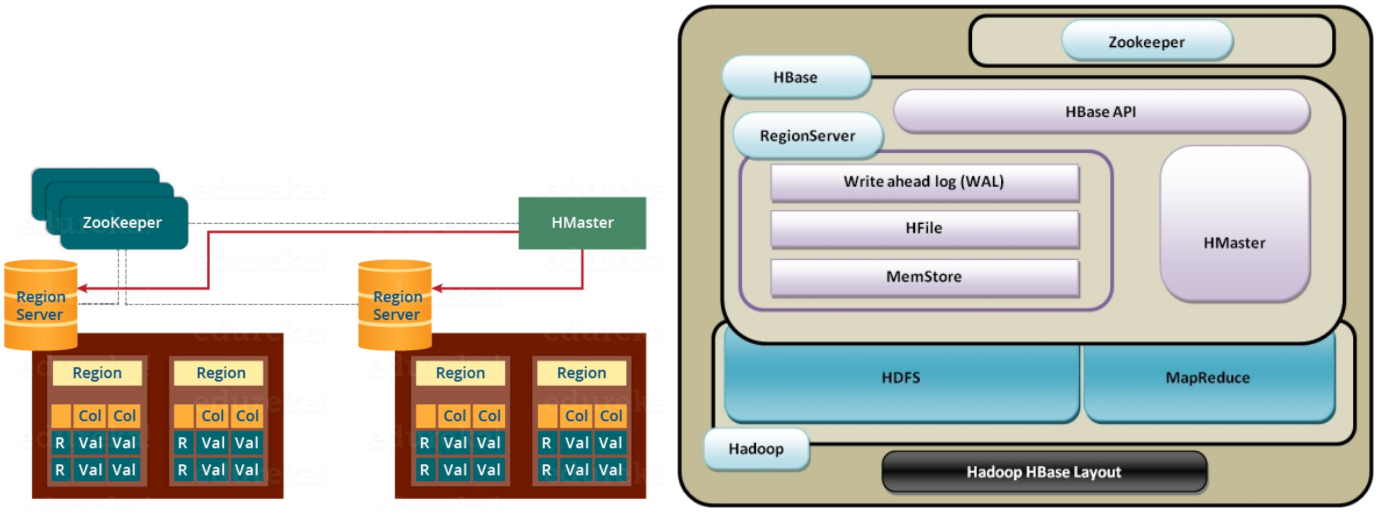
E. HBase Architectural Components

As briefly introduced in Section II, at the architectural level, HBase has three major masterpieces (HMaster, HBase Region Server, and the Client API) and the ZooKeeper. A distributed Apache HBase installation depends on a running ZooKeeper cluster [20]. The later component acts as a coordinator inside the HBase distributed environment and helps maintain the server state inside the cluster by communicating through sessions. Figure 8b shows the components of the HBase architecture.

Before getting into the details of each of these components, first, it is crucial to comprehend the concept of Region as all these Servers (HMaster, Region Server, Zookeeper) are placed to coordinate and manage Regions and perform various operations inside the Regions. As outlined in the work of [7], a Region (HRegion) is the basic unit of horizontal scalability in the HBase ecosystem. HRegions are essentially contiguous ranges of rows stored together. A HRegion contains all the rows between the start key and the end key assigned to that region [24]. They are dynamically split by the system when they become too large. Alternatively, they may also be merged to reduce their number and required storage files.

In HBase, this capability where the HBase tables are dynamically divided into smaller or merged into bigger parts and distributed across the region servers is called Auto sharding. This is because the HBase regions are equivalent to range partitions as used in database sharding. Therefore, they can be spread across many physical servers, thus distributing the load and providing scalability.

Fig. 8: HBase overview



(a) Architecture. Credits: [16]

(b) Components. Credits: [23]

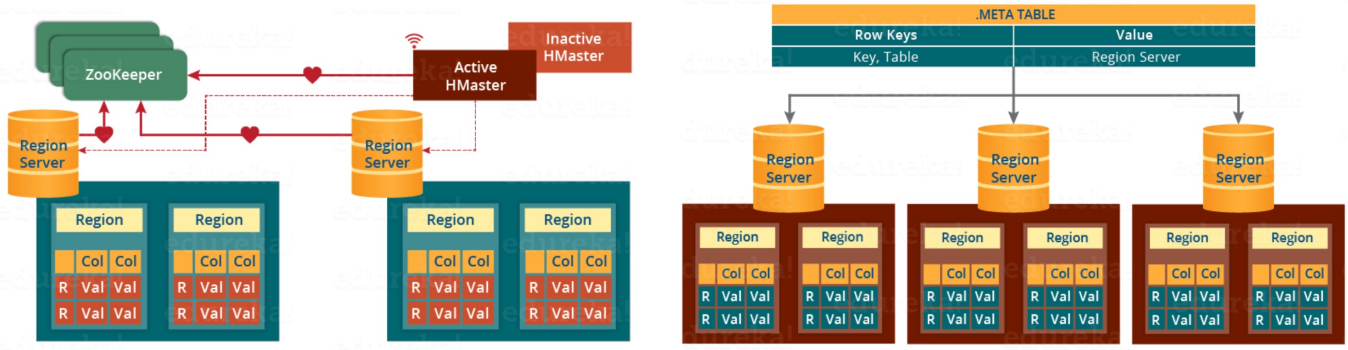
Many regions are assigned to a Region Server, which is responsible for handling, managing, executing reads and writes operations on that set of regions. Region servers can be added or removed as per requirement (i.e., scale-out). In summary, HRegions are a subset of the table's data, and they are essentially a contiguous, sorted range of rows that are stored together. The region servers have regions that communicate with the client and handle data-related operations, read and write requests for all the regions under it, and decide the region's size by following the region size thresholds. For example, figure 8b illustrates the internals of each HRegion Server with HRegion and respective Stores.

Starting from the top of the hierarchy, the HBase HMaster is a lightweight process that assigns regions to region servers in the Hadoop cluster for load balancing [7]. The HMaster is responsible for managing all HRegionServers, which reside on DataNode. HBase HMaster performs DDL operations (create and delete tables). However, it does not store any data; it only stores the mappings (metadata) of data to HRegionServer. All nodes in the cluster are coordinated by Zookeeper and handle various issues encountered during HBase operation [16]. It also handles load balancing of the regions across region servers. It unloads the busy servers and shifts the regions to less occupied servers [16]. Another responsibility is to maintain the state of the cluster by negotiating the load balancing and is also responsible for schema changes and other metadata operations such as the creation of tables and column families. The basic architecture of HBase is shown in Figure 8a.

As mentioned in the previous section, HBase lives in the Hadoop ecosystem, benefiting from its proximity to other related tools. HBase is designed to be fault-tolerant, and to that end, it leverages all the existing Hadoop capabilities. Nevertheles, HBase alone is not sufficient to manage everything. In this sense, the Zookeeper acts like a coordinator inside HBase distributed environment. It helps in maintaining the server state inside the cluster by communicating through sessions. Every Region Server along with HMaster Server sends continuous heartbeat at regular intervals to Zookeeper, and it checks which server is alive and available as illustrated in Figure 9a. It also provides server failure notifications so that recovery measures can be executed. The active HMaster sends heartbeats to the Zookeeper while the inactive HMaster listens for the notification sent by the active HMaster. If the active HMaster fails to send a heartbeat, the session is deleted, and the inactive HMaster becomes active. Zookeeper also maintains the .META Server's path, which helps any client in searching for any region. The Client first has to check with .META Server in which Region Server a region belongs gets the path of that Region Server.

The META table is a special HBase catalog table. It maintains a list of all the Regions Servers in the HBase storage system. The ".META" file maintains the table in the form of keys and values. Key represents the start key of the region and its id, whereas the value contains the path of the Region Server. Figure 9b illustrates the HBase

Fig. 9: HBase internals.



(a) ZooKeeper as Coordination Service. Credits: [16].

(b) HBase META Table. Credits: [16].

META Table organization.

The API offers operations to create and delete tables and column families. In addition, it has functions to change the table and column family metadata, such as compression or block sizes. Furthermore, there are the usual operations for clients to create or delete values and retrieve them with a given row key. The API also allows the HBase user to efficiently iterate over ranges of rows and limit which columns are returned or the number of versions of each cell. The API allows matching columns using filters and select versions using time ranges, specifying start and end times for example [16].

On top of this basic functionality, there are more advanced features. For example, the system has support for single-row transactions, and with this support, it implements atomic read-modify-write sequences on data stored under a single row key. In addition, although there are no cross-row or cross-table transactions, the client can batch operations for performance reasons.

There is also the option to run client-supplied code in the address space of the server. The server-side framework to support this is called co-processors [7]. The code has access to the server's local data and can implement lightweight batch jobs or use expressions to analyze or summarize data based on various operators.

Figure 10 illustrates the relationship between the Thrift/REST gateway running in an HBase cluster and a Web Application. The Thrift and REST client hosts usually do not run any other services, such as DataNodes or HRegionServers, to keep the overhead low and responsiveness high for REST or Thrift interactions [25].

F. HBase Storage

The work of [7] summarizes that a Region Server maintains various regions running on top of HDFS. As depicted in Figure 11, from top-down, each Region Server maintains at least:

- **Block Cache**: it resides at the top of Region Server, and it stores the frequently read data in the memory. If the data in BlockCache is least recently used, then that data is removed from BlockCache.
- **MemStore**: it is the write cache. It stores all the incoming data before committing it to the disk or permanent memory. There is one MemStore for each column family in a region. Hence, there are multiple MemStores for a region because each region contains multiple column families. The data is sorted in lexicographical order before committing it to the disk.
- **Write-Ahead Log (WAL)**: is a file attached to every Region Server inside the distributed environment, whose responsibility is to store the new data that has not been persisted or committed to the permanent storage. It is used in case of failure to recover the data sets.
- **HFile**: it stores the actual cells on the disk (on top of HDFS). HFiles are persistent and ordered immutable maps from keys to values. MemStore commits the data to HFile when the size of MemStore exceeds. Internally, the files are sequences of blocks with a block index stored at the end. The index is loaded when the HFile is

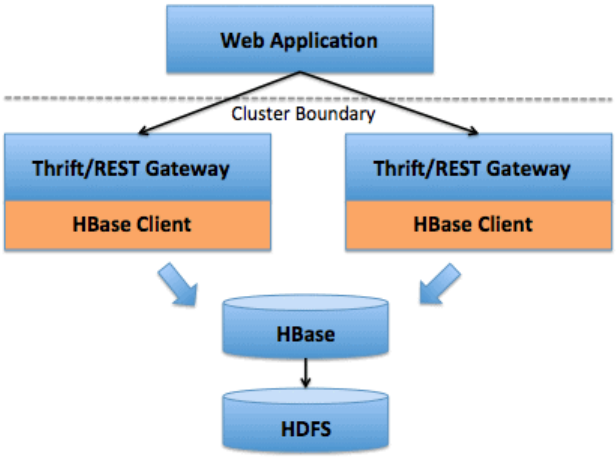


Fig. 10: HBase Thrift and REST gateway overview. Credits: [25].

opened and kept in memory. The default block size is 64 KB but can be configured differently if required. In addition, the store files provide an API to access specific values and scan ranges of values given a start and end key.

1) HBase Read Mechanism: Whenever a client approaches with a read request to HBase if the client does not have it in its cache memory, the following operation occurs:

1) The client retrieves the location of the META table from the ZooKeeper.
2) The client then requests the location of the Region Server of the corresponding row key from the META table to access it. The client caches this information with the location of the META table.
3) Then it will get the row location by requesting from the corresponding Region Server.

For future references, the client uses its cache to retrieve the location of the META table and previously read row key's Region Server. Then the client will not refer to the META table until and unless there is a miss because the region is shifted or moved. Then it will again request to the META server and update the cache. As every time, clients do not waste time retrieving the location of Region Server from META Server. Thus, this saves time and makes the search process faster.

2) HBase Write Mechanism: Hardware failures may be uncommon in individual machines, but node failure is the norm (as are network issues) [15]. HBase can gracefully recover from individual server failures because it uses Write-Ahead Logging (WAL) and distributed configuration. In short, WAL HBase writes data to an in-memory log (HLog) before it is written so that nodes can use the log for recovery rather than disk. The distributed configuration allows nodes to rely on each other for configuration rather than on a centralized source. Figure 4 represents the layout information of HBase on top of Hadoop [23].

Figure 11 illustrates the HBase write mechanism. Whenever the client has a write request, the client writes the data to the commit log (1) (WAL) in HBase. The edits are then appended at the end of the WAL file. As mentioned at the beginning of this Subsection, the WAL file is maintained in every Region Server, and the Region Server uses it to recover data that is not committed to the disk. (2) Once data is written to the WAL, then it is copied to the MemStore. At this point (3), the client receives an acknowledgment confirming that the data has been written to the MemStore. Then, once the data in memory has exceeded the threshold, it is flushed and dumped as an HFile to disk. After the flush, the commit logs can be discarded up to the last unflushed modification. Thus, while the system is flushing the MemStore to disk, it can continue to serve readers and writers without blocking them. This is achieved by rolling the MemStore in memory where the new/empty one is taking the updates while the old/full one is converted into a file. Note that the MemStore data is already sorted by keys matching exactly what HFiles represent on disk, so no sorting or other special processing needs to be performed [16].

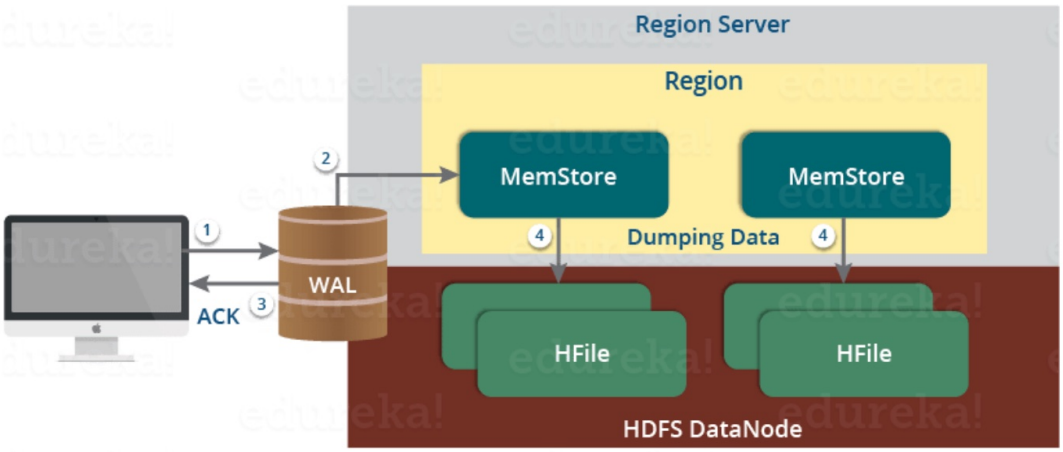


Fig. 11: HBase architecture components. Credits: [16]

3) HBase Data Recovery: Whenever a Region Server fails, ZooKeeper notifies the HMaster about the failure. Then, HMaster distributes and allocates the regions of crashed Region Server to across the other active Region Servers. To recover the data of the MemStore of the failed Region Server, the HMaster distributes the WAL to all the Region Servers. Then, each Region Server re-executes the WAL to build the MemStore for that failed region's column family. The data is written in chronological order (in a timely order) in WAL. Therefore, Re-executing that WAL means making all the changes made and stored in the MemStore file. Therefore, after all the Region Servers executes the WAL, the MemStore data for all column family is recovered.

In this Subsection, we performed a deep dive analysis on HBase internals aiming to understand how it provides distributed storage for massive datasets, while writing and retrieving data at a very fast speed. As outlined in the previous paragraphs, the HBase architectural model can host huge tables on top of clusters with consistent reads and writes. Furthermore, the solution is simple to use via the provided APIs, is linearly scalable, and provides automatic failure support. In summary, whenever the use case requires random, real-time read/write access to Big Data, Apache HBase is a solution.

III. HBase: a hands on approach

The main goal of this section is to present a hands on, scoped tutorial about how to get started with HBase. First and foremost, HBase supports three running modes:

- Standalone mode is a single machine acting alone.
- Pseudo-distributed mode is a single node pretending to be a cluster.
- Fully distributed mode is a cluster of nodes working together.

To facilitate the understanding and help the reader with the ramp-up process, we will use a docker image that installs a local instance of HBase running in standalone mode. Please refer to Appendix A for the instructions on how to download and run the docker image with the HBase in standalone mode.

In the following subsections, we provide a comprehensive guide on using Apache HBase technology in three different scopes. First, we start with a simple "Create, Read, Update, Delete" (CRUD) exercise and develop some ideas around Table Administration to provide the reader with an overview of getting started with the product. Then, we move to a more detailed example of working with a larger dataset, by using a combination of Python and Apache HBase to manipulate and insert bulk data. We offer a limited simulation of using this 'batch' insertion method and some interesting results.

A. CRUD and Table Administration

After the docker image is running, the we should be able to use the HBase both via 'HBase Shell' or via the API to run the commands to Create, Read, Update and Delete (CRUD) data, as well as to perform the basic administration

tasks with the tables. Before manipulating data into HBase, let us perform two simple tests to ensure we have correct access to the HBase system.

1) Test HBase is working via CLI/shell: The HBase shell is a JRuby-based command-line program we can use to interact with HBase. In the HBase shell, we can add and remove tables, alter table schemas, add or delete data, and perform many other tasks. To gain access to the HBase shell, use the command below:

```
$ docker exec -it hbase-docker hbase shell
2021-11-07 19:30:24,851 WARN  [main] util.NativeCodeLoader: Unable to load native-hadoop library
    for your platform... using builtin-java classes where applicable
HBase Shell
Use "help" to get list of supported commands.
Use "exit" to quit this interactive shell.
For Reference, please visit: http://hbase.apache.org/2.0/book.html#shell
Version 2.2.4, r67779d1a325a4f78a468af3339e73bf075888bac, 2020 12:57:39 CST
Took 0.0030 seconds
hbase(main):001:0>
```

To confirm that it is working properly, let us run the 'version' command, to ask HBase for the version information. That should output a version number and hash, and a timestamp for when the version was released, similar to the output below:

```
hbase(main):001:0> version
2.2.4, r67779d1a325a4f78a468af3339e73bf075888bac, 2020 12:57:39 CST
Took 0.0005 seconds
hbase(main):002:0>
```

2) Test HBase is working via Python over Thrift API: Let us repeat the same test, but now using a programmatic approach. Let us connect to the Thrift API port on "localhost" (127.0.0.1) exposed by the container (such as created by the start-hbase.sh script when we started the docker container - see the Appendix A for more details). For this example, the port 55020 is the Thrift API port exposed to the host. Hence, we will use Python's happybase[3] library to talk with this Thrift API port, and Thrift will talk to HBase at the backend.

```

$ pip install --user happybase

$ python3
Python 3.9.6 (default, Jun 29 2021, 06:20:32)
[Clang 12.0.0 (clang-1200.0.32.29)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import happybase
>>> connection = happybase.Connection('127.0.0.1', 55020)
>>> connection.tables()
[]
>>> quit()
```

With a successful connection to Thrift, we can see that, as expected, the connection.tables() method returned an empty list as we have no tables yet created in the database. Hence, up to this point, we are all set to proceed with the following examples.

3) Creating a Table: First, let us use the HBase shell to create a table with a single-family column. Use the 'create' command to create a table called 'wiki' and a single column family called 'text':

```
hbase(main):004:0> create 'wiki', 'text'
Created table wiki
Took 0.7944 seconds
=> Hbase::Table - wiki
```

The first argument is the command, followed by the table name and column-family in quotes separated by comma. The quotes are mandatory, as the shell expects a string. The recently created table is currently empty: it has no

[3]http://happybase.readthedocs.org/en/latest/

rows and thus no columns. Unlike a relational database, in HBase, a column is specific to the row that contains it. Therefore, columns do not have to be predefined in something like a *CREATE TABLE* declaration in SQL. For the educational purpose here, though, we will stick to a schema, even though it is not predefined. When we start adding rows, we will add columns to store data at the same time.

Repeating the same exercise but at this time using Python (with the help of happybase library) access HBase via Thrift API, would look like the listing below. In the output below, we created two more tables called 'table-name' and 'table2-tudublin'. We named the column-family 'family' for both tables.

```
$ pip install --user happybase

$ python3
Python 3.9.6 (default, Jun 29 2021, 06:20:32)
[Clang 12.0.0 (clang-1200.0.32.29)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import happybase
>>> connection = happybase.Connection('127.0.0.1', 55020)
>>> connection.create_table('table-name', { 'family': dict() } )
>>> connection.tables()
[b'table-name']
>>> connection.create_table('table2-tudublin', { 'family': dict() } )
>>> connection.tables()
[b'table-name', b'table2-tudublin']
>>> quit()
```

Furthermore, it is also possible to see the result of 'create$_table()'$ $method by checking the master status page (in this exampl$ $//127.0.0.1 : 55018/master-status according to the output generated when we started the HBase docker container). Figure$

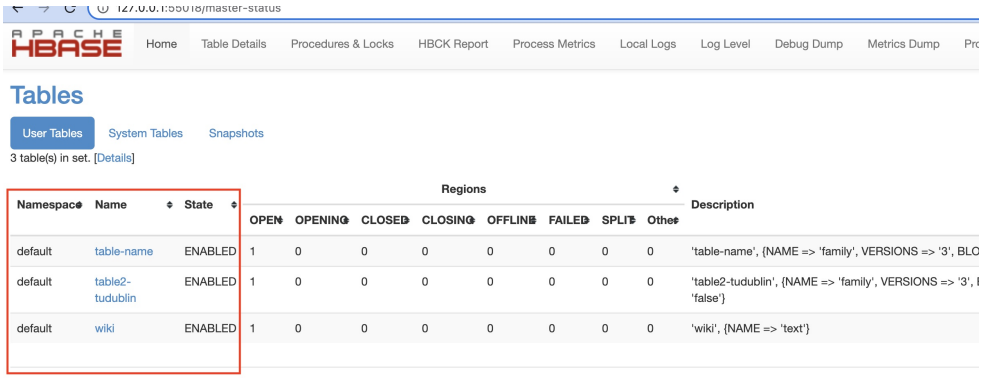


Fig. 12: HBase Master status page example. Credits: author.

4) Inserting, Updating, and Retrieving Data: Now, let us start populating our tables. To this end, we will use the 'put' command to insert a new row into the 'wiki' table with the key named 'Home', adding 'Welcome to the wiki!' to the column called 'text:'. Note the colon at the end of the column name: this is a requirement in HBase if we do not specify a column family in addition to a column (in this case, we have specified no column family).

```
hbase(main):007:0> put 'wiki', 'Home', 'text:','Welcome to the wiki!'
Took 0.0419 seconds
hbase(main):008:0>
```

Exploring again the same concept but in a programmatic way, let us insert some key/value data into table 'table2-tudublin' using the 'put' method, which is similar to the shell command 'put'.

```
$ python3
Python 3.9.6 (default, Jun 29 2021, 06:20:32)
[Clang 12.0.0 (clang-1200.0.32.29)] on darwin
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>> import happybase
>>> connection = happybase.Connection('127.0.0.1', 55020)
>>> connection.tables()
[b'table-name', b'table2-tudublin']
>>> table = connection.table('table2-tudublin')
>>> table.put('row-key', {'family:qual1': 'value1', 'family:qual2': 'value2'})
>>> for k, data in table.scan():
...  print(k, data)
...
b'row-key' {b'family:qual1': b'value1', b'family:qual2': b'value2'}
>>> quit()
```

We can query the data for the 'Home' row using 'get' command, which requires two parameters: the table name and the row key. We can optionally specify a list of columns to return. Here, we will fetch the value of the 'text' column:

```
hbase(main):008:0> get 'wiki', 'Home', 'text:'
COLUMN                                        CELL
 text:                                        timestamp=1636315569925, value=Welcome to the wiki!
1 row(s)
Took 0.0274 seconds
hbase(main):009:0>
```

Notice the timestamp field in the output. HBase stores an integer timestamp for all data values, representing time in milliseconds since the epoch (00:00:00 UTC on January 1, 1970). When a new value is written to the same cell, the old value hangs around, indexed by its timestamp. This versioning feature is unique to HBase amongst similar databases since most require the developer to handle historical data specifically.

Finally, let's perform a scan operation. Scan operations simply return all rows in the entire table. Scans are powerful and significant for development purposes, but they are also very blunt instruments, so we might want to use them with care with production environments.

```
hbase(main):009:0> scan 'wiki'
ROW                                           COLUMN+CELL
 Home                                         column=text:, timestamp=1636315569925, value=Welcome
   to the wiki!
1 row(s)
Took 0.0202 seconds
hbase(main):010:0>
```

It is worth mentioning that the 'put' and 'get' commands allows to specify a timestamp explicitly. If a timestamp is not specified, HBase will use the current time when inserting, and it will return the most recent version when reading.

5) Altering Tables: We created the wiki table with no special options, so all the HBase default values were used. One such default value is to keep only three VERSIONS of column values, so let us increase that. To make schema changes, first, we have to take the table offline with the disable command.

```
hbase(main):011:0> disable 'wiki'
Took 0.7968 seconds
hbase(main):012:0>
```

Below, we are instructing HBase to alter the text column family's VERSIONS attribute.

```
hbase(main):014:0> alter 'wiki',{ NAME => 'text', VERSIONS =>
hbase(main):015:1* org.apache.hadoop.hbase.HConstants::ALL_VERSIONS }
Updating all regions with the new schema...
All regions updated.
Done.
Took 1.2369 seconds
hbase(main):016:0>
```

With the wiki table still disabled, let's add the revision column family. Again, using the alter command:

```
hbase(main):016:0> alter 'wiki',{ NAME => 'revision', VERSIONS =>
hbase(main):017:1* org.apache.hadoop.hbase.HConstants::ALL_VERSIONS }
Updating all regions with the new schema...
All regions updated.
Done.
Took 1.1737 seconds
hbase(main):018:0>
```

With these additions in place, let's re-enable the 'wiki' table:

```
hbase(main):018:0> enable 'wiki'
Took 0.7977 seconds
hbase(main):019:0>
```

The 'wiki' table has been modified to add the column-family 'revision', and we can now start adding data to it.

It is noticeable that the HBase shell is great for tasks such as manipulating tables. However, the shell data insertion support is not the best. For example, the put command allows to set only one column value at a time, and in our newly updated schema, we need to add multiple column values simultaneously, so they all share the same timestamp. For this reason, from this point and beyond, we will focus on using programmatic access using the already mentioned Happybase Python library.

6) Manipulating data programmatically: In this example, besides the programmatic data manipulation, we will experiment the utilization of the Batch#put() function available in HBase. Essentially, when the number of records reaches the batch_size, Batch#send() function will be called. To this end, we will create a wrapper script in Python to help load many items at a time (also called bulk load) and dispatching these items in batches such that they can be written to the database.

In addition to that, in this example, we will experiment another feature of HBase called *namespaces*. A namespace is a logical grouping of tables analogous to a database or a schema in a relational database system. We are not required to create a namespace as HBase automatically assigns a default namespace when creating a table and does not associate it with a namespace. However, the usage of namespaces allows for multitenancy, another outstanding capability of Apache HBase. In short, with namespaces, a group of users can share access to a set of tables, but the users can be assigned different privileges. This capability can be useful when, for example, the company needs to provide access to the same HBase tables that contain sample data for testing.

To create the table, we will use the *hbase shell*. First, we will create a *namespace* called "sample_data". The table for this exercise is called "tripdata", as we will be inserting bike trip data generated a made publicly available by the City of New York. The data for this example was downloaded from https://ride.citibikenyc.com/system-data on 06 November 2021. The file contains 3069240 million rows.

```
$ wc -l 202110-citibike-tripdata.csv
3069240 202110-citibike-tripdata.csv
```

The "tripdata" table will have three column families: "ride_data" for general user trip data, "time_data" aimed to track the trip time/data information, and "geo_data", to track geospatial information for each ride. For all we are accepting all table defaults provided by HBase.

```
$ docker exec -it hbase-docker hbase shell

2021-11-09 13:05:53,672 WARN  [main] util.NativeCodeLoader: Unable to load native-hadoop library
    for your platform... using builtin-java classes where applicable
HBase Shell
Use "help" to get list of supported commands.
Use "exit" to quit this interactive shell.
For Reference, please visit: http://hbase.apache.org/2.0/book.html\#shell
Version 2.2.4, r67779d1a325a4f78a468af3339e73bf075888bac, 2020 12:57:39 CST
Took 0.0065 seconds
hbase(main):001:0>

```

```
hbase(main):025:0> create_namespace "sample_data"
Took 0.2245 seconds
hbase(main):026:0>

hbase(main):026:0> create "sample_data:tripdata", "ride_data", "time_data", "geo_data"
Created table sample_data:tripdata
Took 1.2466 seconds
=> Hbase::Table - sample_data:tripdata
hbase(main):027:0>

hbase(main):027:0> list
TABLE
table-name
table2-tudublin
wiki
sample_data:tripdata
4 row(s)
Took 0.0070 seconds
=> ["table-name", "table2-tudublin", "wiki", "sample_data:tripdata"]
hbase(main):028:0>
```

As we can observe from the output above, the table has been created successfully, and we can now start bulk loading the data using the Python script.

```
"""
Description: This scripts insert data into HBase.
Author: Humberto Galiza
Date: 03 November 2021
Original data source:
    - https://s3.amazonaws.com/tripdata/202110-citibike-tripdata.csv.zip

To create the table, first, use the hbase shell. We are going to create a
namespace called "sample_data". The table for this script is called "tripdata",
as we will be inserting bike trip data from the City of New York.

Our table will have three column families:
    - "ride_data", to store generic ride info
    - "time_data", to store time related info
    - "geo_data", to store geospatial info

For all, we are accepting all table defaults.

% hbase shell (or 'docker exec -it hbase-docker hbase shell' if running hbase from docker)
hbase> create_namespace "sample_data"
hbase> create "sample_data:tripdata", "ride_data", "time_data", "geo_data"
"""

import csv
import happybase
import time
import json
import logging
import statistics

# batch_size = 5000
host = "127.0.0.1"
port = 55038
file_path = "xaa"
namespace = "sample_data"
table_name = "tripdata"
BATCHES = [100, 1000, 2000, 5000, 10000, 50000, 100000, 200000]
REPETITION = 20

```

```
40
41 def connect_to_hbase(batch_size):
42     """
43     Connect to HBase server.
44     This will use the host, namespace, table name, and batch size as defined in
45     the global variables above.
46     """
47     conn = happybase.Connection(
48         host=host, port=port, table_prefix=namespace, table_prefix_separator=":"
49     )
50     conn.open()
51     try:
52         # delete previous run, to avoid influence of disk space/number of records in the
    measurements
53         conn.disable_table(table_name)
54         conn.delete_table(table_name)
55     except Exception as error:
56         logging.warning(f"Couldn't disable table {table_name}, error: {error}")
57         pass
58     # create the new table under the namespace
59     conn.create_table(
60         table_name, {"ride_data": dict(), "time_data": dict(), "geo_data": dict()}
61     )
62     table = conn.table(table_name)
63     batch = table.batch(batch_size=batch_size)
64     return conn, batch
65
66
67 def insert_row(batch, row):
68     """
69     Insert a row into HBase.
70     Write the row to the batch. When the batch size is reached, rows will be
71     sent to the database.
72     Rows have the following schema:
73         [ ride_id, rideable_type, started_at, ended_at, start_station_name, start_station_id,
74         end_station_name, end_station_id, start_lat, start_lng, end_lat, end_lng, member_casual ]
75     """
76     batch.put(
77         row[0],
78         {
79             "ride_data:ride_type": row[1],
80             "ride_data:started_at": row[2],
81             "ride_data:ended_at": row[3],
82             "ride_data:start_station_name": row[4],
83             "ride_data:start_station_id": row[5],
84             "ride_data:end_station_name": row[6],
85             "ride_data:end_station_id": row[7],
86             "ride_data:member_casual": row[12],
87             "geo_data:start_lat": row[8],
88             "geo_data:start_lng": row[9],
89             "geo_data:end_lat": row[10],
90             "geo_data:end_lng": row[11],
91         },
92     )
93
94
95 def read_csv():
96     csvfile = open(file_path, "r")
97     csvreader = csv.reader(csvfile)
98     return csvreader, csvfile
99
100
```

```
result = {}

for batch_size in BATCHES:
    result[batch_size] = {}
    result[batch_size]["mean"] = 0
    result[batch_size]["median"] = 0
    result[batch_size]["stdev"] = 0
    total_tmp = []
    for i in range(1, REPETITION + 1):
        row_count = 0
        start_time = time.time()
        # After everything has been defined, run the script.
        conn, batch = connect_to_hbase(batch_size)
        print(f"Starting execution {i} for batch_size: {batch_size}")
        csvreader, csvfile = read_csv()
        # Loop through the rows. The first row contains column headers, so skip that
        # row. Insert all remaining rows into the database.
        for row in csvreader:
            row_count += 1
            if row_count == 1:
                pass
            else:
                insert_row(batch, row)
        # If there are any leftover rows in the batch, send them now.
        batch.send()
        csvfile.close()
        conn.close()
        duration = time.time() - start_time
        total_tmp.append(duration)
        print(
            f"Finished execution {i} for batch_size: {batch_size}. Inserted row count: {row_count
    }, duration: {duration}"
        )
        print(f"Partial stats")
        print(f"Mean (partial): {statistics.mean(total_tmp)}")
        if i > 1:
            print(f"Stdev (partial): {statistics.stdev(total_tmp)}")
        print(f"Median (partial): {statistics.median(total_tmp)}")
    result[batch_size]["mean"] = statistics.mean(total_tmp)
    result[batch_size]["median"] = statistics.median(total_tmp)
    result[batch_size]["stdev"] = statistics.stdev(total_tmp)
    print("===" * 40)
    print(
        f"Simulation ended for batch_size {batch_size}.\nResults:\n{result[batch_size]}"
    )
    print("===" * 40)
print("*" * 79)
print(f"Simulation ended.\nResults: {result}")
out_file = open("simulation_result.json", "w")
json.dump(result, out_file, indent = 6)
out_file.close()
```

Now let us run the script for the whole dataset.

```

$ python3 hbase_load.py
Connect to HBase. table name: tripdata, batch size: 1000
Connected to file. name: 202110-citibike-tripdata.csv
Load complete. Row count: 3069240, duration: 657.535887002945
```

As we can observe, the above execution took 657.53 seconds to load the entire dataset to HBase running within a docker container when using batches of 1000. Repeating the experiment for the same full dataset one more time, but now with a batch size of 100000 has had an average time of 519.47 seconds to complete the writes.

```
$ python3 hbase_load.py | grep complete | awk '{ print $5 " " $7}'
Execution 1:
3069240, 395.41177797317505
Execution 2:
3069240, 474.87467193603516
Execution 3:
3069240, 529.5993568897247
Execution 4:
3069240, 554.3289170265198
Execution 5:
3069240, 563.4688289165497
Execution 6:
3069240, 606.2561638355255
Execution 7:
3069240, 582.9406297206879
Execution 8:
3069240, 531.4921989440918
Execution 9:
3069240, 462.84353518486023
Execution 10:
3069240, 493.5023319721222
```

Now we will reduce the dataset to 100000 (one hundred thousand) lines (using the Unix/Linux "split" command [4]). The intention here is to decrease the amount of time needed to insert the data, as the lab resources are limited. Let us analyze the relationship between the batch size and the insertion time.

```
$ wc -l 202110-citibike-tripdata.csv
3069240 202110-citibike-tripdata.csv

$ split -l 100000 202110-citibike-tripdata.csv
$ ls x*
xaa xab xac xad

$ wc -l xaa
100000 xaa
```

To automate the tests, we slightly changed the previous script to pick the reduced size file (line 34, variable file_path) and iterate over the number of repetitions (line 38, constant REPETITION).

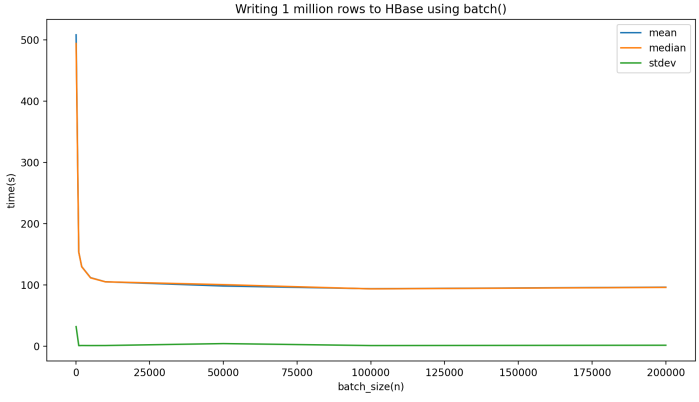


Fig. 13: Simulation results for bulk data insertion to HBase. Credits: author.

Repeating the experiment multiple times (r = 20), but varying the batch sizes from 100 to up to 200000 on

[4]https://man7.org/linux/man-pages/man1/split.1.html

averaged results, have shown that we can only go so fast a little above 93 seconds before we cannot insert any faster. Furthermore, increasing the batch_size did not have a linear increase in the insertion rate (thus reducing the time). The new simulation's measured execution times, mean, median, and standard deviation are shown in the box below. Figure 13 plots the results for a better understanding. The raw data can be found in the links included in Appendix A-E.

Moreover, it is worth mentioning that when we tried any batch size above 5000, our HBase instance started logging warning HBASE-18023[5] during every insertion/commit event: *"The warning message is in the RegionServer log when an RPC is received from a client that has more than 5000 "actions" (where an "action" is a collection of mutations for a specific row) in a single RPC"*.

Batch Size(n)	Mean(s)	Median(s)	Stddev(s)
100	508.501401	494.295124	32.097352
1000	153.770861	153.802854	1.087602
2000	130.065422	129.872940	1.266952
5000	112.004871	111.695448	1.184317
10000	105.308451	105.242499	1.247611
50000	98.384590	100.635257	4.404778
100000	93.979931	93.753836	1.194561
200000	96.461707	96.126112	1.696385

There are many considerations to be taken into account for the results interpretation. First and foremost, the environment where these tests were conducted for this tutorial is very simplistic (docker container running on top of macOS Big Sur, 2.4GHz 8-Core Intel Core i9, 64GB RAM, 1TB SSD disk). If compared to any real-world production (or even testing) environment, in use by enterprises nowadays, the testing environment lacks more resources, it is standalone, is shared with user applications. Nevertheless, it is still worth considering the results, as it is easy to see that the write performances are outstanding for such a big data set with the posed environment restrictions. In addition to that, during the script execution is possible to see the various interactions between the WAL, MemStore, HFiles, and other components we described in Section II-E. We included some of these outputs in Appendix A-D for reference only.

IV. Conclusion

This work highlighted the side effects of the data explosion problem and how the approaches to database systems had to evolve to cope with these challenges' scale and shift in data patterns. Columnar databases are a robust and so far well-tested candidate to deal with massive volumes of data, providing fast writing and retrieval while supporting heterogeneous data types. Furthermore, Hadoop deployment is constantly rising, both for big techies and enterprises, and HBase is the perfect platform for working on top of the powerful and robust HDFS to meet all the big data storage and processing requirements. Whenever the enterprise use case requires random, real-time read/write access to Big Data, Apache HBase is a good fit solution.

[5]https://issues.apache.org/jira/browse/HBASE-18023

Appendix A
HBase Setup

The steps below build a docker container to run HBase (with embedded ZooKeeper) running on the files inside the container.

A. Build Docker Image

```

git clone https://github.com/humbertogaliza/hbase_tutorial.git .
docker build -t dajobe/hbase .
```

B. Run HBase

Use the start-HBase.sh script, which will start the container and inspect it to determine all the local API ports and Web UIs plus will offer to edit /etc/hosts to add an alias for the container IP, if not already present.

```

$ ./start-hbase.sh
start-hbase.sh: Starting HBase container
start-hbase.sh: Container has ID 58172ac65fe1aa566c93c324e3a51607501d443cf92fffd393b8ec8a9103630d
start-hbase.sh: /etc/hosts already contains hbase-docker hostname and IP
start-hbase.sh: Connect to HBase at localhost on these ports
  REST API        127.0.0.1:55022
  REST UI         http://127.0.0.1:55021/
  Thrift API      127.0.0.1:55020
  Thrift UI       http://127.0.0.1:55019/
  Zookeeper API   127.0.0.1:55023
  Master UI       http://127.0.0.1:55018/

start-hbase.sh: OR Connect to HBase on container hbase-docker
  REST API        hbase-docker:8080
  REST UI         http://hbase-docker:8085/
  Thrift API      hbase-docker:9090
  Thrift UI       http://hbase-docker:9095/
  Zookeeper API   hbase-docker:2181
  Master UI       http://hbase-docker:16010/

start-hbase.sh: For docker status:
start-hbase.sh: $ id=58172ac65fe1aa566c93c324e3a51607501d443cf92fffd393b8ec8a9103630d
start-hbase.sh: $ docker inspect $id
```

Figure 14 shows an example of Thrift UI running on the docker container.

C. Check HBase status

Note: assuming the docker container DNS name is 'hbase-docker'

- Master status UI

```
http://hbase-docker:16010/master-status
```

The region server's status pages are linked from the above page.

- Thrift UI

```
http://hbase-docker:9095/thrift.jsp
```

- REST server UI

```
http://hbase-docker:8085/rest.jsp
```

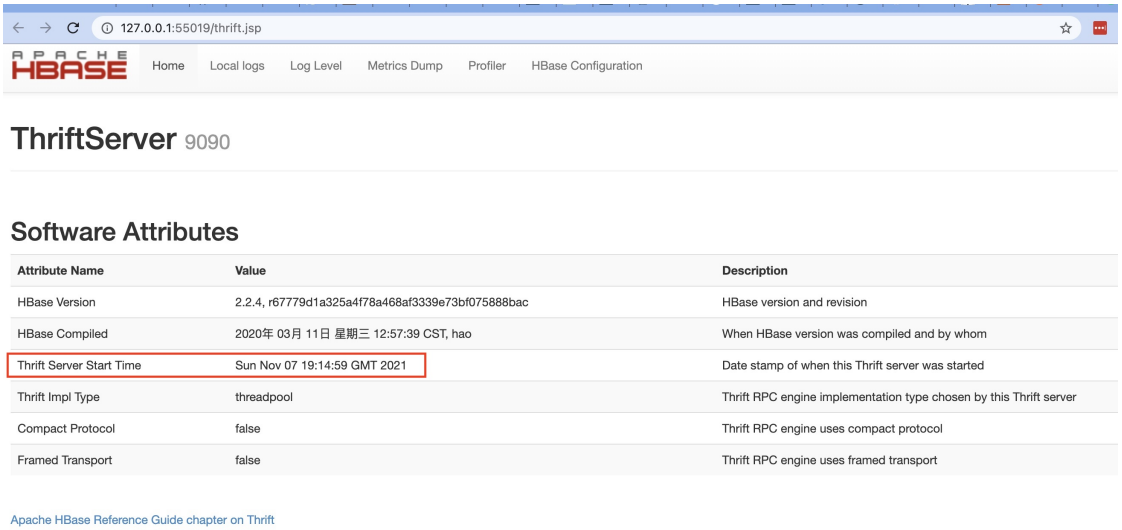


Fig. 14: HBase Thrift UI status page running on local docker container. Credits: author.

- (*Embedded*) Zookeeper status

```
http://hbase-docker:16010/zk.jsp
```

- HBase Logs

To check upon the latest logs live use:

```
$ docker attach $id
```

Control+C to detach.

To see all the logs since the HBase server started, use:

```
$ docker logs $id
```

Control+C to detach.

Example:

```
$ docker logs $id
hbase thrift start logging to /data/logs/hbase-thrift.log
hbase rest start logging to /data/logs/hbase-rest.log
hbase master start logging to /data/logs/hbase-master.log
2021-11-07 19:14:58,803 INFO  [main] master.HMaster: STARTING service HMaster
2021-11-07 19:14:58,806 INFO  [main] util.VersionInfo: HBase 2.2.4
2021-11-07 19:14:58,806 INFO  [main] util.VersionInfo: Source code repository git://hao-OptiPlex
    -7050/home/hao/open_source/hbase revision=67779d1a325a4f78a468af3339e73bf075888bac
2021-11-07 19:14:58,806 INFO  [main] util.VersionInfo: Compiled by hao on 2020? 03? 11? ???
    12:57:39 CST
2021-11-07 19:14:58,807 INFO  [main] util.VersionInfo: From source with checksum 19
    ada8ab3844a5aa8ccaacdd5f2893ca
2021-11-07 19:14:59,210 INFO  [main] master.HMasterCommandLine: Starting a zookeeper cluster
2021-11-07 19:14:59,224 INFO  [main] server.ZooKeeperServer: Server environment:zookeeper.version
    =3.4.10-39d3a4f269333c922ed3db283be479f9deacaa0f, built on 03/23/2017 10:13 GMT
2021-11-07 19:14:59,224 INFO  [main] server.ZooKeeperServer: Server environment:host.name=hbase-
    docker
2021-11-07 19:14:59,224 INFO  [main] server.ZooKeeperServer: Server environment:java.version
    =1.8.0_292
2021-11-07 19:14:59,224 INFO  [main] server.ZooKeeperServer: Server environment:java.vendor=
    Private Build
```

```
2021-11-07 19:14:59,224 INFO  [main] server.ZooKeeperServer: Server environment:java.home=/usr/lib/jvm/java-8-openjdk-amd64/jre
(...)
```

To see the individual log files without using docker, look into the data volume dir e.g.:$PWDdatalogs if invoked as above.

D. Execution logs

Batch size = 1000

```
$ docker attach $id

2021-11-09 13:46:53,344 INFO  [regionserver/hbase-docker:16020.logRoller] wal.AbstractFSWAL: Rolled WAL /data/hbase/WALs/hbase-docker,16020,1636463005353/hbase-docker%2C16020%2C1636463005353.1636465604822 with entries=913, filesize=32.14 MB; new WAL /data/hbase/WALs/hbase-docker,16020,1636463005353/hbase-docker%2C16020%2C1636463005353.1636465613317
2021-11-09 13:46:58,625 INFO  [RS:0;hbase-docker:16020-longCompactions-0] throttle.PressureAwareThroughputController: a8de882b907dffe07580bec196b1d8d4#geo_data#compaction#74 average throughput is 12.36 MB/second, slept 0 time(s) and total slept time is 0 ms. 1 active operations remaining, total limit is 50.00 MB/second
2021-11-09 13:46:58,699 INFO  [RS:0;hbase-docker:16020-longCompactions-0] regionserver.HStore: Completed compaction of 3 (all) file(s) in geo_data of a8de882b907dffe07580bec196b1d8d4 into df8b93a5fc7a448ca34f85a1a9904b44(size=193.9 M), total size for store is 193.9 M. This selection was in queue for 0sec, and took 15sec to execute.
2021-11-09 13:46:58,699 INFO  [RS:0;hbase-docker:16020-longCompactions-0] regionserver.CompactSplit: Completed compaction region=sample_data:tripdata,800063EA8503B6A5,1636465401462.a8de882b907dffe07580bec196b1d8d4., storeName=geo_data, priority=13, startTime=1636465603683; duration=15sec
2021-11-09 13:46:59,241 INFO  [RS:0;hbase-docker:16020-shortCompactions-0] throttle.PressureAwareThroughputController: a8de882b907dffe07580bec196b1d8d4#ride_data#compaction#75 average throughput is 12.92 MB/second, slept 0 time(s) and total slept time is 0 ms. 0 active operations remaining, total limit is 50.00 MB/second
2021-11-09 13:46:59,308 INFO  [RS:0;hbase-docker:16020-shortCompactions-0] regionserver.HStore: Completed compaction of 3 file(s) in ride_data of a8de882b907dffe07580bec196b1d8d4 into 6c850f2bf2734afeb9c41ff82cf4d713(size=141.2 M), total size for store is 431.9 M. This selection was in queue for 0sec, and took 10sec to execute.
2021-11-09 13:46:59,308 INFO  [RS:0;hbase-docker:16020-shortCompactions-0] regionserver.CompactSplit: Completed compaction region=sample_data:tripdata,800063EA8503B6A5,1636465401462.a8de882b907dffe07580bec196b1d8d4., storeName=ride_data, priority=12, startTime=1636465603683; duration=10sec
2021-11-09 13:47:01,739 INFO  [regionserver/hbase-docker:16020.logRoller] wal.AbstractFSWAL: Rolled WAL /data/hbase/WALs/hbase-docker,16020,1636463005353/hbase-docker%2C16020%2C1636463005353.1636465613317 with entries=915, filesize=32.10 MB; new WAL /data/hbase/WALs/hbase-docker,16020,1636463005353/hbase-docker%2C16020%2C1636463005353.1636465621711
```

Batch Size = 100000

```
$ docker attach $id
2021-11-09 14:19:18,328 WARN  [RpcServer.default.FPBQ.Fifo.handler=29,queue=2,port=16020] regionserver.RSRpcServices: Large batch operation detected (greater than 5000) (HBASE-18023). Requested Number of Rows: 8334 Client: root//172.17.0.2 first region in multi=sample_data:tripdata,,1636465401462.836187c88cfcf63b28fd353759cb8c20.
2021-11-09 14:19:19,371 WARN  [RpcServer.default.FPBQ.Fifo.handler=29,queue=2,port=16020] regionserver.RSRpcServices: Large batch operation detected (greater than 5000) (HBASE-18023). Requested Number of Rows: 8334 Client: root//172.17.0.2 first region in multi=sample_data:tripdata,,1636465401462.836187c88cfcf63b28fd353759cb8c20.
2021-11-09 14:19:20,337 WARN  [RpcServer.default.FPBQ.Fifo.handler=29,queue=2,port=16020] regionserver.RSRpcServices: Large batch operation detected (greater than 5000) (HBASE-18023). Requested Number of Rows: 8334 Client: root//172.17.0.2 first region in multi=sample_data:tripdata,,1636465401462.836187c88cfcf63b28fd353759cb8c20.
```

```
2021-11-09 14:19:21,300 WARN  [RpcServer.default.FPBQ.Fifo.handler=29,queue=2,port=16020]
    regionserver.RSRpcServices: Large batch operation detected (greater than 5000) (HBASE-18023).
     Requested Number of Rows: 8334 Client: root//172.17.0.2 first region in multi=sample_data:
    tripdata,,1636465401462.836187c88cfcf63b28fd353759cb8c20.
2021-11-09 14:19:21,537 INFO  [MemStoreFlusher.1] regionserver.MemStoreFlusher: Flush of region
    sample_data:tripdata,800063EA8503B6A5,1636465401462.a8de882b907dffe07580bec196b1d8d4. due to
    global heap pressure. Flush type=ABOVE_ONHEAP_LOWER_MARK, Total Memstore Heap size=183.1 M,
    Total Memstore Off-Heap size=0, Region memstore size=100.5 M
2021-11-09 14:19:21,537 INFO  [MemStoreFlusher.1] regionserver.HRegion: Flushing
    a8de882b907dffe07580bec196b1d8d4 2/3 column families, dataSize=47.85 MB heapSize=100.53 MB;
    ride_data={dataSize=25.24 MB, heapSize=51.71 MB, offHeapSize=0 B}; geo_data={dataSize=22.60
    MB, heapSize=48.82 MB, offHeapSize=0 B}
```

E. Raw data from experiments

The script source code as well as the raw data results in JSON format can be visualized/downloaded at the project GitHub repository testing directory: https://github.com/humbertogaliza/hbase_tutorial/tree/main/testing

Appendix B
Google Trends: exploring insights for columnar databases

We leveraged the "pytrends" [6] a Python library to build a script and gather the search data from Google Trends. We sought for the Top 20 columnar-database search terms, according to the DB-Engine ranking [13], aiming to combine both sources (Google Trends & DB-Engines) and get the most Top 10 popular columnar-databases for October/2021. The script below is based on the ideas outlined in [26].

```
import pandas as pd
import matplotlib
from pytrends.request import TrendReq
from matplotlib import pyplot as plt

pytrends = TrendReq()
search_list = [
    ["Apache Accumulo", "Apache Cassandra", "Apache Elassandra", "Apache HBase", "ScyllaDB"],
    ["Google Bigtable", "Amazon RedShift", "Azure Table Storage", "DataStax Enterprise", "HPE Ezmeral Data Fabric"],
    ["Microsoft Azure Cosmos DB", "Amazon Keyspaces", "Alibaba Cloud Table Store", "Hypertable", "Sqrrl"],
    ["Amazon DynamoDB", "Cloudera"]
    ] #max of 5 values allowed per query
df_ibr = pytrends.interest_by_region(resolution='COUNTRY') # CITY, COUNTRY or REGION
df_ibr.sort_values('Oracle', ascending=False).head(20)
df2 = df_ibr.sort_values('Oracle', ascending=False).head(20)
df2.reset_index().plot(x='geoName', y=['Oracle', 'MySQL', 'SQL Server', 'PostgreSQL', 'MongoDB'],
     kind ='bar', stacked=True, title="Searches by Country")
plt.rcParams["figure.figsize"] = [20, 8]
plt.xlabel("Country")
plt.ylabel("Ranking")
```

```
$ docker attach $id
2021-11-09 14:19:18,328 WARN  [RpcServer.default.FPBQ.Fifo.handler=29,queue=2,port=16020]
    regionserver.RSRpcServices: Large batch operation detected (greater than 5000) (HBASE-18023).
     Requested Number of Rows: 8334 Client: root//172.17.0.2 first region in multi=sample_data:
    tripdata,,1636465401462.836187c88cfcf63b28fd353759cb8c20.
2021-11-09 14:19:19,371 WARN  [RpcServer.default.FPBQ.Fifo.handler=29,queue=2,port=16020]
    regionserver.RSRpcServices: Large batch operation detected (greater than 5000) (HBASE-18023).
     Requested Number of Rows: 8334 Client: root//172.17.0.2 first region in multi=sample_data:
    tripdata,,1636465401462.836187c88cfcf63b28fd353759cb8c20.
2021-11-09 14:19:20,337 WARN  [RpcServer.default.FPBQ.Fifo.handler=29,queue=2,port=16020]
    regionserver.RSRpcServices: Large batch operation detected (greater than 5000) (HBASE-18023).
     Requested Number of Rows: 8334 Client: root//172.17.0.2 first region in multi=sample_data:
    tripdata,,1636465401462.836187c88cfcf63b28fd353759cb8c20.
2021-11-09 14:19:21,300 WARN  [RpcServer.default.FPBQ.Fifo.handler=29,queue=2,port=16020]
    regionserver.RSRpcServices: Large batch operation detected (greater than 5000) (HBASE-18023).
     Requested Number of Rows: 8334 Client: root//172.17.0.2 first region in multi=sample_data:
    tripdata,,1636465401462.836187c88cfcf63b28fd353759cb8c20.
2021-11-09 14:19:21,537 INFO  [MemStoreFlusher.1] regionserver.MemStoreFlusher: Flush of region
    sample_data:tripdata,800063EA8503B6A5,1636465401462.a8de882b907dffe07580bec196b1d8d4. due to
    global heap pressure. Flush type=ABOVE_ONHEAP_LOWER_MARK, Total Memstore Heap size=183.1 M,
    Total Memstore Off-Heap size=0, Region memstore size=100.5 M
2021-11-09 14:19:21,537 INFO  [MemStoreFlusher.1] regionserver.HRegion: Flushing
    a8de882b907dffe07580bec196b1d8d4 2/3 column families, dataSize=47.85 MB heapSize=100.53 MB;
    ride_data={dataSize=25.24 MB, heapSize=51.71 MB, offHeapSize=0 B}; geo_data={dataSize=22.60
    MB, heapSize=48.82 MB, offHeapSize=0 B}
```

[6] https://pypi.org/project/pytrends/

References

[1] Merriam-Webster, *Big data*, in *Merriam-Webster.com Dictionary*. [Online]. Available: https://www.merriam-webster.com/dictionary/big%5C%20data (visited on 09/29/2021), (accessed: 29.09.2021).

[2] J. Hurwitz, A. Nugent, F. Halper, and M. Kaufman, "Big data," *New York*, 2013. [Online]. Available: http://www.dummies.com/how-to/content/big-data-for-dummies-cheat-sheet.html (visited on 10/01/2021), (accessed: 23.10.2021).

[3] Sisense, *Oltp*, in *Sisense glossary*. [Online]. Available: https://www.sisense.com/glossary/oltp/ (visited on 09/29/2021), (accessed: 29.09.2021).

[4] M. Stonebraker and U. Çetintemel, "" one size fits all" an idea whose time has come and gone," in *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, 2018, pp. 441–462.

[5] A. Griffith. "Deciding between row- and columnar-stores | why we chose both." (), [Online]. Available: https://medium.com/bluecore-engineering/deciding-between-row-and-columnar-stores-why-we-chose-both-3a675dab4087 (visited on 08/10/2020). (accessed: 18.10.2021).

[6] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[7] L. George, *HBase: the definitive guide: random access to your planet-size data*. "O'Reilly Media, Inc.", 2011.

[8] Timepasstechies. "Timepasstechies blog - big data tutorial." (), [Online]. Available: https://timepasstechies.com/row-oriented-column-oriented-file-formats-hadoop/ (visited on 10/23/2021). (accessed: 23.10.2021).

[9] D. Ocean. "Understanding database sharding." (), [Online]. Available: https://www.digitalocean.com/community/tutorials/understanding-database-sharding (visited on 10/23/2021). (accessed: 23.10.2021).

[10] A. Petrov, *Database internals: A deep dive into how distributed data systems work*, 2019.

[11] Scylla, *Wide-column database*, in *ScyllaDB glossary*. [Online]. Available: https://www.scylladb.com/glossary/wide-column-database/ (visited on 10/25/2021), (accessed: 25.10.2021).

[12] Rhodes. "Hbase - cs 305 syllabus." (), [Online]. Available: http://jcsites.juniata.edu/faculty/rhodes/smui/hbase.htm (visited on 10/23/2021). (accessed: 23.10.2021).

[13] DB-Engines. "Db-engines ranking - trend of wide column stores popularity." (), [Online]. Available: https://db-engines.com/en/ranking_trend/wide+column+store (visited on 10/23/2021). (accessed: 23.10.2021).

[14] F. Chang, J. Dean, S. Ghemawat, *et al.*, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[15] L. Perkins, E. Redmond, and J. Wilson, *Seven databases in seven weeks: a guide to modern databases and the NoSQL movement*. Pragmatic Bookshelf, 2018.

[16] Edureka. "Hadoop tutorial." (), [Online]. Available: https://www.edureka.co/blog/hadoop-tutorial/ (visited on 10/23/2021). (accessed: 23.10.2021).

[17] M. Chen, S. Mao, Y. Zhang, V. C. Leung, *et al.*, "Big data: Related technologies, challenges and future prospects," 2014.

[18] Edureka. "Hbase tutorial: Hadoop database." (), [Online]. Available: https://www.tutorialspoint.com/hbase/hbase_overview.htm (visited on 10/04/2021). (accessed: 04.10.2021).

[19] DataScienceCentral.com. "Hdfs vs. hbase : All you need to know - data science central." (), [Online]. Available: https://www.datasciencecentral.com/xn/detail/6448529:BlogPost:610315 (visited on 09/29/2021). (accessed: 29.09.2021).

[20] A. HBase. "Apache hbase reference guide." (2021), [Online]. Available: https://hbase.apache.org/book.html (visited on 11/08/2021). (accessed: 08.11.2021).

[21] N. Garg, *HBase Essentials*. Packt Publishing Ltd, 2014.

[22] OpenTSDB. "Opentsdb 2.4 documentation - user guide - storage - hbase schema." (2021), [Online]. Available: http://opentsdb.net/docs/build/html/user_guide/backends/hbase.html (visited on 11/09/2021). (accessed: 09.11.2021).

[23] S. Shriparv, *Learning HBase*. Packt Publishing Ltd, 2014.

[24] towardsdatascience.com. "Hbase working principle: A part hadoop architecture." (), [Online]. Available: https://towardsdatascience.com/hbase-working-principle-a-part-of-hadoop-architecture-fbe0453a031b (visited on 09/29/2021). (accessed: 29.09.2021).

[25] C. Blog. “How-to: Use the hbase thrift interface, part 1.” (), [Online]. Available: https://blog.cloudera.com/how-to-use-the-hbase-thrift-interface-part-1/ (visited on 10/07/2021). (accessed: 07.10.2021).

[26] “Exploring database trends using python pytrends (google trends).” (), [Online]. Available: https://oralytics.com/2020/11/09/exploring-database-trends-using-python-pytrends-google-trends/. (accessed: 20.10.2021).