In pioneer days they used oxen for heavy pulling, and when one ox could not budge a log, they did not try to grow a larger ox. We should not be trying for bigger computers, but for more systems of computers.

– Grace Hopper

University of Alberta

ENHANCING QUERY SUPPORT IN HBASE VIA AN EXTENDED COPROCESSOR FRAMEWORK

by

Himanshu Vashishtha

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

To Ayn Rand
(for opening my eyes, at the most appropriate time)

And, To Krishna, my adorable nephew
(The apple of my eye)

Abstract

Data is growing at an unforeseen rate, with TBs being generated daily. A large part of this data is unstructured in nature. This has pushed the traditional techniques of storing it in relational databases to its limit and new alternatives are necessary. Cloud databases have emerged as a viable candidate and have been gaining popularity due to their high scalability and availability. However, as yet, they lag behind RDBM systems in terms of the support to developers for querying the data. The problem of developing frameworks to support flexibe data queries is a very active area of research. In this work we consider HBase, a popular cloud database, inspired by Google's *BigTable* data structure. Relying on the recent *Coprocessor* feature of HBase, we have developed a framework that developers can use to implement aggregate functions like row count, max, min, etc. We further extended the existing Coprocessor framework to support *Cursor* functionality so that a client can incrementally consume the Coprocessor generated result. We demonstrate the effectiveness of our extension by comparatively evaluating it against the original Coprocessor framework with four queries on three different data sets. We also share our experience while migrating an existing text analysis application, TAPoR, to HBase.

Acknowledgements

As I write this page, I can recollect my journey when I started looking for my Thesis topic in 2010 summer. I was meandering around HBase but was not sure about the exact topic to work on. I got a sniff from HBase community to look the newly developed Coprocessors functionality. It was a long shot for me as I was new to HBase and Coprocessors as such, and it was not even finalized whether it will be added to HBase or not.

I was really lucky to have Dr. Eleni Stroulia as my supervisor as she showed more confidence than I about its potential. I give full credit to her for stretching herself beyond her core area and allowing me to work on the chosen thesis topic. I thank Gary Helmling and Michael Stack of Apache HBase team for their constant guidance and motivation for my work.

During this long run, whenever I felt stressed or emotionally down, it was my lovely sister, Vijeta, and my friend and brother in law, Nav, who were always there to help me out. Without their emotional and moral support, this work wouldn't have been possible. I am really grateful for their unconditional help. Lastly and most importantly, I would like to thank my parents for their constant encouragement and good wishes.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

We are witnessing the unprecedented generation of substantial amounts of digitized data. Books are being digitized on a massive scale [1]. Social-networking platforms like twitter, Facebook, YouTube, Flickr, forums, and blogs collect data at mind-boggling rates. Pingdom [2], ordering social networks and countries according to their population, places Facebook slightly after Brazil and QZone just ahead of it. Twitter alone produces 15-20 terabytes uncompressed data daily. Sensors embedded in most devices, from refrigerators to cars to airplanes, continuously record a multitude of variables. Anecdotally, a Boeing jet generates 10 terabytes data per engine every 30 minutes of flight time [4].

This new massive scale of data to be collected, archived and processed makes the traditional model of computing, where an organization invests capital costs in purchasing its own infrastructure and operation costs in maintaining it, impractical, unaffordable and eventually obsolete. Instead, infrastructure is increasingly provided as a service (IaaS) with specialized providers offering large-scale computational resources (computing power, storage and network bandwidth) at economical rates. In this new model of computing infrastructure, one can lease storage and virtual machines, configured in a manner that meets the computing task at hand. So affordable are the prices of these offerings that more than a few corporations don't own any infrastructure at all; they store "everything" *on the cloud* [5, 6].

The availability of massive and scalable infrastructure brings to the forefront two interesting challenges. The first challenge is to develop support for storing data in a scalable and fault-tolerant manner, assuming infrastructure failures as a norm rather than as an aberrant behavior. The second challenge involves the development of a software layer that will enable software developers to implement scalable analyses methods, be able to process big

[1] `http://books.google.com/googlebooks/library.html`

[2] `http://royal.pingdom.com/2009/03/13/battle-of-the-sizes-social-network-users-vs-country-populations/`

data with high throughput, in order to extract useful, actionable knowledge from it. These two challenges push the idea of virtualization from the *infrastructure* level to the *platform* level, where the software-engineering challenge is to provide layers of software support to make the manipulation of data on the cloud as easy conceptually as it is on the traditional infrastructures to which most developers are accustomed.

The Apache Hadoop project [7] represents one such platform. With the underlying Hadoop Distributed File System (HDFS), it provides a distributed, scalable and fault-tolerant file system for storing big data. It is designed for storing and accessing large files (with an assumed upper limit to its size in terabytes). Relying on HDFS, Hadoop supports the MapReduce[2] programming model, where computation is conceived as a combination of a *map* and a *reduce* phase. In the map phase, the data is processed to generate intermediate results, stored locally on each of the nodes that have run the map phase. In the reduce phase, these intermediary results are collected at the *reducing* node and processed further to form a collective output. The map phase supports data-parallel computation as it can be run concurrently across multiple nodes on a different sets of data. The MapReduce computation is reminiscent of batch processing, where the whole data set needs to be processed in some manner – a typical job in MapReduce runs with in the range of a few minutes to few hours or more – but it is a less than ideal candidate for on-line query type of workloads, where small subsets of data need to be examined and manipulated.

Google developed BigTable, a scalable data structure that provides fast random read performance and support incremental data updates. It is being used in Google for more than 90 of its products such as Google Map, Google App Engine, Gmail, etc. Its data storage pattern can be visualized as distributed, sparse, column-oriented, multi-dimensional table, sorted on a primary key [8]. Apache HBase is an open source Java implementation of BigTable and is widely used in a number of corporations [5]. It has become an active field of research, focused on the optimization and extension of its current feature set. HBase provides APIs for storing and accessing big data, as if it is stored in a table. Developers can develop client-side software to invoke server-side "filters" to sieve through the data, which can then be returned to the client for further processing. HBase stores its data in a data structure similar to the BigTable's SSTAble, called *HTable*. In a typical HBase workflow, data is first accessed at the individual nodes, i.e., the *Region* servers where HTable *Regions* are stored, and is then sent to the client for further processing. A *Region* is a subset of a table.

A canonical example for a database operation is to do a row count of an entire table, or its subset. In HBase, there is a *Scan* API which supports transferring HTable rows to the client, where the actual row count will be computed. Note that, if multiple *Regions* are involved, all their corresponding rows will be transferred to the client sequentially, one *Region* following another. This does not scale well and becomes quite slow, as the number of *Regions* is increased. A better approach would be to parallelize it at the granularity of *Regions*. As mentioned above, *Scan* API of the original HBase is sequential, but one could also run a MapReduce job on the table, where in the map phase, all *Regions* scan their rows and emit out a key with a unary value, and in the reduction phase, all these unary values are summed up giving the entire table row count. This approach provides better throughput, as it is done in parallel on all nodes, and is effective for a large table but still suffers from two major limitations:

- it processes the entire data set (the complete HTable) even if the end user was interested in a small subset of selected rows (this is an inherent design in Hadoop on HBase); and
- because of the above limitation, its latency cost is high, which makes it an undesirable candidate for on-line queries.

The new *Coprocessors* feature of HBase is conceived to address exactly this challenge: developers can now develop software for computing local results server-side, i.e., at each node where HTable *Regions* are stored. These results can then be aggregated (relying on special-purpose *Coprocessor* libraries) at the client-side.

In this thesis, we have developed an extension to the HBase *Coprocessor* framework to enhance its query processing capability. Our work supports "streaming" of the partial results that are generated by the *Coprocessors* , by providing a cursor-like feature at the server, so that the client may get incremental results from the HBase *Regions*. [3] With this support, one can use a *Coprocessor* instance on a *Region* server in a *stateful* way, i.e., can store the client state such as a pointer to the *Region* row read in the last call; and therefore use it across multiple Remote Procedure Calls where each one sending intermediate results to the client. The existing *Coprocessor* framework supports only *stateless* calls, one should access a *Region* in one Remote Procedure Call only and receive the complete result set all at once.

[3]Part of this work has been accepted and committed to the Apache HBase project and the rest is under review.

1.1 Contributions

The main contributions of this work are as follows.

1. We have designed and implemented six standard aggregate functions, viz., sum, average, minimum, maximum, standard deviation and row count, using the *Coprocessors* framework. The design is generic enough to support any kind of data. This piece of work is part of Apache HBase now. This is explained in more detail in Section 3.1.2.

2. We have designed and implemented a *results-streaming* functionality, extending the HBase *Coprocessors* endpoints. The current *Coprocessor* framework supports only stateless RPC to a table *Region*. Its limitation is that a *Region* has to be exhausted in its entirety before returning the RPC result, because there is no support for the server to maintain information about the client state. Some use cases need support for a streaming functionality, like the NGram example we discuss in section 3.2.1. This streaming functionality substantially improves the usability of the *Coprocessor* framework.

3. The above mentioned results-streaming functionality can be used to support a parallel *Scanner* functionality, which augments the current *Scanner* API with the parallel *Coprocessor* feature.

4. We have migrated an existing text analysis application, TAPoR, to HBase. We share our experience of important considerations like HBase schema design and development of REST web services to interact with HBase. We believe that the lessons we learned through our experience are relevant to any developer wishing to migrate similar applications to HBase.

1.2 Thesis organization

This thesis is organized as follows. Chapter 2 gives background information about Hadoop and two of its eco system projects, namely HDFS and HBase, which are core components of this work. We also shed light on the relevance of our work, acceptance of Hadoop as a de-facto tool for data parallel applications. We conclude it by mentioning related work in HBase, and about possible *Coprocessors* use cases. Chapter 3 covers our extension, viz., HBase aggregate functions on *Coprocessors* , Streaming results API, experiments and their discussion. It also discusses the limitation of current aggregate functions in the case of large table size, and a possible solution by using the streaming results. We discuss the TAPoR migration to HBase in Chapter 4. We explain application selection criteria, schema design process and overall application architecture. Finally, we conclude with a summary of our work, a review of our contributions and our plans for future work in Chapter 5.

Chapter 2

Background and Related Research

The focus of this thesis is HBase, an open-source NoSQL database developed under the Apache Hadoop aegis. NoSQL (aka Not only SQL) databases distinguish themselves from the relational databases as they are un-relational, column oriented databases, that can be distributed over 100s of machines, and that provide their own APIs to interact with the persisted data. This impressive scalability comes at the cost that most of them come with no support for *joins*, SQL-like standard query languages, and datatypes (with byte arrays as the only data type). Even with these limitations, NoSQL databases are being increasingly adopted for workloads involving large amount of data (up to the petabytes scale).

In this chapter, we first provide some background knowledge about HBase, including a review of the architecture principles of Hadoop, HDFS, and HBase. In the HBase section, we give an overview of the *Coprocessors* framework, the newly added feature that allows clients to deploy user defined arbitrary code at server side. Then we discuss how this work relates to the family of these tools. Finally, we review related research.

2.1 Hadoop, HDFS and HBase

This thesis belongs to the Hadoop ecosystem, which is a open source implementation of the MapReduce paradigm [2]. Overall, we used three Hadoop projects, viz., Hadoop-core, Hadoop Distributed File System (HDFS), and HBase. The later is a popular NoSQL database and forms the main area of concentration, where we extended the *Coprocessors* framework for more interactive query processing. In this section, we define the basic architecture of Hadoop, HDFS and HBase, in same order. In the HBase part, we also explain the *Coprocessors* framework, which forms the crux of this work.

2.1.1 Hadoop

Starting with the now seminal MapReduce paper[2], there has been a tremendous growth in data analytics using this paradigm. MapReduce is a scalable and easy way of processing large amounts of data (up to petabyte scale) to be processed in parallel on a cluster of commodity machines. It is not a silver bullet for all kinds of big-data problems, rather it is specific to workloads that are data-parallel in nature. It proposes the design of applications in terms of *map* and *reduce* functions, where the map process consumes raw data as key-value pairs and generate intermediary key-value pairs, which are collected by the reducer process. The processing of the map phase relies on the data-locality concept, with computation performed at the data nodes (nodes where data is residing). The MapReduce paradigm has, for a long time, been part of functional languages in general; however, its usage in data-parallel big-data applications, where individual map tasks can be completed independently, has been a key factor in its popularity. It is being used in many large production systems, where multiple terabytes of data are processed on a daily basis.

The main advantage of this model is the inherent support for execution in a distributed environment, where the developer does not need to take care of inter-process communication among the nodes. This helps in using this framework on a cluster of commodity machines instead of requiring high end shared-memory machines.

2.1.1.1 MapReduce architecture

Figure 2.1 depicts the dynamic behavior of a typical MapReduce job.

1. The user library forks a set of separate processes in a cluster. Specifically, the cluster is configured with two kind of machines: a master, which coordinates the MapReduce job execution in the entire cluster, and the worker node that actually execute the computation. This is shown as step (1) in Figure 2.1. These worker nodes are a part of a distributed file system, HDFS (which will be discussed next) and on them reside the data to be processed. It is interesting to mention here that, as all one-master-many-slaves distributed-computing models, the master in the MapReduce model is a single point of failure.

2. In step 2, the master node assigns the map tasks to available nodes. It keeps track of which node is processing what chunk of the data; thus, in a case of a worker node failure, it reassigns the processing of the same data chunk to some other node.

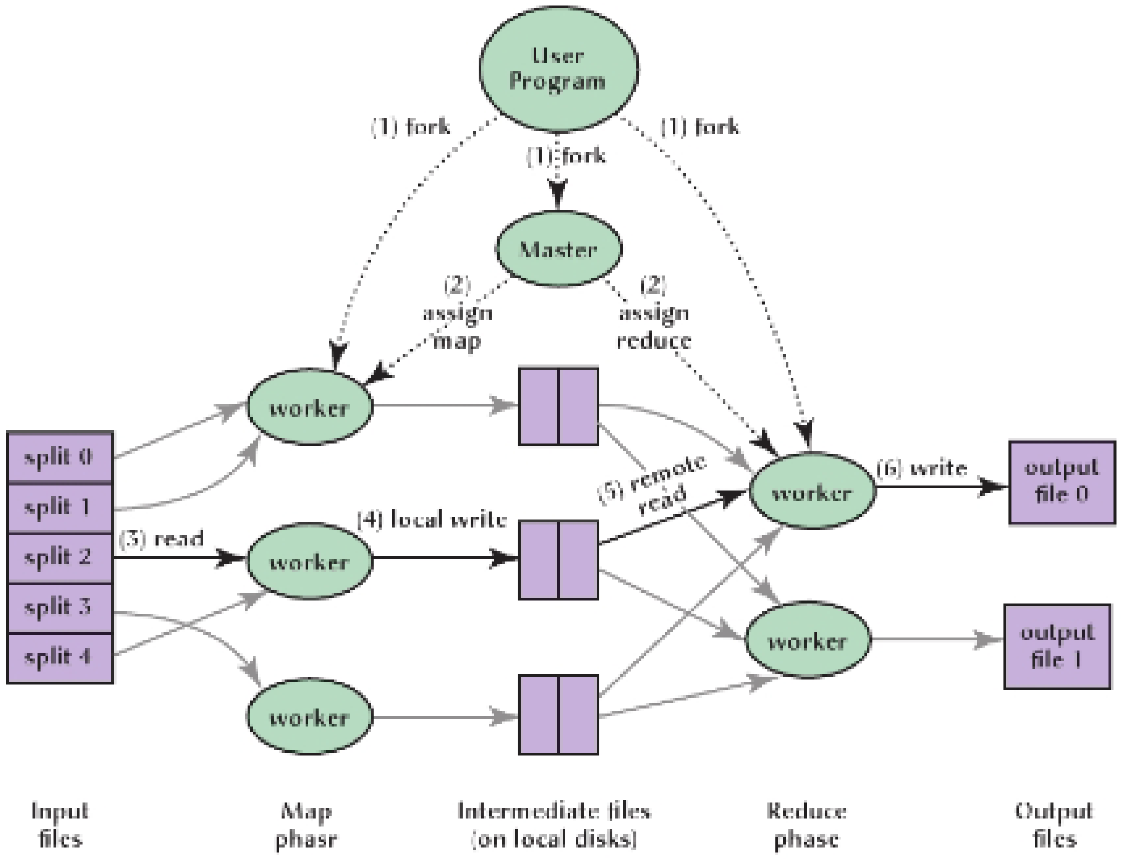


Figure 2.1: Workflow of a typical MapReduce application [2]

3. Step 3 shows the actual map phase, in which worker nodes read raw data from the file system as key-value pairs and generate intermediate key-value pairs. The map function, as defined by the user, is executed in this phase. After completing the map phase, these worker nodes send a signal to master node to inform it of their availability for further tasks.

4. Step 4 is optional. It involves the writing of the intermediate keys to the local native file system. It happens only when the size of this intermediate result is too large for the memory of the worker node.

5. Step 5 is the reduce phase. Here, the master node is aware of some intermediate results of the map phase, so it selects available nodes to do the reduce phase. The reduce phase involves reading of the intermediate key values, so it may involve remote procedure calls (RPCs) to transfer the data to the reduce nodes. The data is transferred such as all the values related to a key are transferred to a single reduce

node. This is an important feature of the MapReduce paradigm.

6. Finally, in step 6, the reduce nodes generate the output of the reduce function. Each reduce node produces its own output.

With the emerging cloud-computing environments, one can easily leverage the potential of the MapReduce paradigm. Amazon, the leading cloud-computing environment provider, has started various services like Elastic MapReduce, suited specifically to MapReduce type of applications. In this environment, software developers can rather easily program distributed software on a virtual cluster, by implementing the two functions. Platform libraries take care of the rest, including RPCs, distributing the workload among nodes, using data locality, handling node failures, etc. Equally importantly, one can add or remove nodes from the cluster simply by editing configuration files, and, again, the migration to make use of the new configuration is taken care by the platform libraries.

One important consideration for MapReduce is that it provides a high throughput when run in a cluster environment. Though there has been some work of using it in multi-core shared-memory machines, its real advantage is using it for a distributing-computing application, on a cluster of machines. Having said that, its core features, namely handling node failures and rerunning of failed tasks, work on the assumption of using a distributed file system. We discuss the essential features of such file systems in the next section.

The Apache Hadoop project offers an open-source Java implementation of MapReduce. We used it for our work, and it is necessary to describe some terms specific to Hadoop. Its overall architecture is as we discussed above, with some minor differences in nomenclature: the master node is called JobTracker and the worker nodes are called TaskTrackers.

2.1.2 HDFS

MapReduce jobs impose a different set of requirements for reading/storing files as compared to traditional native file systems (POSIX). Below are the main requirements for such a file system.

- A distributed file system should run on inexpensive commodity machines, and it should continuously monitor itself and recover from inconsistencies.

- It should be optimized to store large files, with file sizes in the multiple of gigabytes being the common case.

- It should primarily support two kind of "read" workloads: large streaming reads (up to few MBs) and small random reads (up to few KBs).
- It should support large sequential writes; updates are relatively rare, primarily file appends.
- It should support concurrent updates and reads, as there may be more than one application using the file system.

Google designed and implemented the Google File System (GFS) [9], in order to provide the above functionalities. It is based on single master and multiple worker architecture (similar to MapReduce). The workflow in the file system is such that client interaction with master is kept as less as possible. This is done to prevent the master from becoming a bottleneck of the system. Files are divided in blocks (of configurable size), and the Master node contains the meta data about mapping files to GFS blocks, and blocks to worker nodes addresses (these worker nodes are referred to as chunk servers). A client interacts with the master only when it has to look for a chunk location (whether reading an existing or, writing to a new chunk); it caches that location thereafter and connects directly to the chunk server without any Master lookup. Thus, the actual user data never flows through master; the Distributed File System (DFS) clients interact directly with the chunk servers. Fault tolerance is supported by replicating the blocks on multiple nodes, considering intra-rack and inter-rack topology while deciding these nodes. For example, in case the desired replication factor is 3, the second copy is made on the same rack, and the third copy is made on a different rack. This ensures data availability even in case of a rack failure. Though the tradeoff it entails is a longer write operation as copies are made to different racks and the client is not given a write acknowledgement until all the copies are made.

The Apache Hadoop Distribute File System (HDFS) is an open source Java implementation of the GFS. Here, the Master node is named Namenode, and the chunk servers are called Datanodes. As we have used Hadoop and its subprojects for this work, it will be convenient for the reader if we follow only Hadoop terminology in this thesis. We have seen two set of processes so far, MapReduce and HDFS. The MapReduce master and slave are called JobTracker and TaskTracker, and the HDFS master and slave nodes are called NameNode and Datanodes respectively.

Figure 2.2 shows the architecture of HDFS. The Namenode stores metadata of the file system in an in-memory data structure. The metadata has mappings for file-name to block Ids, and block IDs to locations. The Datanodes are worker nodes which store data as blocks.

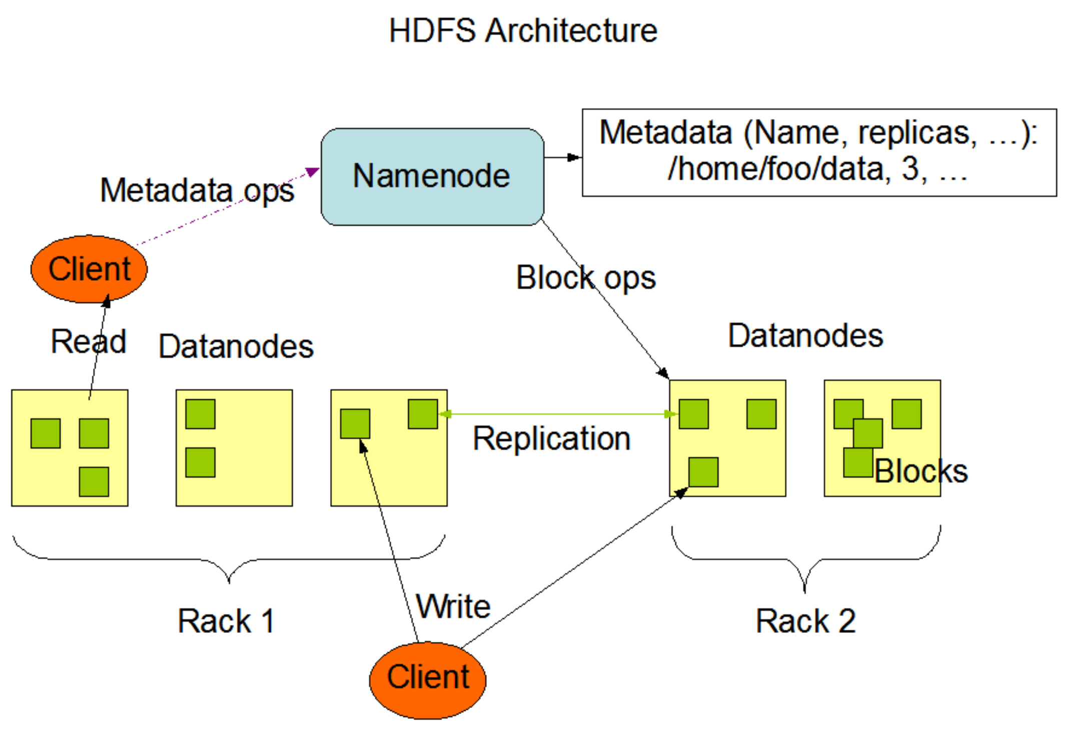


Figure 2.2: HDFS architecture. [3]

These Datanodes periodically send heartbeat signals to the Namenode about their health (liveness). The Datanode's meta information, like available disk space, disk usage ratio, data blocks information is also piggy backed on these signals. This way, the Namenode is aware of entire filesystem state and monitors its health. In case a Datanode goes down or has bad disk, the corresponding blocks will become under replicated and the Namenode assigns additional Datanodes to copy these blocks. In this workflow, one Datanode acts as the DFS client and copies in the under replicated block from other available source.

A HDFS Client first interacts with the Namenode to get the location of file system blocks. Once it knows the Datanodes location, it makes a direct TCP connection to them and read/write the data. It keeps this meta information of the blocks until the connection is reset. This is done to avoid making Namenode a bottleneck of the system. Going back to the requirement of streaming data for the map phase in a Hadoop job, the mapper node initiates a TCP connection to the data node and reads in one block at a time. This is an efficient operation given the block size is 64MB or more.

There are some workloads where random reads are required. In HDFS, this entails seeking to the byte offset and reading the file, which can be a costly operation given the large block size. With sorted data, Hadoop provides a special format, MapFile. It comprises of two separate files: a data file that contains the actual data, and an index file that has byte offset of values after a configurable gap. For example, a file with 1k records will have 10 entries in the index file if gap is defined as 100. When reading, a *binary search* on the index file is performed and the correct byte offset is located. In the latest Hadoop version, MapFile has been deprecated and replaced by TFile. Though discussion of TFile is beyond the scope of this work, it is worth mentioning that TFile is the prime motivation of HBase's HFile. We will look at it in detail in section 2.1.3.2.

2.1.3 HBase

The results of MapReduce computations can be stored on a variety of systems, ranging from HDFS to machine native file system to relational databases, depending upon workload. The choice varies with the access patterns of the target application and the expected latency range. If the computed data is large (in the order of terabytes), storing it in a RDMS is usually not appropriate. HDFS is not an optimal candidate for **random read-write** operations given its affinity for larger files. As mentioned in the previous section, reading a record in HDFS involves

1. TCP connection to the Namenode and reading of the metadata to locate the nearest datanode that has the required chunk data (optional in case the location is already cached in the client);
2. TCP connection to the datanode, and opening the data block that has the data;
3. a *disk seek* to reach the specific record, and
4. reading the data in a buffer to send it to the client.

Therefore, using core HDFS is not always the best solution for random read operations. Also, HDFS is more suitable to write-once-read-multiple-times kind of workloads.

In distributed systems, there are workloads which require fast random read/write operations (in the order of milli seconds), as in a scalable real-time application. Though HDFS provides the basic features of a distributed file system, its support for random read operations is far from optimal, for such use cases. Google designed and implemented BigTable [8], to cater to this requirement. It is a distributed, column oriented database, built on top of GFS [9] in order to support such random-access patterns on large data. Apache HBase is an open source Java implementation of BigTable, which stores its contents (aka tables) on HDFS. We discuss HBase in this section.

HBase belongs to the general class of NoSQL databases. These databases are not a replacement of traditional relational databases, rather they should be considered their orthogonal counterparts, preferable in use cases where the data is unstructured (where the primary requirement for RDBMs is to define and model in the database schema the relations between the data entities). They are scalable to big data (petabytes), run in a clustered environment, provide fault tolerance, offer limited transaction support, and usually the do not support specific data types (other than byte arrays). The other main advantage of NoSQL databases is their theoretically *unlimited scalability and elasticity*. One just need to add/remove nodes to scale up/down. Usually, they do not support any standard querying language like SQL, no joins; they only have custom APIs to access the data, which is stored in de-normalized schemas. This proves good for unstructured data like text, logs, web pages etc where it is difficult to find a meaningful relationships in terms of entities and such.

2.1.3.1 Data Model

In this subsection, we discuss the HBase data model using a top-down approach. One can conceptualize HBase data to be stored in a three-dimensional RDBMs table. Apart from length and breadth, the third dimension is the depth of the table, such that user can see the previous value of a given row:column cell, by asking a specific version number. The updates do not alter the cell value as such, rather a new value is inserted on to the cell stack.

HBase stores records sorted by primary key that is treated as a record identifier. This is the only key in the entire schema, and the original HBase does not support secondary indexes. After defining a HBase table, one has to define a column family. As its name suggests, it represents a collection of columns. To keep the description simple, one can say that these are defined at table creation time and the idea is to group those columns in one family which are accessed in a single transaction, like in one read/write call [1]. Each column family is stored as a separate file on the file system. This helps in limiting the number of disk I/O operations in one transaction. This optimization has significant performance advantage on typical HBase applications and it scales up to 4k reads and 8k reads per second on a 3 node cluster [2]. One can visualize stored data in these tables as a cell. The syntax of a cell value is *columnFamily:columnQualifier:value*.

For flexibility's sake, one need not define columns qualifiers before hand; they are appended to the given column family at run time. Each cell can be treated as an independent

[1]The current HBase version (0.20.6 onwards) supports adding column family at a later stage

[2]We obtained these numbers on an ec2 cluster while inserting data through YCSB

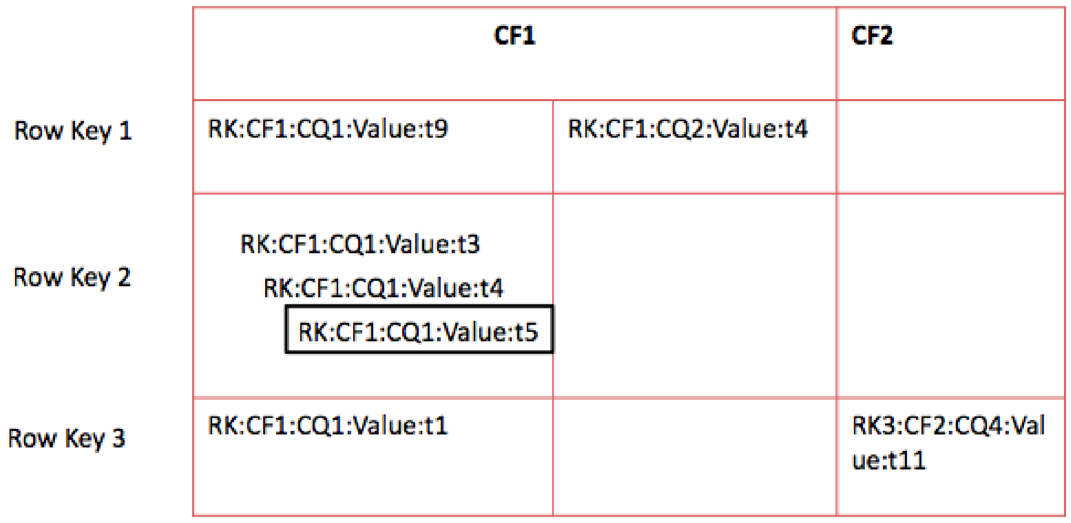


Figure 2.3: HBase conceptual schema

entity because it has a copy of its associated *row key, column family, column qualifier, data content and creation timestamp*. This gives rise to a three-dimensional datastore, where records are sorted based on primary key, one record can have millions of columns, and each cell value can have multiple timestamps, where with each timestamp value a different value is associated to that cell.

Figure 2.3 represents a sample schema of a HBase table. In the figure, the cell value is depicted as *Rowkey:ColumnFamily:ColumnQualifier:Data:Timestamp*. There are two column families (CF1 and CF2), and three rows. Row 1 has two cell values in CF1 and no value in CF2. Row 2 has three different values for CF1:CQ1 combination, latest being of timestamp t5. Row 3 has one cell in CF2 with a qualifier CQ4. It is to show that a row can have any qualifier for a given column family. There is no cost of storing empty cells.

It is possible to store sparse tables expanding to millions of columns to billions of rows. It serves the exact requirement of low latency read and write (random and sequential) with large datasets in a clustered environment.

2.1.3.2 HFile and Data Distribution

The main advantage of using HBase is the better random read/write performance as compared to HDFS in a distributed environment. This is achieved by storing the above mentioned key value (key is row key, value is the cell value), in a special file, *HFile*. When reading a record value, it is looked up based on the row key. A *HFile* is composed of fixed size data blocks, where each of these blocks contains its content in form of key values. For example, a typical size of HFile is 64MB, and each individual data block size is 64KB. At the end of each *HFile*, there is an index block, which contains the starting key value of all the data blocks present in the *HFile*. This index block is kept in memory of the *Region* server, and consulted when ever a user is looking for a specific key. All the key values in a *HFile* are inserted in ascending order, so keys in the index block are also ordered. Thus, a read operation basically involves a binary search on the index block to locate the corresponding data block in the *HFile*. Other optimizations such as using a bloom filter, and timestamps check, are also applied to further improve the response time [9].

While inserting data, HBase stores a data structure similar to these files in memory, called *memstore*, and flushes it in form of HFiles when its size increases beyond a predefined threshold. The resulting HFiles are opened while reading the records. Subsequently, the number of such flushed files keeps increasing. When the number of these HFiles goes beyond a threshold, they are merged to form one HFile (a house keeping process named as *compaction*) and the older HFiles are deleted. During a read access, all these HFiles from a *Region* are referred to form a unified result for the read. Since these files are immutable, these files are merged on a scheduled basis to form a single, compacted HFile to avoid the overhead of reading multiple files.

HBase stores tables across multiple servers by dividing the data set into *Regions* , where each *Region* represents an ordered subset of a table. It is designed such that, irrespective of the number of columns in a row, each row is always stored in its entirety, only within one *Region*. These *Regions* store their data in multiple immutable *HFiles* as explained above. There is one *meta table*, which contains the meta data of each *Region* in the cluster. When a client is looking for a rowkey, this meta table is referred to look for the *Region* server that contains that record. This is very similar to the Namenode lookup in the case of HDFS.

HBase data can be stored on a local or a distributed file system like HDFS. In a cluster environment, HDFS should be used. It leverages the benefits of HDFS and also provides the additional benefits of random reads, random writes, and file appends. HBase has proven to be an effective solution for storing large datasets as it leverages the benefits of already proven HDFS and also provides faster access to the data.

2.1.3.3 APIs

HBase provides several APIs for accessing data. It is worth mentioning here that there are no datatypes in HBase. It stores all its data as byte arrays.

1. *get(byte[] row)*: fetches a given row;
2. *getScanner(Scan)*: returns a *Scanner* object that is used to iterate on a subset of a table; the argument *Scan* defines the start/stop rows, column families, and other filters to be used while scanning the table;
3. *put(byte[] row)*: inserts a new row; and
4. *delete(byte[] row)*: deletes an existing row.

2.1.3.4 Coprocessors

Given the above APIs, the client program requests data through *get* (one row) or *scan* (multiple rows) and proceeds to process the collected rows. It is important to note here that the actual processing occurs at the client side, after the selected rows have been fetched from their respective *regions*. HBase originally did not offer any support to developers for deploying code at the nodes where the table *Regions* are stored in order to perform computations local to the data and return results (and not just table rows) to the client node. This limitation makes the cost of several computations prohibitive. Consider, for example, a row-count process: a developer has to either implement a MapReduce job over the table, or a sequential scan on the entire data, fetched locally. HBase tables are indexed based on a primary key, which enables fast access to the data when it is queried by row key, through a *get*. One can also *scan* a range of table rows by providing the start and end row keys. Sequential access is also supported at the *Region* level (through the *scan* API) and developers can subsequently run a MapReduce job, where the data of a single *Region* is provided to a single mapper.

HBase *Coprocessors*, inspired by Google's BigTable coprocessors [10], are meant as a means of creating supporting functionalities to simplify the design of the main process and they are used to implement solutions for specific types of frequent workloads. In HBase, they are an arbitrary piece of software deployed per table *Region* and can be used to act as a guard against any client or server side operation, or perform a *Region* level computation. They can be used for the following use cases.

1. **Region Observers**. They can be used to observe any *Region* activity, invoked either by a client (Get, Put, Delete, Scan) or server administration process (*Region* split, memstore flush, compactions, etc). It can be compared to the trigger functionality in RDBMs, where one can deploy triggers as *coprocessors*. Figure 2.4 represents a

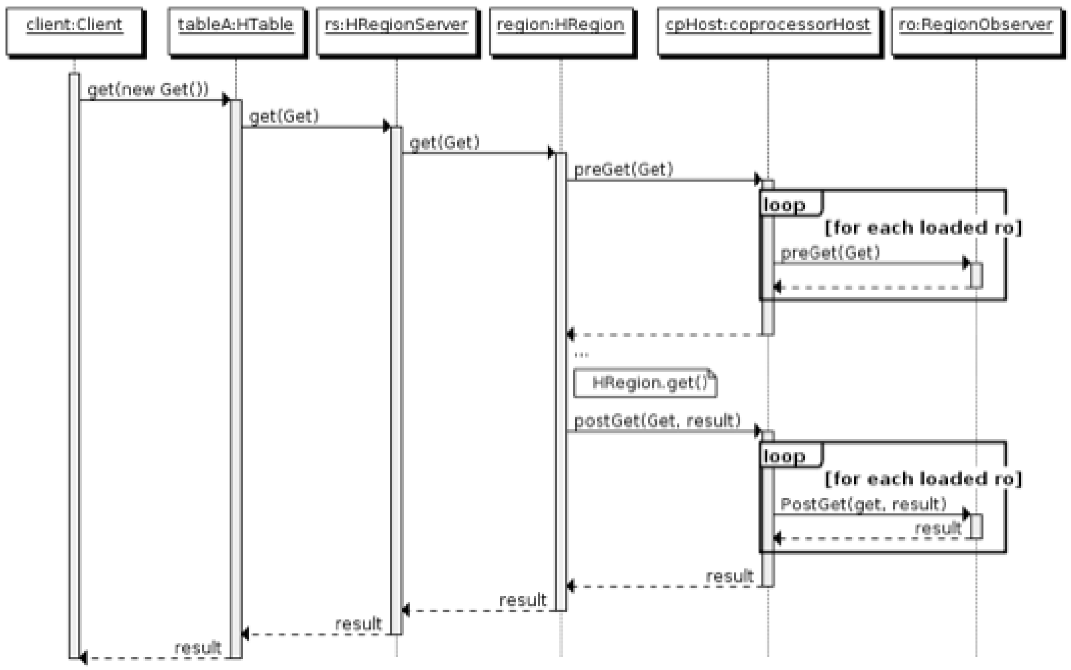


Figure 2.4: *Region* Observer flow, observing a client side *get* operation.

sequence diagram of a *Region* observer that is guarding a client side *get* operation. Note that the *Coprocessor* framework provides hooks for processing before and after these calls.

2. **Endpoints**. One can precompute results at the *Region* level and feed these interim results to the client, instead of the raw table rows. This may result in a reduced RPC traffic depending upon the use case. For example, to compute a row count on a subset of a table, the current way is to scan the entire table, which basically entails passing all rows to the client and it keeps on incrementing itself for all the rows or, run a MapReduce job on the entire table. Here, one can use a *Coprocessor* to send back the number of rows in the target *Region*, and client will do a cumulative sum of all these individual results. The client library of *Coprocessor* framework makes sure to fire all the calls to individual *Regions* in parallel. Developers need to define their own *Coprocessor* interface by extending the *CoprocessorProtocol* interface and instantiating a concrete implementation at the server side. The framework supports invocation of any arbitrary *Coprocessor* APIs from the client side and retrieve results.

2.2 Relevance of this work

In 2009, in their famous and rather "panicky" paper, Stonebraker and others from the Database community claimed that the only advantage of Hadoop over parallel databases was its easy and hassle free deployment compared to databases like Vertica and DBMS-X[11]. Two years later, one of the co-authors, Daniel Abadi [12], is a co-founder of Hadapt [13], a company selling a hybrid solution of relational database and Hadoop sub projects for big data. This is not a coincidence; Hadoop and its ecosystem projects have been getting more and more traction worldwide not only in industry but also in academia. For instance, Hadoop has been added as a part of curriculum in many universities.

In a more recent article in June 2011, Stonebraker argued in favour of the in-memory databases for simple operation applications (read and write and with no complex transactions) because of the continuously decreasing cost of physical memory [14]. He acknowledged that the "one size fits all" solution of RDBMs is not welcome and has faced serious issues in companies like Facebook, which at one point was using 4,000 shards of MySQL instances in its application logic. His suggestion of voting for memory based database does not seem plausible when one is working on data PB scale. But it shows that even a staunch supporter of RDBMs has accepted their limitations for some kind of workloads.

Apache Hadoop project is no longer limited to its map and reduce computations. The CERN project, which daily produces 1 terabyte of data uses the Hadoop Distributed File System (HDFS) as its data store, without doing any MapReduce computations [15]. Though HDFS presents reasonably good read-write performance for streaming applications, it is not an optimal file system for web-based applications that need fast random read write response. This has led to cloud databases, also referred to as NoSQL databases, where NoSQL stands for "Not Only SQL". These non-relational databases either provide a interface over distributed file system, or provide their own ingenious file systems. They are mostly key value based, column oriented, distributed file systems.

There is a plethora of Cloud databases such as HBase, Cassandra, MongoDB, CouchDB, Azure. Among these, Apache HBase is one of the most popular cloud database, having made its mark in academia and industry[5, 21, 22]. There are some interesting use cases. Mendeley, academic social network and a reference manager, has a collection of more than 99 million research papers and uses HBase at its backend [5, 16]. Facebook, which invented Cassandra [17], the closest HBase competitor in its area, picked HBase as its data store for its recently launched messaging service [18]. This is not to say that HBase is better than

Cassandra; there has been a lot of heated discussion between these two communities. In fact this has been an interesting research question in academia [19, 20], and their detailed comparison is beyond the scope of this work. The official HBase clientele web page [5] lists 38 companies that use HBase in their application stack, and has names like Adobe, Facebook, HP, Meetup, Twitter, Yahoo!, etc.

2.3 Related Research

In this section, we discuss the research most related to our work with the *Coprocessor* framework. The original HBase supports single-row transactions. Zhang et al. developed a support for multi-row distributed transactions using snapshot isolation. They proposed a group of meta tables to store the datastore state before and after each transaction [21]. This incurred extra steps of scanning and updating these meta tables before every single transaction. The cost of adding transactional support was fairly high, at least doubling the response time for most of the operations. We do not know the current status of this work, but a similar kind of transaction support using *Coprocessors* is being currently discussed within the community. The idea is that one can access such meta tables within the HBase cluster itself, i.e., one *Region* server communicating with other using *Coprocessors*. In this approach, there is no need to move back and forth between the client and *Region* server nodes.

A common use case of HBase is to store documents. Konstantinou et al. [22] used HBase for storing document indexes for a real time application. They used the existing APIs and commented that their application has to make client-side merging of two queries before rendering the complete solution. It required two server trips before producing the end result. In a similar work of creating and storing document index, N. Li et al [23] defined HIndex, that gets persisted on top of HBase and supports parallel lookup of target indexes. These indexes are fetched and the results are merged at the client side. We believe that having a *Coprocessor* Endpoint that supports streaming of results will help in such use cases, since filtering and merging can be done at the server side. Our claim is supported by the experiment and results as described in Section 3.3, where we did a similar kind of index storing and server side processing.

Apart from storing documents, HBase is also used for persisting emails in some applications. A common schema is a row belongs to a user, and a *column qualifier* for each unique word. The cell value can be the document id, its byte offset and other application specific attributes. Each word can have multiple versions based on time stamp of the corresponding emails. In the original HBase version, when a user searches his inbox with multiple keywords, it filters out all the emails that have those words (*OR* operation), and sends them to the client side. The client is supposed to do an *AND* operation on these emails to filter only those emails that have “all” the input keywords. This operation, if done at the server, helps to reduce the network traffic by sending only the relevant emails to the client side. This can be achieved using the *Coprocessor* approach. We did a similar kind of work when we migrated the TAPoR application to HBase. With this design, one can analyse multiple documents in one request, a functionality not available in existing TAPoR version. This work is explained in Chapter 4 in more detail.

HBase stores all its data in a special type of files called HFile, which is explained in section 2.1.3.2. Facebook recently proposed a new file type for HBase, HFile-v2, which consumes less memory. This work is still under review and is expected to significantly improve the overall memory footprint. Trendmicro, another major contributor, recently announced at Berlin Buzzwords [24] some new features that were added to HBase via the *Coprocessor* framework. These included a Secure version of HBase, support of aggregate functions, and index creation for Apache Lucene [25].

Chapter 3

Enhancing Query Support using Coprocessor Endpoints

In this work we have extended the HBase-Coprocessor functionality, in order to enhance the query-processing capability of the framework This work is accepted as a Research track paper in Services Wave conference, 2011, and will be printed by Springer publication [26]. The reader should remember that in its original incarnation, the core *get* and *scan* APIs of HBase simply support the selection of a (sequence of) row(s) and its (their) return to the client-side for further processing. With Coprocessors, one can define server-side code, which upon execution can return results to the client based on the scanned rows. This gives an option to perform computation at the server side, and return the computed results to the client. We used this feature for supporting aggregate functions in HBase.

The current Coprocessors calls are stateless: no client specific state is stored at the server side. This makes them lightweight and also helps in supporting concurrent requests to the database, with the underlying assumption that a client processes one *Region* in each RPC. However, there are use cases where a client needs to process these results in an incremental manner, i.e., consuming one *Region* in a sequence of multiple RPCs, which would entail saving client's state at the server side, to know its row offset in a *Region* for future RPCs. We designed an extension to the Coprocessors framework to support this *streaming* of results from *Regions* in multiple RPCs. In this chapter, we discuss our design and implementation of the HBase aggregate functions, our result-streaming extension and their evaluation. We also discuss the limitations of the current aggregate functions, which are built on top of the original Coprocessors framework, when the table size becomes too large (row count approaching 100m). We conclude this chapter with a suggestion of using results-streaming functionality to extend the implementation of these functions.

3.1 Extending HBase Endpoints

Our *Coprocessor* extension, to which we will refer to as *extended endpoint* or simply as *endpoint* from now on, is not only able to perform server-side execution of specific types of BigTable queries, but also send the result to the client in an *incremental* manner. The original *Coprocessor* framework provides support only for directly invoking *Coprocessor* code in parallel on all the interested *regions*, from the client code. Its limitation is the end user is supposed to get only one result from a *region*. With our extension, developers enjoy greater flexibility in their decision of how their code should be distributed, instead of having all their functionality at the client side.

3.1.1 Endpoint Queries

As we have already discussed, given their high-latency cost, running MapReduce jobs on a HBase cluster is less than ideal solution for on-line queries. MapReduce provides high throughput for batch analytical jobs, where there is a need to scan the entire table or a large subset of a table. Consider however use cases where (a) one does not need to process the entire table, or (b) one needs to perform a computation on the table in order to produce a single end result for the client, such as "calculating the row count on a sub set of a table with a given property" for example. Currently, one way to do such a task is to invoke a scan of these rows, collect them at the client-side, and then count them and compute the result. Alternatively, one can run a MapReduce job on the entire table (there is no way to run it on a subset of a table). Clearly, it would be better if one could actually compute the counts at the regions' level, at the server, and return them to the client that would still need to aggregate them, i.e., sum the individual counts. In such an alternative scenario, there would be fewer RPCs, less network traffic, and less client-side computation.

There are several use cases that resemble the type of workload we have described above:

- Counting the number of rows within a given row range, with some kind of filtering on the row value, and the boundary condition being a row count on the entire table;
- Aggregate-statistics functions, such as computing the sum, maximum, minimum of a specific column in a given row range; also average, standard deviation for a column in a given row range; and
- More generally, a computationally intensive task at the server-side, which would manipulate and substantially transform the table rows.

This last use case is necessarily specific to each dataset and we emulate it in Section 3.2 using the Bixi dataset.

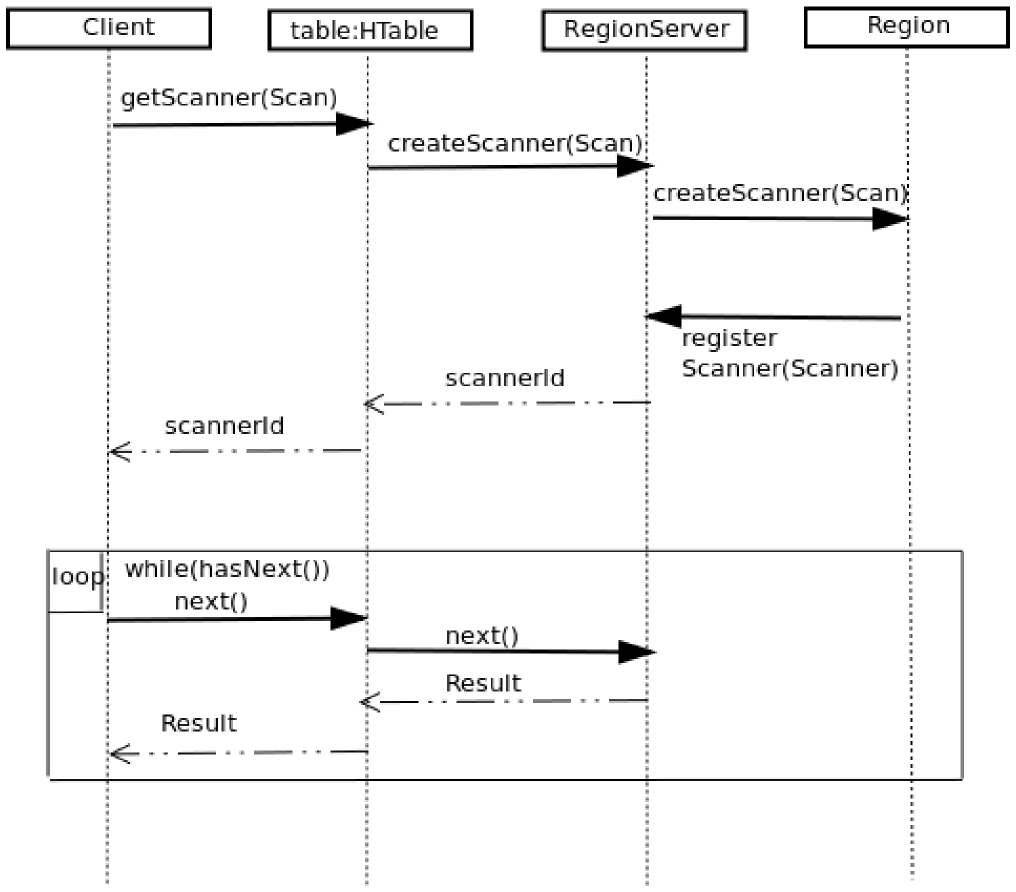


Figure 3.1: Sequence Diagram of Row Count Approach with Scan API.

Let us now discuss, in some detail, how the above use cases would be implemented with the original HBase scan vs. our implementation of scan with *Coprocessor* endpoints. The differences should be apparent in the sequence diagrams of Figures 3.1 and 3.2.

Figure 3.1 shows the sequence diagram for the use case of performing a row count with the original scan API. The first step for the client is to get a Scanner object from the HTable API. It involves instantiating a Scan object that encapsulates query-specific details like start and end row, filtering criteria, batch size of the results, etc. This causes the instantiation of a scan object at the *Region* that has the given start row, its registration with the hosting *Region* server, and the return of its identifier to the client. The client sequentially consumes the table rows from the starting *Region* while iterating over it. When scanning a *Region* is completed, the scanner automatically (from client's perspective) moves to the next *region*. This process stops when it has reached the end row.

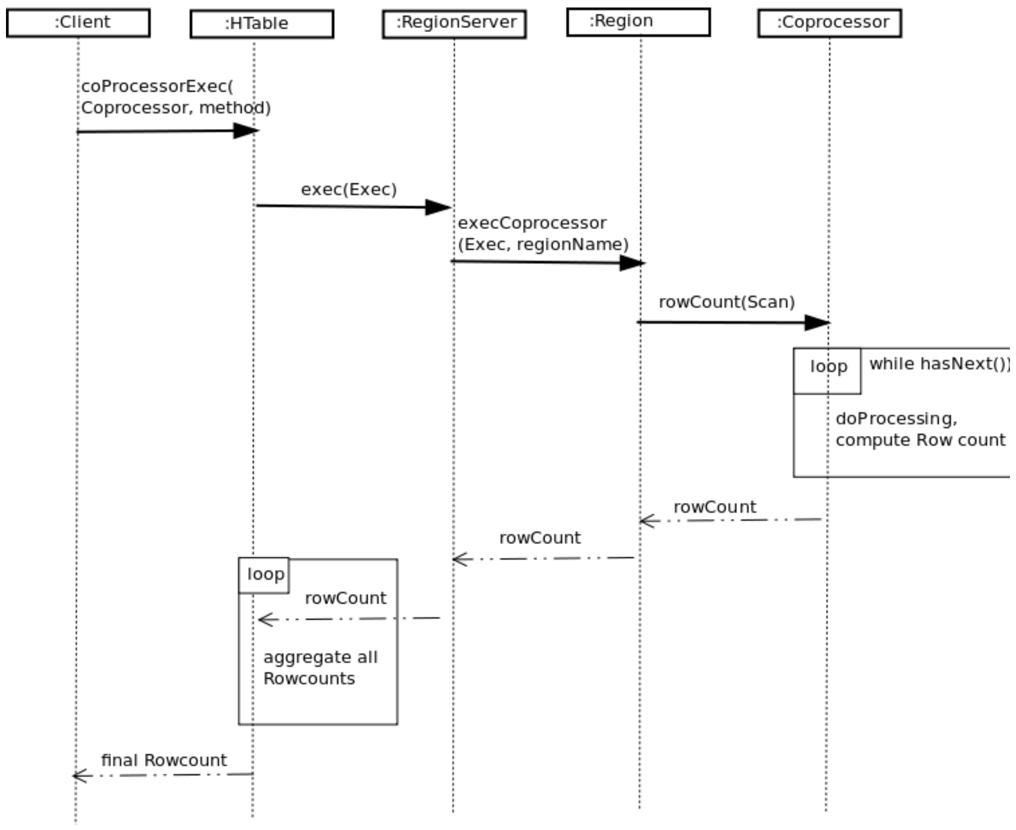


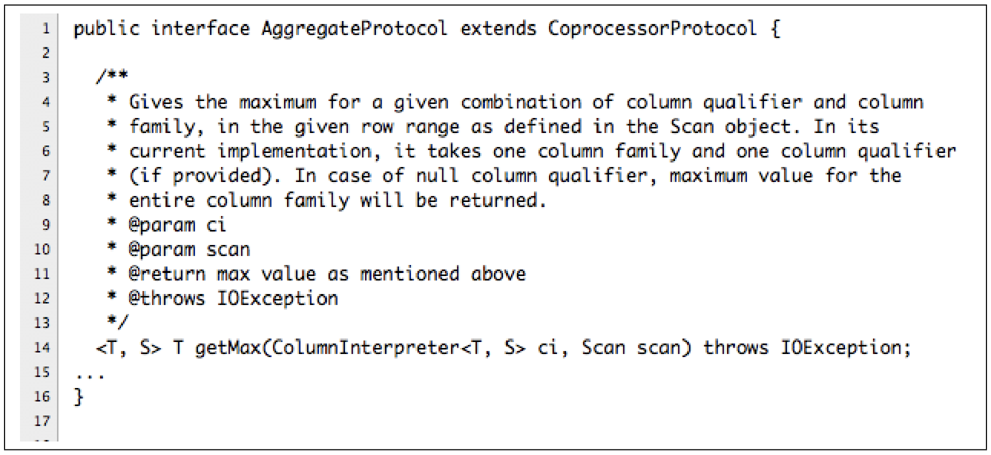
Figure 3.2: Sequence Diagram of Row Count Approach with Coprocessors.

Figure 3.2 shows the sequence diagram for the case of performing a row count using the *Coprocessor* framework. The first step is to define a *Coprocessor* implementation to load it as part of the *Region* instantiation. The next step is to define a pair of *callable* and *callback* objects at the client side. The callable object is used to wrap method invocations to the server, using the *Coprocessor* RPC framework. In this use case, one needs to invoke the row count method with proper arguments present in the deployed *Coprocessor* . The callback object is invoked when the results of the above call become available for the client by the coprocessor. Its purpose is to perform client-side aggregation of the results returned from the individual *Coprocessor* calls on various *regions*. Note that the calls to the *Regions* are made *in parallel* and are executed as a *batch process*. So, if a client calls 10 *Regions* via this mechanism, it gets its result only after when all those 10 *Regions* have returned their individual results. It is invoked per RPC invocation of callable at the server-side. Once the computation is done by the callback, the end result is rendered to the user.

The key advantage of this framework is that the calls to the *Regions* are executed in parallel as compared to the sequential flow in the previous case, shown in Figure 3.1, providing better throughput. The original endpoint functionality provides the support of invoking a *Coprocessor* API at a specific *region*, and return results. But, it suffers from a limitation that these calls are stateless. So, once a *Coprocessor* API executes and returns result to the client, it does not contain/store any reference to the call at server. Therefore, an API should process the entire *Region* before returning any result. In our work we have further improved this feature to produce a cursor infrastructure that streams results from a *Coprocessor* , as opposed to collecting them as a single batch. We explain it in more detail in section 3.1.3.

3.1.2 Aggregate functions using Coprocessor Endpoints

We designed and implemented six standard aggregate functions, i.e., max, min, average, standard deviation, sum and row count using the original *Coprocessor* framework in HBase. It involved writing a *Coprocessor* interface, *AggregateProtocol* that defines the APIs to be called on the *regionserver* side from the client side. The implementation is generic, in the sense it is independent of the nature of the table data content. Since HBase stores all its content as byte arrays, we define a column interpreter *ColumnInterpreter* interface whose implementation is passed to the aggregate APIs from the client side. It is done so that client can define how to interpret specific cell value of a table and compute the result accordingly.

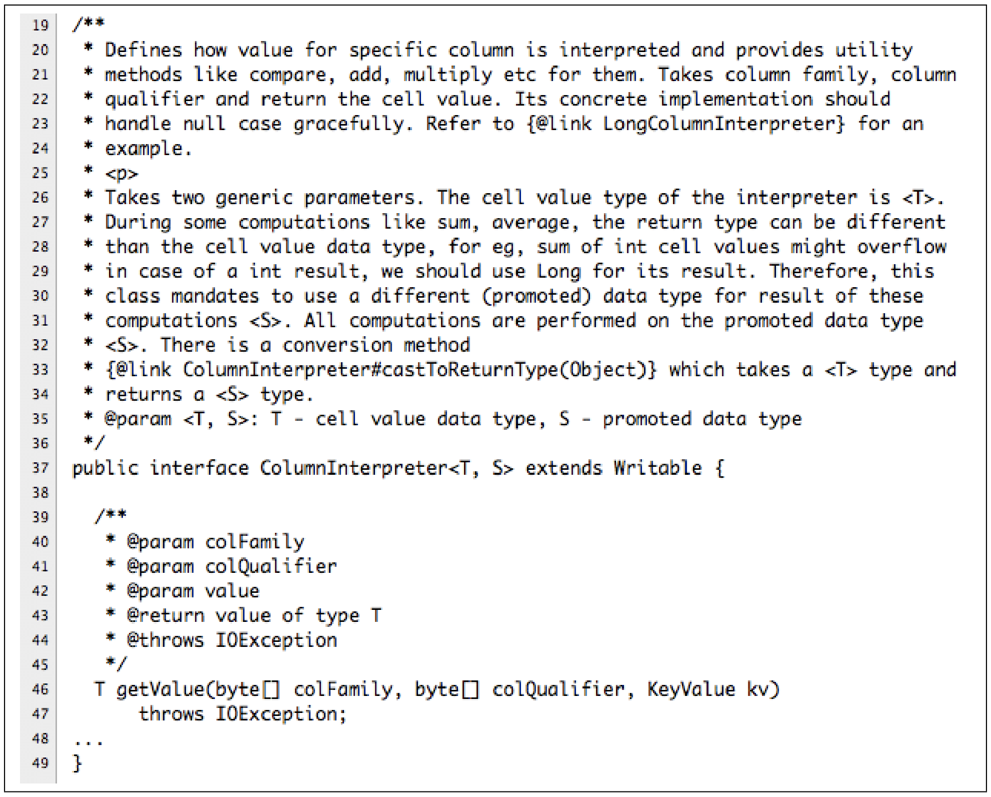


```
public interface AggregateProtocol extends CoprocessorProtocol {

  /**
   * Gives the maximum for a given combination of column qualifier and column
   * family, in the given row range as defined in the Scan object. In its
   * current implementation, it takes one column family and one column qualifier
   * (if provided). In case of null column qualifier, maximum value for the
   * entire column family will be returned.
   * @param ci
   * @param scan
   * @return max value as mentioned above
   * @throws IOException
   */
  <T, S> T getMax(ColumnInterpreter<T, S> ci, Scan scan) throws IOException;
...
}
```

Figure 3.3: Aggregate Protocol interface code sample (shows aggregate method to get maximum value of a table for a given Scan object)

Figures 3.3 and 3.4 represent the snapshot of the code of *AggegateProtocol* and *Column-Interpreter* interfaces respectively. To interpret a cell value to be used for computatation, one needs to provide a concrete implementation of *ColumnInterpreter*. This makes the aggregator implementation generic. While computing aggregate function, one may overflow the value (for example, computing sum of *int* columns may result in a *long* data type),



```
/**
 * Defines how value for specific column is interpreted and provides utility
 * methods like compare, add, multiply etc for them. Takes column family, column
 * qualifier and return the cell value. Its concrete implementation should
 * handle null case gracefully. Refer to {@link LongColumnInterpreter} for an
 * example.
 * <p>
 * Takes two generic parameters. The cell value type of the interpreter is <T>.
 * During some computations like sum, average, the return type can be different
 * than the cell value data type, for eg, sum of int cell values might overflow
 * in case of a int result, we should use Long for its result. Therefore, this
 * class mandates to use a different (promoted) data type for result of these
 * computations <S>. All computations are performed on the promoted data type
 * <S>. There is a conversion method
 * {@link ColumnInterpreter#castToReturnType(Object)} which takes a <T> type and
 * returns a <S> type.
 * @param <T, S>: T - cell value data type, S - promoted data type
 */
public interface ColumnInterpreter<T, S> extends Writable {

  /**
   * @param colFamily
   * @param colQualifier
   * @param value
   * @return value of type T
   * @throws IOException
   */
  T getValue(byte[] colFamily, byte[] colQualifier, KeyValue kv)
      throws IOException;
...
}
```

Figure 3.4: ColumnInterpreter interface, with javadoc explaining cell data type and promoted datatype.

therefore we always return the promoted data type. Hence, the generic *ColumnInterpreter* interface takes two data types, one is the data type of the cell data (denoted by T), and other is the promoted data type (denoted by S). This part of work has been committed to Apache HBase.

3.1.3 Streaming Results from Coprocessor Endpoints

As we have mentioned above, the calls to individual *Regions* from the client are done in parallel by the *Coprocessor* infrastructure. The original *Coprocessor* implementation provides the functionality for the client code to be executed at the *Regions* and the complete result set, the aggregation of the individual *Region* result sets, to be returned as the response to a single call. This is a stateless call; the server doesn't maintain any reference to the client request and, therefore, has no means to consume the results of individual *Regions* result in an incremental way. Clearly, in many cases the result from a *Region* may be quite large and the client is likely to want to consume it in an incremental fashion. This is the functionality that our extension to the *Coprocessor* framework provides: namely, enabling the client to consume these endpoint results incrementally. We demonstrate this functionality in our experiment with the NGram dataset (see Section 3.2.2).

We built our extension using the endpoint feature of the *Coprocessor* framework. An endpoint in the *Coprocessor* framework is a stateless singleton object composed in the *region*. We do not store any client side state in an endpoint as such, however, as each *Coprocessor* has an associated *environment* object that holds the context of the owning *Coprocessor*, such as the associated *Region* and *Region* server; we utilize this fact to create a *registry* or map like data structure in it.

We also define a *Cursor* interface that defines the contract of providing iterative APIs like *next, hasNext*. The concrete implementor of the Cursor interface defines the run-time behavior of the cursor, such as what processing has to be done on raw table rows, how the results per table row are to be aggregated, and how the results per RPC call should be sorted (or if they are to rendered in some specific way, for example a group by). The grouping of results per RPC is required because one RPC can return many result rows. This concrete implementation is part of the endpoint defined by the client. When a client needs this functionality in its endpoint, it defines and instantiates a concrete Cursor object as an anonymous object, registers it in the *Coprocessor registry*, and returns a handler to the client. The client can then use this handle in order to use the registered Cursor object. In this design, multiple cursor objects can be also supported.

Figure 3.5 depicts the class diagram of our design. We have removed many attributes and classes in order to give a simpler representation and to avoid unnecessary cluttering of the diagram. All Endpoints are implementations of BaseEndpointCP and provide the environment object to it. Here, the context is for NGram dataset, please refer to section 3.2.1.2 for more detail about the NGram dataset we have used. NGramImpl is the client-defined endpoint (a concrete coprocessor) that has an API to provide details of words that are similar to a given list of words, i.e., *getSimilarWords*. This API defines a concrete Cursor class and registers its instance with the corresponding Environment object of the coprocessor. The API returns the handler (an int value, *cursorId*) to the client-side. The client invokes the *next* API and pass the cursor handler. This API looks for the respective Cursor object in the *Environment* registry and invokes its *next* API on the cursor object. The cursor object knows how to process the next method and returns result to the caller.

It is important to note here that the client is getting these handlers from all the *Regions* to which the invocation was made. So, we define a client-side cursor object (named as *ClientCursor*) that encapsulates these handlers. It maintains a list of all these handlers and exposes iterative methods like *next, hasNext* to the client. Invoking *next* calls is executed

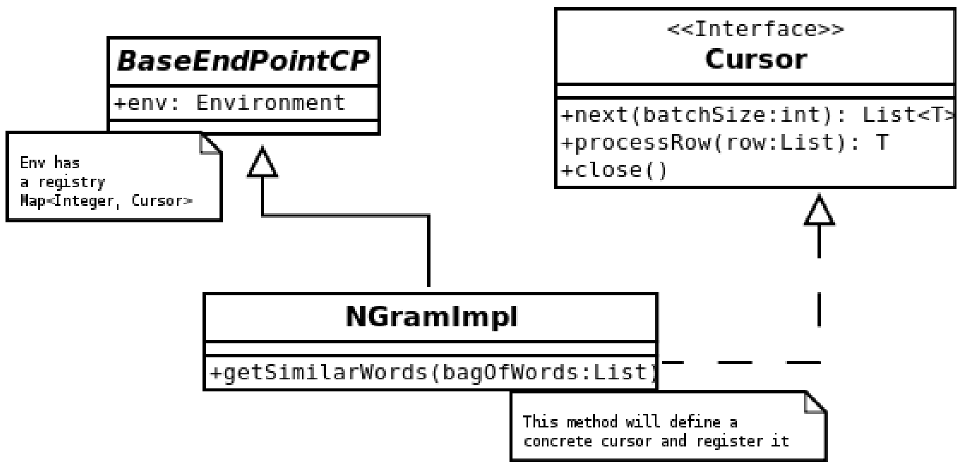

BaseEndPointCP
+env: Environment
Env has a registry Map<Integer, Cursor>
<<Interface>> Cursor
+next(batchSize:int): List<T>
+processRow(row:List): T
+close()
NGramImpl
+getSimilarWords(bagOfWords:List)
This method will define a concrete cursor and register it

Figure 3.5: Conceptual Class diagram of Streaming Results from Endpoint.

at all the interested *Regions* in parallel and aggregates their individual results for the client. Figure 3.6 gives a sequence diagram for the overall flow. Whenever a call results in a *null* result from a *Region* in the Endpoint, it is assumed that the respective cursor at the *Region* has exhausted it and that Cursor object is deregistered before it returns. When the ClientCursor notices a null result from a *Region*, it removes the corresponding handler from its handler list and does not send any request in future when the client invokes *next* on it. Subsequently, it sets its *hasNext* variable to false when the handler list becomes empty, marking the end of results from all *Regions*. To summarize, the ClientCursor takes care of all the parallel calls to the *Regions*. The developer's code is not aware of these cursor handlers that are registered at the server-side. It simply calls next(), and expects all the *Regions* to return their results. This part of work is under review from Apache HBase team.

3.2 Experiment Setup

This section discusses the experiment setup we used for evaluating the coprocessor framework discussing datasets, query selection, schema design and cluster configuration. Query selection and schema design are very much correlated with each other given our propensity over the type of processing we want to support. The reader is advised to read the next two subsections (3.2.1 and 3.2.2) in conjunction to get the comprehensive picture.

3.2.1 Datasets

To evaluate our work and analyze the relative merits of the two variants of the coprocessor framework, i.e., the original and our extension, we had to select appropriate datasets and

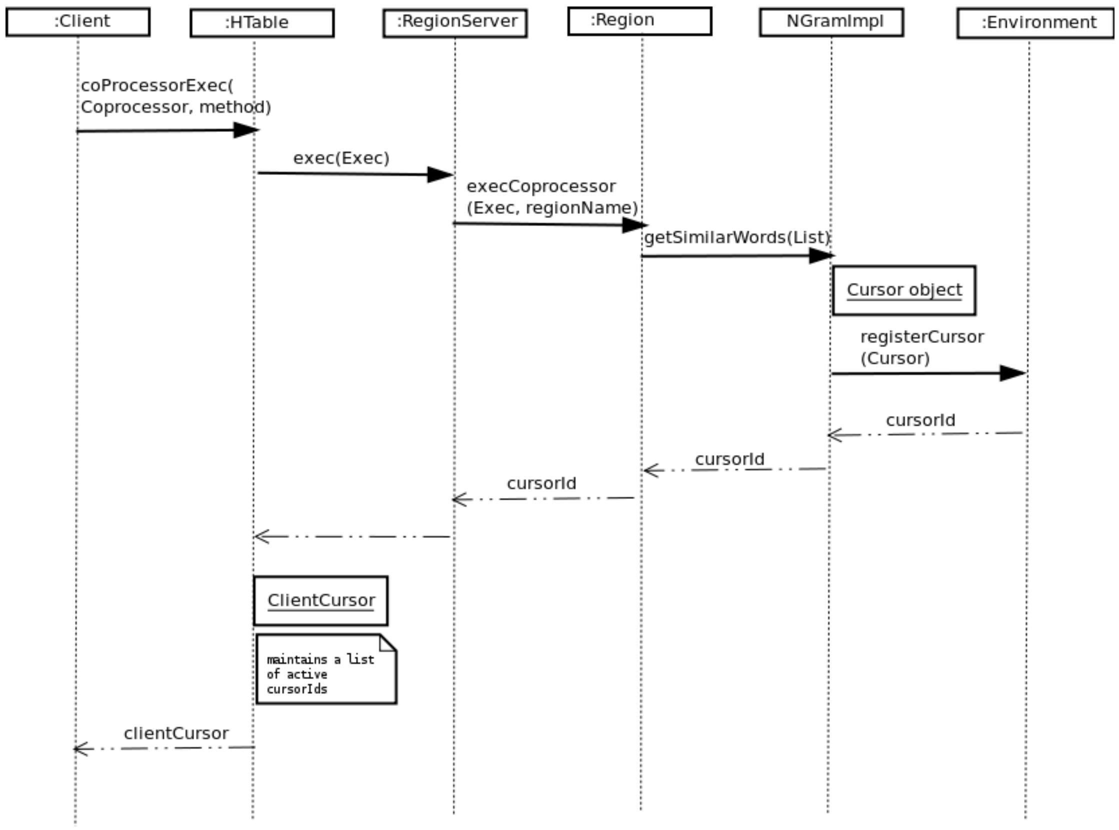
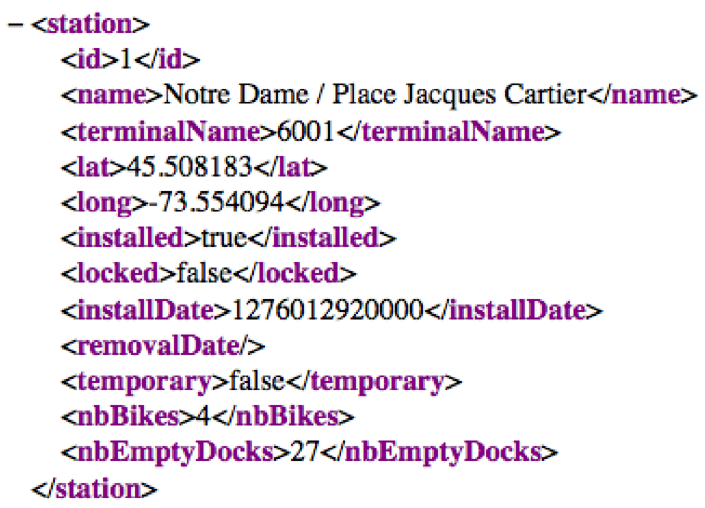


Figure 3.6: Sequence Diagram of Streaming Results from Endpoint.

queries. One one hand, the datasets need to be large enough to be labeled as 'big data' in order to defend our decision of storing it in HBase. On the other hand, they should provide opportunity to select queries that involve substantial server-side computation and also provides large results to use the results streaming functionality. An implicit (pragmatic) constraint was that they should be publicly available. According to these criteria, we decided to use the Bixi and Google NGrams dataset for our experiments.

3.2.1.1 Bixi

Bixi is a public dataset collected by *Public Bike Systems Inc.* for their Montreal operation in Quebec, Canada. This is a service for renting bikes and provides around 404 stations in Montreal. A user has to subscribe to the service order to use the facility. Once he is a member, he can take out and return a bike in a station based on the availability of bikes and empty docks respectively. These stations are equipped with sensors, which transmit this information at regular intervals. This information is stored and is made publicly available to the user, who can issue a query to look for information for all these stations at any given



```
– <station>
    <id>1</id>
    <name>Notre Dame / Place Jacques Cartier</name>
    <terminalName>6001</terminalName>
    <lat>45.508183</lat>
    <long>-73.554094</long>
    <installed>true</installed>
    <locked>false</locked>
    <installDate>1276012920000</installDate>
    <removalDate/>
    <temporary>false</temporary>
    <nbBikes>4</nbBikes>
    <nbEmptyDocks>27</nbEmptyDocks>
  </station>
```

Figure 3.7: Sample bixi data for a station.

time. This status information is rendered as a XML file.

Figure 3.7 shows a segment of the XML document representing the status of one station. It has the information of station id, name, geographical coordinates, docks status and other station related information. This data is publicly available and one can get information for all these 404 stations at any given point of time. [1] We used the dataset that was collected on a per minute basis for a period of 70 days, from September 24, 2010 to December 1, 2010. It is a 12 GB dataset that contains 96,842 data-points for all the Montreal stations.

3.2.1.2 Google Ngram

Google Ngrams is a collection of ngrams from the entire Google-Books collection, which the company made available for research work in 2009 [27]. These are one, two, three, four and five word grams for all the published books that are there in its repository. We used the 1-gram dataset for our experiments, which contains 6,641,214 unique words. Table 3.1 represents the structure of 1-gram dataset. The first column is the word, second column represents year; third, fourth and last columns represent the word count, unique pages and unique books frequencies containing the word in that year. For example, in year 1938, word 'America14' appears 5 times on 5 pages in 5 distinct books.

3.2.2 Sample Queries

Evaluation of the result-streaming functionality requires a specific set of queries where one can use the streaming functionality for coprocessor generated results. As mentioned

[1]`https://profil.bixi.ca/data/bikeStations.xml`

Word	**Year**	**Word count**	**Unique pages**	**Unique Books**
America14	1936	1	1	1
America14	1938	5	5	5
.....	....	....	....	....
Americaensche	2001	17	14	7
Americaensche	2002	11	11	9

Table 3.1: Google 1 gram data sample (from [1])

earlier, this implies two requirements: (a) there should be a substantial computation that can be transferred to server side and, (b) the result set should be large enough so that client needs to do more than one RPC to fetch the result from a *Region* . It is to be noted that original HBase provides optimizations where one can send a customized number of rows from the server to the client in one RPC. To have a fair evaluation between original Scan and Streaming result, we keep this as high as 1000 in most of our experiments, i.e., in each RPC, send 1000 *table rows* for a Scan and similarly, 1000 *result rows* with Coprocessors from server to client. Therefore, the queries should be able to provide a large number of results. We came up with four different type of queries, based on the datasets we chose.

3.2.2.1 Bixi Queries

Our choice of the Bixi dataset was motivated by the fact that it provides opportunities for several interesting queries, including general descriptive statistics (sum, min, max, average and standard deviation) but, more importantly, potentially complex domain specific queries. In our experiments, we have implemented the following two queries, the first invoking a *get* on a given time-stamp and the second invoking a range *scan* over a sequence of time-stamps.

- **Query 1**: For a given time, central location and radius, get a list of stations with available bikes, sorted by their distance from the given location.
- **Query 2**: For a given list of stations and a time, get their average bike usage for last 1, 6, 12 and 18hr. Its boundary condition is to get such an average for all the 404 stations.

3.2.2.2 NGrams Queries

The NGram dataset has already been the subject of much study. In this work, we wanted to take advantage of the HBase platform to explore it in ways that is not currently supported on the Google's NGram viewer [1]. The current NGram viewer can be used to see evolution for a specific word or a set of words. It does an exact match of the given word to its dataset. There are some words that share a common prefix up to a good part of their length, like there are 424 different words that starts with ‘America’, spanning across almost 530 years! We use this dataset for the following queries.

- **Query 3**: For a given word prefix, get the top 3 count frequencies with their respective years for all the words that share that prefix. For the prefix ’America’, a result set of 424 rows is produced.
- **Query 4**: Similarly for a bag of words in one call, look at the evolution of words like, ‘love’, ‘blood’, ‘passion’.

Again, we should note that Query 3 involves a range scan with a common prefix, the range for Query 4 was large enough to span multiple *Regions* such that it was executed in parallel across different *Regions* using the Streaming results API. We will discuss the result of choosing the above 4 queries in section 3.2.3 after describing the schema for both datasets.

3.2.3 Schema Design

The performance of HBase is directly correlated to the underlying schema and data-access patterns. It stores its data on top of HDFS and uses the data locality provided by it. There are some key guidelines in creating the HBase schema.

- **Group frequently read columns in one column family:** A *Region* server acts as a HDFS client and stores the table data in HFiles, a file format specific to HBase. A HFile technically represents one column family; so columns that are accessed simultaneously in a call should be placed in one family. This reduces the number of HFiles to be read for one call. One important consideration is HBase tries to open all HFiles and read their index blocks in memory. This way, it can determine which HFiles are “interesting” for a specific query. Therefore, lesser number of HFiles are better.
- **Minimize number of column family:** Having a minimum number of column families helps in keeping the number of open files during a read operation small and also reduces other *Region* level overheads.
- **Avoid hot *Regions* by sprinkling keys:** For a write-heavy application, the row key selection should be done such that writes are distributed normally across all the *Region* servers. Usually, for most applications the intuitive choice is to select a chronological key, but this might result in all the writes going to one active *Region* at a given

Row Key	CF:1	CF:2	CF:3
<**Timestamp1**>	NotreDame, 45.508153, -73.554094, 4, 27	Saint-Antoine, 45.512323 , -73.5539304, 1, 24	Saint-Laurent, 45.51066, -73.56497, 3,12
<**Timestamp2**>	NotreDame, 45.508153, -73.554094, 3, 28	Saint-Antoine, 45.512323 , -73.5539304, 4, 21	Saint-Laurent, 45.51066, -73.56497, 0,15

Table 3.2: Bixi Schema.

instant of time. One can elaborate such keys with some application-specific information to cause a different distribution pattern, such as hashing those keys. This avoids the problem of having a hot *Region* in the cluster and makes more efficient use of the resources.

- **Do not exceed the cell value size beyond a defined threshold:** As mentioned in section 2.1.3.2, a *HFile* stores its content in form of key values in multiple data blocks, of fixed size. Practical experiments have shown that with larger 'key value' (HBase table cell) size (larger than 50MB), read write performance of HBase suffers badly. It is advised to store large key values directly in HDFS, while just storing a reference to the HDFS location in HBase.

Table 3.2 represents the schema for the bixi dataset. We chose the row key to be timestamp because our access pattern is time-stamp based, and the application is not write heavy; it has only one write per minute (if it was to be a real time application). We define one column family and used the station ids as column qualifier. Instead of storing direct XML structure, we extract relevant information like longitude, latitude, available bikes, empty docks, and stored it in a predefined format as the qualifier value. This removed the redundant XML tags and reduces the dataset size to 9GB. As per Table 3.2, at timestamp1 and timestamp2 value of available bikes and empty docks at station id 1 is 4, 27 and 3, 28 respectively. A typical row size of the table data is approximately 90KB.

With the above schema, Query 1 processes only 1 table row. It entails a complex computation of calculating distance between 404 stations. This query helps us in evaluating whether to pass the entire row to the client or compute the result at the server side. Query 2 accesses variable number of rows depending upon the hours duration. This helps in realizing the response time variation of the two approaches as we increase the number of rows to be processed.

Row Key	**CF:1936**	**CF:1938**	**CF:2001**	**CF:2002**
<**America14**>	1,1,1	5,5,5		
<**Americaensche**>			17,14,7	11,11, 9

Table 3.3: NGram Schema.

Table 3.3 represents a snapshot of the schema for the Ngram dataset. Its value correlates to the sample dataset shown in Figure 3.1. We chose a word as the row key, and years as column qualifiers. All the frequencies are concatenated with a delimiter and stored as the qualifier value. Thus in order to look for a word, we need to read only one row. Accordingly, the row size distribution varies as per the word history, varying from a few bytes to 10s of KB depending upon the popularity of a word. Thus, Query 3 processes a group of rows that are located near each other (rows are sorted by row key, and in this schema, each word is a key) as we are looking for words that share a common prefix; and Query 4 accesses similar groups in parallel.

3.2.4 Cluster Setup

We used a 5-node cluster on Amazon EC2 to run our experiments. Table 3.4 describes our machine specifications. In order to get a better understanding of various processes running in the experiments, we briefly describe the processes that run in a typical HBase cluster.

- Namenode: HDFS master, which keeps track of metadata for all the blocks in the file system.
- Secondary Namenode: HDFS fault tolerance solution for a Namenode failure.
- Job Tracker: master for tracking all MapReduce jobs in the cluster; it assigns tasks to Tasknodes.
- Task Tracker: working process that processes the MapReduce computation. These processes report to Job Tracker.
- Datanode: process that keeps the data block in HDFS. These processes report to Namenode.
- Zookeeper: Quorum for managing cluster. Each active node registers itself in the quorum to become part of the cluster.
- HMaster: HBase master, which coordinates the HBase cluster like coordinating *Region* movements, RegionServer registration.

Node No.	Instance type	Processes running
1	m1.xlarge	Namenode, Secondary Namenode, Job Tracker, HMaster, TestClient
2	m1.xlarge	Zookeeper
3	c1.xlarge	RegionServer, DataNode
4	c1.xlarge	RegionServer, DataNode
5	c1.xlarge	RegionServer, DataNode

Table 3.4: Cluster configuration

- RegionServer: serves *Regions* of HBase tables. It manages HBase data and serves it to the client. It acts as client to the underlying HDFS.
- TestClient: It is the process used to run the experiments.

We used the recommended node configurations, giving more memory to HMaster (HBase Master process) and Namenode, and larger computing resources to Datanode and Region-Server processes. Since Coprocessors are not yet part of the released version, and we have added/modified the HBase source code to add our functionalities, we created our own Amazon machine image (AMI) containing relevant Hadoop and HBase jars [2].

3.2.5 Experiment Results

We uploaded the bixi and ngram datasets to Amazon S3. Because the datasets contain a varying size of rows, in order to have a fair evaluation between the two approaches (reading from cache or from disk), we pre-run a query with the same arguments, a number of times in order to cache the required table rows in the *Region* server. We executed these queries from simple Unix console on to the HBase cluster for at least three to five times till we get almost same value and measured the average response time. This way, even if the operating system cache plays role while reading the values, it should have the same effect for both variations. We performed experiments for all the four queries as described in the section 3.2.2. For all these queries, we set the batch size of scan and the cursor to be 1000 (to minimize the number of RPCs).

Table 3.5 gives the result for Query 1. As expected, since these are simple *Get* like queries on one row and involves computation of calculating distance, they results in almost same response time. We discuss it more in section 3.3.

We used Query 2 for computing the average for the last 1, 6, 12 and 18 hr from a given time value with two variations. In one condition, we compute these values for 3 stations,

[2]This AMI is publicly available with tag name “himanshu-hbase”

Query 1	**Get**	**Coprocessor**
response time in sec	1.57	1.61

Table 3.5: Response time for Query 1, in seconds

Prefix	**Cache size**	**Scanner time (in sec)**	**Coprocessor time (in sec)**	**Number of unique words**
America	1000	1.84	1.66	424
America	100	1.84	1.80	424
A	1000	29.65	**21.15**	219,797
blood, love, change, passion	1000	158.21	**44.7**	NA

Table 3.6: Response time for Query 3 and 4 on NGram dataset.

and in other we compute it for all 404 stations. Figure 3.8 shows the response time of Query 2 for 3 stations. The X and Y axes represent the hour for which the average was computed and the response time. In the scanner approach, the relevant columns are fetched for the rows that fall in the time range and the client computes the average value. In the case of 3 stations, we set the scanner object to fetch only those specific columns. Figure 3.9 shows the response time for all 404 stations. We chose 3 and 402 as the boundary conditions of the query, which should enable us to understand the trends governing the performance of our approach.

Table 3.6 shows the response time for Query 3 on the Ngram dataset. The target word used was ‘America’ and there are 424 distinct words that start with this prefix. The scanner cache size (number of results to be returned in a RPC) was set to 1000, so all the results were returned in a single call. In order to test the streaming result from Endpoint functionality, we executed Query 3 to fetch all words that starts with ‘America’ with batch size of 100. The results were almost the same with the case when the batch size is 1000. We also test the streaming results by executing Query 3 to fetch all words that starts with prefix ‘A’ with a batch size of 1000. There are 219,797 such words spanning across 2 *regions*.

The last row of Table 3.6 shows the experiment results for Query 4. In this query, a user enters a bag of target words and it returns all the words that match the prefix of either of these words. We used words that start with different letters to evaluate the parallelism provided by the Coprocessor framework.

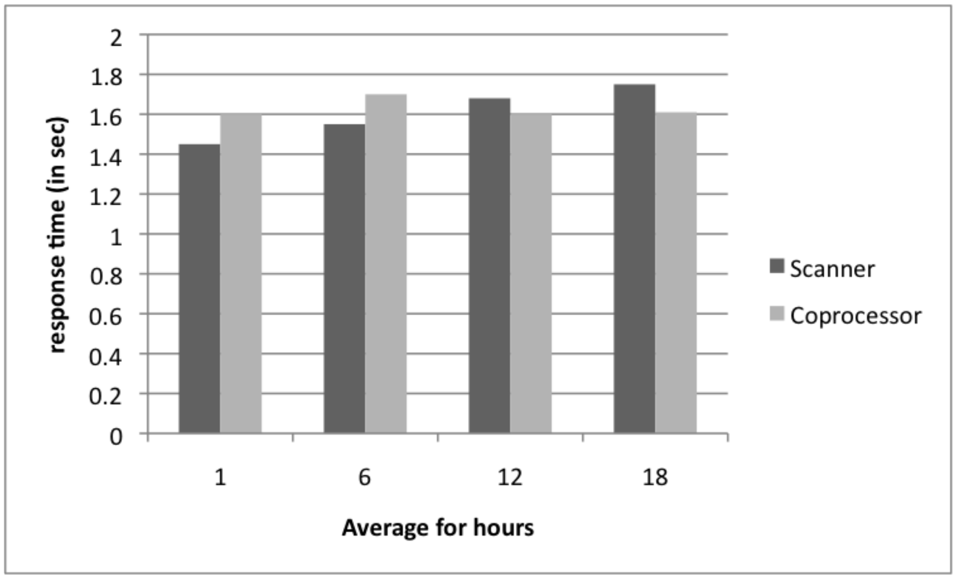


Figure 3.8: Response time for Query 2 for 1, 6, 12 and 18 hrs time range for 3 stations, in seconds.

3.3 Evaluation

The four queries described above represent different use cases and we consider them separately. Table 3.5 reports the response time for Query 1. This query points to a single row using only one row key and results in almost similar response time for both approaches. Although it involves the computation for calculating the distance between a point-of-interest and each of all the 404 stations, the effect of this computation cost on the response time is almost negligible, which can easily be correlated to the node capacity running the client. Thus we almost get similar result for both the variations.

We evaluated Query 2 under two different scenarios, one with 3 stations and other with all 404 stations, with scan cache size and streaming result batch size equal to 1000. The experiment is to compute the average number of the bikes available at each station for last 1, 6, 12 and 18hr from a given time. Since the dataset is on a per-minute basis, 60, 360, 720 and 1080 rows need to be read respectively. Figure 3.8 shows response-time results with 3 stations. Note for the bixi schema, stationIds become the column qualifiers, and one can scan at a granularity of qualifier level. So, we select only those 3 columns for this scenario. The response time for 1 hour computation was better in the case of the scanner and as we move towards 18 hr average, the Coprocessor method started winning. This is because, in the case of the scanner, all the selected values are sent to the client and with a larger time window, the number of such rows increased from 60 to 1080. In the case of

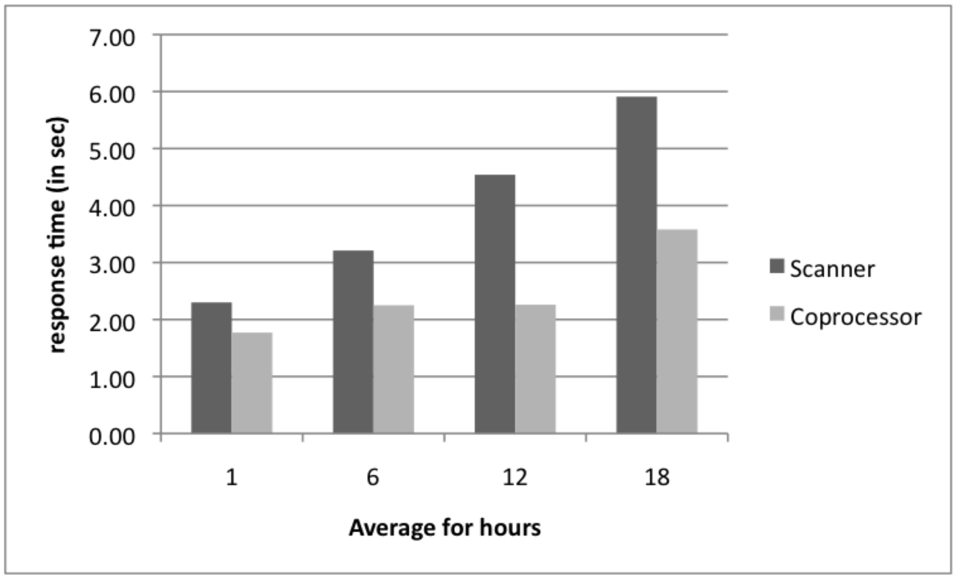


Figure 3.9: Response time for Query 2 for 1, 6, 12 and 18 hrs time range for All stations, in seconds.

the Coprocessor, the computation of the average was performed at the server-side and only the average value was sent to the client. This difference gets manifested when we increase the number of stations from 3 to 404. Figure 3.9 shows the result of Query 2 with all 404 stations. In this case, the scanner needs to pass the entire row to the client. With each row sizing 90KB, RPC and client-side computation cost builds up as the time range is increased. In this use case, the Coprocessor gives better result than scanner for all the time ranges that we tested. This shows the Endpoint are more effective than normal scanner when it involves computation with a large amount of data.

We executed Queries 3 and 4 on the NGram dataset. We used this dataset to test the result-streaming functionality and tried to build up the case when we can exploit the parallelism that comes inherent with the Coprocessor framework. We tested Query 3 with the prefix ‘America’. As mentioned in the section 3.2.2 we were interested in top 3 frequencies of words that share the same prefix. The motivation in designing this query was to ensure that, apart from fetching the rows, it should have some computation that reduces the result size. For example, the row key ‘America’, which matches the criteria, has more than 500 frequencies for these many years. Table 3.6 shows the results for this experiment. The difference in response time for Scanner and Streaming Endpoint approach is not large, and its reason is once again, the computation and result size is fairly small to make any impact. The overall size of the result was about 630 KB only, which can be termed as an insignificant amount and explains the near same time (rather Scanner doing it marginally better).

Interestingly, there was almost no effect of changing the cache value from 1000 to 100 in the case of a Scanner. This can be attributed to the smaller result size and nuances of virtual environment.

We aimed to test the result streaming, therefore we executed a query where the prefix was ‘A’;, i.e.,, give the computation for all words starting with the letter ‘A’. There are around 0.2 million such words, distributed in 2 *regions*. The benefit of Endpoint approach becomes evident when it did the task in 21 seconds as compared to 29 seconds taken by the scanner. We further expanded this test to consider the query for a bag of words (Query 4). We aimed to further increase the number of target *Regions* and test the result-streaming functionality. We tested it with a group of 4 words, chosen such that they are popular (good enough to produce large amount of results) and start with different letters to make it more distributed. Before executing requests to individual *regions*, the Coprocessor framework first looks for the range of row keys that are provided. So, it needs to look at the start and end row, which are derived from the argument list, and then it executes requests across all the *Regions* that lie in the range. There are 12 such *Regions* . It may happen that some interleaving *Regions* do not have any of the required words, in that case a null result is returned from them. The ClientCursor makes sure that these *Regions* are not called in the next invocation. In the case of scanner, the call begins at the *Region* having the start key, and flows sequentially to the *Region* containing the end row. The benefit of the Endpoint approach becomes clear when it finishes the task in 44 seconds as compared to 158 seconds taken by the Scanner API. Figures 3.10 and 3.11 present a schematic diagram of both these cases. The former figure shows the Coprocessors client making RPCs to different *Regions* in parallel, whereas the later is a sequential scan across multiple *Regions*. The *Regions* are shown as an *ordered* stack with their start and end keys marked on it. The ordering is based on the row keys. The client requests are shown in black, and the server responses are shown in blue.

In the case when an interleaving *Region* does not have any result rows, the Endpoint has to scan the entire *Region* before sending the null result. Scanning and entire *Region* in one RPC may take a while in the case when there are many such *Regions* on a *Region* server. Therefore, in the first call to fetch result, these null resulting *Regions* behave as bottleneck because ClientCursor needs to aggregate results from all *Regions* before rendering result to client. We discuss this bottleneck in more detail in the next section.

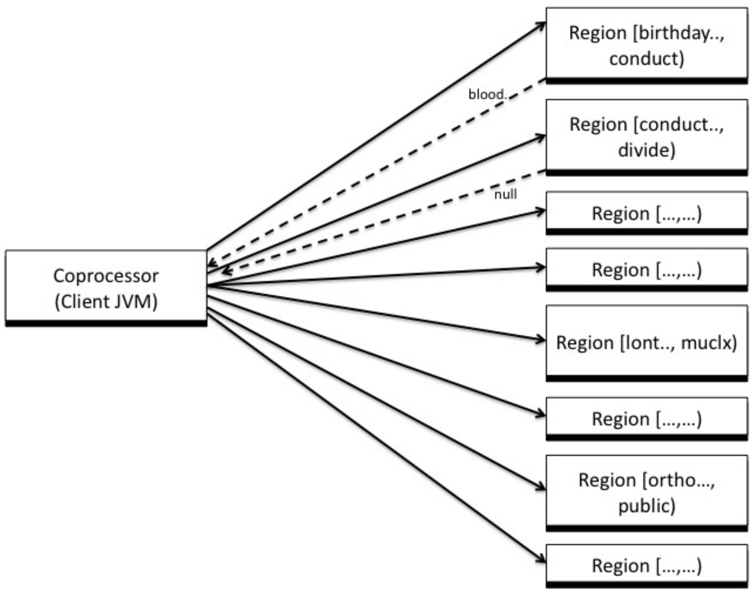


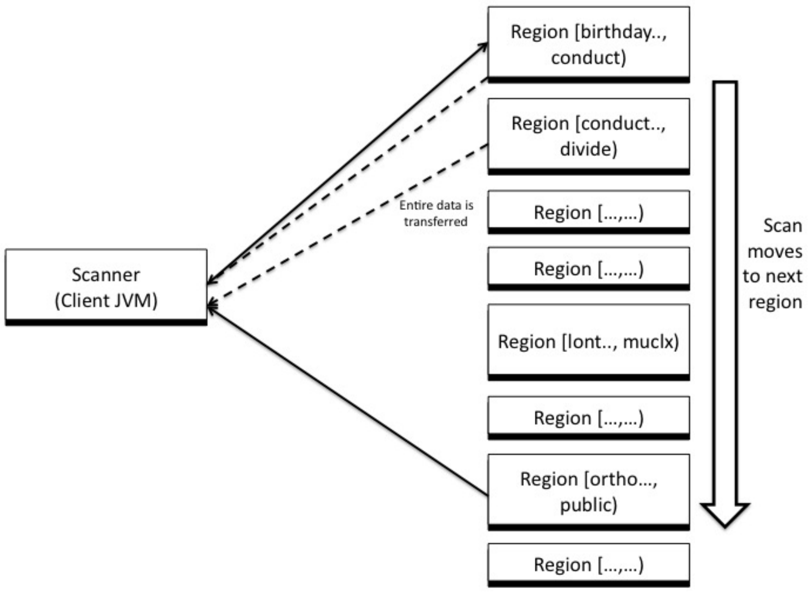
Figure 3.10: Response time for Query 2 for 1, 6, 12 and 18 hrs time range for 3 stations, in seconds.

3.4 Aggregate Functions Benchmarking and Future Work

We now discuss the benchmarking results for the aggregation functions 3.1.2 that we developed using Coprocessor Endpoint. Since all these functions follow a similar call stack, we will discuss only the Row count operation and it can be generalized to other functions. We used the Yahoo Cloud System Benchmarking tool to generate our dataset [19]. One can use it to load a configurable dataset (in terms of record count and row size). On a similar set up as used for other experiments discussed above, we loaded the table with 1m, 10m and 100m records of size 1KB each in three iterations. We ran the row counter task with a using a normal scan, MapReduce job and row counter using Coprocessor framework. The cache size for all these three approaches was set to 1000.

Table 3.7 details the response time for these three datasets. It is evident from the results that the coprocessor-based approach is faster for datasets size up to 10m. MapReduce provides high throughput with larger datasets. It has a higher *starting cost* and it gradually picks up as the data set grows to 100m, which justifies its use for batch analytical jobs. We used FirstKeyValueFilter (a filter provided in HBase that makes sure we are reading only one column per row to avoid reading the entire row). It is just an optimization for row count like operation.

As the dataset increased to 100m, we get time out exception at the coprocessor client-



Region [birthday.., conduct)
Region [conduct.., divide)
Entire data is transferred
Region [...,...)
Region [...,...)
Scanner (Client JVM)
Scan moves to next region
Region [lont.., muclx)
Region [...,...)
Region [ortho..., public)
Region [...,...)

Figure 3.11: Response time for Query 2 for 1, 6, 12 and 18 hrs time range for 3 stations, in seconds.

Row Count	**Scan**	**MapReduce Job**	**Row count with Co-processor**
1m	8.2	24	**1.6**
10m	240	118	**100**
100m	Not tested	**631**	Failed

Table 3.7: Response time of row count operation with 3 approaches, in seconds.

side.

This is because there are 256 *Regions* across 3 *Region* server nodes for the 100m rows table. The row-count operation essentially involves reading all the rows of the table, i.e, 256 *Regions*. In HBase, the default "timeout" time is set to be 6 minutes. This means that the maximum time for a request to be executed by HBase is 6 minutes, else it will get a timeout exception. In the row count case with Coprocessors, when the client issues the request, it goes to all the 256 *Regions* in parallel. Though each server has 100 request handlers (each handler is a thread listening for client request), it takes more than 6 minutes to read all the 256 Region s. In other words, we are setting an upper time limit of 6 minutes for this operation. The actual computation takes more time than the default value. The client library is fail fast, in the sense that even if one *Region* fails to render response in the set timeout limit, the entire process is failed. So, in effect, we are trying to read the entire table in 6 min, and it fails otherwise.

It is worth mentioning that the row counter (or any other aggregate function built on top of Coprocessor) does not uses result streaming, where we can define the cache size and use the cursor like functionality. Therefore, the entire *Region* has to be scanned before returning result to the Coprocessor client. In the near future, we plan to rectify this by using the result-streaming approach. In this approach, we can define a cache limit, and there is no need to scan the entire *Region* in one RPC, which should alleviate this problem. Resolving this limitation is our immediate future task.

We also plan to test it under a real time system under active load, in order to explore its behaviour not only with "read" operations but with a mix of "read" and "write" operations. One important application of the result-streaming functionality is in parallel scanning. One can use the inherent parallelism provided by the Coprocessor framework in coming up with a design to support for parallel scanning, as compared to the existing sequential *Scan* API. It has been long sought in HBase project [34], and we plan to extend result-streaming functionality to support parallel scans.

3.5 Summary

In this chapter, we discussed the design and implementation of aggregate functions using original HBase Coprocessors. The implementation is generic to cater to support any kind of data in the table; the end user is supposed to provide an implementation to interpret the byte arrays in terms of the application data types. They present a better alternative as compared to a sequential scan or a mapreduce job, in workloads when one is interested in a subset of table. Though increasing the size of the subset (to 100m approximately) results in timeout exceptions as the default timeout limit is 6min. One possible workaround is increase the timeout limit. A cleaner approach would be to use streaming Coprocessors results API, where one can get incremental results from the *regions*. This way, one RPC can in much lesser time (as it is not exhausting the entire *Region* before returning). We discuss this extension, and also supported our claim with experiments on NGram and Bixi datasets.

Chapter 4

Migrating TAPoR to HBase

Up to this point, we have discussed Hadoop, HDFS, HBase and its detailed design, including Coprocessors endpoints and result streaming. This gives us an insight about the features and behavior of these distributed computing projects. In this chapter, we switch our perspective to that of a developer, and build an application leveraging these projects, specifically HBase. From this perspective, we share our experience migrating an existing text-analysis web application over to HBase. By migration we mean re-designing the business logic and the back end store, and exposing it as web services so that any kind of front end (existing or future) can use it. This approach can be used for other types of applications also. Overall, any such migration involves selecting a *proper* application (scalable application with some degree of *un-relational* domain objects), designing a corresponding HBase schema, implementing the application services and exposing them as web services to support existing or any future front-end client.

In this chapter, we explain our motivation for selecting TAPoR as the target application for migration. There are some basic steps that are to be followed when migrating an application to HBase. For instance, one should know the application workload, its read-write access pattern, load, peak and average load. This plays a vital role in coming with a good schema, which is one of the key factor in any application build on top of the HBase stack. We explain our methodology for schema design in following sections. We also discuss in detail our use of Rest web services, and Stargate (HBase Rest server) and Restlet (TAPoR specific *middleman* application), that forms the core parts of our application stack. In the end, we provide a list of the TAPoR APIs reimplemented in HBase.

It is to be noted that we did a similar kind of migration of TAPoR in a previous work and used core HDFS as the backend, and significantly improving the over all end-user response time as compared to the original version [28].

4.1 TAPoR

As we have seen so far, HBase is about storing *big data*, and it does it in a de-normalized form. Text analysis, especially syntactic and semantic analysis of electronic texts provides an exciting opportunity to deal with unstructured data with HBase. This is hot area of research for Digital Humanists, who tend to explore electronic data for both these kinds of analysis, semantic for analysis of intertextuality of documents to infer influence relations among them, and syntactic for analyzing the words in the texts, their frequency and patterns. The later provides a varying workload from computing a simple word count to finding top K words, to finding concordance of given words. A common use case includes doing the analysis for one document or on a collection of documents, and repeating the experiment with different request parameters. These are among the many operations supported by TAPoR, Text Analysis Portal for Research. TAPoR is a lexical analysis web based application, developed by Digital Humanists [29]. The tool offers recipes for listing words and word counts, finding word co-occurrence, finding concordance, generating word clouds, and more. TAPoR has several deployments around the world and an increasing number of users, who use it to analyze the lexical properties of texts and collections. The existing TAPoR implementation suffers several limitations that make the flexible experimentation of Digital Humanists with different collections a challenge.

One major bottleneck is its lack of scalability to larger documents, due to its design; it was originally built to cater to small documents, processing the document in its entirety for all requests each time, though the new request may have only a different parameter from the previous one. So, if one does a concordance for a word ‘love’ and now wants to do for ‘blood’, it will process the entire document and look for ‘blood’. In this approach, we are doing the same computation again but with different request parameters. This approach suffices when the document is small, but it is not scalable to larger documents (size in few MBs is sufficient enough it to give a response time out error). This results in a poor end-user experience, as each request takes the same O(n) time, where n is proportional to the document size. An alternative to this approach is the standard way of creating indexes of the document, and then using these indexes for TAPoR operations. Index creation process is also costly and its cost is proportional to document size; but it pays off in the long run as this cost is incurred only once and we use these indexes over and over again. But for smaller documents (such as when a single-page document, like a web page needs to be indexed), it may still lead to a poor user experience. Therefore we decide to add more power to TAPoR

Operation	**%**
Word Cloud	45.6
List Words	22.0
Concordance	16.3
Collocation	4.6
Co-occurrence	3.1
Pattern Distribution	3.0
Extract Text	3.0
Visual Collocation	1.8
Googlizer	0.6

Table 4.1: Percentage of requests for each operation of the existing service

by having an indexing mechanism for larger documents. This will give a new dimension to it and Digital Humanists can now explore Shakespeare's collective works (containing 901,325 words), for example, in matter of seconds.

This reasoning motivates us to migrate TAPoR to the new paradigm of the Hadoop ecosystem, using (a) Hadoop MapReduce to create the document indexes, (b) HBase to store them, and (c) HDFS to store the raw documents. This will enable TAPoR to analyze not only small web pages (which it does now), but also large files sizing in MBs to GBs. Storing the indexes in HBase enables us to solve the scalability issue where the user can process large documents, and also provides fast random-read performance, which is an essential attribute for a web-based application.

4.2 Schema design

The first step in designing an application in HBase is to come up with a schema. The overall performance of an application in HBase is very much dependent on its schema, which should be designed while considering the application workload. Though there are no hard and fast rules about this process, but there are some useful guidelines like the ones mentioned in section 3.2.3.

We analyzed TAPoR access logs ranging for 3 years and contains 53,769 requests, and profile out the workload. This proved quite helpful especially in case of TAPoR, as it has around 44 recipes in its belly, but we found out that 4 most popular services cover up to 87% of the total requests spanning across the above mentioned timeframe. We prioritized the operations to be implemented in the new service based on the frequency of their use in the old service (Table 4.1).

So, the lesson learnt was even though we are interested in migrating the application to a whole new paradigm, the legacy version is essential for eliciting the requirements for the design of the newer version. It can provide insights like existing usage pattern, which helped us here in schema designing.

Having identified the top 4 recipes, *list words, word cloud, concordance* and *co-occurrence*, that form the major part of TAPoR workload, we decided to focus on these services while designing the schema. Before thinking about the actual schema design however, it is worth looking at these recipes in a bit more detail, mainly from the point of view of their read-access pattern. One important point is that the current TAPoR implementation supports analyzing only one document in one request, but we don't want this limitation in the new version.

1. **List word:** It lists out all the words with their frequency found in the document. One can add a filter list to remove some words from the final output.

2. **Word cloud:** It creates a word cloud for top K words found in the document. Essentially, it uses the List words output and filter out top K and pass it to a third-party cloud building library.

3. **Concordance:** It takes a input word and renders its occurrence in the document with some user defined context (like 5 words or 5 sentences on either side).

4. **Co-occurrence:** It takes two words and a context length (for example, 20 words), and renders the subset of document where the two input words are within the given contextual distance.

Close analysis of these 4 operations reveals that the first two can be pre-computed (during the indexing phase). They will not change as the document is not editable. Concordance can be helped by having a *positional based inverted index*, where we know the byte offsets of target words in the document. It will help in directly seeking to the desired offset rather than sequentially reading the entire file up to that offset. For co-occurrence, we can use the byte offset indexes and then do the computation on these indexes to figure out their proximity. Thereafter, one can read the final offsets. It will help in an approximate solution, which can be further refined while reading the file and generating the final output.

Row Key	CF:byteLoc
doc#1,foo	3123, 4223,#2
doc#1,bar	553,643,5544#3
doc#1,...	....
doc#1,Top100	hello:105, world:56, love:45, blood:40
doc#2,foo	909, 656,6786#3

Table 4.2: Possible TAPoR Schema.

4.2.1 Possible schema options

Our analysis suggests two level of indexes, one at the document level (for list words and word cloud), and other at the document-word level (for concordance and co-occurrence). We initially designed a schema, with each row corresponding to a word, containing as its values the byte-offset locations of the word in the document. To distinguish a word for a document, each such word is prefixed with a auto generated document id. Table 4.2 gives an example of such a schema for a document, with its id as #1. The row key of a document level index in such a schema will be special keyword, like "Top100" for the top 100 words, as shown in the table. Document with Id 2 also has the word *foo* but it is in a separate row, as shown in the row of the table.

This schema meets all the requirements for looking a specific word (one just need to add the document Id as the prefix), and also for the list-words functionality where the *keyword* is given as input. The only drawback with this design is that it results in a tall table as there is one row for each unique word in the document. The reason this is a drawback is due to how HBase stores its data and does its read operation. We have discussed HFile in section 2.1.3.2. This schema results in a large number of rows (a 1 MB document has on average 220,000 words), and it increases the index block size in the HFile. This index block size is directly proportional to the number of rows in the table. While doing a read operation, HBase tends to keep this index block of all HFiles in memory (the reason why HBase has a large memory foot print), and as a result this is not an optimal schema. So even though all our the required functionalities are met by this schema, it is still suboptimal from HBase's perspective.

We finally settled on the schema shown in Table 4.3. It has two *column families*, viz., "bl" and "spl", which stand for "byte location" and "special keywords" respectively. This design has only one row per document, therefore it does not suffer from the HFile index size bottleneck. Each unique word becomes a *column qualifier* in "bl" *column family*, and

Row Key	**bl:foo**	**bl:bar**	**bl:sports**	**spl:Top100**
1	3123, 4223, #2	553, 643, 5544, #3		hello:105, world:56, love:45, blood:40
2			434, 423, 545, 646, #4	games:10, soccer:5, sports:4

Table 4.3: Final TAPoR Schema.

its cell value is the byte offset locations in the document, followed by its frequency. To read the top *K* words, one can simply read the *topK* column qualifier in the "spl" family, and byte offset of a word can be fetch by a *Get* operation on "bl" family with the target word as the column qualifier, viz., "bl:word". Therefore, one need to do a single read operation (transaction) with HBase for any of these operation. Though to form a complete answer for concordance and co-occurrence, we also need read the actual file(s) from HDFS. It is to be noted that while inserting the data in the HBase table, we used large Put objects (with usual size of 10 MB, containing multi column qualifiers). This is done because HBase supports row level transactions, and using large number of small Put objects for a row is not efficient.

4.3 Workflow Design and Overall Architecture

After finalizing the schema design, the next task was to design the workflow of the application in order to be able to use the new back-end infrastructure. There were three important processes that were to be joined together to make it a coherent and complete application:

1. Uploading the document on the distributed file system, viz., HDFS, to make it accessible for other processes. HDFS provides a command line interface for such operations. We wrote a bash script for this purpose.

2. Kickstarting the indexing process, to create positional inverted index and special indexes for *top K*, *dates*, *Acronyms* etc.

3. Implementing the web services for accessing these values at the request of the end user from the front end.

Figure 4.1 shows the conceptual flow of the new indexed-based TAPoR. It starts with an end user uploading a document to be analysed. The front end invokes the application web service and passes the URI of the document. This triggers a bash script to upload

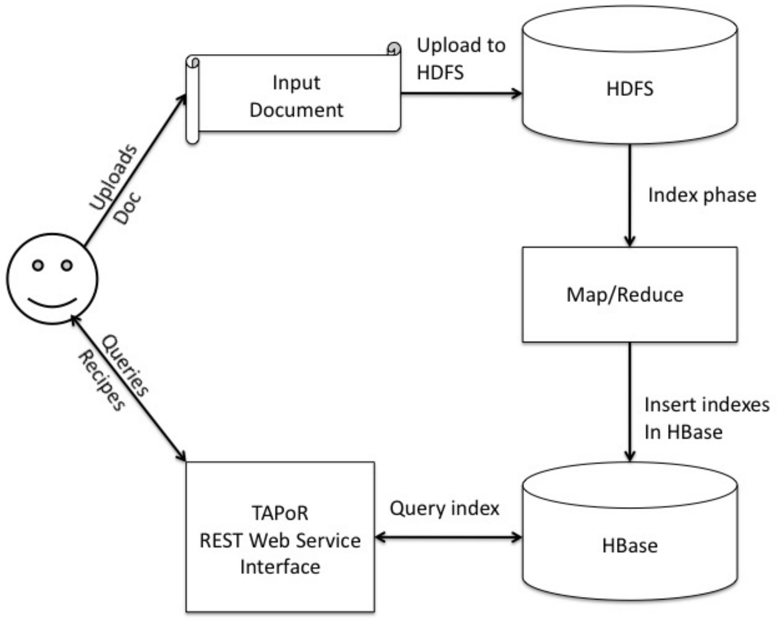


Figure 4.1: Overall workflow of modified Tapor, from document uploading to recipe querying .

the document to Hadoop Distributed File System (HDFS). This is done so that all Hadoop daemons can access the file. Once uploaded, the script kick starts the indexing process, i.e., start the Hadoop mapreduce job to create the indexes. These indexes are inserted in HBase, and the value of document Id is the system time of the *JobTracker* node at the time when the job was started, in nanoseconds. Once HBase is updated, the user can access its data via the TAPoR specific application services, implemented as RESTweb services[30]. These services manipulate the data fetched from the back-end to make it compatible for the front-end. This TAPoR application acts as a client to HBase and accessed indexes and other data via standard REST web services that comes with HBase. Another option of accessing HBase data was to make direct client connection to it. The former method is a standard approach and has some advantages like one is using already instantiated clients (maintained by the HBase Web service application), so it can use the cached locations of table rows, as explained in Section 2.1.3. In later case, we will be creating new TCP connection to HBase for each request. It incurs cost of doing a .META. table lookup for the target rows for all these requests.

Figure 4.2 shows the overall architecture of the new TAPoR application. This gives a top level view of how various software pieces (processes) are running and interacting with each other.

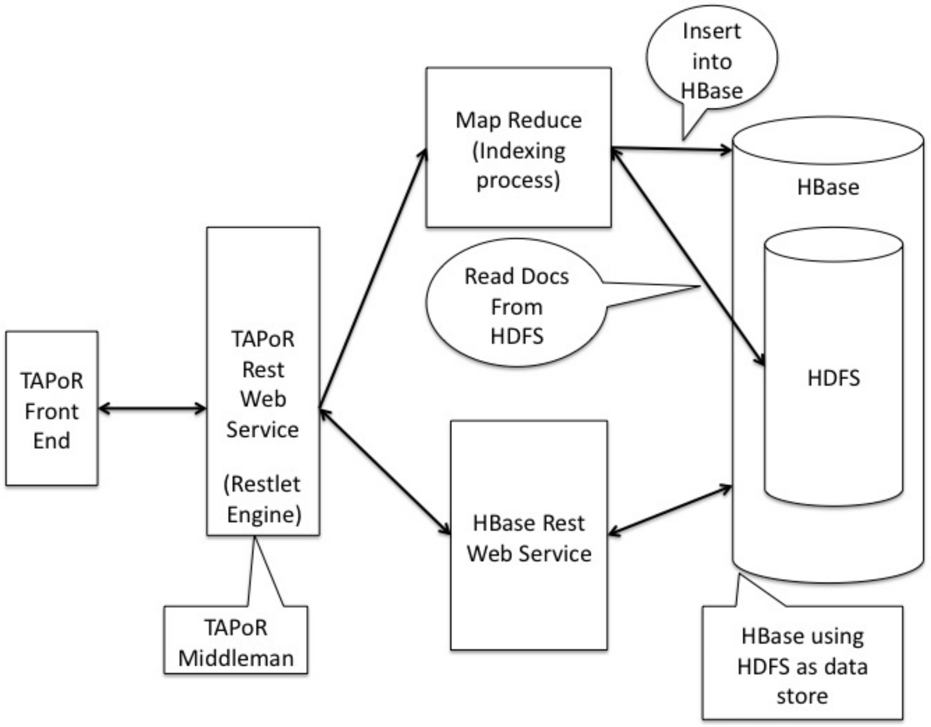

Insert into HBase
Map Reduce (Indexing process)
HBase
Read Docs From HDFS
TAPoR Front End
TAPoR Rest Web Service (Restlet Engine)
HDFS
HBase Rest Web Service
TAPoR Middleman
HBase using HDFS as data store

Figure 4.2: Overall architecture of Indexed based TAPoR application

Now that we have discussed the overall workflow of the application, we discuss each of these sub-components in more detail in the next section.

4.4 Design Discussion of individual components

In this section, we explain the individual components of the migrated TAPoR in more detail. We discuss the index-generation process (Hadoop MapReduce job), a Coprocessor Endpoint use case for computing the top-*K* words and storing them in HBase, and the development of the TAPoR functionalities as Rest web services.

4.4.1 Inverted Index with MapReduce

As per the workflow in section 4.3, our first major use of Hadoop is creating indexes of the newly updated document (step 2 in the enumeration). This is essentially a MapReduce computation.

Figure 4.3 shows the workflow of the indexing phase. We have seen in section 2.1.1 that the input and output for a MapReduce task are key value pairs. The input to the map phase is provided by reading the input file line by line. The *key* of the Map input is the byte

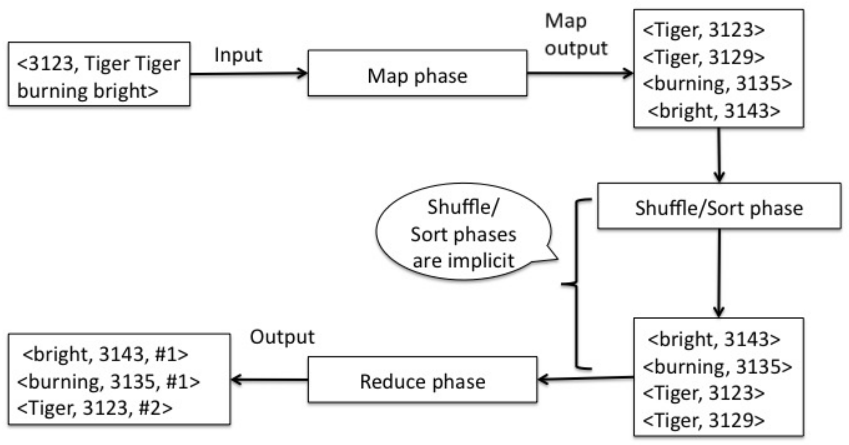


Figure 4.3: Indexing MapReduce job, showing Map, Reduce and the implicit Shuffle phase

offset of the line read from the input file, and *value* is the read line itself.

The figure shows the map phase receiving an input line "Tiger Tiger burning bright" and its offset in the input file is 3123 bytes. The *value* is split in terms of words along with their byte offset, using space as the delimiter, computed by adding cumulative words length to the line-starting offset. This forms the *map* phase output. Before sending this intermediary output to the reduce phase, Hadoop does a shuffle and sort of the data based on the key's value. This is an essential step if we consider the distributed mode this Hadoop works in: A single key goes to only one reducer, and it has all its associated values in a list. The output of this intermediary step shows that *map* output is sorted on keys. Thus the reduce phase appends all the byte offsets for a word, and post append the word frequency to the list. Finally, we have the word, all its byte offset positions and its frequency. This final data structure is used as a cell value in HBase, where it is stored under the column family *bl* and the column qualifier is the word itself.

4.4.2 Frequency index with Coprocessor Endpoint

The "word-level" index, where the byte offset position of each word is recorded, is created in the *reduce* phase. The "document level" index for the top-*K* words is computed after the *reduce* phase using a Coprocessor implementation.

After the reduce phase, we had the frequency of each word (it is appended as the last value in the list of *values* for a *key*. The requirement is to sieve out top K words out of these values and store them separately in a cell value. This could not be done in the indexing *reduce* phase because there can be more than one *reduce* process in one MapReduce job (especially in case of large document), and there is no way to communicate among different *Reduce* processes. Assuming the basic computation for computing top K words remains the same (like storing it in a data structure based on a priority and sieving out top "K" words), there are three ways to compute the frequency index.

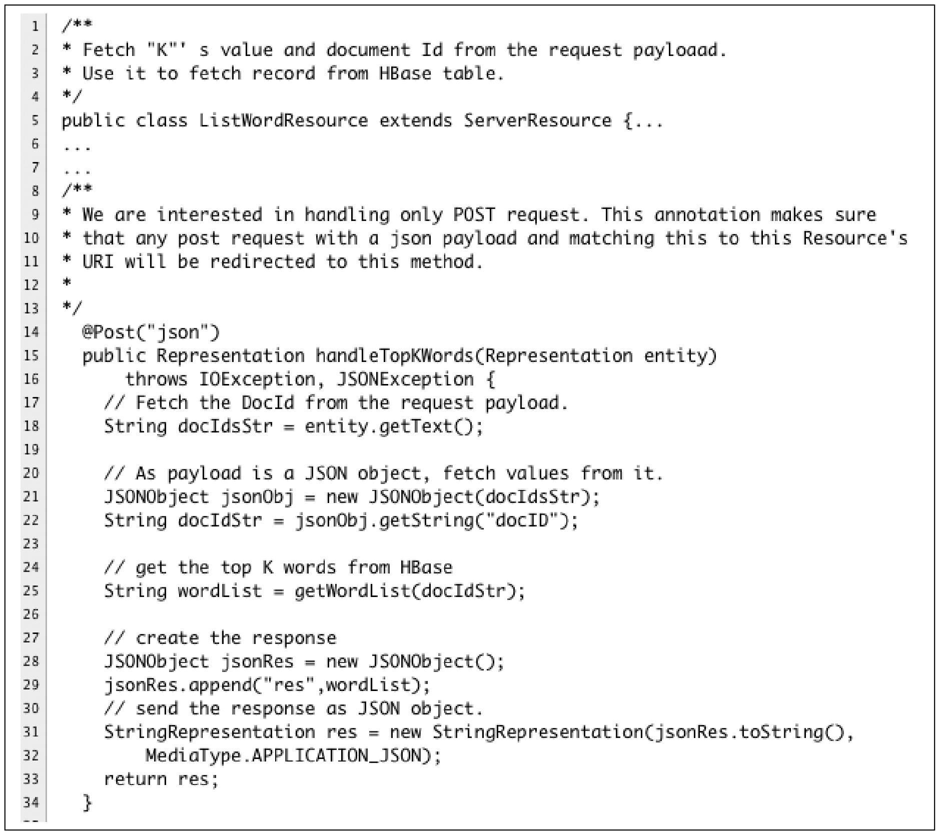
1. Do a *Get* on the document row, and fetch the entire "word level" index from HBase, and do the computation at the client side,
2. Use the MapReduce paradigm to do it on server side, or
3. Use a coprocessor endpoint to go to the Region server node containing that document row, and compute it locally.

Since we are interested in a scalable application, the first option is ruled out since it involves transferring the entire "word-level index" to the client side. Its size is likely to be comparable to the original document size. Running a Hadoop job involves scanning the entire 'docIndex' table, but we are interested only in a single row. This approach will not scale when we upload more and more documents (where each document accounts for one row in the table). Using a Coprocessor endpoint seemed to be the better alternative as it goes to the target row and will does the processing at the server node itself.

We defined an interface *TaporCoprocessor*, which defines a method to create *frequency index* for a given document Id. It uses the Coprocessor endpoint logic that is covered in section 3.1.1. In our experiments with a local 3 node cluster, it takes 3 seconds to create the *frequency index* for a document of size 110MB.

4.4.3 TAPoR middleman: Restlet based Webservices

The index-creation workflow, discussed above, is a one-time activity for any given document. Let us now move to discussing the actual business logic of the application. The end user is more interested in exploring the document in ways as supported by the existing TAPoR application, using various text-analysis recipes as listed in Table 4.1. We explain our TAPoR API, which acts as an interface for front end applications. This *middleman* is a REST web service based application that provides signature for all required functionalities, from uploading a document to actually running the recipes [30].



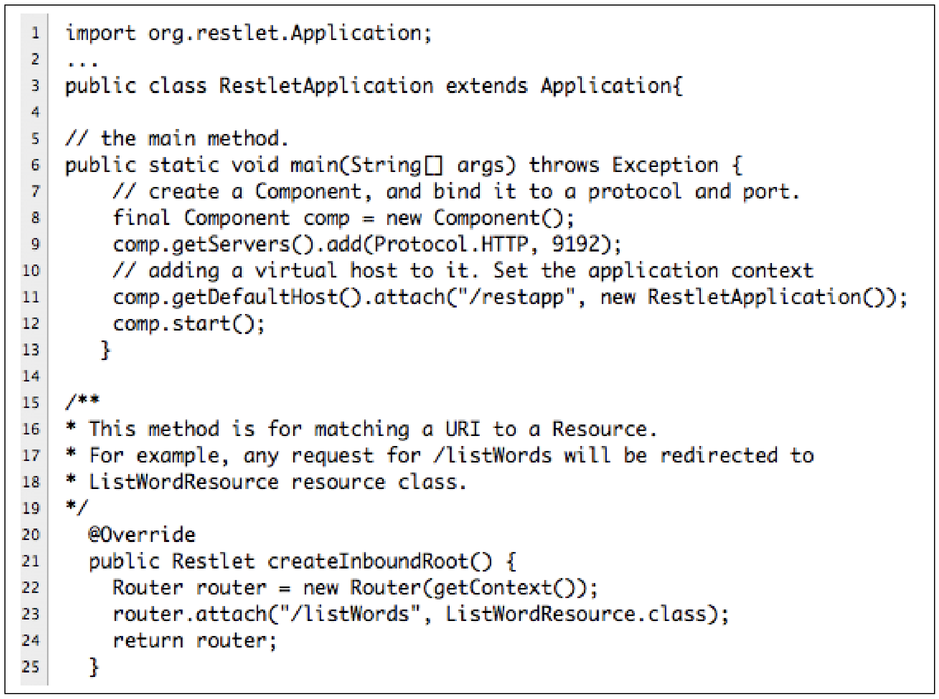
```
 1 /**
 2 * Fetch "K"' s value and document Id from the request payloaad.
 3 * Use it to fetch record from HBase table.
 4 */
 5 public class ListWordResource extends ServerResource {...
 6 ...
 7 ...
 8 /**
 9 * We are interested in handling only POST request. This annotation makes sure
10 * that any post request with a json payload and matching this to this Resource's
11 * URI will be redirected to this method.
12 *
13 */
14   @Post("json")
15   public Representation handleTopKWords(Representation entity)
16       throws IOException, JSONException {
17     // Fetch the DocId from the request payload.
18     String docIdsStr = entity.getText();
19
20     // As payload is a JSON object, fetch values from it.
21     JSONObject jsonObj = new JSONObject(docIdsStr);
22     String docIdStr = jsonObj.getString("docID");
23
24     // get the top K words from HBase
25     String wordList = getWordList(docIdStr);
26
27     // create the response
28     JSONObject jsonRes = new JSONObject();
29     jsonRes.append("res",wordList);
30     // send the response as JSON object.
31     StringRepresentation res = new StringRepresentation(jsonRes.toString(),
32         MediaType.APPLICATION_JSON);
33     return res;
34   }
```

Figure 4.4: ListWord Resource

Since we had to create a whole new backend, we had the opportunity of re-thinking about the web-service interface. We could have used the existing SOAP based services signature, but decided to use REST services for two reasons. First, we already are using REST web services to interact with HBase (comes as a part of HBase), and secondly, we do not need the SOAP specific advantage such as web security and transactions. Also, using REST is much simpler programmatically.

We developed the TAPoR REST web services using Restlet, an open source Servlet API based Rest service provider. It is a framework that provides a API supporting Rest concepts[30], and a Restlet Engine that provides an implementation of the API. This can be correlated to as having standard JDBC API and concrete JDBC drivers.

Restlet provides APIs to follow the Rest principle of Resources. In our case, we expose text analysis recipes as resources. For example, we have created a ListResource class for the List words functionality. Figure 4.4 shows an interesting code fragment, exposing a functionality as a Restlet resource. One needs to extend the class *org.restlet.resource.ServerResource* to expose an entity as a REST resource (as shown in line 5). Next, one should provide the



```
1   import org.restlet.Application;
2   ...
3   public class RestletApplication extends Application{
4
5   // the main method.
6   public static void main(String[] args) throws Exception {
7       // create a Component, and bind it to a protocol and port.
8       final Component comp = new Component();
9       comp.getServers().add(Protocol.HTTP, 9192);
10      // adding a virtual host to it. Set the application context
11      comp.getDefaultHost().attach("/restapp", new RestletApplication());
12      comp.start();
13     }
14
15  /**
16  * This method is for matching a URI to a Resource.
17  * For example, any request for /listWords will be redirected to
18  * ListWordResource resource class.
19  */
20    @Override
21    public Restlet createInboundRoot() {
22      Router router = new Router(getContext());
23      router.attach("/listWords", ListWordResource.class);
24      return router;
25    }
```

Figure 4.5: Main class of TAPoR Rest application

implementation of the HTTP request methods POST/GET/DELETE/PUT one wants to support. We design all TAPoR web services to accept POST requests. Annotation at line number 14 suggest that the framework will invoke the *handleTopKWords* method when a POST request is submitted to a URI matching to the ListResource resource. Match a Resource to a URI is done in the main class of the application, as shown in Figure 4.5. The figure also shows an important line of code in order to define an application in Restlet. This driver class should extends the class *org.restlet.Application* as shown in lines 1-3. The Resource to URI mapping is done in the *createInboundRoot* method as shown in lines 20-25 where we match a URI "/listWords" to the ListWordResource resource class. The application context and its port binding is done as shown in lines 8-12. In summary, a HTTP POST request of the form "http://hostname:9192/restapp/listWords" will be redirected towards the *handleTopKWords*() method of *ListWordResource* class. We created 3 such resources for List words, Concordance and Co-occurrence functionalities.

4.4.4 TAPoR APIs

Here we list the signature of the three Restful APIs that we implemented as part of TAPoR migration on to HBase. The request and response are shared in JSON format.

1. List words. URI: /restapp/listWords/

The request payload should have the following parameters:

- docID of the document.
- resultSetSize: an int value, multiple of 100 (like 2 for top 200 words)

The response will be a JSON object, with following three fields

- status: 0 or 1. 0 for success, 1 for failure
- result list: sorted array of words with their frequency.
- message: Error message (in case of any error, else blank).

The following is a sample response object:

{ “status”:“0”, “list”:[“foo:100, bar:90, sacred: 25, remembered: 25”], “message”:“” }

2. Concordance. URI: /restapp/concordance/

   The request payload should have the following parameters:

   - docID of the document.
   - word: the target word whose concordance has to be computed
   - resultSetSize: maximum number of instances (byte offsets) of the words are to be send in the response.

   The response will be a JSON object, with the following three fields

   - status: 0 or 1. 0 for success, 1 for failure
   - result list: an array containing the read concordance from the document.
   - message: Error message (in case of any error, else blank).

   The following is a sample response object:

   { “status”:“0”, “concordance”:[“ you; your enemies be silenced;”, “life;true the burning spirit of love eternity.”], “message”:“” }

3. FileUpload. URI: /restapp/fileUpload/

   The request payload should have the following parameters:

   - filePath: URI of the file where it is to be fetched and uploaded to HDFS

The response will be a JSON object, with following three fields

- status: 0 or 1. 0 for success, 1 for failure
- documentId: an int value which is equal to document Id in HBase as explained in section 4.3.
- message: Error message (in case of any error, else blank).

The following is a sample response object:

{ “status”:“0”, “docID”:“12342343”, “message”:“” }

4.5 Summary

Data-intensive applications, such as social networking sites like Facebook, twitter, necessitate the development of more scalable options than relational databases. NoSQL or Not only SQL databases or so called Cloud databases, with their scalability, robustness and better performance present a good alternative for some kind of workloads. These databases do not include complex, multi-transaction oriented financial institutions; rather a more unstructured, single row transaction workloads like the one mentioned at the beginning. These cloud databases are more appropriate for applications that produce tremendous amount of data at a high rate, like Facebook which was generating 60 terabytes every week [31] in 2009. Such workloads presents an exciting use case of these NoSQL databases. And one can now leverage cloud computing to spin a cluster with in a matter of few minutes and at an economical price.

In this area, there are many options such as HBase, Cassandra, MongoDB, VoltDB to name a few. HBase is increasingly becoming more popular and its recent adoption by Cassandra’s inventor Facebook for its inbox messages [18], yfrog as its image store[32] and Twitter [33] has given it a major boost.

This chapter discussed HBase, its design, the extensions we developed for it and the process of using it for a web based application. We discussed the use case of Coprocessor Endpoint while creating the Frequency index of the document by reading the *word index* at the server side. We can also extend the TAPoR functionality by analyzing multi documents in one request. This can be done by using a Coprocessor Endpoint to process the document at the server side (for example, finding common top K words in two documents). This supports our claim about Coprocessors usability in HBase for many real world applications. We believe that the lesson we learned will be useful in migration of other applications.

Chapter 5

Conclusion

We are seeing an unprecedented growth of data in the last decade, primarily unstructured. This has led researchers and practitioners to look for new alternatives to traditional RDBMs. Cloud databases (also called NoSQL databases) are designed to address the scalability issues, as they can be configured to run on computational clusters in the order of 100s of nodes. The fundamental shortcoming of these databases, however, is that they lack the support that software developers have grown accustomed to expect from traditional RDBMs, such as a structured query language, stored procedures and active triggers, and support for complex data types. This current limitation has motivated an active field of research focused on exploring novel ways to improve software-development support around cloud databases. In this dissertation, we focused on HBase and extended its newly developed Coprocessor framework to improve the current query-execution mechanism.

5.1 Contributions

This thesis makes the following contributions.

- We have designed and implemented standard aggregate functions, such as row-count, max, min and others, using existing Coprocessors infrastructure. These aggregates are useful features and all existing relation database systems provide them as inbuilt functions. In HBase, these functions provide a better alternative than the existing approaches of running a sequential Scan or a MapReduce job. This work has been accepted by HBase team and committed to the main trunk.

- We have designed a cursor framework that provides support for streaming results from the Coprocessor endpoints. With the cursor framework, a coprocessor can process a *Region* in an incremental way. It maintains the client request state (in a cursor) and the client can invoke its *next()* API to incrementally receive results. We demonstrated that it gives better response time when there is some big computation that can be transferred to the server-side, as mentioned in Section 3.3. This work has been submitted to the HBase team and is up for review.

- The cursor framework can also be used to create a parallel scanner infrastructure. The idea is it can use the parallelism inherently provided by the Coprocessor framework, and one can stream in results in parallel. We used its variation while executing **Query 4** on the NGram dataset, where results from 12 *Regions* are fetched in parallel. This is a much desired functionality in the current HBase [34].

- Finally, we investigated the issues around migrating an existing application to HBase. We migrated TAPoR, a text analysis application which suffered from poor scalability, onto HBase. We shared our experience and believe that lessons learnt such as designing a schema, using HBase REST service (to utilize connection pooling) as compared to creating a new connection per each request, can be used for migrating other web-based applications.

Our experiments validate the streaming results extension for Coprocessors and demonstrate that it can substantially improve performance in a variety of types of data-access patterns such as TAPoR.

5.2 Future Work

The work of this thesis consists of two themes: at the systems' level, it extends the HBase Coprocessors framework; at the application level, with the TAPoR migration, it demonstrates the relevance of the HBase technology to the task of migrating and scaling up existing web-based applications.

In the context of the first theme, we plan to continue working on the result-streaming support in order to make it robust enough to be incorporated in Apache HBase. The Coprocessor framework will greatly enhance the next major HBase release (version 0.92) in near future and we intend to make the result-streaming functionality an important part of it. In addition to support a variety of data-access use cases, at the very least, it will help eliminate the timeout exception in current aggregate functions for large table sets. We can also use it for providing a parallel scanner functionality, where a Coprocessor can simply return batch of raw rows from server, in parallel. This will be a useful feature as compared to the existing *sequential* Scan API.

In the context of the TAPoR application, we plan to add more functionality such as comparing a collection of documents in one request. This require a TAPoR specific Coprocessor to perform the computation at the server side and send the intermediate results to the client. This way, users will be able to process large documents without transferring them to the client.

Bibliography

[1] (2011). [Online]. Available: http://ngrams.googlelabs.com/datasets

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] (2011). [Online]. Available: http://hadoop.apache.org/common/docs/current/hdfs_design.html/

[4] (2010). [Online]. Available: http://gigaom.com/cloud/sensor-networks-top-social-networks-for-big-data-2/

[5] (2011). [Online]. Available: http://wiki.apache.org/hadoop/Hbase/PoweredBy

[6] (2011). [Online]. Available: http://aws.amazon.com/ec2/

[7] (2011). [Online]. Available: http://wiki.apache.org/hadoop/

[8] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[9] S. Ghemawat, H. Gobioff, and S. Leung, "The google file system," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[10] (2011). [Online]. Available: www.odbms.org/download/dean-keynote-ladis2009.pdf

[11] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of the 35th SIGMOD international conference on Management of data*. ACM, 2009, pp. 165–178.

[12] (2011). [Online]. Available: http://www.cs.yale.edu/people/abadi.html

[13] (2011). [Online]. Available: http://www.hadapt.com/

[14] M. Stonebraker and R. Cattell, "10 rules for scalable performance in'simple operation'datastores," *Communications of the ACM*, vol. 54, no. 6, pp. 72–80, 2011.

[15] (2011). [Online]. Available: http://www.linuxjournal.com/content/the-large-hadron-collider

[16] (2011). [Online]. Available: http://www.mendeley.com/

[17] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[18] (2011). [Online]. Available: http://www.facebook.com/note.php?note_id=454991608919

[19] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[20] Y. Shi, X. Meng, J. Zhao, X. Hu, B. Liu, and H. Wang, "Benchmarking cloud-based data management systems," in *Proceedings of the second international workshop on Cloud data management*. ACM, 2010, pp. 47–54.

[21] C. Zhang and H. De Sterck, "Supporting multi-row distributed transactions with global snapshot isolation using bare-bones hbase," *Proc. of Grid2010*, 2010.

[22] I. Konstantinou, E. Angelou, D. Tsoumakos, and N. Koziris, "Distributed indexing of web scale datasets for the cloud," in *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*. ACM, 2010, pp. 1–6.

[23] N. Li, J. Rao, E. Shekita, and S. Tata, "Leveraging a scalable row store to build a distributed text index," in *Proceeding of the first international workshop on Cloud data management*. ACM, 2009, pp. 29–36.

[24] (2011). [Online]. Available: http://berlinbuzzwords.de/

[25] (2011). [Online]. Available: http://www.slideshare.net/ghelmling/new-hbase-features-coprocessors-and-security

[26] H. Vashishtha and E. Stroulia, "Enhancing query support in hbase via an extended coprocessors framework," *ServicesWave*, 2011, "To appear".

[27] J. Michel, Y. Shen, A. Aiden, A. Veres, M. Gray, J. Pickett, D. Hoiberg, D. Clancy, P. Norvig, J. Orwant *et al.*, "Quantitative analysis of culture using millions of digitized books," *Science*, vol. 331, no. 6014, p. 176, 2011.

[28] H. Vashishtha, M. Smit, and E. Stroulia, "Moving text analysis tools to the cloud," in *2010 6th World Congress on Services*. IEEE, 2010, pp. 107–114.

[29] (2011). [Online]. Available: http://taporware.mcmaster.ca/~taporware/textTools/

[30] R. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, Citeseer, 2000.

[31] D. Beaver, S. Kumar, H. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: Facebooks photo storage," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 1–8.

[32] (2011). [Online]. Available: http://nosql.mypopescu.com/post/7794033125/hbase-at-yfrog

[33] (2011). [Online]. Available: http://blog.muehlburger.at/2010/05/06/twitters-use-of-cassandra-pig-and-hbase-for-highly-distributed-data-processing-and-analysis/

[34] (2011). [Online]. Available: https://issues.apache.org/jira/browse/Hbase-1935