

Pragamana: Performance Comparison and Programming α -miner Algorithm in Relational Database Query Language and NoSQL Column-Oriented Using Apache Phoenix

Kunal Gupta

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

1 May 2015

-
- 1.: Prof. Ashish Sureka (Thesis Adviser)
 2. Prof. Sachit Butail (Internal Examiner)
 3. Dr. Satya Valluri (External Examiner)

Day of the defense: 1 May 2015

Signature from Post-Graduate Committee (PGC) Chair:

Abstract

Process-Aware Information Systems (PAIS) support business processes and generate large amounts of event logs from the execution of business processes. An event log is represented as a tuple of CaseID, Timestamp, Activity and Actor. Process mining is a new and emerging field that aims at analyzing the event logs to discover, enhance and improve business processes and check conformance between run time and design time. A large volume of event logs that are generated are stored in the databases such as relational, NoSQL and NewSQL. While relational databases perform well for a certain class of applications, there are a certain class of applications for which such databases create bottlenecks (like *Scalability* and *Sharding*). To handle such class of applications, NoSQL database systems have emerged. A relevant application of interest is the process mining task of discovering a process model (workflow model) from event logs. The α -miner algorithm is one of the first and most widely used Process Discovery technique. Our objective is to investigate which of the databases (Relational or NoSQL) perform better for a Process Discovery application under Process Mining. We implement the α -miner algorithm on relational (row-oriented) and NoSQL (column-oriented) databases in database query languages so that our algorithm is tightly coupled to the database. We do a performance benchmarking of the α -miner algorithm on a row-oriented database and a NoSQL column-oriented database to compare which database can efficiently store massive event logs and analyze it in seconds to discover a process model.

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

Acknowledgements

I would take this wonderful opportunity 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 Astha Sachdev for her 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 **Pragamana: Performance Comparison and Programming α -miner Algorithm in Relational Database Query Language and NoSQL Column-Oriented Using Apache Phoenix** submitted by **Kunal Gupta** 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	vi
List of Tables	viii
1 Research Motivation and Aim	1
1.1 Process Mining	3
1.2 Comparison of NoSQL Column-Oriented Database and Row-Oriented Database	6
1.3 SQL Interface Over NoSQL Column-Oriented Database	6
1.4 Research Aim	8
2 Related Work and Research Contributions	9
2.1 Related Work	9
2.1.1 Implementation of Mining Algorithms in Row-Oriented Databases	9
2.1.2 Implementation of Mining Algorithms in Column Oriented Databases	10
2.1.3 Performance Comparison of Mining Algorithms in Row-Oriented and Column-Oriented Databases	10
2.2 Thesis Contributions	11
3 Introduction of Apache Hadoop, Apache HBase and Apache Phoenix	12
3.1 Apache Hadoop	12
3.1.1 HDFS: Hadoop Distributed File System	12
3.2 Apache HBase-NoSQL Column-Oriented Database	13
3.2.1 Introduction	13
3.2.2 Architecture	14
3.3 Apache Phoenix-SQL Skin Over HBase	15

4	Description of α-Miner Algorithm with an Example	16
4.1	Description of α -Miner Algorithm	16
4.2	Example of α -miner Algorithm	17
5	Implementation of α-Miner Algorithm in SQL on Row-Oriented Database (MySQL) and Column-Oriented Database (HBase)	20
5.1	Implementation of α -Miner Algorithm in SQL on Row-Oriented Database (MySQL)	20
5.2	Implementation of α -Miner Algorithm on NoSQL Column-Oriented Database (HBase) Using Apache Phoenix	23
5.3	Output of α -Miner Algorithm from the Database	26
6	Experimental Dataset	27
7	Benchmarking and Performance Comparison	31
7.1	Loading Multiple Datasets	31
7.2	Execution of α -Miner Algorithm	33
7.3	Read Intensive Steps of α -Miner Algorithm	34
7.4	Write Intensive Steps of α -Miner Algorithm	35
7.5	Disk Usage of Tables	37
7.6	Disk Usage of Tables Using Compression Technique	38
7.7	Execution of α -Miner Algorithm Using Compression Technique	39
7.8	Real Time Insertion of an Event Logs	41
8	Conclusion	44
9	Limitations and Future Work	45
	Appendix	
A	Implementation of α-Miner Algorithm in SQL on Row-Oriented Database (MySQL)	46
B	Implementation of α-Miner Algorithm on NoSQL Column-Oriented Database (HBase) Using Apache Phoenix	54
	References	59

List of Figures

1.1	Types of Process Mining Techniques	4
1.2	Small Process Model	5
1.3	SQL Interface over NoSQL Column-Oriented Databases	7
3.1	HBase Architecture Adapted From [1]	14
3.2	HBase-Phoenix Architecture	15
4.1	Example of α -miner Algorithm	18
4.2	Continued Example of α -miner Algorithm	19
4.3	α -miner Algorithm-Input and Output Transitions	19
5.1	Small output of α -miner algorithm	26
6.1	Event Log	28
6.2	Activities in BPI 2014 Dataset	28
6.3	Number of Activities Per Case in BPI 2014 Dataset	29
6.4	Matrix of Events per Case	30
7.1	Dataset Load Time in Seconds	32
7.2	α -miner Stepwise Execution	33
7.3	Read Intensive Time in Seconds	35
7.4	Write Intensive Time in Seconds	36
7.5	Disk Usage of Tables	37
7.6	Disk Usage of Tables With Compression	39
7.7	α -miner Stepwise Execution Time with Compression	40
7.8	Batch wise Insertion Time in Seconds	41

LIST OF FIGURES

7.9	Number of Inserts per Second in Batch	42
7.10	Single Row Insertion Time in Seconds	43

List of Tables

5.1	Representation of Schema Table in MySQL	21
5.2	Comparison of Apache Phoenix and MySQL Query Language	23
5.3	Representation of Schema Table in HBase	24
7.1	Dataset Load Time	31
7.2	Stepwise Execution Time	33
7.3	Read Intensive Time	34
7.4	Write Intensive Time	35
7.5	Disk Usage of Tables	37
7.6	Disk Usage of Tables With Compression	38
7.7	Stepwise Execution Time with Compression	39
7.8	Batch wise Insertion Time	41
7.9	Number of Inserts per Second in Batch	41
7.10	Single Row Insertion Time	42

Research Motivation and Aim

A Process-Aware Information System (PAIS) is an Information Technology (IT) system that manages and supports business processes. A PAIS generates data from the execution of business processes. The data generated by a PAIS like Enterprise Resource Planning (ERP) and Customer Relationship Management (CRM) [2] is in the form of event logs (represented as a tuple $\langle \text{CaseID}, \text{Timestamp}, \text{Activity}, \text{Actor} \rangle$). In an event log, a particular CaseID, that is a process instance, has a set of activities associated with it, ordered by timestamp. Process Mining is a domain that analyzes business processes from event logs. Process Mining helps the organizations to improve their business processes by extracting useful insights from event logs. There are three major techniques of Process Mining *viz.* Process Discovery, Process Conformance and Process Enhancement [3]. The classification is based on whether there is an *a priori* model and, if present, how that model is used. In this thesis, we focus on Process Discovery aspect of Process Mining. In Process Discovery, there is no *a priori* model. Process Discovery aims to construct a process model, which is a computationally intensive task, from the the information present in event logs. One of the most fundamental algorithm under Process Discovery is the α -miner algorithm [4] which is used to generate process model from event logs.

The event logs used in discovering a process are very large and constantly growing in size. Before the year 2000, all the organizations used traditional relational database management system to store event logs. Process models were discovered by analysing event logs stored in the database with the help of programming (using a client application). Most of the traditional relational databases focus on Online Transaction

Processing (OLTP) applications [5] but are not able to perform Online Analytical Processing (OLAP) applications efficiently. Row-oriented databases are not well suited to execute analytical functions (like *Dense_Rank*, *Sum*, *Count*, *Rank*, *Top*, *First*, *Last* and *Average*) but work well for the retrieval of an entire row or insertion of a new record. On the other hand, NoSQL column-oriented databases are well suited for analytical queries but result in poor performance for insertion of individual records or retrieving all the fields of a row. Another problem with traditional relational databases is impedance matching [6]. Impedance matching occurs when the representation of data in memory is different from that in the databases is different. This is because in-memory data structures use lists, dictionaries, nested lists while relational databases store data only in the form of tables and rows. Thus, we need to translate data objects present in the memory to tables and rows and vice-versa. Performing the translation is complex and costly. NoSQL databases on the other hand are schema-less. Records can be inserted at run time without defining any rigid schema. Hence, NoSQL databases do not face the problem of impedance matching.

Recent years have seen the introduction of a number of NoSQL column-oriented database systems [7]. These database systems have been shown to perform more than an order of magnitude better than the traditional relational database systems on analytical workloads [8]. The underlying reason is that column-oriented are more I/O efficient for read only queries since they have to read queried attributes either from disk or from memory [8]. Our objective is to implement a process discovery algorithm α -miner algorithm on a row-oriented database and a NoSQL column-oriented database and to benchmark the performance of the algorithm on both the row-oriented and column-oriented databases.

A lot of research has been done in implementing data mining algorithms in database query languages. Previous work suggests that tight coupling of the data mining algorithms to the database systems improves the performance of the algorithms significantly [9]. We aim to implement α -miner algorithm in Structured Query Language (SQL) so that our Process Discovery application is tightly coupled to the database.

NoSQL column-oriented databases and Apache Hadoop¹ are still in research and can handle large data with help of new file system (HDFS). Combination of both Hadoop component and column-oriented databases allow accessing large data and storing data

¹<http://hadoop.apache.org/>

easily as compared to single machine databases [10]. There are various NoSQL column-oriented databases [7]. We aim to analyze event logs in real time and since HBase is used in real time messaging system, we focus on Apache HBase¹ (NoSQL column-oriented database) for our current work and benchmark its performance against MySQL² (row-oriented database) which is one of the most popular row-oriented database and is integrated with most of the applications for analysing, transforming and processing the data. To perform analytical functions, NoSQL column-oriented databases either use MapReduce programming model or use their own simple query language that just supports create, read, update and delete (CRUD). They do not support an SQL interface. We integrate Apache Phoenix³(SQL layer over HBase) into HBase to support SQL interface in it. It converts SQL queries to HBase scans rather than MapReduce jobs. It executes converted scans in parallel over the regions in a regionserver and targets low latency query over HBase tables as compared to MapReduce framework and client API's.

1.1 Process Mining

Process Mining creates process models by analyzing the business process event logs. These models are used for analysis in the growing business needs [3]. They help in providing comprehensive support for flexible business processes. There are various techniques such as α -miner algorithm and α^+ -miner algorithm which can be used to extract useful insights from the event logs [3]. These algorithms record events in a sequential way. An event is represented by a tuple $\langle \text{CaseID}, \text{Timestamp}, \text{Activity}, \text{Actor} \rangle$. This information is then used by process mining to construct process models. *For example*, the α -miner algorithm [4] can construct a Petri net model describing the behavior observed in the event logs.

Process mining focuses only on event logs which refers to discrete events that happen in real time. They also store additional information which is related to the activities. For example, Resources (i.e., the person or device), Timestamp of the event, or Data elements recorded with the event (e.g. size of an order).

¹www.hbase.apache.org

²<http://www.mysql.com/>

³<http://www.phoenix.apache.org>

Process mining will work even if there is a case of data explosion. Thus it's an opportunity that has emerged out of Big Data. The volume of data in organizations is increasing exponentially. Therefore, it is important to be able to process the massive event logs in order to make critical business decisions and satisfy user demands.

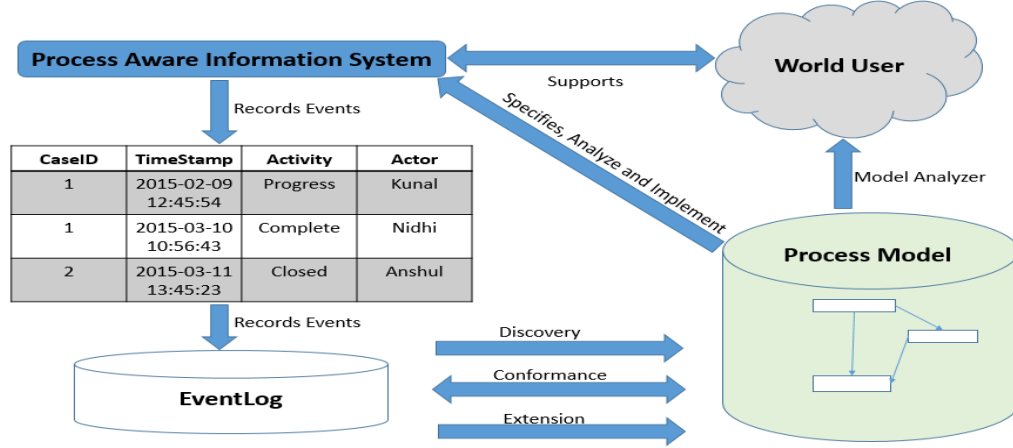


Figure 1.1: Types of Process Mining Techniques

There are three types of process mining techniques:

1. **Process Discovery:** It takes input as event log and produces a process model.
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 takes an existing process model as input extract new information from it as well as rephrase it.

We consider three different perspectives of process mining [4]:

1. **Process Perspective:** It focuses on the control flow of a process and order the activities from event logs.
2. **Organizational Perspective:** It focuses on how the originators (actor) of activities communicate with each other.
3. **Case Perspective:** It focuses on the property of a particular process instance. For example, property can be identified by the path instance takes in a process model.

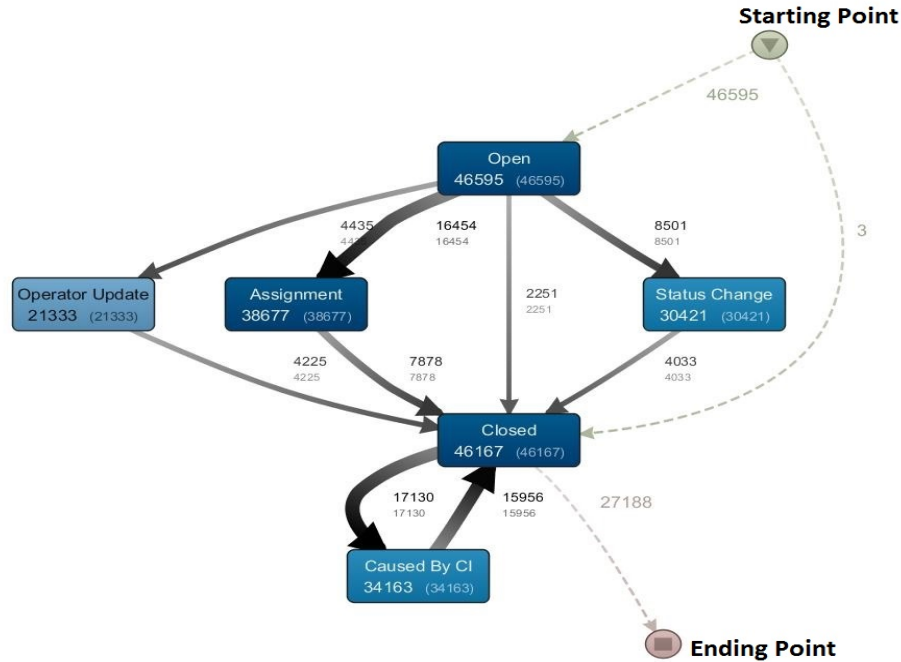


Figure 1.2: Small Process Model

Fig. 1.2, depicts the process model. Fig. 1.2, has a starting point and an ending point. All the activities lie between these two points. In the event log, set of activities correspond to a particular process instance (case). In all cases, there is an initial and a final activity. All initial activities are connected to the starting point and all the final activities are connected to the ending point. **"Open"** is an initial activity and **"Closed"** is a final activity. Arrow shows transitions from one state to another and rectangular box represents a state. Frequencies specified on arrows represent count of transitions and activities inside rectangular box represent a state. It basically shows how many times one state (activity) goes to another state (activity) like **"Open"** activity goes 8501 times to **"Status Change"** activity.

In process mining, logs are an essential part of any computing system, supporting error management, like to identify key problems within a business process. As logs grow and their number of sources increases, a scalable system is necessary to efficiently process these logs.

1.2 Comparison of NoSQL Column-Oriented Database and Row-Oriented Database

Row-oriented databases and column-oriented databases support OLAP. They are capable of supporting analytical process mining but have some key differences in performing operations.

1. In a column-oriented database, information about an entity is stored in multiple locations on a disk (e.g. name, e-mail address and phone number are all stored in separate columns) whereas in a row-oriented database, such information is usually located contiguously in a single row of a table.
2. Row-oriented databases are primarily used for transactional processing in comparison to analytical processing while NoSQL column-oriented databases does not support OLTP and focus on OLAP.
3. Compression algorithms perform better on data with low information entropy (high data value locality). Using compression algorithms over column-oriented database has shown significant improvement in query performance [11].

1.3 SQL Interface Over NoSQL Column-Oriented Database

NoSQL column-oriented databases do not support OLTP and focuses on OLAP. To process analytical queries these database use Create/Read/Update/Delete operations, MapReduce job or Client API's but they do not have any SQL like capabilities to perform any analytical queries. The challenges of using client API against SQL over NoSQL databases are:

1. To write an application using MapReduce and Client API's requires expertise.
2. Writing code may not be as straightforward as SQL (Refer Figure 1.3 adapted from¹).
3. Applications are tied closely to specified data model.

¹<http://phoenix.apache.org/presentations/OC-HUG-2014-10-4x3.pdf>


1.3 SQL Interface Over NoSQL Column-Oriented Database

To perform aggregation functions such as min, max, average and count in NoSQL column-oriented database there is a need to run MapReduce jobs. MapReduce is a programming model for processing large datasets in a cluster of machines. This model is based on key-value pair and consists of a Map procedure and Reduce procedure. The Map procedure performs filtering and sorting of all the keys while a Reduce procedure combines all the similar keys. It slows down if a cluster of machines contain distinct large set of keys. Processing these MapReduce jobs will take from minutes to hours.

```
SELECT * FROM foo WHERE bar > 30
```

Versus

```
HTable t = new HTable("foo");
RegionScanner s = t.getScanner(new Scan(...,
    new ValueFilter(CompareOp.GT,
        new CustomTypedComparator(30)), ...));
while ((Result r = s.next()) != null) {
    // blah blah blah Java Java Java
```



Your BI tool
probably can't do
this

Figure 1.3: SQL Interface over NoSQL Column-Oriented Databases

Accessing data stored in tables using SQL interface can be much easier in comparison to using client API's or MapReduce jobs. SQL 2003 supports window functions (like RANK, DENSE_RANK, FIRST_VALUE and LAST_VALUE) for analytical applications. Window functions allow us to remove self-joins and explicit cursors. These functions help in processing analytical applications in lesser time as compared to MapReduce framework. Business intelligence tools can be integrated with SQL interface by using JDBC and ODBC connector. These tools help to retrieve, analyze, process and transform the data in order to improve business performance. Lately, developers are working on a NoSQL column-oriented database to support Atomicity, Consistency, Isolation and Durability (ACID) properties. Many companies like Cloudera¹, Hortonworks² and Apache³ are working to provide SQL layer over NoSQL column-oriented databases.

¹<http://www.cloudera.com>

²<http://www.hortonworks.com>

³<http://www.apache.org>

1.4 Research Aim

The research aims of this study are-

1. To implement α -miner algorithm in SQL. The underlying row-oriented database for implementation is MySQL using InnoDB¹ engine.
2. To implement α -miner algorithm on column-oriented database HBase using Phoenix and HDFS.
3. To conduct series of experiment on publicly available real world dataset, to compare the performance of α -miner algorithm on both the databases. The experiment considers multiple aspects such as α -miner stepwise execution, bulk loading across various datasets, write intensive time, read intensive time, disk space of tables, disk space of tables using compression technique, α -miner stepwise execution using compression technique, real time batch wise insertion and real time single record insertion.

¹<http://dev.mysql.com/doc/refman/5.5/en/innodb-storage-engine.html>

2

Related Work and Research Contributions

The description of related work is taken from our work done in collaboration with Astha Sachdev [12]. In this Section, we review closely related work to the study presented in this thesis and list the novel contributions of our work in context to existing work.

2.1 Related Work

2.1.1 Implementation of Mining Algorithms in Row-Oriented Databases

Ordonez et al. presents an efficient SQL implementation of the EM algorithm [13]. Their approach is to perform clustering in very large databases. It can effectively handle high dimensional data, a high number of clusters and more importantly, a very large number of data records. 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.

Xuequn Shang et al. presents an efficient frequent pattern mining in relational databases [14]. They proposed, called Propad (PROjection PAttern Discovery). Propad fundamentally differs from an Apriori like candidate set generation-and-test approach. This approach successively projects the transaction table into frequent itemsets to avoid making multiple passes over the large original transaction table and generating a huge sets of candidates. Ordonez et al. present a method to implement k-means clustering

algorithm in SQL. They cluster in large datasets in RDBMS [15]. Their work concentrates on defining suitable tables, indexing them and writing suitable queries for clustering purposes.

2.1.2 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 [16]. 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. Suresh L et al. implemented k-means clustering algorithm on column store databases [17]. 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.

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

Hasso conducted common database approach for OLTP and OLAP using an in-memory column database [5]. He presented a comparison of OLAP and OLTP considering row oriented database and column oriented database. 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 [18].

Rana et al. implement Apriori algorithm on MonetDB and Oracle database and compare their performance in terms of execution time [19]. Bazar et al. present a study on the reasons for the transition from relational to column oriented databases [20]. They conduct experiments to perform benchmarking on three column store databases Cassandra, MongoDB and CouchBase. Andreas et al. presents workload representation across different storage architectures for relational DBMS [21]. For relational database management systems, two storage architectures have been introduced: the row-oriented and the column-oriented architecture. To select the optimal architecture for a certain application, we need workload information and statistics. In this paper, we present an

approach that enables us to represent workloads across different DBMSs and architectures.

2.2 Thesis Contributions

In context of existing work, this study makes the following novel contributions:

1. A first implementation of the Process Mining α -miner algorithm on row-oriented databases using MySQL and InnoDB storage engine.
2. A first implementation of the Process Mining α -miner algorithm in HBase using Phoenix and HDFS file system.
3. We present a performance benchmarking of α -miner algorithm in MySQL and HBase on multiple aspects such as α -miner stepwise execution, bulk loading across various datasets, write intensive time, read intensive time disk space of tables, disk space of tables using compression technique, α -miner stepwise execution using compression technique, real time batch wise insertion and real time single record insertion.

3

Introduction of Apache Hadoop, Apache HBase and Apache Phoenix

3.1 Apache Hadoop

¹ The Apache Hadoop framework is composed of the following basic components-

1. **Hadoop Distributed File System (HDFS)**: It is a distributed file system that stores data in a cluster of machines.
2. **Hadoop MapReduce**: It is a programming model that process large volume of data in parallel stored on commodity machines.

3.1.1 HDFS: Hadoop Distributed File System

HDFS is a distributed, scalable and portable file system for the Hadoop framework [22]. Hadoop clusters consist of one namenode and multiple datanodes. The file content is split into large blocks (64 megabytes), and each block of the file is replicated at multiple datanodes [22]. The namenode monitors the number of replicas of a block in multiple datanodes. When a replica of a block is lost due to a datanode failure (dead datanode), the namenode creates another replica of the block and store it in live datanode.

For this study, HDFS default block size is configured to 64 MB and the overall real dataset size is 40 MB. As mentioned above, HDFS splits the file into default block size and puts it in different datanodes. For example, a 128 MB source file will split into two

¹<http://opensource.com/life/14/8/intro-apache-hadoop-big-data>

3.2 Apache HBase-NoSQL Column-Oriented Database

64 MB blocks and these 64 MB blocks will reside in datanode. In this case, we have a 40 MB dataset file and the default block size of HDFS is 64 MB. Hence, the file is not split. All operation are being performed on a single machine. There is no downside for storing smaller files with larger block size in HDFS. For example, we have a system with 300 MB HDFS block size. To store a 1100 MB file, HDFS will break that file into 300 MB blocks and store it on datanodes. Note that last split file is not exactly divisible by 300. Therefore, final block of the file is sized as modulo of the file by block size, i.e a 200 MB block size. There will be no waste of space because it is not equivalent to traditional file systems.

The differences between HDFS and a generic file system are-

1. Data on HDFS block is key-value (support sequence file format).
2. Data on HDFS blocks is immutable (cannot be modified) but can be appended.
3. The default block size of HDFS is 64 MB while generic file system has 4 KB.
4. HDFS built on top of POSIX.
5. Suppose if we want to store file size of 2 KB in HDFS, then all the remaining space of block (64 MB) can be reused by other files. Vice-versa for generic file system.

3.2 Apache HBase-NoSQL Column-Oriented Database

3.2.1 Introduction

Apache HBase is similar to the Google Bigtable [23] and has extended various features of it. HBase is a NoSQL column-oriented store. The characteristics of HBase are Sparse, Consistent, Distributed, Multidimensional and Sorted map

HBase provides random and real time read/write access to the stored data but mostly it is important to know, how to retrieve data stored in HBase tables efficiently. Naive users will try to do it by using a MapReduce job or Client API's. These are not efficient depending upon application requirements. SQL interface with advanced features over HBase are more efficient than MapReduce job and easy for naive users. However, HBase has real time processing which can help the process discoveries algorithms to discover process models in a real time.

3.2.2 Architecture

1

Figure 3.1, is an architecture of HBase and consists of:

1. Master node: Master node is similar to the namenode of Hadoop framework.
2. Slave node: Each datanode is slave node and has one regionserver.
3. Zookeeper: Zookeeper is a distributed coordinator management of clusters. Master and regionserver will be linked to Zookeeper by registering themselves.
4. Region: Storing table in HBase requires regions. One table can have multiple regions and within each region there is one memstore and multiple stores. Multiple regions are stored in a regionserver.
5. WAL (Write Ahead Log): Before writing the data to region, first it is stored in WAL and then it goes to a particular region of a regionserver. There is one WAL for every corresponding regionserver.

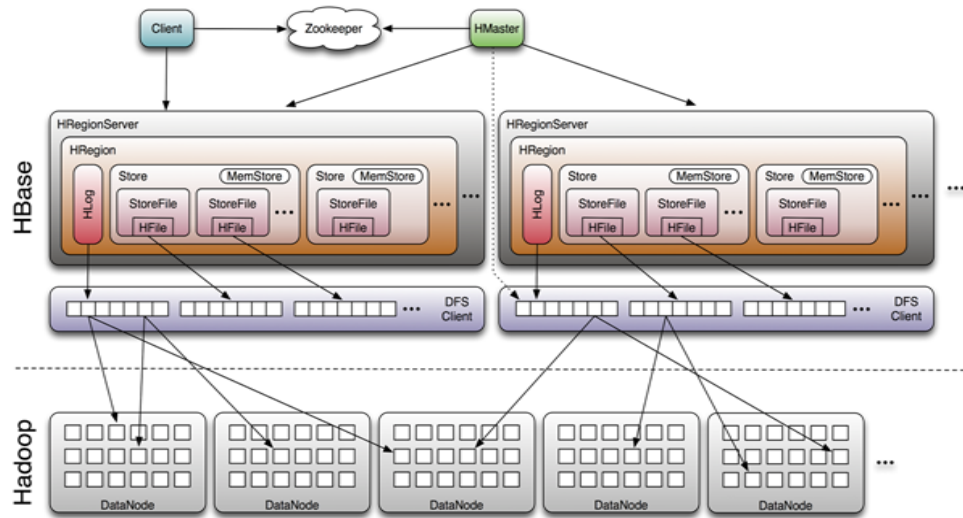


Figure 3.1: HBase Architecture Adapted From [1]

¹<http://www.cyanny.com/2014/03/13/hbase-architecture-analysis-part2-process-architecture/>

3.3 Apache Phoenix-SQL Skin Over HBase

Apache Phoenix¹ is a relational database layer over HBase. It work as a client JDBC driver over HBase and targets low latency queries over HBase data. It increases performance by

1. Converting queries into HBase scans.
2. Automating arrangement of parallel execution for scans in a table.
3. Using push down predicates that bring the computation near to the data (datanodes).
4. Executing aggregate queries through co-processors.

Figure 3.2, shows Phoenix integration in HBase architecture. Adapted from²

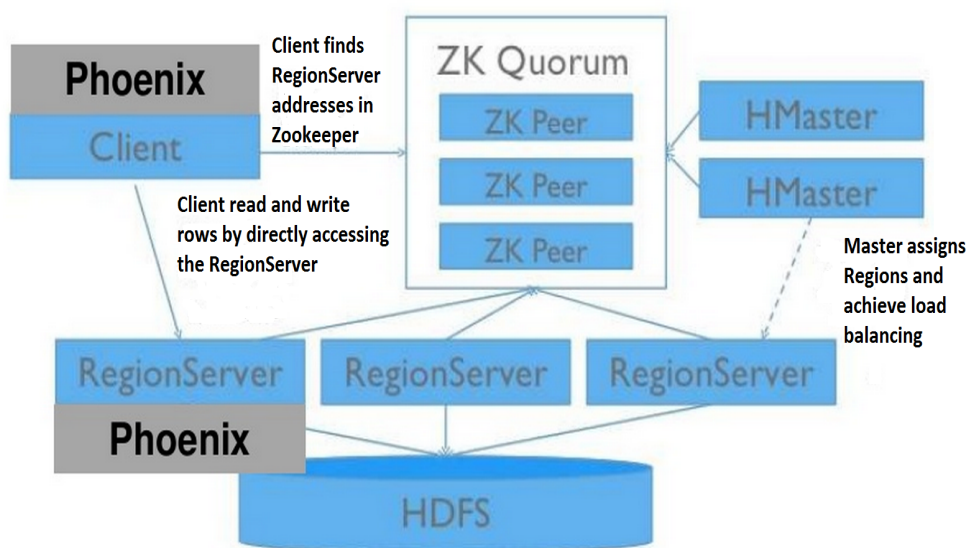


Figure 3.2: HBase-Phoenix Architecture

¹<http://phoenix.apache.org/>

²http://www.slideshare.net/Hadoop_Summit

4

Description of α -Miner Algorithm with an Example

4.1 Description of α -Miner Algorithm

The description of α -miner algorithm is taken from our work done in collaboration with Astha Sachdev¹. The α -miner algorithm is an algorithm used in discovering process mining, [4]. It was first put forward by van der Aalst, Weijter and Maruster [4]. Input for the α -miner 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 $a \succ_L b \wedge b \not\succ_L a$ (b does not precedes a).
3. $a \parallel b$ iff $a \succ_L b$ and $b \succ_L a$ in some trace.
4. $a \# b$ iff $a \not\succ_L b \wedge b \not\succ_L a$.

Let L be an event log over $T \subseteq A$. $\alpha(L)$ is defined as follows [4].

1. $T_L = \{ t \in T \mid \exists \sigma \in L \ t \in \sigma \}$
2. $T_I = \{ t \in T \mid \exists \sigma \in L \ t = \text{first}(\sigma) \}$

¹<https://repository.iiitd.edu.in/jspui/bitstream/123456789/220/1/MT2013034.pdf>

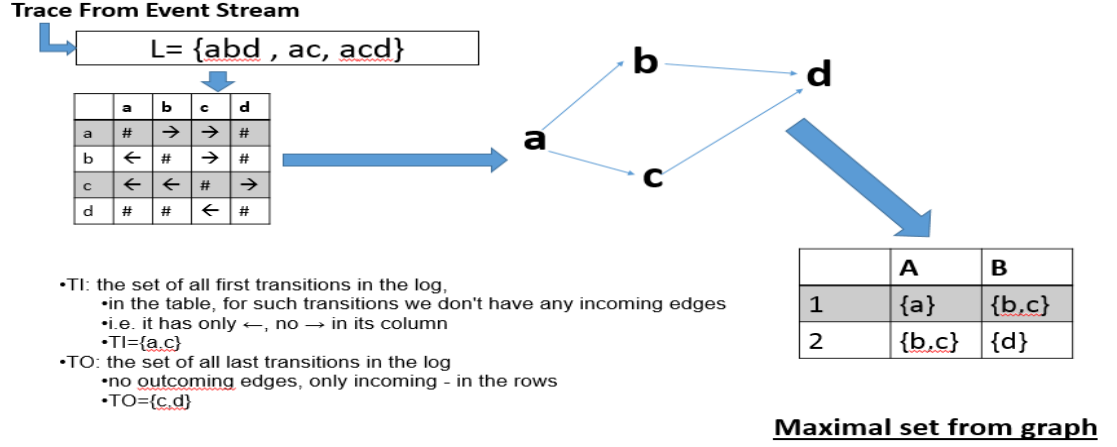
3. $T_O = \{ t \in T \mid \exists \sigma \in L \ t = \text{last}(\sigma) \}$
4. $X_L = \{ (A,B) \mid \{A \subseteq T_L \wedge A = \emptyset\} \wedge \{B \subseteq T_L \wedge B = \emptyset\} \wedge \{ \forall a \in A \forall b \in B \ a \rightarrow_L b \} \}$
5. $Y_L = \{ (A,B) \in X_L \mid (A,B) \in X_L \ A \subseteq A \wedge B \subseteq B \longrightarrow (A,B)=(A,B) \}$
6. $P_L = \{ P(A,B) \mid (A,B) \in Y_L \} \cup (i_L, o_L)$
7. $F_L = \{ (a, P(A,B)) \mid (A,B) \in Y_L \wedge a \in A \} \cup \{ (P(A,B), b) \mid (A,B) \in Y_L \wedge b \in B \} \cup \{ (i_L, t) \mid t \in T_I \} \cup \{ (t, o_L) \mid t \in T_O \}$
8. $\alpha(L) = (P_L, T_L, F_L)$

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

1. The Step 1 computes T_L (Total Events) which represents the set of distinct activities present in the event log L .
2. The Step 2 computes T_I (Initial Events) which computes the set of all the initial activities of corresponding trace.
3. Step 3 computes T_O (Final Events) which represents the set of distinct activities which appear at the end of some trace in the event log.
4. In order to compute Step 4, we compute the relationships between all the activities in T_L . This computation is presented in the form of a footprint matrix and is called pre-processing in α -miner 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. Step 5 keeps only the maximal pairs of sets generated in the fourth step, eliminating the non-maximal ones.
6. Step 6 adds 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. Step 7 is the final step of the α -miner algorithm that presents all the places including the input and output places and all the input and output transitions from the places.

4.2 Example of α -miner Algorithm

Figure 4.1, shows a trace coming from an event stream. Trace contains $\{abd, ac, acd\}$ so footprint (table) will be defined over all events in trace. Considering all the events in a trace, we will get initial events and final events. Fig. 4.1, shows the trace representing relations like Causality, Parallel, NotConnected and Precedes. From this


 Figure 4.1: Example of α -miner Algorithm

table, we conclude which activity follows which type of relation and can build a directed graph shown in Fig. 4.1. After creating graph, we need to find out maximal event set. Maximal event set can be defined by looking causality relation in a graph like $a \rightarrow b$, $a \rightarrow c$, $b \rightarrow d$, $c \rightarrow d$ and need to follow rule describe below. Recall in Figure 4.1, A and B column in last table.

1. $\forall a_1, a_2 \in A: a_1 \nparallel a_2$
2. $\forall b_1, b_2 \in B: b_1 \nparallel b_2$
3. $\forall a \in A, b \in B: a \rightarrow b$

According to the above specified rules we can generate maximal set like $\{a\} \rightarrow \{b,c\}$ and $\{b,c\} \rightarrow \{d\}$ and from this maximal set we can generate process model as shown in Figure 4.2.

We use the experimental dataset (Refer Chapter 7). The input to the α -miner algorithm is the event log (Refer Fig. 7.1). We obtain the output in the form of a table that shows the input and the output transitions. Fig. 4.3, shows a part of the output we obtain in α -miner algorithm. The column activity_{in} represents the set of input activities and the another column activity_{out} represents the set of output activities. All the activities in the set represented by activity_{in} are not connected to each other and all the activities in the set represented by activity_{out} are not connected to each other. All the activities in the set represented by activity_{in} have causality relationship with all other activities represented by the set in activity_{out}. This figure is a snapshot of the FW table that we obtain in Apache Phoenix.

4.2 Example of α -miner Algorithm

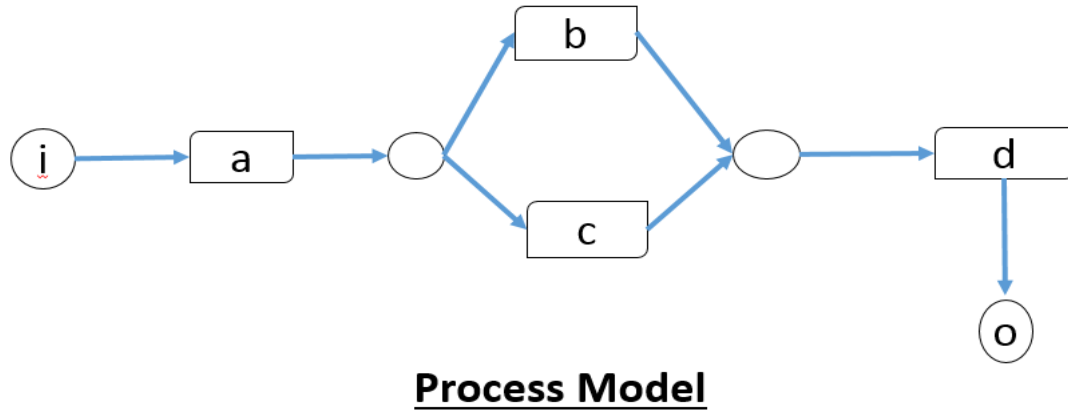


Figure 4.2: Continued Example of α -miner Algorithm

Analysis/Research & External update	External update
Communication with customer & Problem Workaround, Incident reproduction, OO Response	Problem Workaround
Communication with customer & Problem Workaround, Incident reproduction, OO Response	Incident reproduction
Communication with customer & Problem Workaround, Incident reproduction, OO Response	OO Response
Communication with vendor & External Vendor Reassignment	External Vendor Reassignment
Contact Change, Problem Closure & Description Update	Description Update
Description Update & Notify By Change, Callback Request	Notify By Change
Description Update & Notify By Change, Callback Request	Callback Request
External update, Affected CI Change, Service Change, OO Response & External Vendor Assignment	External Vendor Assignment
External update, Problem Closure, Callback Request & Update from customer	Update from customer
External Vendor Assignment & Notify By Change, Problem Closure	Notify By Change
External Vendor Assignment & Notify By Change, Problem Closure	Problem Closure
External Vendor Reassignment & Vendor Reference Change, Impact Change	Vendor Reference Change
External Vendor Reassignment & Vendor Reference Change, Impact Change	Impact Change
External Vendor Reassignment, Resolved & Vendor Reference Change	Vendor Reference Change
Impact Change & External update	External update
Incident reproduction, alert stage 1 & Status Change	Status Change
Incident reproduction, OO Response & Caused By CI	Caused By CI
Mail to Customer & Affected CI Change, Problem Closure, Callback Request, Quality Indicator, Impact Change	Affected CI Change
Mail to Customer & Affected CI Change, Problem Closure, Callback Request, Quality Indicator, Impact Change	Problem Closure
Mail to Customer & Affected CI Change, Problem Closure, Callback Request, Quality Indicator, Impact Change	Callback Request

Figure 4.3: α -miner Algorithm-Input and Output Transitions

Implementation of α -Miner Algorithm in SQL on Row-Oriented Database (MySQL) and Column-Oriented Database (HBase)

5.1 Implementation of α -Miner Algorithm in SQL on Row-Oriented Database (MySQL)

Earlier in our joint work we have implemented α -miner algorithm in SQL (MySQL)¹ [12] and Cassandra-NoSQL column oriented database but it need some optimization. For our current work, MySQL code has been taken from Khanan: Performance Comparison and Programming Alpha Algorithm in Column-Oriented and Relational Database Query Languages [12] and we have optimized it. Therefore, it can perform efficiently on large datasets. For generating power set for activities in MySQL code is taking days so we have optimized it for our study. Finding initialEvent in MySQL code for large dataset is taking lot of time so we have optimized SQL query of generating initialEvent too. Before implementing α -miner algorithm, we do pre-processing in JAVA to create the following two tables *viz.* causality table (consist of two column eventA and eventB) and NotConnected table (consist of two column eventA and eventB). For more detailed information of an implementation refer to Appendix A. The below table describe about schema representation of table in MySQL where input table consist of four columns (CaseID, Timestamp, Status, Activity) where we consider Primary Key a combination of CaseID, Timestamp and Status. The reason of choosing three column for primary key because there is a case where one CaseID with same Timestamp is redundant in

¹Link is <https://repository.iiitd.edu.in/jspui/bitstream/123456789/220/1/MT2013034.pdf>

5.1 Implementation of α -Miner Algorithm in SQL on Row-Oriented Database (MySQL)

CaseID	Timestamp	Status	Activity
1	2014-09-12 12:43:06	001A3689763	InProgress
1	2014-12-19 01:23:06	001A9085723	Closed
2	2014-09-22 02:14:09	001B3645873	Cancelled
2	2014-11-23 13:04:09	001B3641275	Closed

Table 5.1: Representation of Schema Table in MySQL

CSV file that why we consider Status as another column for primary key. For each column we have column Family so that HBase has different storefile with in region of regionserver for each column.

1. We create a table eventlog using create table¹ keyword consisting of 5 columns (CaseID, Timestamp, Status, Activity and Actor) each of which are varchar datatype except Timestamp which is of timestamp datatype. The primary key is a composite primary key consisting of CaseID, Timestamp and Status.
2. We load the data into table eventlog using LOAD DATA INFILE² command.
3. For Step 1, we create a table totalEvent that contains a single column (event) which is of varchar datatype. To populate the table we select distinct activities from the table eventlog.
4. For Step 2, we create a table initialEvent that contains a single column (initial) which is of varchar datatype. To populate the table
 - (a) We first select the minimum value of Timestamp from table eventlog by grouping CaseID.
 - (b) Then we select distinct activities from table eventlog for every distinct value of CaseID where Timestamp is the minimum Timestamp.
5. For Step 3, we create a table finalEvent that contains a single column (final) which is of varchar datatype. To populate the table
 - (a) We first select maximum Timestamp from a table eventlog by grouping CaseID.
 - (b) Then we select distinct activities from a table eventlog for every distinct value of CaseID where Timestamp is the maximum Timestamp.
6. For Step 4, we create five tables *viz.* SafeEventA, SafeEventB, EventA, EventB and XL. All the five tables contain two columns (setA and setB) which are of varchar datatype.

¹<http://dev.mysql.com/doc/refman/5.1/en/create-table.html>

²<http://dev.mysql.com/doc/refman/5.1/en/load-data.html>

5.1 Implementation of α -Miner Algorithm in SQL on Row-Oriented Database (MySQL)

- (a) In table causality we use `group_concat`¹ to combine the values of column eventB of corresponding value of a column eventA and insert the results in a table EventA.
 - (b) In table causality we use `group_concat` to combine the values of column eventA of corresponding value of a column eventB and insert the results in the table EventB.
 - (c) To populate tables SafeEventA and SafeEventB-
 - i. Select setA and setB from tables EventA and EventB
 - ii. For every value of setB in table EventA, if value is present in table notconnected, insert the corresponding value of setA and setB in table SafeEventA. Repeat the same step for populating table SafeEventB.
 - (d) To populate table XL, we insert all the rows from the three tables SafeEventA, SafeEventB and causality.
7. For Step 5, we create three tables *viz.* eventASafe, eventBSafe and YL. All the three tables contain two columns (setA and setB) which are of varchar datatype.
- (a) We create a stored procedure to split the values of column setB of table SafeEventA on comma separator. Insert the results in safeA table.
 - (b) We create a stored procedure to split the values of column setA of table SafeEventB on comma separator. Insert the results in safeB table.
 - (c) To populate table eventASafe, insert all the rows from table safeA.
 - (d) To populate table eventBSafe, insert all the rows from table safeB.
 - (e) To populate table YL, insert all the rows from tables SafeEventA, SafeEventB, eventASafe, eventBSafe and causality.
8. For Step 6, we create two tables *viz.* terminalPlace that contains a single column (event) which is of varchar datatype and PL which also contains a single column (Place) which is of varchar datatype.
- (a) To populate table terminalPlace, insert 'i' and 'o' in the table.
 - (b) To populate table PL, we use `concat_ws`² to combine the values of column setA and column setB of a table YL using & separator and insert the results in table PL. Furthermore, we insert all the rows of table terminalPlace into table PL.
9. For Step 7, we create 3 tables *viz.* Place1 and Place2 which consist of two columns (id and value) which are of varchar datatype and FL which consists of two columns (firstplace and secondplace) which are of varchar datatype.

¹http://dev.mysql.com/doc/refman/5.0/en/group-by-functions.html#function_group_concat

²http://dev.mysql.com/doc/refman/5.0/en/string-functions.html#function_concat_ws

5.2 Implementation of α -Miner Algorithm on NoSQL Column-Oriented Database (HBase) Using Apache Phoenix

- To populate table Place1, we use concat_ws to combine the values of column setA and column setB of table YL using & separator. Insert the results in column setB of table Place1. Insert all the values of column setA of table YL into column setA of table Place1.
- To populate table Place2, we use concat_ws to combine the values of column setA and column setB of table YL using & separator. Insert the results in column setA of table Place2. Insert all the values of column setB of table YL in column setB of table Place2.
- We create a stored procedure to split column setB of table Place1 on comma separator. In stored procedure we create table temp_place2 to insert the results.
- We create a stored procedure to split column setA of a table Place2 on comma separator. In stored procedure we create table temp_place2 to insert the results.
- To populate table FL, insert all the rows from tables temp_place1 and temp_place2. Insert the results of cross join of two tables *viz.* terminalPlace and intialEvent and of table finalEvent and table terminalPlace.

5.2 Implementation of α -Miner Algorithm on NoSQL Column-Oriented Database (HBase) Using Apache Phoenix

Table 5.2: Comparison of Apache Phoenix and MySQL Query Language

SQL Commands	Comparison of HBase and MySQL	
	MySQL	HBase
Creating and altering Database	Yes	No
Creating and calling Stored Procedures	Yes	No
Creating and calling user functions	Yes	No
Defining Block Size on table creation	Yes	Yes
Defining different Compression technique on table creation	No	Yes
Creating Triggers	Yes	No
Multiple Insert SQL statments like Insert Ignore	Yes	No
Support Array and Array functions	No	Yes
Update Statistic of running queries	No	Yes
Multi-tenant Table	No	Yes
Salting on created table	No	Yes
Geospatial	Yes	No
Support Window Function	Yes	Yes (Less in number)
Dynamic column addition	No	Yes

The below table describe about schema representation of table in HBase where input table consist of three columns (CaseID, Timestamp, Activity) where we consider row key of HBase a combination of CaseID, Timestamp and Status. The reason of choosing three column for row key because there is a case where one CaseID with same Timestamp is redundant in CSV file that why we consider Status as another column for row key.

5.2 Implementation of α -Miner Algorithm on NoSQL Column-Oriented Database (HBase) Using Apache Phoenix

For each column we have column Family so that HBase has different storefile with in region of regionserver for each column.

Table 5.3: Representation of Schema Table in HBase

RowKey	ColumnFamily:Timestamp:	ColumnFamily:Activity:
CaseIdTimestampStatus	Column:Timestamp	Column:Activity
1-2014-09-22 23:21:45-001A368	2014-09-22 23:21:45	InProgress
1-2014-10-12 23:21:45-001A3653	2014-10-12 23:21:45	Resolved
2-2014-11-02 13:21:45-001B3662	2014-11-02 13:21:45	Change
2-2014-11-02 13:21:45-001B3914	2014-11-02 13:21:45	Cancelled

To implement α -miner algorithm in HBase. First of all we need Apache Hadoop that act as distributed file system for storing large dataset, Apache HBase is column oriented database that will work as datastore over Hadoop and Apache Phoenix act as SQL layer over Apache HBase. Before implementing α -miner algorithm, we do pre-processing in JAVA to create the following two tables *viz.* causality table (consist of two column eventA and eventB) and NotConnected table (consist of two column eventA and eventB). For more detailed information of an implementation refer to Appendix B.

1. To create table eventlog (Refer section 5 point 1). To load the data in table eventlog, we use MapReduce framework¹.
2. For Step 1, 2 and 3 (Refer section 5 point 3, point 4 and point 5)
3. For Step 4, we create three tables *viz.* SafeEventA, SafeEventB and XL. All the three tables consist of two columns (setA and setB) which are of varchar datatype.
 - (a) Select values of column eventA and eventB from the table causality.
 - (b) Select values of column setA and setB from the table notconnected.
 - (c) Compare Function that compares whether set of activities is notconnected.
 - (d) Loop over values of column eventA in the table causality. Form single group say *grp* of all activity present in column eventB. Pass *grp* to Compare function. For any such combination returning true, insert eventA in setA and that combination into setB of table SafeEventA.
 - (e) Loop over values of column eventB in the table causality. Form single group say *grp* of all activity present in column eventA. Pass *grp* to Compare function. For any such combination returning true, insert that combination into setA and eventB in setB of table SafeEventA.
 - (f) To populate table XL, we insert all the rows from three tables SafeEventA, SafeEventB and causality.
4. For Step 5, we create three tables *viz.* EventA, EventB and YL. All the three tables consist of two columns (setA and setB) which are of varchar datatype.

¹http://phoenix.apache.org/bulk_dataload.html

5.2 Implementation of α -Miner Algorithm on NoSQL Column-Oriented Database (HBase) Using Apache Phoenix

- (a) Select values of column setA and setB from the table SafeEventA.
 - (b) Select values of column setA and setB from the table SafeEventB.
 - (c) Loop over values of column setA and setB present in the table SafeEventA. Delimit value of setB. For all such value setB_i, insert setA and setB_i in table EventA.
 - (d) Loop over values of columns setA and setB present in the table SafeEventB. Delimit value of setA. For all such value setA_i, insert setA_i and setB in table EventB.
 - (e) To populate table YL, we insert all the rows from three tables EventA, EventB and causality.
5. For Step 6 (Refer section 5 point 8).
6. For Step 7, we create table FL that consists of two columns (Place1 and Place2) which are of varchar datatype.
- (a) Select values of column setA and setB from the table YL.
 - (b) Select values of column final from the table FinalEvents.
 - (c) Select values of column initial from the table InitialEvents.
 - (d) Loop over values of column final in the table FinalEvents. Insert values of column final in column Place1 and 'o' in column Place2 of table FL.
 - (e) Loop over values of column initial in the table InitialEvents. Insert 'i' in column Place1 and values of column initial in column Place2 of table FL.
 - (f) Loop over values of column setA and setB in the table YL. If value of column setA has set of activities instead of single activity then delimit. Each split value will be stored in column Place1 and combination of values of columns setA and setB in column Place2 of table FL else choose column setB and delimit. Each split value will be stored in column Place2 and combination of values of columns setA and setB in column Place1 of table FL.

5.3 Output of α -Miner Algorithm from the Database

The Fig. 5.1 shown below is small output of α -miner algorithm. It depicts the output of implemented α -miner algorithm in both the databases (MySQL and HBase).

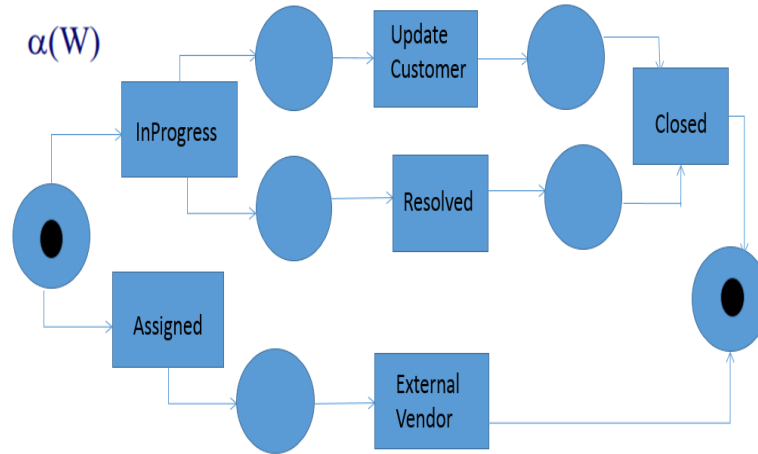


Figure 5.1: Small output of α -miner algorithm

6

Experimental Dataset

We conduct our study on a publicly available large real world dataset downloaded from Business Process Intelligence 2014 (BPI 2014)¹. The dataset is provided by Robobank Information and Communication and Technology (ICT) organization. The data is related to Information Technology Infrastructure Library (ITIL) process implemented in the bank. ITIL is a process which starts when a client reports an issue regarding disruption of ICT service to Service Desk Agent (SDA). SDA records the complete information about the problem in an Interaction record. If the issue does not get resolved on first contact then an Incident record is created for the corresponding Interaction else the issue is closed. If an issue appears frequently then a request for change is initiated. Robobank provides 4 files in CSV format *viz.* Change records, Incident records, Interaction records and Incident activity records. We imported Incident activity records CSV file in MySQL and HBase for benchmarking and performance comparison of α -miner algorithm. Incident activity records file contains 4,66,738 number of records and contains the following fields *viz.* Incident ID, DateTimeStamp, IncidentActivity_number, IncidentActivity_Type, Interaction ID, Assignment Group and KM Number. Out of these we use the following fields:

1. Incident ID: The unique ID of a record in the Service Management tool. It is represented as CaseID in our data model.
2. DateTimeStamp: Date and time when a specific activity starts. It is represented as timestamp in our data model.
3. IncidentActivity_Type: Identifies which type of an activity takes place.
4. Assignment Group: It represent the team responsible for an activity.

¹<http://www.win.tue.nl/bpi/2014/start>

caseid	timestamp	status	activity	actor
M0000004	2013-01-07 08:17:17	001A3689763	Reassignment	TEAM0001
M0000004	2013-09-25 08:27:40	001A5544096	Operator Update	TEAM0003
M0000004	2013-11-04 12:09:37	001A5849978	Reassignment	TEAM0003
M0000004	2013-11-04 12:09:37	001A5849979	Assignment	TEAM0003
M0000004	2013-11-04 12:09:37	001A5849980	Operator Update	TEAM0003
M0000004	2013-11-04 13:41:30	001A5852941	Reassignment	TEAM0002
M0000004	2013-11-04 13:41:30	001A5852942	Assignment	TEAM0002
M0000004	2013-11-04 13:41:30	001A5852943	Update from customer	TEAM0002
M0000004	2013-11-04 13:51:18	001A5852172	Closed	TEAM0003
M0000004	2013-11-04 13:51:18	001A5852173	Caused By CI	TEAM0003
M0000005	2013-01-07 08:17:54	001A3689771	Reassignment	TEAM0001
M0000005	2013-03-27 08:57:33	001A4287716	Operator Update	TEAM0003
M0000005	2013-03-27 08:57:33	001A4287717	Description Update	TEAM0003
M0000005	2013-04-03 11:29:45	001A4327775	Reassignment	TEAM0003
M0000005	2013-04-03 11:29:46	001A4327776	Assignment	TEAM0003
M0000005	2013-04-03 11:29:46	001A4327777	Operator Update	TEAM0003
M0000005	2013-04-12 11:03:27	001A4396942	Status Change	TEAM0003
M0000005	2013-04-12 11:03:27	001A4396943	Operator Update	TEAM0003
M0000005	2013-04-16 14:19:09	001A4419475	Assignment	TEAM9999
M0000005	2013-04-16 14:19:09	001A4419476	Status Change	TEAM9999
M0000005	2013-04-23 08:22:09	001A4466088	Status Change	TEAM0003
M0000005	2013-05-24 08:42:34	001A4664412	Reassignment	TEAM9999

Figure 6.1: Event Log

Fig. 6.1, shows the dataset that is used in our experimentation. Since α -miner algorithm is concerned only with the sequence of events, and we need to 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, timestamp which is the timestamp value giving the sequence order of activities for a particular caseid and activity which represents the activities in our data model. Thus, we load only these three fields in our event log table, and corresponding to each caseid we obtain the sequence of events as the events are ordered according to the timestamp values.

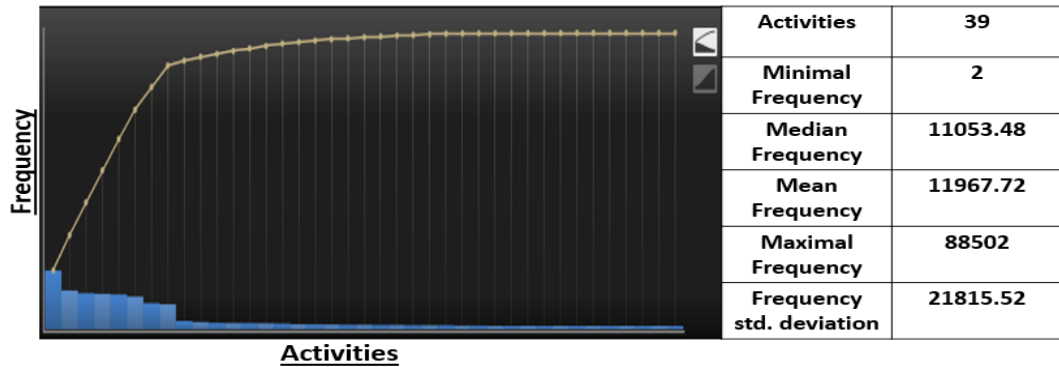


Figure 6.2: Activities in BPI 2014 Dataset

Fig. 6.2, shows activities of BPI 2014 dataset. It shows number of activities in an event log with frequency. On right side of image-

1. we can see number of distinct activities is 39 in an event log.
2. Minimum frequency of one activity is 2.
3. Mean, median and standard deviation of frequency of an activities.
4. One activity is an event log, whose frequency is maximum.



Figure 6.3: Number of Activities Per Case in BPI 2014 Dataset

Fig. 6.3, shows activities of BPI 2014 dataset. It shows number of activities in an event log per cases. On right side of image-

1. we can see number events in an event log that is 4,66,738.
2. Number of cases in an event log is 46,616.
3. Mean, median of case duration is also shown.
4. Start and end time describes only starting time of event in an event log and end time of an event log.

Fig. 6.4, shows frequency of events in cases of BPI 2014 dataset. Fig. 6.4, shows top 13 cases arranged in descending order of frequency of events.

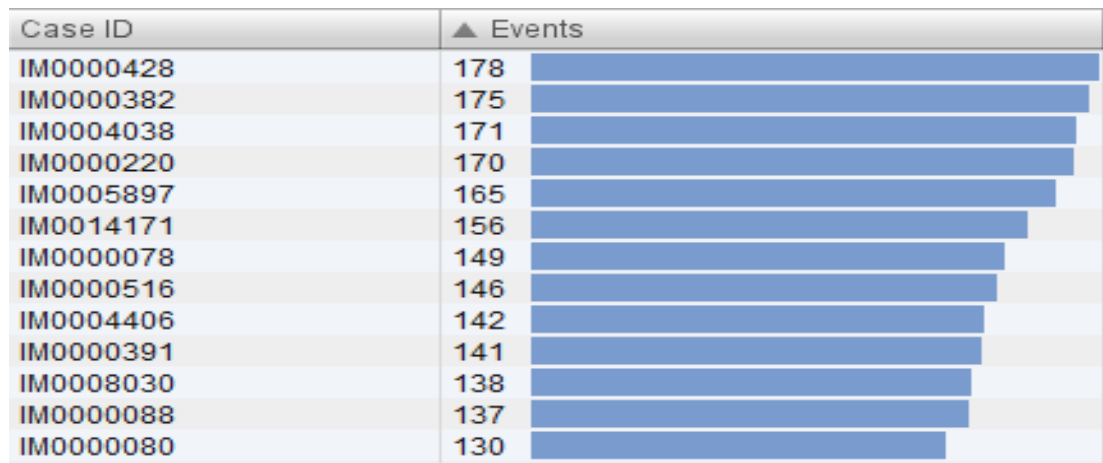


Figure 6.4: Matrix of Events per Case

Benchmarking and Performance Comparison

Our benchmarking system consists of Intel Core i3 2.20 GHz processor, 4 GB Random Access Memory (RAM), 500 GB Hard Disk Drive (HDD), Operating System (OS) is Linux Ubuntu 14.04 LTS and Cache of 3 MB. The experiments were conducted on MySQL 5.6 (row-oriented database) and HBase 0.96.1 (NoSQL column-oriented database) with HDFS 2.3.0 as the file system below it and a layer of Phoenix 4.2.1 above it. We conduct series of experiments on a single machine.

The α -miner algorithm interacts with the database. The underlying data model for implementing α -miner algorithm consists of 5 columns (CaseID, Timestamp, Status, Activity and Actor) each of which are of datatype varchar except Timestamp which is of timestamp datatype. The primary key is a composite primary key consisting of CaseID, Timestamp and Status. We use the same data model while performing bulk loading of datasets through the database loader. We take each reading five times for all the experiments and the average of each reading is reported in the thesis. Output of the α -miner algorithm is same in both the databases (MySQL and HBase).

7.1 Loading Multiple Datasets

Table 7.1: Dataset Load Time

Dataset Size	Load Time in Seconds	
	MySQL	HBase
1,00,000	12.98	12.03
4,00,000	46.79	42.94
8,00,000	156.79	64.48
12,00,000	3654.14	89.55
16,00,000	8408.20	123.85
20,00,000	13536.42	145.53

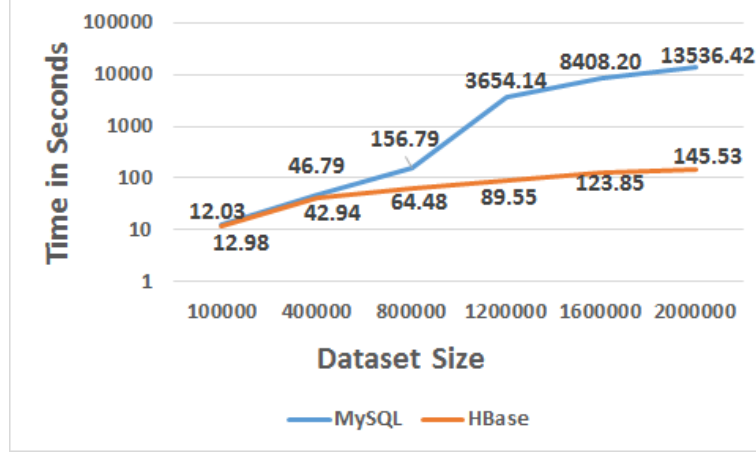


Figure 7.1: Dataset Load Time in Seconds

Our first experiment consists of investigating the time taken to perform bulk loading in both the databases across various dataset sizes. Table 7.1 shows that the average time taken to load data in HBase is 29 times lower as compared to MySQL.

This is likely because bulk loading in HBase is done using the MapReduce framework. The Phoenix database layer has an inbuilt script of MapReduce using which we conduct our experiment. We use two mappers and two reducers for running MapReduce jobs. The script requires two parameters before running MapReduce *viz.* input file and output file. Input file must be present in HDFS and script creates empty output file in HDFS after executing it. Due to parallelism, all the key-values of the input file are mapped to two mappers and the output of each mapper is passed to two reducers. MapReduce converts all the data of the input file into the format of HFiles (HBase file format) before it handovers to HBase. HFile stores data in key-value pairs and reducers also generate output in key-value pairs. The output of reducers can be stored on multiple HFiles directly without interacting with HBase. At the end all the created HFiles will be handovered to HBase to store on HDFS. In contrast, bulk loading in MySQL is done using LOAD DATA INFILE command which is designed for mass loading of records in a single operation as it overcomes the overhead of parsing and flushing batch of inserts stored in a buffer to MySQL server. The LOAD DATA INFILE command also creates an index and checks uniqueness while inserting records in a table. Therefore, in case of MySQL, while inserting large datasets, most of the time is spent in checking uniqueness and creating indexes. Fig. 7.1 reveal that when the dataset size increases then the difference between the time taken in loading data in MySQL and HBase also increases. The performance of HBase is better as compared to MySQL because the percentage increase of time in MySQL is 3.5 times more as compared to HBase.

7.2 Execution of α -Miner Algorithm

Table 7.2: Stepwise Execution Time

Steps	Execution Time in Seconds	
	MySQL	HBase
1	4.19	2.89
2	6.29	5.82
3	6.71	5.74
4	4.09	3.89
5	8.23	5.64
6	2.04	1.00
7	9.18	3.09

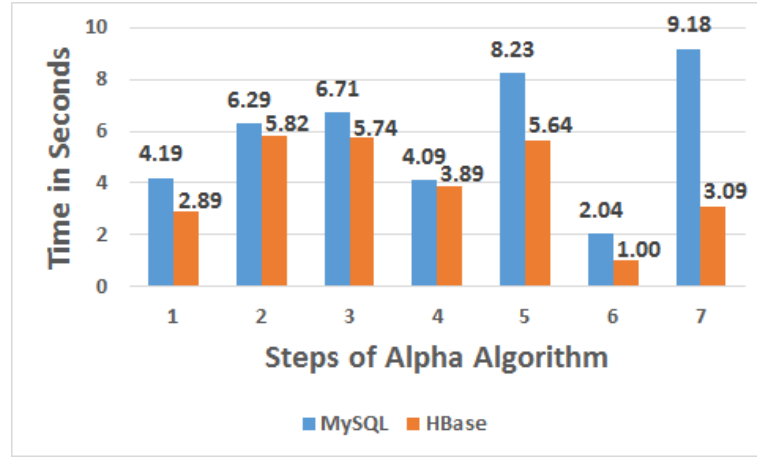


Figure 7.2: α -miner Stepwise Execution

α -miner algorithm is a seven step algorithm (Refer to Section 3). We perform an experiment to compute the α -miner algorithm execution time of each step in both MySQL and HBase to examine which database performs better for each step. In MySQL default size of `innodb_buffer_pool_size` is 8 MB that is used to Cache data and indexes of its tables. The larger we set this value, the lesser is the disk I/O needed to access the data in tables. Table 7.2 and Fig. 7.2 reveal that the the stepwise time taken in HBase is always lower as compared to MySQL for all the Steps. We conjecture the reason for HBase performing better than MySQL can be the difference in the internal architecture of MYSQL and HBase. For the first three steps, both MySQL and HBase perform full table scans. In case of MySQL, the entire row is first retrieved sequentially and then the specific attributes are retrieved. However, in case of HBase, table is stored on multiple regions and Phoenix performs parallelism on multiple regions of a table leading to better performance of HBase in comparison to MySQL. Furthermore, in HBase, only the specific attributes specified in the query are retrieved. The overhead of retrieving

7.3 Read Intensive Steps of α -Miner Algorithm

the entire row is not present in HBase. Hence, HBase gives a better performance for the first three steps.

The remaining steps read data from the tables obtained in the first three steps and write it to the tables created during their execution. In MySQL, in order to read the data from a table we need to scan the B-Tree index to find the location of block where data is stored. In case of HBase data is read from the memstore. If values are not in memstore they are read from HDFS. Thus, the read performance of HBase is better as compared to MySQL. Similarly, in MySQL, in order to write data, the entire B-Tree index needs to be scanned to locate the block where we need to write data. HBase follows log structure merge tree index. In case of HBase, values are written in append only mode. The writes in HBase are sequential because first it writes to Write Ahead Log (WAL) of regionserver and then to memstore of corresponding region. HBase lags in persisting data to disk. Hence, HBase gives better write performance as compared to MySQL. Therefore, the total execution time of α -miner algorithm in HBase is 1.44 times lower than that of MySQL.

7.3 Read Intensive Steps of α -Miner Algorithm

Table 7.3: Read Intensive Time

Steps	Read Time in Seconds	
	MySQL	HBase
1	2.06	1.60
2	4.78	4.48
3	4.95	4.70
4	1.36	1.29
5	0.37	0.37
6	0.12	0.11
7	1.06	0.19

Few Steps of α -miner algorithm are read intensive while few steps are write intensive. We conduct an experiment to investigate which of the steps of α -miner algorithm are read intensive as well as we compare which of the database performs better for read operations. As can be seen from Fig. 7.3 and Table 7.3, the first three steps are read intensive as compared to other steps of α -miner algorithm. From the experimental results, we conclude that HBase gives better read performance as compared to MySQL for all the Steps. According to us, the reason for HBase giving better read performance can be the difference in the data structure of both the databases. In MySQL, B-Tree index needs to be scanned to find the location of block where the data is stored. In case of HBase data is read as described below-

1. To find the data, HBase client will hit the memstore first.
2. When the memstore fails, HBase client will hit the BlockCache [24].

7.4 Write Intensive Steps of α -Miner Algorithm

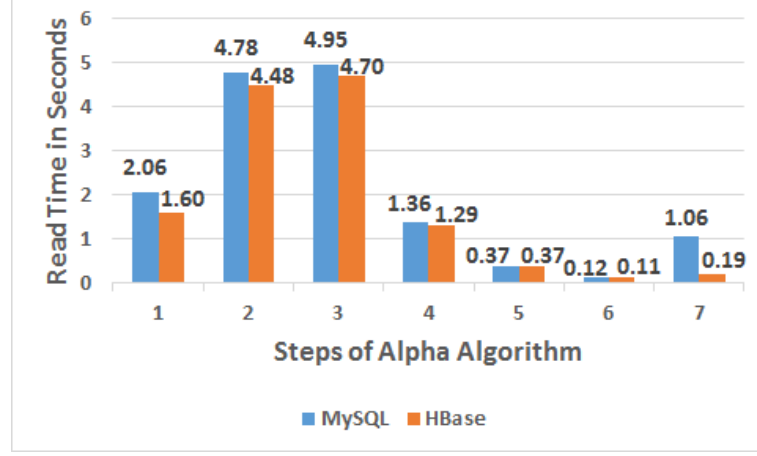


Figure 7.3: Read Intensive Time in Seconds

3. If both the memstore and BlockCache fail, HBase client will locate the target HFiles in HDFS (contains target data) using log structure merge tree and load it into the memory.

The total time taken to read the data in each of the Step of α -miner algorithm is 1.16 times lower in HBase as compared to MySQL.

7.4 Write Intensive Steps of α -Miner Algorithm

Table 7.4: Write Intensive Time

Steps	Write Time in Seconds	
	MySQL	HBase
1	2.12	1.29
2	1.50	1.33
3	1.76	1.04
4	2.73	2.59
5	7.85	5.27
6	1.92	0.89
7	8.12	2.90

We conduct an experiment to investigate which of the Steps of α -miner algorithm are write intensive as well as we compare which of the database performs better for write operations. Fig. 7.4 depicts Step 5 and Step 7 are more write intensive as compared to the other Steps of α -miner algorithm. Fig. 7.4 and Table 7.4 show that the write performance of HBase is better as compared to MySQL. We believe the reason for HBase giving better write performance can be the difference in the way writes are performed in both the databases. In MySQL, the B-Tree index needs to be scanned

7.4 Write Intensive Steps of α -Miner Algorithm

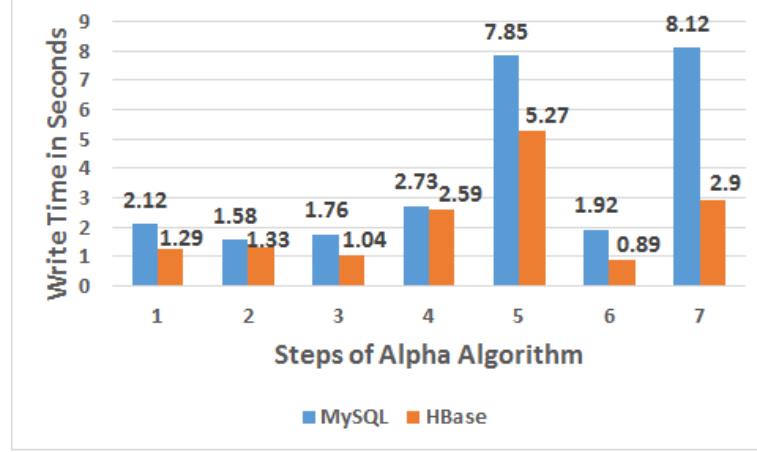


Figure 7.4: Write Intensive Time in Seconds

to find the location of block where the data needs to be written. Almost all the leaf blocks of B-Tree are stored on the disk. Hence, at least one I/O operation is required to retrieve the target block in memory. Fig. 7.4 illustrates that Step 5 and Step 7 of α -miner algorithm in MySQL are more write intensive than the other steps. We believe the reason can be the generation of maximal sets and places by stored procedures in MySQL. A large number of insert operations are executed in the stored procedure to generate the maximal sets. In HBase we perform the same steps using Java because SQL interface over HBase does not support advanced features of SQL. Writes in HBase are performed by first locating regionserver from zookeeper¹, then regionserver writes to WAL and finally to memstore of the corresponding region. Phoenix allows to perform parallelism in reading and writing the data on multiple regions of a table stored in HBase regionserver in comparison to sequential reads and writes of MySQL. The total time taken in writing the data in each of the Step of α -miner algorithm is 1.70 times lower in HBase as compared to MySQL. Thus, writes in HBase are more optimized as compared to that in MySQL.

¹<http://www.zookeeper.apache.org>

7.5 Disk Usage of Tables

Table 7.5: Disk Usage of Tables

No. of Tables	Disk Usage in Bytes	
	MySQL	HBase
1	16384	2048
2	16384	1945
3	16384	1945
4	16384	6384
5	16384	3481
6	16384	4505
7	49152	13414

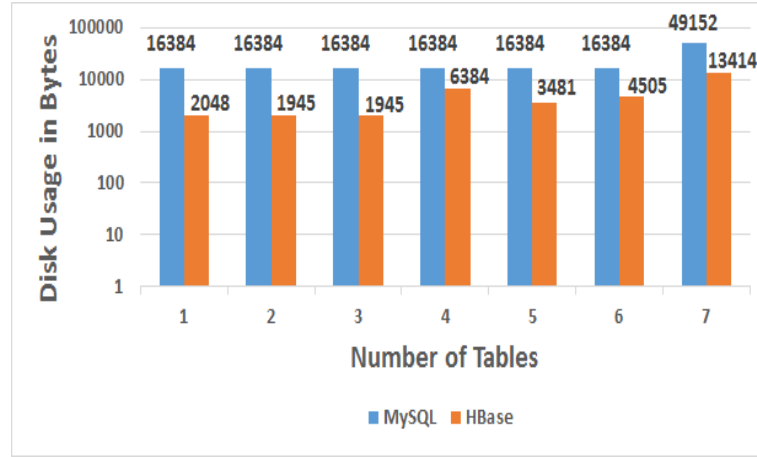


Figure 7.5: Disk Usage of Tables

We perform an experiment to investigate which database can efficiently store results of each Step of α -miner algorithm in tables with minimum disk space. Table 7.5 and Fig. 7.5 reveal the disk space occupied by tables created in each step of α -miner algorithm. We include only data length (excluding the size of index tables) in disk space of table because we did not create index for any of the tables. Experimental results show that HBase on an average uses disk space 6 times lower than MySQL for tables created at each step of the algorithm. Hence, cumulative disk space for storing all the tables in MySQL is 147456 bytes while for HBase is 33722 bytes. We believe the underlying reason for MySQL occupying more space is the difference in the way memory is allocated to tables in both the databases. In MySQL, the operating system allocates fixed size blocks of size 16 KB for the data to be stored in a table. Number of blocks assigned to a table is computed by dividing the dataset size by the block size. In MySQL if set of blocks or one block has been allocated for a table then that set of blocks or block can be used only by that table. Either data in a table completely utilizes the space of all blocks or the space of the last block is unutilized. Storing smaller size file (< 16 KB)

7.6 Disk Usage of Tables Using Compression Technique

in 16 KB block leads to under utilization of space and the remaining space cannot be utilized by other files.

HFile is a file format of HBase which is stored over HDFS block (default size is 64 MB). Maximum size of a HFile is 64 KB after which a new HFile needs to be created. HFiles are created when memstore reaches its threshold value (default value is 64 MB) or commit occurs. When memstore reaches its threshold value it flushes 64 MB data of key-value pairs and creates 1024 numbers of HFiles. If commit occurs before it reaches the threshold value then it flushes only that amount of data present in a memstore. HFile size will be equivalent to flushed amount of data from memstore. HDFS allocates blocks for incoming files by dividing the file size with the block size. For example, we have a system with 300 MB HDFS block size. To store a 1100 MB file, HDFS will break that file into three 300 MB blocks and one 200 MB block size and store it on the datanodes. The 200 MB file is not exactly divisible by 300. Therefore, the final block of the file is sized as modulo of the file size by block size, i.e a 200 MB block size. Similarly, the same process is applied to the HFiles of HBase for storing in HDFS. We conclude that the disk space for each table created in each step is more efficiently utilized in HBase as compared to MySQL.

7.6 Disk Usage of Tables Using Compression Technique

Table 7.6: Disk Usage of Tables
With Compression

No. of Tables	Disk Usage in Bytes	
	MySQL	HBase
1	8192	1536
2	8192	1433
3	8192	1433
4	8192	2355
5	8192	1843
6	8192	1945
7	8192	3584

A way to utilize disk space efficiently is by using the well known compression technique. Data compression enables smaller database size, reduced I/O and improved throughput. We conduct an experiment to compute the disk space occupied by tables at each Step of the α -miner algorithm using compression technique. When we compare the disk space occupied by each table without compression and with compression technique we observe that the compression ratio (Actual size of table/Compressed size of table) is better in MySQL as compared to HBase. As can be seen from Table 7.5 and Table 7.6, the compression ratio in MySQL for Step 7 is equal to 6 (49152/8192) while the compression ratio in HBase for Step 7 is equal to 3.7. Minimum and maximum compression ratio in HBase is 1.3 and 3.7 respectively while in MySQL is 2 and 6 respectively. We believe the reason for MySQL having a higher compression ratio can be the difference in the compression techniques used by both the databases. MySQL uses the zlib compression

7.7 Execution of α -Miner Algorithm Using Compression Technique

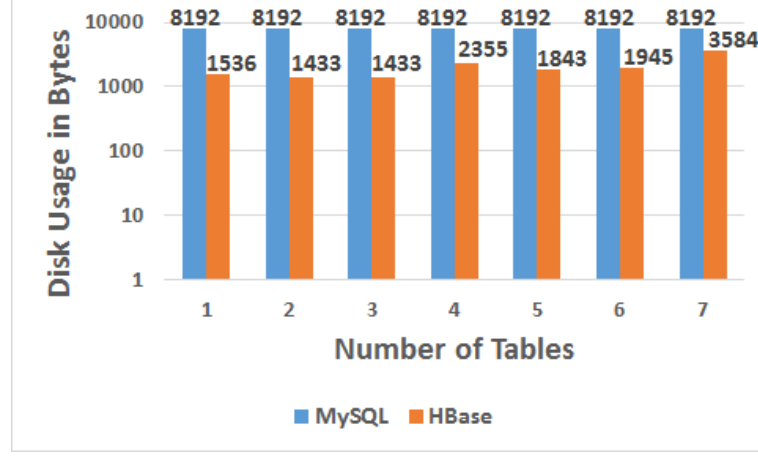


Figure 7.6: Disk Usage of Tables With Compression

technique which provides a better compaction using only six bytes of header and trailer of compressed block. HBase uses gzip compression technique and gzip wrapper uses a minimum of eighteen bytes of header and trailer for compressed block. The maximum compression ratio provided by MySQL is 2 times more as compared to HBase. In the context of α -miner algorithm, MySQL performs better than HBase in utilizing the disk space when compression technique is applied.

7.7 Execution of α -Miner Algorithm Using Compression Technique

Table 7.7: Stepwise Execution Time with Compression

Steps	Execution Time in Seconds	
	MySQL	HBase
1	9.95	3.02
2	12.96	6.87
3	12.35	6.92
4	5.15	4.12
5	9.82	6.04
6	2.62	2.01
7	12.42	3.43

We conduct an experiment to examine the time taken by each Step of α -miner algorithm with compression technique. In α -miner algorithm we create tables in each Step with the compression keyword. Table 7.7 and Fig. 7.7 illustrate that the performance of HBase is better as compared to that of MySQL for each Step of α -miner algorithm. We believe the reason for HBase giving better step wise execution time, with compression

7.7 Execution of α -Miner Algorithm Using Compression Technique

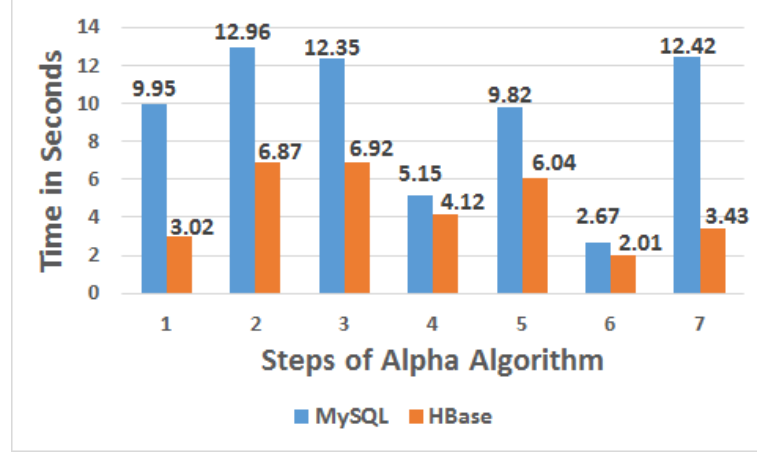


Figure 7.7: α -miner Stepwise Execution Time with Compression

enabled can be the difference in the way compression is performed in both the databases. MySQL uses a block size of 1 KB, 2 KB, 4 KB, 8 KB, 16 KB. The default block size after compression in MySQL is 8 KB. Suppose the size of the compressed block is 5 KB. The block will then be uncompressed, split into two blocks and then recompressed into blocks of size 4 KB and 1 KB. All the data in a table is stored in blocks comprising a B-Tree index. The compression of B-Tree blocks is handled differently because they are frequently updated. It is important to minimize the number of times B-Tree blocks are split, uncompressed and recompressed. MySQL maintains system information in B-Tree block in uncompressed form for certain in-place updates. MySQL avoids unnecessary uncompression and recompression of blocks when they are changed because it causes latency and degrades the performance. HBase does not have fixed block size constraint after compressing the block. We conjecture that another reason for HBase giving a better stepwise execution time, with compression enabled can be the difference in the internal architecture of both the databases that was explained in experiment (Refer to Table 7.2 and Fig. 7.2). From Table 7.2 and Table 7.7, we infer that the total execution time of α -miner algorithm in MySQL is 2 times more as compared to HBase using compression technique. We compare the total time taken in executing α -miner algorithm without compression and with compression technique in MySQL and HBase. We observe that total time taken in executing α -miner algorithm by HBase without compression technique is 1.33 times lower than HBase with compression technique. Similarly, MySQL without compression technique is 1.60 times lower than MySQL with compression technique.

7.8 Real Time Insertion of an Event Logs

Table 7.8: Batch wise Insertion Time

Batch Size for 500 Thousand Records	Batch wise Insertion Time in seconds	
	MySQL	HBase
30,000	522	25
60,000	529	28
90,000	523	30
1,30,000	527	32
2,00,000	519	32
2,50,000	527	32
5,00,000	527	34

Table 7.9: Number of Inserts per Second in Batch

Batch wise for 500 Thousand Records	Number of Inserts per Second	
	MySQL	HBase
30,000	957	19614
60,000	944	17498
90,000	955	15340
1,30,000	947	15134
2,00,000	962	15090
2,50,000	948	15065
5,00,000	947	14613

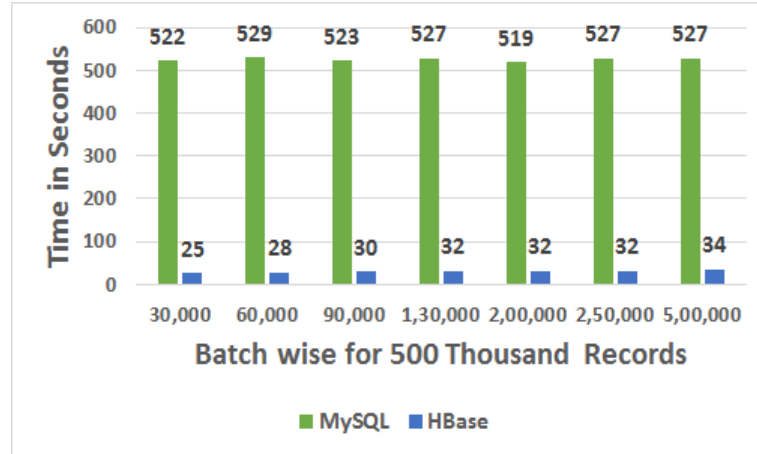


Figure 7.8: Batch wise Insertion Time in Seconds

In all the experiments described above the event logs generated from business processes is stored in a CSV file and then loaded in the database. In the context of Process Mining, PAIS are getting continuously updated with event logs. We setup our experiment to import the event logs directly into the database server from a client application, that is real time data (event logs) loading. The real time loading experiment can be conducted in two ways *viz.* batch insertion and single row insertion. In the batch insertion, the client application inserts 5,00,000 records in different batch sizes. The results of batch insertion are shown in Fig. 7.8 and Table 7.8. We believe that batch insertion might be faster than single record insertion because when we execute a batch, then multiple records in a batch are inserted in a table in a single round trip.

Within the batch insertion experiment we find the number of inserts per second for different batch sizes. We calculated inserts per second by dividing total inserts with the total time taken in seconds. Fig. 7.9 illustrates that the number of inserts per second

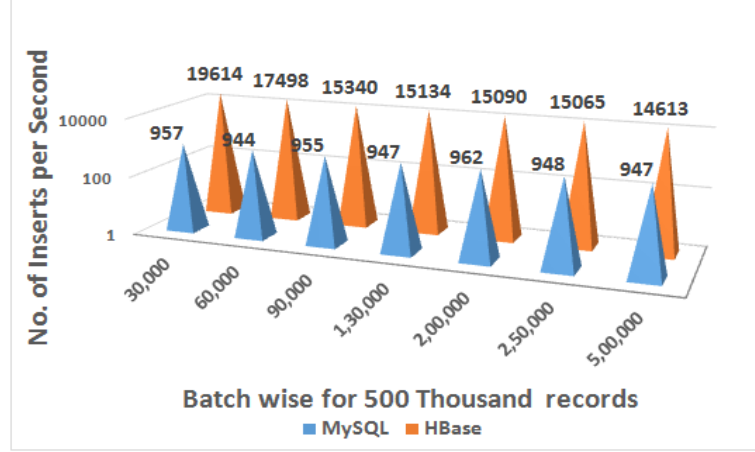


Figure 7.9: Number of Inserts per Second in Batch

decreases as batch size increases in HBase while in MySQL it remains constant. On an average, number of inserts per second in HBase is 17 times more in comparison to MySQL. The results are shown in Fig. 7.9 and Table 7.9. As can be seen from Fig. 7.8 and Fig. 7.9, the performance of HBase is better as compared to MySQL. For batch insertion MySQL uses InnoDB default buffer size of 8 MB to add batch records in it until buffer reaches a threshold value or commit occurs. On the other hand, HBase stores all its batch records in HBase write client buffer which is configured as 20 MB in HBase configuration file. We perform an experiment with the same configuration. Thus, we change InnoDB buffer size from 8 MB to 20 MB.

Table 7.10: Single Row Insertion Time

Dataset Size	Single Row Insertion Time in Seconds	
	MySQL	HBase
30,000	38	5
60,000	68	8
90,000	95	9
1,30,000	134	10
2,00,000	202	16
2,50,000	255	18
5,00,000	523	39

In HBase there is a lag in persisting the data stored on memstore to disk and it is by default asynchronous. On the other hand, MySQL persists data on disk and it is by default synchronous. To have the same configuration we change the durability of HBase to FSYNC_WAL in HBase configuration file. FSYNC_WAL writes the data to WAL synchronously and forces it to the disk. From the results it can be seen that time taken in HBase is 25 times lower in loading 5,00,000 records with different batch sizes

as compared to MySQL. We believe the reason for this can be the difference in the way records are inserted in MySQL and HBase. In MySQL, executing an insert statement is a five step process. The batched insert statements in a buffer are first sent to the server, then parsed, then values are checked for uniqueness (intent hidden query), then data is inserted in actual table and finally data is inserted in index table. In HBase executing an insert statement is a two step process. The first step is writing the data to WAL then to the memstore and finally to the disk synchronously. Thus, the performance of HBase is better as compared to MySQL for batch insertion.

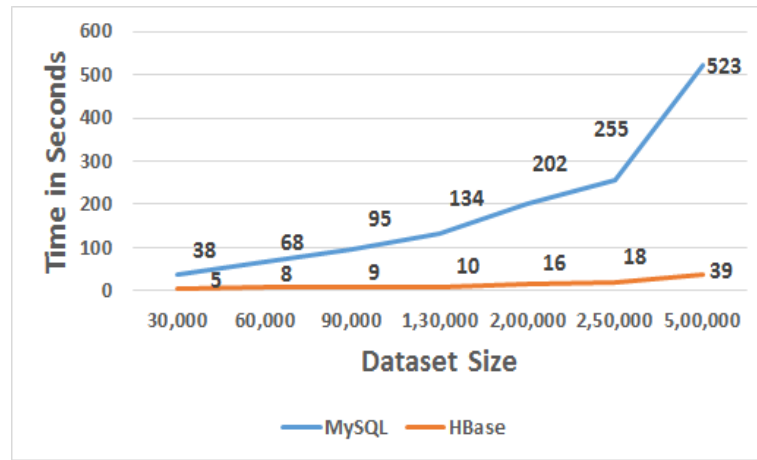


Figure 7.10: Single Row Insertion Time in Seconds

We also conduct a single row insertion experiment to examine which database can perform better for single row insertion. Fig. 7.10 and Table 7.10 reveal that the performance of HBase is better as compared to MySQL for all the datasets. The reason is same as batch insertion but here instead of sending records in a batch we are sending a single record in a single round trip. Fig. 7.10 reveals that when the dataset size increases then the difference between the time taken in loading real time data in MySQL and HBase also increases. We examine that the difference is 14 times lower in HBase as compared to MySQL. Hence, performance of HBase is better as compared to MySQL in loading different datasets with single record insertion.

Conclusion

In this thesis, we present an implementation of α -miner algorithm in MySQL and HBase using structured query language. Furthermore, we also compare the performance of α -miner algorithm in MySQL and HBase. The α -miner implementation in MySQL is a one tier application which uses only standard SQL queries and advanced stored procedures. Similarly, implementation in HBase is done using Phoenix. Experimental results reveal that HBase performs better than MySQL in terms of bulk loading large datasets. We conclude that HBase on an average is 29 times faster than MySQL in loading large datasets and also performs better in loading real time data through client application. We do a real time data (event logs) loading experiment to examine which one of the database can efficiently load the event logs from various sources of PAIS.

Our experimental results shows that HBase outperforms MySQL in loading real time data (event logs) by having 17 times more number of inserts per second. Furthermore, the total time taken to read the data while execution of α -miner algorithm is 1.16 times lower in HBase as compared to MySQL. Similarly, for writing the data, time taken by HBase is 1.70 times lower as compared to MySQL. The total execution time of α -miner algorithm improves significantly in HBase as compared to MySQL. HBase outperforms MySQL in terms of the disk usage of tables. The disk space occupied by tables in HBase is 4.37 times lower as compared to MySQL. Thus, we conclude that HBase is more efficient than MySQL in terms of storing data and performing query. Using a well known compression technique, HBase outperforms MySQL in disk usage as well as execution of α -miner algorithm.

Limitations and Future Work

We have used the BPI challenge 2014 dataset, consisting of only 466738 records requiring 50 MB of storage. According to the Big Data standards, this is not a large dataset. Thus, our future work includes applying the algorithm over multiple and larger datasets (PetaBytes and ExaBytes). We are currently using a pseudo distributed mode of HBase for performance benchmarking and comparison. In pseudo-distributed mode, Hadoop processing is distributed over all of the cores/processors on a single machine. Hadoop writes all files to the Hadoop Distributed File System (HDFS), and all services and daemons communicate over local TCP sockets for inter-process communication. Future work includes creating a multi-node cluster and implement the algorithm on it.

We have currently implemented only one algorithm called the α -miner algorithm. Future work includes implementing more complex and advanced process mining algorithms like α^+ -miner algorithm, heuristic miner algorithm and genetic algorithm. SQL interface over HBase does not support advanced features of SQL such as CONCAT_WS. Hence, we need a client application to perform advanced functions of SQL. Support of SQL over HBase with advanced functions can lead to better performance of HBase as the application will then be tightly coupled to the database.

We perform all our experiments on MySQL and HBase only. Future work includes experimenting with more relational databases like PostgreSQL and more column oriented databases like Cassandra, Vertica, Teradata, and MonetDB so that we can come to a firm conclusion as which database is better for implementing α -miner algorithm. Future work also includes experimenting with NewSQL database systems like NuoDB, voltDB, SpliceMachine and MemSQL.

Appendix A

Implementation of α -Miner Algorithm in SQL on Row-Oriented Database (MySQL)

1. Create Table for Storing the Event Logs.

```
CREATE TABLE eventlog (caseid VARCHAR(200),timestamp TIMESTAMP,status  
                        VARCHAR(200),activity VARCHAR(200),PRIMARY KEY(caseid,timestamp,status)) ;
```

2. Load Data from the CSV File into Table.

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

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

```
CREATE TABLE totalEvents(events VARCHAR(200) PRIMARY KEY) ;  
  
INSERT INTO totalEvents(events)  
(SELECT DISTINCT eventlog.activity FROM eventlog);
```

The above query states that: The Insert statement inserts all rows returned by first inner subquery **Select statement** into the table named totalEvents. Inner subquery will return all distinct activities from table eventlog. **First step of α -miner algorithm** is completed.

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

```
CREATE TABLE initialEvents(InitialEvents VARCHAR(200)) ;  
INSERT INTO initialEvents(InitialEvents)  
SELECT DISTINCT A.activity  
FROM eventlog AS A  
WHERE A.timestamp IN
```

```
(SELECT MIN(derived.timestamp) AS timestamp
FROM eventlog AS derived
GROUP BY derived.caseid);
```

The above query states that: The **Insert statement** inserts all rows returned by first outer subquery **Select statement** into the table named **initialEvents**. The outer subquery **Select statement** selects distinct activities from temporary table returned by inner subquery by comparing their timestamp with inner subquery minimum timestamp using **IN** operator and the inner subquery Select statement works on eventlog table by considering all caseid with their minimum timestamp using Group By. **Second step of α -miner algorithm** is completed.

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

```
CREATE TABLE finalEvents(FinalEvents VARCHAR(200) PRIMAR KEY) ;
INSERT INTO finalEvents(FinalEvents)
SELECT DISTINCT A.activity
FROM eventlog AS A
WHERE A.timestamp IN
(SELECT MAX(derived.timestamp) AS datetime
FROM eventlog AS derived
GROUP BY derived.caseid);
```

The above query states that: The **Insert statement** inserts all rows returned by first outer subquery **Select statement** into the table named **finalEvents**. The outer subquery **Select statement** selects distinct activities 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.

Third step of α -miner algorithm is completed.

6. **Create table SafeEventA**

```
CREATE TABLE SafeEventA(setA VARCHAR(200), setB VARCHAR(200)) ;
```

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($\#$).

7. **Create table SafeEventB**

```
CREATE TABLE SafeEventB(setA VARCHAR(200), setB VARCHAR(200)) ;
```

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 be-

tween 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(#).

8. **Create Table EventA.**

```
CREATE TABLE eventA (setA VARCHAR(200), setB VARCHAR(200)) ;
```

This table contains two columns **setA** and **setB**. The **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.

9. **Create table eventB.**

```
CREATE TABLE eventB(setA VARCHAR(200), setB VARCHAR(200)) ;
```

This table contains two columns **setA** and **setB**. The **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.

10. **Create table X_W .**

```
CREATE TABLE xw (setA VARCHAR(200),setB VARCHAR(200)) ;
```

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 above query states that: 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**.

Algorithm 1: Populating table SafeEventA and SafeEventB

```
1 Select setA, setB from eventA .
2 Select setA, setB from eventB .
3 Compare Function that compares whether set of activity is NotConnected.
4 foreach setB in the eventA do
5   | Check whether all combination of activity from setB can occur in
   | NotConnected table. If any combination return true then Insert setA and
   | returned combination into column setA and setB of table SafeEventA.
6 end
7 foreach setA in the eventB do
8   | Check whether all combination of activity from setA can occur in
   | NotConnected table. If any combination return true then Insert returned
   | combination and setB into column setA and setB of table SafeEventB.
9 end
```

13. Populate Table xw.

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

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

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

Fourth step of α -miner algorithm is completed.

14. CALL stored procedure explode_tableYW1.

```
18 DELIMITER $$
19 DROP PROCEDURE IF EXISTS explode_tableYW1 $$
20 CREATE PROCEDURE explode_tableYW1 (bound VARCHAR(255))
21 BEGIN
22   DECLARE A text ;
23   DECLARE B TEXT;
24   DECLARE occurance INT DEFAULT 0;
25   DECLARE i INT DEFAULT 0;
26   DECLARE splitted_value text;
27   DECLARE done INT DEFAULT 0;
28   DECLARE cur1 CURSOR FOR SELECT SafeEventB.setA, SafeEventB.setB FROM SafeEventB;
29   DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
30   DROP TEMPORARY TABLE IF EXISTS safeB;
31   CREATE TEMPORARY TABLE safeB('setA' VARCHAR(255) NOT NULL, 'setB' VARCHAR(255) NOT
    NULL);
32   OPEN cur1;
33   read_loop: LOOP
34     FETCH cur1 INTO A, B;
35     IF done THEN
36       LEAVE read_loop;
```

```

37     END IF;
38     SET occurrence = (SELECT LENGTH(A)- LENGTH(REPLACE(A, bound, ','))+1);
39     SET i=1;
40     WHILE i<= occurrence DO
41         SET splitted_value =
42         (SELECT REPLACE(SUBSTRING(SUBSTRING_INDEX(A, bound, i),
43         LENGTH(SUBSTRING_INDEX(A, bound, i - 1)) + 1), ',', ''));
44         INSERT INTO safeB VALUES (splitted_value,B);
45         SET i = i + 1;
46     END WHILE;
47 END LOOP;
48 CLOSE cur1;
49 END; $$

```

The above stored procedure splits the column **setA** of table **SafeEventB** on comma separator.

15. CALL stored procedure explode_tableYW.

```

50 DELIMITER $$
51 DROP PROCEDURE IF EXISTS explode_tableYW $$
52 CREATE PROCEDURE explode_tableYW (bound VARCHAR(255))
53 BEGIN
54     DECLARE A text ;
55     DECLARE B TEXT;
56     DECLARE occurrence INT DEFAULT 0;
57     DECLARE i INT DEFAULT 0;
58     DECLARE splitted_value text;
59     DECLARE done INT DEFAULT 0;
60     DECLARE cur1 CURSOR FOR SELECT SafeEventA.setA, SafeEventA.setB FROM SafeEventA ;
61     DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
62     DROP TEMPORARY TABLE IF EXISTS safeA;
63     CREATE TEMPORARY TABLE safeA( 'setA' VARCHAR(255) NOT NULL, 'setB' VARCHAR(255) NOT
        NULL);
64 OPEN cur1;
65 read_loop: LOOP
66     FETCH cur1 INTO A, B;
67     IF done THEN
68         LEAVE read_loop;
69     END IF;
70     SET occurrence = (SELECT LENGTH(B)- LENGTH(REPLACE(B, bound, ','))+1);
71     SET i=1;
72     WHILE i<= occurrence DO
73         SET splitted_value =
74         (SELECT REPLACE(SUBSTRING(SUBSTRING_INDEX(B, bound, i),
75         LENGTH(SUBSTRING_INDEX(B, bound, i - 1)) + 1), ',', ''));
76         INSERT INTO safeA VALUES (A,splitted_value);
77         SET i = i + 1;
78     END WHILE;
79 END LOOP;
80 CLOSE cur1;
81 END; $$

```

The above stored procedure splits the column **setB** of table **SafeEventA** on comma separator.

16. Create and populate table eventASafe.

```
CREATE TABLE eventASafe(setA VARCHAR(200),setB VARCHAR(200)) ;
INSERT INTO eventASafe (setA,setB)
SELECT setA,setB from safeA;
```

The above query states that: 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 stored procedure **explode_tableYW**.

17. Create and populate table eventB Safe.

```
CREATE TABLE eventB Safe(setA VARCHAR(200), setB VARCHAR(200)) ;
INSERT INTO eventB Safe (setA,setB)
SELECT setA,setB from safeB;
```

The above query states that: The **Insert statement** inserts all rows of **safeB** table in **eventB Safe** table. The **Select statement** selects all rows of **safeB** table that was created by stored procedure **explode_tableYW1**.

18. Create and populate table Y_W .

```
CREATE TABLE yw(setA VARCHAR(200), setB VARCHAR(200), PRIMARY KEY(setA,setB)) ;
CREATE TABLE temporary_table(setA VARCHAR(200), setB VARCHAR(200), PRIMARY
KEY(setA,setB)) ;

INSERT INTO temporary_table(setA, setB)
SELECT setA, setB FROM eventB Safe;

INSERT IGNORE INTO temporary_table(setA, setB)
SELECT setA, setB FROM eventASafe;

INSERT IGNORE INTO yw (setA, setB)
SELECT eventA, eventB
FROM causality
WHERE eventB NOT IN(SELECT distinct setB FROM temporary_table)
AND eventA NOT IN (SELECT distinct setA FROM temporary_table);

INSERT INTO yw (setA, setB)
SELECT setA, setB
FROM SafeEventA;

INSERT INTO yw (setA, setB)
SELECT setA, setB
FROM SafeEventB;
```

Fifth step of α -miner algorithm is completed.

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

```
CREATE TABLE terminalPlaces(event VARCHAR(200)) ;
INSERT INTO terminalPlaces Values ('i');
INSERT INTO terminalPlaces Values ('o');
```

20. Create and populate table P_W .

```

CREATE TABLE pw(setA VARCHAR(200)) ;
INSERT INTO pw(setA)
    SELECT CONCAT_WS(' & ',setA,SetB) AS place From yw;
INSERT INTO pw(setA)
    SELECT event FROM terminalPlaces;

```

Sixth step of α -miner algorithm is completed.

21. 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;

```

22. Call stored procedure explode_table_for_place2.

```

126 DELIMITER $$
127 DROP PROCEDURE IF EXISTS explode_table_for_place2 $$
128 CREATE PROCEDURE explode_table_for_place2(bound VARCHAR(255))
129 BEGIN
130     DECLARE A text ;
131     DECLARE B TEXT;
132     DECLARE occurrence INT DEFAULT 0;
133     DECLARE i INT DEFAULT 0;
134     DECLARE splitted_value text;
135     DECLARE done INT DEFAULT 0;
136     DECLARE cur1 CURSOR FOR SELECT place2.id, place2.value FROM place2 ;
137     DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
138     DROP TABLE IF EXISTS temp_place2;
139     CREATE TABLE temp_place2(
140         'id' VARCHAR(255) NOT NULL,
141         'value' VARCHAR(255) NOT NULL );
142     OPEN cur1;
143     read_loop: LOOP
144         FETCH cur1 INTO A, B;
145         IF done THEN
146             LEAVE read_loop;
147         END IF;
148         SET occurrence = (SELECT LENGTH(B)- LENGTH(REPLACE(B, bound, '')) +1);
149         SET i=1;
150         WHILE i<= occurrence DO
151             SET splitted_value =
152             (SELECT REPLACE(SUBSTRING(SUBSTRING_INDEX(B, bound, i),
153             LENGTH(SUBSTRING_INDEX(B, bound, i - 1)) + 1), ',', ''));
154             INSERT INTO temp_place2 VALUES (A,splitted_value);
155             SET i = i + 1;
156         END WHILE;
157     END LOOP;
158     CLOSE cur1;
159 END; $$

```

23. CALL stored procedure explode_table_for_place1.

```

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

```

24. 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';

```

Seventh step of α -miner algorithm is completed.

Appendix B

Implementation of α -Miner Algorithm on NoSQL Column-Oriented Database (HBase) Using Apache Phoenix

1. Create Table for Storing the Event Logs.

```
CREATE TABLE EventLog (caseid VARCHAR(200) not null, timestamp TIMESTAMP not null, status  
    VARCHAR(200),activity VARCHAR(200),stlevel VARCHAR(200),area VARCHAR(200),actor  
    VARCHAR(200),CONSTRAINT pk PRIMARY KEY(caseid,timestamp,status)) ;
```

2. Load Data from the CSV File into Table by help of MapReduce code.

```
HADOOP_CLASSPATH=/path/to/hbase-protocol.jar:/path/to/hbase/conf hadoop jar  
    phoenix-4.2.1-incubating-client.jar org.apache.phoenix.mapreduce.CsvBulkLoadTool  
    --table eventlog --input eventlog.csv
```

The above command do mapreduce job by help of jobtracker and tasktraker. Jobtracker assigns job to tasktraker and tasktacker execute job by help of mappers and reducers.

3. Create and populate table TotalEvents.

```
CREATE TABLE TotalEvents(event VARCHAR(200) not null PRIMARY KEY) ;  
UPSERT INTO TotalEvents(event)  
    SELECT DISTINCT activity FROM EventLog;
```

The above query states that: First of all create table **TotalEvents** and then insert all **distinct activities** of table **EventLog**. **First step of α -miner algorithm** is completed.

4. Create and populate table InitialEvents.

```
CREATE TABLE InitialEvents(initial VARCHAR(200) not null PRIMARY KEY) ;
UPSERT INTO InitialEvents(initial)
SELECT DISTINCT E1.activity
FROM EventLog AS E1
WHERE E1.timestamp IN(SELECT MIN(E2.timestamp)
FROM EventLog AS E2 GROUP BY E2.caseid);
```

The above query states: The **UPSERT statement** inserts all rows returned by first outer subquery **Select statement** into the table named **InitialEvents**. The outer subquery **Select statement** selects distinct activities from temporary table returned by inner subquery by comparing their timestamp with inner subquery minimum timestamp using **IN** operator and the inner subquery **Select statement** works on **EventLog** table by considering all caseid with their minimum timestamp using **Group By**. **Second step of α -miner algorithm** is completed.

5. Create and populate table FinalEvents.

```
CREATE TABLE FinalEvents(final VARCHAR(200) not null PRIMARY KEY);
UPSERT INTO FinalEvents(final)
SELECT DISTINCT E1.activity
FROM EventLog AS E1
WHERE E1.timestamp IN(SELECT MAX(E2.timestamp)
FROM EventLog AS E2 GROUP BY E2.caseid);
```

The above query states: The **UPSERT statement** inserts all rows returned by first outer subquery **Select statement** into the table named **FinalEvents**. The outer subquery **Select statement** selects distinct activities 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**.

Third step of α -miner algorithm is completed.

6. Create table SafeEventA

```
CREATE TABLE SafeEventA(setA VARCHAR(200),setB VARCHAR(200),CONSTRAINT PK PRIMARY
KEY(setA,setB)) ;
```

7. Create table SafeEventB

```
CREATE TABLE SafeEventB(setA VARCHAR(200),setB VARCHAR(200),CONSTRAINT PK PRIMARY
KEY(setA,setB)) ;
```

Algorithm 2: Populating table SafeEventA and SafeEventB

```
1 Select eventA, eventB from CAUSALITY.
2 Select setA, setB from NotConnected.
3 Compare Function that compares whether set of activity is NotConnected.
4 foreach eventA in the CAUSALITY do
5   | Form single group say grp of all activity present in column eventB. Pass grp
   | to Compare function. For any such combination returning true, insert
   | eventA in setA and that combination into setB of table SafeEventA.
6 end
7 foreach eventB in the CAUSALITY do
8   | Form single group say grp of all activity present in column eventA. Pass grp
   | to Compare function. For any such combination returning true, insert that
   | combination into setA and eventB in setB of table SafeEventA.
9 end
```

8. Create and populate table XW.

```
CREATE TABLE XW(setA VARCHAR(200), setB VARCHAR(200), CONSTRAINT pk PRIMARY
KEY(setA,setB)) ;
UPSERT INTO XW(setA, setB)
SELECT setA, setB FROM SafeEventA;
UPSERT INTO XW(setA, setB)
SELECT setB, setA FROM SafeEventB;
UPSERT INTO XW(setA, setB)
SELECT eventA, eventB FROM CAUSALITY;
```

The above query states: **Upsert** statement will insert all rows from table **SafeEventA**, **SafeEventB**, **CAUSALITY**.

Fourth step of α -miner algorithm is completed.

9. Create table EventA

```
CREATE TABLE EventA(setA VARCHAR(200),setB VARCHAR(200),CONSTRAINT PK PRIMARY
KEY(setA,setB)) ;
```

10. Create table EventB

```
CREATE TABLE EventB(setA VARCHAR(200),setB VARCHAR(200),CONSTRAINT PK PRIMARY
KEY(setA,setB)) ;
```

Algorithm 3: Populate table EventA and EventB

```
1 Select setA, setB from SafeEventA.
2 Select setA, setB from SafeEventB.
3 foreach setA,setB in the SafeEventA do
4   | Delimit value of setB. For all such value setBi, insert setA and setBi in table
   | EventA.
5 end
6 foreach setA,setB in the SafeEventB do
7   | Delimit value of setA. For all such value setAi, insert setAi and setB in table
   | EventB.
8 end
```

11. Create and populate table YW.

```
CREATE TABLE YW (setA VARCHAR(200),setB VARCHAR(200),CONSTRAINT pk PRIMARY
KEY(setA,setB)) ;
CREATE TABLE TemporaryTable(setA VARCHAR(200),setB VARCHAR(200),CONSTRAINT pk PRIMARY
KEY(setA,setB));
UPSERT INTO TemporaryTable(setA,setB)
SELECT setA,setB FROM EventB;
UPSERT INTO TemporaryTable(setA,setB)
SELECT setA,setB FROM EventA;
UPSERT INTO YW (setA,setB)
SELECT eventA ,eventB
FROM CAUSALITY
WHERE eventB NOT IN
(SELECT DISTINCT setB FROM TemporaryTable)
AND eventA NOT IN
(SELECT DISTINCT setA FROM TemporaryTable);
UPSERT INTO YW(setA,setB)
SELECT setA, setB FROM SafeEventA;
UPSERT INTO YW(setB,setA)
SELECT setA,setB FROM SafeEventB;
```

The above query states: All rows from table **EventA** and **EventB** will store in **TemporaryTable** and TemporaryTable will be used to remove all events that present in TemporaryTable from CAUSALITY table. At last all rows from CAUSALITY will store in YW. **Fifth step of α -miner algorithm** is completed .

12. Create table PW

```
CREATE TABLE PW(Place VARCHAR(255),CONSTRAINT PK PRIMARY KEY(Place)) ;
```

Algorithm 4: Populating Table PW

```
1 Select setA, setB from YW.
2 foreach setA,setB in the YW do
3   | Combine value of setA column and setB column together and insert it into
   | column Place of PW table
4 end
```

13. **Populate table PW.**

```
CREATE TABLE s (event VARCHAR(200) not null PRIMARY KEY) ;
UPSERT INTO s VALUES('O');
UPSERT INTO s VALUES('I');
UPSERT INTO PW
SELECT event FROM s;
```

Sixth step of α -miner algorithm is completed.

14. **Create and populate table FW.**

```
CREATE TABLE FW(Place1 VARCHAR(200),Place2 VARCHAR(200),CONSTRAINT PK PRIMARY
KEY(Place1,Place2)) ;
```

Algorithm 5: Populating Table FW

```
1 Select setA, setB from YW.
2 Select final from FinalEvents.
3 Select initial from InitialEvents.
4 foreach final in the FinalEvents do
5   | Insert final in column Place1 and 'o' in column Place2 of table FW
6 end
7 foreach initial in the InitialEvents do
8   | Insert 'i' in column Place1 and initial in column Place2 of table FW
9 end
10 foreach setA,setB in the YW do
11   | If value of column setA has set of activities instead of single activity then
      |   delimit. Each split value will store in column Place1 and
      |   combination of setA and setB in column Place2 of table FW
12   | else choose column setB and delimit. Each split value will store in
      |   column Place2 and combination of setA and setB in column Place1
      |   of table FW
13 end
```

The above query states: All rows in form of places will store in table **FW** by help of algorithm. **Seventh step of α -miner algorithm** is completed.

References

- [1] LARS GEORGE. **HBase The Definitive Guide**. vi, 14
- [2] NICHOLAS CHARLES RUSSELL. **Foundation of Process-Aware Information Systems**. 1
- [3] WIL VAN DER AALST. **Process Mining: Overview and Opportunities**. *ACM*, 2012. 1, 3
- [4] SAWITREE WEERAPONG, PARHAM POROUHAN, AND WICHIAN PREMCHAIWADI. **Process Mining Using α -Algorithm as a Tool**. *IEEE*, 2012. 1, 3, 4, 16
- [5] HASSO PLATTNER. **A common database approach for OLTP and OLAP using an in-memory column database**. *ACM SIGMOD International Conference on Management of data*, 2009. 2, 10
- [6] MARY A. FINN. **FIGHTING IMPEDANCE MISMATCH AT THE DATABASE LEVEL**. 2
- [7] VATIKA SHARMA AND MEENU DAVE. **SQL and NoSQL Database**. *International Journal of Advanced Research in Computer Science and Software Engineering*, 2012. 2, 3
- [8] DANIEL J. ABADI, SAMUEL R. MADDEN, AND NABIL HACHEM. **Column-Stores vs. Row-Stores: How Different Are They Really?** *SIGMOID*, 2008. 2
- [9] K-U. SATTLER AND O. DUNEMANN. **SQL Database Primitives for Decision Tree Classifiers**. *Conference on Information and Knowledge Management*, pages 379–386, 2001. 2, 9
- [10] RAJA APPUSWAMY, CHRISTOS GKANTSIDIS, DUSHYANTH NARAYANAN, ORION HODSON, AND ANTONY ROWSTRON. **Scale-up vs scale-out for Hadoop: time to rethink?** 2013. 3
- [11] S. R. MADDEN D. J. ABADI AND M. FERREIRA. **Integrating compression and execution in column-oriented database systems**. *SIGMOD*. 6

-
- [12] ASTHA SACHDEV. **Khanan: Performance Comparison and Programming Alpha Algorithm in Column-Oriented and Relational Database Query Languages.** 2015. 9, 20
 - [13] C.ORDONEZ AND P.CEREGHINI. **SQLEM: fast clustering in SQL using the EM algorithm.** *International Conference on Management of Data*, pages 559–570, 2000. 9
 - [14] XUEQUN SHANG, KAI UWE SATTLER, AND INGOLF GEIST. **Efficient Frequent Pattern Mining in Relational Databases.** 2004. 9
 - [15] CARLOS ORDONEZ. **Programming the K-means clustering algorithm in SQL.** (6):823–828, 2004. 10
 - [16] 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. 10
 - [17] L.SURESH, J.B SIMHA, 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. 10
 - [18] CIPRIAN PUNGILA, TEODOR FLORIN FORTIS, AND OVIDIU ARITONI. **Benchmarking Database Systems for the Requirements of Sensor Readings.** pages 1–5, 2009. 10
 - [19] 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. 10
 - [20] 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. 10
 - [21] ANDREAS LÜBCKE AND GUNTER SAAKE. **Workload Representation across Different Storage Architectures for Relational DBMS.** 2012. 10
 - [22] K. SHVACHKO, HAIRONG KUANG, AND AND R. CHANSLER S. RADIA. **The Hadoop Distributed File System.** *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, 2010. 12
 - [23] SANJAY GHEMAWAT WILSON C. HSIEH DEBORAH A. WALLACH MIKE BURROWS TUSHAR CHANDRA ANDREW FIKES FAY CHANG, JEFFREY DEAN AND ROBERT E. GRUBE. **Bigtable: A Distributed Storage System for Structured Data.** *OSDI'06: Seventh Symposium on Operating System Design and Implementation*. 13

-
- [24] NICK DIMIDUK AND AMANDEEP KHURANA. **HBase In Action**. 34
- [25] 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.
- [26] M.N. VORA. **Hadoop-HBase for large-scale data**. *Computer Science and Network Technology (ICCSNT), 2011 International Conference on*, 2011.
- [27] D. CARSTOIU, A. CERNIAN, AND A. OLTEANU. **Hadoop Hbase-0.20.2 performance evaluation**. *New Trends in Information Science and Service Science (NISS), 2010 4th International Conference on*, 2010.
- [28] CHONGXIN LI. **Transforming relational database into HBase: A case study**. *Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference on*, 2010.
- [29] TYLER HARTER, DHRUBA BORTHAKUR, SIYING DONG, AMITANAND AIYER, LIYIN TANG, ANDREA C. ARPACI-DUSSEAU, AND REMZI H. ARPACI-DUSSEAU. **Analysis of HDFS Under HBase: A Facebook Messages Case Study**. *12th USENIX conference on File and Storage Technologies*, 2014.
- [30] FERNANDO BERZAL, JUAN-CARLOS CUBERO, NICOLAS MARIN, AND JOSE MARIA SERRANO. **TBRAR: An efficient method for association rule mining in relational databases**. *elsevier*, 2001.
- [31] DONALD D. CHAMBERLIN, MORTON M. ASTRAHAN, MICHAEL W. BLASGEN, JAMES N. GRAY, W. FRANK KING, BRUCE G. LINDSAY, RAYMOND LORIE, JAMES W. MEHL, THOMAS G. PRICE, FRANCO PUTZOLU, PATRICIA GRIFFITHS SELINGER, MARIO SCHKOLNICK, DONALD R. SLUTZ, IRVING L. TRAIGER, BRADFORD W. WADE, AND ROBERT A. YOST. **A history and evaluation of System R**. 1973.
- [32] HOWARD GOBIOFF SANJAY GHEMAWAT AND SHUN-TAK LEUNG. **Google File System**. *19th ACM Symposium on Operating Systems Principles*.
- [33] J. ZHOU AND K. A. ROSS. **Buffering databse operations for enhanced instruction cache performance**. *SIGMOD*.
- [34] CHEN ZHANG AND HANS DE STERCK. **HBaseSI: Multi-row Distributed Transactions with Global Strong Snapshot Isolation on Clouds**. *Scientific International Journal for Parallel and Distributed Computing*, 2011.
- [35] RAJA APPUSWAMY, CHRISTOS GKANTSIDIS, DUSHYANTH NARAYANAN, ORION HODSON, AND ANTONY ROWSTRON. **Scale-up vs scale-out for Hadoop: time to rethink?** 2013.