Column-based Databases and HBase

- Advanced Databases project -

Project Report

Edoardo Conte
Carlos Muñiz Cuza

Université Libre de Bruxelles

Université Libre de Bruxelles
http://www.ulb.ac.be

Title:
Column-based Databases and HBase for Logs Stores

Course:
Advance Databases

Project Period:
Fall Semester 2018

Project group:

Edoardo Conte
Carlos Muñiz Cuza

Supervisor(s):
Esteban Zimányi

Page Numbers: 30

Date of Completion:
December 17, 2018

Abstract:

This report presents a toy application implemented using the column-oriented database HBase. The database is supposed to store big number of logs of a software company and retrieve this information taking into account specific attributes. The choice of HBase as the DBMS was subtended in the idea that traditional relational database where not able to fulfill the task due to the unstructured nature of the data and the impossibility to retrieve huge volume of data in a short period of time.
This report shows each process of the implementation of the problem. From choosing a suitable row key and column families to the Java implementation. In the first part, some details about the choice of a secondary index are also provided; for the second part, we also show the implementation of concurrent threads to insert the data.
The final application fulfills the expectation running really fast over 1Gb of information. Further experiments are required in order to measure the distributed power of HBase and to calculate performance metric against traditional databases.

Contents

Introduction

Nowadays, everything need to be store as potential information. Also, it is expected to get access to this information instantaneously. Moreover, it is also expected the results to be useful and tailored to our needs. In this context, companies need a cost-effective way to store all that data.

Google and Amazon are prominent examples of companies that realized the value of data and started developing solutions to fit their needs. For instance, in a series of technical publications, Google described a scalable storage and processing system based on commodity hardware. These ideas were then implemented outside of Google as part of the open source Hadoop project: HDFS and MapReduce[2]. Also the advent of NoSQL database open the spectrum to handle unstructured data throwing out the limiting factors to obtain truly scalable systems.

The goal of this project is to explore a NoSQL database called column-oriented databases through a real application, with a particular focus on Apache HBase. The application has the goal to show all the main features of the database emphasizing its strength points. However, it also tries to show the main weaknesses of the technology, giving an idea of HBase limitations. During the discussion, a comparison with a similar representation in a relational database is presented at a logical level.

In order to give a full overview of the technology, we divided the report in two main chapters. The first one deals with the general concepts related to column-oriented databases and how they are applied in HBase. It also deeply analyses the structure of the database, exploring some features that are not necessarily present in every columnar database. The second chapter, describes the implementation of the application in self.

Chapter 1

Technologies Fundamentals

In this section we are going to explore the concept of columnar database making emphasis in the particular case of HBase. Understanding the logical and physical data model of this technology deeply could make the difference between a bad design and a very good one. Also understanding the benefits with respect to traditional database is important in this context. This is why we go in detail about how HBase stores the data and how it reads and writes it. In order to do that, a brief explanation about Hadoop file system is also presented. Finally, an explanation about how HBase computes MapReduce is also given.

1.1 Column-based Database Management Systems

A Column-based Database Management System is a database management system that stores data tables by column rather than by row. What this means is that each column or family of columns is stored in a different physical file. This have several advantages like allowing data compression. But the most important aspect is that we have more precisely access to the data needed to answer a query rather than scanning and discarding unwanted data in rows. Table 1.1 shows a User table storing information in a row based schema. In the physical model we will then find the same, a list of different rows with a predefined structure per block. On the opposite, a column-oriented database serializes all values of a column together, then the values of the next column and so on, as shown in table 1.2 .

Some thoughts come up just by looking the table 1.2. The first one is that in order to retrieve the name of the user in the table we do not need to load in memory the last name, the IDs, or the Salary. The second idea is that not Null value is kept in the file system as difference of the row based database. This new type of database are known as schemaless, which means that it does not need a predefined structure or data type for each of the attributes. Instead, records can have divergent columns, variance in field size, and so on. This kind of data is said

RowID	EmpID	LastName	FirstName	Salary
001	10	Muñiz	Carlos	1000
002	21	Conte	Edoardo	2000
003	13	Arora	Kunal	Null
004	35	Patrascu	Eugen	5000

Table 1.1: Row based schema of User table.

10,12,11,22;
Muñiz,Conte,Arora,Patrascu;
Carlos,Edoardo,Kunal,Eugen;
1000,2000,5000;

Table 1.2: Columnar based store of User table.

to have a semi-structured shape. Understanding this assumption in the logical model is very important to take advantage of the physical implementation of the database system manager.

In addition semi-structure data brings the opportunity of scaling. The loose coupling of data components in a semi-structured logical model has the benefit of being easier to physically distribute [1]. Of course, the lack of structure forces to give up to some important characteristics like relational constraints.

This is the basic idea of columnar database, but it does not explain how it really works. Next section will answer those questions using HBase logical and physical data model as reference.

1.1.1 Logical and physical data model: HBase

In table 1.2 presented an example of how a columnar database would be having each column stored in a different file. However, this is just a brief idea of the physical model. Normally we do not specify in which file we want the data to be stored. In particular we just need to create a table and a set of column families. The column families are the basic unit of store in HBase. At the same time, each of them keeps a set of rows which at the same time contains different columns (also know as column qualifiers). The column families are specified from the beginning taking into account the information to store. Rows with related columns may want to be store under the same column family. Figure 1.1 show this idea in a general perspective using *Keyspace* as the notation for the table.

As explained before on the level below the column families are rows contained in them. Each row has an unique ID that is used as primary key. Also, we can access to particular columns of this row in a specific column family. In addition, rows within the same column family are not required to have the same set of

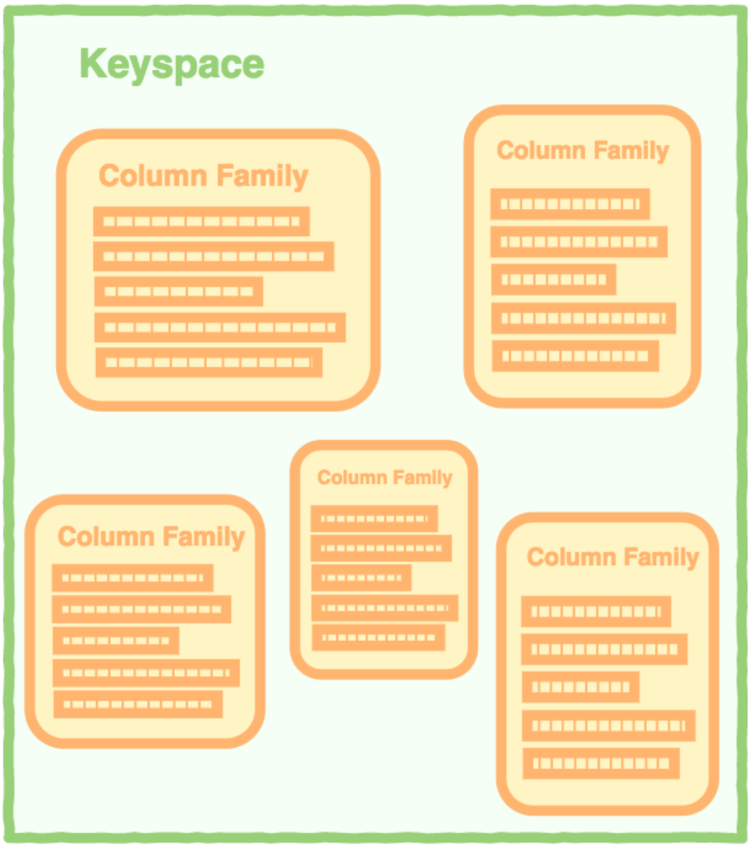


Figure 1.1: A keyspace containing columns families, picture taken from https://database.guide/what-is-a-column-store-database/.

columns. Each column contains a name/value pair, along with a timestamp (also know as version). This provides the date and time that the data was inserted. Then every time we insert a new element in the column even if the name and value are equal a new entry is going to be created with a new timestamp. Figure 1.2 shows an example where each row have different columns type.

In order to make it more clear next we explicitly mention the logical schema entities for the particular case of HBase and some other characteristics of each of them[1].

- **Table:** HBase organizes data into tables. Table names are Strings and composed of characters that are safe for use in a file system path.
- **Row:** Within a table, data is stored according to its row. Rows are identified uniquely by their rowkey. Rowkeys do not have a data type and are always treated as a byte[].
- **Column family:** Data within a row is grouped by column family. Column

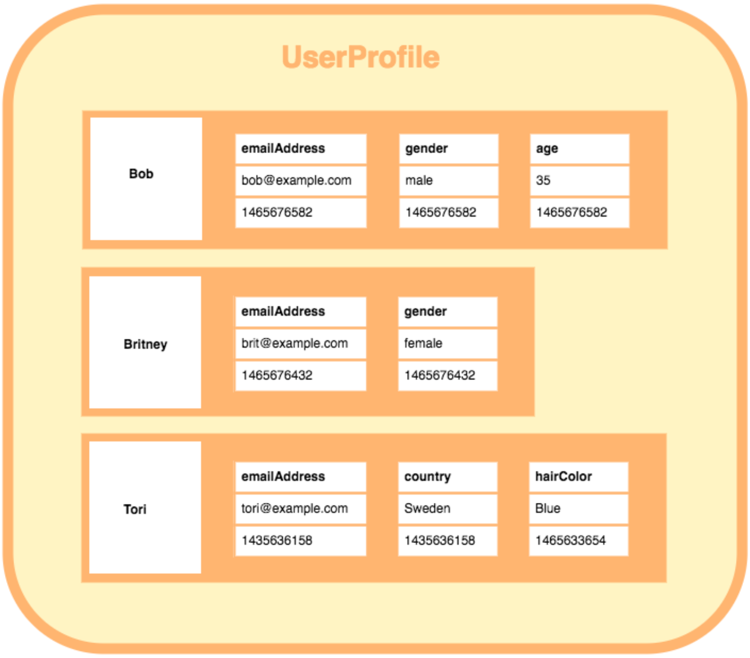


Figure 1.2: A column family containing 3 rows. Each row contains its own set of columns, picture taken from https://database.guide/what-is-a-column-store-database/.

families also impact the physical arrangement of data stored in HBase. In fact each column family is store separately from each other allowing to load into memory only the columns in specifics column family without loading the others. For this reason, they must be defined up front and are not easily modified. Every row in a table has the same column families, although a row need not store data in all its families. Column family names are Strings and composed of characters that are safe for use in a file system path.

- **Column qualifier:** Data within a column family is addressed via its column qualifier, or column. Column qualifiers need not be specified in advance. Column qualifiers need not be consistent between rows. Like rowkeys, column qualifiers do not have a data type and are always treated as a byte[].
- **Cell** A combination of rowkey, column family, and column qualifier uniquely identifies a cell. The data stored in a cell is referred to as that cell's value. Values also do not have a data type and are always treated as a byte[].
- **Version:** Values within a cell are versioned. Versions are identified by their timestamp, a long. When a version is not specified, the current timestamp is used as the basis for the operation.

As stated bellow, all data type in HBase are byte, so no data type. This provides

HBase of the ability to keep the records sorted. Knowing this is very important when creating the *rowkey* and computing the queries. Understanding this detail allows to design the schema to take advantage of this feature. Also this fact allow us to understand HBase as an *ordered map of maps*. This is because each of the entities describe above can be seen as a coordinate (except the cell in self). Figure 1.3 will help us to understand this idea.

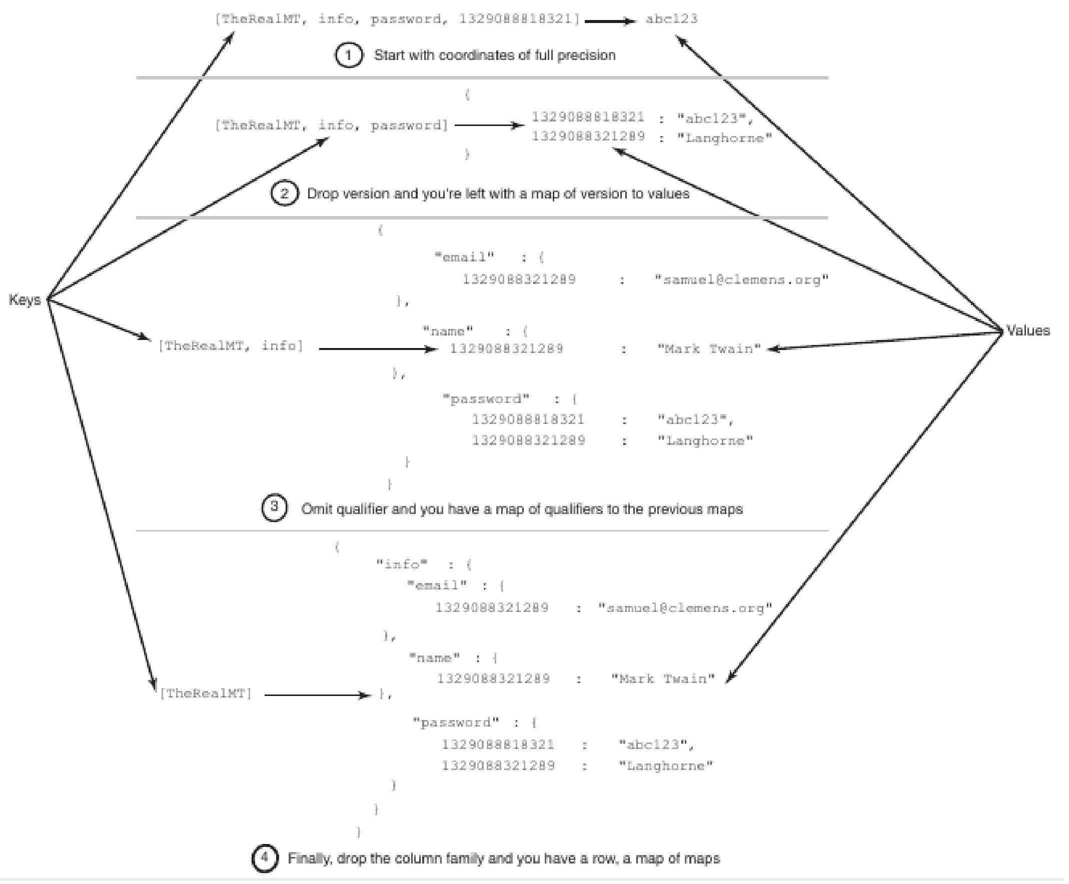

[TheRealMT, info, password, 1329088818321] abc123
1 Start with coordinates of full precision
[TheRealMT, info, password]
{ 1329088818321 : "abc123", 1329088321289 : "Langhorne" }
2 Drop version and you're left with a map of version to values
Keys
[TheRealMT, info]
{ "email" : { 1329088321289 : "samuel@clemens.org" }, "name" : { 1329088321289 : "Mark Twain" }, "password" : { 1329088818321 : "abc123", 1329088321289 : "Langhorne" } }
3 Omit qualifier and you have a map of qualifiers to the previous maps
Values
[TheRealMT]
{ "info" : { "email" : { 1329088321289 : "samuel@clemens.org" }, "name" : { 1329088321289 : "Mark Twain" }, "password" : { 1329088818321 : "abc123", 1329088321289 : "Langhorne" } } }
4 Finally, drop the column family and you have a row, a map of maps

Figure 1.3: Alternative view of the physical model of HBase as map of maps [1].

Consider those coordinates from the inside out. The cell could be consider as a map keyed on version with the stored data as the value. One layer up, a column family is a map keyed on column qualifier with the cell as the value. At the top, a table is a map keyed on rowkey to the column family. Described in Java, we could have this: $Map < RowKey, Map < ColumnFamily, Map < ColumnQualifier, Map < Version, Data >>>>$ [1].

1.1.2 Benefits

- **Query speed:** Columnar databases boost performance by reducing the amount of data that needs to be read from disk, both by efficiently compressing the similar columnar data and by reading only the data necessary to answer the query.
- **Aggregation queries:** Due to their structure, columnar databases perform particularly well with aggregation queries (such as SUM, COUNT, AVG, etc).
- **Scalability:** Columnar databases are very scalable. They are well suited to massively parallel processing (MPP), which involves having data spread across a large cluster of machines often thousands of machines.

1.2 HBase Fundamentals: Distributing Data

Along this section and the next ones, we mainly took the theoretical information from [1]. In the previous section it has been explained how HBase technology effectively stores data and how this solution affects the logical data model of the database. In this section, the architecture of an HBase deployment will be analyzed in detail. HBase is primarily used to store large dataset, having fast random read-/write access to it. When the amount of data becomes significant, it is therefore really difficult to store it on a single machine. The architecture needs to be backed by a high reliable and scalable distributed system. This section explains in deep the underlying architecture of an HBase cluster.

1.2.1 Apache Hadoop

Hadoop is a project currently maintained by the Apache Software Foundation. It is widely used in the field of Big Data and it is the underlying structure for several data management application. It can be considered as a framework mainly composed of three elements:

- Hadoop YARN
- Hadoop MapReduce
- Hadoop File System (HDFS)

YARN is a resource manager. It is used in order to schedule resource in the Hadoop cluster. MapReduce is the core computational element of the Hadoop framework. It can be considered as a functional programming paradigm, where each task is achieved through the completion of a Map and a Reduce job. In order to retrieve and batch process large amount of data, stored in a distributed file system, it is needed to use a highly parallel approach. Through the Map jobs, the user specifies a method in which the data is mapped from different data sources. Then, this result is forwarded to the Reduce jobs, which reconstruct the data again. This processes are executed by each server in the cluster. The two jobs will run only on partial datasets, then, the results are reconstructed in the reduce phase.

The HDFS is a fault tolerant and highly available distributed storage technology. It allows fast retrieval of data on large dataset and it is designed to run on commodity hardware. Each file stored in the HDFS is divided in blocks of equal size.

The Hadoop File System is made of a master-slaves architecture, where the master is called the NameNode and the slaves are the DataNodes. It is important to note that user data only resides in DataNodes.

The NameNode has several task, one of which is managing the namespaces of the file system. By accessing different namespaces, it is possible to access different directories and file structures. Moreover, the NameNode knows where each block is stored in the DataNodes. Along with this information, it keeps all the needed metadata related to each stored file. These are the only information that the NameNode stores. However, it has several additional tasks for the correct functioning of the HDFS. It manages the communication between the DataNodes, it guarantees load balancing and it detects failures in the slaves cluster, replicating and substituting data. DataNodes, instead, provide data requested by the user through low-level read and write operations. Moreover, they can run on commodity hardware and they do not need to be highly available.

Good practice is to dedicate a single machine for the NameNode and one machine for each DataNode.

It can be noted that this architecture gives to the system a single point of failure. If the NameNode crashes or becomes unavailable, all the system can not be used. HDFS designed a solution to this problem, adding a new component to the current architecture, the Secondary NameNode. This node is responsible for getting metadata from NameNode memory and save it on the file system. Moreover, it combines this information with the current state of the file system, saved in the disk of the NameNode. With this information, it creates the new state of the distributed file system, saving it back on the NameNode.

Through this procedure, even if the NameNode fails, all the meta-data is stored safely on the file system until it restarts.

1.2.2 HBase architecture

HBase is a database built on top of the HDFS. As explained in the previous section, HDFS is a distributed system able to store efficiently large files. HBase has itself a distributed master-slave architecture, where data in HDFS becomes logically organized in order to guarantee query efficiency. HBase architecture is mainly composed of three server types:

- Master Server
- Region Server
- Zookeeper

The master process, called HMaster, is entitled of DDL operations and region assignments. A region is a portion of a table, with a fixed maximum dimension, which is entirely stored in a region server. Due to a load balancer inside the HMaster, it is able to move and organize regions in a smart way. Region servers, moreover, accept client requests, serving data. They are entitled of executing query over the region they own, sending back the results. In order to be efficient with data retrieval, they need to be physically close to data location. By working on top of the HDFS, it is a best practice to run region servers on the same machine which is running the DataNode. The zookeeper process, instead, is entitled of maintaining an active state of the network. Through a periodic heartbeat, it knows when a node has failed and communicates it to the HMaster in order to restore the alive situation.

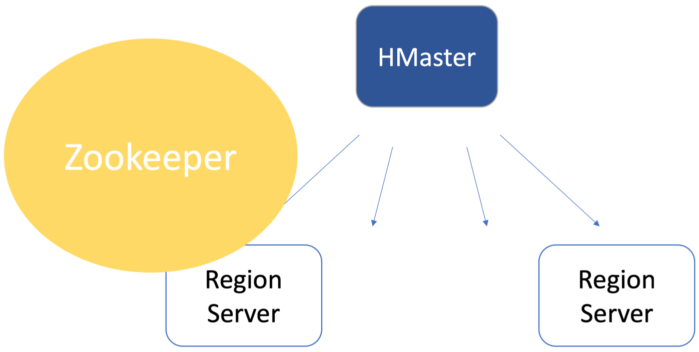


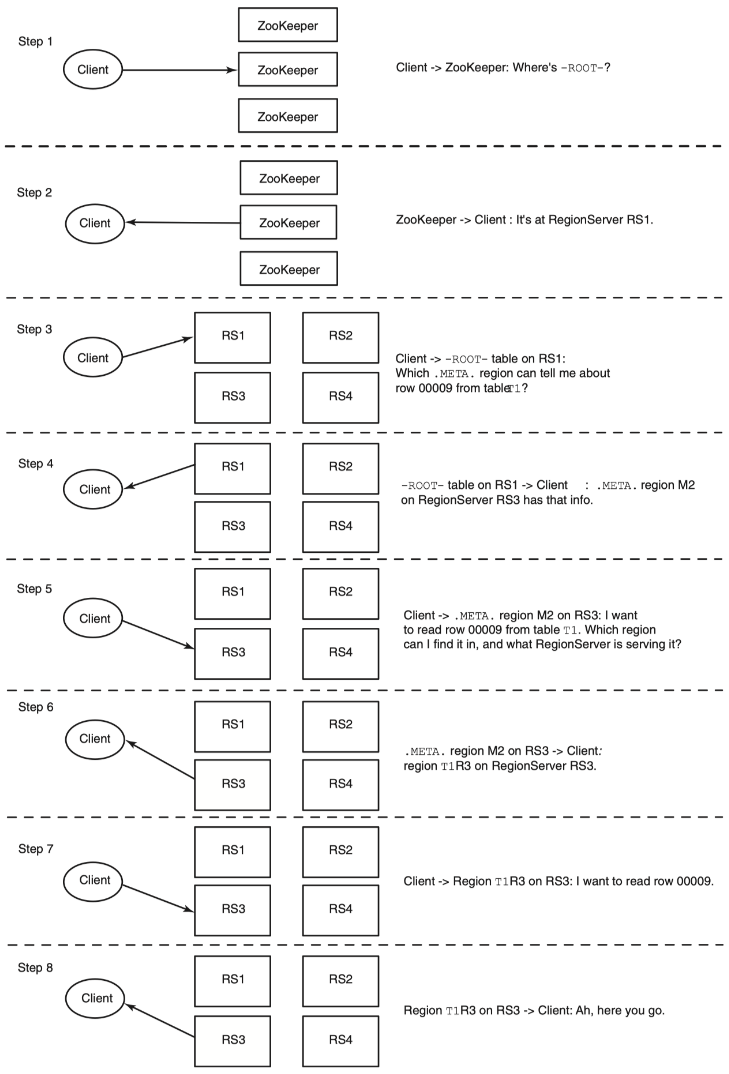
Figure 1.4: HBase architecture.

1.2.3 Transaction Flow

In the section above, the HBase architecture has been described briefly. Now, through the subsequent sections, we are going to deeply analyze it through the practical operations the client may do. In practice, we are going to analyze read and write operations, basic ways to interact with the database. First, the client may want to know where it has to insert or read data. This procedure is common for both operation, so it leaves large possibilities for caching. We are going to learn in the Java client set up that the interaction happens mainly through the Zookeeper server. This means that the only information the client must know is the IP address of that server and the port to interact with (typically 2181). After this operation, the Zookeeper server gives to the client the location of the table named -ROOT-, which is also a region. We briefly introduced the term region. A region is a portion of data stored on a RegionServer. The main characteristic of regions is that they only store data belonging to a specific table and column family. The format has been explained in section [link]. Whenever a region grows too big, the HMaster may decide to split it, assigning its pieces to different RegionServers. In this configuration, data is distributed across RegionServers, which themselves work with HDFS to maintain consistency and availability of information. In this context, where each table is divided in different regions, the system allocates space for two special kind of tables. They are the -ROOT- and the .META. tables. While the .META. table behaves like a normal table, the -ROOT- table has the characteristic to not divide. This means, that the HMaster can never split the -ROOT- table in different RegionServers. Thanks to these tables, the client may find the right region in which reading/inserting the data. Next section will explain this in detail.

1.2.4 Indexing, retrieving the right region

In previous section it has been explained the data model of HBase. It presents itself as a database with random read/write access to big data. This characteristic makes it a suitable choice for an operational database which has to handle a considerable amount of data. In the current section, it is explained how effectively HBase achieves this result. It has already been shown that data are indexed by the rowKey. This is a unique value used to distinguish among different rows. However, an interesting characteristic is that these values are sorted. This implies that whenever a client wants to retrieve information related to several rows with adjacent rowKeys (alphabetically), the execution time would be significantly optimized. In short words, this structure allows to have fast search and range queries. Whenever the client requests for a certain range of rowKeys, this can be retrieved very efficiently looking in the same location. Starting from the knowledge of -ROOT- and .META. tables, it is possible to understand how effectively this system works. In order to open a connection to HBase, the client must ask to Zookeeper the position of the -ROOT- table. After this information is given, the client will ask to the corresponding RegionServer the location of the region of interest. It will query the -ROOT- table giving as input the range of rowKeys it is interested in and the server will return the corresponding .META. region which contains that information. The .META. table contains associations between rowKey ranges and corresponding region locations. As said before, it behaves like a normal table, therefore, it may be split in several regions. After the client receives the location of the .META. region, it queries the corresponding RegionServer to finally discover the location of the



Step 1
Client
ZooKeeper
ZooKeeper
ZooKeeper
Client -> ZooKeeper: Where's -ROOT-?
Step 2
Client
ZooKeeper
ZooKeeper
ZooKeeper
ZooKeeper -> Client : It's at RegionServer RS1.
Step 3
Client
RS1
RS2
RS3
RS4
Client -> -ROOT- table on RS1: Which .META. region can tell me about row 00009 from table T1?
Step 4
Client
RS1
RS2
RS3
RS4
-ROOT- table on RS1 -> Client : .META. region M2 on RegionServer RS3 has that info.
Step 5
Client
RS1
RS2
RS3
RS4
Client -> .META. region M2 on RS3: I want to read row 00009 from table T1. Which region can I find it in, and what RegionServer is serving it?
Step 6
Client
RS1
RS2
RS3
RS4
.META. region M2 on RS3 -> Client: region T1R3 on RegionServer RS3.
Step 7
Client
RS1
RS2
RS3
RS4
Client -> Region T1R3 on RS3: I want to read row 00009.
Step 8
Client
RS1
RS2
RS3
RS4
Region T1R3 on RS3 -> Client: Ah, here you go.

Figure 1.5: Description of HBase transaction flow [1].

region requested. Querying the last RegionServer, the client is now able to insert or read information in the right location. This is necessary in order to maintain an overall order of rowKeys. Looking at this process from above, we can immediately recognize this structure to be similar to a B+tree with fixed height of 3. The -ROOT- region (table) is always the root, then the first layer is represented by all the .META. regions and then, the leaves of the tree are the regions of the user table, sorted by rowKey. This strucure also well explains why search and range queries are so efficient in HBase. It is also important to note that in such an architecture, it becomes expensive to add new regions and/or move them. These operations are supposed to be executed by the HMaster in not heavy-load conditions. Having talked about the indexing structure of HBase, now it is possible to complete the transaction flow, analyzing how read and write operations are performed on the database.

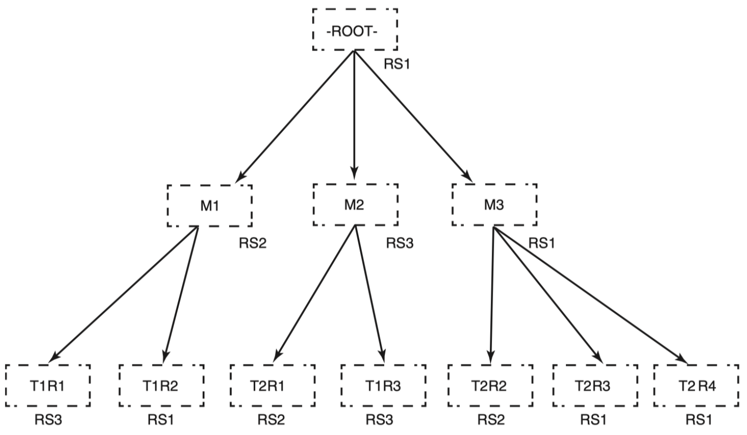


Figure 1.6: Visual explanation of the index structure exploited by HBase[1].

1.2.5 Write Path

When the client needs to write some data to HBase, the following transaction flow is activated. Recalling the structure of the database, the client must first specify the table he wants to modify and then the interested column family. It has already been shown that data are organized internally by means of the column families. Each physical table represents one column family, where each row is made of a row key, a timestamp and a column name with the corresponding value. Whenever a client requests to write some data to HBase, it has to specify the table name, column family and all the columns it wants to insert. Through the CLI, the PUT command is responsible for inserting or updating a row. Internally, insertion and update are exactly the same operation. Recalling the indexing architecture, the client is able to retrieve the location of the correct region. This is given by the combination of the table name, the column family and the rowKeys range. The corresponding RegionServer is responsible for data insertion. A write operation is completed after the server writes the information to the Write Ahed Log (WAL) and to the MemStore. In order to maintain consistency, in case of crashes or unexpected shut down, the system needs to have a log with recent changes, not yet propagated. This is the role of the WAL. It is a file on disk, storing all the changes not yet persisted on the database. The MemStore, instead, is a in-memory table and can be considered as a buffer. It maintains the same structure of a region, therefore, it can be associated to only one column family and each row contains rowKey, timestamp and column qualifier plus its value. As soon as a MemStore becomes full, its content is written on disk, persisting the data. The file just generated is called HFile. Each region is then made of several HFiles. If an unexpected crash happens before the MemStore was full, all the data can still be recovered through the WAL. While each RegionServer has one MemStore for each table and column families, it has only one WAL for the entire database. Considering the structure of MemStore and HFiles, it is clear why inserting and update operations are considered the same. Everything consists of a new insertion in the database, simply with a more recent timestamp. By default, only the latest 3 timestamps are considered, but it can be tuned.

1.2.6 Read Path

Whenever a client needs to retrieve data from HBase, it has to specify a table name, a rowKeys range and optionally a column family and a column qualifier. Once it retrieves the correct region where the data is stored, it has to riconciliate information between MemStore and HFiles. In order to avoid frequent HFile accesses, each RegionServer maintains a cache, the so called BlockCache, storing the most requested records. It works closely to the MemStore and again, it can only store data belonging to a specific table and column family. The Block is the basic measure for fetching data from HBase and usually, it has a size of 64kB. The BlockCache is a set of Blocks. Each HFile is then composed of several Blocks with an index in order to reach all of them. The reading operation is composed of a sequence of three operations:

1. The MemStore is checked for any pending modification
2. BlockCache is checked, there should be some blocks with the rowKeys range specified
3. Valuable HFiles are checked

Whenever one step succeeds, the remaining ones are not executed.

With this articulated structure, HBase is trying to minimize the HFiles access. Moreover, in a traditional RDBMS, like PostgreSQL, the default block size is 8kB, which results to be 8 times smaller than the one in HBase. This remarks the idea of using HBase to process large amount of data, asking prevalently for "close" information. It results in a really strong application-dependent system, where frequent access to subsequent data plays a fundamental role.

In order to read some values from the CLI, HBase gives to the user two possible commands, SCAN and GET. The first one works as explained in the current section, a table name, a rowKeys range and eventually a column family and a column qualifier are given. However, it can also work without any rowKeys range specified, in this case, a whole scan of the table is performed. The GET command, instead, needs a unique rowKey to be specified. It can be thought as a SCAN operation with the rowKeys range of length 1.

1.2.7 Compactions

While read and write operations have been described in details, we did not talk about delete operations. Similarly to previous actions, deletion works on a specific table and column. In order to understand how this operation works, it is important to note that HFiles are immutable on disk. This implies that deleting a value means adding a new default value for that column, comparable to NULL. This marker is called "tumbstone". However, there is a process during HBase execution which permanently removes the deleted values. It is called Compaction and it comes in two versions:

- Minor Compaction. It is executed frequently and merges several HFiles in a larger compacted file. This operation is needed to keep data as less fragmented as possible inside the disk. It is designed not to affect performances of the database, so, a max number of HFiles considered can be set by the user.
- Major Compaction. It is scheduled rarely by the HMaster, because this process affects HBase performances. However, it is the only way to permanently remove deleted columns. Indeed, during minor compactions, the system cannot remove values, because the actual column and the marked column (tumbstone) can be located in different HFiles. In this scenario, the only possible way to delete a value is to aggregate all HFiles of a specific region. With this consideration, the system is certain to find all possibile values associated to that column.

By means of compaction operations, HBase is able to aggregate data and remove useless information. However, this comes with a great cost, a huge I/O and networking (due to HDFS) operation. This description clearly shows how delete operations are so expensive in HBase. Adding a tombstone to a column in the database is achieved by means of the DELETE command in the HBase CLI.

1.3 HBase and MapReduce

In previous sections, the detailed architecture of HBase has been described, giving an understanding of how it reaches scalability, consistency and random read/write access. Moreover, it has been explained briefly the Hadoop framework, consisting of MapReduce computation and HDFS. It has already been explained how HDFS and HBase are strictly related. In this section we are going to cover possible interactions between HBase and MapReduce jobs.

In the current context, there could be three possible scenarios, in which MapReduce jobs are executed on top of HBase. HBase can either behave as a data source, a data sink or a shared resource. While the first two implementations are pretty straightforward, the third one has interesting implications.

Since explaining implementation details of these techniques goes beyond the scope of this project, we limit the discussion to qualitative solutions.

Generally, MapReduce jobs consume data residing in the HDFS. In the view of HBase as a data source to MapReduce jobs, the user can define a table as the input of such jobs. The scan operation performed by HBase can be parallelized by breaking into pieces the rowKeys range of the requested tuples. In this way, several scan operations are performed on different processing units, looking concurrently for data. A map task is assigned to every region of the table, allowing parallel search. This is ideal when the user needs to access a large amount of data stored on HBase. In the view of Hbase as a data sink, similarly to the previous solution, data are reduced in the database. The system will schedule several reduce jobs which have the goal to write data on a specific table. Again, these jobs are scheduled on different region, in order to guarantee concurrency. These two implementations come useful whenever the user needs batch processing over HBase. This brief description gives a vague idea of how reading and writing can be parallelized and optimized. However, everything comes with a price. Executing MapReduce jobs over the whole cluster results to be really expensive. All the map and reduce jobs are parallelized over the cluster, using a lot of resources. It is advisable to exploit these operations only when a low-latency response is not requested from HBase. If the database is highly used for transactional operation, an execution of a MapReduce job will definitely decrease its performances.

In order to describe the third use of Hbase in the MapReduce framework, we need to introduce the concept of join. Joining two tables is a common task in the relational world. By having data in one single physical machine, indexed in a smart way, it is easier to execute joins. However, through MapReduce jobs, joins can also be performed on the HDFS. In this scenario, there could be two types of join: reduce-side joins and map-side joins. We are going to assume that HBase is the data source of both operations. Starting from reduce-side joins, during the map task, data from both the tables is mapped in key-value pairs, where the key is the joining key. In order to describe this scenario, it is needed to add more information about the MapReduce algorithm. Before the reduce task takes place, Hadoop executes two operations automatically, shuffle and sorting of data. Given the output of the map task, a set of key-value pairs, Hadoop needs to organize them in order to be consumed by the reduce task. This results in shuffling and sorting data. Through this operation all key-value pairs are sorted according the key and grouped by its value. Then, the reduce task is a method to aggregate all the values of a certain key. In the context of reduce-side joins, data is merged during this operation. However, this comes with a big problem. Hadoop shuffles and sort all possible data the user wants to merge. This operation is done with a big network and I/O cost, which happens to be the bottleneck of every distributed system. For large datasets, this operation may be too expensive. This is where map-side joins come into play. Map-side joins need to satisfy some constraints in order to be executed efficiently. In particular, one of the two tables needs to have random read access. If one of the two tables can fit in memory, then, the user can put the data in a proper structure (for example a hash table) in order to have random accesses. During the map operation, the biggest table is scanned concurrently and all the values are joined with the values contained in memory. In this case, the shuffle and the sort steps can be skipped entirely, because the join has already been executed in the map job. However, if both the tables are too big to fit in memory, the algorithm does not work. HBase saves this situation thanks to its feature of random read/write access. Given the structure already explained, the user can retrieve efficiently specific tables and values. In this case, one table is stored in HBase, while the other is scanned through the map job. For each value so obtained, a comparison is made with the values contained in HBase. The most important thing is that the rowKey of HBase table must be designed properly in order to let instant access to requested data. In this application, the rowKey must be related to the joining key. For each tuple in the second table, it must be easy to retrieve the key for the associated values in the first one. HBase can be seen as an external hash table, to which all servers have access.

Chapter 2

Logs Stores Application

In the last section we explained the characteristics of HBase, data model and more important its strengthens and weakness. We conclude that columnar databases are well-suited for OLAP-like workloads (e.g., data warehouses) which typically involve highly complex queries over all data and multiple aggregation. While row-oriented databases are well-suited for OLTP-like workloads which are more heavily loaded with interactive transactions. This is essential in order to choose an proper application for this project. Also, we need to address a real problem for which row-oriented databases are not suitable. Taking this into account the application is oriented to solve the problem of keeping track of thousands of logs of a software company. The first section of this chapter is an explanation of the problematic situation and next we address HBase implementation details.

2.1 Analysis of the Problem

Suppose that big software companies like IBM need to keep track of its software's logs for each of its clients (other companies). This logs could come in three different ways as *Error*, *Warning* and *Success* for several family of products (e.g. Database, BPM, Office, etc). The content of the logs is variable. Some of them could be predefined in advance but other can not. For example, an specific software could keep track of every state and have implemented a crash recovery algorithm. This software could give details in the logs about the step made to restore the system and the final output. This info should be store in a different place than the message sent by the current state. In any case the content of the log could be divided in two main group, *Human Readable Information* and *Programming Language Specific Information*.

The company is interesting in store this information and be able to retrieve it according to three different aspect: the kind of error, the product family and the company in a fast way. For example, they maybe be interesting in get all the logs of a company, or maybe all the logs for a specific family of software in a company and so on. Taking into account this and assuming that the company's departments take care of specific products family we could give an order of importance to this attributes. In this case then, getting information aggregated by product family is more important that getting it by company and at the same time company is more important than type of log. In addition, the company is sometimes interesting in getting all the errors of a specific type. For example, all the errors related with Networking functionalities. All the above is important to decide the schema of the database as we will see in the next section.

2.1.1 Building the schema

One way to model this situation with a traditional row-based database is to create a fact table with all the attributes as presented in the following table.

Company	SType	LogType	EType	Message	wasSolved	fPathSolve	newProb	time
Proximus	Database	Warning	Null	"Slow connection warning"	Null	Null	Null	17545
TaxHere	BPM	Error	InvFlow	"The specific Bussiness Process Flow is not correct"	Null	Null	Null	17550
ING	Office	Error	InvChar	"The character is not valid for current coding	Yes	Change coding for Asci	Null	17705
...	...	...	...	...	...	...	...	...

Table 2.1: Row-based solution of the problem using just one fact table.

The first problem we can notice is that the table is full of Null values. This is because different software do not produce the same information in the logs. For example, any *Warning* type log have values in the last three columns (*wasSolver, fPathSolve* and *newProb*). At the same time, the *Error* in the second row do not produce values either for those columns because maybe it does not have implemented a crash recovery algorithm. Finally, the last row bellows to a software for which a crash recovery algorithm was implemented and then we need to save the solution and the new problem if not solved.

Now suppose we what to see all the *Error* logs from the company *Proximus* we had in the last three weeks. A regular row-based database will need to load every column of the rows matching the condition even when they are fill with *Null* values. Also in order to prevent the database manager to do a full scan table we will need to add indexes in the three main attributes we mention before. This will bring some performance problem mainly for inserting and updating the database. One way to solve these problems could be with a different schema. Let us consider the following schema where instead of having just one table to store all the information we have one table for each of the software in the data warehouse. With this schema we do not need to store null values any more because we can create specific schemes for each of the software. Also if we need to get information about an specific program we just need to look at one table. However, what happen if we want to make a simple query aggregating like counting the number of error in a period of time for a product type. For doing this the Database will need to load every table and for each table count the number of error if the software is of the specified kind of error. Programming this query in SQL could be really challenging and the cost really high if we have been keeping track of the logs for years. So we can conclude that this solution is not also convenient for our problem.

On the other hand, a columnar database will solve most of the problems found above. Taking into account the data model of HBase and using its advantages. We just need to define a row key and the column families where the information will be store. Lets consider the following schema for solving this problem.

- **Row Key:** Defining a good key is the most important feature in HBase. In this case, our row key was defined as the sum[1] of the attributes *SType*, *Company*, *LogType* and *Time* respectively. This way we are assuring constant access to most of the queries since we already now the user is interesting in filtering and aggregations of this three attributes mainly. Also as we said before, one of the business rules is that the company is organized by departments of specific products type. So our preceding order in the row will allow us more efficient aggregations in the most frequent kind of queries.

- **Column families:** HBase store the information by column family. This way, related attributes can be retrieve at the same time. For this particular application, we set two column family: *Human Readable Information* and *Programming Language Specific Information*. Using this configuration, we are assuming that the user ask for this two kind of data individually.

- **Secondary index:** The system need to retrieve information also according to the type of the error. Then, we can not put this information in the row key because is not needed for each log. To solve this we created a new table with the log type as the row key and the row key of the main table as secondary index as the value of the cell. This way we could index in constant time by type of error. Notice that even when logically looks like the table is growing horizontally adding new columns for each of the logs, physically the table is growing vertically inserting a new row for each new column with a different timestamp.

[1]The sum is this contexts can be see as string concatenation.

In table 2.2 we show the schema for the main table in HBase. All not filled values indicate complete absence of the column in the row. This means for example that the first row is just kept in secondary memory with the row key and the message. Table 2.3 show the secondary index table for the error type. Note that each of the column for this table is the row key in the main table. This allow us to link each type error with the specific row it was produced.

Row Key	HumanReadInfo		ProgLangSpecifInfo		
	wasSolved	Message	EType	fPathSolve	detSol
ProximusDatabaseWarning21333434		"Slow connection warning"			
TaxHereBPMError232343413		"The specific Bussiness Process Flow is not correct"	InvFlow		
INGOfficeError12323453535	Yes	"The character is not valid for current string codin	InvChar	Change coding for Asci	

Table 2.2: Main table for the column-based solution of the problem.

Row Key	Logs row key	
	Log2	Log3
InvFlow12123334	TaxHereBPMError232343413	
InvChar12233445		INGOfficeError12323453535

Table 2.3: Secondary index on attribute error type.

Next section presents the implementation details in HBase and Java to provide the reader fully understanding of the project.

2.2 Implementation

In the previous chapter it became clear how the choice of the rowKey is important for an HBase application. Mainly, the format of the rowKey and the column families are the only designing choices. these two information can be compared to the schema of the table. Once they are defined, the application is free to load any kind of data that reflects these features.

2.2.1 Technology specification

The first problem our team came across was the installation of HBase. As explained before, it has a really complicated structure and in order to make it work properly, a particular set up of Hadoop must be made. In order to avoid this type of complexity, we decided to run HBase over a Docker version of Cloudera. Cloudera is a company that provides full integrated solution in the Apache Big Data stack. Among the software available in the Virtual Machine, there are Hadoop, HBase and Zookeeper. After properly running the Docker container associated to Cloudera, we were able to run HBase inside the container. HBase gives to the user the possibility to interact with itself through a JRuby shell. Thanks to this possibility, the user is able to run simple queries in order to test the state of the database.

Despite that ease of usage, we implemented a more sophisticated application in Java programming language. Indeed, Hbase exposes a Java client API. However, we needed to open a communication between the host machine and the Docker container. This problem has been solved through several ports forwarding from the host machine to the VM. Then, we set up the project through Maven, letting it download all the dependencies and finally, we started interacting with the database.

2.2.2 Implementation details

In order to simulate a possible situation with several software sending Logs to a central Database, we decided to split our program in several threads. In our scenario, each thread represents a different software simulation. For example, one thread could be associated to the Robot class, a particular product installed by the owner company in several other companies. The technical manger of this department could be interested in seeing all the logs related to this product family. In order to achieve this goal, the Robot thread writes a huge number of pseudo random logs to the database. In these logs, some mandatory information is present: Log Type, Product Family ID, Company ID, and Timestamp. These 4 information are the composing the rowKey. Each software sending logs to the database must specify them. In our simulation these information are randomly chosen among a set of possible values. Moreover, in order to query correctly the database afterwords, we need to assign a specific space to each attribute. Later, we will see how to manipulate queries through the rowKeys. The table 2.4 shows the byte size of each attribute present in the rowKey.

Row key	Byte size
Log Type	3
Product Family ID	4
Company ID	4
Timestamp	13

Table 2.4: Format of the Row Key.

However, even if HBase offers great flexibility in the choice of the rowKey and in the related schema of the table, here the first problem arises. The format of the rowKey is totally in the hands of the application programmer. This implies that if a software wants to write a rowKey in another format than the one specified, nothing will stop it. In principle, everyone with access to the database could put information with different schema, with no exception thrown by the database itself. HBase is only sorting this tuple with respect to the others. In a real scenario, the user must be careful not to split the software communicating with the database. Instead, he should have different software producing logs and a centralized server that receives all the requests, it formats them and loads them in the database.

After the rowKey has been defined and implemented, each log is made of a rowKey and several data. We represented this data through an HashMap, a key/value association among columns and values. Specifically, each key represents a column family and each value is again a key/value store for column qualifier and values. Through this structure we are able to map any possible information the software wants to transmit. Again, each thread simulates this log filling, choosing randomly among several possible data. This happens when, for example, two different software, both belonging to the family of robots, want to share different data, according to their execution environment. A possible output of this scenario could be the following:

```
ERR + ROBO + ORNG + 1544443519800
d -> Job ID: 7
d -> Main Path: /home/user/
h -> MESSAGE: Left arm is not responding
```

In the output above, it is shown the row key as first line, as a composition of attributes. Then, subsequent lines are columns of that row, specifying column family, column qualifier and value. Reading the example, there is an error log produced by a robot product family, installed in a company named Orange. This log is composed of two column families, *d* and *h*: respectively, *LogData* and *HumanReadableData*. For each of them, there can be several column qualifier, for example, *JobID* and *MainPath* are column qualifiers for the *d* column family, while *MESSAGE* is a column qualifier for *h* column family. Finally, for each of them, a value is associated.

2.2.3 Operations in HBase

In the previous section the implementation has been fully described. These few pages want to give an idea to the reader of how to communicate with the database. The first operation it is needed is the opening of a connection.

```
Configuration config = HBaseConfiguration.create();
//config.set("hbase.zookeeper.quorum", "192.168.0.102");
//config.set("hbase.zookeeper.property.clientport", "2181");
```

```
4 HBaseAdmin.checkHBaseAvailable(config);
```

Listing 2.1: Checking availability of HBase

In code 2.1 we are checking the connection to the HBase cluster. First thing to be noted is the class *Configuration*, which belongs to the $hadoop - common$ package. This is a configuration instance used to connect in general to an Hadoop cluster. Through the subsequent commented lines, it is possible to set some configuration parameters. If they were uncommented, the IP address and the port of Zookeeper server would be specified. In that scenario, we would execute the VM through a bridged network configuration. This means that the VM would act as a proper computer in the local network, having a proper IP address. With this configuration we would need to specify this address to the HBase configuration. Moreover, 2181 is the standard port for Zookeeper service. As explained in the transaction flow, the client application just needs the Zookeeper location and then it is able to reach all the region servers. Finally, line 4 checks the connection to the specified location. Basically, it checks that an active HMaster with at least one RegionServer are active in the given location. An exception is thrown if this condition is not met.

```
1  // Opening the connection
2  Connection conn = ConnectionFactory.createConnection(config);
3  // Creating the admin instance
4  Admin admin = conn.getAdmin();
5
6  // Creating the table
7  TableName tableName = TableName.valueOf(tableNameStr);
8  byte[] hr_cf = Constants.CF_HUMAN_READABLE;
9  byte[] d_cf = Constants.CF_DATA;
10 HTableDescriptor desc = new HTableDescriptor(tableName);
11 desc.addFamily(new HColumnDescriptor(hr_cf));
12 desc.addFamily(new HColumnDescriptor(d_cf));
13
14 // Committing
15 admin.createTable(desc);
```

Listing 2.2: Creation of the "Logs" table.

In 2.2 it is shown how to open a connection to HBase and to build a table. Through the first line, a new *Connection* object is created. This is achieved through the usage of the configuration created before. Next, a new *Admin* object is created. Through this variable it will be possible to execute DDL operations on top of the database, like table creation, enabling, disabling and deleting. As explained in previous section, the structure of an HBase table is extremely simple. Only the column families need to be specified a priori. The format of the *rowKey* is defined by the user at every insertion. From line 7 to line 12, this structure is defined. It is evident how this operation is simple. However, this comes with greater complexity at application level in order to manage the data types inserted in the database. Only the two column families are specified, together with the *TableName*. Subsequently, a new *HTableDescriptor* is instantiated, in order to store all this information. Finally, this object is committed to the database, sending the create request.

```
// Retrieving the table
Table table = conn.getTable(tableName);

// Creating the log
Log log = ...

// Inserting it into the database
Put put = new Put(log.getRowKey());
List<byte[][]> listCoord = log.getValues();
for (byte[][] coord : listCoord) {
    put.addColumn(coord[0], coord[1], coord[2]);
}
table.put(put);
```

Listing 2.3: Putting a log in the database

In 2.3 a new *Log* is created and inserted in the database. For simplicity reasons, the whole code will be uploaded to a git repository, while in these example we are only keeping the HBase related information. Through the *Connection* instance we are able to retrieve the effective table present in the database. Next, we create the *Log* to be inserted and we extract values from it. In order to append some values in an HBase table, the object *Put* is instantiated. As constructor parameter it requires the rowKey it needs to be inserted. It is clear now how all operations are strictly related to the rowKey value. Subsequently, all data is extracted from the log over a coordinate representation. Basically, it consists of a column family, a column qualifier and a value. Through the *addColumn* method, it is possible to append all this information to the *Put* object. Finally, the operation is committed to the table through the *put* method. It is clear how it is important to understand the internal structure of the database in order to communicate with it; all practical operations map the logical execution flow of the database.

```
Scan s = new Scan();
ResultScanner rs = table.getScanner(s);
for (Result r : rs) {
    // Extracting the rowKey
    String rowKey = new String(r.getRow());

    // Iterate over all Cells of the row
    Cell[] cells = r.rawCells();
    for (Cell c : cells) {
        String cf = new String(CellUtil.cloneFamily(c));
        String cq = new String(CellUtil.cloneQualifier(c));
        String value = new String(CellUtil.cloneValue(c));
    }
}
```

Listing 2.4: Scan the table

In 2.4 a read operation is performed on the *Logs* table. As briefly introduced in the read path section, the basic operation to achieve this goal is the *Scan* operation. The *Scan* object can be defined either on the whole table or on a specific rowKey range. In the example, a full scan is performed. The result is given as a list of rows, each of them defined by a rowKey. Moreover, from one row, it is possible to extract the list of columns, identified as Cells. Thanks to the static methods exposed by the *CellUtil* class, it is possible to extract all the info from the *Cell* object. It is important to remember that all the values stored in HBase are byte arrays. In order to read them, they need to be converted to *String*.

2.2.4 Querying data

In the previous sections we have analyzed the main building blocks of HBase. We understood how to perform a Put and a Scan operation. In order to validate the application we need to fulfill the requirements we introduced at the beginning. The user must be able to aggregate logs according to different attribute values. In the next examples, we are analyzing how to build a proper rowKey for the Scan operation, in order to retrieve the desired information.

The first goal of the application is to aggregate logs based on Log Type. This is the example where the technical manager of the company wants to know all error logs of its products. This can also be extended to Warning and Info logs. According to the structure of the rowKey, the first attribute that appears is the Log Type. Knowing that rowKeys are sorted alphabetically, the user only needs to provide the system with the first Error log and the last one. The problem is that the user only knows that he wants Error logs, he knows nothing about product families or companies. In some ways, the product families and the companies are the answer to his query, because in the end, he will know in which scenario the error has occurred. Considering the start rowKey we need to find a way to represent the first tuple starting with *ERR*. Since everything is represented as byte array in HBase, we know that the character with the smallest ASCII encoded value is the space (' '). If the rowKey has 23 bytes and the first 3 are occupied by the Log Type, it is certain that there will be no error logs before `'ERR                    '`. If the user scans the database with such a rowKey, which clearly is not present, HBase will return the first non-empty row following the one used as input. This solves half of the problem, because now the user can use such rowKey as start RowKey.

Considering the end rowKey, we apply the same reasoning. We need to find the upper bound of the search, looking for a rowKey slightly bigger that the one we are looking for. In the application scenario, the user is looking for Error logs. It is known that all logs he wants start with *ERR*, so he only needs to put as end rowKey *ERS*. This represents the same prefix of the start rowKey, incremented by 1. In this way, the user is sure to cover all possible logs with Error log type.

```java
RowKey startRK = new RowKey.Builder()
        .setLogType("ERR")
        .build();
RowKey endRK = new RowKey.Builder()
        .setLogType("ERS")
        .build();
List<Log> logs = Scanner.scan(hbase.getTable(), startRK, endRK);
```

Listing 2.5: Retrieving all error logs

In 2.5, it can be noted how the rowKey is built. Thanks to our software architecture, we are able to build the rowKey with any value and the remaining ones will be filled with the space character. Other two things need to be specified. Through the *Scanner.scan* static function the user can also specify the column families he wants to retrieve. For example, in the previous scenario, the user may be interested only in the messages of the error logs. Only after, he will conduct a deeper analysis for a specific log. The modifcation would be the following:

```java
List<byte[]> columnFamilies = new ArrayList<>(1);
columnFamilies.add(Constants.CF_HUMAN_READABLE);
List<Log> logs = Scanner.scan(hbase.getTable(), columnFamilies, startRK,
    endRK);
```

Listing 2.6: Retrieving only messages of error logs

Second thing to be noted is the order in which they are displayed. It is already well known that the order depends first by the Log Type, then by the Product Family, then by the Company ID and finally, by the Timestamp. The Timestamp has been inserted in order to distinguish between two logs coming from the same product family, the same company and with the same type. However, since the ordering in HBase is ascending, all the logs will be sorted in the opposite logical way. It could happen that the user requests only for the most recent log of a certain combination of the previous attributes. In that case, he would specify that combination as the start rowKey and then he would limit the search to 1 value. However, if the keys are sorted in ascending order, he would retrieve the oldest log, not the most recent. In order to solve this problem, the application saves the timestamps in the rowKey as reverted timestamp. In order to do so, the real timestamp is subtracted from the maximum timestamp allowed and it is written to the rowKey. The maximum timestamp allowed is limited by the size allocated for the timestamp attribute in the rowKey.

```java
revertedTimestamp = String.valueOf(getMaxTSValue() - Long.parseLong(timestamp));
```

Thanks to this artifact, the system is able to store the logs in reverse order and the user is now able to get the most recent log. Moreover, in order to retrieve back the real timestamp of a specific log, it is just needed to apply again the previous formula to the reverted timestamp.

Proceeding forward with the requests of the application, the user may also want to retrieve all the logs for a specific product family. In this scenario, the user may be a technical engineer working on a specific product of the company, for example the DBMS. In this case, he is not interested in the logs produced by the Drones or by the Cloud-based systems. However, it is not possible to perform such a query with the maximum performance of the database. In order to understand the last sentence it could be useful to think of the rowKey as the key of a BTree index. When the BTree is used for multidimensional indexing, it guarantees only certain ordering sequences. For example, if the BTree is built on attributes A,B,C of a certain relation, only these sequences are guaranteed to be sorted:

```
A,B,C
A,B
A
```

In our example, A can be associated to the Log Type and B to the Product Family. In the scenario we proposed, the user wants to retrieve all the logs for a specific product family (B). In order to perform efficiently such a query, we would need to have the data sorted on that attribute. However, the structure we defined before does not allow for this particular sorting. This problem has been faced in the design of the rowKey, where we tried to give an importance to each attribute. However, that is not enough. Even if the sorting is not guaranteed on Product Family, we may solve the problem specifying three different types of query. In the first one, we look for the Error logs of that specific product family, knowing that this combination will have sorted data and therefore, the operation will be efficient. In the second one we repeat the process with the Warning log type and in the third, we use Info Log Type. Proceeding with this method, we will execute three queries instead of one in order to retrieve the desired information.

```java
String[] logTypes = {"INF", "WAR", "ERR"};
for (String logType : logTypes) {
    startRK = new RowKey.Builder()
            .setLogType(logType)
            .setProductFamily("DBMS")
            .build();
    endRK = new RowKey.Builder()
            .setLogType(logType)
            .setProductFamily("DBMT")
            .build();
    logs = Scanner.scan(hbase.getTable(), startRK, endRK);
    Scanner.print(logs);
}
```

Listing 2.7: Retrieving all logs for DBMS product family.

While this method allows us to maintain a certain level of performance, it requires that the user knows a priori the list of all possible values for the preceding attributes. For example, if the user wants to know all the logs related to a specific company, he needs to know all possible log types and all possible product families installed in such a company. We believe that such information is available for this kind of application, but maybe in other scenario, this would compromise the effectiveness of HBase. However, while the information of previous values can be retrieved, performances need to be taken into account. We said that for 2.7 the system will execute 3 queries instead of 1. The explanation of the HBase architecture will make clear the following performance analysis. According to the indexing structure of Hbase, the client has to perform three request every time he wants to access to some information. First is toward the *.ROOT.* table, then to the *_META_* table and then to the region itself. Following these three steps the client is able to get or to insert some data. We can conclude that, independently of the size of the database, the user will always perform 3 network I/Os (*.ROOT.* table never splits). In our analysis we are considering the network I/Os as the main factor influencing performances. Considering again 2.7, if the client requests for 3 scan operations instead of 1, this implies that the operation executes 9 network I/Os in the worst case (no caching applied). Considering this growing factor, if we put as first parameter in the rowKey one attribute with a high cardinality, in order to retrieve a specific value following the first one we need to perform a big number of scan request. This can be generalized as: Given a sequence of attributes in the rowKey, in order to retrieve a specific value for one of those, the client will perform a number of scan requests which is equal to the cartesian product of all the previous attribute values. In other words, in our example, if we keep the Log Type as first attribute, in order to retrieve information related to a product family, we need to perform 3 scans, which is the cardinality of Log Type. Instead, if the user wants to look for a specific company, the number of requests is equal to all the possible combinations of Log Type and Product Family values. We can define this combination as the cartesian product of the two attributes.

Concluding this section, we can say that HBase provides a really efficient way to query data. However, it is limited to the possible queries that can be executed. Even though the user can ask for any particular information inside the database, in order to retrieve the data quickly, some artifacts must be designed. It is not always possible in a real application to abandon query flexibility. However, for large size databases, that can accept this tradeoff, HBase presents itself as an optimal solution.

Conclusion

This report presented a general introduction to column-oriented databases. We discovered that they physically store tables according to their columns. Through this architecture, considering only a subset of attributes results to be more performing than in a row-oriented architecture. However, when the the user wants to reconstruct an entire row, this is certainly more expensive. Moreover, through a columnar disposition of data, a more efficient compression can be applied.

Subsequently, we analyzed the structure of HBase, understanding why it is a random Read/Write access database and which are its limits. One of the most evident is the complexity needed for secondary indexes implementation. Since HBase does not support them, the user needs to implement them through some custom schemas. We also learned how to exploit the rowKey as efficiently as possible, being the only indexing criterion available in the database.

In the last chapter we discussed about a possible application on top of HBase. We chose a Logs store in order to exploit the schema-less feature of the database and the possibility to divide the information in Human Readable and Program Data. Moreover, the application allows a user to efficiently query the database for a certain set of queries. According to our reasoning, they will be the most frequent ways of interaction with the system. It can be noted how the database limits the complexity allowed by the application and how its schema needs to be defined according to the final queries the user needs to execute.

According to its structure we can conclude that HBase and, more generally, column-oriented databases are well suited for large amount of data, which needs to be queried often and on some specific attributes. Moreover, HBase is an optimal solution for frequent updates, since, internally, they are the same as insertions. Scalabilty and efficiency come with a complexity reduction cost. Moreover, the absence of a SQL-like language makes the management of applications more cumbersome.

Bibliography

[1] Nicholas Dimiduk and Amandeep Khurana. *HBase in Action*. Manning Publications, 2012.

[2] Lars George. *HBase: The definite guide*. O'Reilly Media, 2011.