Secondary indexing for the HBase distributed database

Roger Ruiz-Carrillo

School of Computer Science

McGill University, Montréal

July 2013

A thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements of the degree of

Master of Science in Computer Science

Abstract

Most traditional database systems offer secondary indexing to increase the performance of data reads during query execution. With the advent of newer, NoSQL distributed datastores, some don't yet have a secondary indexing feature built in. HBase is one such distributed database system. It partitions tables in so-called table regions across many nodes but does not offer secondary indexing. Since some types of queries may benefit from secondary indexing in order to improve their performance, we endeavoured to implement such secondary indexing for HBase. In this thesis, we present and compare two coprocessor based secondary indexing implementations.

The first implementation, *Table Based Secondary Indexing*, leverages HBase tables to store the secondary indices. Query processing in our first implementation is split between the client and the servers.

Our second implementation, *In Memory Secondary Indexing*, relies on indices that are partitioned and are co-located in the main memory of the table regions they index. Query processing in our second implementation is executed solely at the server level.

Our experimental results show that both our implementations offer a substantial gain in read performance over the default HBase method of

executing queries, which scans all table regions when executing read queries.

Résumé

La majorité des systèmes de bases de données traditionnels ont un mécanisme d'indexation secondaire qui offre une performance accrue lors de requêtes en lecture. Du au développement récent des systèmes de base de données distribués, certains n'ont pas encore de mécanisme d'indexation secondaire. HBase est un de ces systèmes qui n'offre pas d'index secondaires. Dû au fait que la performance de certains types de requêtes peuvent bénéficier grandement des index secondaires, nous nous sommes engagés à créer ce mécanisme pour HBase. Dans cette thèse, nous faisons la présentation et la comparaison de deux solutions que nous avons créées qui basées sur l'utilisation des coprocesseurs de HBase.

La première solution repose sur l'utilisation des tables offertes par HBase pour emmagasiner les index secondaires. Dans notre première solution, l'exécution de requêtes utilisant un index secondaire est partagée entre le client et le serveur.

La seconde solution se base sur le fait que les index secondaire sont partitionnés pour correspondre aux régions de tables et sont co-localisées avec celles-ci. De plus, les index secondaires résident dans la mémoire active des serveurs. Dans notre deuxième solution, l'exécution de

requêtes utilisant un index secondaire est faite uniquement par les serveurs.

Nos résultats expérimentaux montrent que nos deux solutions offrent un gain de performance substantiel lors d'exécutions de requêtes en lecture comparé à la méthode native à HBase.

Acknowledgments

I would first and foremost like to address special thanks to my thesis supervisor Professor Bettina Kemme for her support, suggestions and attentive guidance. Her knowledge of database and distributed systems was invaluable to me for the achievement of this thesis.

I would also like to thank M. Yousuf Ahmed for sharing his knowledge of some of the more subtle HBase internals as well as for all the insightful discussions we had.

Lastly, I am deeply grateful to my wife for her love, patience and continuous support that provided me with the encouragement I needed to complete my thesis.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

More and more data is being collected, stored and used every day by companies on their customers, by bioinformatics applications [1], by sensor networks [2], by analytical engines [3], social network sites [4], etc.; this is known as Big Data [5]. Many of the traditional single machine or small cluster relational database management systems are not well suited to store and exploit that Big Data efficiently because they can't scale well enough to handle the terabytes and even petabytes [6] [7] of data. Distributed database are designed to scale horizontally: the answer to more data to store and process is simply to add more nodes to the distributed database. For that reason, distributed database systems are gaining in popularity and are being developed to take advantage of the parallel processing and expanded storage multiple computers offer. One well known example taken from the industry is Google's. They developed their own

distributed database system to accommodate their massive amount data used for web indexing, Google Earth and other services.

Some distributed database systems are proprietary, such as Google's BigTable [8] and Amazon's DynamoDB [9]. Some others are commercially available, for example: Objectivity/DB[1] and ObjectStore[2]. Finally some of the best-known open source examples are HBase [10] and Cassandra [11].

Most open source distributed databases that offer a tabular view of the stored data offer secondary indexing in order to speed up predicate queries that look for a specific subset of rows depending on the value of a secondary attribute, that is, for queries that do not look for rows depending on their row key. HBase does not support secondary indexing, and we think that such a feature would be beneficial to some applications and as such, we decided to implement that feature. We decided to implement secondary indexing in HBase by using HBase's coprocessor framework (described in Section 2.1.5) in order to decouple our implementation as much as possible from the core HBase development which is in constant evolution. This allows our implemented libraries to be treated as add-ons that can be simply plugged into an existing HBase environment and start

[1] http://www.objectivity.com/products/objectivitydb/

[2] http://www.objectstore.com/

being used without having to re-install an entire new HBase version; this also allows for upgrades of our implementations to be much easier to conduct.

Our first implementation, *Table Based Secondary Indexing*, leverages standard HBase tables and their associated properties to store the secondary indices; as such, index tables may be split and distributed according to HBase's internal splitting and balancing algorithms, and are thus not necessarily co-located with the data they index. Query processing in our first implementation is split between the client and the server. First, from a user query Q, our extended HBase client executes a first query to the server(s) hosting the index table to retrieve the location (rows keys) of the rows matching the user query Q's predicate. The client then makes a second query to the real table requesting exactly the rows that have the row keys determined in the first query. These rows build the result set of query Q.

Our second implementation, *In Memory Secondary Indexing*, relies on indices that are partitioned and are co-located in the main memory of the table regions they index. Query processing in our second implementation is executed solely at the server level: A user query Q will be broadcast to all the servers hosting a given table and each server will process the query, looking up their local indices,

and then return the portion of the entire result set corresponding to the data they host.

1.1 Contribution

We offer the following contributions with this thesis:

- We design and implement two secondary indexing libraries for the HBase distributed database, built with the available HBase coprocessor framework: Table Based Secondary Indexing and In Memory Secondary Indexing.

- We provide an extensive evaluation of our strategies and comparison between them.

1.2 Thesis Outline

Our thesis is organized into chapters as follows:

- Chapter 2 covers the background information required for the understanding of our work. In particular, it describes the HBase distributed database and its architecture.
- Chapter 3 describes our first implementation for secondary indices within HBase: Table Based Secondary Indexing.

- Chapter 4 describes our second implementation for secondary indices within HBase: In Memory Secondary Indexing.
- In Chapter 5, the performance evaluation of both secondary indexing implementations is discussed.
- Chapter 6 covers the conclusions that are drawn from and proposes possible enhancements to the implementations and other avenues to be explored for secondary indexing within HBase.

Chapter 2

Background and Related Work

2.1 Apache HBase

2.1.1 Overview

HBase [10] [12] is a distributed database which has been designed based on Google's BigTable [8], a distributed storage system for structured data. It has the same goal: to provide storage and retrieval functionality for Big Data. It uses Apache Hadoop[3] and the Hadoop Distributed File System (HDFS) to store its data files; HDFS provides high availability for the data files via built-in replication. It is an Apache Software Foundation open-source project programmed entirely using the Java programming language.

[3] http://hadoop.apache.org/

HBase is part of the NoSQL database family. NoSQL databases do not follow the relational model and as such do not support the SQL language. Other examples of NoSQL distributed databases are Apache Cassandra [11], MongoDB[4] and Amazon's DynamoDB [9].

The following sections provide a quick overview of traditional relational database management systems (RDBMS) and secondary indexing followed by HBase's data format, HBase's architecture overview and a detailed look at HBase's coprocessors.

2.1.2 Relational Database Management Systems and Secondary Indexing

2.1.2.1 Data Model

RDBMS are based on the relational model proposed by E.F. Codd [13]. Some examples of RDBMS are IBM DB2, Oracle RDBMS, MySQL, Microsoft SQL Server, Terradata, etc.

The basis of the RDMBS's data model is the relation which is comprised of a relation schema and a relation instance. The relation schema is the definition of a table (e.g. column names and domains for the data stored in the columns). The data stored in a table is called the relation instance. Figure 2.1 shows the tabular

[4] http://www.mongodb.org/

view of a simple relation (Employee) used to store information (employee ID, social security number and name) about the employees of a company.

Employee		
ID : integer	SocialSecurityNumber : integer	FullName : string
112473	555123789	John Smith
210791	555145217	John Doe
...	...	...

Figure 2.1: Example of a relation

In addition to the relation, RDBMS offer *Constraints* that are essentially conditions that data must satisfy in order to be stored in a table. There are different types of constraints such as integrity constraints and key constraints. Integrity constraints are defined to ensure that data stored in a table is coherent. In the example shown in Figure 2.1, we would want the social security numbers to be unique (e.g. no two employees can have the same one). In such a case, a constraint of type UNIQUE would be applied to the social security number column. Key constraints are defined as the minimal subset of fields of a table that identify a row uniquely. In Figure 2.1, the primary key would be the ID column since it is used to uniquely identify within the Employee relation. Keys are used to link different tables together. For example, in Figure 2.2 below, we extend the database containing the employees of a company from Figure 2.1 to add information about which office the employees work in. To do so, we add an Office

table containing the list of offices for the company comprised of an office identifier, an office name and the office address; the primary key of the office table is the ID column. We then link the employees to their offices, by entering the office table's key in a new column (OfficeID) within the Employee table. When a table's primary key is used in a table other than the one it is created to define tuple uniqueness, it is referred to as a foreign key. A foreign key, unlike a primary key does not need to be unique. In Figure 2.2, OfficeID is defined as a foreign key referencing the primary key (ID) of the Office table. In the example of Figure 2.2, both John and Jane Doe work in the Uptown office and John Smith works in the Downtown office.

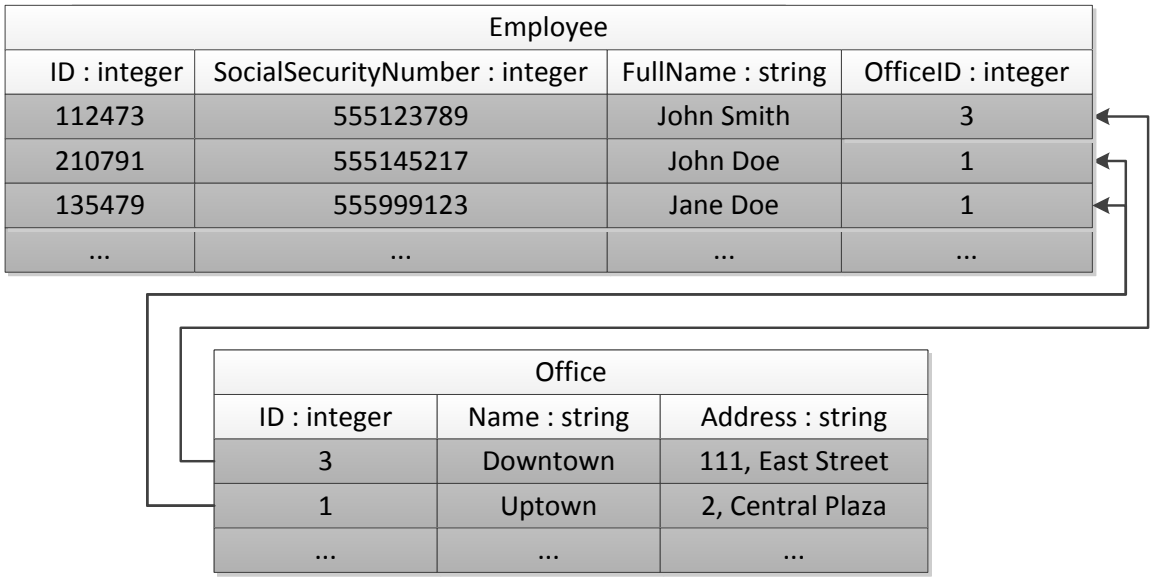

Figure 2.2: Linking relations with keys

Data stored in an RDBMS is manipulated and accessed by using a Data Manipulation Language (DML) and tables are created and modified by using the

Data Definition Language (DDL). Both DDL and DML are subsets of the Structured Query Language (SQL) standard.

2.1.2.1 Indexing and Secondary Indexing

When tables become very large, looking up specific records (rows) becomes a costly endeavor. The naïve way of performing a search for a given, unique record (search on primary key) is to go through each record in the table until the record we're looking for is found. In the worst case, where the record is the last of the table, the entire table has to be searched. An even worse case scenario is to search for all the records of a table that have a specific, non-unique value in one of their columns or even ask for a range of attribute values in such a secondary column (predicate query); this kind of search must always examine the entire table in order to return the matching records.

To provide increased performance when looking up records in a table, RDBMS implement indexing services that pinpoint where the queried data resides within the entire table data. Lookups that use indices can have a performance increase orders of magnitude faster than their non-indexed counterparts. Primary keys of tables are always indexed; their index and any index on a set of columns that includes the primary key are called a primary index. Any index, other than a primary index, is called secondary index. For example, from Figure 2.2, if

employees are often looked up by their name instead of their employee id, then creating a secondary index on the column “FullName” would increase the search performance. Indices can be stored in different kinds of data structures depending on the type of queries they will be used for. For example, B-Trees work best for range queries (e.g., “Retrieve all employees aged between 25 and 35”) but also work fairly well for equality queries (e.g., “Retrieve all employees working in the Downtown office”). Hash based indexes, on the other hand, are designed to work only for equality and inequality queries, but provide very fast lookups.

2.1.3 HBase Table Format

Data in HBase is stored into and accessed from tables; rows are defined by row key (primary key) and contain data in multiple columns. HBase has the particularity that the columns of each table are grouped by column family and within each column family, a set of qualifiers further specifies the column name where given data pertains; a column is the combination of column family and qualifier separated by a colon (:). The purpose behind this column scheme is to store the data pertaining to separate column families into separate files on the file system. The reasoning behind this column scheme is that usually, when data is retrieved for a row, the user only requires a subset of all the columns as opposed

to all the columns and as such, the data retrieval operation is optimized by organizing the storage of data in this way.

Figure 2.3 shows an example of how data is organized in an HBase table. For example, to retrieve the counter update data of row1, the column *counters:updates* must be accessed.

Row Key	Time Stamp	Column "data:"	Column "meta:" "mimetype"	"size"	Column "counters:" "updates"
"row1"	t_3	"{ "name" : "lars", "address" : ...}"		"2323"	"1"
	t_6	"{ "name" : "lars", "address" : ...}"			"2"
	t_8		"application/json"		
	t_9	"{ "name" : "lars", "address" : ...}"			"3"

Figure 2.3: HBase table example rendered as a spreadsheet [12]

Each of the tables in HBase (with the exception of the -ROOT- system table) may be partitioned into multiple regions; the partitioning criterion is the range of the row key and partitioning is triggered by region size. Each of a table's regions is served by a separate Region Server.

HBase allows queries based on row key but also predicate queries. A particularity of HBase is that only the column containing the row key is indexed. As such, queries based on row key values execute very quickly whereas queries based on the values contained in any other column suffer a performance penalty because they require a complete table scan to complete. HBase allows queries

based on values contained in columns other than the row key column by using a filtered *Scan* operation. Each Region Server will sequentially scan its entire table region and check each row against the filter on the values of the secondary column(s) so as to return only the matching rows to the client.

2.1.4 HBase Architecture Overview

HBase has five main components: Region Servers, a Master Server, an Apache Zookeeper [14] cluster, a client library and Hadoop. The architecture is depicted in Figure 2.4.

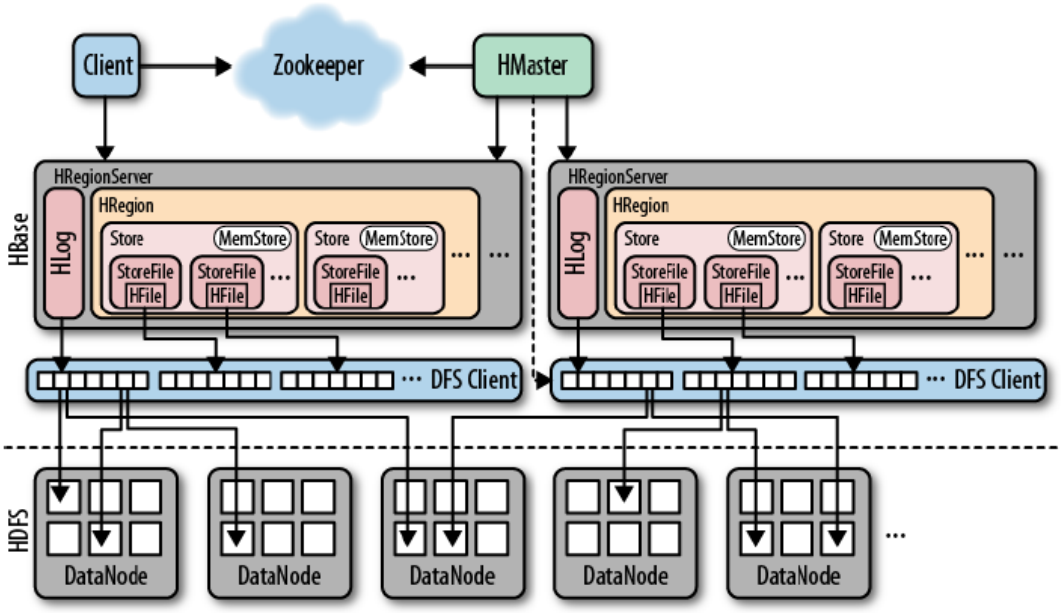

Figure 2.4: HBase architecture overview [12]

Region Servers (HRegionServer in Figure 2.4) handle all the read and write operations to the table regions (HRegion in Figure 2.4) they are assigned. The most frequently accessed data of each region is stored in main memory (MemStore in Figure 2.4). Data is persisted into HDFS data nodes (DataNode in Figure 2.4). In order to increase performance, updates are written to the MemStore and Write-Ahead-Log (HLog in Figure 2.4) but only the Write-Ahead-Log is flushed to HDFS during execution. HBase favors consistency over availability and, as such, a given region is associated with only one Region Server at a time. The Write-Ahead-Log is only used in case of recovery after a region server crashes; it contains all updates to the regions, located within their volatile main memory, which have not yet been flushed to HDFS. Each Region Server instance runs on a separate node; although not required, Region Servers and Hadoop DataNodes will be collocated on the same nodes in order to provide data locality for the region's data files which increases performance.

The Master Server (HMaster in Figure 2.4)'s task is to assign regions to Region Servers and to perform load balancing by shuffling regions around Region Servers in order to spread the load as evenly as possible. The Master Server will also take charge of the metadata associated with table creation, deletion and table alteration (creation or deletion of column families within a given table).

Zookeeper is another Apache project based on the paper “ZooKeeper: Wait-free coordination for Internet-scale systems” [14]. It provides a coordination service in a distributed environment. It is typically configured as a cluster and it maintains a list of the locations of the active Region Servers and the Master Server by way of a heartbeat mechanism. This ensures that there is only one Master Server running; a standby Master Server process exists but it will not take over as long as there is already an active one registered within Zookeeper. Zookeeper also allows the Master Server to detect the failure of a Region Server and assign the regions of the failed Region Server to active Region Servers; in such a case, the failed regions would not be available until their newly assigned Region Servers had opened them.

Finally, the HBase client library offers an API to manage tables and read / write to tables.

2.1.5 HBase Coprocessors

HBase’s Coprocessor framework allows for custom code to be executed on the server side. There are two main categories of coprocessors: observers and endpoints.

2.1.5.1 Observers

The observers are analogous to triggers within traditional relational databases systems. Custom code will execute when certain operations are done by the server, be it in response to a client call (e.g. deletion of a row) or due to the normal server lifecycle (e.g. region compaction).

There are three types of observers: The WAL Observer, the Master Observer and the Region Observer.

The WAL Observer provides hooks for custom code during processing of the Write-Ahead-Log.

The Master Observer allows for the execution of custom code during operations managed by the Master Server such as regions moves, DDL operations on tables and load balancing.

Finally, the Region Observer provides hooks into Region Server operations such as region splitting, DML operations on region data, flushing of data to Hadoop, etc.

The coprocessors within the observer category have the particularity that they are chained together. For example, a first coprocessor could be in charge of security and either allow or deny a client's request on a region while a second coprocessor could do something else once the security check has succeeded, for

example, maintain an index on a secondary column when data is inserted into the region.

Figure 2.5 below illustrates the chaining of Region Observer coprocessors

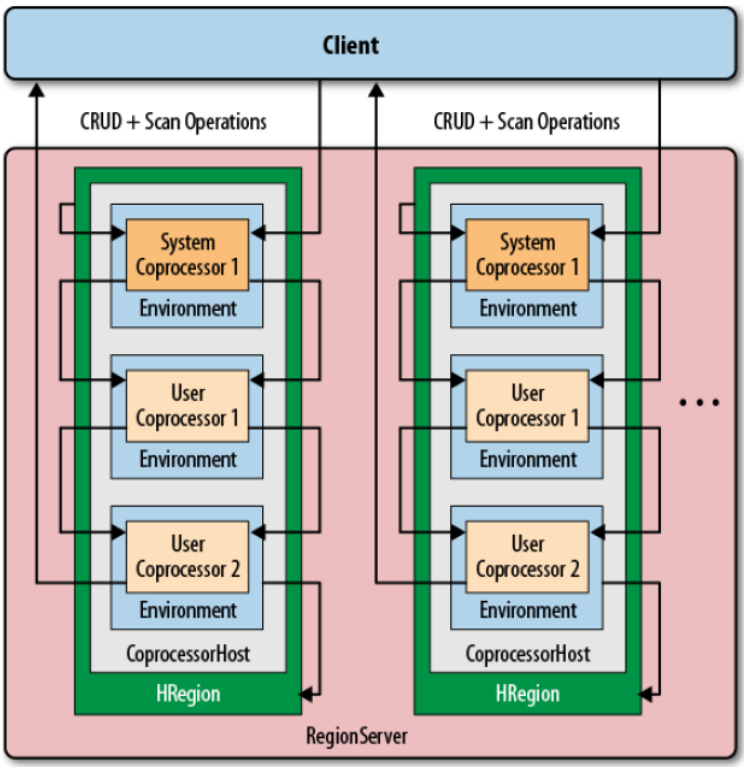

Figure 2.5: Coprocessor chaining – Region Observers [12]

2.1.5.2 Endpoints

The Endpoint coprocessors are analogous to stored procedures within traditional relational databases systems. Custom code will execute when directly invoked by the client via a remote procedure call.

2.1.6 HBase Version

We built our two secondary indexing implementations against the latest alpha version of HBase (0.95.0). The motivation behind using an alpha version is that our in memory implementation uses a special type of coprocessor, the endpoint coprocessor, which requires the client to invoke methods via an RPC protocol. This RPC protocol changes radically and breaks compatibility from the latest stable version (0.94) and the future versions of HBase. Versions 0.94 and earlier use a proprietary RPC protocol and the future versions use Google's Protocol Buffers to manage the RPCs. We wanted our implementations to work with the newest and soon to be released versions of HBase. This decision did cause a few problems along the way; one of them having to do with lower performance after a region splits.

2.2 Related Work

When researching how we could implement a secondary indexing system for HBase, we found that different distributed database systems, Apache Cassandra and MongoDB, used different methods for their secondary indices which share some similarities with our implementations.

2.2.1 Apache Cassandra

Apache Cassandra [11] is a distributed database initially developed by Facebook, later by the Apache Software Foundation. It has a similar data model to HBase's: it is a key-value store represented as a table and rows have columns and columns reside in column families. The difference resides in the fact that, in Cassandra, a column is part of a super column family which in turn is part of a column family, whereas in HBase, there are no super column families.

Cassandra also uses a table partitioning scheme in order to provide scalability. Where HBase uses range partitioning based on the row keys, Cassandra partitions its tables using a type of consistent hashing [15] derived from the Chord [16] distributed hash table on row keys.

Another main difference between HBase and Cassandra is that from the perspective of Brewer's (CAP) Theorem [17] [18], HBase's architecture satisfies the CP (Constistency and Partition Tolerance) properties and Cassandra satisfies the AP (Availability and Partition Tolerance) properties.

Cassandra's secondary indexing implementation stores secondary indices in hidden tables. This method has similarities with both our implementations. It uses tables just as our *Table Based Secondary Indexing* implementations described in Chapter 3. These index tables are partitioned and co-located with the data they index. We do this partitioning and co-locating of indices in our second implementation, *In Memory Secondary Indexing*, described in Chapter 4.

2.2.1 MongoDB

MongoDB [19] is a fairly different type of distributed database. Unlike HBase and Cassandra, MongoDB's data model is document oriented. Instead of storing key values to have a tabular representation, it offers an interface to read, write and run queries on JSON structures. These documents are grouped into collections. Documents in a collection are not required to have the same schema.

MongoDB offers a secondary indexing feature in the sense that any field of a JSON [20] document can be indexed within a collection. There is always a primary index on the mandatory and unique *_id* field of each document. MongoDB stores its secondary indices in B-Tree data structures. The MongoDB indices are partitioned so that each shard (partition) manages its index. MongoDB's documentation strongly advises to have the secondary indices reside in RAM when using MongoDB in order to achieve optimal performance. This partitioning and in memory approach is similar to our second implementation, described in Chapter 4, for which each HBase table region manages its own RAM resident secondary indices.

Chapter 3

Table Based Secondary Indexing

3.1 Introduction

This chapter describes the first approach we used to provide secondary indexing in HBase, which was to use HBase tables to store an inverted index [21].

Our implementation uses two types of specialized HBase tables in addition to the standard HBase user table. The first one is the Index Table which stores the inverted index for a specific column of a specific user table. The second type of table is the Master Index Table which stores information about which table has indices on which of its columns.

To manage the two types of special tables and their contents, we implemented three types of region coprocessors: an HTable Index Coprocessor which processes events on regular user tables, an Index Table Coprocessor which updates the index stored in Index Tables and the Master Index Table Coprocessor which manages the Master Index Table.

Figure 3.1 below provides an overview of what our solution looks like in the form of an example. A given table (T1), which contains user data, has one of its columns (column A) indexed. The inverted index for column A of table T1 is held in table I1. The information of which table holds the inverted index for a given column is stored in the Master Index Table. In our specific example, the information about the location of the index for column A of table T1 is held as tuple (“T1”, (“IDXCOL:A”, “I1”)) in region M2 of Table M. This metadata is used by each HTable Index Coprocessor (HIC), that runs on table regions containing user data, to determine which column is indexed for the table region it is attached to as well as which table contains the index for that column in order to maintain the indices. Each of the tables, T1, M and I1 can be split into a number of regions and each of these regions has its own instance of the corresponding coprocessor attached to it.

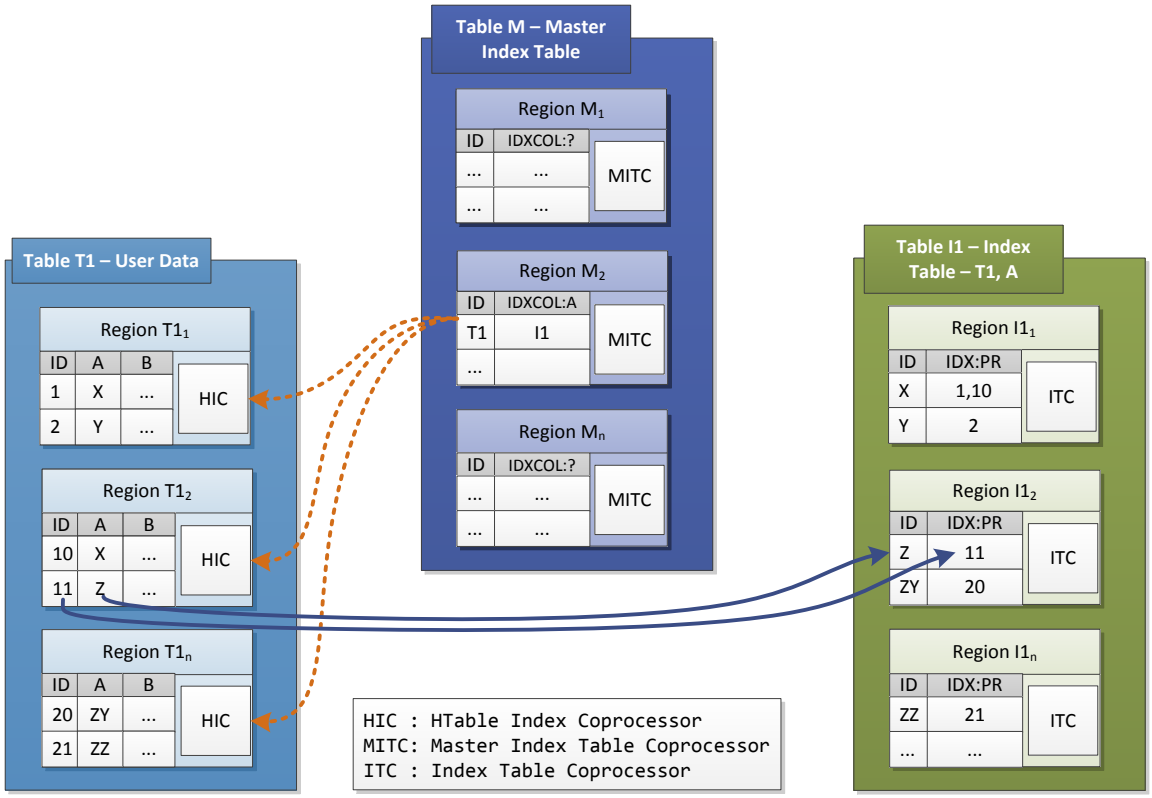

Figure 3.1: Table Based Secondary Indexing Overview

The prepackaged HBase client has been extended in order to provide the methods required for our indexing solution. The added methods allow creating and deleting indices on given columns for given tables as well running optimized queries based on these indices.

3.2 Specialized Tables

This section describes in more detail the contents and structure of the two types of specialized tables, we created for our table based secondary indexing solution: the Master Index Table and the Index Table.

3.2.1 Metadata Storage

The metadata containing the information required for our solution about which column(s) of which table(s) have indices reside in the Master Index Table (MIT), named *__sys__indextable*.

The structure of The Master Index Table is as follows: the row ids are the names of the tables for which secondary indices exist and there is one column family, *idxcol*, which contains a column for each of the indexed columns of the user tables; the name of a column is the concatenation of the user table's column family containing the column to index and the column name.

A cell in the Master Index Table, for a specific row/column combination, contains the name of the Index Table containing the inverted index for the column.

Figure 3.2 below shows an example of the Master Index Table for a system that has a table named *usertable* that has one of its columns (*family1:col2*) indexed and another table, *T2* with column *A:X* also indexed.

__sys__indextable		
Row id	idxcol	
	family1col2	AX
usertable	usertablefamily1col2__idx	
T2		T2AX__idx

Figure 3.2: Example of a Master Index Table

3.2.2 Index Storage

The Index Tables store the inverted index for a specific column of a specific table. The name of an Index Table results from the concatenation of the name of the table, the column family name and the column name where the values to be indexed reside and the __*idx* suffix.

The structure of the Index Tables is simple, it has only one column family *idx* and one column within that column family *pr*: The contents of an Indexed Table T_1 is as follows: given an index on column C of table T_2, the row ids in T_1 are the distinct values found in C and the data stored in *idx:pr* within T_1 is a serialized Java TreeSet containing the row ids from T_2 where the distinct values are located.

We chose the TreeSet object, which is based on the Red-Black tree data structure, to store the row ids instead of a plain list because of performance reasons:

> *This implementation provides guaranteed log(n) time cost for the basic operations (`add`, `remove` and `contains`).* [22]

Figure 3.3 below shows an example of an Index Table containing the inverted index of values stored in column *family:col2* of table *usertable*. In that example, rows *row1* and *row2* of table *usertable* both contain the value *value1* in column *family:col2*.

usertablefamily1col2__idx	
Row id	idx
	pr
value1	Serialized TreeSet(row1,row2)

Figure 3.3: Example of an Index Table

3.3 Specialized Region Coprocessors

Our coprocessor driven table based secondary indexing solution relies on three types of region coprocessors which provide us with some hooks where we can run custom code when modifications occur to the region's data. These are the

`HTableIndexCoprocessor`, the `IndexTableCoprocessor` and the `MasterIndexTableCoprocessor`. This section will describe in details how these coprocessors function and how they react to changes to the regions they observe.

Our solution started with a naïve implementation that did not take care of being optimal in terms of performance. This first implementation was then optimized as much as possible in order to see if it would make coprocessor driven table based secondary indexing a viable, production grade, solution.

3.3.1 Intercepting Client Operations

The `HTableIndexCoprocessor` extends the class `BaseRegionObserver`. As such, instances of `HTableIndexCoprocessor` run on regions. This particular type of coprocessor will be active only on standard user tables. Its goal is to intercept client calls that update the contents of the region it is attached to and send updates to the corresponding index tables if any of the intercepted client calls apply to an indexed column.

The client calls that we intercepted in our implementation were the *Put* and *Delete* operations; this is done by overriding the `prePut()` and `preDelete()`

of the `BaseRegionObserver` which are executed before the actual update in the regular table's data takes place.

The first thing that needs to occur whether the intercepted operation is a *Put* or a *Delete* is to check if any of the indexed columns are affected by the operations. Our first, naïve implementation did a lookup in the Master Index Table every time a *Put* or *Delete* operation was intercepted in order to retrieve the list of indexed columns. This incurred a non-negligible delay to the entire operation and therefore was one of the first areas to be optimized in our final implementation. The final implementation uses a local list of indexed columns which is loaded up when the coprocessor starts and gets updated only when a modification occurs in the contents of the Master Index Table. The update mechanism is described in the MasterIndexTableCoprocessor section below.

3.3.1.1 Put Operation Interception

In the case where a *Put* operation has been intercepted and the `HTableIndexCoprocessor` has asserted that the operation will have an effect on one or more of the indexed columns within the table, the `HTableIndexCoprocessor` executes a *Get* operation locally to retrieve the current values contained for the row to be updated. From the inspection of the

current column values for the row to be updated, one of two sequences of actions may ensue.

The first and simplest of sequences occurs when the current row does not contain any value for an indexed column to be updated. The second sequence occurs when the current row already contains a value for an indexed column and the *Put* operation will change that value.

In the simpler case where no value already exists in the indexed column, the `HTableIndexCoprocessor` will send a tweaked *Put* operation to the appropriate Index Table to add the current row id into the index. The tweaked *Put* operation overrides the normal one by setting a special value (`type_-00001-_put`) in the field normally reserved for the column name. The rest of the tweaked *Put* operation's fields retain their original signification: the row is the new value for the indexed column and column family remains *idx*. It is safe to hijack the column name field because the column name plays no part into pinpointing which region the *Put* operation is destined to, and there is only one known column in an Index Table. The reason why we use this method is that we need the actual index update to be executed at the region server of the index and not at the region server of the user table, thereby minimizing traffic between the regions and spreading the computing time. Once the index is updated, the

`prePut()` method completes and control is given back to the region server so the *Put* initiated by the client can complete.

Figure 3.4 is an example sequence diagram where *col1* is indexed, *col2* is not indexed and there is no value in *col1* for row *row1*. In this example, the user table is contained within one region (Region A) but the index table for *col1* is split into two regions; Region B contains the inverted index of *col1* for value *z* and Region C the one for value *x*. The client puts a row *row1* with *col1* =*x* and *col2*=*y*. In this case, only one extra remote operation is incurred by using our table based secondary indexing solution.

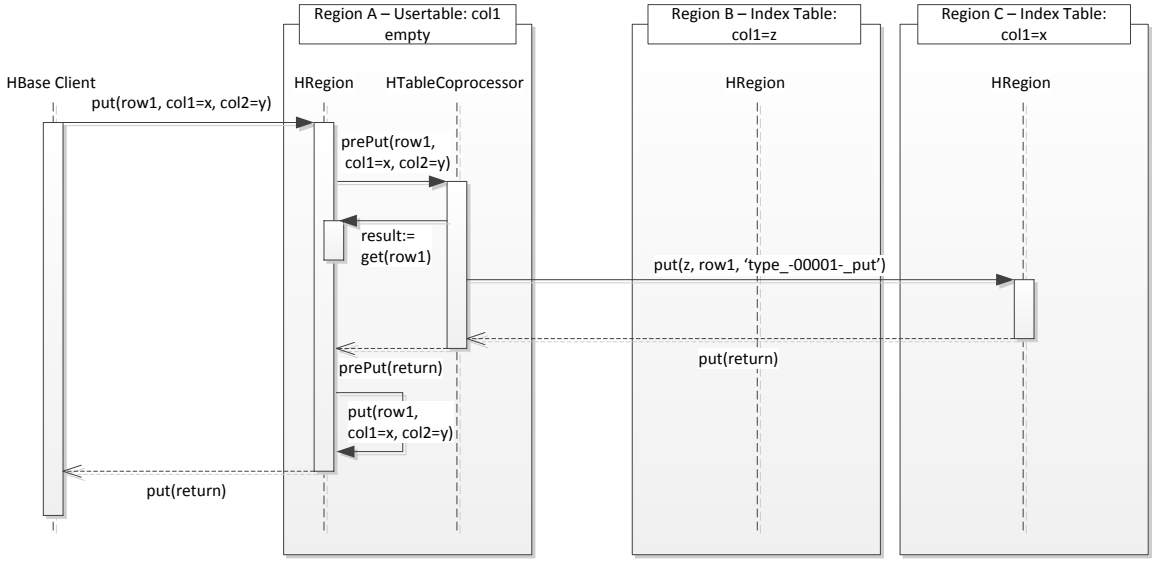

Figure 3.4: Sequence for a Put that updates an empty indexed cell

In the case where a value already exists in the indexed column, an extra operation will need to be executed in order to remove the existing index

reference to the updated row for the existing value prior to adding a reference to the updated row for the new column value. In this scenario, the `HTableIndexCoprocessor` will first send a tweaked *Put* operation to the appropriate Index Table to remove the row id from the index of the current value. This tweaked *Put* operation also overrides the normal one by setting a special value (`type_-00001-_del`) in the field normally reserved for the column name and sets the current value of the indexed column in the row field, thereby sending a request to the region which contains the inverted index of the current value to remove the reference to the row for that value.

Once this first operation has been completed, the indexing system is effectively in the state where the simpler sequence described previously can be executed, therefore, the tweaked *Put* operation of type `type_-00001-_put` can be sent.

Figure 3.5 is an example sequence diagram where *col1* is indexed, *col2* is not indexed and *col1* has an initial value of *z* for row *row1*. In this example, the user table is contained within one region (Region A) but the index table for *col1* is split into two regions; Region B contains the inverted index of *col1* for value *z* and Region C the one for value *x*. The client puts a row *row1* with *col1* =*x* and *col2*=*y*. In this case, two extra remote operations are incurred by using our table based secondary indexing solution.

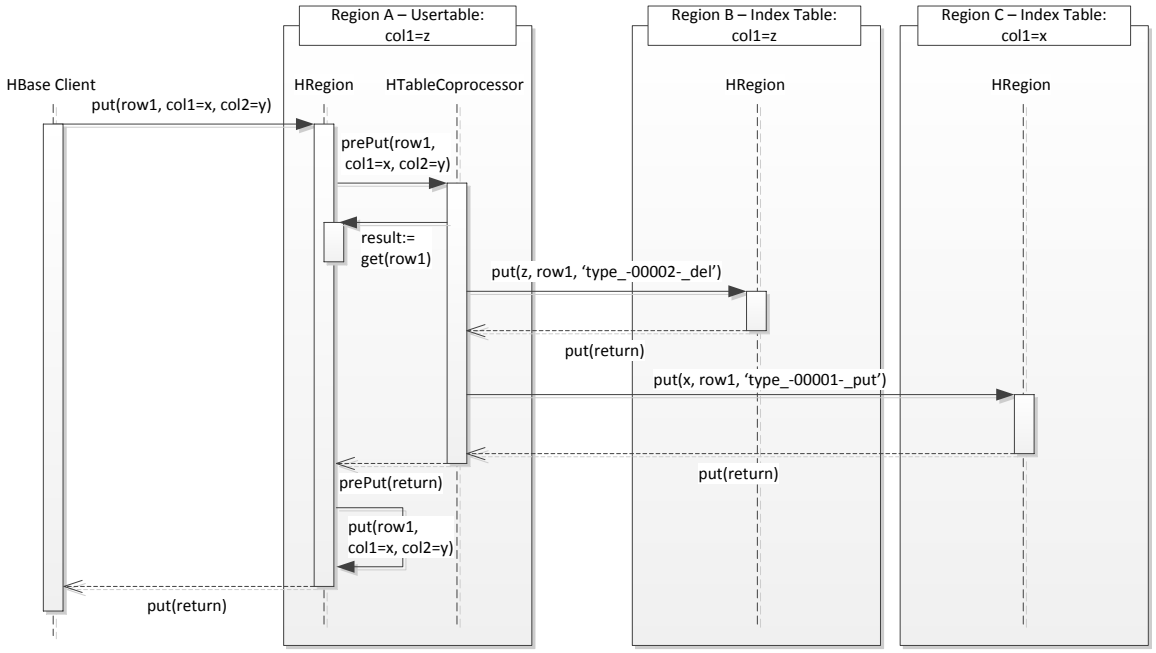

Figure 3.5: Sequence for a Put that updates a non-empty indexed cell

3.3.1.2 Delete Operation Interception

In the case where a *Delete* operation has been intercepted and the `HTableIndexCoprocessor` has asserted that the operation will have an effect on one or more of the indexed columns within the table, the `HTableIndexCoprocessor` executes a *Get* operation locally to retrieve the current values contained for the row to be updated. The `HTableIndexCoprocessor` will then send a tweaked *Put* operation to the appropriate Index Table to remove the row id from the index of the current value for each indexed value that will be deleted. This tweaked *Put* operation also overrides the normal one by setting a special value (`type_-00001-_del`) in the

field normally reserved for the column name and sets the current value of the indexed column in the row field, thereby sending a request to the region which contains the inverted index of the current value to remove the reference to the row for that value.

Figure 3.6 is an example sequence diagram where both *col1* and *col2* are indexed; *col1* has an initial value of *z* for row *row1* and *col2* has an initial value of *x*. In this example, the user table is contained within one region (Region A), the index table for *col1* is in Region B and the index table of *col2* is in Region C. The client deletes columns *col1* and *col2* from row *row1.* In this case, two extra remote operations are incurred by using our table based secondary indexing solution.

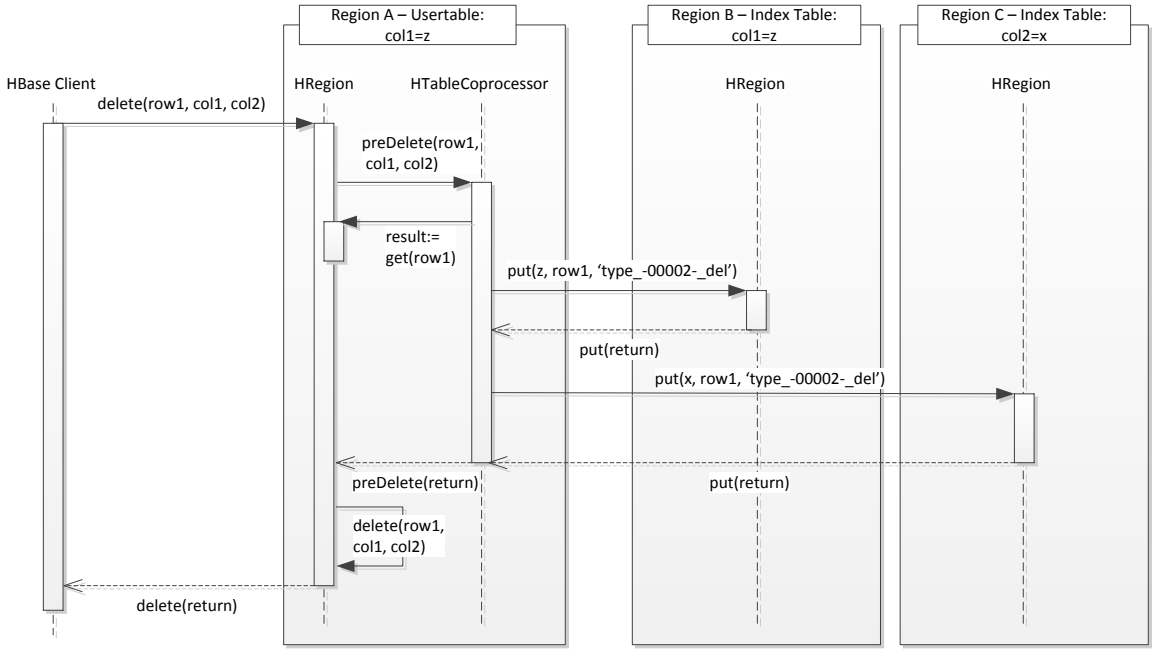

Figure 3.6: Sequence for a Delete that updates non-empty indexed cells

3.3.2 Updating the Indices

The `IndexTableCoprocessor` extends the class `BaseRegionObserver`. As such, instances of `IndexTableCoprocessor` run on regions. This particular type of coprocessor will be active only on Index Tables. Its goal is to manage the update of the index contained within the region.

This goal is achieved by intercepting two special types of *Put* operations that are sent by `HTableIndexCoprocessor` when values of indexed columns are updated.

Upon receiving a *Put* operation of type `type_-00001-_del` or `type_-00001-_put`, the `HTableIndexCoprocessor` first acquires a lock on the row containing the index for the requested value. Then, it runs a *Get* operation on the local Region to retrieve the serialized TreeSet containing the index. Once retrieved and de-serialized, the index is updated accordingly, i.e. the reference is added or removed from the TreeSet structure depending on the type of *Put* operation being processed. Once the update is completed, the TreeSet containing the updated index is serialized and a local *Put* operation is issued to the local region in order to put the index back into the Index Table. Finally the row lock is released and the index processing is complete.

Figure 3.7 below is an example sequence diagram where column *col1* of table *Usertable* is indexed; *col1* has an initial value of *z* for row *row1*. In this example, the user table is contained within one region (Region A), the index table for *col1* is in Region B. The client deletes columns *col1* and *col2* from row *row1.* In this case, one extra remote operation is incurred by using our table based secondary indexing solution.

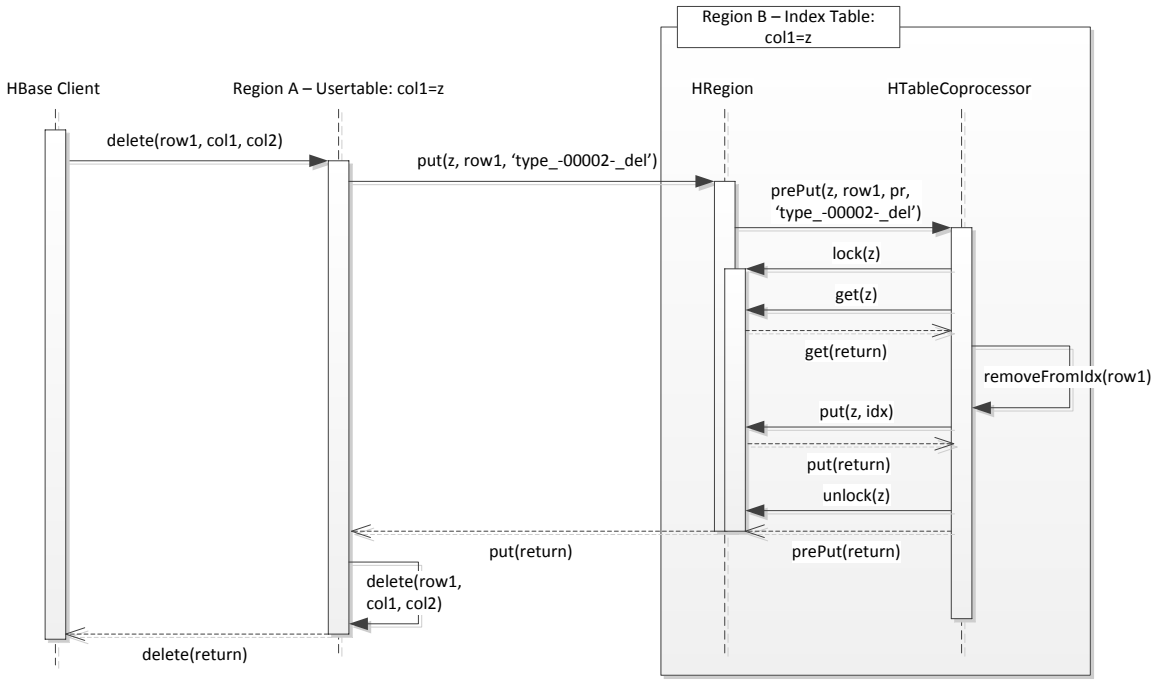

Figure 3.7: Sequence for a removal of an entry in an index.

3.3.3 Maintaining Indexing Metadata

The `MasterIndexTableCoprocessor` extends the class `BaseRegionObserver`. As such, instances of this coprocessor run on regions. This particular type of coprocessor will be active only on the Master Index Table. Its goal is to manage the update of the index contained within the region and

notify the running `HTableIndexCoprocessor` instances when they need to refresh their cached metadata about what columns are indexed. Updates to the metadata are issued by our modified HBase client when an index is either created or deleted on a column of a given table.

We introduced this coprocessor as part of the optimization of our first naïve implementation in order to minimize the lookups needed by the `HTableIndexCoprocessor` instances to decide whether a received operation has an impact on an indexed column or not; prior to the introduction of this coprocessor, a lookup was done for every intercepted operation.

To distribute the notification from the `MasterIndexTableCoprocessor` to all the regions, which can be spawned dynamically by HBase when a region is split, we used the open source group communication library named JGroups[5].

We configured JGroups to provide reliable and totally ordered message multicast over TCP. We decided to use TCP because it is supported in a wider range of environments whereas IP multicast may not be. Since messages occur with a very low frequency, each time an index is created or deleted, the overhead for TCP is not a factor that would prevent us from using that type of transport.

Upon startup, each `MasterIndexTableCoprocessor` and `HTableIndexCoprocessor` join a channel named

[5] JGroups - A Toolkit for Reliable Multicast Communication, http://www.jgroups.org/

`MasterIndexUpdateCluster`. The `MasterIndexTableCoprocessor` instances send notification messages whenever and update is done to the Master Index Table over that channel and each `HTableIndexCoprocessor` instances listens for the notifications over the channel and update their indexed column lists whenever they receive a notification.

Figure 3.8 depicts the architecture of the notification sent in reaction to updates to the Master Index Table to each HTable Index Coprocessor in the system using JGroups.

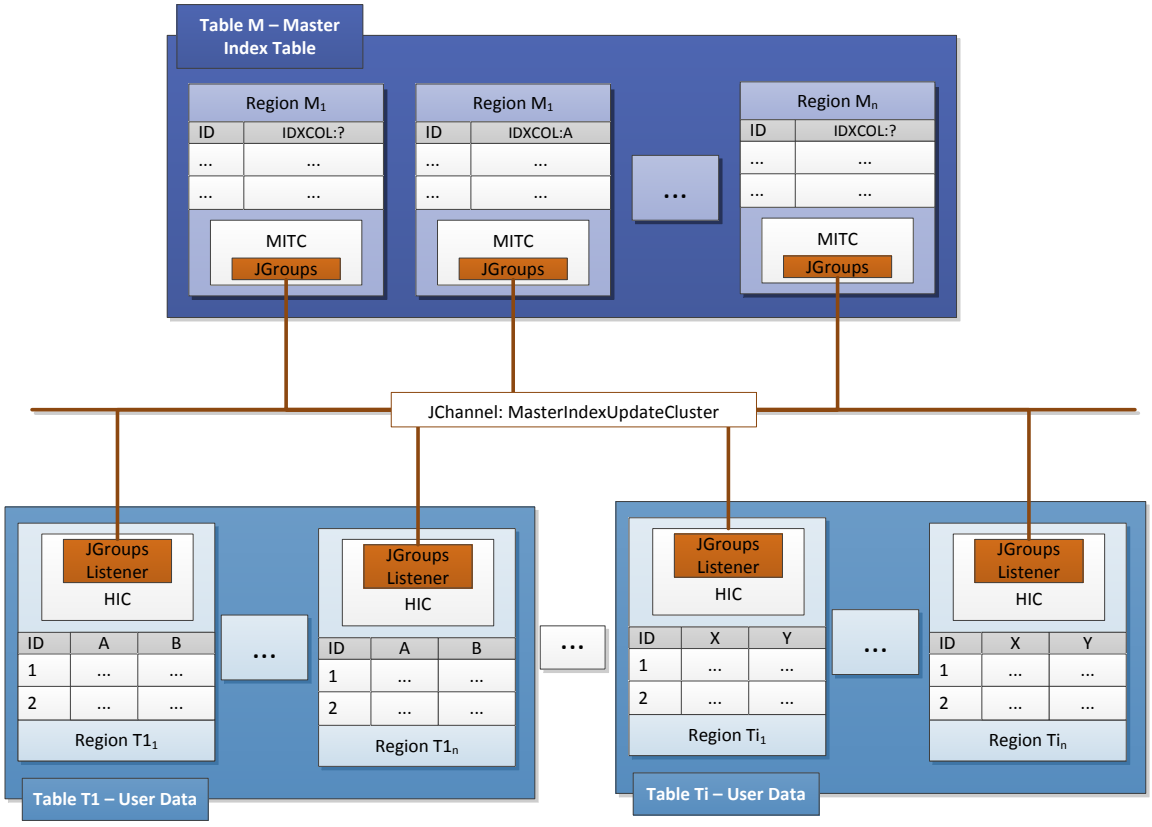

Figure 3.8: Notification architecture for updates to the Master Index Table.

3.4 Extended HBase Client Operations

In order to make use of the secondary indexing we created a class named `HIndexedTable` which extends the HBase `HTable` class; this table is part of the HBase client. We did not override any of the `HTable` methods; this provides us with backwards compatibility. Instead, we added methods such as `createIndex()`, `deleteIndex()` and `getBySecondaryIndex()`.

3.4.1 Creating A Column Index

To create an index on a column for a given table T_1, a user must invoke the `createIndex()` method with the column name as a parameter which in turn, does the operations described below.

First, it creates a table to contain the new index, T_2. The name given to T_2 is the result of the concatenation of T_1's name, the column family containing T_1's column to be indexed, T_1's column name to be indexed and the *__idx* suffix.

Once T_2 has been successfully created, the client launches a Map/Reduce [23] job that will populate T_2 using Hadoop's Map/Reduce framework.

The mapper we implemented, `IndexMapper`, extends the `TableMapper` class which uses an HBase table as an input; in our case, it uses Table T_1 as an input. For each row, `IndexMapper`'s map() method retrieves the row id (RID) and the value (V_i) contained in the column that is to be indexed then outputs the tuple (V_i, RID). The Map/Reduce framework then processes the outputs of `IndexMapper`'s map() method and generates the map V -> $List_{RID}$ where $List_{RID}$ is a list of RID associated with each V_i.

The reducer we implemented, `IndexReducer`, extends the `TableReducer` class which uses an HBase table as an output; in our case, table T_2 is the output table. `IndexReducer` receives a tuple (V_k, $List_{RID}$) for each entry in the map previously created by the Map/Reduce framework. For each of the tuples received, `IndexReducer` creates a new Java `TreeSet` object, fills it with the contents of the $List_{RID}$ then inserts a new row into T_2 where the row id is V_k and the value is the generated TreeSet.

Finally, `createIndex()`'s last task is to insert an entry in the Master Index Table indicating that there is an index for T_1's specified column.

Figure 3.9 below shows an example sequence diagram of a map/reduce request done during index creation. In this example, HBase nodes are co-located with

HDFS nodes. Table T1 which contains a column to be indexed is split into two regions ($T1_A$ and $T1_B$) and Index Table I1 which has been allocated region $I1_B$. The requests to create the index table and add an entry into the Master Index Table have been omitted from the figure to keep the diagram more compact.

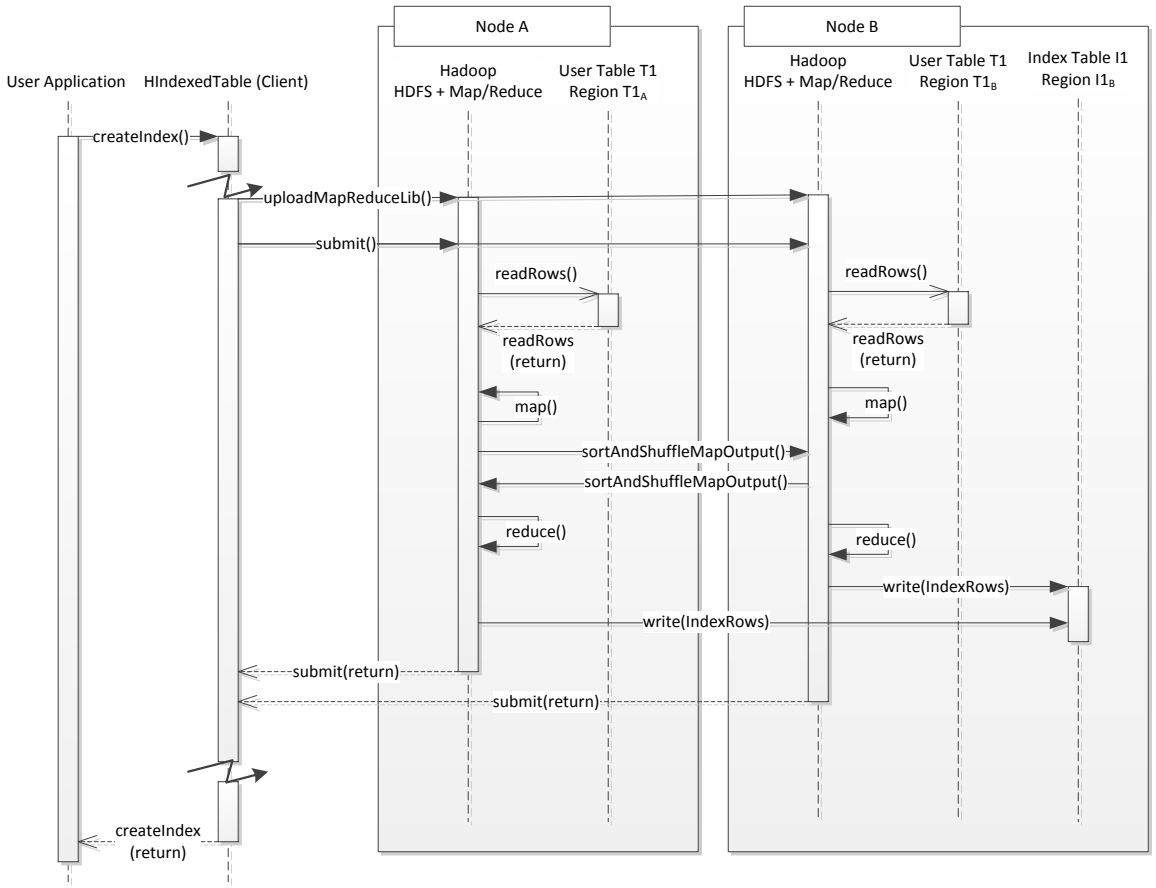

Figure 3.9: Map/Reduce index creation sequence diagram.

3.4.2 Deleting A Column Index

To delete an index on a column C_1 for a given table T_1, a user must invoke the `deleteIndex()` method with the column name as a parameter which in turn, does the two following operations.

First, a request is issued to delete the cell at row T_1 column idxcol:C_1 from the Master Index Table. This, as a side effect, updates the indexed columns list within the HTable Index Coprocessors. Therefore, no more accesses will be made to the Index Table holding the index to be deleted.

To complete the removal of an index, once the index's metadata has been removed from the Master Index Table, the request to delete the actual table holding the index for column C_1 of table T_1 is issued.

3.4.3 Query Using An Index

The getBySecondaryIndex() method, invoked with the column name and the value to query, performs the SQL equivalent of a WHERE clause on a single column with a given value; the query must be done on an indexed column. This method was primarily implemented to test the performance of queries, by comparing them with *Scan* operations issued with a specific filter on a column.

A filtered *Scan* operation will traverse the entire table and return the rows where the filter matches a given column's content; the filtering is done on the region server side.

The getBySecondaryIndex() method will instead, first query the index table for a specific column to retrieve all the rows where the queried value exists, then query the table for all the returned rows. The multiple row query is optimized within the standard HBase client by executing a batch query.

Figure 3.10 illustrates the sequence of actions taken when an invocation to the getBySecondaryIndex() method occurs. In this example, a user application wants all the rows where column C_1 is *x*. The *Get* operation issued of the index table returns the list row IDs for the rows that match the query "C_1 = *x*" then the actual rows are retrieved from the two regions where they reside and returned to the user application.

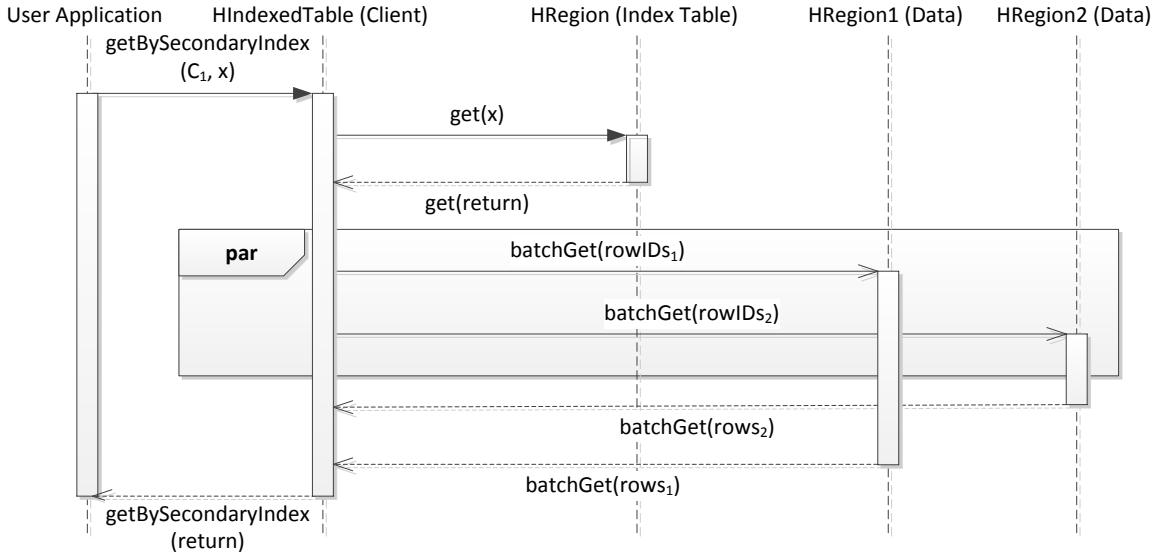

Figure 3.10: getBySecondaryIndex() sequence diagram.

Chapter 4

In Memory Secondary Indexing

4.1 Introduction

This chapter describes the second approach we used to provide secondary indexing in HBase.

We decided to do a second type of secondary indexing implementation that would not have some of the drawbacks our first implementation had.

The basic idea is that indices are partitioned into region indices in such a way that each table region has its own index. These region indices are stored in each Region Server's main memory to minimize performance impact. With this, we

achieve several things. First, we reduce the amount of network traffic required by our first implementation. This is true because no requests are needed between different region servers to either maintain an index or query an index as each region is responsible for its indices. We also removed the requirement for the client to first do the index lookup then query the actual table for the data using the information retrieved from the index lookup. We achieved this by having each region server process the index lookup from a client query using their co-located indices and return the queried data; all the client needs to do is send the query in parallel to all region servers and await, then combine the results.

A second advantage is a lower write overhead. We keep the index in a specialized memory structure and only persist it when a region gets closed or split in order not to have to rebuild the indices the next time the regions are restarted.

Our in memory implementation, like the table based implementation, uses one type of specialized HBase table to store the indexing system's metadata, the Master Index Table.

The region indices are maintained and accessed by two types of coprocessors running on user data regions. One of them, the HTable Index Coprocessor,

maintains the region's index once it exists. The other coprocessor, the Index Coprocessor Enpoint, receives requests directly from a client and allows it to create and delete an index as well as run queries that take advantage of the index.

Figure 4.1 provides an overview of what our solution looks like. Table T1 represents a table containing user data and table M is the Master Index Table. Each of the tables T1 and M is split into a number of regions. T1's regions are spread across 2 region servers. One column is indexed (column A) for the specific user data table T1; this meta data is stored in the Master Index Table (Table M) and used by each HTable Index Coprocessor and Index Coprocessor Enpoint. The inverted index for column A of table T1 is partitioned for each region $T1_n$ and stored in the region servers' main memory.

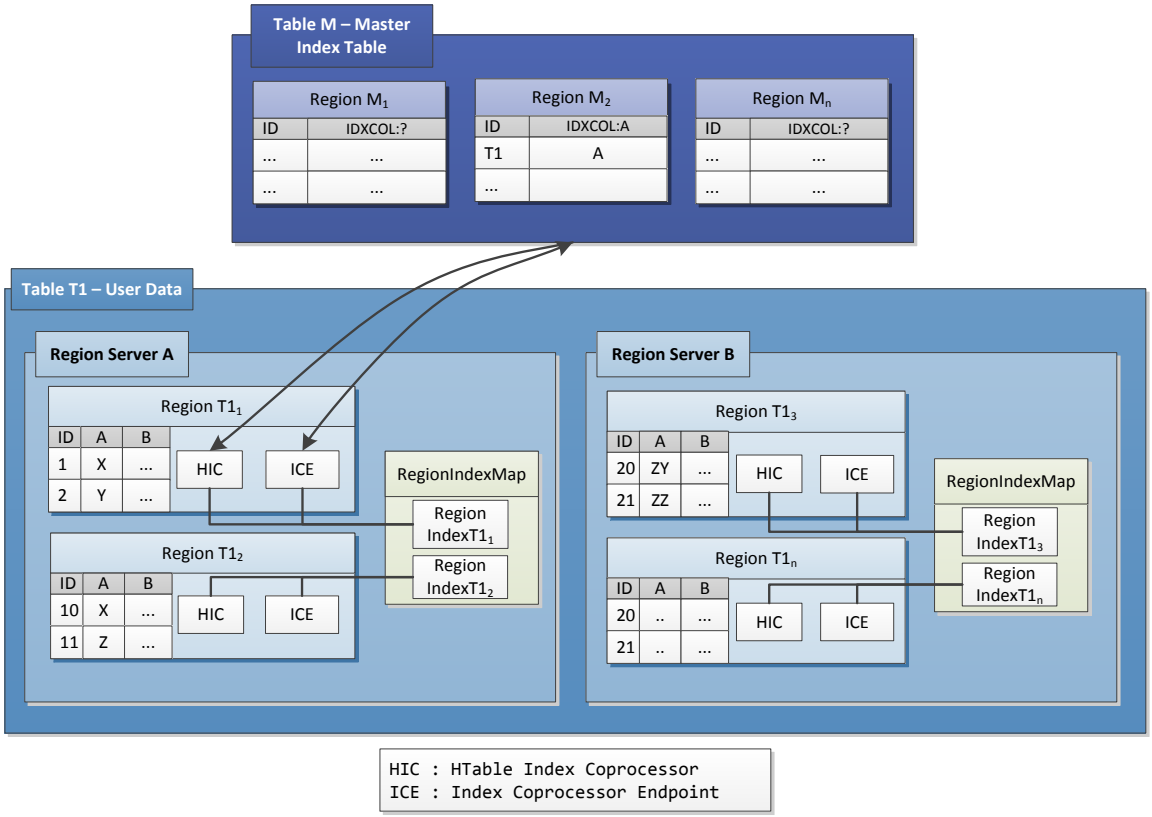

Figure 4.1: In Memory Indexing Overview

The structure of the Master Index Table is identical to the one used for the Table Based Secondary Indexing solution described in section 3.2.1 Metadata Storage.

4.2 Region Index Structure and Storage

Figure 4.2 below shows the class diagram for the in memory index. It consists of four classes, the RegionIndexMap class, the RegionIndex class, the RegionColumnIndex class and the RowIndex class.

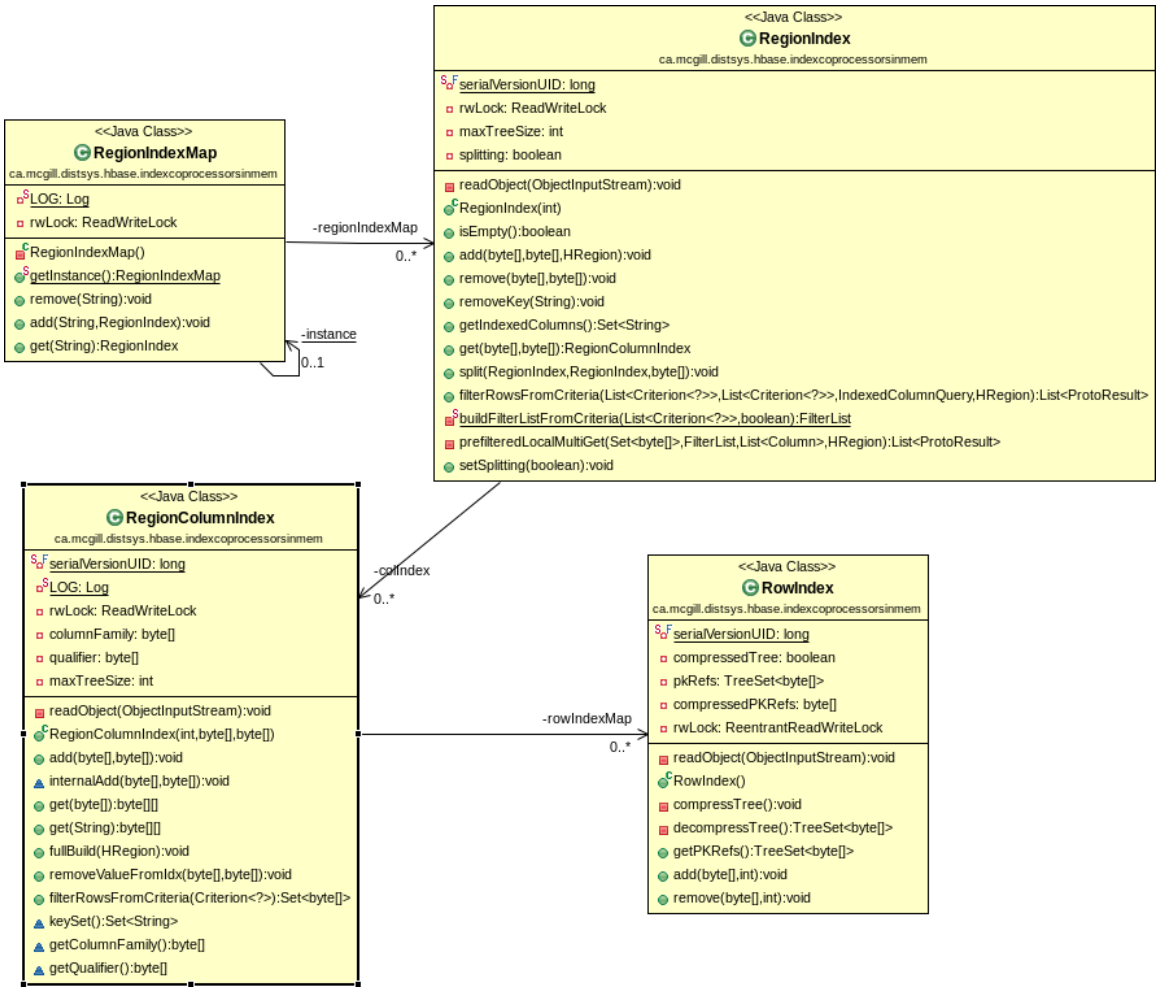

Figure 4.2: Region Index Class Diagram

4.2.1 Region Index Map

An HBase Region Server may serve more than one region. For that reason, the RegionIndexMap class has been designed as a simple singleton containing a map of all the RegionIndex objects for a given Region Server; the internal map is addressed by region name. It does not provide any functionality other than getting, adding and removing RegionIndex objects from its internal map.

All the RegionIndexMap methods are synchronized using a reentrant reader-writer lock in order to provide thread safety.

4.2.2 Region Index

The RegionIndex class's purpose is to hold the index for a given region and provide methods to manipulate, serialize and query the index.

The region's index is contained in an internal map of RegionColumnIndex objects, one map entry per indexed column, addressed by column name.

Methods have been implemented to create ((*add()*) and delete (*remove()*) an index on a specific column. The RegionIndex's *add()* method fully scans the region in order to generate the index data prior to adding it to the internal map whereas the *remove()* method simple removes the entry from the internal map.

A RegionIndex also offers methods for querying the region while taking advantage of the index (*filterRowsByCriteria()*).

When a region reaches a certain predetermined size threshold, HBase's Region Server splits that region in two new regions. When this happens, a region's index has to be split in such a way that each newly created region will have its corresponding index. For that reason, a method has been implemented in the RegionIndex class that splits (*split()*) the region index. The split method marks the RegionIndex as being split and no further operations are allowed on it; an exception is raised if any attempt is made to access that index.

RegionIndex objects are serialized and persisted to HDFS when a region is closed and as such, RegionIndex and all the objects it contains implement specific deserialization methods for their transient attributes.

All the methods of the RegionIndex class are synchronized using a reentrant reader-writer lock in order to provide thread safety.

4.2.3 Region Column Index

A RegionColumnIndex object contains the index for a single indexed column of a given region. It contains an internal map $V_i \rightarrow R_j$ where the V_i are the distinct

values stored in the indexed column and R_j are RowIndex objects containing the row ids where these values are located.

RegionColumnIndex offers methods to add and remove entries from the index as well as perform the parts of queries based on criteria which are specific to an indexed column (*filterRowsByCriteria()*)..

4.2.4 Row Index

The RowIndex class's purpose is a wrapper around a Java TreeSet object used to store an ordered list of row ids. It also provides the functionality to compress that set when a certain configurable size threshold is reached in order to conserve memory; doing so decreases significantly the performance but may be an acceptable tradeoff in some circumstances.

4.3 Specialized Region Coprocessors

Our In Memory Secondary Indexing solution relies on two types of region coprocessors. One provides us with some hooks where we can run custom code when modifications occur to the region's data, the `HTableIndexCoprocessor`. The other is called explicitely by the client via remote procedure calls, the `IndexCoprocessorInMemEndpoint`. This section will describe in details how

these coprocessors function and how they react to changes to the regions they observe and respond to client requests.

4.3.1 Intercepting Client and Server Operations

The `HTableIndexCoprocessor` extends the class `BaseRegionObserver`. This particular type of coprocessor will be active only on standard user data tables. Its many functions include: intercept client operations that update the contents of the region it is attached to, and then update the index for the region if required, as well as intercept the region server open, split and close operations in order to load, split and persist the index.

The client operations that we intercepted in our implementation were the *Put* and *Delete* operations; we did this by implementing the `prePut()` and `preDelete()` methods. The region open, split and close operations are intercepted, respectively by implementing the `postOpen()`, `preSplit()` and `postClose()`. These specific methods are overridden methods of the `BaseRegionObserver` in our `HTableIndexCoprocessor`.

4.2.1.1 Region Open Operation Interception

After a region is started by a region server but just before it is made available to clients, the region server invokes the `postOpen()` method of all the coprocessors attached to that region. Our implementation takes advantage of this hook into the region server startup sequence to load the region's index into memory. This is done by reading a specifically named file from HDFS which contains the compressed serialized index for the opened region. In case the file does not exist or is corrupted, the entire region is scanned in order to recreate the index in memory; this can be a lengthy operation but should only occur in cases of recovery after a region server has crashed.

4.2.1.2 Region Close Operation Interception

After a region is closed by a region server but just before it is stopped and unloaded from memory, the region server invokes the `postClose()` method of all the coprocessors attached to that region.

Our implementation uses this step in the `HTableIndexCoprocessor` to persist the compressed RegionIndex object corresponding to the region being closed to HDFS in the same directory where the region data files are stored. Doing this speeds up region bootstrapping since a full index rebuild will not be necessary. Also, by using that specific HDFS location, we ensure that the index file will be

available to any region server that will be elected to serve the region once it is started again.

The last step that is done is to remove the closed region's RegionIndex from the RegionIndexMap since the region it indices will be unloaded and no longer available until it is restarted.

4.2.1.3 Region Split Operation Interception

Upon intercepting a *split* operation, the `HTableIndexCoprocessor` splits the current region's RegionIndex into two RegionIndex objets by using the same split point the region server will use to split the regions. Once the index is split, it is serialized and persisted to HDFS using a specific filename pattern that will be recognized by the daughter regions that result from a region split when they start up. Finally, the parent's RegionIndex is removed from the RegionIndexMap since the region it indices will be unloaded and no longer exists.

4.2.1.4 Put Operation Interception

When a *Put* operation is intercepted and the `HTableIndexCoprocessor` has asserted that the operation will have an effect on one or more of the indexed columns within the region, the `HTableIndexCoprocessor` executes a *Get* operation locally to retrieve the current values contained for the row to be

updated. From the inspection of the current column values for the row to be updated, one of two sequences of actions may ensue.

The first and simplest of sequences occurs when the current row does not contain any value for an indexed column to be updated. The second sequence occurs when the current row already contains a value for an indexed column and the *Put* operation will change that value.

In case where no value already exists in the indexed column, the `HTableIndexCoprocessor` retrieves the affected RegionColumnIndex from the RegionIndex and invokes the *add()* method on it which effectively adds the new reference to the new value's row id in the index.

If the *Put* operation's effect is to modify an existing cell's value for an indexed column, the `HTableIndexCoprocessor` will first invoke the *removeValueFromIdx()* on the appropriate RegionColumnIndex to remove the old value's reference to the row id before invoking the *add()* method.

Figure 4.3 below is an example sequence diagram where *col1* is indexed, *col2* is not indexed and there is no value in *col1* for row *row1*. In this example, the user table is contained within one region (Region A). The client puts a row *row1* with

col1 =*x* and *col2*=*y*. This diagram shows the fact that each *Put* operation incurs an added random read operation (*Get* on the local region) to determine whether the current row contains a value in *col1*. Note however that no extra network latency is involved in this scenario to manage an indexed column vs. a non-indexed one due to the fact that the index data is co-located with the region data.

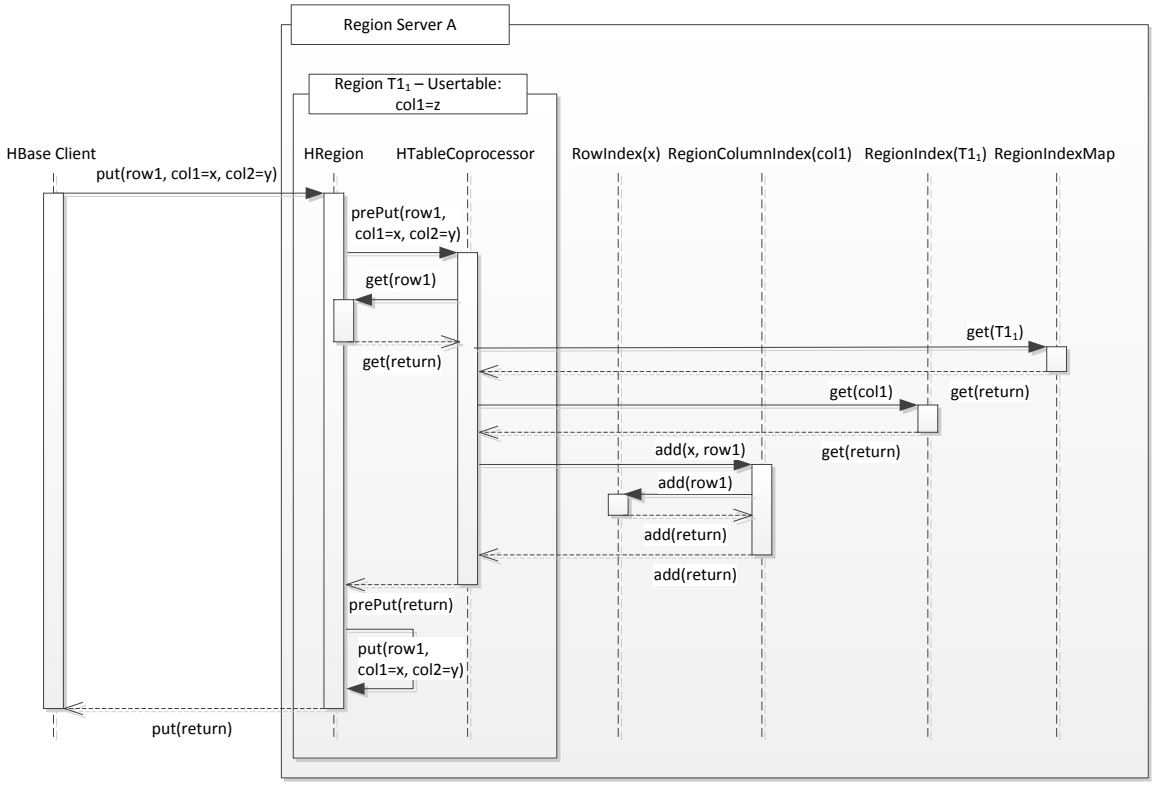

Figure 4.3: Sequence for a Put that updates an empty indexed cell

4.2.1.5 Delete Operation Interception

Upon intercepting a *Delete* operation, the `HTableIndexCoprocessor` determines whether that the operation will have an effect on one or more of the indexed columns within the table. If this is the case, the

`HTableIndexCoprocessor` executes a *Get* operation locally to retrieve the current values contained for the row to be updated. The `HTableIndexCoprocessor` then retrieves the affected RegionColumnIndex from the RegionIndex and invokes the *removeValueFromIdx()* method on it which effectively removes the reference to the deleted value's row id in the index. Figure 4.4 is an example sequence diagram where *col1* is indexed and *col2* is not; *col1* has an initial value of *z* for row *row1*. In this example, the user table is contained within one region (Region A). The client deletes columns *col1* and *col2* from row *row1.* This diagram shows the fact that each *Delete* operation incurs an added random read operation (get on the local region) to determine whether the current row contains a value in *col1*. No extra network latency is involved in this scenario to manage an indexed column vs. a non-indexed one.

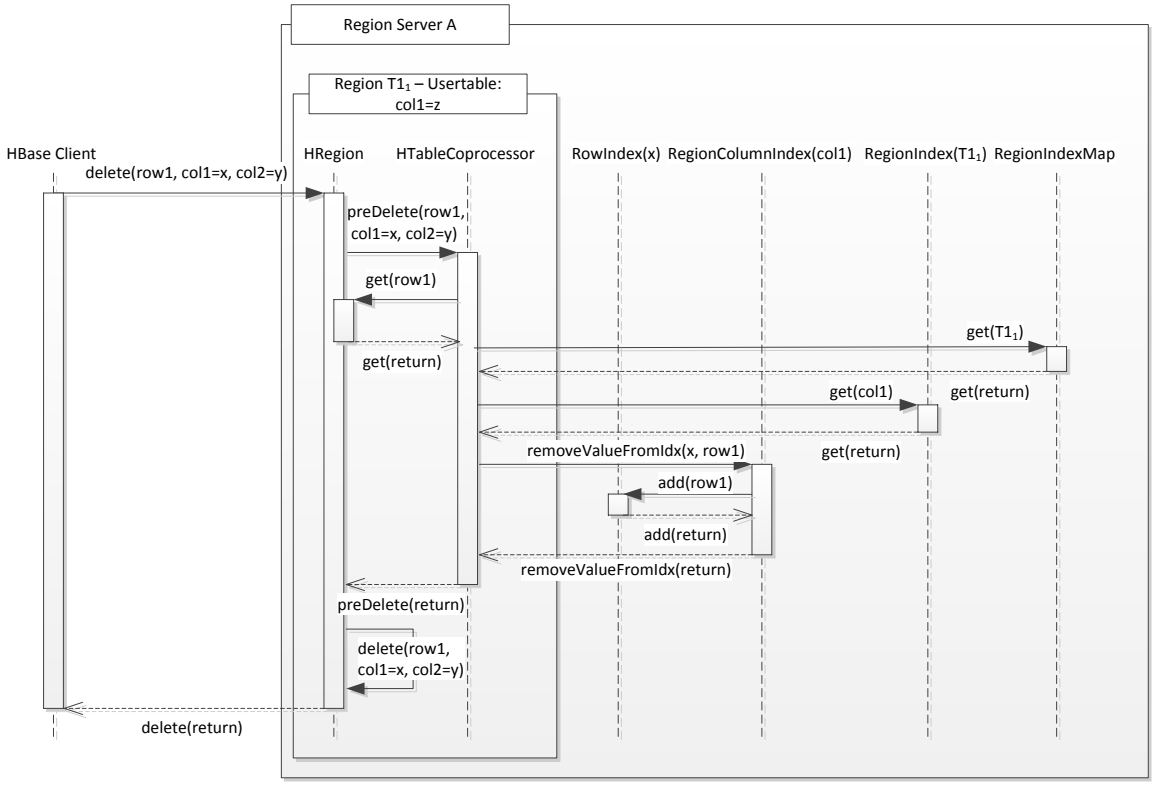

Figure 4.4: Sequence for a Delete

4.3.2 Index Management and Querying

The `IndexCoprocessorInMemEndpoint` implements the `CoprocessorService` interface and as such, it offers methods that are remotely and explicitly invoked by the client. There are three methods available to a client in this coprocessor: `createIndex()`, `deleteIndex()` and `execIndexedQuery()`. These methods respectively allow a client to create and delete and index on a column of a table and run a query that will take advantage of the index.

Our implementation extends HBase's HTable and offers access to these remote procedures.

4.3.2.1 Creating an Index

To create an index on a given column of a table, one invokes the `createIndex()` method of our extended HBase client. This method first checks that the Master Index Table exists, if not, it creates it. The method will then update the Master Index Table in order to persist the information that a given column of a given table is indexed. The last operation the client's `createIndex()` method does is, for each region server serving a region of the table for which the column is to be indexed, to remotely invoke `IndexCoprocessorInMemEndpoint`'s `createIndex()` method in parallel.

When a `IndexCoprocessorInMemEndpoint`'s `createIndex()` method is invoked, it gets the region's existing RegionIndex from the RegionIndexMap or creates and adds a new one to the map if it doesn't exist. It then adds a new RegionColumnIndex to the RegionIndex if it doesn't already exist.

4.3.2.2 Deleting an Index

To delete an existing index on a given column of a table, the `deleteIndex()` method of our extended HBase client must be invoked. The method first updates

the Master Index Table in order to persist the information that a given column of a given table is no longer indexed. The client's `deleteIndex()` method then, for each region server serving a region of the table for which the column is no longer to be indexed, remotely invokes `IndexCoprocessorInMemEndpoint`'s `deleteIndex()` method in parallel.

When a `IndexCoprocessorInMemEndpoint`'s `deleteIndex()` method is invoked, it gets the region's existing RegionIndex from the RegionIndexMap then removes the corresponding RegionColumnIndex from that RegionIndex.

4.3.2.3 Querying a Table Using an Index

Our extended HBase client offers the possibility to query a table and return the rows where a specific value is contained in one or more indexed columns. The closest native HBase operation to this is a filtered Scan operation. The difference resides in the fact that a filtered Scan will need to traverse the entire table whereas our indexed queries only access the stored rows that match the query criteria. Our implementation is thus expected to offer a better performance, especially when the ratio of matching rows to total rows is small.

In order to perform an indexed query, a user of our extended client creates an IndexedQuery object which contains a list of criteria that suits their purpose, a flag specifying whether the criteria are joined by logical AND or OR operators and optionally a list of columns to be included in the result. We based our design of the criteria on the filters that can be used by HBase native operations in order to offer a familiar interface to the users of our client. This IndexedQuery object is then passed to the *execIndexedQuery()* method. This client method will, in turn, remotely invoke each `execIndexedQuery()` method of the target table's region's `IndexCoprocessorInMemEndpoint` endpoint coprocessor in parallel and wait until it collected all the results before presenting the combination of the results to the user. This also offers a performance and scalability advantage over the filtered Scan operation given the fact that the filtered Scan processes regions sequentially and our query mechanism does so in parallel.

On the server side, when a `IndexCoprocessorInMemEndpoint` endpoint coprocessor has its `execIndexedQuery()` method invoked, it will first ensure that at least one of the criteria within the query applies to an indexed column and if so split the criteria list into two lists: one list containing criteria on indexed columns and another on non-indexed columns. The first list is used to scan the

index and retrieve the row ids of the rows matching the criteria. The second list is transformed as a list of HBase filters.

The method then performs a series of Get operations by using each row id identified from the index and, if necessary (AND operator used to join the criteria together) the list of HBase filters. These results are packed into a list and returned to the client.

Chapter 5

Experimental Results

5.1 Introduction

In this chapter, we evaluate the performance of both our coprocessor driven secondary indexing systems for HBase described in the previous two chapters. The first part of this chapter describes our environment, hardware and software, as well as the benchmarking tool used to collect performance results. The last part contains the performance results and their comparison to HBase without using indices obtained from different test configurations.

5.2 Experimental Test Bed

Our experiments were conducted on a cluster of 2 nodes. Each node is a virtual machine running on the same physical machine (host). Each virtual machine has the same specifications as shown in Figure 5.1

Host Configuration	
CPU	Intel(R) Core(TM) i7-3820 @ 3.6GHz
RAM	64 GB 1600MHz
Storage	300GB SATA-3 10K RPM HDD
Operating System	Windows 7 Ultimate, 64-bit SP1
Virtualization Software	
Product	VMWare® Workstation
Version	9.0.0
Virtual Machines	
CPU / Node	1 CPU / 2 Cores
RAM / Node	8GB (Dedicated)
Storage	Local – 20GB
Operating System	CentOS 6.3 (Final)

Figure 5.1: Test Bed Host Environment

Each node runs Hadoop 1.0.3 which accesses the local disk. HBase is configured to use Hadoop's HDFS for storage. Hadoop, HBase and our extended HBase client run using the 64 bit version of Java Runtime Environment 1.6.

5.2.1 Detailed Test Configuration

We use the default HBase settings in our tests except where we increase the heap size for our region servers to 2GB so as to provide enough memory for the large data sets loaded during testing.

Testing of both the Table Based Secondary Indexing implementation and the In Memory Secondary Indexing implementation have been done on the development version 0.95 of HBase.

5.2.2 Benchmarking Suite

For our tests, we used the Yahoo! Cloud Serving Benchmark (YCSB) tool. YCSB's default workloads involve inserting 100% randomized byte arrays as cell data. For our purposes, this type of workload did not offer the necessary control with respect to the number of occurrences of a specific data item for a given indexed column. For that reason, we created our own workload generator. Instead of randomizing the data bytes to be inserted into the table, our workload generator reads from a dictionary file and, if required by a parameter, selects one word at random to be inserted, in a given column, a configured amount of times relative to the total number of rows inserted. This allows us to collect data for

questions such as: “How well does our implementation work with queries that are fulfilled by a small amount of rows vs. a large amount of rows for the same table size?”

5.3 Methodology

For both our implementations, we ran the same types of tests in order to evaluate and compare their performance vs. the default HBase client operations as well as vs. each other.

For all the tests, we warmed up Hadoop and HBase by running the tests once before starting the actual test runs for metrics collection. This ensured that the Java machines were properly warmed up. We configured the heap sizes of the Java machines running Hadoop and HBase to start at their maximum capacity in order to prevent the latency that occurs when they are dynamically adjusted at runtime.

We ran each test a total of three consecutive times, not including the warm-up, and used the average of the collected performance metrics as our result.

5.4 Motivation

The goal of our experiments is to determine the viability of our implementations by determining if the costs of write operations to an indexed table vs. a non-

indexed table are reasonable. Our tests also allow us to verify that the gains in performance for queries leveraging the index are substantial enough to justify the performance hit taken by the write operations.

5.5 Experimental Results

In this section, we will first present the performance evaluation of the Table Based Secondary Indexing implementation we described previously in chapter 3. This will be followed by the performance evaluation of the In Memory Secondary Indexing implementation described in chapter 4.

5.5.1 Evaluation of Table Based Secondary Indexing

5.5.1.1 Writes

For the evaluation of our Table Based Secondary indexing implementation, we first wanted to compare the performance between writing to an indexed table vs. writing to a non-indexed one. At the same time, we also wanted to evaluate the impact of the number of identical entries for a given indexed column on write operation throughput while keeping the total number of writes fixed. More identical occurrences in an indexed column have the consequence to make larger entries in the index table for a given value and we wanted to measure what effect this would have. First we ran the test using only one region server to host

both the user data table and the index table. We then proceeded to run the same test using two region servers, one having the user data table and the other hosting the index table.

For this test, we executed 800 000 writes into the user table having 0.5%, 1%, 3%, 5%, 7% and 10% of identical occurrences of a randomly selected value in the indexed column and collect the average throughput. Each of these series of writes was executed on:

- An indexed user data table having one indexed column out of ten columns using batched write operations;
- A non-indexed user data table, using batched write operations;
- A non-indexed user data table, not using batched write operations;

The results of the tests executed on one region server are shown in Figure 5.2.

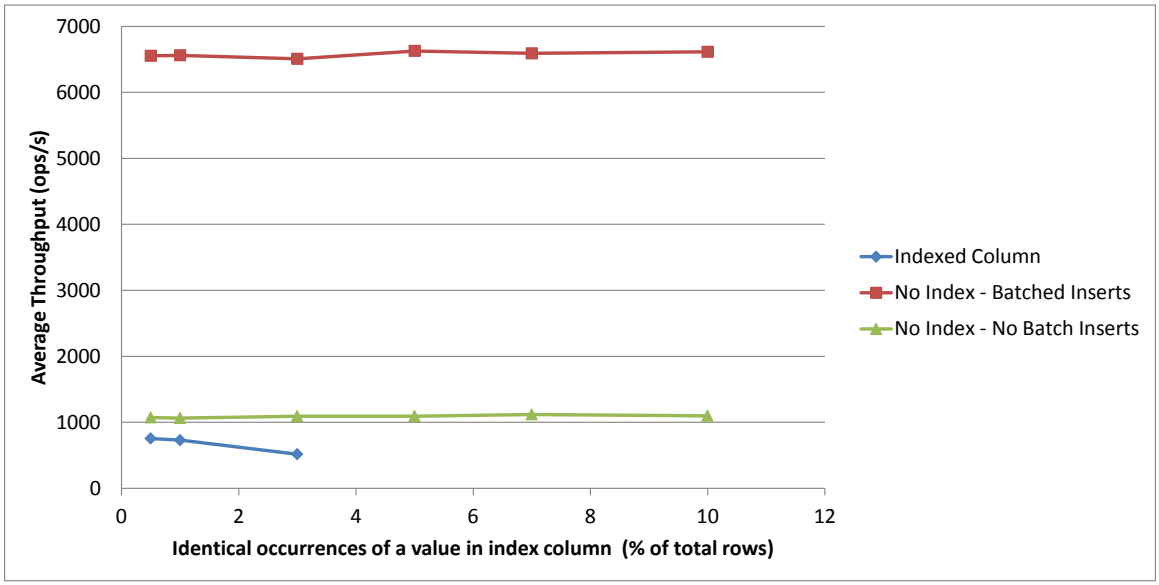

Figure 5.2: Average write throughput vs. identical occurrences – 1 Resource Server

Figure 5.3 below shows the results of the same tests on two separate region servers, one hosting the user data table and the other hosting the index table.

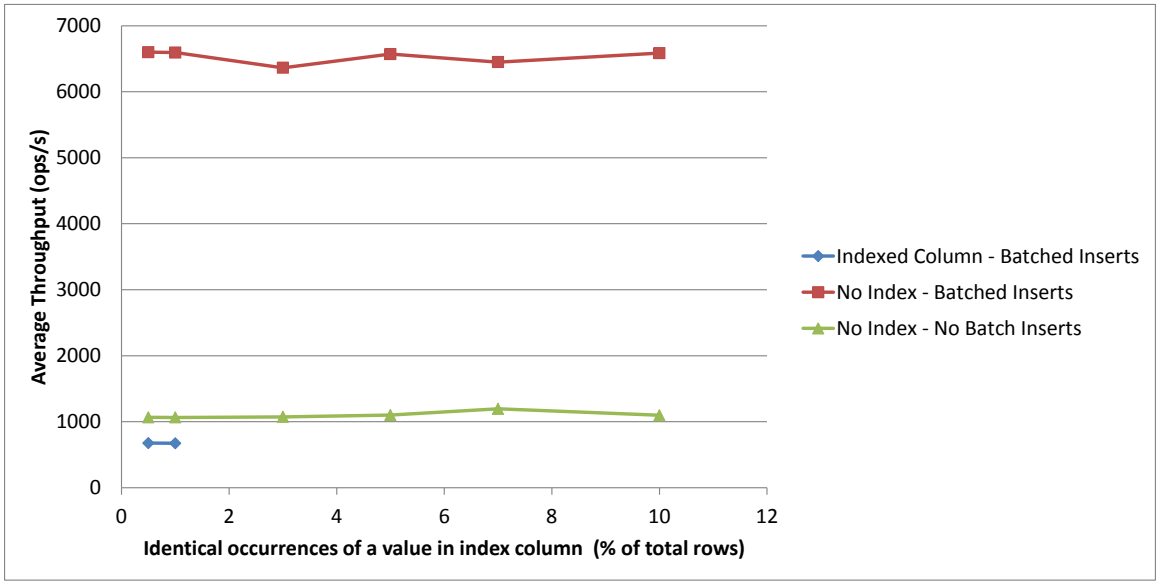

Figure 5.3: Average write throughput vs. identical occurrences - 2 Resource Servers

We can see from Figures 5.2 and 5.3 that the average write throughput performance is greatly degraded when comparing the writes executed within a batch in a non-indexed table vs. an indexed one. The performance degradation is not as noticeable when comparing writes that are not batched; in this scenario, at best, the average write throughput on the indexed table is 80% of the non-indexed one.

The steep performance degradation between batched inserts on an indexed table compared to a non-indexed table can be explained in part by the fact that, due to framework limitation, coprocessors are not able to tell if the operation they

are intercepting from the client is part of a batch process or not. This limitation forces our implementation, for consistency reasons, to update the table containing the index in a non-batched manner. This in turn has the effect that all write operations sent to the indexed table are done outside of the batch process at the region server level even if the client requested them to be done in a batch. When comparing non-batched writes in a non-indexed table vs. batched writes in an indexed table, the small performance degradation is consistent with the overhead to update the index when writing to a table.

We also notice from Figures 5.2 and 5.3 the fact that the number of identical occurrences of an indexed value does have a major impact. The average throughput decreases with the increase of identical indexed values. Moreover, at some point, HBase crashes when the index for a specific value becomes too big. This occurs past 3% of identical occurrences when executed on one region server and past 1% when executed on two region servers. This is due to the fact that our benchmarking tool sends write request very quickly one after another and causes the big index to be re-written at a very fast pace. This, in turn, causes the Hadoop HDFS to run out of resources which has the unfortunate consequence to crash HBase's. It occurs earlier on a two region server setup because of the extra network latency involved in the HDFS file replication

between nodes which causes Hadoop to run out of resources earlier since they are tied up longer. Of course, we simulate an extreme situation where updates to the same row happen quickly. If there isn't a flood of update requests for the big index entry, the problem does not occur; unfortunately HBase on HDFS does not appear to provide a way of throttling that type of request as of version 0.95.

5.5.1.2 Reads

The second type of tests we ran was to see what performance improvements we could get on table queries by using our indexing implementation (described in section 3.4.3) vs. using the default HBase functionality. For a client to get results for a query of type "Get all rows where value in column X is A", HBase uses a filtered scan. This means that, without an index, the entire table is scanned and only the values matching the filter are returned in the scanner to the client, also, the different regions of a table are scanned sequentially.

To test this, we used the indexed tables containing 0.5%, 1%, 3%, 5%, 7% and 10% of an identical value in the indexed column. Note that we first loaded non indexed tables with the data, then created the index using a map-reduce job; the map reduce job involved in the index creation does not cause the resource drain problem described above since index entries are written sequentially and only once (there is no constant rewriting).

We ran queries on these tables of the type “Get all rows where value in the indexed column is *A*”. We set *A* to be the identical value contained in 1.4%, 5%, 7% and 10% of the rows in the indexed column. To these cases, we also added the cases where *A* does not exist or exists only once in the indexed column. We executed the series of tests for the case where all regions, both for the user data table and the index table, are hosted by one Region Server and then where these regions are spread evenly across two Region Servers. We then collected the response times that the queries took to return the full result set.

The results of the test for one Region Server are shown in Figure 5.4 below.

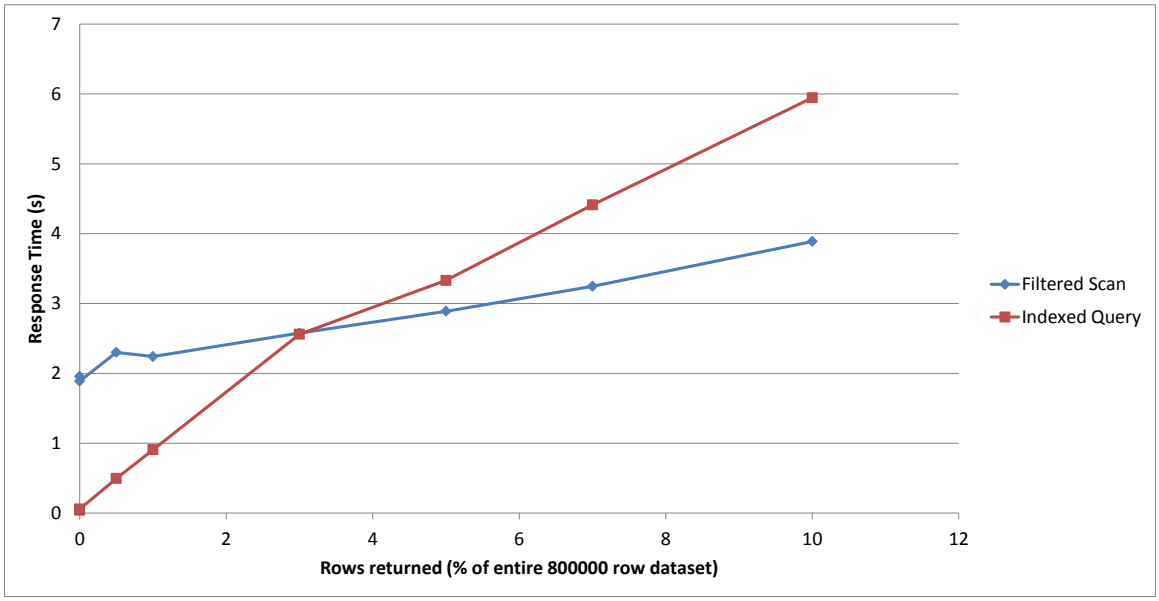

Figure 5.4: Filtered Scan vs. Indexed Query Graph – 1 Resource Server

Table 5.1 below shows a different view of the results collected in a tabular format that includes the relative efficiency of our implementation compared to the filtered

scan method in percentage. A value below 100% means that the indexed version takes longer than a filtered scan.

Occurrence of queried value in indexed columns	Filtered Scan (s)	Get By Index (s)	Relative efficiency
0 times	1.96	0.04	4900%
1 time	1.89	0.06	3150%
0.5% of rows	2.30	0.50	460%
1% of rows	2.24	0.91	246%
3% of rows	2.58	2.56	101%
5% of rows	2.89	3.33	87%
7% of rows	3.25	4.41	74%
10% of rows	3.89	5.95	65%

Table 5.1: Filtered Scan vs. Indexed Query Table - 1 Resource Server

The results of the test executed on two Region Servers are shown in Figure 5.5 below.

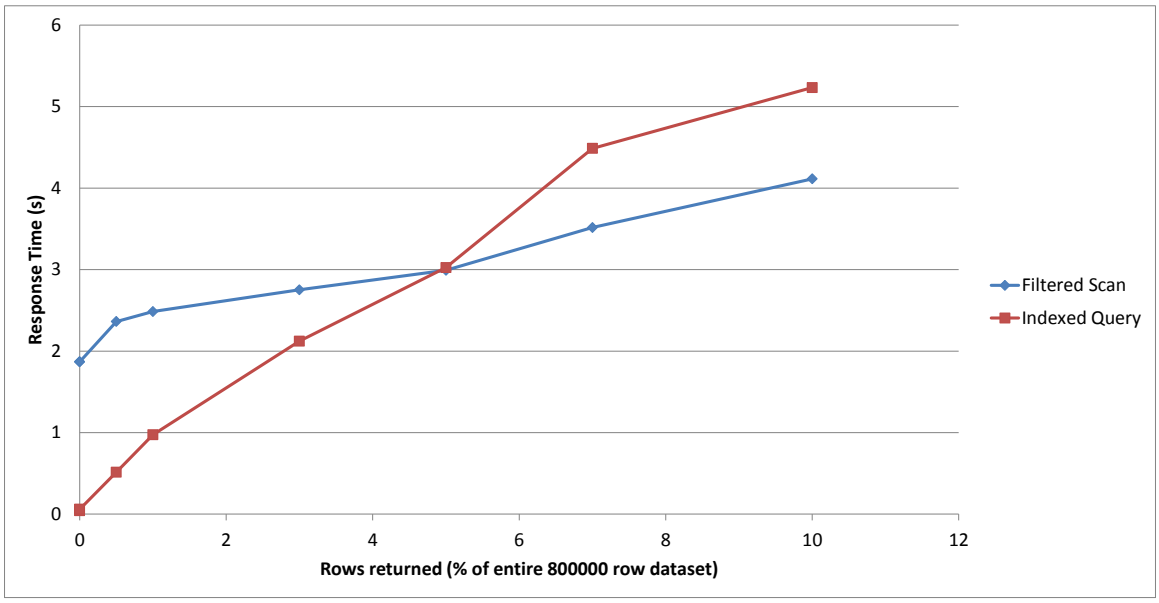

Figure 5.5: Filtered Scan vs. Indexed Query Graph – 2 Resource Servers

Table 5.2 below shows the tabular view, which includes the relative efficiency of our implementation compared to the filtered scan method in percentage of the results collected from the execution of the tests on two Region Servers.

Occurrence of queried value in indexed columns	Filtered Scan (s)	Get By Index (s)	Relative efficiency
0 times	1.87	0.04	4675%
1 time	1.87	0.06	3117%
0.5% of rows	2.36	0.51	463%
1% of rows	2.48	0.97	256%
3% of rows	2.75	2.12	130%
5% of rows	2.99	3.02	99%
7% of rows	3.52	4.49	78%
10% of rows	4.11	5.23	79%

Table 5.2: Filtered Scan vs. Indexed Query Table – 2 Resource Servers

We can see, from Figures 5.4 and 5.5 as well as Tables 5.1 and 5.2, that the performance of the queries is greatly improved when using our indexing implementation. Also, the smaller the amount of rows matching the query, the faster the response time and performance gain is. We notice that past a certain result set size, the indexed query becomes slower than a filtered scan. There are two reasons for that. First, the larger the result set is, the larger the amount of data from the index table that has to be transferred to the time, which costs in

time. Secondly past a certain result set size, a filtered scanner offers a better performance than a series of multiple get operations. A scan benefits from loading the table in large blocks whereas with get operations, each one of them has to first locate the block where the data resides, load the block, and then retrieve the data. When this occurs, the effect may be mitigated by spreading the user data table's regions over multiple region servers to take advantage of the parallel nature of multiple get commands vs. the sequential nature of scans. We can observe that the indexed query performs considerably better one two regions servers compared to having only one region server.

5.5.2 Evaluation of In Memory Secondary Indexing

5.5.2.1 Writes

For the evaluation of our In Memory Secondary indexing implementation, we first wanted to compare the performance between writing to an indexed table vs. writing to a non-indexed one. At the same time, we also wanted to evaluate the impact of the number of identical entries for a given indexed column on write operation throughput while keeping the total number of writes fixed. More identical occurrences in an indexed column have the consequence to make larger entries in the index for a given value and we wanted to measure if this factor would have as much an effect as the one experienced using the Table Based Secondary Indexing.

For this test, we executed 800 000 writes into the user table having 0.3%, 0.5%, 1%, 3%, 5%, 7% and 10% of identical occurrences of a randomly selected value in the indexed column and collect the average throughput. Each of these series of writes was executed on:

- An indexed user data table having one indexed column out of ten columns using batched write operations;
- An indexed user data table having one indexed column out of ten columns using non-batched write operations;
- A non-indexed user data table, using batched write operations;

- A non-indexed user data table, using non-batched write operations;

The results of the test runs are shown in Figure 5.6 below.

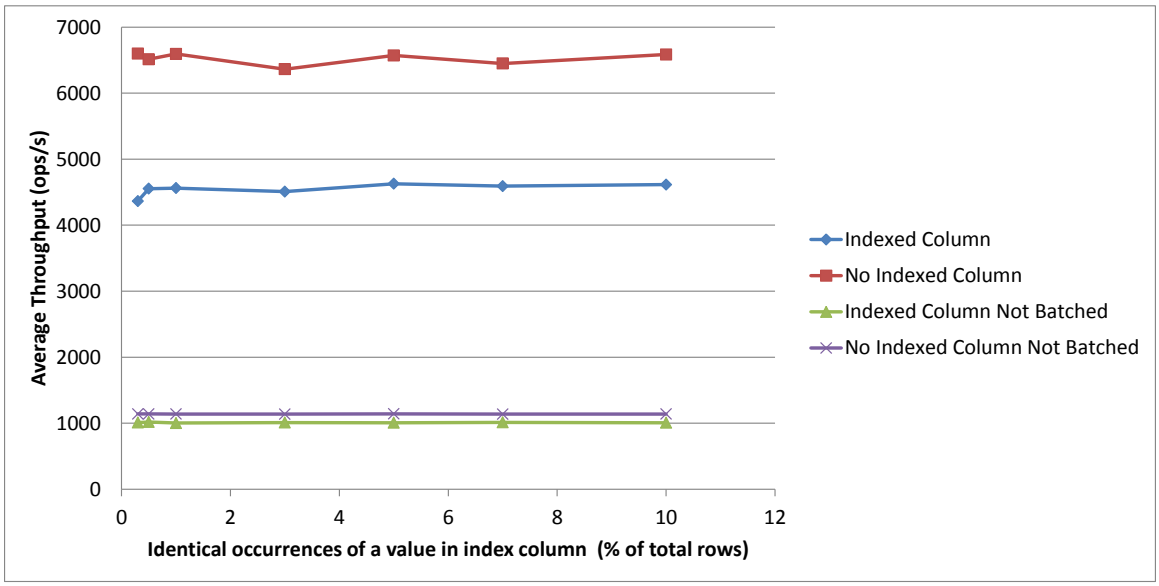

Figure 5.6: Average write throughput vs. identical occurrences

We can see from Figure 5.6 that the average write throughput performance degradation is not so extreme, more so when inserts are not executed as part of a batch. Writes executed against an indexed table have an average throughput of 70% of those executed against a non-indexed table when batched and 89% when not batched. We can also see that the number of identical occurrences of a value in the indexed column has no significant impact to the performance.

The main cause of the performance degradation in this implementation is the local read that occurs for each write involving one or more indexed columns as

described in section 4.2.1.4. When writes are not batched, they are expensive and in comparison the random read's cost is not so significant.

5.5.2.2 Reads

The second type of tests we ran was to see what performance improvements we could get on table queries by using our in memory indexing implementation (described in section 4.3.2.3) vs. using the default HBase functionality.

To test this, we used the indexed tables resulting from the previous test that contained 1.4%, 5%, 7% and 10% of an identical value in the indexed column. We ran queries on these tables of the type "Get all rows where value in the indexed column is *A*". We set *A* to be the identical value contained in 0.5%, 1%, 3%, 5%, 7% and 10% of the rows in the indexed column. To this we also added the case where *A* does not exist in the indexed column. We then collected the response times that the queries took to return the full result set. Due to the highly parallelizable nature of the indexed queries, we ran the series of tests in scenarios where the user data table's regions were hosted either on one or two region servers.

The results of the test executed in the scenario where all the user data table's regions were hosted on one region server are shown in Figure 5.7 below.

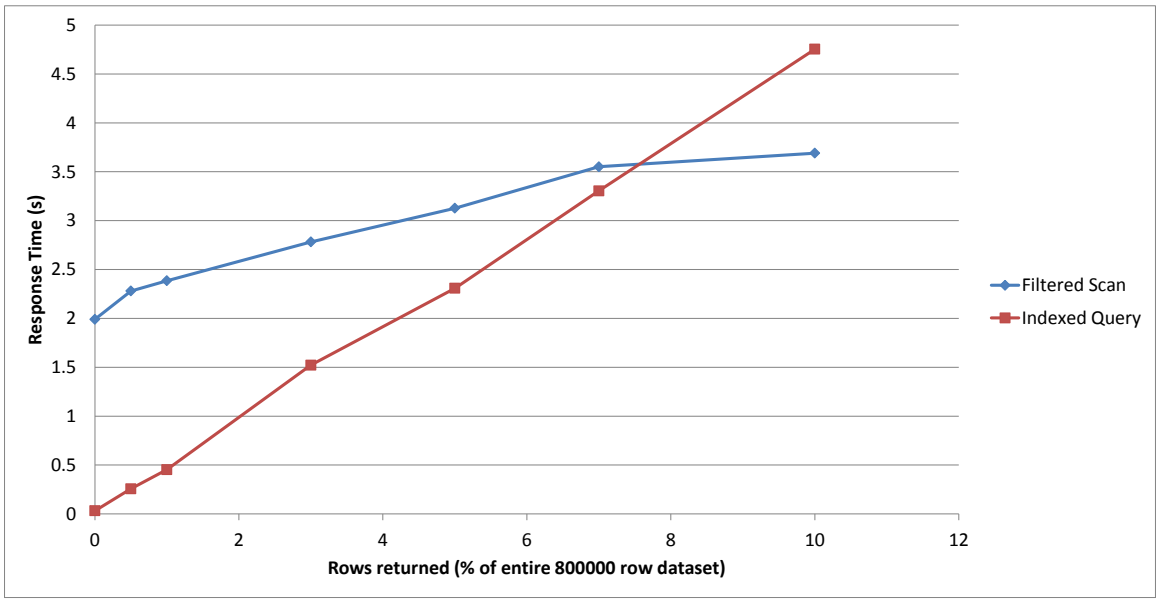

Figure 5.7: Filtered Scan vs. Indexed Query – 1 Resource Server

The same results in tabular format showing the relative efficiency between the Filtered Scan and the Indexed Query are shown in Table 5.3 below.

Occurrence of queried value in indexed columns	Filtered Scan (s)	Get By Index (s)	Relative efficiency
0 times	1.99	0.03	6633%
0.5% of rows	2.28	0.26	877%
1% of rows	2.39	0.45	531%
3% of rows	2.78	1.52	182%
5% of rows	3.12	2.31	135%
7% of rows	3.55	3.30	108%
10% of rows	3.69	4.75	78%

Table 5.3: Filtered Scan vs. Indexed Query Table – 1 Resource Server

The results of the test executed in the scenario where all the user data table's regions were hosted on two region servers are shown in Figure 5.8 below.

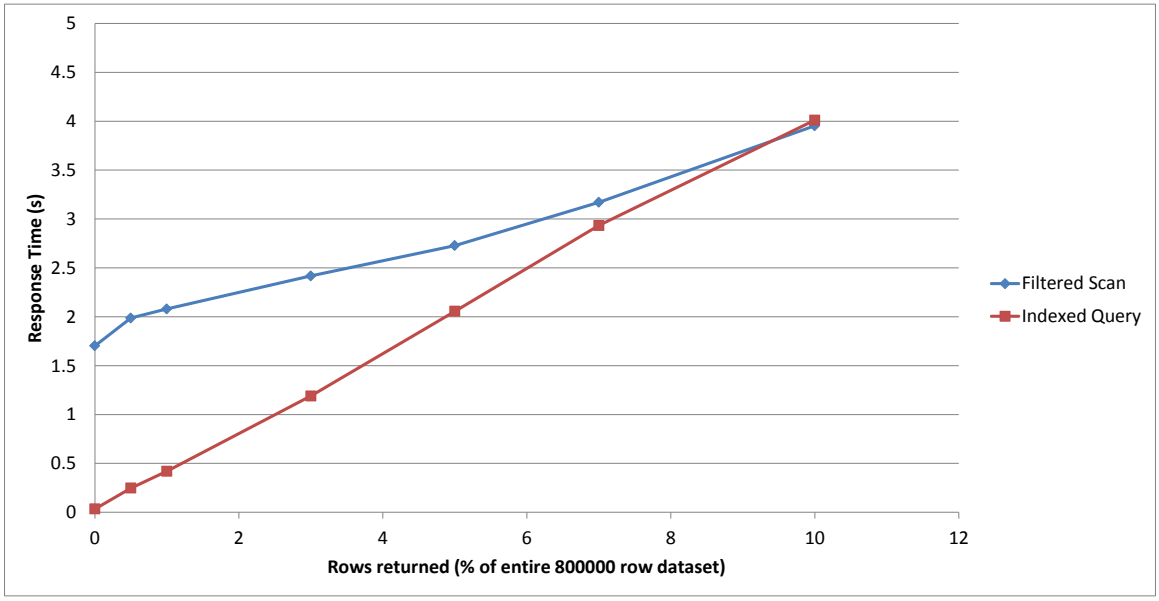

Figure 5.8: Filtered Scan vs. Indexed Query – 2 Resource Servers

Again, the same results in tabular format showing the relative efficiency between the Filtered Scan and the Indexed Query are shown in Table 5.4 below.

Occurrence of queried value in indexed columns	Filtered Scan (s)	Get By Index (s)	Relative efficiency
0 times	1.70	0.03	5667%
0.5% of rows	1.99	0.25	796%
1% of rows	2.08	0.42	495%
3% of rows	2.42	1.19	203%
5% of rows	2.73	2.06	134%
7% of rows	3.17	2.93	108%
10% of rows	3.95	4.01	99%

Table 5.4: Filtered Scan vs. Indexed Query Table – 2 Resource Servers

We can see, from Figures 5.7 and 5.8 and Tables 5.3 and 5.4, that the performance of the queries is greatly improved when using our indexing implementation. Also, the smaller the amount of rows matching the query, the faster the response time and performance gain is. Past a certain result set size, a scanner offers a better performance than a series of multiple get commands; this causes the indexed query to become slower than a filtered scan. When this occurs, the effect may be mitigated by spreading the user data table's regions over multiple region servers to take advantage of the parallel nature of multiple get commands vs. the sequential nature of scans. We can observe the scalability

of the indexed query from the results gathered from tests executed using one region server versus the ones executed using two region servers.

These results were what we expected because when using the indexed query, the table does not need to be fully scanned to retrieve the matching rows and the operations of the indexed query are executed in parallel if the data matching the query happens to be distributed among multiple regions.

5.5.3 Comparison of our implementations

Both our implementations offer good performance increase when running queries that take advantage of the secondary index although the in memory indexing implementation offers the best performance of the two. This can be explained by the fact that, in the case of the in memory secondary indexing, the client does not need to first receive a set of row ids as the first step before sending the requests to get the rows for these row ids. This saves on network latency and the overhead of querying two tables instead of just one.

In the case of write operations, our in memory implementation is far superior in terms of performance to our table based one and provides good performance even when dealing with batch modifications. Moreover, our in memory implementation does not cause HBase to crash because the only time the index is written to HDFS is when a region is closed or split.

Overall, our in memory implementation offers the best performance and reliability which makes it the most viable solution. The downside is that one must take into account the extra RAM needed to host the index when doing the sizing for region servers.

Chapter 6

Conclusion and Future Work

6.1 Conclusions

Distributed Database Systems are becoming more pervasive with the emergence of big data and cloud computing. Some of these distributed database systems are within the open source realm and are still in their infancy with respect with the features they offer.

In this thesis, we presented two implementations of a secondary indexing feature for HBase, a table based implementation which uses HBase tables to store indices and an in memory implementation were the indices are stored in main memory.

We analyzed our implementations and compared them between each other as well as with HBase's default mechanism for running queries. Both our implementations offered a marked improvement on read queries based on a secondary column value up to a certain point where scanning the entire table became more efficient. We found that for both implementations, we could push this point further away by partitioning the tables containing indexed columns into more regions in order to take advantage of the parallelism our query mechanism exploits. This fits perfectly with HBase's method for providing scalability which is to split tables regions as they grow and distribute them across region servers. Finally, we confirmed that our in memory implementation offered the best read performance gain to write performance degradation ratio and was the most reliable solution.

6.2 Future Work

In this thesis, we did not implement any fault tolerance for the data modification operation interceptions (*Put* and *Delete*). That means that if a modification to a table fails, the update to the index will not be rolled back and it will contain what should have been modified in the actual user data table. Both implementations for secondary indexing may be updated to take advantage of a planned feature for the HBase's Coprocessor framework (HBase Project's JIRA ref: HBASE-5827) which will provide error handlers for operations that perform writes. As it is now, there is no straightforward and efficient way to apply rollbacks either on a modification to the table after a failure to update the index or a rollback on the index after a failure to update the table.

Neither of our implementations supports multiple versions of the data: only the latest version of a given cell is stored in the index. Some applications may need to use the multiple version features HBase provides and as such, it could be a beneficial addition to our implementations.

Bibliography

[1] Ronald C. Taylor. (2010, December) Taylor: An Overview of the Hadoop/MapReduce/HBase Framework and its Current Applications in Bioinformatics. PDF Document. [Online]. http://www.biomedcentral.com/1471-2105/11/S12/S1

[2] Byunggu Yu, Alfredo Cuzzocrea, and Dong Jeong, "On Managing Very Large Sensor-Network Data Using Bigtable," in *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Ottawa, ON , 2012, pp. 918 - 922.

[3] Alfredo Cuzzocrea, Il-Yeol Song, and Karen C. Davis, "Analytics over Large-Scale Multidimensional Data: The Big Data Revolution!," in *ACM 14th international workshop on Data Warehousing and OLAP (DOLAP)*, New York, NY, 2011, pp. 101-104.

[4] Dhruba Borthakur, Joydeep Sen Sarma, Jonathan Gray, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer, "Apache Hadoop Goes Realtime at Facebook," in *ACM Special Interest Group on Management of Data (SIGMOD)*, Athens, Greece, 2011, pp. 1071-1080.

[5] Divyakant Agrawa, Sudipto Das, and Amr El Abbadi, "Big Data and Cloud Computing: Current State and Future," in *ACM 14th International Conference on Extending Database Technology (EDBT/ICDT)*, New York, NY, 2011, pp. 530-533.

[6] Dhruba Borthakur, "Petabyte Scale Databases and Storage Systems at Facebook," in *ACM Special Interests Group on Management of Data (SIGMOD)*, New York, NY, 2013, pp. 1267-1268.

[7] Denice Deatrich, Simon Liu, Chris Payne, Réda Tafirout, Rodney Walker,

Andrew Wong, and Michel Vetterli, "Managing Petabyte-Scale Storage for the ATLAS Tier-1 Centre at TRIUMF," in *High Performance Computing Systems and Applications (HPCS)*, vol. 28, Quebec City, Que., June 2008, pp. 167-171.

[8] Fay Chang, Sanjay Ghemawat, Wilson C. Hsieh, and Deborah A. Wallach, "Bigtable: A Distributed Storage System for Structured Data," in *Symposium on Operating System Design and Implementation (OSDI)*, Seattle, WA, 2006.

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," in *ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, 2007.

[10] Apache Software Foundation. (2013, Jan.) HBase - Apache HBase Home. [Online]. http://hbase.apache.org/

[11] Avinash Lakshman and Prashant Malik, "Cassandra - A Decentralized Structured Storage System," *ACM Special Interest Group on Operating Systems (SIGOPS) Operating Systems Review*, vol. 44, no. 2, pp. 35-40, April 2010.

[12] Lars George, *HBase: The Definitive Guide*. Sebastopol, California, United States of America: O'Reilly, 2011.

[13] Edgar F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM (CACM)*, vol. 13, no. 6, pp. 377-387, June 1970.

[14] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed, "ZooKeeper: Wait-Free Coordination for Internet-Scale Systems," in *USENIX Conference on USENIX Annual Technical Conference*, Boston, MA, USA, 2010, pp. 11-11.

[15] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine,

and Daniel Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *ACM Symposium on Theory of Computing (STOC)*, El Paso, TX, USA, 1997, pp. 654-663.

[16] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Protocol," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17-32, 2003.

[17] Eric A. Brewer, "Towards Robust Distributed Systems (abstract)," in *ACM Symposium on Principles of Distributed Computing (PODC)*, Portland, OR, USA, 2000, p. 7.

[18] Seth Gilbert and Nancy Lynch, "Brewer's Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services," *ACM Special Interest Group on Algorithms and Computation Theory (SIGACT) News*, vol. 33, no. 2, pp. 51-59, 2002.

[19] Kyle Banker, *MongoDB in Action*. Shelter Island, NY: Manning Publications Co., 2012.

[20] Crockford Douglas. (2006, July) RFC 4627 - The application/json Media Type for JavaScript Object Notation (JSON). [Online]. http://tools.ietf.org/html/rfc4627

[21] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis, "An efficient indexing technique for full-text database systems," in *18th International Conference on Very Large Databases (VLDB)*, Vancouver, 1992, pp. 352-362.

[22] Oracle. (2011, Dec.) Java™ Platform, Standard Edition 6 API Specification. [Online]. http://docs.oracle.com/javase/6/docs/api/java/util/TreeSet.html

[23] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, USA, 2004, pp. 10-10.