Khanan: Performance Comparison and Programming Alpha Algorithm in Column-Oriented and Relational Database Query Languages

Astha Sachdev

Computer Science

Indraprastha Institute of Information Technology, Delhi (IIIT-D), India

A Thesis Report submitted in partial fulfilment for the degree of

MTech Computer Science

13 February 2015

1.: Prof. Ashish Sureka (Thesis Adviser)

2. Prof. Sambuddho Chakravarty (Internal Examiner)

2. Prof. Subhash Bhalla (External Examiner)

Day of the defense: 13 February 2015

Signature from Post-Graduate Committee (PGC) Chair:

Abstract

Process mining consists of mining business process event-logs for discovering run-time process models, process compliance verification and extracting useful insights on process efficiency. Process model discovery from event-logs is one of the most important and challenging process mining tasks. Process model discovery consists of learning a System Net (such as a Petri Net) from an event log. The α-algorithm is first and most widely used process discovery technique. There are several extensions proposed to α-algorithm but we use the basic α-algorithm as a baseline and benchmark algorithm for our study. We present a CQL (Cassandra Query Language) and SQL (Structured Query Language) implementation of the basic α-algorithm (translation of α-algorithm computations into CQL and SQL). Column-oriented databases have shown to improve the performance of several functions and algorithms that require analytical query processing on a large dataset. We conduct a benchmarking study consisting of a series of experiments on a large real-world dataset to compare the performance of the α-algorithm CQL and SQL implementations.

I dedicate my MTech Thesis to my father Anil Sachdev who has always encouraged me in all phases of life and is my greatest source of inspiration.

Acknowledgements

I would take this wonderful oppurtunity to express my deepest gratitude to my advisor Prof. Ashish Sureka for his continuous guidance, support, constant motivation and patience throughout my thesis. Without his guidance and support this thesis would not have been possible. I feel really blessed to have him as my thesis advisor.

I would also like to thank my fellow mate Kunal Gupta for his insightful comments, suggestions and constant support during the course of my thesis. I would like to thank God for all his blessings.

Finally, I would like to thank my parents and brother for their constant support, encouragement, love and trust in me.

Declaration

This is to certify that the MTech Thesis Report titled **Khanan: Performance Comparison and Programming Alpha Algorithm in Column-Oriented and Relational Database Query Languages** submitted by **Astha Sachdev** for the partial fulfillment of the requirements for the degree of *MTech* in *Computer Science* is a record of the bonafide work carried out by her under my guidance and supervision at Indraprastha Institute of Information Technology, Delhi. This work has not been submitted anywhere else for the reward of any other degree.

Professor Ashish Sureka
Indraprastha Institute of Information Technology, New Delhi

Contents

List of Figures

List of Tables

1

Research Motivation and Aim

Relational databases are very good at solving certain data storage problems but they can create problems when it is time to scale. When the size of the dataset increases the time taken to compute joins heavily increases. In such cases we need to find a way to get rid of the joins and denormalize the data. Recent years have seen the introduction of a number of column oriented database systems. These database systems have been shown to perform more than an order of magnitude better than the traditional relational database systems on analytical workloads [2]. This is because column stores are more I/O efficient for read only queries since they only have to read from disk or from memory those attributes that are accessed by a query [2].

The volume of data in an organization is increasing at a very fast rate. With this, the need to store this data to in order to make critical business decisions and satisfy user demands becomes very important. Both users and decision makers need a faster and more convenient way to access the data as quickly as possible. These factors put a lot of pressure on the IT systems. The data warehouses storing this data used the traditional row oriented systems. Though the relational model supports both transactional and analytical processing, as the size of the data warehouses kept increasing, a new approach to store and handle such large amounts of data was introduced. This approach is the column-oriented approach which stores the data in a column oriented manner i.e., the data storage layer in column oriented databases contains columns instead of rows [3]. In a relational model, data is stored and managed as rows. Each row contains all the attributes of an element of a particular entity. This model is well suited for transactional applications which access and process almost all the attributes

of a record. However, analytical applications need to read only a few attributes of a large number of records. Thus, if row oriented approach is used for storing data in analytical applications, then a large amount of unuseful data needs to be read in the memory. Hence, for analytical applications column oriented databases have shown better performance over row oriented databases as all instances of a single column are stored together in column oriented databases and can be accessed together. Also, since column oriented databases read only the specific columns required by a query and not the irrelevant ones, the operations to be performed on a column database can be completed with less I/O operations. Though a relational database can also be partitioned vertically and an index can be created for every column for faster data retrieval, still the performance of column oriented databases have proven to perform better than row oriented databases [2]. Also, column oriented databases need a smaller disk space to store data rather than row oriented databases as they do not require additional storage for indexes, because data is stored within the indexes themselves.

Process mining focuses on the analysis of processes using event data. One of the key aspects of process mining is to generate process models that can be used for analysis purposes in the growing business needs [1].Thus process mining is basically an analytical application. Both column and row oriented databases have been used for data storage and optimal retrieval, but column oriented databases have shown to perform better over row oriented databases for analytical workloads, especially in terms of disk storage. The query languages SQL(for row oriented database) and CQL(for column oriented database) have been growing over the years and have become a standard way of interacting with the database. In our paper, we attempt to implement a process mining algorithm in these database languages to the extent possible.
The research motivation of the study presented in this paper is to implement α-miner algorithm, which was one of the first process mining algorithms, on row oriented data store MySQL and column oriented data store Cassandra and investigate the performance of the algorithm on both the databases. The specific research aim of the work presented in this paper are the following:

1. To implement α-miner algorithm in a row-oriented data store MySQL.

2. To implement α-miner algorithm in a column-oriented data store Cassandra.

3. To compare the performance of α-miner algorithm in MySQL and Cassandra database.

2

CAP Theorem and Cassandra Architecture

Cassandra is an open source, column oriented, distributed, decentralized, elastically scalable, highly available, fault tolerant and tuneably consistent database [4]. Cassandra is distributed which means that it is capable of running on multiple machines. It is also decentralized. This means that there is no single point of failure in Cassandra. This means that all the nodes in a Cassandra cluster function exactly the same and there is no master/slave relationship between the nodes. This mechanism of decentralization in Cassandra is one of the key reason for Cassandra's higher availability. Cassandra is elastically scalable. This means that we can keep adding or removing nodes from a Cassandra cluster without any major disruption or reconfiguration of the whole cluster. Once a new node is added to the cluster, Cassandra finds it and starts sending it work. Cassandra is highly available. All the nodes in Cassandra function exactly the same way and data is replicated to multiple node within a cluster. Thus, even if a node fails, it can be replaced with no downtime. Cassandra is eventually consistent. This means that if an update is made to data at a particular node in Cassandra, then that update will propagate to all other replica nodes within some time i.e. all the replica nodes will eventually be consistent.

When we consider large scale distributed systems, we have three main requirements:

1. Consistency: It requires that all database clients read the same value for a given query.

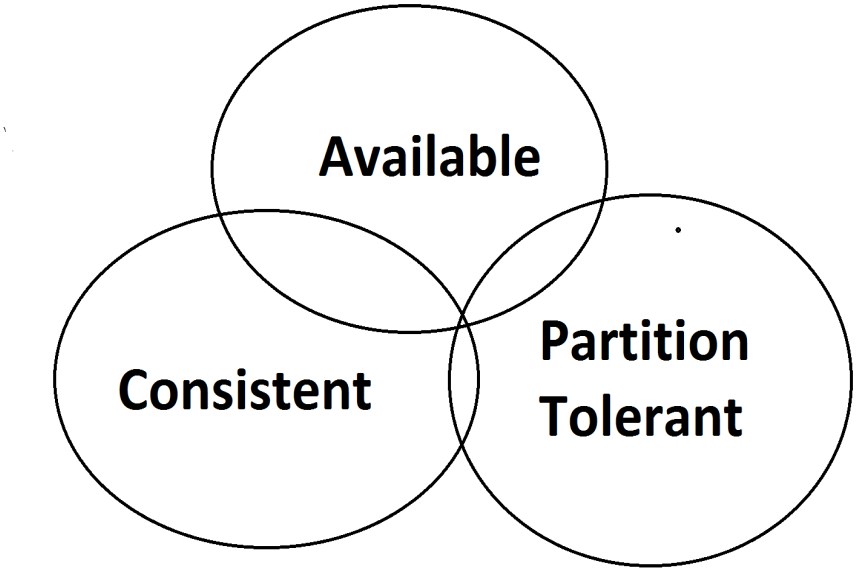


Figure 2.1: CAP Theorem
(Image taken from [5])

2. Availability: It requires that the database should always be available, even if a node fails. It requires that a database client should always have the database available to it for reading and writing data.

3. Partition Tolerant: It requires that there database should be able to function even when there is a break in the network.

Eric Brewer introduced the CAP Theorem [5], according to which in any distributed system we can strongly support only two of the above three requirements.As can been seen from Fig. 2.1, there is no overlapping region between all the three requirements. Cassandra database supports availabilty and partition tolerance. Hence, Cassandra is a highly available and partition tolerant system.

Cassandra is a distributed key value store and allows data to be queried only by its key. Cassandra does not implement the concept of referential integrity and hence, there are no joins in Cassandra. Thus, the data in stored in denormalized form in Cassandra database. The most fundamental unit of Cassandra is a column and it is expressed in the form of name:value. In Cassandra, the primary key consists of partition

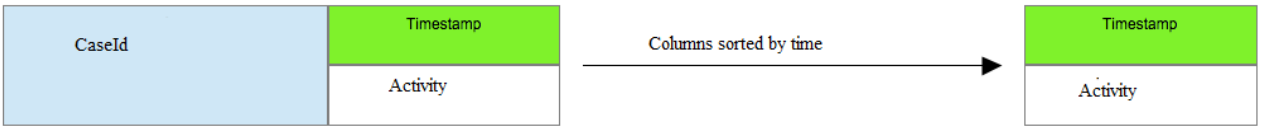


Figure 2.2: Cassandra Data Model

caseid	dateplustime	activity
1-736720411	2104-03-22 15:59:42India Standard Time	In Progress
1-736720411	2104-03-23 16:00:56India Standard Time	In Progress
1-736720411	2104-03-24 15:59:42India Standard Time	Awaiting Assignment
1-736720411	2104-03-25 16:00:56India Standard Time	In Progress
1-736720411	2104-03-26 15:59:42India Standard Time	Awaiting Assignment
1-736720411	2104-03-27 16:00:56India Standard Time	In Progress
1-736720411	2104-03-28 15:59:42India Standard Time	Resolved
1-736720411	2104-03-29 16:00:56India Standard Time	Closed
1-738247053	2120-01-20 15:59:42India Standard Time	In Progress
1-738247053	2120-01-21 16:00:56India Standard Time	In Progress

Figure 2.3: Cassandra Data Sorted by Time

key and clustering key. If only one field has been defined as the primary key then it is the partition key. The partition key determines which node stores the data.The other column, in the primary key definition is the clustering column.The data for each partition is clustered by this column. On a physical node, when rows for a partition key are stored in order based on the clustering column, retrieval of rows is very efficient.

For example, when we load our CSV file(eventlog file) in the eventlog table in Cassandra, we use casid and timestamp as the primary key:

```
CREATE TABLE eventlog
(
caseid text ,
datetime timestamp,
activity text,
PRIMARY KEY(caseid,datetime)
);
```

Here, caseid and datetime form the primary key. Thus, caseid is the partition key and datetime is the clustering column. The way Cassandra works is that it maps the caseid to the internal row key and datetime to the internal cell name which in turn is mapped to the cell value activity.

Cassandra's data model is excellent in terms of handling data in sequence. When data is written to Cassandra, it is sorted and written sequentially to disk. Thus, when we retrieve data by row key and then by range, we get a fast retrieval. Fig 2.2, shows how data is sorted in Cassandra. Fig. 2.3 shows how our data is stored in Cassandra for a particular caseid. As can been seen from the figure, the data is sorted by timestamp values. Hence, Cassandra is excellent for storing time series data.

3

Related Work and Novel Research Contributions

In this Section, we review closely related work to the study presented in this paper and list the novel contributions of our work in context to existing work. We divide related work into following three lines of research:

3.0.1 Implementation of Mining Algorithms in Row Oriented Databases

Ordonez et al. present a method to implement k-means clustering algorithm in SQL. They cluster in large datasets in RDBMS [6]. Their work concentrates on defining suitable tables, indexing them and writing suitable queries for clustering purposes. Ordonez et al. presents an efficient SQL implementation of the EM algorithm to perform clustering in very large databases [7].The implementation presented by them can handle a large number of data records,high dimensional data and a large number of clusters. They present three strategies to present to implement EM in SQL:horizontal strategy, vertical strategy and a hybrid strategy. Berzal et al. presents a TreeBased Association Rule Mining [8] to discover interesting patterns in relational database. They conduct experiments on real world datasets to and show that Tree-Based Association Rule Mining outperforms Apriori algorithm Sattler et al. present a study of applying data mining primitives on decision tree classifier [9]. Their framework provides a tight coupling of data mining and database systems and links the essential data mining primitives that supports several classes of algorithms to database systems.

3.0.2 Implementation of Mining Algorithms in Column Oriented Databases

Mehta et al. conducted a study on the impact of data mining algorithms on column oriented database systems [10]. They study the architecture of various open source column oriented database systems and implement simple tree based classification algorithm on MonetDB and discretization algorithm on MonetDB and Infobright. Rana et al. conducted experiments on the utilization of column oriented databases like MonetDB with oracle 11g which is a row oriented data store for execution time analysis of Association Rule Mining algorithm [11]. Suresh L et al. implemented k-means clustering algorithm on column store databases [12]. They introduce a Novel Seeding Algorithm to implement k-means in column store databases. This algorithm identifies the median gaps in the data in each of the columns and using these median gaps it identifies other clusters by identifying the difference in the median gaps.

3.0.3 Performance Comparison of Mining Algorithms in Row-Oriented and Column Oriented Databases

Bazar et al. present a study on the reasons for the transition from relational to column oriented databases [13]. They conduct experiments to perform benchmarking on three column store databases Cassandra, MongoDB and CouchBase. Pungila et al. conduct an experiment to test collection speed and aggregation speed for reasonable data streams of sensor data on relational databases and column stores and perform benchmarking on them [14]. Mauroux et al. perform a qualitative study of characterizing and comparing column stores for RDF processing [15]. Rana et al. implement Apriori algorithm on MonetDB and Oracle database and compare their performance in terms of execution time [11].Suresh L et al. implemented k-means clustering algorithm on column store databases [12]. They implement clustering of large datasets stored inside a relational database. Their work concentrates on defining suitable tables,indexing them and optimizing queries for clustering. The experiments performed by them show that he aggregations on column store were efficient as compared to row store and the difference calculations in a row store DBMS out performed that on column store DBMS.

In context to existing work, the study presented in this paper makes the following novel contributions:

1. While there has been work done on implementing data mining algorithms in row oriented databases, we are the first to implement process mining α miner algorithm in row oriented database MySQL using Structured Query Language(SQL).

2. While column oriented data stores have been used to implement data mining

algorithms, they have never been used to implement and study the performance of process mining algorithm. We in this paper present an implementation of α miner algorithm in Cassandra using Cassandra Query Language(CQL).

3. We present a performance benchmarking of α miner algorithm on both row oriented database MySQL and column oriented database Cassandra.

4

Process Mining and α Miner Algorithm

Process Mining basically focuses on the analysis of processes using event data. The various data mining techniques such as classification, association, clustering are generally used to analyze a particular step in the overall business process but cannot be applied to understand and analyze a process as a whole. Process mining extracts knowledge from event logs. It helps to discover, analyze and improve a process model [1].
Each event in an event log refers to an activity which is a well defined step in some process. Each of this activity is associated with a particular caseid i.e., a process instance. The events belonging to a particular caseid are ordered. An event log may also contain additional information such as timestamp associated with each event.

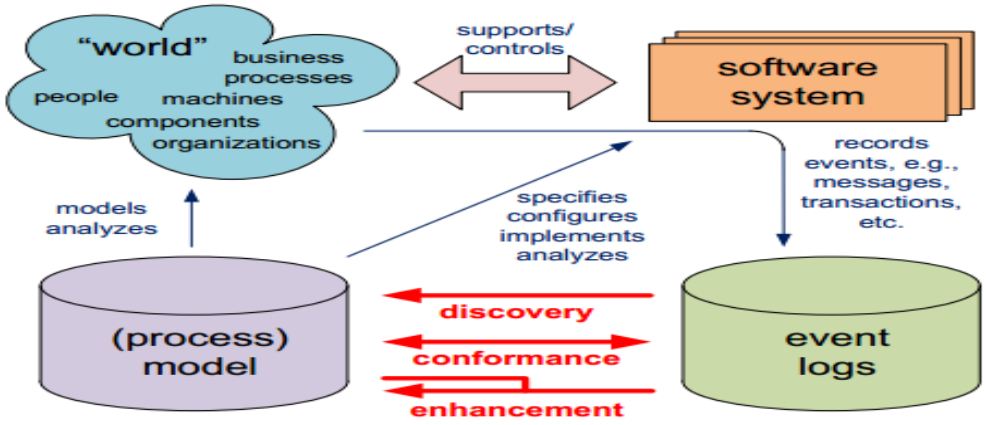


Figure 4.1: Types of Process Mining Techniques (Figure taken from [1])

As can be seen from Fig. 4.1, there are basically three types of process mining techiques:

1. Process Discovery: This technique takes an event log as an input and produces a process model without using the information stored in the event log only.

2. Process Conformance: This technique takes an existing process model as a reference and compares it with the given event log to check if the given event log conforms to the process model and vice versa.

3. Process Enhancement: This technique extends or improves the existing process model. It takes the existing process model as input and tries to extract new information from it.

We consider three different perspectives in process mining [16]:

1. Process Perspective: This perspective focuses on the control flow of a process i.e. the ordering of the tasks in an event log. It aims to generate a process model from the event log.

2. Organizational Perspective: This perspective focuses on how the originators of various activities interact with each other.

3. Case Perspective: This perspective focuses on the property of a particular process instance. There are different ways in which a process instance can be identified, for example, it can be identified by the path it takes in a process model or by the originators working on it.

In this paper we implement α-miner algorithm which was one of the first process mining algorithm. The α-miner algorithm focuses on **process discovery in the process perspective**.

The α-miner algorithm is an algorithm used in process mining, aimed at discovering causality from a set of sequences of events [16]. It was first put forward by van der Aalst,Weijter and Maruster. It constructs a process model with special properties from event logs.
Input for the alpha algorithm is an event log L. The α-miner algorithm scans the event log for particular patterns. It computes ordering relations of the events contained in the log and deals with concurrency of activities. The basic ordering relations determined by α Miner Algorithm are the following:

1. a $\succ_L$b iff a directly precedes b in some trace.

2. a $\rightarrow_L$b iff a$\succ_L$b $\wedge$ b$\succ_L$a.

3. a$\|$b iff a$\succ_L$b and b$\succ_L$a in some trace.

4. a$\sharp$b iff a$\succ_L$b $\wedge$ b$\succ_L$a.

4.0.4 Algorithm

Let L be an event log over T $\subseteq$ A. α(L) is defined as follows.

1. TL = { t $\in$ T | $\exists$ σ $\in$ L t $\in$ σ }

2. TI = { t $\in$ T | $\exists$ σ $\in$ L t = first(σ) }

3. TO = { t $\in$ T | $\exists$ σ $\in$ L t = last(σ) }

4. XL = { (A,B) | {A $\subseteq$ TL $\wedge$ A = $\emptyset$ } $\wedge$ {B $\subseteq$ TL $\wedge$ B = $\emptyset$ } $\wedge$ { $\forall$ a $\in$ A $\forall$ b $\in$ B a $\rightarrow_L$b } }

5. YL = { (A,B) $\in$ XL | (A,B)$\in$ XL A $\subseteq$ A $\wedge$ B $\subseteq$ B $\longrightarrow$ (A,B)=(A,B)}

6. PL = { P(A,B) | (A,B) $\in$ YL } $\bigcup$ (iL,oL)

7. FL = { (a,P(A,B)) | (A,B) $\in$ YL $\wedge$ a $\in$ A } $\bigcup$ { (P(A,B),b) | (A,B) $\in$ YL $\wedge$ b $\in$ B } $\bigcup$ { (iL,t) | t $\in$ TI } $\bigcup$ { (t,oL) | t $\in$ TO }

8. α(L) = (PL,TL,FL)

The stepwise description of the α algorithm can be given as:

1. The first step computes TL which is the set of all the distinct activities present in the eventlog L.

2. The second step computes TI which computes the set of all the initial activities i.e. all those activities which appear in the beginning of some trace in the eventlog.

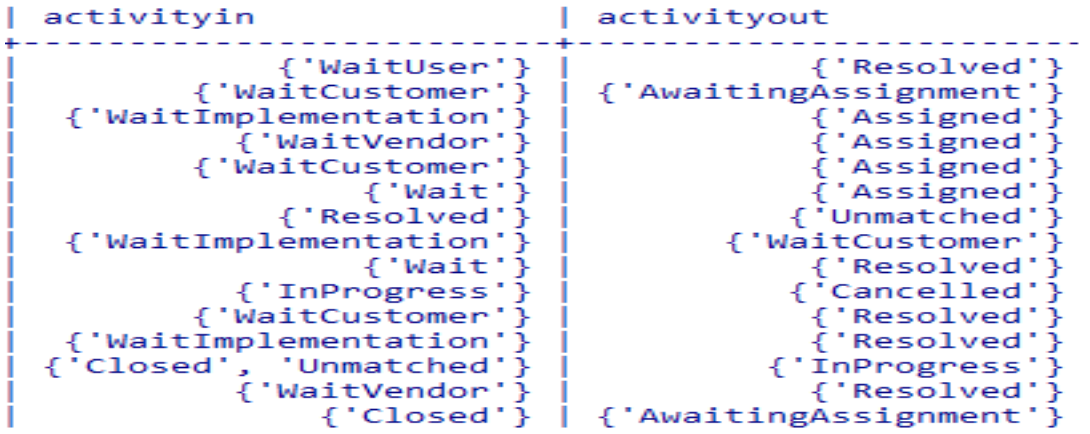
3. The third step computes the end activities i.e. all those activities which appear at the end of some trace in the eventlog.

4. In order to compute the fourth step, we compute the relationships between all the activities in TL. This computation is presented in the form of a footprint matrix and is called preprocessing in α algorithm. Using the footprint matrix we compute pairs of sets of activities such that all activities in the same set are not connected to each other while every activity in first set has causality relationship to every other activity in the second set.

5. In the fifth step we keep only the maximal pairs of sets generated in the fourth step, eliminating the non-maximal ones.

6. In the sixth step we add the input place which is the source place and the output place which is the sink place in addition to all the places generated in the fifth step.

7. The seventh step is the final step of the α algorithm that presents all the places including the input and output places and all the input and output transitions from the places.



activityin	activityout
{'WaitUser'}	{'Resolved'}
{'WaitCustomer'}	{'AwaitingAssignment'}
{'WaitImplementation'}	{'Assigned'}
{'WaitVendor'}	{'Assigned'}
{'WaitCustomer'}	{'Assigned'}
{'Wait'}	{'Assigned'}
{'Resolved'}	{'Unmatched'}
{'WaitImplementation'}	{'WaitCustomer'}
{'Wait'}	{'Resolved'}
{'InProgress'}	{'Cancelled'}
{'WaitCustomer'}	{'Resolved'}
{'WaitImplementation'}	{'Resolved'}
{'Closed', 'Unmatched'}	{'InProgress'}
{'WaitVendor'}	{'Resolved'}
{'Closed'}	{'AwaitingAssignment'}

Figure 4.2: α Algorithm-Input and Output Transitions

We use the experimental dataset as described in Chapter 5. The input to the α algorithm is the event log shown in Fig. 5.1. We obtain the output in the form of a table that shows the input and the output transitions. Fig 4.2, shows a part of the output we obtain in α algorithm. This part is the main part of the algorithm as it shows the input and the output transitions. The column activityin represents the set of input activities and the column activityout represents the set of outputactivities. All the activities in the set represented by activityin are not connected to each other and all the activities in the set represented by activityout are not connected to each other. All the activities in the set represented by activityin have causality relationship with all other activities represented by the set in activityout. This figure is a snapshot of the YW table that we obtain in Cassandra.

5

Experimental Dataset

We use Business Process Intelligence 2013(BPI 2013) dataset to conduct our experiments. The log contains events from an incident and problem management system called VINST. The event log has been prepared from Volvo IT Belgium. The data is related to the process of serving requests in an IT department. The process describes how a problem is handled in the IT services operated by Volvo IT. Tha dataset is provided in CSV format. We use the VINST cases incidents CSV file dataset, which contains 65533 records to conduct our experiments. The various fields in the dataset can be explained as:

1. Problem Number: It represents the unique ticket number for a problem. It is represented as caseid in our data model.
2. Problem Change Time: It represents the time when the status of the problem changes. It is represented as timestamp in our data model.
3. Problem Status: It describes the status of the problem whether the problem has been accepted, completed or closed.
4. Problem Substatus: It represents the current state of the problem. It is represented as activity in our data model.
5. Impact: It specifies the level of the impact -high, medium or low, the problem creates for the customer.
6. Organization: It specifies the business area of the helpdesk which the user reports to.
7. Function Division: The IT organization is divided into different functions based on technology.

8. Support Team: It specifies the actual team that solves the problem.

9. Country Code: It specifies the location that takes ownership of the support team.

10. Action Owner: It specifies the person within the support team that is working with the incident. An action owner can transfer a problem to another action owner within the same support team or can escalate or return it to another support team.

caseid	dateplustime	actor	area	country	countrycode	impact	product	status	stlevel	substatus	suppteam
1-736720411	2104-03-22 15:59:42India Standard Time	Michael	Org line A2	Sweden	se	Medium	PROD235	Accepted	A2_1	In Progress	D7
1-736720411	2104-03-23 16:00:56India Standard Time	Michael	Org line A2	Sweden	se	Medium	PROD235	Accepted	A2_1	In Progress	D7
1-736720411	2104-03-24 15:59:42India Standard Time	Michael	Org line A2	Sweden	se	Medium	PROD235	Queued	A2_1	Awaiting Assignment	D4
1-736720411	2104-03-25 16:00:56India Standard Time	Paulo	Org line A2	Brazil	se	Medium	PROD235	Accepted	A2_1	In Progress	D4
1-736720411	2104-03-26 15:59:42India Standard Time	Paulo	Org line A2	Brazil	se	Medium	PROD235	Queued	A2_5	Awaiting Assignment	A10
1-736720411	2104-03-27 16:00:56India Standard Time	Mikael	Org line A2	Sweden	se	Medium	PROD235	Accepted	A2_5	In Progress	A10
1-736720411	2104-03-28 15:59:42India Standard Time	Mikael	Org line A2	Sweden	se	Medium	PROD235	Completed	A2_5	Resolved	A10
1-736720411	2104-03-29 16:00:56India Standard Time	Siebel	Org line A2	0	se	Medium	PROD235	Completed	A2_5	Closed	A10
1-738247053	2120-01-20 15:59:42India Standard Time	Reza	Org line C	Sweden	se	Medium	PROD660	Accepted	V3_2	In Progress	S42
1-738247053	2120-01-21 16:00:56India Standard Time	Reza	Org line C	Sweden	se	Medium	PROD660	Accepted	V3_2	In Progress	S42
1-738247053	2120-01-22 15:59:42India Standard Time	Reza	Org line C	Sweden	se	Medium	PROD660	Queued	E_10	Awaiting Assignment	G140 2nd
1-738247053	2120-01-23 16:00:56India Standard Time	Klas	Org line C	Sweden	se	Medium	PROD660	Accepted	E_10	In Progress	G140 2nd
1-738247053	2120-01-24 15:59:42India Standard Time	Klas	Org line C	Sweden	se	Medium	PROD660	Accepted	E_10	Assigned	G140 2nd
1-738247053	2120-01-25 16:00:56India Standard Time	Nicky	Org line C	Sweden	se	Medium	PROD660	Accepted	E_10	In Progress	G140 2nd
1-738247053	2120-01-26 15:59:42India Standard Time	Nicky	Org line C	Sweden	se	Medium	PROD660	Completed	E_10	Resolved	G140 2nd
1-738247053	2120-01-27 16:00:56India Standard Time	Siebel	Org line C	0	se	Medium	PROD660	Completed	E_10	Closed	G140 2nd
1-738241883	2119-07-24 15:59:42India Standard Time	Przemyslaw	Org line C	POLAND	pl	Low	PROD754	Accepted	V3_2	In Progress	G96
1-738241883	2119-07-25 16:00:56India Standard Time	Przemyslaw	Org line C	POLAND	pl	Low	PROD754	Accepted	V3_2	In Progress	G96
1-738241883	2119-07-26 15:59:42India Standard Time	Przemyslaw	Org line A2	POLAND	pl	Low	PROD754	Queued	A2_2	Awaiting Assignment	G21 2nd
1-738241883	2119-07-27 16:00:56India Standard Time	Padmanabha	Org line A2	Sweden	pl	Low	PROD754	Accepted	A2_2	In Progress	G21 2nd

Figure 5.1: VINST Eventlog Dataset

Fig. 5.1 shows the dataset that is used in our experimentation. Since α algorithm is concerned only with the sequence of events, and we need find all transitions that a problem can go through and compute causality between the activities, we consider only three fields in our dataset that have been highlighted in red- caseid which is unique for a particular problem, dateplustime which is the timestamp value giving the sequence order of activities for a paricular caseid and substatus which represents the activities in our data model. Thus, we load only these three fields in our eventlog table, and corresponding to each caseid we obtain the sequence of events as the events are ordered according to the timestamp values.

Fig. 5.2 shows the graph between the total number of cases and the case duration. As can be seen from the figure, the total number of records in the event log are 65533,

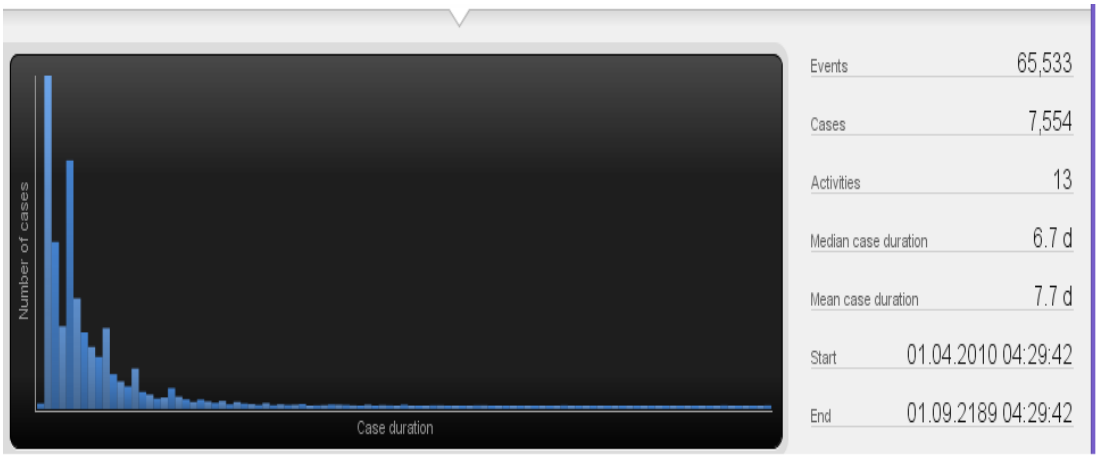


Figure 5.2: Case Duration

the total number of cases i.e. process instances are 7554 and the total number of activities in the event log are 13.

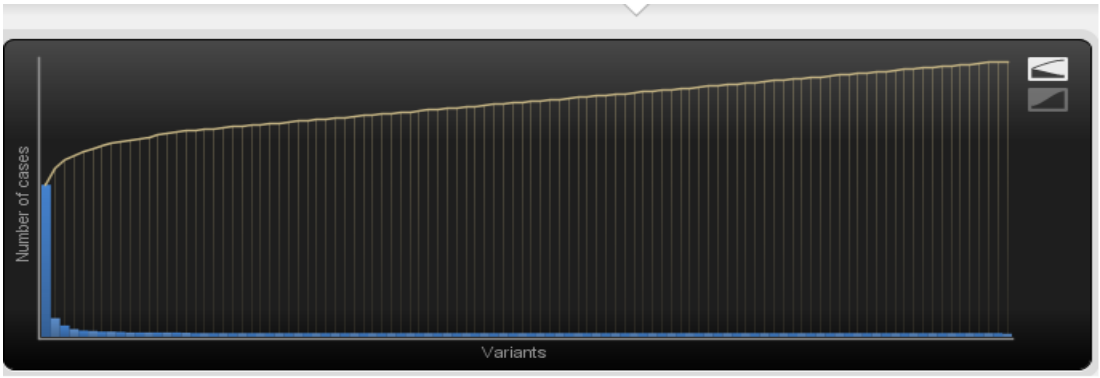


Figure 5.3: Number of Cases vs Case Variants

Fig 5.3 shows how the total number of cases vary with the variants. Variants here means the total number of unique traces in the event log. The total number of unique traces in this event log are 2278.
Both Fig. 5.2 and 5.3 have been obtained using Disco tool which is an open source process mining toolkit. We give the CSV file(eventlog file) as input to the Disco tool. We then simply configure the columns that hold our caseid, timestamp and activities and the tool gives the above statistic for the process.

6

Implementation Of α Miner Algorithm on Row Oriented Data Store MySQL

In order to implement α miner algorithm in MySQL, we do some preprocessing steps in Java. These steps are not a part of α miner algorithm. We create the following two tables during preprocessing:

1. Causality Table: Stores activities that have Causality relationship with each other.
2. Compare Table: Stores the set of all those activities that are not connected to each other.

1. Create Table for Storing the Event Logs.

```
CREATE TABLE eventlog(caseid varchar,datetime timestamp,activity
    varchar);
```

2. Load Data from the CSV File into Table.

```
Load Data Local Infile 'eventlog.csv' INTO table eventlog fileds
    terminated by ; ignore 1 lines (caseid,datetime,activity);
```

3. Create table for storing all the distinct activities in the eventlog.

```
CREATE TABLE totalEvents(events varchar PRIMARY KEY);
INSERT INTO totalEvents(events) (SELECT DISTINCT eventlog.activity FROM
    eventlog);
```

This completes the first step of α-miner algorithm.

4. Create table for storing all the initial activities, i.e., all activities that appear first in some trace.

```
CREATE TABLE initialEvents(initialEvents varchar);

INSERT INTO initialEvents(initialEvents)
    SELECT distinct eventlog.activity
        FROM
    SELECT eventlog.activity,
    MIN(eventlog.datetime),
                eventlog.activity
            FROM eventlog GROUP BY eventlog.caseid)AS derived);
```

The above query has two parts of statements: Insert statement inserts all rows returned by first outer subquery Select statement into table named initial. The outer subquery Select statement selects distinct events from temporary table returned by inner subquery considering all caseid and the inner subquery Select statement works on eventlog table by considering all caseid with their minimum timestamp using Group By.

This completes the second step of α-miner algorithm.

5. Create table for storing all the final activities, i.e., all activities that appear last in some trace.

```
CREATE TABLE finalEvents(final varchar);
INSERT INTO finalEvents(finalEvents) SELECT DISTINCT A.activity FROM
    eventlog AS A WHERE A.datetime IN
 (SELECT MAX(derived.datetime) AS datetime FROM eventlog As derived
     GROUP BY derived.caseid);
```

The above query has two parts of statement: The Insert statement inserts all rows returned by first outer subquery Select statement into the table named final. The outer subquery Select statement selects distinct events from temporary table

returned by inner subquery by comparing their timestamp with inner subquery maximum timestamp using IN operator and the inner subquery Select statement works on eventlog table by considering all caseid with their maximum timestamp using Group By.

This completes the third step of α-miner algorithm.

6. Create table eventA.

```
CREATE TABLE eventA(setA varchar,setB varchar);
```

This table contains two columns setA and setB. setA consists of single events and setB consists of single or set of events such that the relationship between every element in setA to every element in setB is causality.

7. Create table eventB.

```
CREATE TABLE eventB(setA varchar,setB varchar);
```

This table contains two columns setA and setB. setA consists of single or set of events and setB consists of single events such that the relationship between every element in setA to every element in setB is causality.

8. Create table safeEventA

```
CREATE TABLE safeEventA(setA varchar,setB varchar);
```

This table contains two columns setA and setB. setA consists of single events and setB consists of single or set of events such that the relationship between every element in setA to every element in setB is causality and relationship between every element in setB to every other element is not-connected($\sharp$).

9. Create table safeEventB

```
CREATE TABLE safeEventB(setA varchar,setB varchar);
```

This table contains two columns setA and setB. setA consists of single events or set of events and setB consists of single events such that the relationship between every element in setA to every element in setB is causality and relationship between every element in setA to every other element is not-connected($\sharp$).

10. Create table X_W

```
CREATE TABLE xw (setA varchar,setB varchar);
```

This table stores all rows from eventA, eventB, safeEventA, safeEventB.

11. Populate the table eventA.

```
INSERT INTO eventA(setA,setB)
    SELECT eventA, GROUP_CONCAT(eventB)
        FROM causality GROUP BY eventA;
```

The above query has two parts of statement: The Insert statement inserts all rows returned by the inner subquery Select statement. The Select statement selects column eventA from table causality and group_concat(eventB). Therefore, we have a set of combination of activities such that each activity in each set has causality relationship with the corresponding activity in eventA.

12. Populate Table eventB.

```
INSERT INTO eventB(setA,setB)
    SELECT GROUP_CONCAT(eventA),eventB
        FROM causality GROUP BY eventB;
```

The Insert statement inserts all rows returned by the inner subquery Select statement. The Select statement selects column eventB from table causality and group_concat(eventA). Therefore, we have a set of combination of activities such that each activity in each set has causality relationship with the corresponding activity in eventB.

13. Populate Table safeEventA.

```
INSERT INTO safeEventA(setA,setB)
    SELECT setA,setB FROM eventA
        JOIN compare WHERE eventA.setB=compare.event
```

The above query has two statements: The Insert statement inserts all rows returned by inner subquery statement into safeEventA table. Join operation is performed on tables eventA and compare. This gives us all those sets of activities in setB that are not connected to each other.

14. Populate Table safeEventB

```
INSERT INTO safeEventB(setA,setB)
    SELECT eventB.setA,eventB.setB FROM eventB
        JOIN compare WHERE eventB.setA=compare.event;
```

The above query has two statements: The Insert statement inserts all rows returned by inner subquery statement into safeEventB table. Join operation is performed on tables eventB and compare. This gives us all those sets of activities in setA that are not connected to each other.

15. Populate Table xw.

```
INSERT INTO xw(setA,setB)
    SELECT eventA,eventB
        FROMcausality;

INSERT INTO xw (setA,setB)
    SELECT setA,setB
        FROM safeEventA;

INSERT INTO xw (setA,setB)
    SELECT setA,setB
        FROM safeEventB;
```

16. CALL stored procedure explode_tableYW1.

```
DROP PROCEDURE IF EXISTS explode_tableYW1 $$
CREATE PROCEDURE explode_tableYW1(bound VARCHAR(255))
  BEGIN
    DECLARE A text ;
    DECLARE B TEXT;
    DECLARE occurance INT DEFAULT 0;
    DECLARE i INT DEFAULT 0;
    DECLARE splitted_value text;
    DECLARE done INT DEFAULT 0;
    DECLARE cur1 CURSOR FOR SELECT safeEventB.setA, safeEventB.setB
                                        FROM safeEventB;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
    DROP TEMPORARY TABLE IF EXISTS bpi_2013.safeb;
    CREATE TEMPORARY TABLE bpi_2013.safeb(
    `setA` VARCHAR(255) NOT NULL,
```

```
39     `setB` VARCHAR(255) NOT NULL
40     );
41     OPEN cur1;
42 read_loop: LOOP
43         FETCH cur1 INTO A, B;
44         IF done THEN
45           LEAVE read_loop;
46         END IF;
47         SET occurance = (SELECT LENGTH(A)- LENGTH(REPLACE(A, bound,
               ''))+1);
48         SET i=1;
49         WHILE i<= occurance DO
50           SET splitted_value =
51           (SELECT REPLACE(SUBSTRING(SUBSTRING_INDEX(A, bound, i),
52 LENGTH(SUBSTRING_INDEX(A, bound, i - 1)) + 1), ',', ''));
53           INSERT INTO bpi_2013.safeb VALUES (splitted_value,B);
54           SET i = i + 1;
55         END WHILE;
56       END LOOP;
57       SELECT distinct setA,setB FROM bpi_2013.safeb;
58     CLOSE cur1;
59   END; $$
```

The above stored procedure splits the column setA of table safeEventB on comma.

17. CALL stored procedure explode_tableYW.

```
60 DELIMITER $$
61 DROP PROCEDURE IF EXISTS explode_tableYW $$
62 CREATE PROCEDURE explode_tableYW(bound VARCHAR(255))
63   BEGIN
64     DECLARE A text ;
65     DECLARE B TEXT;
66     DECLARE occurance INT DEFAULT 0;
67     DECLARE i INT DEFAULT 0;
68     DECLARE splitted_value text;
69     DECLARE done INT DEFAULT 0;
70     DECLARE cur1 CURSOR FOR SELECT safeEventA.setA, safeEventA.setB FROM
           safeEventA ;
71     DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
72     DROP TEMPORARY TABLE IF EXISTS bpi_2013.safea;
```

```
73     CREATE TEMPORARY TABLE bpi_2013.safea( `setA` VARCHAR(255) NOT
           NULL,`setB` VARCHAR(255) NOT NULL);
74  OPEN cur1;
75 read_loop: LOOP
76         FETCH cur1 INTO A, B;
77         IF done THEN
78           LEAVE read_loop;
79         END IF;
80         SET occurance = (SELECT LENGTH(B)- LENGTH(REPLACE(B, bound,
               ''))+1);
81         SET i=1;
82         WHILE i<= occurance DO
83           SET splitted_value =
84           (SELECT REPLACE(SUBSTRING(SUBSTRING_INDEX(B, bound,
                 i),LENGTH(SUBSTRING_INDEX(B, bound, i - 1)) + 1), ',', ''));
85           INSERT INTO bpi_2013.safea VALUES (A,splitted_value);
86           SET i = i + 1;
87         END WHILE;
88       END LOOP;
89       SELECT distinct setA,setB FROM bpi_2013.safea;
90     CLOSE cur1;
91   END; $$
```

The above stored procedure splits the column setB of table safeEventA on comma.

18. Create and populate table eventASafe

```
CREATE TABLE eventASafe(setA varchar(255),setB varchar (255));
INSERT INTO eventASafe (setA,setB)
    SELECT setA,setB from safeA;
```

The above query has two statements: The Insert statement inserts all rows of safea table in eventASafe table. The Select statement selects all rows of safea table that was created by the above stored procedure.

19. Create and populate table eventBSafe.

```
CREATE TABLE eventBSafe(setA varchar,setB varchar);
INSERT INTO eventBSafe (setA,setB)
    SELECT setA,setB from safeB;
```

The above query has two statements: The Insert statement inserts all rows of safeB table in eventBSafe table. The Select statement selects all rows of safeB table that was created by the above stored procedure.

20. Create and populate table Y_W.

```
CREATE TABLE yw(setA varchar,setB varchar,PRIAMRY KEY(setA,setB));
CREATE TABLE temporary_table(setA varchar,setB varchar,PRIMARY
    KEY(setA,setB));
INSERT INTO temporary_table(setA,setB)
    SELECT setA,setB FROM eventBsafe;
INSERT INTO temporary_table(setA,setB)
    SELECT setA,setB FROM eventAsafe;
INSERT INTO yw (setA,setB)
    SELECT eventA ,eventB FROM bpi_2013.casulaity
        WHERE eventB NOT IN
            (SELECT DISTINCT setB FROM temporary_table)
                   OR
            eventA NOTIN(SELECT DISTINCT setA FROM temporary_table);

INSERT INTO yw (setA,setB)
    SELECT setA ,setB
        FROM bpi_2013.safeEventA;
INSERT INTO yw (setA,setB)
    SELECT setA,setB
    FROM bpi_2013.safeEventB;
```

This completes the fifth step of alpha algorithm.

21. Create and populate table that stores the input and output places.

```
CREATE TABLE terminalPlaces(event varchar);
INSERT INTO terminalPlaces Values ('i');
INSERT INTO terminalPlaces Values ('o');
```

22. Create and populate table P_W

```
CREATE TABLE pw(setA varchar);
INSERT INTO pw(setA)
    SELECT CONCAT_WS(' & ',setA,SetB) AS place From yw;
INSERT INTO pw(setA)
    SELECT event FROM terminalPlaces;
```

23. Create and populate table Place1 and Place2

```
CREATE TABLE place1 (id varchar(200),value varchar(200));
INSERT INTO place1
    SELECT yw.setA, CONCAT_WS(' & ',yw.setA,yw.setB)
     FROM yw;
CREATE TABLE place2(id varchar(200),value varchar(200));
INSERT INTO place2
     SELECT CONCAT_WS(' & ',yw.setA,yw.setB),yw.setB
      FROM yw;
```

24. CALL stored procedure explode_table_for_place2.

```
DELIMITER $$
DROP PROCEDURE IF EXISTS explode_table_for_place2 $$
CREATE PROCEDURE explode_table_for_place2(bound VARCHAR(255))
  BEGIN
    DECLARE A text ;
    DECLARE B TEXT;
    DECLARE occurance INT DEFAULT 0;
    DECLARE i INT DEFAULT 0;
    DECLARE splitted_value text;
    DECLARE done INT DEFAULT 0;
    DECLARE cur1 CURSOR FOR SELECT place2.id, place2.value FROM place2 ;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
    DROP TABLE IF EXISTS bpi_2013.temp_place2;
    CREATE TABLE bpi_2013.temp_place2(
    `id` VARCHAR(255) NOT NULL,
  `value` VARCHAR(255) NOT NULL );
    OPEN cur1;
read_loop: LOOP
        FETCH cur1 INTO A, B;
        IF done THEN
          LEAVE read_loop;
        END IF;
        SET occurance = (SELECT LENGTH(B)- LENGTH(REPLACE(B, bound, ''))
            +1);
        SET i=1;
        WHILE i<= occurance DO
```

```
158          SET splitted_value =
159          (SELECT REPLACE(SUBSTRING(SUBSTRING_INDEX(B, bound, i),
160 LENGTH(SUBSTRING_INDEX(B, bound, i - 1)) + 1), ',', ''));
161          INSERT INTO bpi_2013.temp_place2 VALUES (A,splitted_value);
162          SET i = i + 1;
163        END WHILE;
164      END LOOP;
165      SELECT distinct id,value FROM bpi_2013.temp_place2;
166    CLOSE cur1;
167  END; $$
```

25. CALL stored procedure explode_table_for_place1.

```
169 DELIMITER $$
170 DROP PROCEDURE IF EXISTS explode_table_for_place1 $$
171 CREATE PROCEDURE explode_table_for_place1(bound VARCHAR(255))
172  BEGIN
173    DECLARE A text ;
174    DECLARE B TEXT;
175    DECLARE occurance INT DEFAULT 0;
176    DECLARE i INT DEFAULT 0;
177    DECLARE splitted_value text;
178    DECLARE done INT DEFAULT 0;
179    DECLARE cur1 CURSOR FOR SELECT place1.id, place1.value FROM place1 ;
180    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
181    DROP  TABLE IF EXISTS bpi_2013.temp_place1;
182    CREATE TABLE bpi_2013.temp_place1(
183    `id` VARCHAR(255) NOT NULL,
184    `value` VARCHAR(255) NOT NULL );
185    OPEN cur1;
186 read_loop: LOOP
187        FETCH cur1 INTO A, B;
188        IF done THEN
189          LEAVE read_loop;
190        END IF;
191        SET occurance = (SELECT LENGTH(A)- LENGTH(REPLACE(A, bound,
               ''))+1);
192        SET i=1;
193        WHILE i<= occurance DO
194          SET splitted_value =
195          (SELECT REPLACE(SUBSTRING(SUBSTRING_INDEX(A, bound, i),
```

```
LENGTH(SUBSTRING_INDEX(A, bound, i - 1)) + 1), ',', ''));
          INSERT INTO bpi_2013.temp_place21 VALUES (splitted_value,B);
          SET i = i + 1;
        END WHILE;
      END LOOP;
      SELECT distinct id,value FROM bpi_2013.temp_place1;
    CLOSE cur1;
  END; $$
```

26. Create and populate table F_W

```
CREATE TABLE fw(setA varchar(200),setB varchar(200));
INSERT INTO fw
    SELECT * FROM temp_place2;
INSERT INTO fw
    SELECT * FROM temp_place1;
INSERT INTO fw(setA,setB)
    SELECT S.event , I.initialEvents
    FROM terminalPlaces AS S, initialEvents AS I
    WHERE S.event='i';
INSERT INTO fw(setA,setB)
    SELECT F.finalEvents , S.event
    FROM terminalPlaces AS S, FinalEvents AS F
    WHERE S.event='o';
```

7

Implementation Of α Miner Algorithm on Column Oriented Data Store Cassandra

In Cassandra we don't start with the data model. We start with the query model. We first need to determine our queries and then model the data around the queries. In relational databases, even if the data is not modelled correctly, we can still get the desired results using complex joins. However, in column oriented databases like cassandra if the data model is not correct then we cannot obtained the desired results as cassandra does not support joins. Thus, in cassandra we need to do more work when we insert data.

1. Create keyspace.

```
CREATE KEYSPACE eventdataset WITH replication = {
  'class': 'SimpleStrategy',
  'replication_factor': '1'
};
```

   In Cassandra, keyspace is the container for the data we need to store and process. It is similar to a schema in a relational database.

2. Create and populate table eventlog.

```
CREATE TABLE eventlog
(
caseid text ,
```

```
datetime timestamp,
activity text,
PRIMARY KEY(caseid,datetime)
);
COPY eventlog(caseid,datetime,activity) FROM 'eventlog.csv'
WITH HEADER=TRUE and DELIMITER=';';
```

In this table the Primary Key consists of caseid and datetime. Here, caseid is the partition key and datetime is the clustering key.

3. Create and populate table totalEvents.

```
CREATE TABLE totalEvents(tevents text PRIMARY KEY);
COPY eventlog(activity) TO 'totalEvents.csv';
COPY totalEvents(tevents) FROM 'totalEvents.csv' WITH HEADER=TRUE;
```

The query can be explained as: The Create command creates the table totalEvents with tevents(that represents activities in the eventlog) as the primary key for storing the total distinct activities in the eventlog. The first Copy command loads all the activities of the eventlog into 'totalEvents.csv' file. The second Copy command copies the activities stored in 'totalEvents.csv' file to the table totalEvents. Since, totalEvents table has activities as the primary key, all the distinct activities are loaded into the table and we get the total number of distinct activities in the eventlog.

4. Create and populate table for storing all the initial activities.

```
CREATE TABLE initial
(
caseid text ,
dateplustime timestamp,
activity text,
PRIMARY KEY(caseid,datetime)
)WITH CLUSTERING ORDER BY (dateplustime DESC);
CREATE TABLE starteve(caseid text PRIMARY KEY,dateplustime
    timestamp,sevents text);
CREATE TABLE startevents (ievents text PRIMARY KEY);
COPY initial(caseid,dateplustime,activity)
FROM 'eventlog.csv' WITH HEADER=TRUE and DELIMITER=';';
COPY initial(caseid,datetime,activity) TO 'initial.csv';
```

```
COPY starteve(caseid,dateplustime,sevents) FROM 'initial.csv';
COPY starteve(sevents) TO 'startEve.csv';
COPY startevents(ievents) FROM 'startEve.csv' WITH HEADER=TRUE;
```

The table startevents is used to store all the initial activities i.e., all those activities that appear in the start of some trace. The above steps can be explained as: The table initial is created with the rows being ordered in the decreasing order of timestamp and data from 'eventlog.csv' is loaded into it. The columns caseid, datetime and activity of table initial are copied to the file 'inital.csv'. Therefore, the file 'initial.csv' will contain the rows in decreasing order of timestamp. The data from file 'initial.csv' is loaded into table starteve. Thus, the row with the earliest value of timestamp corresponding to a caseid will sustain in the table. Finally, the table startevents is created, with ievents(that represent activities in the eventlog) as the primary key. The column sevents, which represents activities in the eventlog is copied into the file 'startEve.csv' and this file is then loaded to the table startevents giving all the distinct initial activities in the eventlog.

5. Create and populate table finalEvents.

```
CREATE TABLE final
(
caseid text PRIMARY KEY ,
dateptime timestamp,
activity text

);
CREATE TABLE finaleve(fevents text PRIMARY KEY);
COPY final(caseid,dateplustime,activity)FROM 'eventlog.csv' WITH
    HEADER=TRUE and DELIMITER=';';
COPY final(activity) TO 'finalEve.csv';
COPY finalEvents(fevents) FROM 'finalEve.csv' WITH HEADER=TRUE;
```

The table finalEvents is used to store all the ending activities i.e., all those activities that appear in the last of some trace. The above steps can be explained as: The table final is created and data from 'eventlog.csv' is loaded into it. Since, only caseid is the primary key in the table final, it will contain only rows with the highest value of timestamp corresponding to a caseid. The column activity of table final is copied to file 'finalEve.csv'. Finally, the table finalEvents is created, with fevents(that represent activities in the eventlog) as the primary key. The

data from file 'finalEve.csv' is loaded into the table finalEvents. Thus, we get all the final distinct activities in the table finalEvents.

6. Perform preprocessing to create the footprint matrix.

```
CREATE TABLE trace(
caseid text PRIMARY KEY,
events list<text>);
```

Initialize an arraylist traceList.

foreach caseid in the log **do**

| Get the sequence of events in arraylist traceList corresponding to that caseid and and insert it into the table trace .

end

```
INSERT INTO TABLE trace(
caseid text PRIMARY KEY,
tracelist);
```

```
CREATE TABLE preprocesstable(ACTIVITY TEXT PRIMARY KEY,
activity1.....activityN);
```

Initialize all the values in the preprocesstable which represents the footprint matrix as not connected.

foreach distinct trace in the eventlog **do**

| Scan the trace and update the corresponding entry in the preprocesstable.

end

```
UPDATE preprocesstable SET activityA='RELATION' WHERE
    activity='activityB';
```

7. Create and populate Table SetsReachabeTo and SetsReachableFrom

```
CREATE TABLE setsreachablefrom(Place text PRIMARY KEY,activityin
    text,activityout set<text>);
CREATE TABLE setsreachableto(Place text PRIMARY KEY,activityout
    text,activityin set<text>);
```

foreach activity in totalActivities Table **do**
  Read from the preprocesstable its relation to every other activity. **if** *Relation is CAUSALITY* **then**
    Insert it into Table setsreachablefrom
  end
end

```
INSERT INTO setsreachablefrom
(Place,activityin,activityout)
values(placeN,
activityA,{'activity1',....'activityN');
```

Similarly, we create table setsreachableto.

Finally we merge the two tables.

```
COPY setsreachablefrom(...) to 'xwFrom.csv'
COPY setsreachableto(activityin,activitout) to 'xwTo.csv';
CREATE TABLE XW(Place int PRIMARY KEY,
inactivities set<text>,
outactivities set<text>);
COPY XW from 'xwFrom.csv';
COPY XW from 'xwTo.csv';
```

8. Create and Populate Table YW

```
create table YW(Place text PRIMARY KEY, activityin set<text>,
    activityout set<text>);
```

foreach *inputActivity* in table XW **do**
  foreach outputActivity in table XW **do**
    If inactivity and outactivity of place X is a subset of inactivity and outactivity of place Y, add placeY into table YW.
  end
end

```
select inactivities from xw where place='placeX';
select outactivities from xw where place='placeY';
```

```
INSERT INTO YW
(Place,activityin,activityout)
```

```
values
(PlaceY,inactivities,outactivities);
```

9. Create Places.

```
ALTER TABLE YW ADD placeName text;
```

foreach inputA and output in table YW **do**
| Concatenate the activity values.
| Insert into column placeName for that place.
end

Then we need to add the input place i.e the source place and the output place i.e the sink place.

```
INSERT INTO TABLE YW(place,placeName)
values('inputPlace',iL);
INSERT INTO TABLE YW(place,placeName)
values('outputPlace',oL);
```

10. Insert the set of initial and final events.

Create a hashset initialActivity.
Create a hashset finalActivity.
foreach activity in initial Table **do**
| Read from the table and add it to hashset initialActivity.
end
foreach activity in final Table **do**
| Read from the table and add it to hashset finalActivity.
end

```
INSERT INTO TABLE YW(place,activityout)
values('inputPlace',initialActivity);
INSERT INTO TABLE YW(place,activityin)
values('outputPlace',finalActivity);
```

8

Performance Comparison

Our benchmarking system is a 64 bit Windows Operating System with 4.00 GB of RAM and 245 GB of disk space. We have taken each reading five times and recorded the average of these readings in the paper.

Table 8.1: Load Time

Dataset Size	**Load Time in Sec**	
	MR	**CC**
65,000	1.934	19.22
1,00,000	2.8865	31.977
1,50,000	4.644	35.986
3,00,000	5.318	40.081
4,66,738	10.906	56.49

For our experiments we use single node cassandra cluster and single node MySQL database. Table 8.1 shows the time taken to load datasets of different sizes on Cassandra and MySQL database and we observe that for a single node cluster, MySQL always outperforms Cassandra in terms of the time taken to load data in the database. Hence, we can conclude that MySQL is optimized to be really fast on a single node.

Table 8.2 shows the step wise execution time taken by both the databases on implementing α algorithm. As can be seen from the values, the time taken by Cassandra in first three steps is very high as compared to the time taken by MySQL. This is because in order to implement the first three steps, Cassandra requires to copy a lot of data into CSV files and copy back data from CSV files to the designed tables. In Cassandra

Table 8.2: Stepwise Time

Steps	**Execution Time in Sec**	
	MR	**CC**
1	1.01	24.66
2	1.09	33.38
3	1.18	4.02
4	3.759	3.89
5	4.559	6.158
6	1.679	4.98
7	6.957	1.830

we need to design the query model first and then design the data model around the query. Hence, in Cassandra we need to do more work while inserting data. There is however not much difference in the stepwise execution time of Cassandra and MySQL for the remaining steps, but MySQL proves to be faster as compared to Cassandra in most of the steps.

Table 8.3: Read Intensive Time

Steps	**Read Time in Seconds**	
	MR	**CC**
1	0.171	9.46
2	0.250	14.28
3	0.438	1.07
4	0.203	1.95
5	2.298	5.29
6	0.121	3.12
7	2.27	1.335

Table 8.4: Write Intensive Time

Steps	Write Time in Seconds	
	MR	**CC**
1	0.1571	13.66
2	0.172	17.146
3	0.077	1.812
4	0.900	0.117
5	0.118	0.1048
6	0.285	0.208
7	1.172	0.031

Table 8.3 shows the stepwise read time taken by both Cassandra and MySQL. In read time, we have included only the time taken in executing select commands i.e. time taken to read from the database in each step. We have not taken into account any create, insert or Java preprocessing required. As can be seen in Fig. 4, the time taken to read from the database in the first two steps is very high as compared to that in MySQL. The reason being that we have to copy a lot of data into CSV files as per the design of the data model. Though the third step also requires copying into CSV files, the time taken in this step is not much as compared to MySQL. This is because in this step the table used for loading the entire dataset uses only caseid as the primary key, hence the size of table will be greately used as it will contain data equal to the number of caseid(s). Hence, loading to and from CSV files in this step does not take much time. For the remaining steps, the time taken by Cassandra though not much, but is always greater as compared to MySQL. The reason being that MySQL supports subqueries but for Cassandra we need to read multiple times from the database in order to compute a particular step.

Table 8.4 shows the stepwise write time taken by both Cassandra and MySQL. In write time, we have included only the time taken in executing insert and update statements. i.e. time taken to write to the database in each step. We have not taken into account any create, select or Java preprocessing required. As can be seen in Fig. 5, the time taken to write to the database in Cassandra is high as compared to the time taken in MySQL for the same reasons as discussed above. However, for all other steps the time taken to write to Cassandra is less as compared to the time taken to write to MySQL. This is because Cassandra is designed for faster writes. For each step,

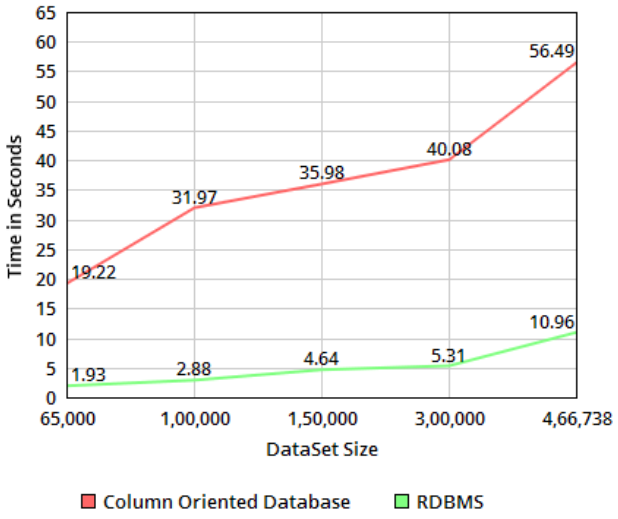
we need to read multiple times from the database, but the number of times we write to the database after all processing is much less as compared to the number of reads. Thus, we can conclude that Cassandra gives better write performance as compared to MySQL.

Table 8.5: Disk Usage

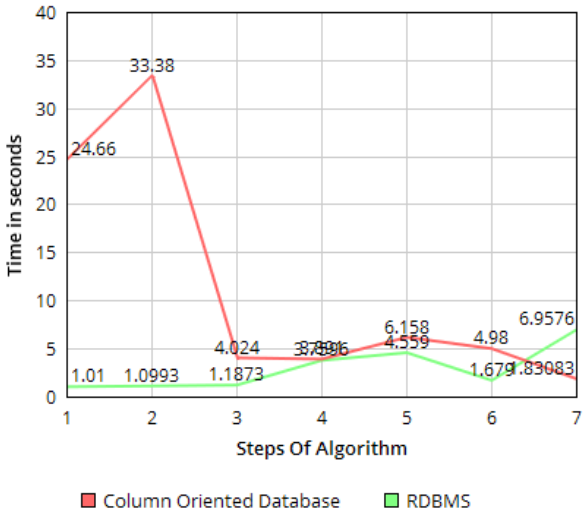
Table	**Disk Usage in Bytes**	
	MR	**CC**
total	20000	5263
initial	20000	5090
final	20000	5025
xw	20000	5932
yw	20000	6142
pw	20000	6142
fw	20000	6142

Table 8.6: Java Execution Time

Steps	**Time in Seconds**	
	MR	**CC**
1	1.01	3.49
2	1.09	4.95
3	1.18	4.23
4	3.759	3.89
5	4.559	6.158
6	1.679	4.98
7	6.957	1.830



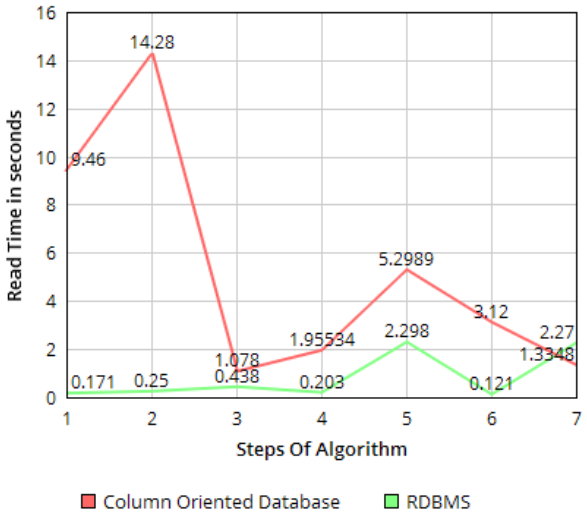
(a) Scalability Load Time



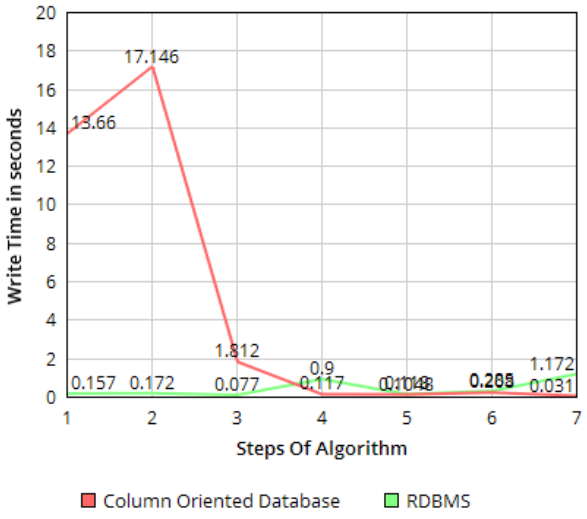
(b) Stepwise Execution Time

Figure 8.1: Load Time and Stepwise Execution Time

Table 8.5 shows the disk usage in bytes of the tables that are created at each step of the α algorithm by both the databases. We use the nodetool utility in Cassandra to compute the disk space occupied by the tables. The nodetool utility is a command line interface that is used to manage a Cassandra cluster. We use nodetool utility with



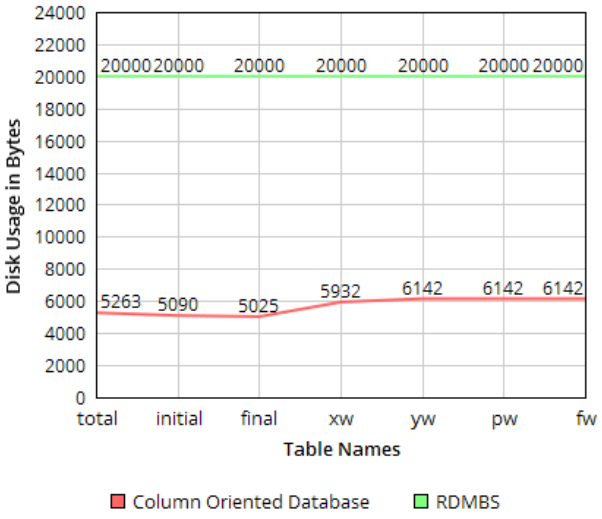
(a) Stepwise Read Time



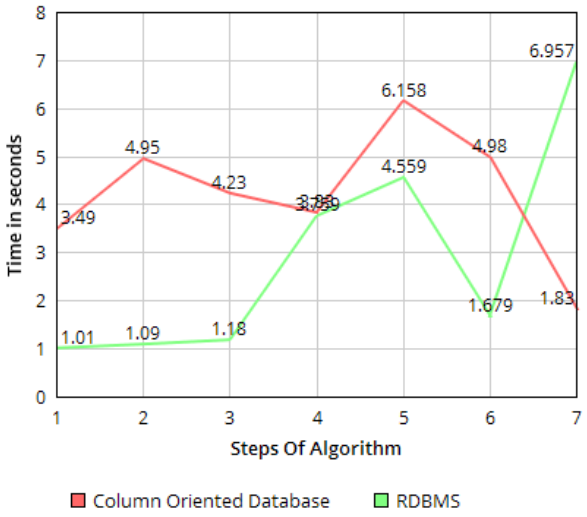
(b) Stepwise Write Time

Figure 8.2: Read and Write Time Execution

the option cfstats, which gives us the statistics of the table specified. In MySQL, we run a query over the entire schema, which gives us the disk space occupied by each table. As can be seen from the table, the disk space used by Cassandra is much lower as compared to the disk space used by MySQL. Thus, we can conclude that Cassandra



(a) Disk Usage



(b) Stepwise Time using Java

Figure 8.3: Disk Usage and Java Execution Time

stores data more efficiently as compared to MySQL.

Table 8.6 shows the step wise execution time taken by both the databases on implementing α algorithm. However, in this case we do not use any COPY commands for the first three steps for designing the data model. The COPY command is used just once

to load the main dataset into the eventlog table. We use only Java programming that contains CQL embedded read/write statements to read from the database, process the data and write back to the database. For the remaining steps, the procedure remains the same. Hence, the step wise time remains the same as in Table 8.2 for the rest of the steps. But as can be seen from Table 8.6, the execution time for the first two steps reduces significantly. Hence, we can conclude that two tier application in Cassandra gives a better performance as compared to one tier application. Also, the COPY command is slow in writing to the database as compared to the Insert command. One more drawback of the COPY command is that every row in the CSV input should contain the same number of columns defined in the table i.e, the number of columns in the CSV input is always the same as the number of columns in the Cassandra table metadata. Thus, a lot of overhead is required in terms of modelling data using COPY command as we need to create a number of tables according to the column values required and repeatedly copy to and from CSV files, to achieve the required data model.

Table 8.7: Stepwise Read Time Using Java

Steps	**Read Time in Seconds**	
	MR	**CC**
1	0.171	1.01
2	0.250	1.34
3	0.438	1.26
4	0.203	1.95
5	2.298	5.29
6	0.121	3.12
7	2.27	1.335

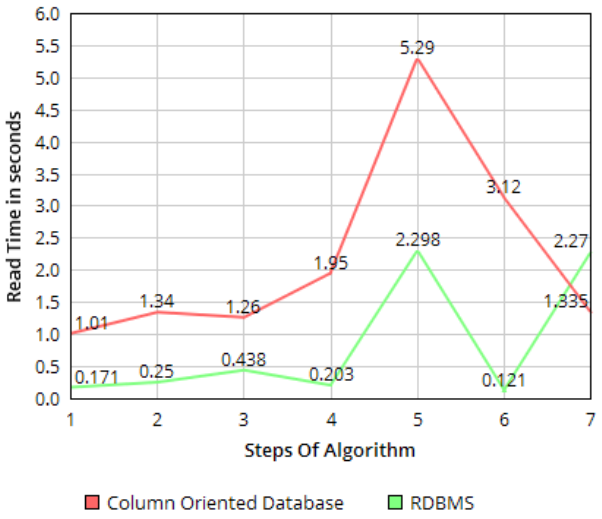
Table 8.8: Stepwise Write Time Using Java

Steps	**Write Time in Seconds**	
	MR	**CC**
1	0.1571	0.87
2	0.172	1.12
3	0.077	1.08
4	0.900	0.117
5	0.118	0.1048
6	0.285	0.208
7	1.172	0.031

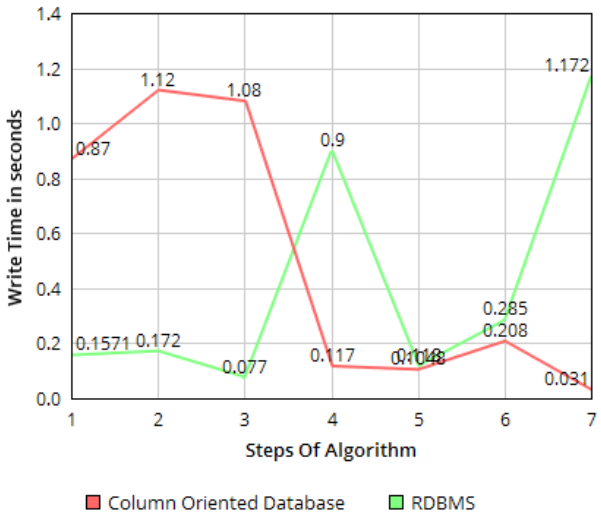
Table 8.7 and Table 8.8 shows the stepwise read time taken by both Cassandra and MySQL, where the entire α algorithm has been implemented using Java processing in Cassandra. That is, the first three steps have been performed using Java processing only and we do not use any COPY command. We make a table that stores the caseid and the corresponding sequence of activities, which represent a trace. We create three tables, storing the total, initial and final activities, with activity as the primary key. We iterate over the table that stores the traces, and store the first activity of each trace in the initial table, the last activity of each trace in the final table and each activity in the trace in total table. This, gives us the all the initial, final and total activities.

For calculating the read time, we have only included the read statements that is the Select statements that are peformed to obtain the relevant data. Similarly, for calculating the write time, we have only included the Insert statements that are performed while inserting the data into the respective tables. As can be seen from the graphs, the read and write time in Cassandra reduces by a great amount as compared to the values above(Table 8.3 and 8.4). Therefore, we can conclude that a complete two-tier application is better in performance. Since Java is a high level programming language and is also very flexible, it outperforms CQL language in terms of time taken.

Fig. 8.4(a) and Fig. 8.5(b) shows the comparison graph between read and write times in MySQL and Cassandra. We can see that Cassandra's performance has increased significantly as compared to the read and write times shown in Fig. 8.2(a) and Fig. 8.2(b).



(a) Stepwise Read Time using Java



(b) Stepwise Write Time using Java

Figure 8.4: Srepwise Read and Write Java Execution Time

9

Limitations and Future Work

In our work we have used the BPI challenge 2013 dataset which is a small dataset, that consists of only 65533 records. Future work includes applying the algorithm over multiple and larger datasets.

We are currently using a single node Cassandra cluster, but Cassandra works best in a distributed environment [4]. Future work includes creating a multi node cluster and implement the algorithm on it.

We have currently implemented only one algorithm that is the α miner algorithm. It is the most basic process mining algorithm and is generally not very useful for real time logs. Future work includes implementing more complex and advanced process mining algorithms like alpha plus miner algorithm, heuristic miner algorithm.

We have used performed the experiments on MySQL in row oriented and Cassandra in column oriented database only. Future work includes experimenting with more relational databases like PostgreSQL and more column oriented databases like HBase, Mongo DB so that we can come to a firm conclusion as which database is better for implementing α miner algorithm.

Future work also includes performing more optimizations and tuning like compression on both MySQL and Cassandra so that their performance can be improved.

10

Conclusion

This paper introduced the implementation of α miner algorithm in Structured Query Language and Cassandra Query Language and performance comparison of the algorithm on MySQL and Cassandra. The SQL implementation is a one tier application which uses only standard SQL queries. The work concentrated on defining suitable tables and writing efficient queries. The Cassandra implementation used Java program to read from the database, process the results and write back to the database. Experimental results reveal that MySQL performs better than Cassandra in terms of loading large datasets. Thus, we can conclude that MySQL is optimized to be really fast on a single node as compared to Cassandra. Also, the time taken to read from the database is more in Cassandra as compared to MySQL. However, Cassandra gives a better write performance as compared to MySQL in some of the steps if we are using only insert commands rather than COPY. The stepwise execution time improves significantly in Cassandra if we use only insert commands and Java processing rather than using COPY command. Since, Cassandra does not support any joins and nested queries, it becomes very important in Cassandra to model the data correctly. Sometimes we need to create similar tables with different primary key and load the same data, depending upon our requirement. This leads to a lot of overhead in Cassandra, but this is how Cassandra is designed. We need to do more work while inserting data. Cassandra, however outperforms MySQL in terms of the disk usage. The disk space occupied by tables in Cassandra is very less as compared to that in MySQL. Thus, we can conclude that Cassandra is more efficient than MySQL in terms of storing data but less efficient in terms of execution time, on a single node cluster.

References

[1] WIL VAN DER AALST. **Process Mining: Overview and Opportunities**. *ACM*, 2012. vi, 2, 11

[2] Nabil Hachem Daniel J.Abadi, Samuel R.Madden. **Column-Stores vs. Row-Stores: How Different Are They Really?** *SIGMOID*, 2008. 1, 2

[3] Gheorghe MATEI. **Column-Oriented Databases, an Alternative for Analytical Environment**. *Database Systems Journal*, 2010. 1

[4] Prashant Malik Avinash Lakshman. **Cassandra - A Decentralized Structured Storage System**. *ACM*. 4, 45

[5] Yahya Slimani Balla Wade Diack, Samba Ndiaye1. **CAP Theorem between Claims and Misunderstandings: What is to be Sacrificed?** *International Journal of Advanced Science and Technology*, 2013. 5

[6] Carlos Ordonez. **Programming the K-means clustering algorithm in SQL**. (6):823–828, 2004. 8

[7] C.Ordonez and P.Cereghini. **SQLEM: fast clustering in SQL using the EM algorithm**. *International Conference on Management of Data*, pages 559–570, 2000. 8

[8] Nicolas Marin Jose Maria Serrano Fernando Berzal, Juan-Carlos Cubero. **TBRAR: An efficient method for association rule mining in relational databases**. *elsevier*, 2001. 8

[9] K-U.Sattler and O.Dunemann. **SQL Database Primitives for Decision Tree Classifiers**. *Conference on Information and Knowledge Management*, pages 379–386, 2001. 8

[10] R. G. Mehta and and Dr. M. Raghuvanshi N.J. Mistry. **Impact of Column-oriented Databases on Data Mining Algorithms**. *International Journal of Advanced Research in Computer and Communication Engineering*, pages 2503–2507, 2013. 9

[11] D. P. Rana, N. J. Mistry, and M. M. Raghuwanshi. **Association Rule Mining Analyzation Using Column Oriented Database**. *International Journal of Advanced Computer Research*, **3**(3):88–93, 2013. 9

[12] J.B Simha L.Suresh and Rajappa Velur. **Implementing k-means Algorithm using Row store and Column store databases-A case study**. *International Journal of Recent Trends in Engineering*, **4**(2), 2009. 9

[13] C.Bazar and C.Sebastian. **The Transition from RDBMS to NoSQL. A Comparative Analysis of Three Popular Non-Relational Solutions:Cassandra, MongoDB and Couchbase**. *Database Systems Journal*, **5**(2):49–59, 2014. 9

[14] Ovidiu Aritoni Ciprian Pungila, Teodor Florin Fortis. **Benchmarking Database Systems for the Requirements of Sensor Readings**. pages 1–5, 2009. 9

[15] P. Cudre-Mauroux, I. Enchev, S. Fundatureanu, P. Groth, A. Haque, A. Harth, F.L. Keppmann, D. Miranker, J. Sequeda, and M. Wylot. **NoSQL Databases for RDF: An Empirical Evaluation**. **8219**, 2013. 9

[16] Wichian Premchaiswadi Sawitree Weerapong, Parham Porouhan. **Process Mining Using α-Algorithm as a Tool**. *IEEE*, 2012. 12

[17] V. Manoj. **Comparative Study Of NoSQL Document, Column Store Databases And Evaluation Of Cassandra**. **6**(4):11–26, 1996.

[18] L. Suresh and J.B Simha. **Novel and efficient clustering algorithm using structured query language**. *Computing, Communication and Networking, 2008. ICCCn 2008*, pages 1–5, 2008.