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

Astha Sachdev[1], Kunal Gupta[1], and Ashish Sureka[2](✉)

[1] Indraprastha Institute of Information Technology, Delhi (IIITD), New Delhi, India
http://www.iiitd.ac.in/

[2] Software Analytics Research Lab (SARL), New Delhi, India
ashish@iiitd.ac.in
http://www.software-analytics.in/

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 business processes. The large volume of event logs generated are stored in the databases. Relational databases perform well for a certain class of applications. However, there is a certain class of applications for which relational databases are not able to scale well. To address the challenges of scalability, NoSQL database systems emerged. Discovering a process model (workflow) from event logs is one of the most challenging and important Process Mining tasks. The α-miner algorithm is one of the first and most widely used Process Discovery techniques. Our objective is to investigate which of the databases (Relational or NoSQL) performs 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 application is tightly coupled to the database. We conduct a performance benchmarking and comparison of the α-miner algorithm on row-oriented database and NoSQL column-oriented database. We present the comparison on various aspects like time taken to load large datasets, disk usage, stepwise execution time and compression technique.

Keywords: Apache Cassandra · α-miner algorithm · Column-oriented database · MySQL · Process Mining · Performance comparison · Row oriented database

1 Research Motivation and Aim

A PAIS is an IT system that manages and supports business processes. A PAIS generates data from the execution of business processes. The data generated

N. Kumar and V. Bhatnagar (Eds.): BDA 2015, LNCS 9498, pp. 165–180, 2015.
DOI: 10.1007/978-3-319-27057-9_12

by a PAIS like Enterprise Resource Planing (ERP) and Customer Relationship Management (CRM) [11] is in the form of event logs (represented as a tuple of <CaseID, Timestamp, Activity, Actor>). 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 new and emerging field which consist of analyzing event logs generated from the execution of business process. The insights obtained from event logs helps the organizations to improve their business processes. There are three major techniques within Process Mining *viz.* Process Discovery, Process Conformance and Process Enhancement [14]. The classification of Process Mining techniques is based on whether there is a priori model and how the a priori model is used, if present. In this paper 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 information present in event logs. One of the most fundamental algorithm under Process Discovery is the α-miner algorithm [13] which is used to generate a process model from event logs.

Before the year 2000, majority of the organizations used traditional relational database management systems (RDBMS) to store the data. Most of the traditional relational databases focus on Online Transaction Processing (OLTP) applications [10] but are not able to perform Online Analytical Processing (OLAP) applications efficiently. Row-oriented databases are not well suited for analytical functions (like Dense_Rank, Sum, Count, Rank, Top, First, Last and Average) but work fine when we need to retrieve the entire row or to insert a new record. Recent years have seen the introduction of a number of NoSQL column-oriented database systems [12]. These database systems have shown to perform more than an order of magnitude better than the traditional relational database systems on analytical workloads [3]. 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 [5]. When representation of data in memory and that in database is different, then it is known as impedance matching. This is because in-memory data structures use lists, dictionaries, nested lists while traditional 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.

There is a certain class of applications, like facebook messaging application, for which row-oriented databases are not able to scale. To handle such class of applications, NoSQL database systems were introduced. Process Discovery is a very important application of Process Mining. Our objective is to investigate an approach to implement a Process Discovery 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 database query language is one of the most standard way to interact with the database. Structured Query Language (SQL) has grown over the past few years and has become a fairly complex language. SQL takes care of the various aspects like storage management, concurrent access, memory leaks and fault tolerance and also gives the flexibility of tuning the database [1]. 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 [8]. We aim to implement α-miner algorithm in the specific database query languages so that our Process Discovery application is tightly coupled to the database.

There are various NoSQL column-oriented databases [12] but for our current work, we will focus on Apache Cassandra[1] (NoSQL column-oriented database) and MySQL[2] (row-oriented database).

The research aim of the work presented in this paper is the following:

1. To investigate an approach to implement α-miner algorithm in Structured Query Language (SQL). The underlying row-oriented database for implementation is MySQL using InnoDB[3] engine.
2. To investigate an approach to implement α-miner algorithm in Cassandra Query Language (CQL) on column-oriented database Cassandra.
3. To conduct a series of experiments on a publicly available real world dataset to compare the performance of α-miner algorithm on both the databases. The experiments consider multiple aspects such as α-miner stepwise execution time, bulk loading across various datasets, write intensive time and read intensive time and disk space of tables.

2 Related Work and Novel Contributions

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

2.1 Implementation of Mining Algorithms in Row-Oriented Databases

Ordonez et al. present a method to implement k-means clustering algorithm in SQL. They cluster in large datasets in RDBMS [1]. Their work concentrates on defining suitable tables, indexing them and writing suitable queries for clustering purposes. Ordonez et al. presents an efficient SQL implementation of the EM algorithm to perform clustering in very large databases [2]. Sattler et al.

[1] http://cassandra.apache.org/.
[2] http://www.mysql.com/.
[3] http://dev.mysql.com/doc/refman/5.5/en/innodb-storage-engine.html.

present a study of applying data mining primitives on decision tree classifier [8]. 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.

2.2 Implementation of Mining Algorithms in Column-Oriented Databases

Rana et al. conducted experiments on the utilization of column oriented databases like MonetDB with oracle 11 g which is a row oriented data store for execution time analysis of Association Rule Mining algorithm [4]. Suresh L et al. implemented k-means clustering algorithm on column store databases [9].

2.3 Performance Comparison of Mining Algorithms in Column-Oriented and Graph Databases

Gupta et al. conduct a performance comparison and programming alpha-miner algorithm in relational database query language and NoSQL column-oriented using Apache Phoenix [6]. Joishi et al. [7] presents a performance comparison and implementation of process mining algorithms (organizational perspective) in graph-oriented and relational database query languages.

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

1. While there has been work done on implementing data mining algorithms in row-oriented databases, we are the first to implement Process Mining α-miner algorithm in row-oriented database MySQL using Structured Query Language(SQL).
2. While data mining algorithms have been implemented in column-oriented databases, we are the first to implement Process Mining α-miner algorithm in column-oriented database Cassandra using Cassandra Query Language(CQL).
3. We present a performance benchmarking and comparison of α-miner algorithm on both row-oriented database MySQL and column-oriented database Cassandra.

3 α-Miner Algorithm

The α-miner algorithm is a Process Discovery algorithm used in Process Mining [13]. It was first put forward by van der Aalst, Weijter and Maruster. Input for the α-miner algorithm is an event log and output is a process model. The α-miner algorithm consists of scanning the event logs for discovering causality between the activities present in the event log.

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

1. Compute Total Events which represents the set of distinct activities present in the event log.
2. Computes Initial Events which represents the set of all the initial activities of corresponding trace.
3. Compute Final Events which represents the set of distinct activities which appear at the end of some trace in the event log.
4. Compute the relationships between all the activities in Total Events. 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. Keep only the maximal pairs of sets generated in the fourth step, eliminating the non-maximal ones.
6. Add the input place which is the source place and the output place which is the sink place in addition to all the places obtained in the fifth step.
7. Present all the places including the input and output places and all the input and output transitions from the places.

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

We present a few segments of our implementation due to limited space in the paper. The entire code and implementation can be downloaded from our website[4]. Before implementing α-miner algorithm, we do pre-processing in JAVA to create the following two tables *viz.* causality table (consists of two columns eventA and eventB) and NotConnected table (consists of two columns eventA and eventB).

1. We create a table eventlog using create table[5] keyword consisting of 3 columns (CaseID, Timestamp and Activity) each of which are varchar datatype except Timestamp which is of timestamp datatype. The primary key is a composite primary key consisting of CaseID and Timestamp.
2. We load the data into table eventlog using LOAD DATA INFILE[6] 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 and insert into table totalEvent.
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 minimum value of Timestamp from table eventlog by grouping CaseID.

4 http://bit.ly/1C3JgIx.
5 http://dev.mysql.com/doc/refman/5.1/en/create-table.html.
6 http://dev.mysql.com/doc/refman/5.1/en/load-data.html.

   (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 table
   (a) We first select maximum 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 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.
   (a) In table causality we use group_concat[7] to combine the values of column eventB for corresponding value of column eventA and insert the results in table EventA.
   (b) In table causality we use group_concat to combine the values of column eventA for corresponding value of column eventB and insert the results in 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 and causality excluding all rows in tables eventAsafe and eventSafeB.
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. 'i' and 'o' represent the source and sink places respectively.

[7] http://dev.mysql.com/doc/refman/5.0/en/group-by-functions.html#function_group-concat.

   (b) To populate table PL, we use concat_ws [8] to combine the values of column setA and column setB of 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.
   (a) 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.
   (b) 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.
   (c) 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.
   (d) 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.
   (e) To populate a 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 Implementation of α-Miner Algorithm on NoSQL Column-Oriented Database Cassandra

Before implementing α-miner algorithm, we do pre-processing in JAVA to create the following two tables *viz.* causality table (consists of two columns eventA and eventB) and NotConnected table (consists of two columns eventA and eventB).

The data in Cassandra is modelled as a wide column family, where CaseID is the partition key and Timestamp is the clustering key. Timestamp represents the cell names and Activity represents the cell values. Since Timestamp is the clustering key, all the Activities are stored in ascending order of Timestamp values for a particular partition key, that is CaseID.

1. Create keyspace.
   We create a keyspace called eventdataset. In Cassandra, keyspace is the container for the data we need to store and process.
2. Create and populate table eventlog.
   We create a table eventlog with CaseID as the partition key and Timestamp as the clustering key and load the dataset into the table eventlog using COPY command.

[8] http://dev.mysql.com/doc/refman/5.0/en/string-functions.html#function_concat-ws.

3. Create and populate table totalEvents.
   The Create command creates the table totalEvents with tevents(that represents activities in the eventlog) as the primary key for storing the total distinct activities in the eventlog. We use the COPY command two times to obtain the set of distinct activities. The first COPY command loads all the activities of the eventlog into ‘totalEvents.csv’ file. The second COPY command copies the activities stored in ‘totalEvents.csv’ file to the table totalEvents.
4. Create and populate table for storing all the initial activities.
   The table startevents is used to store all the initial activities i.e., all those activities that appear in the start of some trace. We create the table initial with the rows being ordered in the decreasing order of Timestamp and load the data from ‘eventlog.csv’ into it. The columns CaseID, Timestamp and Activity of table initial are copied to the file ‘inital.csv’. Therefore, the file ‘initial.csv’ will contain the rows in decreasing order of timestamp. Then we load the data from file ‘initial.csv’ into table starteve. The row with the earliest value of timestamp corresponding to a CaseID will sustain in the table. Finally, we create the table startevents with ievents(that represent activities in the eventlog) as the primary key and copy the column sevents, which represents activities in the eventlog into the file ‘startEve.csv’ and load this file into the table startevents giving all the distinct initial activities in the eventlog.
5. Create and populate table finalEvents.
   The table finalEvents is used to store all the ending activities i.e., all those activities that appear in the last of some trace. The query can be explained as: We create the table final and load the data from ‘eventlog.csv’ into it. Since only CaseID is the primary key in the table final, it will contain only rows with the highest value of Timestamp corresponding to a CaseID. The column Activity of table final is copied to file ‘finalEve.csv’. Finally, we create the table finalEvents with fevents(that represent activities in the eventlog) as the primary key. We load the data from file ‘finalEve.csv’ into the table finalEvents.
6. Perform preprocessing to create the footprint matrix.
   We create a table trace with CaseID as the primary key and list of events corresponding to each CaseID.
   (a) We iterate over the table eventlog and store the list of activities corresponding to each CaseID in an arraylist traceList.
   (b) We insert the value of traceList corresponding to each CaseID in table trace.
   (c) We create the table called preprocesstable with ACTIVITY as the primary key and columns as the number of activities in the event log. We initialize all the values in the preprocesstable as NOT CONNECTED.
   (d) We scan the table trace and update the corresponding entries in the table preprocesstable.

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

7. Create and populate Table SetsReachabeTo and SetsReachableFrom. We create table SetsReachableTo with 3 columns - Place which is the primary key, activityin which represents the set of input activities and activityout which represents the set of the output activities. Similarly, we create table SetsReachableFrom with 3 columns - Place, activityout, activityin. For each activity in the totalActivities table, we read from the preprocesstable its relation to every other activity. If the relation is CAUSALITY, then we insert it into table setsreachablefrom. Similarly, we populate table setsreachableto. Finally we merge the two tables using COPY command.
8. Create and Populate Table YW.
   We create a table YW with Place as the primary key and columns-activityin storing the input activities and column activityout storing the output activities. We iterate over the table XW and check if a particular place is a subset of other place. If Place X is a subset of Place Y, we insert Place Y in table YW.
9. Create Places.
   We alter the table YW by adding a column called placeName. For each input and output activity in table YW, we concatenate the activity values and insert into the corresponding value of column placeName. Then, we insert the input place i.e. the source place represented as 'iL' and the output place i.e. the sink place represented as 'oL'.
10. Insert the set of initial and final events.
    (a) We create two hashsets initialActivity and finalActivity.
    (b) We read from the table initial obtained in Step 2 and add the activities to hashset initialActivity. Similarly, we obtain the hashset finalActivity.
    (c) Finally, we insert both the hashsets into table YW for corresponding input and output places.

6 Experimental Dataset

We use Business Process Intelligence 2013(BPI 2013)[9] dataset to conduct our experiments. The log contains events from an incident and problem management system called VINST. The event log is provided by Volvo IT Belgium. The data is related to the process of serving requests in an IT department. The process describes how a problem is handled in the IT services operated by Volvo IT. The dataset is provided in CSV format. We use the VINST cases incidents CSV file dataset, which contains 65533 records. We use the following fields:

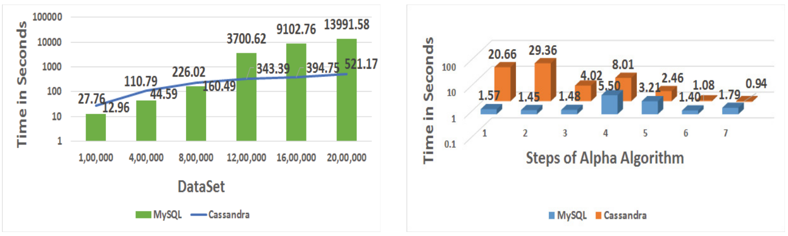
1. Problem Number: It represents the unique ticket number for a problem. It is represented as CaseID in our data model.
2. Problem Change Time: It represents the time when the status of the problem changes. It is represented as Timestamp in our data model.
3. Problem Substatus: It represents the current state of the problem. It is represented as Activity in our data model.

[9] http://www.win.tue.nl/bpi/2013/challenge.

7 Benchmarking and Performance Comparison

Our benchmarking system consists of Intel Core i3 2.20 GHz processor, 4 GB Random Access Memory (RAM), 245 GB Hard Disk Drive (HDD), Operating System (OS) is Windows 2008 LTS. The experiments were conducted on MySQL 5.6 (row-oriented database) and Cassandra 2.0.1 (NoSQL column-oriented database). 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 3 columns (CaseID, Timestamp and Activity) each of which are of varchar datatype except Timestamp which is of timestamp datatype. 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 paper.



(a) Dataset Load Time in Seconds (b) α-miner Stepwise Execution Time in Seconds

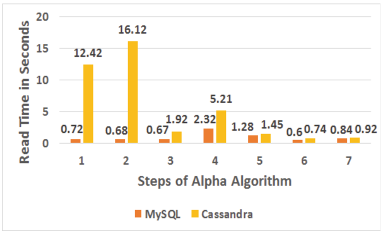
Fig. 1. Dataset load time and α-miner stepwise execution time

Our first experiment consists of investigating the time taken to perform bulk loading in both the databases across various dataset sizes. Figure 1(a) shows that the time taken to load data in MySQL is initially lower as compared to Cassandra. However, as the dataset size increases beyond a certain point, the time taken to load data in MySQL increases as compared to Cassandra. On an average the time taken to perform bulk loading in Cassandra across various dataset sizes is 16 times lower than that of MySQL. We believe the reason for the increase in load time in MySQL when the dataset size increases can be the difference between how bulk loading is performed in both the databases. 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. LOAD DATA INFILE command also creates an index and checks uniqueness while inserting records in a table. Bulk loading in Cassandra is done using COPY command which expects the number of columns in CSV file to be the same as the number of columns in the table metadata. When the size of the dataset increases then in case of MySQL most of the time is spent in checking uniqueness and creating indexes.

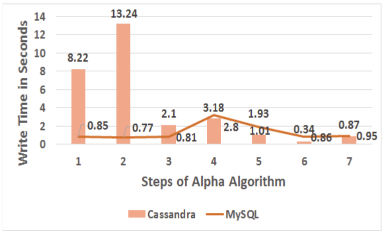
There is no such overhead in Cassandra. Hence, the time taken to load the data in MySQL increases when the dataset size increases.

α-miner algorithm is a seven step algorithm (Refer to Section 3). Few steps are read intensive (Steps 1–3) while few steps are write intensive (Steps 4–7). We perform an experiment to compute the α-miner algorithm execution time of each step in both MySQL and Cassandra 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. Figure 1(b) reveals that the stepwise time taken in MySQL and Cassandra varies differently for each of the steps. We conjecture the reason for the difference in time taken can be the difference in the internal architecture of MYSQL and Cassandra and the number of reads and writes performed at each step. For the first three steps, the time taken in Cassandra is 12 times more as compared to MySQL. The reason for this is that we copy the data to and from CSV files in order to obtain the total, initial and final activities. We create a number of tables and load the data a number of times to get the set of activities. This overhead of reading from CSV files and writing to CSV files is not there in MySQL. The fourth step involves reading from the tables created in the first three steps, the causality table and the eventlog table to create the table XL. 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 Cassandra data is read from the memtable. If values are not in memtable they are read from Sorted String Tables (SSTables). All entries of a row are present across multiple SSTables. Thus, the read operation in Cassandra involves reading from the Memtable and multiple SSTables. Hence, read performance of MySQL is better as compared to Cassandra. The fifth step includes more number of insert operations as compared to read operations. 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. Cassandra follows log structure merge tree index. In case of Cassandra, values are written in an append only mode. The writes in Cassandra are sequential. There are no in place updates. Hence, Cassandra gives better write performance as compared to MySQL. Similarly for Step 6, there are more number of insert operations and hence, Cassandra gives a better performance for step 6. Step 7 involves the execution of stored procedures and a lot of data is read from the database, processed and inserted into the FL table. However, in Cassandra, Step 7 involves inserting only the set of initial and final activities and adding the input and output place names to the YL table. Since, the number of insert operations performed are more as compared to the select operations, the overall time taken in MySQL is more as compared to Cassandra.

We conduct an experiment to compare which of the database performs better for read and write operations in both MySQL and Cassandra. As can be seen from Fig. 2(a), the total time taken in MySQL to read data is 3.7 times lower than the time taken to read data in Cassandra. According to us, the reason for Cassandra giving better read performance can be the difference in the data



(a) Read Intensive Time in Seconds



(b) Write Intensive Time in Seconds

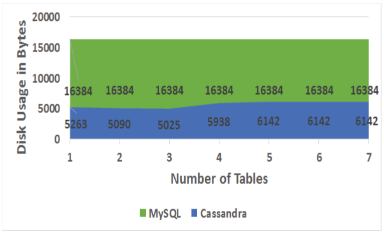
Fig. 2. Read and write intensive time

structure used by both the databases. In MySQL, only B-Tree index needs to be scanned to find the location of block where the data is stored. In case of Cassandra data is read as described below-

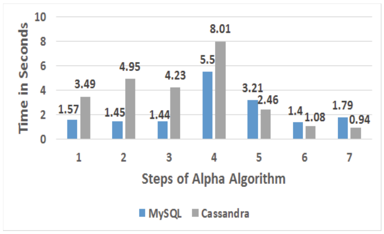
1. To find the data, Cassandra client first scans the memtable.
2. When the data is not found in the memtable, Cassandra scans all the SSTables as all the fields of a particular record can be present across multiple SSTables.

Figure 2(b) shows that the write performance of Cassandra is better as compared to MySQL except for the first three steps as the first three steps have the overhead of copying to and from CSV files. For the remaining steps Cassandra performs writes 1.37 times faster than MySQL. We believe the reason for Cassandra giving better write performance for the remaining steps can be the difference in the way writes are performed in both the databases. In MySQL, the B-Tree index needs to be scanned 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. Figure 2(b) 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 Cassandra we perform the same steps, that is all the analytical processing using Java because CQL does not support advanced features of SQL. Writes in Cassandra are performed in an append only mode. There is no overwrite in place. All update operations append the new data which gets merged with the older data in the background. Thus, writes in Cassandra are more optimized as compared to MySQL.

We perform an experiment to investigate which database can efficiently store the results of each step of α-miner algorithm using minimum disk space. We include only the data length (excluding the size of the index table) in the disk space of a table. Figure 3(a) reveals that Cassandra on an average uses disk space 4.06 times lower than that of MySQL for tables created at each step of the algorithm. The cumulative disk space for storing all the tables in MySQL



(a) Disk Usage of Tables in Bytes

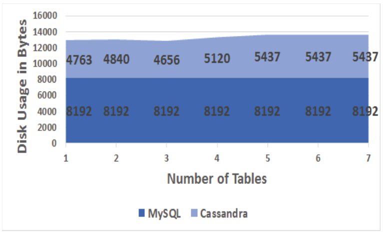


(b) α-miner Stepwise Execution Time in Java

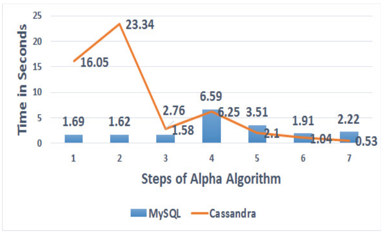
Fig. 3. Disk usage of tables in bytes and α-miner stepwise execution time in Java

is 162388 bytes while for Cassandra the space is 39742 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 equal to the dataset size divided by the block size. In MySQL, if a particular set of blocks or block has been allocated for a table then that particular set of blocks or block can be used only by that table. Either the table completely utilizes the space provided to it or the space in the last block is not utilized. Storing tables of size less than 16 KB leads to under-utilization of space and the remaining space cannot be utilized by other tables. Cassandra stores the key-value pairs in an in memory data structure called memtable. Memtables are flushed to the disk when the size of the data increases beyond a certain threshold value. The data is then written to the disk in data files called SSTables which is a file of key-value string pairs and is equal to the size of the data flushed from the memtable. Hence, the disk space at each step is more efficiently utilized in Cassandra as compared to MySQL.

The step wise execution time of α-miner algorithm for the first three steps in Cassandra is very high as compared to MySQL. The reason is the overhead of copying data to and from CSV files to get the desired results. In order to avoid this overhead, we instead of using the COPY command use only Select and Insert statements with some Java processing. We create a table trace that stores CaseID as the primary key and list of activities ordered by timestamp. We extract the first activity from the list corresponding to each CaseID and insert it in the table storing initial activities. We extract the last activity from the list corresponding to each CaseID and insert it in the table storing the final activities. Similarly we insert all the activities in the list and insert the values in the table storing all the activities. As can be seen from Fig. 3(b), the stepwise execution time in Cassandra reduces by 4.26 times as compared to the time taken using COPY command. Thus, the performance of Select and Insert statements with Java processing gives a much better performance as compared to the COPY command.



(a) Disk Usage of Tables in Bytes with Compression



(b) α-miner Stepwise Execution Time with Compression

Fig. 4. Disk usage of tables in bytes and α-miner stepwise execution time with compression

Disk space can be efficiently utilized by using the compression technique. We conduct an experiment to compute the disk space occupied by tables at each step of the α-miner algorithm with compression enabled. Figure 4(a) illustrates that when we compare the disk space occupied by each table without compression and with compression the compression ratio (Actual size of table/Compressed size of table) is better in MySQL as compared to Cassandra. The maximum and minimum compression ratio in MySQL is 2. Similarly, the maximum compression ratio in Cassandra is 1.15 and minimum compression ratio is 1.05. We believe the reason for MySQL having a better compression ratio as compared to Cassandra is the difference in the compression techniques used by both the databases. MySQL uses zlib compression technique which provides a better compression ratio but not a good decompression speed while Cassandra uses LZ4 compression technique which does not provide a very good compression ratio but provides very fast decompression speed.

We conduct an experiment to examine the time taken by each step of the α-miner algorithm with compression technique. Figure 4(b) reveals that the stepwise performance improves for each step in Cassandra while it degrades for each step in MySQL. We believe the reason for performance degradation when compression is enabled can be the difference in the way compression is performed in both the databases. MySQL uses blocks of sizes 1 KB, 2 KB, 4 KB, 8 KB and 16 KB. The default block size after compression in MySQL is 8 KB. Thus, if a block size after compression is 5 KB, then the block will be first uncompressed, then split into two blocks and finally recompressed into blocks of size 1 KB and 4 KB. Cassandra does not have a fixed block constraint after compressing a block. Also, with compression Cassandra can quickly find the location of rows in the SSTables. SSTables in Cassandra are immutable and hence recompression is not required for processing write operations. We observe that the total time taken in executing α-miner algorithm by Cassandra without compression is 2.02 times lower than time taken with compression. Similarly, the total time taken in MySQL with compression is 1.11 times lower than time taken without compression.

8 Conclusion

In this paper, we present an implementation of α-miner algorithm in MySQL and Cassandra using the database specific query language (SQL and CQL). Furthermore, we present the performance comparison of α-miner algorithm in MySQL and Cassandra. The α-miner implementation in MySQL is a one tier application which uses only standard SQL queries and advanced stored procedures. Similarly, implementation in Cassandra is done using CQL and Java. We conclude that when the size of the dataset increases, the time taken by Cassandra on an average is 16 times faster than MySQL. Based on experimental results, we conclude that Cassandra outperforms MySQL in loading real time large datasets data (event logs).

The total time taken to read the data while execution of α-miner algorithm is 3.70 times lower in MySQL as compared to Cassandra. Similarly, for writing the data, time taken by Cassandra is 1.37 times lower as compared to MySQL. Cassandra outperforms MySQL in terms of the disk usage of tables. The disk space occupied by tables in Cassandra is 4.06 times lower as compared to MySQL. Thus, we conclude that Cassandra is more efficient than MySQL in terms of storing data and performing query. When compression is enabled, performance of Cassanadra improves by 0.22 % while that of MySQL degrades by 0.19 %.

References

1. Carlos, O.: Programming the k-means clustering algorithm in SQL. In: ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 823–828 (2004)
2. Ordonez, C., Cereghini, P.: SQLEM: fast clustering in SQL using the EM Algorithm. In: International Conference on Management of Data, pp. 559–570 (2000)
3. Abadi, D.J., Madden, S.R., Hachem, N.: Column-stores vs. row-stores: how different are they really? In: SIGMOID (2008)
4. Rana, D.P., Mistry, N.J., Raghuwanshi, M.M.: Association rule mining analyzation using column oriented database. Int. J. Adv. Comput. Res. **3**(3), 88–93 (2013)
5. Finn, M.A.: Fighting impedance mismatch at the database level. White paper (2001)
6. Gupta, K., Sachdev, A., Sureka, A.: Pragamana: performance comparison and programming alpha-miner algorithm in relational database query language and NoSQL column-oriented using apache phoenix. In: Proceedings of the Eighth International C* Conference on Computer Science & Software Engineering, C3S2E 2015, pp. 113–118 (2008)
7. Joishi, J., Sureka, A.: Vishleshan: performance comparison and programming process mining algorithms in graph-oriented and relational database query languages. In: Proceedings of the 19th International Database Engineering and Applications Symposium, IDEAS 2015, pp. 192–197 (2014)
8. Sattler, K.-U., Dunemann, O.: SQL database primitives for decision tree classifiers. In: Conference on Information and Knowledge Management, pp. 379–386 (2001)
9. Suresh, L., Simha, J., Velur, R.: Implementing k-means algorithm using row store and column store databases-a case study. Int. J. Recent Trends Eng. **4**(2) (2009)

10. Plattner, H.: A common database approach for OLTP and OLAP using an in-memory column database. In: ACM SIGMOD International Conference on Management of Data (2009)
11. Russell, N.C.: Foundation of process-aware information systems. Dissertation (2007)
12. Sharma, V., Dave, M.: SQL and NoSQL database. Int. J. Adv. Res. Comput. Sci. Softw. Eng. **2**(8), 20–27 (2012)
13. Weerapong, S., Porouhan, P., Premchaiswadi, W.: Process mining using α-algorithm as a tool. IEEE (2012)
14. Aalst, W.V.D.: Process mining: overview and opportunities. ACM Trans. Manage. Inf. Syst. **3**(2), 1–17 (2012)