UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

FELIPE MATHIAS SCHMIDT

Change Data Capture Solutions for Apache Cassandra

Thesis presented in partial fulfillment
of the requirements for the degree of
Bachelor of Computer Science

Prof. Dr. Alberto Egon Schaeffer Filho
Advisor

M. Sc. Yong Hu
Coadvisor

Porto Alegre, July 20th, 2014

CIP – CATALOGING-IN-PUBLICATION

Felipe Mathias Schmidt,

Change Data Capture Solutions for Apache Cassandra /

Felipe Mathias Schmidt. – Porto Alegre: Graduação em Ciência da Computação da UFRGS, 2014.

66 f.: il.

Monograph – Universidade Federal do Rio Grande do Sul. Curso de Bacharelado em Ciência da Computação, Porto Alegre, BR–RS, 2014. Advisor: Alberto Egon Schaeffer Filho; Coadvisor: Yong Hu.

1. Change Data Capture. 2. Apache Cassandra. 3. Hadoop MapReduce. 4. HDFS. 5. Big Data. I. Filho, Alberto Egon Schaeffer. II. Hu, Yong. III. Título.

"Every discovery in pure science is potentially subversive; even science must sometimes be treated as a possible enemy. Yes, even science." - Aldous Huxley, Brave New World

ACKNOWLEDGMENTS

First of all I need to thanks my family, specially my mother Irenita and my father Eugênio. They support me in whatever decision I need to do, pushing me always forward. The one thing that is always present in our home is the education; not just scholar and academic education, but also moral education. Without it, I would probably not have studied so much. We descend from very simple people and, although sometimes the life was not so easy, my parents always found ways to teach me valuable lessons. My entire family have a great influence on me, I would never be where I am without them.

Second, the Universidade Federal do Rio Grande do Sul (UFRGS) was extremely important to me. All the good Professors, the course colleges and the infrastructure of this university made possible a huge personal grown during the undergraduate bachelor course. Here I made friends, increased my knowledge about Computer Science and discovered many gears that moves the world.

My exchange to Technische Universität Kaiserslautern (TU-KL) during a year made possible this work, with a big help from M. Sc. Yong Hu. Also important, the exchange made me see how different cultures are and how an amazing first world country works. This experience changed my life and I am very grateful for this opportunity given by Prof. Dr. Stefan Deßloch and the TU-KL team. I hope that the collaboration between UFRGS and TU-KL remains for many more years, so more students can have the same chance as I had.

In the last year, Prof. Dr. Alberto helped me a lot. His knowledge and patience made this part of the bachelor course less stressful. Also, the students from the Parallel and Distributed Processing Group (GPPD) and Prof. Dr. Geyer, which allowed me to use their infrastructure and helped me to run my tests.

Finally, thanks to all my friends that one way or another helped me in whatever situation. Specially to Felisberta, Luiz Vogt; without them, the change from country side to Porto Alegre would be almost impossible.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Apache Cassandra is a powerful NoSQL database. Its implementation provides a high performance database, also aiming high scalability. In the same manner, the Hadoop MapReduce framework provides a highly scalable API for parallel and distributed computation. All in a transparent way to the programmer. Change Data Capture (CDC) solutions are capable of speeding up services that track modifications in a source database, passing the changes to a target database. In this context, we discuss in this thesis several techniques for extracting data that has changed in a source database; later on, making the changes available for use at a target database. The techniques use MapReduce to implement their logics and also to interact with the source database Apache Cassandra. The same API stores the results in Hadoop Distributed File System (HDFS). All technologies are for distributed and/or parallel environments, e.g., clusters. The proposed techniques are designed to work in this scenario, with the best possible performance.

Keywords: Change Data Capture, Apache Cassandra, Hadoop MapReduce, HDFS, Big Data.

Soluções de Change Data Capture para Apache Cassandra

RESUMO

O Apache Cassandra é um banco de dados NoSQL poderoso. Sua implementação provê um banco de dados de alta performance, visando também alta escalabilidade. Da mesma forma, o framework Hadoop MapReduce fornece uma API altamente escalável para computação paralela e distribuída. Tudo de uma forma transparente para o programador. Soluções de Change Data Capture (CDC) são capazes de acelerar serviços que monitoram modificações em um banco de dados fonte, passando as mudanças para um banco de dados destino. Neste contexto, nesta tese discutimos diferentes técnicas para extrair dados que foram alterados em um banco de dados fonte, posteriormente disponibilizando as mudanças para uso em um banco de dados destino. As técnicas usam MapReduce para implementar suas lógicas e interagir com o banco de dados fonte Apache Cassandra. A mesma API armazena os resultados no Sistema de Arquivos Distribuídos do Hadoop (HDFS). Todas tecnologias são para ambientes distribuídos e/ou paralelos, e.g., clusters. As técnicas propostas são projetadas para atuar neste cenário, com a melhor performance possível.

Palavras-chave: Change Data Capture, Captura de Dados Alterados, Apache Cassandra, HDFS, Hadoop MapReduce, Big Data.

1 INTRODUCTION

This chapter gives an introduction for this work. This thesis has the objective of designing, implementing and testing several different methods to capture changed data in a specific source system. In the following sub-sections the motivation, objective and outline of this document are given.

1.1 Motivation

The costs of maintaining and managing relational databases are high. Even on a small scale, this is a very significant job, as stated in (RAMANATHAN; GOEL; ALAGUMALAI, 2011). To solve this issue, highly available databases at massive scales have begun to be developed. These databases aim also at providing reliability, *"because even the slightest outage has significant financial consequences and impacts customer trust"*, as extracted from (DECANDIA et al., 2007). One example of this type of database is a solution proposed by Amazon called Dynamo (DECANDIA et al., 2007). Another solution for distributed storage system, with a high-performance and high scalability is Google's BigTable (CHANG et al., 2006), which is also highly available, handling data from several Google products. Inspired by these first research efforts, new databases have emerged. One of them is Apache Cassandra, which is used in many companies, handles large amounts of data every day and has a growing user base. Another relevant aspect is that, differently from relational databases that are table oriented, Cassandra is a column oriented storage solution.

This scenario is perfect to create and innovate. There are still concepts that can be explored, with possibility of creation of new features on top of these databases. As example, CDC techniques, which are already used in relational databases. Such mechanism might also boost the performance of column oriented databases for several applications. In particular, capturing changed data is a necessary basic feature. This feature is used to maintain materialized views, keep data warehouses up to date and even to create complex incremental applications as proposed by (SCHILDGEN; JöRG; DEßLOCH, 2013). Although techniques for capturing changed data have been studied for several years in the relational area, there are many barriers to be transposed while translating the same idea to column oriented databases. There are many different column oriented databases currently available, including MongoDB[1], Apache HBase [2] and Apache Cassandra [3].

[1] http://www.mongodb.org/ accessed 22-June-2014
[2] http://hbase.apache.org/ accessed 22-June-2014
[3] http://cassandra.apache.org/ accessed 22-June-2014

We chose to work with Cassandra because it is largely used, has an active and increasing community and also is still being developed, with many new features added to the system very frequently. Furthermore, as proposed by (HU; DESSLOCH, 2013), it is possible to efficiently implement Change Data Capture solutions for NoSQL databases. Undoubtedly, with such feature the use of these storage systems increases, mainly because it will provide an improved service, covering more scenarios of possible use.

In the work presented in this thesis, the MapReduce framework is used to implement the logic of the methods. This framework can integrate with Apache Cassandra, enabling a transparent parallel computation, which is very powerful in multi-nodes clusters. HDFS also integrates with MapReduce, creating an easy to use distributed file system. Both technologies are part of Apache Hadoop, which also has a column oriented database solution, called HBase.

This thesis is based on the work proposed in (HU; DESSLOCH, 2013). The methods described here are derived from the last referenced paper, but applied specifically in Apache Cassandra. To the best of our knowledge, there is no other attempt to create change data capture solutions to Apache Cassandra. There are general models as described in chapter 3, but no one is build to fit specifically the base system chosen for this thesis.

1.2 Objective

The main objective of this work is to model and implement change data capture solutions for Apache Cassandra. Although some solutions presented in this work already exists for relational databases, they cannot be straightforwardly applied in this system. The goal is to adapt these methods to match the details of the NoSQL database.

We want to build robust models, implementing and integrating them with Hadoop MapReduce, HDFS and Cassandra. The models must take care of all details existent in each technology, allowing them to work together in an efficient and simple way.

Furthermore, this work compares the methods, describing their pros and cons, considering also the performance observed while executing each of the methods in a given test case. Finally, a conclusion outlines relevant aspects discussed in this document, summarizing important information.

1.3 Contribution

The contributions of this thesis are:

- Create a delta model to Apache Cassandra. The deltas must be able to identify the performed operation, as well as the location where the modification occurred.
- Build and describe different techniques of CDC, exploring several possibilities to do so.
- Apply CDC approaches to the base system. For such, implementations of the different techniques with the ability to communicate with Cassandra and MapReduce are required.

- Compare the techniques. An evaluation of the techniques, comparing their functionalities, performances and limitations.

1.4 Outline

In Chapter 2 the base environment is described, detailing important and relevant aspects that must be considered while implementing the change data capture methods. In this part, necessary details of all the used systems are explained. In Chapter 3, models applied to Cassandra data and working model are specified, taking care of the previously mentioned aspects, and giving all the details necessary for a good comprehension of them. Chapter 4 lists the strengths and the weaknesses of the approaches, comparing also their performances. Chapter 5 shows the existent methods, applied in RDBMS or in other column oriented databases. Finally, Chapter 6 gives a conclusion for this document, describing also possible future works.

2 BACKGROUND: CASSANDRA, MAPREDUCE AND HDFS

This chapter aims to give the necessary background for a good understanding of the work proposed. Most of the information presented here refers to characteristics of the technologies used. In particular, we describe Cassandra, which is a NoSQL database solution for large scale data volume. Further, we describe an API for parallel and distributed data processing called MapReduce, which integrates with Cassandra and HDFS. Finally we present HDFS, which is a distributed file system from the Apache Hadoop project that integrates with MapReduce. Our goal is to show just the relevant aspects of these tools for this work, without considering unnecessary details. Further details can be found in the references provided.

2.1 Apache Cassandra, a NoSQL solution

To fulfill the requirements of increasingly high data volume systems, a new concept of database called NoSQL has been proposed, being nowadays widely used for many large-scale applications e.g, Facebook, Amazon and Google Earth. This new type of database has not come to replace the popular RDBMS. Instead, it is intended to provide different characteristics for applications that would take advantages of them, serving as a complementary storage solution.

Apache Cassandra is one of the projects that follows this concepts of data storage, the so called NoSQL - an abbreviation for Not Only SQL rather than No SQL as could be interpreted. This term is designated for databases that do not follow the relational data model and do not have a SQL language. Although there is no standardized definition of a NoSQL database, it normally provides a key-value, highly available, schema free and highly scalable storage system. These are the main characteristics that distinguish NoSQL from RDBMS. Some programs that follow this structure are Google's Big Table (CHANG et al., 2006) and Amazon's Dynamo (DECANDIA et al., 2007); in the open source space we have HBase (GEORGE, 2011), MongoDB (CHODOROW; DIROLF, 2010) and Cassandra (HEWITT, 2011) as some of the most popular solutions.

2.1.1 Data Model

In order to build the data model, we must know how the data is structured into Cassandra. The structure is divided in six different concepts: Keyspace, Column Family, Super Column, Column, Key and Value. Each one will be explained in the following paragraphs.

The Keyspace has a similar meaning as a database in the RDBMS world. The goal of this structure is to make it possible to have unrelated data stored in the same instance of Cassandra without mixing them. Going one level deeper, we have the Column Family, that acts as a Table in the RDBMS, storing correlated data; inside this structure we can have Columns or Super Columns. The Columns - also named Standard Columns when not under a Super Column - have the same meaning of columns in RDBMS, working as a sort of label for the stored data. Super Columns are used as a container of columns, used to bring closer columns with similar meanings, without any equivalent concept in the relational databases; columns stored under a super column can be also named Sub Columns. Cassandra's key can be thought of as an ID, that identifies uniquely our data. Finally, the value is the data itself. Notice that one Column Family in Cassandra can carry data using standard column structure or super column with sub columns structure; these two structures never appear together under the same column family. Figure 2.1 structures all these concepts together.

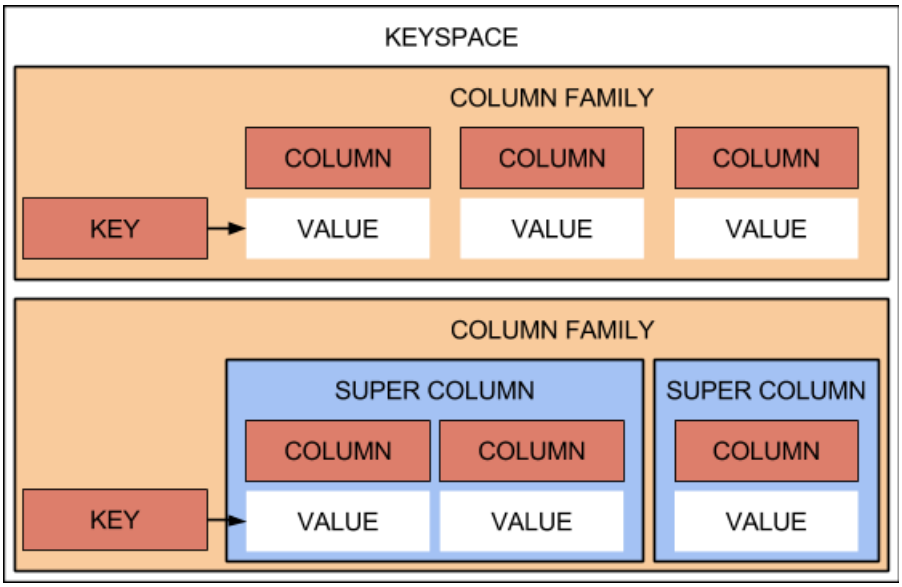

Figure 2.1: Column Family structure for Super and Standard Column family

In the following, we illustrate an example of data structure using the aforementioned data model. Suppose one is interested in storing information of some school, so a generic name for our keyspace would be 'School'. For a particular school, it is necessary to store the grades for all the students, therefore we can create a column family named 'Grades' that will have super columns to distinguish semesters and sub columns to separate the grades of different courses. Considering we will never have more than one student with the same name, we can use the students names as column family keys. Figure 2.2 shows in a better fashion the modeled data.

Notice that this abstraction of 'table' is not entirely compatible with Cassandra's data model. The reason for this is because if we do not have any data in a specific column we also do not have the column structure; differently from relational databases where we will have the column structure anyway with some null value under it. For instance, in a relational database for certain table there will be the same column structures for all keys stored in a given table. But Cassandra's schema free data model allows keys under the same column family (similar to a table) with different column structures for each key or

	Keyspace: School			
	Column Family: Grades			
	Super Column: First Semester		Super Column: Second Semester	
	Mathematic	History	Mathematic	History
Albert	85	65	90	60
Bertha		58		
Carol			89	75
George				57

Figure 2.2: Cassandra Table Example

even without the column structure for a specific column and a certain key. Because of this very same reason, every time we mention some similarity with the relational world, it does not mean both concepts are equal, but that they share some of the main ideas.

Finally, in Cassandra the timestamp is uniquely used to resolve data conflicts when reading the data rather than keeping multiple data versions, like other NoSQL systems such as HBase.

2.1.2 Operational Model

As extracted from (HEWITT, 2011), we can briefly describe Cassandra as a NoSQL, key-value, schema free, highly available, peer-to-peer distributed, no single point of failure, with customized consistency levels storage solution. In this section we will explain all the necessary and important details of Cassandra Operational Model. The aspects that will be further explained have impacts in the way our change data capture approaches will take place, some of them being definitive for the choice of one or other way to develop the methods.

A data center with Cassandra is normally constituted by several computers, called nodes, that are connected to each other in a ring structure, where all nodes have equal behavior and all of them can handle any read or write request. Each node of the ring is responsible for keeping the data of a range of keys. These keys can be translated with a hash map function, e.g MD5 that guarantees the nodes will be well balanced, or defined by the system administrator with specific configuration. Cassandra also implements solutions called Anti-Entropy, Hinted Handoff and Read Repair, responsible for avoiding different data versions across the ring and for recovering and maintaining data in case of node crash or failure. These aspects will not be further explored because they are not relevant in the context of this work.

Tunable consistency level is an aspect that deserves further discussion. This parameter must be set when either a write or a read operation is performed in the system, in each case with a different meaning. As expected, it defines in how many nodes the data must be read or written; the possible values are: One, Two, Three, All, Any and Quorum, such parameters are also discussed at (COULOURIS et al., 2011). Table 2.1, modified from cassandra documentation (HEWITT, 2011), gives further details for a better understanding of the differences between each of them in both possible data operations. For the CDC approaches, the read consistency is more important: the weaker the consistency level used is, the higher is the possibility of processing old data. The choice of which level of consistency to use is up to the application and its system administrator.

Cassandra supports different mechanisms to retrieve and manipulate data, which are

Level	Description:Read	Description: Write
ANY	Not possible.	Data must be written to at least one node. If the nodes responsible for the key are down, the write can still succeed once a hinted handoff has been written. If nodes down, the data is unreadable until the nodes have recovered.
ONE	Return a response from the closest replica.	Data must be written to at least one replica node.
TWO	Return the most recent data from two of the closest replicas.	Data must be written to at least two replica nodes.
THREE	Return the most recent data from three of the closest replicas.	Data must be written to at least three replica nodes.
QUORUM	Returns the most recent data after a quorum of replicas has responded.	Data must be written to at least the most of the replica nodes.
ALL	Returns the most recent data after all replicas has responded. It fails if a replica does not respond.	Data must be written to at least all replica nodes for the row key.

Table 2.1: Cassandra Consistency Levels

the Client API, Thrift API, Avro, MapReduce and some higher level API's. For our purposes the Thrift API and MapReduce are the desirables ways of interaction: the MapReduce framework is used mainly for reading data from Cassandra and the Thrift API is used either to read and write data from the database.

In the following we detail the possible operations to be performed in the database. In a developer side we have *create* and *drop*, performed under keyspaces and column families. These operations are not relevant to the work presented in this document, and their meanings are self-explanatory, therefore we will look at the client side operations. Data manipulation and retrieve are performed using *get*, *del* and *set* commands. The first two are applicable in any desirable granularity, namely key, column or super column; the second, just in column granularity. In these operations we need also to set as a mandatory parameter the consistency level. This one is preferable to be set to ALL when reading data for analysis, so the system guarantees that we are reading the latest data version.

The trigger concept will also be used. Cassandra's community is currently investigating how to add triggers to the system. However, what is currently available is either an unstable patch for an older version of Cassandra or a trigger implementation in Cassandra version 2.0.X, which is also not stable and not in its final version. Both implement an asynchronous trigger, which is similar to an after trigger of the relational databases, as covered in (GROFF; WEINBERG, 2010). The main difference is on the characteristic of being asynchronous, i.e without being performed straight after the event that makes it be fired. Even though triggers are not in its final and stable version, it is at least usable for research purposes, also helping to find errors and bugs. Because of its usability, this feature will be certainly added to the system in the future. Furthermore, for real applications

it is not recommended the use of this patch precisely because of its instability.

Although Cassandra has several other complex mechanisms to keep the system healthy and working and also some other important details of its architecture, we will not discuss them in further details because they are not mandatory for the work presented in this document.

2.2 Introduction to MapReduce

As described in (DEAN; GHEMAWAT, 2004), *"MapReduce is a programming model and an associated implementation for processing and generating large data sets."*. The provided API that implements such model is supported by Cassandra, and it is a good choice of use while processing large data sets because of its facility of use and scalability. In the very same paper, a better definition of what the model actually does is provided, i.e *"Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key"*.

We chose this model for two reasons: it is a largely used distributed way of computation and it has an API fittable into Cassandra. As described in the last paragraph, we have map and reduce functions, those functions run in parallel and distributed across the cluster. This can normally be used to speed up the processing of large amounts of data. Implementation details are not relevant for this work. However, it is important to know that MapReduce is a parallel and efficient way of processing data and for this reason we chose it.

2.3 Introduction to HDFS

HBase, which is also a NoSQL database, is build on top of HDFS[1]. As the MapReduce framework, it is also a part of the Apache Hadoop project, which offers many solutions for distributed storage and computation systems. Because of the easy way to use this file system and also the integration with the MapReduce framework, we chose it to store the output deltas or intermediary results in the approaches. However, it is not necessary to use this very same file system when building one of the methods for a real application. Further, HDFS can run in the same base system that our Cassandra instance is running, without causing any incompatibility problem. Finally, it is also an open source project like Cassandra.

Given that we are dealing with a distributed database, it is reasonable to use also a distributed file system to store outputs. Another remarkable detail is the integration with the MapReduce framework, supporting an easy interaction between them. Figure 2.3 shows the data flow between the technologies described in Chapter 2. When a reducer phase is not present in the described technique, the mapper is who sends the data to HDFS. For the Snapshot Differential technique described in Section 3.4.3, in the differential phase, the input data is taken from HDFS rather than from Apache Cassandra. Finally, for the Log Based technique presented in Section 3.4.4 the input data is the commit log.

[1] http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html accessed 22-June-2014

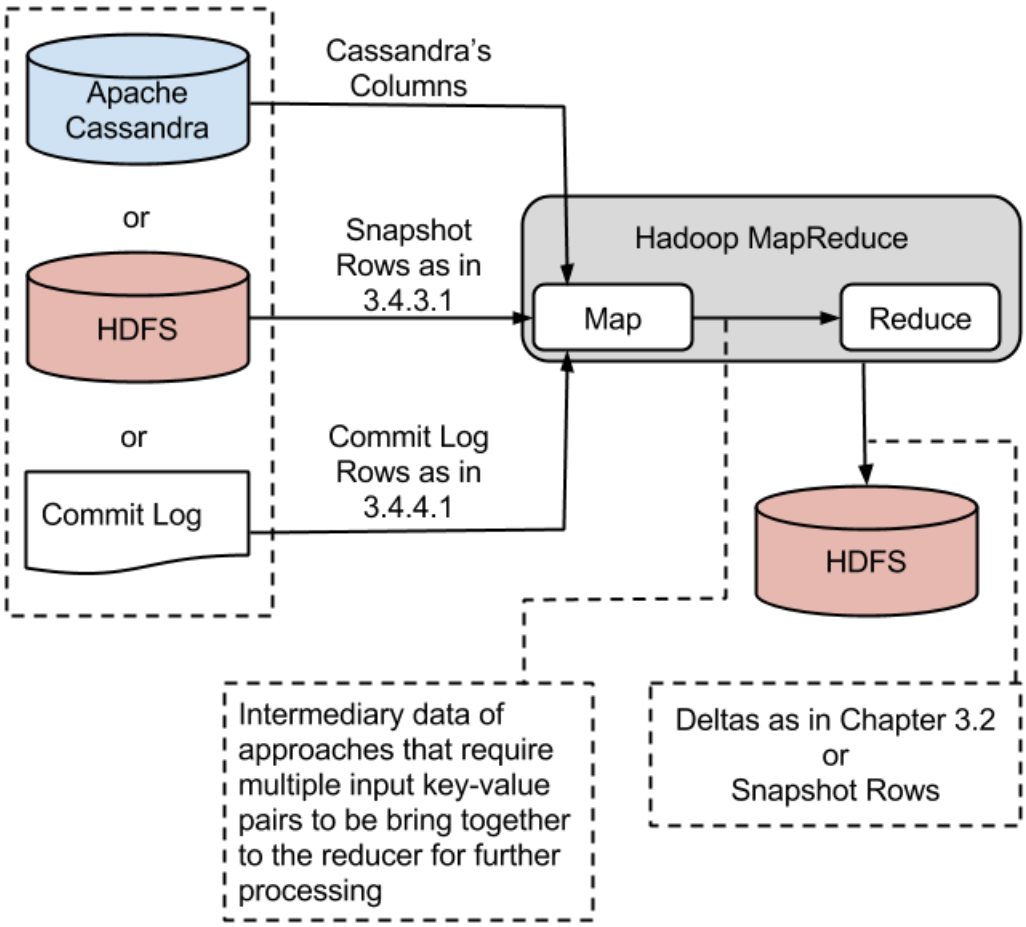

Figure 2.3: Data Flow between Cassandra, MapReduce and HDFS

3 CHANGE DATA CAPTURE OVER CASSANDRA

After describing the important characteristics of the base system in sufficient detail, the following presents the change data capture methods.

3.1 Introduction

Known as CDC, change data capture is the process of detecting the data that has been changed in a source system and sending the modifications to a target system. The goal is to reduce the costs of such operation by using just the necessary data instead of the entire source system data to keep both of them consistent. This is a widely used technique in data warehousing because of the performance improvements that it offers, implemented commonly in SQL databases as SQL Server (MICROSOFT, 2008). As in (KIMBALL; CASERTA, 2004), there are many techniques for extracting data from a source system to deliver them to a data warehouse. This work shows some ways of achieving it.

The core idea is to lay aside unnecessary and no effect transactions between systems, decreasing the data volume to be transmitted, copied or computed. To picture a scenario where this technique shows its efficiency, lets use a data warehouse that keeps a copy of a specific table as example. In this case, we will have the source system as the production database and the target system as the data warehouse - that will be another database. Assume that initially both are equal, in other words, they have exactly the same data (rows and values). When one row from the source table gets an update, we need to reflect this operation also in the target table, therefore we have two possibilities: rescan all the source table and store everything again or detect and store just the changed data. Indeed the second option seems to be more reasonable. For example, in the case of a table with one thousand tuples, if the changed data were just five hundred tuples, the difference between rescanning all the source table and capturing the changed data will be fifty percent less tuples in the second case. This means it is possible to cut by half the volume of data to be processed and transmitted and, in the best case scenario, also the time necessary to keep both systems up to date. Keeping in mind that data warehouses can typically store large amounts of data, in the scale of tera and petabytes, this improvement can determine weather the system is practical and efficient or not. Figure 3.1 illustrates the idea of the CDC.

As a requirement, this method must efficiently detect all the possible operations in the source system and make them available to the target system, which will handle them and perform the necessary operations based on the given results. Relational database systems,

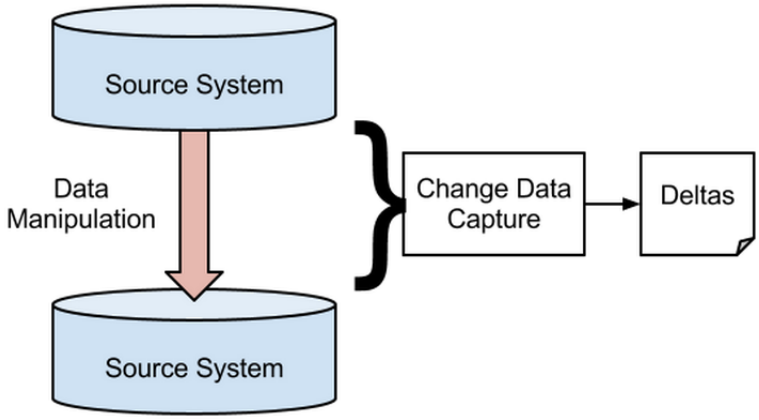

Figure 3.1: CDC Strcuture

as we commonly know, have already explored this area and have successfully built some solutions to capture the changed data, some of them are the snapshot differential, log based, trigger based and the table scan as techniques cited in (VASSILIADIS, 2011) for data extraction. Further details for them are given in the following sections.

3.2 Delta Model

The most important detail when capturing the change data is the delta model. This model must identify correctly the performed operation that was captured and also the row where it happened. One delta will be the union of operation and row, representing uniquely one modification in the database.

The operations can be deletion, insertion and update - which is actually an insertion of an existing column. We can also add here the concept of *upsertion*, which is exactly the idea of inserting again one existing column resulting in an update. In some approaches we will not be able to distinguish between insertion and update because of this peculiarity of the system. In these cases we can correctly name the operation as *upsert*.

In order to identify the row where an operation has been performed and which operation was performed, we present a generic structure for the deltas. Merging the data model structures of the base system allows us to specify which row was modified. Remembering, Cassandra has Keyspace, Column Family, Super Column, Column and Key as layers of data structure. If we add the operation and the value to this information, we build a generic model as:

- *Operation/Keyspace/ColumnFamily/Key/SuperColumn/Column:Value*.

This single string is able to identify uniquely any cell stored in the database, as well as any possible operation. Optionally, we can append the timestamps of the operation, that could be in the end of the delta, creating a delta standard as:

- *Operation/Keyspace/ColumnFamily/Key/SuperColumn/Column:Value-Timestamp*.

It is possible to use different name separator instead of slash (‘/’) in the application, however it is important to have some for parsing reasons when applying these deltas in our

target system. A distinct separator for the value and the timestamp will also help in this manner. The choice of which delta to use is up to the application and its requirements. The biggest difference is in the case of one operation being performed twice with the same value, e.g two sequential updates. If we use also the timestamp, both deltas will be different, if we do not, they will be identical. The following lines exemplify possible deltas for each of the operations listed before, using the example of data structure given in Section 2.1.1.

- Insert: ins/School/Grades/Carol/SecondSemester/Mathematic:89
- Update: upd/School/Grades/Carol/SecondSemester/Mathematic:89
- Upsert: ups/School/Grades/Carol/SecondSemester/Mathematic:89
- Column deletion: del/School/Grades/Carol/SecondSemester/Mathematic
- Super column deletion: del/School/Grades/Carol/SecondSemester
- Key deletion: del/School/Grades/Carol

These are examples for a super column family but we can also have deltas for standard column families. In this case, we can choose either to suppress the super column information or just insert something that shows that there is no super column, e.g 'null'. Notice that we have different deletion granularities. Depending on the approach, there is no way to change this granularity when generating the delta. Thus, it is up to the application to handle such operations when necessary or the case must be ignored, creating a new limitation in the approach.

3.3 Net Effect Operations

Some of the CDC approaches are not able to output all modifications applied to Cassandra or sometimes they output a delta different from the last operation. Such situations are named *net effect operations*. There are common possibilities of net effect operations. In this Section, we name the net effect operations, describe what it represents and give a brief description of the scenario where it might happen. Later, while enumerating the net effect operations in the techniques, the names referred in this Section are given.

Beginning with a simple case, some of the approaches are not able to detect deletion operations. In this text such limitation is named *Deletion* while enumerating net effect operations of methods. Normally, the reason for inability to detect deletions is because a scan is performed at the source column family. Consequently, it is not possible to retrieve any deleted column.

Other recurrent situation is the *Multiple Updates* net effect operation. When multiple updates are performed within a column between two CDC cycles, sometimes it is only possible to retrieve the last updated value. The reason for it is because Cassandra overwrite the previous data when performing an update (i.e., a set command performed in an existing column). The same reason introduces the *Update(s) after Insertion* net effect operation. In this situation, the output is an insertion but with an updated value. It is possible also to have *Deletion after Insertion*, which is the deletion of a value inserted

after the last CDC. This is a special case of *Deletion* net effect operation. To summarize, Table 3.1 relates net effect operations with their names and results in the approaches.

Name	**Description**	**Output**
Deletion	Inability to detect deletions	No output
Update after Insertion	Overwrite of value inserted after the last CDC cycle	Insertion with updated value
Multiple Updates	Overwrite of updated value. Can occur several times consecutively	Update with last updated value
Deletion after Insertion	Deletions of value inserted after the last CDC cycle	No output

Table 3.1: Net Effect Operations

As cited in (HU; DESSLOCH, 2013), *the notion of net-effect change data is utilized to appraise the quality of change data set. It represents the change data has no "noisy data". "noisy data" denotes the change data set which has no dedication to the target data set*. In this work, net effect operations are limitations perceived in several methods, rather than noisy data useless to the target data set. Nevertheless, the result is also a lack of output for specific scenarios. In the end, such limitations must be considered while choosing which approach to implement in a given system.

3.4 Approaches

Given the previous background, we build several CDC approaches. Each method has its mechanisms detailed in the following sections. For all of them, the delta model built previously is used as the output pattern and the net effect operations are listed in each case. The MapReduce framework is used to implement the logic of the approaches and the HDFS is the distributed storage facility used as repository to the outputs.

3.4.1 Audit Column

The audit column technique is borrowed from RDBMS. The main idea is simple: when we insert at first a row we also insert at the same time an additional column with its insertion timestamp or the inserted value. The addition of this new column can be either controlled by the application or by a trigger.

As explained in Section 2.1.2, triggers are not currently well supported, therefore in real applications if this approach is chosen, the audit column is added and controlled by the application rather than by trigger. However, the expected behavior of the additional column can be easily translated to a trigger when it becomes stable and available.

Controlling the additional column is an easy task, as it is just necessary to distinguish between insertions and updates. This is done by trying to retrieve some value from the audit column of the inserting column. If data exists under the audit column, the operation performed in the column is an update, if it does not, the operation is an insertion. After ensuring that the operation is an insertion, it is necessary to insert the data or the value also in the audit column. As expected, it is not possible to use the same column name in the base column and in the audit column, otherwise we will overwrite data. In order to

solve this issue, a new column name is used, which can be ‘audit’ plus the base column name that it belongs, e.g ‘audit-Math’ for the base column ‘Math’. Notice that a separator as hyphen is desirable for parsing reasons. On the other hand, if an update is being performed, it is just necessary to update the base column without changing its audit column. With respect to delete operation, because key and super column deletions will also delete audit columns, it is better if the application or the trigger also deletes the audit column if a column deletion is performed. This makes the operational logic clearer and consistent.

Choosing between storing the operation timestamp or the modified value is up to the requirements of the application. If it stores the timestamp, no further detail besides the previously explained is needed. In case of storing the inserted data, we gain the possibility to catch the old analyzed value. However, it is necessary to maintain the additional column every time a change data capture cycle is performed, storing the updated value in case of a performed update, thus keeping the logic of this method.

3.4.1.1 Extracting Deltas

Knowing the structure of this method, it is possible to extract the changed data in a simple way. First the CDC for audit columns storing the insertion timestamp is shown, later on the differences for audit columns with the inserted value are explained.

To extract the changed data, we scan all the rows of the desired column family. Columns with timestamp greater than the last CDC cycle are the ones to be processed. To distinguish between update and insertion, it is just necessary to compare the timestamps from the column and the stored timestamp in its audit column. The pseudo-code in Figure 3.2 shows the logic that must be implemented.

```
if (Column Timestamp == Audit Timestamp
        && Both > Last CDC Cycle)
        Operation: insertion;
else if (Column Timestamp != Audit Timestamp
        && Both > Last CDC Cycle)
        Operation: insertion;
else if (Column Timestamp != Audit Timestamp
        && Column Timestamp > Last CDC Cycle
        && Audit Timestamp < Last CDC Cycle)
        Operation: update;
```

Figure 3.2: Audit Column Pseudo-code

When storing the value in the audit column instead of the timestamp, the pseudo code will not be different. The only change is that the timestamp taken is the one of the audit column itself. The goal of storing the value is to be able to output the old updated or inserted value. Therefore, we must also maintain this column while extracting the changed data. To do so, it is just necessary to perform some maintenance process if an update is detected. In this case, we must store the updated value in the audit column using the same timestamp as the source column.

Enough information to generate deltas is provided in the end of the analysis of each

```
if (Operation == Update)
        Audit Value = Column Value;
        Audit Timestamp = Column Timestamp;
```

Figure 3.3: Audit Column Maintenance

row. Our implementation always uses the HDFS to store the outputs, however it is possible to store the outputs wherever is more convenient to the application.

3.4.1.2 Net Effect Operations

For this method there are many operations and sequence of operations that cannot be extracted or generate some output different from the real last one. These are the *Multiple Updates*, *Update after Insertion* and *Deletions*. The reason for each of them is as follows: for the multiple updates, the problem is the overwrite of the previous data, without being able to capture all the performed operations; for the very same reason, we have in this approach the Update after Insertion issue. In the concerning of deletion, it is not possible to retrieve any row when scanning the column family if the row was deleted.

3.4.2 Column Family Scan

This approach is similar to the previous one. The difference is on the use of the column timestamp instead of one additional column just to store it. As can be seen, it is much simpler to use this method, but there are several limitation.

3.4.2.1 Extracting Deltas

Column family scan performs a scan in the column family to retrieve all the stored rows. Later on a simple comparison between timestamps of this columns decides if they were modified since the last change data capture cycle or not. To do so, it is necessary a simple comparison between the last CDC time and the column timestamp. If the column has a timestamp greater than the last CDC cycle, than its data was modified, thus allowing the output of a delta. The outputs are limited to upserts, since it is not possible to distinguish between insertions and updates. Furthermore, deletions are also not caught using this method. Figure 3.4 illustrates this method.

```
if (Column Timestamp > Last CDC Cycle)
        Operation: upsertion;
else
        No output;
```

Figure 3.4: Column Family Scan Pseudo-code

One of the most remarkable advantages of this method is that it can be applied in production systems. No further data structure or any sort of change in the database is

required.

3.4.2.2 Net Effect Operations

As previously mentioned, this approach presents several limitations. In this case, all the possibilities of net effect operations may occur, *Multiple Updates*, *Update after Insertion* and *Deletion*. The biggest difference and limitation compared to the audit column technique is that it is not able to distinguish between insertion and updates, with upserts as the only possibility of output.

3.4.3 Snapshot Differential

Borrowed from the Extract, Transform, Load (ETL) area, this approach obtains the changed data by generating snapshots and comparing them, as discussed in (LABIO; GARCIA-MOLINA, 1996). One snapshot is a copy of the database, with all latest version of rows and tables, reflecting the current state of the system. To extract the changed data using this method, it is necessary first to have at least two snapshots, taken in two different time points. These create pictures of the data in two different states. The time difference between two snapshots depends on the application and how often it demands to be kept up to date. After snapshotting, it is necessary to compare them using a simple logic in order to extract the changed data. Further details of this process will be described later.

Figure 3.5 summarizes the general idea behind this method and all stages that compose it.

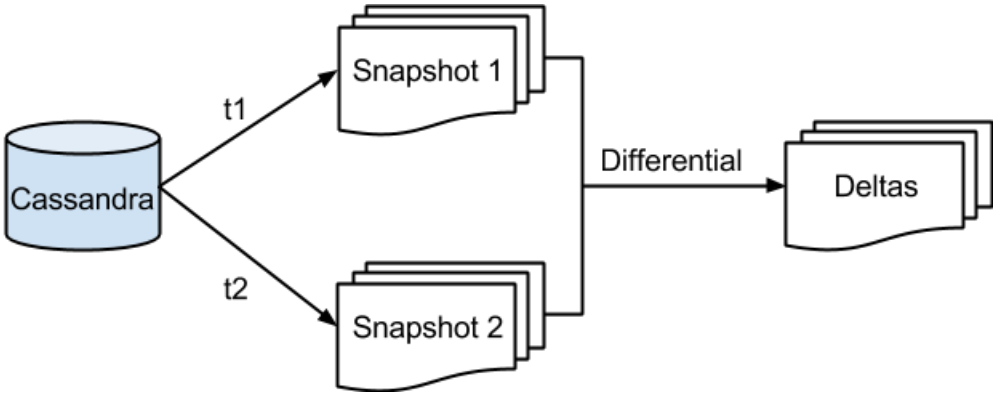

Figure 3.5: Snapshot Differential Cycle

3.4.3.1 Snapshotting

To create the snapshots, it is necessary to define where and how to store the data. Furthermore, the snapshot rows must uniquely identify the rows of the source database. Our implementation stores these results into HDFS and uses a row structure for each value as described in the next paragraph.

The data model can be used to uniquely identify each value of the source database. Similarly as when defining generic deltas in Section 4.2, in Cassandra a merge of the names of the Keyspace, Column Family, Super Column, Column and Key structures identifies uniquely a row. The value must be appended to the previous structure, and optionally also the timestamp. A generic row of a snapshot is as:

- *Keyspace/ColumnFamily/Key/SuperColumn/Column:Value-Timestamp*.

For snapshots of standard column family, it is possible either to suppress the super column information or add a mark in its place. Our implementation adds the 'null' string as a mark, which makes the process more general, being consistent either working with standard or super column families. It is possible to choose other separator rather than slash ('/'), but it is important to have some for parsing reasons. It is also preferable to have a different separator to the timestamp. When creating deltas and sending them to target systems, these separators will be useful. As we are in a distributed ambient, we can use the MapReduce framework to take all advantages of the base architecture while implementing the snapshotting program. Our implementation consists of only a mapper phase, with no need of reducer. Its pseudo-code is presented in Figure 3.6.

```
for (Columns into Column Family) {
        Create Key:
                Keyspace/ColumnFamily/Key/SuperColumn/Column;
        Create Value:
                Current Database Value-Timestamp;
        Store Key and Value;
}
```

Figure 3.6: Snapshot Mapper

Notice that it is important to follow the defined row structure and store them in some reachable place.

3.4.3.2 Extracting Deltas

After snapshotting at least twice the source system, it is possible to extract the changed data. To do so, a comparison between snapshots is done, outputting also the operation that was performed in the row. Suppose one has an old snapshot 'OS', a snapshot row 'r' with value 'v' and a new snapshot 'NS'. Table 3.2 shows the possible situation during comparisons in this scenario and how to interpret them. The 'X' in the cell indicates that the condition is satisfied.

$r1 \in OS$	$r2 \in NS$	$(v1 \in r1)\&\&(v2 \in r2)\&\&(v1 == v2)$	$PerformedOperation$
X	X	X	No Change.
X			Row Deleted.
	X		Row Inserted.
X	X		Row Updated.

Table 3.2: Delta Extraction from Snapshots

The granularity of the comparison and the outputs are always the column granularity, even if the operation was of key or super column granularity.

To implement this logic using the MapReduce framework, a reducer side join is required. This uses mappers to tag and join values of different snapshots in the same list. It is necessary to have multiple inputs, defining which mapper receives each snapshot. Later a reducer phase is used to compare the mapper output list and extract deltas. The key outputted from the mapper to the reducer must have the Cassandra's meta data information, thus each reducer will be able to extract a delta for one row. A pseudo-code for the mapper phase is as in Figure 3.7.

```
for (all Key-Values) {
        Create Key = Keyspace/ColumnFamily/Key/SuperColumn/Column;
        Tag value with snapshot number;
        Send Key and Value to reducer;
}
```

Figure 3.7: Differential Mapper

When the mapper phase is complete, the tagged data for both snapshots are send to the same reducer, where the comparison logic is implemented as in the pseudo-code in Figure 3.8.

```
if (Value Tag 1 == Value Tag 2)
        No Change, No Output;
else if (Value Tag 1 == null)
        Output: inserted Value Tag 2;
else if (Value Tag 2 == null)
        Output: deleted Value Tag 1;
else if (Value Tag 2 != Value Tag 2)
        Output: updated to Value Tag 2;
```

Figure 3.8: Differential Reducer

3.4.3.3 Net Effect Operations

The Snapshot Differential method has the following two net effect operations: Multiple Updates and Update(s) after Insertion. The reason for the existence of both is the overwrite of data when performing an update. There is also another possibility of net effect operation not explained previously in this document. It is not possible to detect different deletion granularities. The output deltas for such manipulation will be always in the column granularity, even if a key or super column deletion was performed instead.

3.4.4 Log Based

Capturing the changed data would be better if using something already provided by the system instead of creating new structures, storing new row replicas or adding complexity to the database. For this reason, the use of the existent commit log seems to lead to a less invasive solution, with the best possible performance without interfering too much in the existing system.

This method performs the extraction of changed data by reading and interpreting the log, specifically the commit log. This log keeps track of all operations performed in the database and can be used to restore data after a system crash or for durability purpose. The first place where an operation is registered is in the commit log. If it fails, the operation is not validated and must be repeated. There are three basic requirements to implement the CDC in this scenario, they are:

1. Knowledge of the log structure, being able to interpret the different operations for each log row. In Cassandra it is not straightforward.
2. Ability to have a certain level of control of the log, avoiding deletion of rows not analyzed, that would possibly discard significant data.
3. Use of the log without interfering in the system, neither performance nor consistency.

The complexity of an implementation that fulfills these requirements relies on the structure and control of the log. After understanding these elements and knowing the output structure, the following steps are straight forward. A study of a log based method applied in a real-time data warehousing is proposed in (SHI et al., 2008).

3.4.4.1 All About the Commit Log

As cited above, the commit log is the first place where data is written. It has a significant importance to the system, since losing data is not an option, therefore the register of every operation performed in the database must be done. As explained in (HEWITT, 2011) *"All writes to all column families will go into the same commit log..."* and *"...only one commit log is ever being written to across the entire server"*, these are main notions to understand the Cassandra's commit log handling.

Our application is data sensitive, that is why a fine tuning between Cassandra's configuration and the Log Based approach is desirable. There are several options to setup the log handling in Cassandra. It is possible to choose its synchronization between *batch* and *period*, where the first is to write mutations to commit log after a defined number of operations and the second option makes the log writes be done after a specified time window. Both can be seen as the number of database changes that we accept to lose in case of a system fail, e.g., if the synchronization period is of 1 minute and after 30 seconds the last sync the system crashes, all the operations done in this time frame will be lost. Some would want to set the maximum size for the commit log, which is also tunable. These setups among others are made at *"conf/cassandra.yaml"*, the main Cassandra's configuration file. To set the previous explained properties there are the following parameters:

- *commitlog_sync*: the way how the log must be synchronized.
- *commitlog_sync_period_in_ms*: time threshold to synchronize.
- *commitlog_total_space_in_mb*: maximum size of log.

These are sufficient parameters to guarantee not losing relevant logged operations. We use the default synchronization type *periodic*, with a short period of time to sync the mutations into the commit log (e.g., 10ms) and a maximum log size large enough to bear the used amount of data in our implementation.

It is also necessary to know how is the inner structure of the commit log. The log is composed of serialized log rows, each one related to one mutation. The structure of each log row depends on which operation was performed (we are interested in *set* and *del*) and in which kind of column family the manipulation was performed (i.e., Standard Column Family or Super Column Family). After performing a reverse engineering of Cassandra's source code, it was possible to figure out the general structure of the commit log. The main data structures of the commit log are described in detail in Appendix A. The structures refers to Cassandra version 1.0.9, for other versions the commit log might be different.

In order to retrieve the serialized information, it is important to know the fields types. Otherwise, it is not possible to read correctly every provided information. Cassandra's implementation is in Java as well as our programs, so we listed the Java types related to each serialized field. Notice that the serialization flag is an integer. Appendix A also presents the relationship between flags and their serialized numbers.

All explained above is what is necessary to build and implement programs capable of interpreting and extracting intelligible information from Cassandra's commit log. To exemplify, there are two serialized log rows representing an operation at a standard column

family as in Figure 3.9 and a mutation at a column under a super column family as in Figure 3.10. These examples are related to a set operation performed into the database, which can either represent an insertion or an update.

```
(School)(Stephanie)(1)
(1018)(true)(1018)(-2147483648)(-9223372036854775808)(1)
(Mathematic)(0)(1349430004096)(5)
```

Figure 3.9: Log Row Standard Column Insertion Example

```
(School)(ELucile)(1)
(1021)(true)(1021)(-2147483648)(-9223372036854775808)(1)
(Employees)(-2147483648)(-9223372036854775808)(1)
(Bonus)(0)(1349942428550)(150)
```

Figure 3.10: Log Row Super Column Insertion Example

Because the operation is a set, the deletion timestamps are negatives, signalizing that there was no deletion performed. The numbers used for such operations are always the same as in Figures 3.9 and 3.10. Furthermore, the column family ID is an internal ID used by Cassandra. It is up to the system administrator to establish the relation between the column family ID and its name. A way to do so, is by inserting a specific key into the desired column family and deserialize the log searching for the key. When the key is found, the relation between the internal ID and its column family name can be established. This process is done just once, since this ID will not change unless the entire column family is dropped and a new one is created.

3.4.4.2 Extracting Deltas

There are two main ways to get deltas using this technique, which may lead to one or another situation: using a partial log or using the entire log. If a set of the log is used, it is simpler to extract information but the method has more limitations. On the other hand, if the entire commit log is analyzed, it is possible to find any database mutation, but with the cost of a more complex and cumbersome data processing.

Beginning with the simple case, it is necessary first to define which part of the log to keep. An easy and effective choice is to always preserve mutations occurred after the last CDC cycle. This ensures that any performed operation is analyzed. One advantage of this partitioning of the log is the simple logic that must be implemented to keep the method consistent. It is required just to keep and process log rows added in between two CDC cycles. Before starting to capture the changed data, it is necessary to 'cut' the log until the current time. Meanwhile, new mutations are registered in a new cut of the log. By the end of the analysis, the method discards the analyzed log rows and waits as long as necessary to start processing new rows. With this simple logic to keep the log, it is possible to build

the CDC process. Therefore, there are a mapper and a reducer in Figure 3.11 with a logic capable of extracting deltas of a partial log processing.

Mapper

```
for (Log Rows) {
    send Log Rows of each RowKey
    together to reducer;
}
```

Reducer

```
for (All Mutations) {
    Generate delta based on mutations;
    Output: delta;
 }
```

Figure 3.11: Partial Log Map-Reduce Program

In a real application, it is possible to choose to build a more complex reducer, for example, one that outputs just the last operation for a given key. In this case, it is necessary to take care of deletions of bigger granularity, which may delete previous insertions or updates of smaller granularity. Unlike the previous methods that output deltas of column level operations, the log based approach generates outputs with several different granularities, which depends on the performed operation.

In the following we build the log based method using the entire log. Notice that it is mandatory to have the full commit log in order to have this model working consistently. If it is not possible to handle such amount of data for the entire database, one can save log entries just for the column families that are necessary to keep track. One way to do so, is creating a new log using triggers or, a more suitable approach for databases that does not support triggers, constantly process the commit log and copy new log rows of the desired column family somewhere else.

Despite the cumbersome prerequisite, for some applications this might be the only desired method. If one wants to retrieve deltas for any operation applied into the database in any given time or desires to retrieve any kind of operation in any granularity, this is the method to be considered. Outputs of the log based technique with the entire commit log are in several different granularities as in the partial log, e.g., key, column or super column. One advantage of using the whole log is the ability to transform deletions of bigger granularity into a smaller granularity. For instance, one might want to transform super columns deletions into columns deletions. Therefore, it is necessary to analyze the previous insertions under the given super column. The same idea is applied for key deletions. Furthermore, using the full log enables us to distinguish between insertions and updates, while the partial log is capable just to output *upserts* because both logical operations are registered identically in the commit log. With the full log, it is possible to make a back-trace of mutations performed in the same column to discover if a specific mutation was an insertion or an update.

In our implementation we use the same mapper as in the partial log from Figure 3.11 to send log rows to same reducer. Later on a reducer as in Figure 3.12 outputs all analyzed operation in the registered granularity. Insertions and updates are distinguished by including in the process all previous mutations of the key, as can be seen in the referred pseudo-code.

```
List<OldLogRows>
List<NewLogRows>
for (Key of NewLogRows) {
    if (Key == Deleted)
        Output: LogRow key deleted;
    else {
        if (Key == Inserted && OldLogRows contains Key)
            Output: LogRow key update;
        else
            Output: LogRow key inserted;
    }
}
```

Figure 3.12: Full Log Reducer

The provided model can be changed to fit the application requirements and expectations. Also, it is possible to create an operations log, maintained by a trigger or the application itself. The disadvantage then are the higher processing overhead added that will affect the performance of the system and the increase in the amount of stored data.

3.4.4.3 Net Effect Operations

This method has the advantage of not having any net effect operation. The reason is because all performed operation are analyzed and can be handled as desired. This means, basically, that the technique is able to see and process all the operations performed within the database. Also, another reason for the lack of net effect operations is that there is no data overwrite, so no data are lost.

For instance, in the common scenario of *Update after Insertion*, the techniques are not able to detect the previous insertion because its previous value is overwritten by the updated value. On the other hand, the log have an entry for the insertion operation and another log row for the update operation, thus avoiding the mentioned net effect operation.

3.4.5 Trigger Based

In (VASSILIADIS, 2011), trigger based techniques are *"non-traditional, rarely used"* techniques for data warehousing. Mainly because they *"require the modification of the source applications to inform the warehouse on the performed alterations at the source, or, the usage of triggers at the source side"*. Nevertheless, in the following it is presented an alternative of such technique to Cassandra.

Deltas are obtained in this method by tracking all the operations performed in the

system and storing them in a specialized table. Such table can be called 'tracking table' and it must be capable to output all kinds of possible operations provided by the source database.

The model of the tracking table must be capable to uniquely identify one operation of a row and also to distinguish between the different operations possibilities. It is required an algorithm to fill the tracking table when data manipulation is performed in the source system, as well as a logic to extract the operations and rows when scanning this table. To achieve these requirements, it is possible to create a trigger, which is fired for every single row mutation in the database and is capable to enforce and maintain the logic of the tracking table. Later on, a mechanism to read the tracking table extracting deltas and refresh it is executed. This mechanism must change the table to a state where the trigger can successfully identify which changed data was already analyzed, thus keeping the tracking logic immutable. The next subsections cover the creation and analysis of such a tracking table.

3.4.5.1 Tracking Table

This section explains how to implement a tracking table logic using triggers. In the following, we build an approach to get the last performed operation. Basically, the tracking table stores columns under super columns, the first representing where the mutation occurred and the second represents which kind of mutation was executed. It is also important to save the last extracted value, in order to distinguish between insertions and updates. Therefore, it is possible to fix the names of such super columns as: *Old Value*, *Insertion*, *Update* and *Deletion*. Each of these super column have names closely related to the operations that they keep track. Under them, the changed columns or super columns will be stored. To store other super columns in the tracking table, it is necessary to merge the super column name with its standard column name. Thus, a column uniquely identifies where the change happened, e.g., an insertion of standard column *Math* under super column *1^{o}Semester* will be stored under super column *Insertion* as column *1^{o}Semester/Math* in the tracking table.

A robust logic is build if the recorded operations are in the same granularity, e.i., in the column granularity. This specific granularity is used because it is the only possible one when performing set operations, which are very important to this work. Deletions can have several different granularities, so it is necessary to change them in order to build a cleaner approach. To do so, the pseudo-code in Figure 3.13 is merged to the trigger, which will then identify any deletion of any other granularity rather than column and translate it into columns operations.

```
if (Deletion of Super Column)
        for (All Standard Column under Super Column)
                Delete Column;
else if (Deletion of Key)
        for (all Columns or Super Columns under Key)
                Delete Column;
```

Figure 3.13: Changing to Column Granularity

The trigger is built easily after the first processing phase previously described, mainly because it deals with a more uniform possibility of operations. Figure 3.14 represents the trigger logic built on top of the previous computation.

```
if (Operation == Del)
    if (Column under Old Value Super Column)
        Copy Old "value:timestamp" to Delete Super Column;
        Delete Column from Insertion and Update Super Columns;
    else
        Delete Column from Insertion and Update Super Columns;

else if (Operation == Set)
    if (Column under Insertion Super Column)
        Overwrite "value:timestamp" for Colum under Insertion Super Column;
    else if (Column under Old Value Super Column)
        Overwrite "value:timestamp" for Column under Update Super Column;
    else
        Insert "value:timestamp" for Column under Insertion Super Column;

    if (Column under Deletion Super Column)
        Delete Column under Deletion Super Column;
```

Figure 3.14: Trigger Logic

Some applications might not need the first processing step. That is why the trigger and the changing granularity algorithm are described separately. Real implementations might need to merge both mechanisms.

3.4.5.2 Extracting Deltas

In the following we present an implementation of a MapReduce program to extract deltas from a tracking table and maintain it. Figure 3.15 describes how a program with just a mapper phase successfully extracts deltas and maintain a tracking table, outputting the last performed operations of columns. Notice that each mapper has a key and all columns of the particular key as inputs.

3.4.5.3 Net Effect Operations

In this method the net effect operations are: *update after insertion*, *update after update* and *deletion after insertion*. They are all related to the triggers logic, which overwrites data not yet maintained. This is the reason for the first and second listed net effect operations. The deletion after insertion happens because the triggers delete data under 'Insertion Super Column' if the inserted column is deleted before a CDC cycle.

```
for (All Super Columns)
    for (Each Column)
        if (Last CDC timestamp < Column timestamp < Current CDC timestamp)
            if (Column under Insertion Super Column)
                Output insertion of Sub Column;
                Move Column under Insertion to Old Value;
            else if (Column under Update Super Column)
                Output update of columns;
                Move Column under Update to Old Value;
            else if (Column under Deletion Super Column)
                Output deletion of Column;
                Delete Column from Deletion and Old Value Super Columns if exists;
```

Figure 3.15: Trigger Based Mapper

4 IMPLEMENTATION AND EVALUATION

This chapter aims to evaluate functionally the implementations as well as their performance. Furthermore, a summary of the important aspects previously discussed are stated here. The goal is to give an overview of relevant details covered in this document.

4.1 Implementation

We implemented the techniques for CDC described in Chapter 3 in Java, with Cassandra version 1.0.9 and MapReduce framework version 1.0.4 in a Unix environment. Other versions are also supported, as MapReduce 0.20.2. Finally, porting the code to the latest version of Cassandra and MapReduce was attempted, unfortunately without success.

The implementation supports all the described approaches. It is possible to use input arguments to choose between algorithms, as well as run performance tests and create or maintain tracking tables or audit columns. A property file is used by the API to set necessary parameters to interact with the source database. Such a file can also be passed to the API as an argument. A fine configuration of the CDC techniques is achieved by changing parameters of the property file. Finally, the structure presented in Figure 2.3 is used to guide our implementation.

With the plugin EclipseAid for Eclipse, it is possible to create class diagrams which represent the implementations. In the following sections, we present class diagrams of each implemented package. These packages are integrated in a main function, used as the front end of the API.

4.1.1 Audit Column

Figure 4.1 shows the four classes implemented in this technique. The Java class *AuditColumn* is the main class, which calls and runs classes implementing map and reduce functions. There are two different mapper classes: one to handle standard column families and other to handle super column families. These mappers are responsible to process and send column and audit column together to the reducer, which is implemented by the *AnalysisReducer* class.

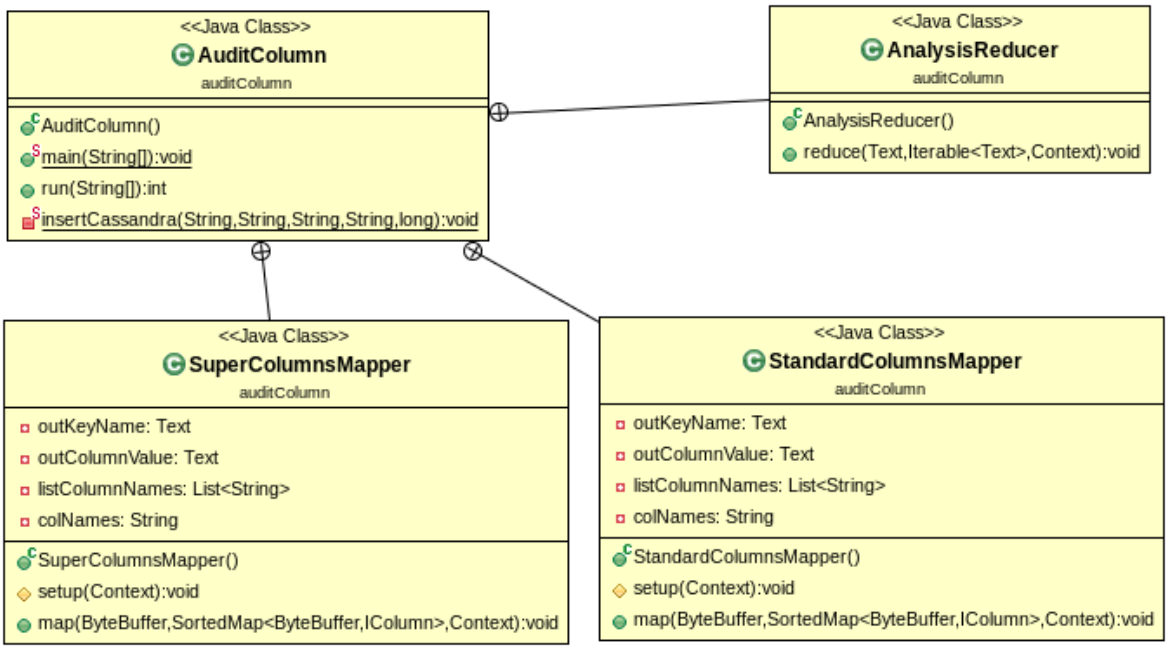

Figure 4.1: Audit Column Class Diagram

4.1.2 Column Family Scan

This is the simplest implementation. There is a main class *TableScan*, which starts the MapReduce computation. The mappers scan an entire column family, detect deltas and output them to HDFS. Each mapper is responsible to process data from different column families, i.e., standard and super column family. Figure 4.2 shows the class diagram for such classes.

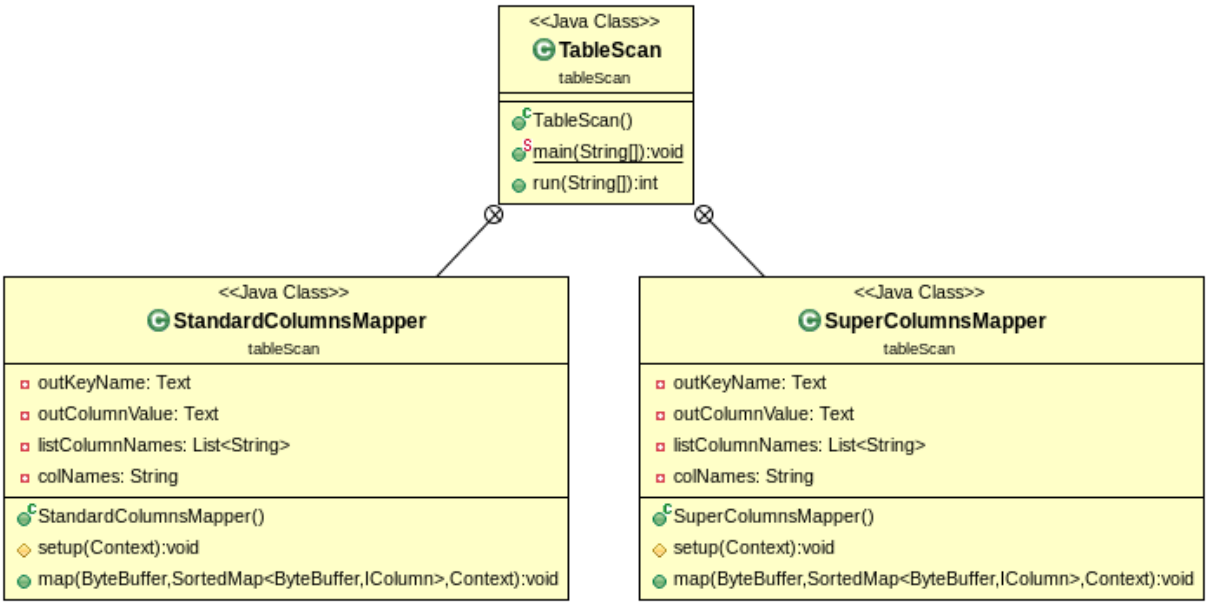

Figure 4.2: Column Family Scan Class Diagram

4.1.3 Snapshot Differential

This package has separated implementations for each phase of this technique. The snapshot phase has the class *Snapshot* as the main class, which starts the MapReduce framework with mappers as the classes *SuperColumnStoreMapper* or *StandardColumn-StoreMapper*, depending on the column family type being processed.

The class *Differential* executes the second phase of the CDC solution. Two different mapper tag the input data with their source and send them together to *DifferentialReducer*, which processes the data and outputs deltas to HDFS. Figure 4.3 shows a class diagram for both phases.

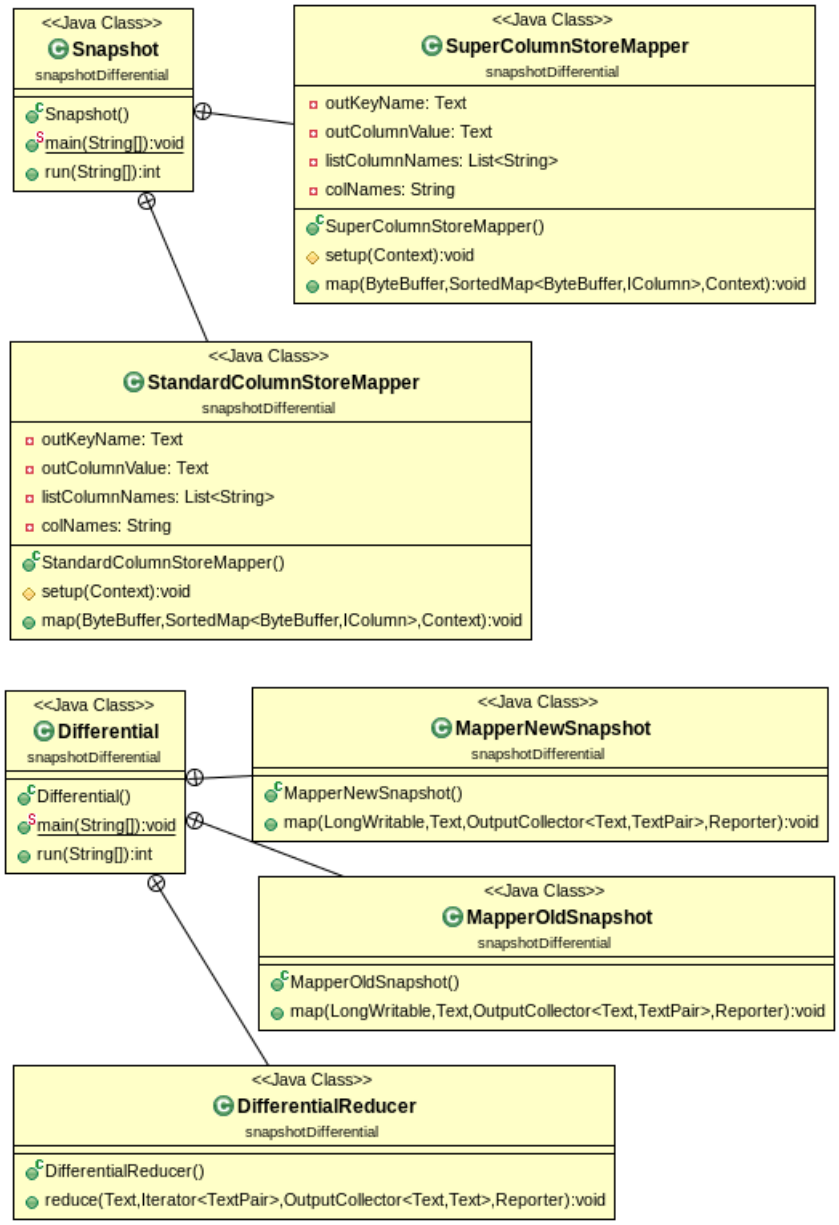

Figure 4.3: Snapshot Differential Class Diagram

In the *utils* package, there are three important classes for this technique. These classes implement a *TextPair* input format to MapReduce framework; they are required in order

to tag the input data. Furthermore, it is necessary to implement comparators used to distribute the input data across mappers, as seen in the class diagram in Figure 4.4.

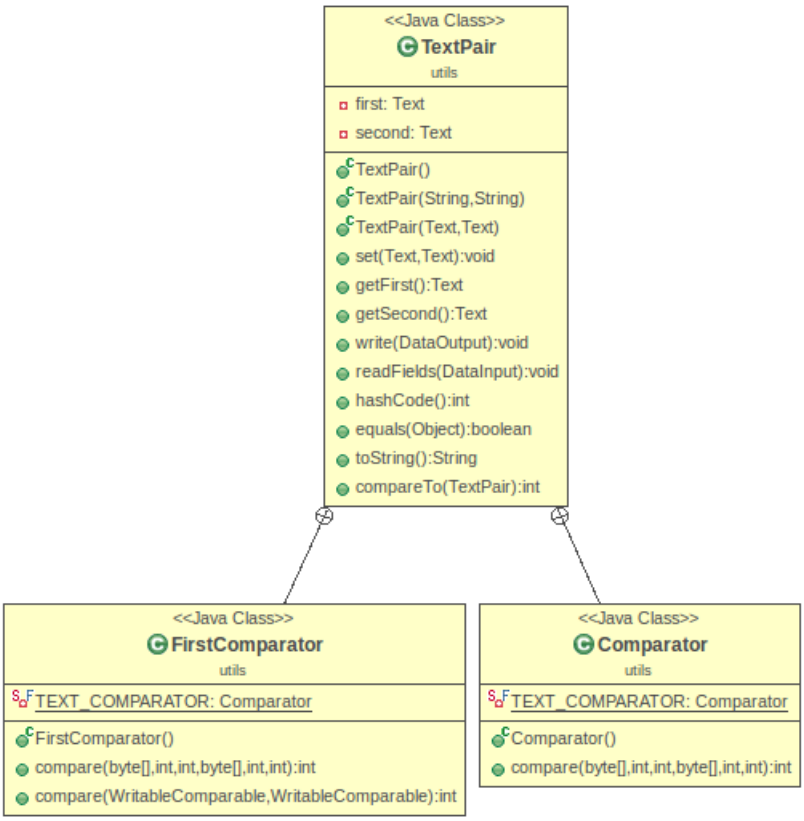

Figure 4.4: Utils Class Diagram

4.1.4 Log Based

The log based approach uses log rows objects to represent serialized log rows. Figure 4.5 describes the object with all its methods and attributes.

Figure 4.6 shows the class diagram for the log based approach itself. There are several classes, the *LogBased* is the main class, which runs the MapReduce framework with the mapper as *ReadSegmentsMapper* and the reducer as *AnalysisReducer*. Both use the class *KeyValueGenerator* to generate key-value pairs based on the deserialized data, which are used as the output from mapper to reducer and from reducer to HDFS. Class *SegmentInputFormat* tells MapReduce framework how it must handle the given input, using *MutationRecordReader*. The other classes have self-explaining names.

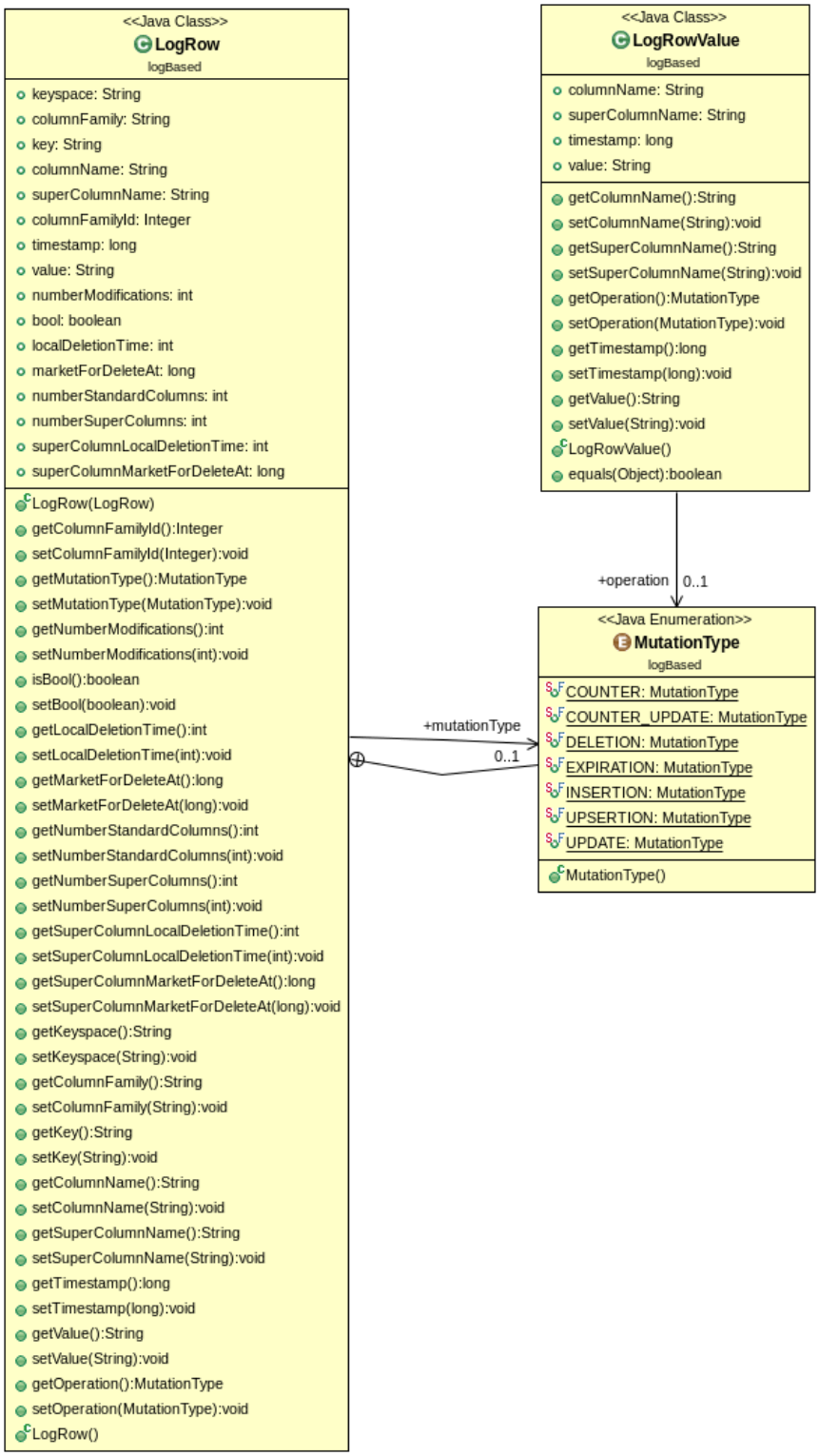

Figure 4.5: Log Row Objects Class Diagram

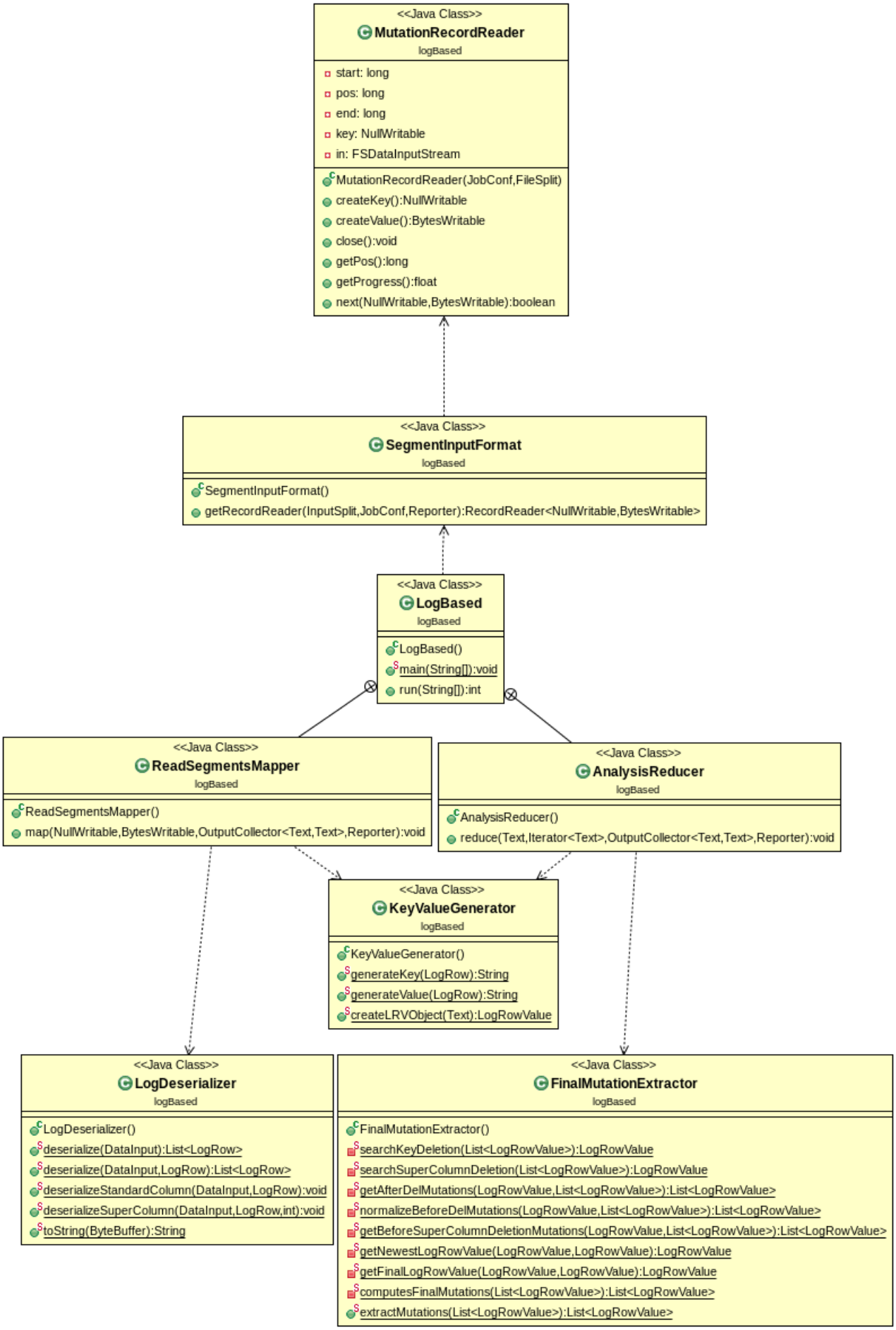

Figure 4.6: Log Based Class Diagram

4.1.5 Trigger Based

This technique has a main class *TrackingTableMapRed*, which tells the MapReduce framework to run the mapper *TrackingTableMapper*. The mapper outputs data to HDFS and also maintains tracking tables.

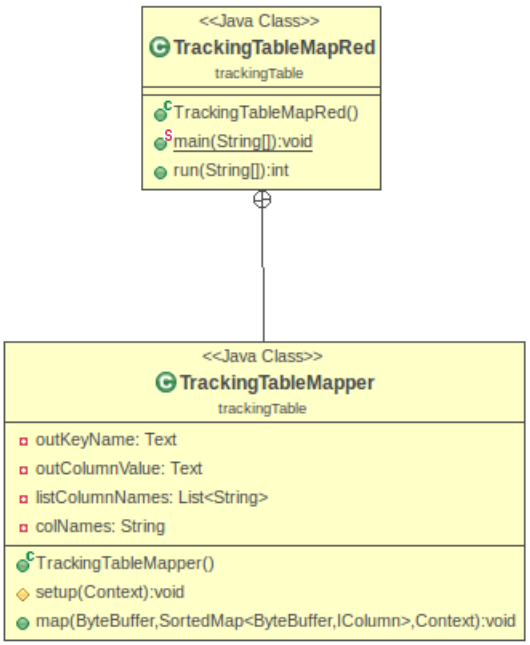

Figure 4.7: Tracking Table Class Diagram

4.2 Evaluation

The goal of this section is to present an evaluation of the implemented techniques. In the following we show a functional evaluation, due to different limitations presented by each approach. Later, a performance evaluation comparing the algorithms is given.

4.2.1 Functional Evaluation

As aforementioned, all the methods implemented have net effect operations or some limitations, which must be considered while choosing which approach to use in a real application. To give a better view of the complete scenario, Table 4.1 summarizes the detectable operations related to the presented approaches. These are the first details to be analyzed by the developer. For some applications, the performance is irrelevant if the solution used is not able to deliver the required data, thus making the approach useless.

	Prev Val	Update	Insert	Delete	Upsert	History
Audit Column	✗	✓	✓	✗	✗	✗
Table Scan	✗	✗	✗	✗	✓	✗
Snapshot Differential	✓	✓	✓	✓	✗	✗
Partial Log	⚠	⚠	⚠	⚠	✓	⚠
Full Log	✓	✓	✓	✓	✗	✓
Tracking Table	✓	✓	✓	✓	✗	✗

Table 4.1: CDC Detectable Operations

In Table 4.1, the techniques developed in this work are listed. The columns of the referred table represent which operations each technique is able to extract. Columns *Update, Insert, Delete* and *Upsert* are self-explanatory. The column *Prev Val* is related to the ability of the approach to deliver the value of the previously extracted delta for a determined Cassandra's column. The column *History* refers to the ability of the technique to extract historic changes for a specific Cassandra's column, e.g., in a given time window, output all the modifications of column 'Math' for key 'Tom'.

The symbol '✗' indicates that the technique is not able to detect the operation of the referred column. On the other hand, the symbol '✓' indicates that the approach has the ability to do it. Finally, the symbol '⚠' is used when the technique is able to detect the referred operation but in a limited way.

4.2.2 Experimental Setup

We used the open source tool called YCSB, described in (COOPER, 2010) capable to run tunable workloads in several databases - one of them is Apache Cassandra. Although this tool does not perform deletions within the database, the ability to run insertions, reads and updates is sufficient to characterize how the techniques behave when they are required to process the same amount of data. The following command lines are examples of loading and running a workload, setting also the number of operations and records to be manipulated. These are the commands used to run our tests:

- Load: *$bin/ycsb load cassandra-10 -P workloads/workloada -p insertorder=ordered -p recordcount=1000000 -p operationcount=1000000*
- Run: *$bin/ycsb run cassandra-10 -P workloads/workloada -p insertorder=ordered -p recordcount=1000000 -p operationcount=1000000*

As previously discussed, Cassandra with triggers is unstable. To test the performance of approaches that require triggers, it is possible to generate artificially the necessary data and structures. In Audit Column approach for instance, after loading and updating or inserting data into the source database, a simple program is executed to create all the audit columns. Figure 4.8 describes such algorithm, which simulated the behavior of a trigger maintaining audit columns of a specific column family. Notice that the performance test does not cover the overhead added by using a trigger.

```
for (all Columns in Source Column Family)
    if (Insertion Workload)
        if (Column Timestamp < Workload Run Phase Timestamp)
            Create Audit Column with Timestamp < Workload Run Phase Timestamp
        else
            Create Audit Column with Timestamp == Column Timestamp
    else if (Update Workload)
        if (Column Timestamp < Workload Run Phase Timestamp)
            Create Audit Column with Timestamp == Column Timestamp
        else
            Create Audit Column with Timestamp < Workload Run Phase Timestamp
```

Figure 4.8: Audit Columns Creation

Another trigger based method is the tracking table. In this case, is also possible to synthesize the necessary data using an algorithm to maintain a tracking table. Figure 4.9 shows how such algorithm must be implemented. Notice that the algorithm scans a source column family and inserts data into a tracking table, which is a different column family.

```
for (all Columns in Source Column Family)
    if (Insertion Workload)
        if (Column Timestamp > Workload Run Phase Timestamp)
            Insert Column under 'Old Value' and 'Insertion' Super Columns
        else
            Insert Column under 'Old Value' Super Column
    else if (Update Workload)
        if (Column Timestamp > Workload Run Phase Timestamp)
            Insert Column under 'Old Value' and 'Update' Super Columns
        else
            Insert Column under 'Old Value' Super Column
```

Figure 4.9: Maintaining a Tracking Table

After detailing the logic of algorithms to simulate the behavior of the necessary triggers, it is possible to prepare the test scenarios. Therefore, it is necessary to do the following steps, in the same described order:

1. Load the workload with the base data
2. Save timestamp
3. Run the worload, changing the initial state of the database
4. Maintain Audit Columns and Tracking Table

It is fundamental to have a timestamp marking the initial state of the database, otherwise the methods interpret all performed operations as insertion even though updates were performed.

The system used to test our implementation is a cluster composed by 12 nodes. The cluster has 9 processors Intel Pentium 4 HT @ 2.8Ghz, 3 Intel Pentium 4 @ 2.8Ghz, a total RAM memory of 24.25GB and a total storage size of 10.2TB. The nodes are connected with each other with 1000Mbps ethernet cables. The cluster runs Hadoop version 1.0.4 (for both MapReduce and HDFS services), as well as Apache Cassandra version 1.0.9. The Java used is version 1.6.

We used YCSB (COOPER, 2010) to generate the workloads in different data volumes and data operations. The tests use a variation from 25%, 50% and 75% of data manipulation, either for insertions or for updates. Each one performed in a separated workload. Also, two different amounts of data are used: 500 thousand rows keys and 1 million row keys. In the load phase, this workload inserts the base data, representing the initial state of the database. In the run phase, the workload updates the previously loaded data, with insertions or update operations. Due to technical problems and time limitations, the workloads using 500 thousand row keys for the insertion workload are not presented in this document, as well as the 50% insertion workload for 1 million row keys initial data set. One operation in a workload is responsible for manipulating 10 columns of a key and each column contains 1KB of data. As we set the number of operations to 1 million, the total amount of data inserted into the database is around 9.6GB - 1 million x 10 x 1KB. In the case of 500 thousand keys, the amount of data loaded into the database is around 4.8GB.

4.2.3 Performance Evaluation

The tests presented in the following are for the extraction of deltas. The additional overhead of using triggers is not measured; also, the maintaining time of tracking table or audit columns are not considered. The total amount of time necessary to complete the execution of each of the presented techniques are as follows:

- Audit Column:
  - create audit column + extract delta + delete audit columns
- Column Family Scan:
  - extract delta
- Snapshot Differential:
  - create snapshots + extract deltas + maintain snapshots
- Log Based:
  - extract deltas
- Log Based (with trigger):

  - create the log + extract deltas + maintain the log

- Trigger Based:

  - create the tracking table + extract deltas + maintain the tracking table

As presented at (JAIN, 1991), in order to have a significant performance comparison, it is required to run at least 30 times each test for each method. After collecting all this data, it is possible to calculate the 95th and the 99th percentile, meaning that in ninety five and ninety nine percent of the our test cases the algorithm completes under the related times respectively. Further, the mean of the processing time is calculated, presenting also confidence intervals for confidence levels of 95% and 99%. These statistics data are summarized in the following tables.

Method	**Mean (s)**	**95% Confidence**	**99% Confidence**
Tracking Table	284	± 2.27	± 2.99
Column Family Scan	279	± 1.32	± 1.73
Differential	460	± 0.47	± 0.62
Snapshot	489	± 0.73	± 0.96
Audit Column	661	± 2.48	± 3.26

Table 4.2: 9.6GB Data Set 25% Update Workload Confidence Intervals

Method	**95 Percentile (s)**	**99 Percentile (s)**
Tracking Table	300	306
Column Family Scan	287	289
Differential	463	463
Snapshot	492	494
Audit Column	672	687

Table 4.3: 9.6GB Data Set 25% Update Workload Percentiles

Method	**Mean (s)**	**95% Confidence**	**99% Confidence**
Tracking Table	264	± 0.73	± 0.97
Column Family Scan	265	± 0.83	± 1.08
Differential	460	± 0.49	± 0.64
Snapshot	484	± 1.41	± 1.86
Audit Column	646	± 1.84	± 2.41

Table 4.4: 9.6GB Data Set 50% Update Workload Confidence Intervals

Method	95 Percentile (s)	99 Percentile (s)
Tracking Table	267	273
Column Family Scan	270	270
Differential	463	463
Snapshot	489	492
Audit Column	653	662

Table 4.5: 9.6GB Data Set 50% Update Workload Percentiles

Method	Mean (s)	95% Confidence	99% Confidence
Tracking Table	288	± 6.18	± 8.12
Column Family Scan	289	± 0.92	± 1.2
Differential	460	± 0.51	± 0.67
Snapshot	489	± 1.43	± 1.87
Audit Column	665	± 2.17	± 2.85

Table 4.6: 9.6GB Data Set 75% Update Workload Confidence Intervals

Method	95 Percentile (s)	99 Percentile (s)
Tracking Table	297	303
Column Family Scan	297	297
Differential	463	463
Snapshot	492	494
Audit Column	672	753

Table 4.7: 9.6GB Data Set 75% Update Workload Percentiles

Method	Mean (s)	95% Confidence	99% Confidence
Tracking Table	150	± 3.57	± 4.7
Column Family Scan	152	± 0.4	± 0.53
Differential	229	± 0.45	± 0.59
Snapshot	265	± 0.56	± 0.74
Audit Column	311	± 1.02	± 1.34

Table 4.8: 4.8GB Data Set 25% Update Workload Confidence Intervals

Method	95 Percentile (s)	99 Percentile (s)
Tracking Table	155	156
Column Family Scan	155	155
Differential	230	231
Snapshot	267	267
Audit Column	315	321

Table 4.9: 4.8GB Data Set 25% Update Workload Percentiles

Method	Mean (s)	95% Confidence	99% Confidence
Tracking Table	152	±0.28	±0.37
Column Family Scan	152	±0.32	±0.42
Differential	229	±0.51	±0.67
Snapshot	264	±0.57	±0.74
Audit Column	310	±1.39	±1.79

Table 4.10: 4.8GB Data Set 50% Update Workload Confidence Intervals

Method	95 Percentile (s)	99 Percentile (s)
Tracking Table	155	155
Column Family Scan	155	155
Differential	231	231
Snapshot	267	267
Audit Column	315	323

Table 4.11: 4.8GB Data Set 50% Update Workload Percentiles

Method	Mean (s)	95% Confidence	99% Confidence
Tracking Table	152	±0.18	±0.24
Column Family Scan	152	±0.27	±0.36
Differential	230	±0.3	±0.4
Snapshot	265	±0.7	±0.91
Audit Column	310	±0.86	±1.12

Table 4.12: 4.8GB Data Set 75% Update Workload Confidence Intervals

Method	95 Percentile (s)	99 Percentile (s)
Tracking Table	153	154
Column Family Scan	155	155
Differential	230	231
Snapshot	267	270
Audit Column	315	315

Table 4.13: 4.8GB Data Set 75% Update Workload Percentiles

Method	Mean (s)	95% Confidence	99% Confidence
Tracking Table	405	±0.88	±1.15
Column Family Scan	374	±1.11	±1.46
Differential	535	±0.51	±0.68
Snapshot	606	±2.05	±2.69
Audit Column	848	±3.55	±4.66

Table 4.14: 9.6GB Data Set 25% Insertion Workload Confidence Intervals

Method	95 Percentile (s)	99 Percentile (s)
Tracking Table	408	410
Column Family Scan	378	387
Differential	538	538
Snapshot	617	620
Audit Column	865	889

Table 4.15: 9.6GB Data Set 25% Insertion Workload Percentiles

Method	Mean (s)	95% Confidence	99% Confidence
Tracking Table	719	± 2.48	± 3.26
Column Family Scan	632	± 1.57	± 2.06
Differential	681	± 0.68	± 0.89
Snapshot	931	± 2.24	± 2.94
Audit Column	1278	± 2.23	± 2.93

Table 4.16: 9.6GB Data Set 75% Insertion Workload Confidence Intervals

Method	95 Percentile (s)	99 Percentile (s)
Tracking Table	739	743
Column Family Scan	636	651
Differential	684	685
Snapshot	943	946
Audit Column	1289	1291

Table 4.17: 9.6GB Data Set 75% Insertion Workload Percentiles

Notice that the snapshot differential technique has separated measurements. To extract deltas with this method, it is necessary to have at least two snapshots in order to run the differential phase, so the necessary time to execute it is twice the time of snapshot plus the necessary time o calculate the differential.

Finally, using the 95 percentiles values of each technique in the different presented update workloads, it is possible to generate the graphics presented in Figures 4.10 and 4.11. It is possible to see that the different update workloads does not affect significantly the time necessary to process the data.

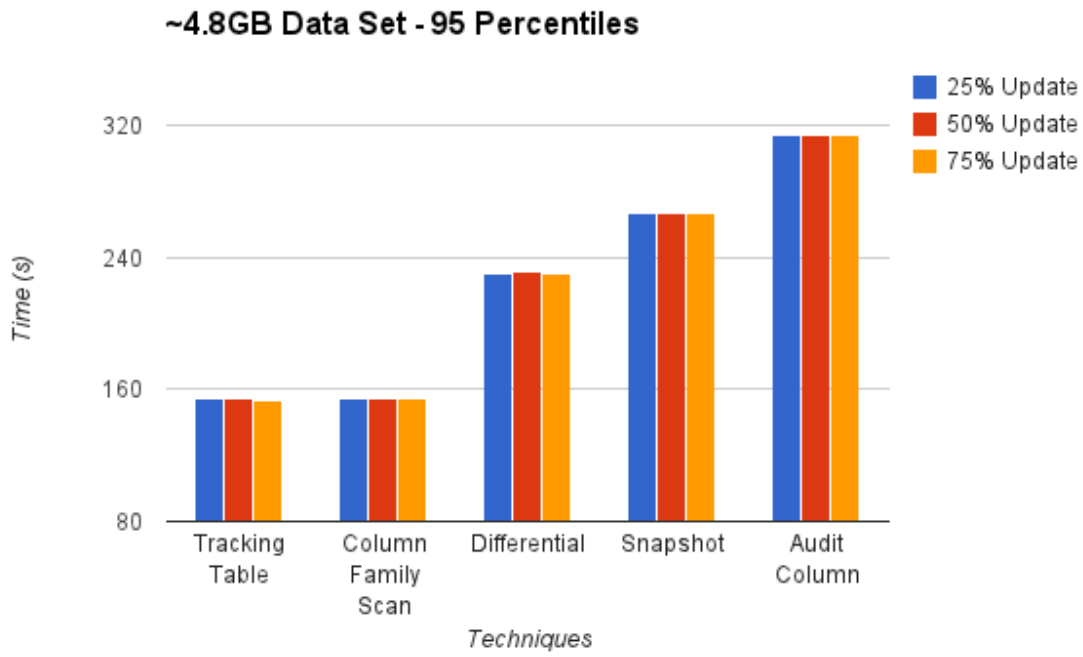

Figure 4.10: 4.8GB Data Set Update Workloads

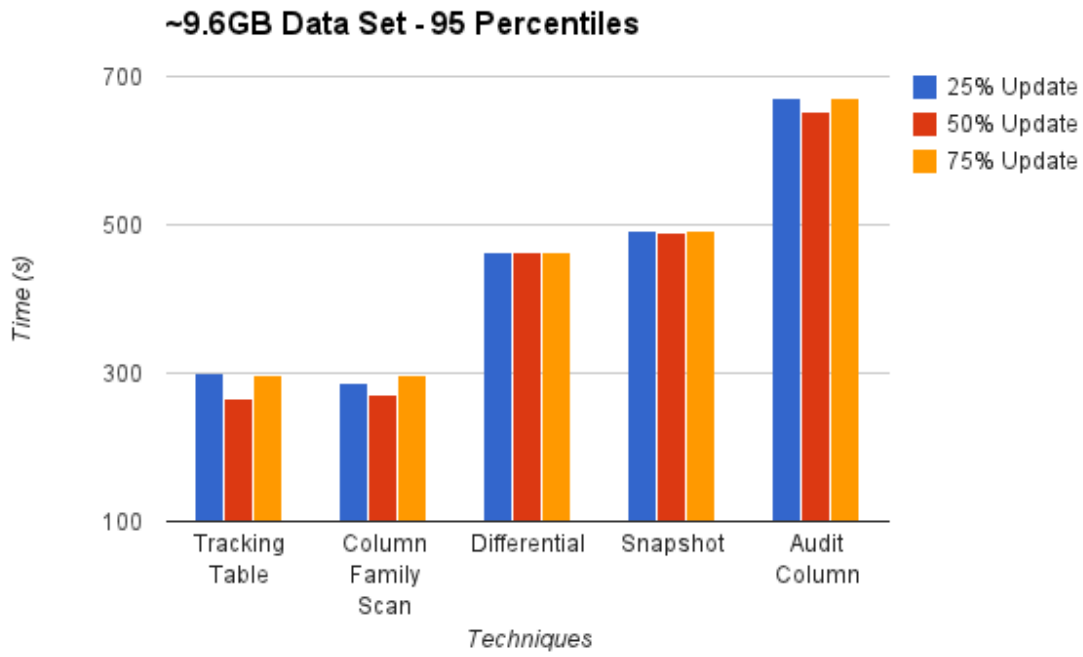

Figure 4.11: 9.6GB Data Set Update Workloads

On the other hand, Figures 4.12, 4.13 and 4.14 represent graphically the difference of time necessary to process the different update workloads for initial data sets of 4.8GB and 9.6GB. These figures show that the processing time is more sensitive to the input data volume rather than to the amount of deltas extracted.

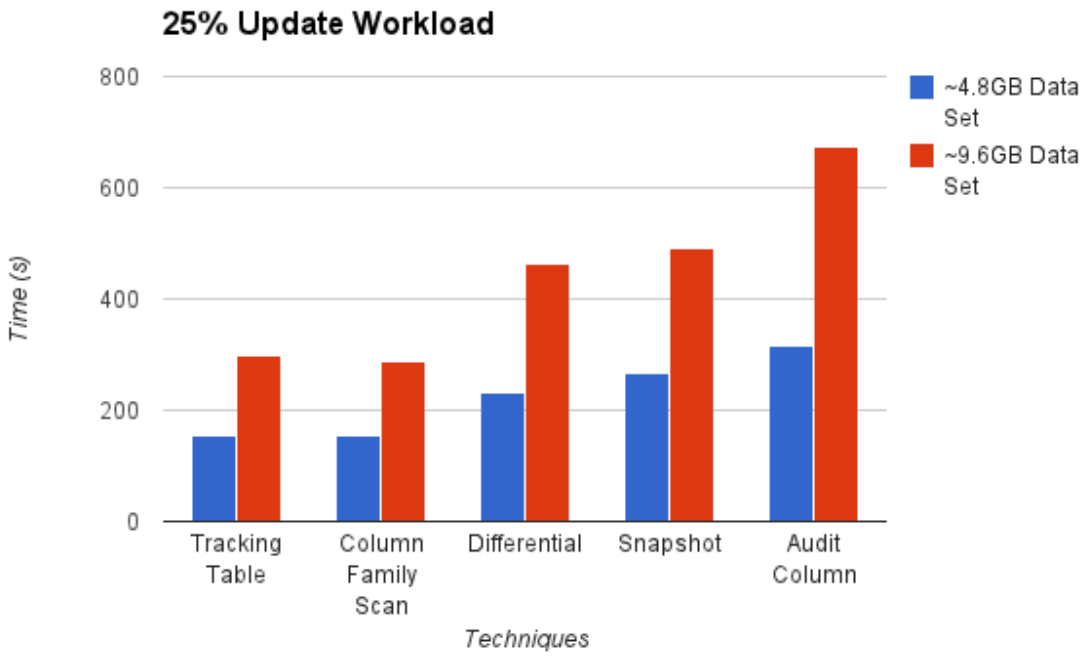

Figure 4.12: 25% Update Workload

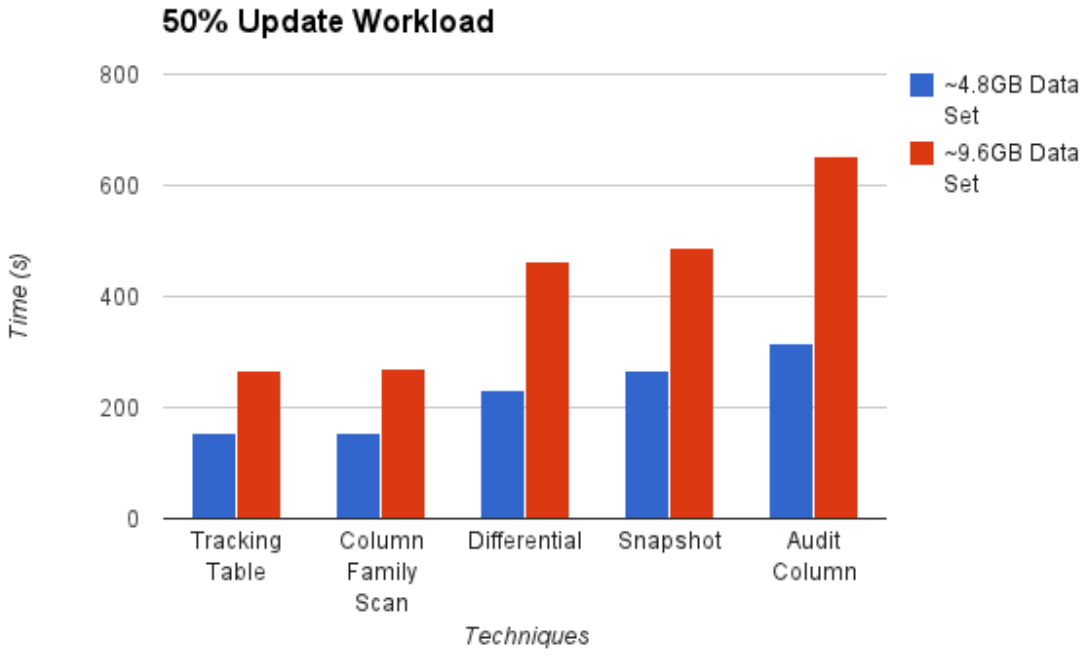

Figure 4.13: 50% Update Workload

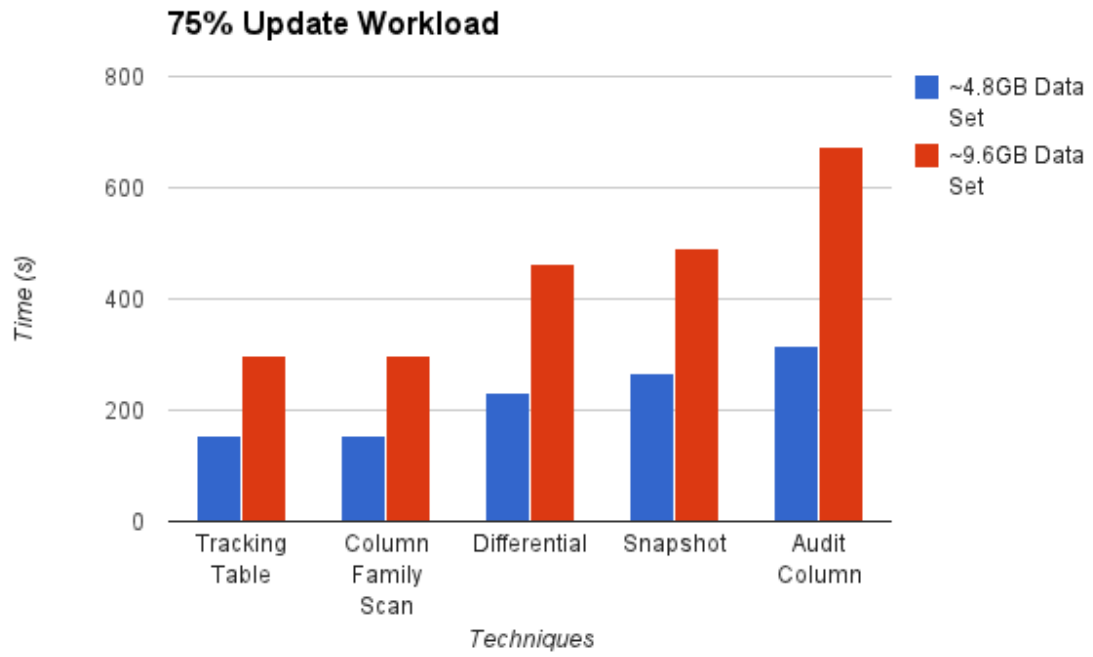

Figure 4.14: 75% Update Workload

With an initial data set of around 9.6GB, there are differences in the required processing time to the workloads of 25% and 75% of insertion operations, as presented in Figure 4.15. The difference is again related to the amount of data analyzed. In the first workload there are 25% more data than the initial database state and, in the second, the amount of data is 75% larger than the same initial state.

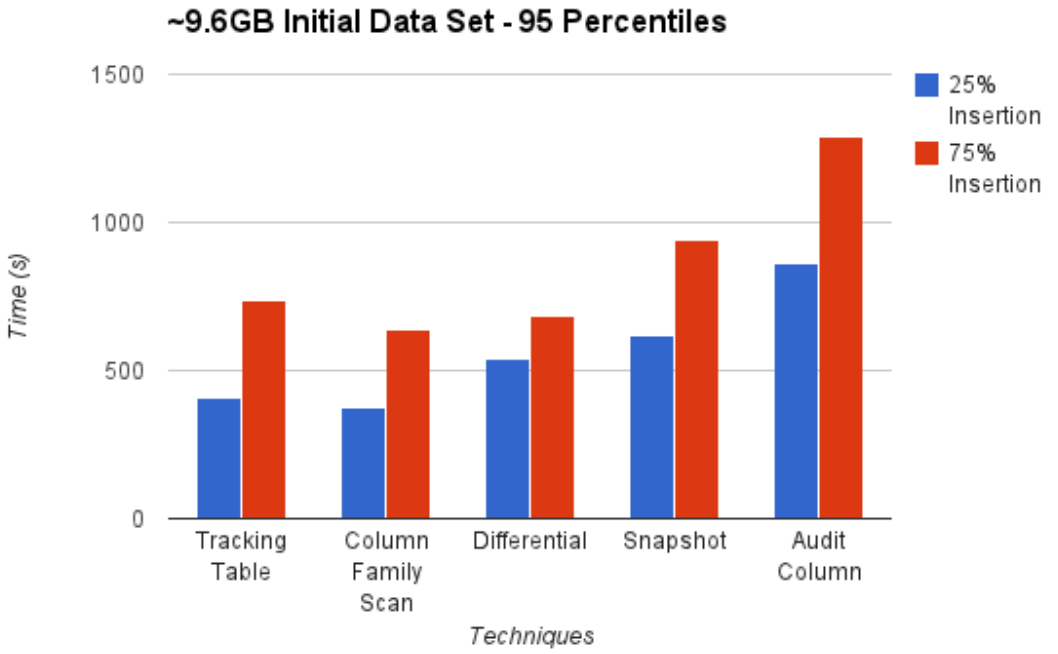

Figure 4.15: 9.6GB Initial Data Set Insertion Workloads

The presented results reflects the expected result based on the implemented logic of each presented logic. With an equal data set, there should not be significant processing time differences for the several possibilities of generated deltas. The reason is because the logic is basically a sequence of chained *if's*, which have a constant complexity. On the other hand, the analyzed amount of data dictates how many times the implemented logic must be executed. Thus, changing the perceived performance more significantly.

The perceived difference of performance for each presented approach are due to the complexity of each of them. The column scan and tracking table extracts deltas simple by scanning a column family, which makes them the less time consumption of the techniques. The snapshot differential has the disadvantage of requiring the double amount of data to process its differential phase, which make it takes almost the double amount of time for extract the deltas compared to tracking table of column family scan. The less time effective technique audit column is so because it analyzes the double amount of columns, specifically the audit columns.

5 DISCUSSION AND RELATED WORK

This chapter shows some of the state of the art related to this work. As mentioned, the techniques presented in this document are already implemented for data warehousing in relational databases. There are many textbooks and papers related to data warehousing area discussing about extracting changed data of SQL databases. However, such ideas are not largely studied in the field of NoSQL storage solutions.

Current data warehousing use all of the methods presented in Chapter 3. Normally, the most desirable techniques are those which do not interfere in the source database. With respect to this, it is possible to highlight snapshot differential and log based approaches. Trigger based methods are not so common because they require a modification in the source database, which possibly downgrades the efficiency and performance of services and applications using the database to store data. Furthermore, if it is necessary to do some modification in the source system, in many scenarios the trigger will need also to be maintained in order to handle such modifications.

In the following subsections, existing approaches for relational and column oriented databases are presented. The focus then is to give a short description of techniques related to the ones presented in this thesis.

5.1 Existing Approaches for Relational Databases

There are several CDC techniques implemented in different ETL tools as in (JöRG; DEßLOCH, 2011) and (KIMBALL; CASERTA, 2004). Relational databases normally keep a log containing all operations performed within it. An approach to extract changed data is to analyze the transactional log and extract deltas from it. Such method applied to relational databases can be named *Database Log Scraping*. Other very common technique is to create snapshots of the source database in different time points. With a comparison between snapshots, it is possible to define which data was altered and which was not. Finally, other commonly used approach is to append an additional column in the end of each table, storing the data modification date. This technique is named *Audit Column* and the additional column is normally maintained by a trigger. These are techniques commonly implemented in relational databases for data warehousing.

As extracted from (KIMBALL; CASERTA, 2004), Database Log Scraping technique *"takes a snapshot of the database redo log at a scheduled point in time (usually midnight) and scours it for transactions that affect the tables you care about for your ETL load"*. Also, in the same book is stated a similar technique called *Database Log Sniffing*, which

has the same functionality but extracting transactions by sniffing the redo log while it is being modified. The Log Based approach presented in this document is related to the last described approach; more specifically related to Database Log Scraping, since we suggest to capture the transactions by reading the log entirely rather than in an interactive way.

In the same book, the technique *Timed Extracts* is used to take changed data based on the modification date, similarly to the technique Column Family Scan presented in this thesis. Although this is a very simple technique, there are some considerations about the reliability of this approach in case of a crash while extracting changed data. As stated in the same work, *"manual intervention and data cleanup is required if the process fails for any reason"*.

Furthermore, an approach with the same name used here is described for relational databases: *Audit Column*. In (KIMBALL; CASERTA, 2004), a short description of this technique is given: *"Audit columns are usually populated via database triggers fired off automatically as records are inserted or updated. Some-times, for performance reasons, the columns are populated by the front-end application instead of database triggers. When these fields are loaded by any means other than database triggers, you must pay special attention to their integrity."* Such description fits also on the technique when applied to NoSQL databases.

Finally, *Process of Elimination* is a technique equivalent to Snapshot Differential. Also in relational databases, this approach *"has the advantage that rows deleted from the source can be detected"*. Although not the more efficient, this approach is described as the most reliable among others. One advantage is *"because the process makes a row-by-row comparison, looking for changes, it's virtually impossible to Extracting miss any data"*, as stated in (KIMBALL; CASERTA, 2004). All the techniques above are described in Chapter 3 of the related book, specifically in Part 3:Extracting Changed Data.

5.2 Existing Approaches for Column Oriented Databases

There are few studies that investigate how to extract changed data from column oriented databases. In (HU; DESSLOCH, 2013) the notion of a *delta* is defined. As in our work, a delta identifies uniquely where the change occurs and what kind of modification was performed. As extracted from the last cited work, a delta has the following structure:

- *OPts/Metadata/Identifier/Operation*

Where:

- *OPts* = date of the operation
- *Metadata* = metadata of source column family (e.g., column family name, column name
- *Identifier* = row key
- *Operation* = performed operation

A *Timestamp-Based* approach is also defined, which consist of scanning an entire table and extracting deltas for columns modified after determined time constraints. This is possible due to having modification date information as attribute of a column. The same concept is used in this thesis to create the Column Family Scan technique.

In this work, the approach Audit Column is an improvement of the Column Family Scan technique. Its goal is *"to address some drawbacks"* of Column Family scan, as in (HU; DESSLOCH, 2013) and the Audit Column and Timestamp-Based approaches. As previously described, with an additional column controlled by a trigger or the application used to store the insertion date, it is possible to distinguish between insertions and updates.

The techniques *Log Based*, *Snapshot Differential* and *Trigger Based* are described in the previously cited work and in (HU; QU, 2013). The main difference between these works and the work presented in this document is that here we build the models to fit the specifications of Apache Cassandra rather than Apache HBase. Another detail to be considered is that, because the cited works use HBase as the source system, they could successfully implement a full Trigger Based approach, due to the triggers support present in HBase. Such method uses triggers to maintain a tracking table, which will keep necessary modification information for later extraction of deltas.

Furthermore, the Log Based technique presented in the cited works extracts deltas by processing the write-ahead log (WAL) of HBase. In this work, we do the same, but by a deserialization of Cassandra's commitlog. Finally, the Snapshot Differential approach implemented either in the previously mentioned works or in this thesis, extracts deltas by creating snapshots of the desired source column family in different time points. Later on, the snapshots are compared, outputting all the detected deltas.

To conclude, the general ideas behind each of the previously cited techniques are very similar to the ones used in this document, with important modifications necessary to fit the models to Apache Cassandra.

6 CONCLUSIONS AND FUTURE WORK

This thesis presented all the important aspects of CDC solutions for Apache Cassandra. As depicted along this document, it is possible to generate several different models of CDC techniques and apply them on the proposed base systems. For all of the implementations, is is necessary to consider not just the performance observed while extracting changed data from a source database, but also overheads caused by the use of triggers. These overheads can be due to processing time or storing data.

Some techniques do not need modifications in the source database, e.g., column family scan, requiring just processing time to extract deltas. On the other hand, the trigger based approach for instance, needs a trigger added to the source system, which will keep a tracking table. Such technique requires not just processing time but also storage capacity. Furthermore, the presented solutions have limitations that must be also considered. Finally, each solution is capable of extracting specific types of deltas.

Given all explained before, it is not possible to take one CDC approach as the best. Depending on the application and its requirements, one solution might fit better than the others. In the end, it is up to the developer the choice of which technique to use, taking into account all the limitations and capacities of each solution given in this document.

Finally, as presented in Chapter 5, there is few work for CDC and NoSQL databases. None of them are applied to Apache Cassandra. The contribution of this work is to give different models of CDC techniques applied to Apache Cassandra, also evaluating their functions and their performances in different workload scenarios.

6.1 Future Work

Because nowadays Apache Cassandra does not support a reliable and final trigger model, it is necessary to wait the release of version 2.1 of this database. This release will support triggers in its final state, so it will be possible to integrate our implementations with triggers. Therefore, it will be necessary to port all the code to the newer database version. Furthermore, there is a newer version of the Map Reduce framework, which could also be supported. With this new version, we can program the algorithms and integrate them into a single API. This API could even be part of the Apache Cassandra in the future.

Sometimes we used the Cassandra Thrift API to interact with the database, which is discouraged because the API reveals a lot of the internals of the system. This API will be changed to Cassandra Query Language (CQL), which will probably open new

possibilities that could be explored.

Furthermore, there are other structures in Apache Cassandra that this thesis does not cover, for example counter columns and expiration columns. The presented models can be extended to also take care of such structures. Moreover, it would be interesting to have different models for different target systems, e.g., relational databases and column oriented databases. In this work we considered Apache Cassandra also as the target system which will handle the extracted deltas. For other target systems, it is probably necessary to modify the techniques presented in this work in order to satisfy the target system's requirements.

Finally, performance tests for different operations can be performed to further analyze the behavior of the techniques in several distinct workloads.

REFERENCES

CHANG, F. et al. Bigtable: a distributed storage system for structured data. In: OSDI'06: SEVENTH SYMPOSIUM ON OPERATING SYSTEM DESIGN AND IMPLEMENTATION, SEATTLE, WA, NOVEMBER, 2006. **Anais...** [S.l.: s.n.], 2006. p.205–218.

CHODOROW, K.; DIROLF, M. **MongoDB - The Definitive Guide**: powerful and scalable data storage. [S.l.]: O'Reilly, 2010. I-XVII, 1-193p.

COOPER, B. **Yahoo! Cloud Service Benchmark**. [Online; accessed 25-May-2014], `https://github.com/brianfrankcooper/YCSB`.

COULOURIS, G. et al. **Distributed Systems**: concepts and design. 5th.ed. USA: Addison-Wesley Publishing Company, 2011.

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. In: OSDI'04. **Anais...** [S.l.: s.n.], 2004. p.205–218.

DECANDIA, G. et al. Dynamo: amazon's highly available key-value store. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.41, n.6, p.205–220, 2007.

GEORGE, L. **HBase**: the definitive guide. 1.ed. [S.l.]: O'Reilly Media, 2011.

GROFF, J.; WEINBERG, P. **SQL The Complete Reference, 3rd Edition**. 3.ed. New York, NY, USA: McGraw-Hill, Inc., 2010.

HEWITT, E. **Cassandra - The Definitive Guide**: distributed data at web scale. [S.l.]: Springer, 2011. I-XXIII, 1-301p.

HU, Y.; DESSLOCH, S. Extracting Deltas from Column Oriented NoSQL Databases for Different Incremental Applications and Diverse Data Targets. In: CATANIA, B.; GUERRINI, G.; POKORNý, J. (Ed.). **Advances in Databases and Information Systems**. [S.l.]: Springer Berlin Heidelberg, 2013. p.372–387. (Lecture Notes in Computer Science, v.8133).

HU, Y.; QU, W. Efficiently Extracting Change Data from Column Oriented NoSQL Databases. In: PAN, J.-S.; YANG, C.-N.; LIN, C.-C. (Ed.). **Advances in Intelligent Systems and Applications - Volume 2**. [S.l.]: Springer Berlin Heidelberg, 2013. p.587–598. (Smart Innovation, Systems and Technologies, v.21).

JAIN, R. **The Art of Computer Systems Performance Analysis**. [S.l.]: Wiley, 1991.

JöRG, T.; DEßLOCH, S. View Maintenance using Partial Deltas. In: BTW. **Anais...** GI, 2011. p.287–306. (LNI, v.180).

KIMBALL, R.; CASERTA, J. **The Data Warehouse ETL Toolkit**: practical techniques for extracting, cleaning, conforming, and delivering data. Indianapolis, IN: Wiley, 2004.

LABIO, W.; GARCIA-MOLINA, H. Efficient Snapshot Differential Algorithms for Data Warehousing. In: VLDB. **Anais...** Morgan Kaufmann, 1996. p.63–74.

MICROSOFT. **Change Data Capture**. [S.l.]: Microsoft Corporation, 2008. [06.12.2008].

RAMANATHAN, S.; GOEL, S.; ALAGUMALAI, S. Comparison of Cloud database: amazon's simpledb and google's bigtable. In: RECENT TRENDS IN INFORMATION SYSTEMS (RETIS), 2011 INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2011. p.165–168.

SCHILDGEN, J.; JöRG, T.; DEßLOCH, S. Inkrementelle Neuberechnungen in MapReduce. **Datenbank-Spektrum**, [S.l.], v.13, n.1, p.33–43, 2013.

SHI, J. et al. Study on Log-Based Change Data Capture and Handling Mechanism in Real-Time Data Warehouse. In: CSSE (4). **Anais...** [S.l.: s.n.], 2008. p.478–481.

VASSILIADIS, P. A Survey of Extract-Transform-Load Technology. In: TANIAR, D.; CHEN, L. (Ed.). **Integrations of Data Warehousing, Data Mining and Database Technologies**. [S.l.]: Information Science Reference, 2011. p.171–199.

AppendixA

A.1 Commit Log Structures

(Keyspace)(Key)(ModificationSize)
[(ColumnFamilyID)(isNotNull)(ColumnFamilyID)
(LocalDeletiontime)(MarketForDeletionAt)
(NumColumns)(ColumnName)
(SerializationFlag)(?)(Timestamp)
(Value)]^{nc}

^{nc} = NumColumns

case (SerializationFlag):
    COUNTER_MASK: ? = (timestampOfLastDelete)
    EXPIRATION_MASK: ? = (ttl)(expiration)
    COUNTER_UPDATE_MASK || DELETION_MASK: ? = empty

Figure A.1: Standard Column Log Row Structure

(Keyspace)(Key)(ModificationSize)
{(ColumnFamilyID)(isNotNull)(ColumnFamilyID)
(LocalDeletiontime)(MarketForDeletionAt)
(NumSuperColumns)(SuperColumnName)
(LocalDeletiontime)(MarketForDeletionAt)
(NumColumns)[(ColumnName)
(SerializationFlag)(?)(Timestamp)
(Value)]^{nc}}^{nsc}

^{nc} = NumColumns
^{nsc} = NumSuperColumns

case (SerializationFlag):
COUNTER_MASK: ? = (timestampOfLastDelete)
EXPIRATION_MASK: ? = (ttl)(expiration)
COUNTER_UPDATE_MASK || DELETION_MASK: ? = empty

Figure A.2: Super Column Log Row Structure

CommitLogHeader:
HashMap<ColumnFamilyID, FilePosition>
Followed by:
(Length)(ChecksumLenght)
(SerializedRowMutation)(ChecksumRowMutation)

Serialized Row Mutations:
(KeyspaceName)(Key)(ModificationsSize)
{(ColumnFamilyID)(SerializedColumnFamily)}^{n}

Serialized Column Family:
(HasColumns?)(ColumnFamilyID)(LocalDeletionTime)
(MarketForDeleteAt)(NumColumns)
{(SerializedStandardOrSuperColumn)}^{n}

Serialized Super Column:
(SuperColumnName)(LocalDeletionTime)
(MarketForDeletedAt)(NumberSubColumns)
(SerializedStandardColumns)

Serialized Standard Column:
(Name)(SerializationFlag)(0 || 1 || 2)(Timestamp)Value)
0 = Column: ()
1 = CounterColumn: (TimestampOfLastDelete)
2 = ExpiringColumn: (TimeToLive)(LocalDeletionTime)

Serialization Flags:
0x01: DELETION_MASK (DeletedColumn)
0x02: EXPIRATION_MASK
0x04: COUNTER_MASK (CounterColumn)
0x08: COUNTER_UPDATE_MASK (CounterUpdateColumn)

Figure A.3: Commit Log General Pattern

(Keyspace) = String (UTF-8)
(Key) = ByteBuffer
(ModificationSize) = int
(ColumnFamilyID) = int
(isNotNull) = boolean
(LocalDeletiontime) = int
(MarketForDeletionAt) = long
(NumSuperColumns) = int
(SuperColumnName) = ByteBuffer
(NumColumns) = int
(ColumnName) = ByteBuffer
(SerializationFlag) = unsigned int
(timestampOfLastDelete) = long
(ttl) = int
(expiration) = int
(timestamp) = long
(value) = ByteBuffer

Figure A.4: Fields Java Types

DELETION_MASK = 0x01
EXPIRATION_MASK = 0x02
COUNTER_MASK = 0x04
COUNTER_UPDATE_MASK = 0x08

Figure A.5: Serialization Flags