Vishleshan: Performance Comparison and Programming Process Mining Algorithms in Graph-Oriented and Relational Database Query Languages

Jeevan Joishi
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

9 March 2015
1. Prof. Ashish Sureka (Thesis Adviser)

2. Prof. Sandip Aine (Internal Examiner)

2. Dr. Radha Krishna Pisipati (External Examiner)

Day of the defense: 9 March 2015

Signature from Post-Graduate Committee (PGC) Chair:
Abstract

Information Systems today record the execution of activities into event logs. Process Mining is an area of research that deals with the study and analysis of various business processes based on these event logs. These event logs also record the performers of each of the activities. Mining social network using this information, understanding work-flow management and deriving relationships between actors based on different metrics viz. handover of task, subcontract, etc. is what constitutes Organizational Mining. Metrics based on Joint Activities and Metrics based on (possible) causality, commonly referred to as Similar-Task Algorithm and Subcontract Algorithm forms the basis of this paper. We present Cypher Query Language (Neo4j) and SQL (Structured Query Language) implementations of Similar-Task and Subcontract Algorithms. Graph Databases have shown to perform well in cases where information follows linked structure and needs to query to depth(s) of a hierarchical setup. We conduct an empirical study on a large real world data set to compare the performance of Neo4j against MySQL. We benchmark performance factors like query execution time, CPU usage and disk/memory space usage when implementing Similar-task and Subcontract algorithms in Cypher Query Language and SQL.
I dedicate my MTech Thesis to my late Aunt, Mrs. Jamuna Joishi, who had always been a source of inspiration, blessed me and had given me strength to ever move forward in my academic ventures.
Acknowledgements

“Tell me and I forget, teach me and I may remember, involve me and I learn”

— Benjamin Franklin

Learning never stops. Dr. Ashish Sureka has been an embodiment of this philosophy. He urges us to move forward with conviction and his pursuit of quality work is relentless. I would like to take this wonderful opportunity to thank Dr. Ashish Sureka for his constant guidance during the work. His knowledge on the topic and his motivation and patience to lead us towards quality work is overwhelming source of inspiration. Its an honor and a privilege to have him as my mentor.

I would also like to thank my parents for their constant support, encouragement and patience during the course of the work.
Declaration

This is to certify that the MTech Thesis Report titled Vishleshan: Performance Comparison and Programming Process Mining Algorithms in Graph-Oriented and Relational Database Query Languages submitted by Jeevan Joishi 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 xi

List of Tables xii

1 Research Motivation and Aim 1

1.1 Graph Database ........................................ 2

1.1.1 What is a Graph? ................................... 2

1.1.2 The Property Graph Model ........................ 2

1.1.3 Graph Databases .................................... 3

1.1.4 Neo4j ..................................................... 5

1.1.4.1 Internals of Neo4j ............................... 5

1.2 Process Mining and Organizational Perspective .......... 9

1.2.1 Process Mining ..................................... 9

1.2.2 Organizational Perspective ........................ 11

1.2.2.1 Metrics for Organizational Perspectives ...... 11

1.3 Research Aim .......................................... 12

2 Related Work 15

2.1 Implementation of Mining Algorithms in Relational Databases 15

2.2 Implementation of Mining Algorithms in Graph Databases 15

2.3 Performance comparison between Relational Databases and Graph Databases 16

3 Algorithm Description 17

3.1 Similar-Task Algorithm .................................. 18

3.1.1 Description .......................................... 18
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.2</td>
<td>Algorithm</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>Subcontract Algorithm</td>
<td>20</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Description</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>Implementation of Algorithms on RDBMS</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>Similar-Task Algorithm</td>
<td>24</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Pre-Processing</td>
<td>24</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Initial Similarity Calculation</td>
<td>25</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Final Similarity Calculation</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Subcontract Algorithm</td>
<td>28</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Native SQL Implementation</td>
<td>28</td>
</tr>
<tr>
<td>4.2.1.1</td>
<td>Pre-Processing</td>
<td>28</td>
</tr>
<tr>
<td>4.2.1.2</td>
<td>Create Initial Matrix</td>
<td>29</td>
</tr>
<tr>
<td>4.2.1.3</td>
<td>Get Distinct Cases</td>
<td>30</td>
</tr>
<tr>
<td>4.2.1.4</td>
<td>Find Subcontraction</td>
<td>31</td>
</tr>
<tr>
<td>4.2.1.5</td>
<td>Normalize the Final Matrix</td>
<td>33</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Memoization</td>
<td>34</td>
</tr>
<tr>
<td>4.2.2.1</td>
<td>Bottleneck</td>
<td>34</td>
</tr>
<tr>
<td>4.2.2.2</td>
<td>Improved alternative approach</td>
<td>35</td>
</tr>
<tr>
<td>5</td>
<td>Implementations of Algorithms on Neo4j-NoSQL</td>
<td>37</td>
</tr>
<tr>
<td>5.1</td>
<td>Similar-Task Algorithm</td>
<td>37</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Schema Definition</td>
<td>37</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Native CYPHER Implementation</td>
<td>37</td>
</tr>
<tr>
<td>5.2</td>
<td>Subcontract Algorithm</td>
<td>39</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Schema Definition</td>
<td>39</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Native CYPHER Implementation</td>
<td>40</td>
</tr>
<tr>
<td>5.2.2.1</td>
<td>Identify Subcontracting Actors</td>
<td>40</td>
</tr>
<tr>
<td>5.2.2.2</td>
<td>Collect distinct Actors</td>
<td>40</td>
</tr>
<tr>
<td>5.2.2.3</td>
<td>Set Subcontraction values</td>
<td>41</td>
</tr>
<tr>
<td>5.2.2.4</td>
<td>Calculate normal</td>
<td>41</td>
</tr>
<tr>
<td>5.2.2.5</td>
<td>Normalize the result</td>
<td>41</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Memoization</td>
<td>42</td>
</tr>
<tr>
<td>5.2.3.1</td>
<td>Bottleneck</td>
<td>42</td>
</tr>
</tbody>
</table>
CONTENTS

5.2.3.2  Improved alternative approach                    43

6  Experimental Dataset                                  45

7  Performance Comparison                              50
   7.1  Similar-Task Algorithm                          50
   7.2  Sub Contract Algorithm                          54

8  Limitations and Future Work                         62

9  Conclusion                                           64

References                                             66
List of Figures

1.1 A small social graph .................................. 3
1.2 Graph with properties .................................. 4
1.3 Architecture of Neo4j (Figure adapted from [1]) .......... 5
1.4 Non-native Graph Processing using Global lookup index .......... 7
1.5 Native Graph Processing using index-free adjacency .......... 7
1.6 Storage Pattern for nodes, relationships and properties (Figure adapted from [1]) .............. 8
1.7 Types of Process Mining Techniques ......................... 10
3.1 An example of MXML event log ............................. 22
5.1 Schema-Definition for Similar-Task .......................... 38
5.2 Schema-Definition for Sub-Contract ......................... 39
6.1 Incident Activity Record Dataset ........................... 46
6.2 Number of Events per Case ............................... 46
6.3 Top Cases ordered by number of Events .................... 47
6.4 Frequency of Actors .................................... 47
6.5 Highest Frequency Actors ................................ 48
6.6 Frequency of Activities ................................... 48
6.7 Highest Frequency Activities .............................. 49
7.1 Data Load Time for Similar-Task Algorithm ................ 51
7.2 Cosine-Similarity calculation in Step-8 ..................... 53
7.3 Time Taken to update results in Step-9 ..................... 54
7.4 Disk Usage in MySQL (Similar Task) ....................... 55
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.5</td>
<td>Disk Usage in Neo4j (Similar Task)</td>
<td>56</td>
</tr>
<tr>
<td>7.6</td>
<td>Data Load Time for Sub-Contract Algorithm</td>
<td>57</td>
</tr>
<tr>
<td>7.7</td>
<td>Execution Time in MySQL (Sub-Contract)</td>
<td>58</td>
</tr>
<tr>
<td>7.8</td>
<td>Execution Time in Neo4j (Sub-Contract)</td>
<td>59</td>
</tr>
<tr>
<td>7.9</td>
<td>Disk Usage in MySQL (Sub-Contract)</td>
<td>60</td>
</tr>
<tr>
<td>7.10</td>
<td>Disk Usage in Neo4j (Sub-Contract)</td>
<td>61</td>
</tr>
</tbody>
</table>
## List of Tables

3.1 Event Log ................................................................. 18
3.2 Actor-Activity Matrix .............................................. 19
3.3 Cosine-Similarity Values ........................................... 20
3.4 Sub-contraction Values ............................................. 23

7.1 Number of Unique Actors per dataset size. ..................... 50
7.2 Data Load Time \((Sim\textit{ilar} − \textit{Task})\) ............................. 51
7.3 Execution Time for Step-8 and Step-9 \((Sim\textit{ilar} − \textit{Task})\) .......... 52
7.4 Disk Space Usage (bytes) for MySQL tables \((Sim\textit{ilar} − \textit{Task})\) ...... 53
7.5 Disk Space Usage (bytes) for Neo4j Elements \((Sim\textit{ilar} − \textit{Task})\) .... 54
7.6 Data Load Time \((Sub − \textit{Contract})\) ............................... 55
7.7 Execution Time for Sub-Contract Algorithm in MySQL ......... 56
7.8 Execution Time for Sub-Contract Algorithm in Neo4j ........... 57
7.9 Disk Space Usage (bytes) for MySQL tables \((Sub − \textit{Contract})\) ..... 59
7.10 Disk Space Usage (bytes) for Neo4j elements \((Sub − \textit{Contract})\) .... 60
Research Motivation and Aim

Relational databases handle tabular structures exceedingly well. But for many years now, developers have faced problems in trying to handle highly connected data with relational databases.

Relational databases handle relationships poorly, mostly due to join intensive queries leading to JOIN BOMB. The reason is that relationships in relational databases can be modeled by means of joins only, and an increase in connectedness of data implies increased number of joins. Join intensive queries are an impediment to performance and scalability in a dynamic system with ever-changing business needs. Furthermore, complications arise when, in addition to modeling the relationships, we also need to weigh the strength of these relationships.

Graph databases have emerged to address the issue of leveraging complex and dynamic relationships in highly connected data. In contrast to relational databases, where performance deteriorates as the size of the dataset increases, performance of a graph database is expected to remain constant, even as the dataset grows. This is because queries would be localized to a portion of the graph and hence, the execution time for each query would depend only on the part of the graph traversed to satisfy that query, instead of the overall size of the graph [1].

Process Mining is young, yet a broad field of research. Organizational Mining is one of the three perspectives of Process Mining that deals with the study of social relation-
ship between individuals. It is similar to social mining. Analysing social relationship with Relational Databases involves performing joins between multiple tables. However Graph Databases are built to avoid these performance intensive joins.

1.1 Graph Database

1.1.1 What is a Graph?

Formally, a graph is just a collection of vertices and edges, or in less intimidating language, a set of nodes and the relationships that connect them. Graphs represent entities as nodes and the ways in which those entities relate to the world as relationships. This general-purpose, expressive structure allows us to model all kinds of scenarios, from the construction of a space rocket, to a system of roads, and from the supply-chain or provenance of foodstuff, to medical history for populations, and beyond.

For example, Twitter's data is easily represented as a graph. In Figure 1.1, a small network of followers is presented. The relationships are key here in establishing the semantic context: namely, that Pooja follows Astha, and that Astha, in turn, follows Pooja. Astha and Kunal likewise follow each other, Pooja follows Kunal, Kunal follows Nidhi and Astha follows Nidhi, but sadly, Kunal hasn't (yet) reciprocated to Pooja, Nidhi neither follows Kunal nor Astha.

1.1.2 The Property Graph Model

The Property Graph Model is an extension to the basic Graph Model wherein nodes and edges are populated with properties to reflect the nature of that graph element. These properties are basically key value pairs. Properties are pertinent information that relates to nodes and relationships. Properties are intuitive and easy to understand. Properties on a node may indicate its name or some characteristics, whereas properties on edges may indicate the relationships between nodes, any strength or weight associated with the nodes connected by that edge and so on.

Figure 1.2 informally introduces the most popular variant of graph model, the property graph. A property graph has the following characteristics:
1. A property graph is made up of nodes, relationships, and properties.

2. Nodes contain properties. Think of nodes as documents that store properties in the form of arbitrary key-value pairs. The keys are strings and the values are arbitrary data types.

3. Relationships connect and structure nodes. A relationship always has a direction, a label, and a start node and an end node; there are no dangling relationships. Together, a relationship’s direction and label add semantic clarity to the structuring of nodes.

4. Like nodes, relationships can also have properties. The ability to add properties to relationships is particularly useful for providing additional metadata for graph algorithms, adding additional semantics to relationships (including quality and weight), and for constraining queries at runtime.

1.1.3 Graph Databases

In computing, a graph database is a database that uses graph structures for semantic queries with nodes, edges, and properties to represent and store data [1]. A graph database is any storage system that provides index-free adjacency. This means that

![A small social graph](image)

**Figure 1.1**: A small social graph
1.1 Graph Database

Figure 1.2: Graph with properties

every element contains a direct pointer to its adjacent elements and no index lookups are necessary. General graph databases that can store any graph are distinct from specialized graph databases such as triplestores and network databases.

There are two properties of graph databases one should consider when investigating graph database technologies:

1. The underlying storage: Some graph databases use native graph storage that is optimized and designed for storing and managing graphs. Not all graph database technologies use native graph storage, however. Some serialize the graph data into a relational database, an object oriented database, or some other general-purpose data store.

2. The processing engine: Some definitions require that a graph database use index-free adjacency, meaning that connected nodes physically point to each other in the database.
1.1 Graph Database

1.1.4 Neo4j

Neo4j is an "embedded, disk-based, fully transactional Java persistence engine that stores data structured in graphs rather than in tables". Neo4j is a native graph processing and a native graph storage model. Neo4j is the most popular graph database. Neo4j is an open-source graph database, implemented in Java.

1.1.4.1 Internals of Neo4j

Nodes and relationships in Graph Databases can have processing capability using either a global index lookup or index-free adjacency. A graph databases is called native if it employs index-free adjacency.

1. Architecture of Neo4j

The general architecture of Neo4j is shown in the Figure 1.3. Traversal API contains functionalities for graph traversal. Traversal happens from node to node via edges (relationships). Core API provides functionalities for initiating embedded graph databases that receives client connections. It also provides capabilities to create nodes, relationships and properties [1].

Figure 1.3: Architecture of Neo4j (Figure adapted from [1])
1.1 Graph Database

CYPHER is a query language used to query elements in Neo4j. Cache in Neo4j are just part of the system memory used when Neo4j instances are created and queries are being performed on those instances. Transaction Management and Transaction log keep tracks of transactional consistency and atomicity, while their record being maintained in the log. Record Files or Store Files are those where Neo4j stores the graph data. Each store file contains data for specific part of the graph (e.g. nodes, relationships, properties, etc.). Some of the store files commonly seen are

- neostore.nodestore.db
- neostore.relationshipstore.db
- neostore.propertystore.db
- neostore.propertystore.db.index
- neostore.propertystore.db.strings
- neostore.propertystore.db.arrays

2. Native Graph Processing

A database engine that utilizes index-free adjacency is one in which each node maintains direct references to its adjacent nodes; each node, therefore acts as a micro-index of other nearby nodes. It means that query times are independent of the total size of the graph, and are instead simply proportional to the amount of the graph searched.

A non-native graph database engine, in contrast, uses (global) indexes to link nodes together, as shown in Figure 1.4. These indexes add a layer of indirection to each traversal, thereby incurring greater computational cost.

Index lookups are fine for small networks, such as the one in Figure 1.4, but too costly for complex queries over larger graphs. Instead of using index lookups
1.1 Graph Database

Figure 1.4: Non-native Graph Processing using Global lookup index

to make relationships concrete and navigable at query time, Neo4j with native graph processing capabilities use index-free adjacency to ensure high performance traversals. Figure 1.5 shows how relationships eliminate the need for index lookups.

Figure 1.5: Native Graph Processing using index-free adjacency

3. Native Graph Storage

Neo4j stores graph data in a number of different store files. Each store file contains the data for a specific part of the graph (e.g., nodes, relationships, properties).
1.1 Graph Database

The division of storage responsibilities, particularly the separation of graph structure from property data, facilitates performant graph traversals, even though it means the users view of their graph and the actual records on disk are structurally dissimilar. The storage pattern for nodes, relationships and properties is shown in Figure 1.6.

![Figure 1.6: Storage Pattern for nodes, relationships and properties (Figure adapted from [1])](image)

The node store file stores node records [1]. Every node created in the user-level graph ends up in the node store, the physical file for which is neostore.nodestore.db. Like most of the Neo4j store files, the node store is a fixed-size record store, where each record is nine bytes in length. Fixed-size records enable fast lookups for nodes in the store file: if we have a node with id 100, then we know its record begins 900 bytes into the file. Based on this format, the database can directly compute a records location, at cost \(O(1)\), rather than performing a search, which would be cost \(O(\log n)\).

The first byte of a node record is the in-use flag. This tells the database whether the record is currently being used to store a node, or whether it can be reclaimed.
on behalf of a new node (Neo4js .id files keep track of unused records). The next 
four bytes represent the ID of the first relationship connected to the node, and 
the last four bytes represent the ID of the first property for the node. The node 
record is pretty lightweight: its really just a couple of pointers to lists of relation-
ships and properties.

Correspondingly, relationships are stored in the relationship store file, 
neostore.relationshipstore.db Like the node store, the relationship store consists 
of fixed-sized records in this case each record is 33 bytes long. Each relationship 
record contains the IDs of the nodes at the start and end of the relationship, a 
pointer to the relationship type (which is stored in the relationship type store), 
and pointers for the next and previous relationship records for each of the start 
and end nodes. These last pointers are part of what is often called the relation-
ship chain[1].

1.2 Process Mining and Organizational Perspective

1.2.1 Process Mining

Process Mining basically focuses on the analysis of processes using event data. The var-
iouls data mining techniques such as classification, association, clustering are generally 
used to analyze a particular step in the overall business process but cannot be applied 
to understand and analyze a process as a whole. Process mining extracts knowledge 
from event logs. It helps to discover, analyze and improve a process model [2].

Event Log are the logs recorded by any Process Aware Information System (PAIS). 
Each event in an event log refers to an activity which is a well defined step in some 
process. Each of this activity is associated with a particular caseid i.e., a process in-
stance. The events belonging to a particular caseid are ordered. An event log may also 
contain additional information such as timestamp associated with each event and the 
actor performing the action.
1.2 Process Mining and Organizational Perspective

As can be seen from Fig. 1.7, there are basically three types of process mining techniques:

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

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

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

An activity or task in an event log refers to a well-defined step in some process. Event Log usually records the activity that is performed either by using a full description of the act or an equivalent identifier for that activity. These activities form an integral part in various Process Mining analysis algorithms like α-miner algorithm or various
organizational mining algorithms like Similar-Task or Sub-Contract algorithm. We consider three different perspectives in process mining [3]:

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

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

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

### 1.2.2 Organizational Perspective

Existing Process Aware Information System (PAIS) record information of human activity. This information can be structured in the form of a sociogram that forms the basis of Social Network Analysis. Therefore, it is both interesting and feasible to use this as a starting point for investigating the social context of work processes. A better understanding of this social context may reveal a mis-alignment between the information system and its users and may provide insights that can be used to increase the efficiency and effectively of processes and organizations.

#### 1.2.2.1 Metrics for Organizational Perspectives

Various metrics are defined in [4] for measuring the strength of relationship between performers/actors of activity. These metrics are

1. metrics based on possible causality.

2. metrics based on joint cases.

3. metrics based on joint activities.

4. metrics based on special event types.
1.3 Research Aim

*Metrics based on possible causality* monitor for individual cases how work moves among performers. One of the examples of such a metric is *handover* of work. Within a case (i.e., process instance) there is a handover of work from individual i to individual j if there are two subsequent activities where the first is completed by i and the second by j. A related metric is *subcontracting* where the main idea is to count the number of times individual j executed an activity in-between two activities executed by individual i. This may indicate that work was subcontracted from i to j.

*Metrics based on joint cases* ignore causal dependencies but simply count how frequently two individuals are performing activities for the same case. If individuals work together on cases, they will have a stronger relation than individuals rarely working together.

*Metrics based on joint activities* do not consider how individuals work together on shared cases but focus on the activities they perform. The assumption here is that people doing similar things have stronger relations than people doing completely different things. Each individual has a profile based on how frequent they conduct specific activities. *Similar-Task* algorithm is an algorithm based on this metric.

*Metrics based on special event types* consider the type of event. Thus far we assumed that events correspond to the execution of activities. However, there are also events like reassigning an activity from one individual to another. For example, if i frequently delegates work to j but not vice-versa it is likely that i is in a hierarchical relation with j. From an SNA point of view these observations are particularly interesting since they represent explicit power relations.

1.3 Research Aim

Relational databases are very good at solving certain data storage problems but they can create problems when it is time to scale. When the size of the dataset increases the time taken to compute joins heavily increases. In such cases we need to find a way to get rid of the joins and avoid compute intensive joins. On the other hand, graph databases
came into existence with the aim of overcoming these shortcomings. Relationships are first-class citizens of the graph data model, unlike other database management systems, which require us to infer connections between entities using contrived properties such as foreign keys, or out-of-band processing like map-reduce. By assembling the simple abstractions of nodes and relationships into connected structures, graph databases enable us to build arbitrarily sophisticated models that map closely to our problem domain. The resulting models are simpler and at the same time more expressive than those produced using traditional relational databases and the other NOSQL stores.

The volume of data in an organization is increasing at a very fast rate. With this, the need to store this data to in order to make critical business decisions and satisfy user demands becomes very important. Both users and decision makers need a faster and more convenient way to access the data as quickly as possible. Analytical applications need to read only a few attributes of a large number of records. Thus, if row oriented approach is used for storing and querying data in analytical applications, then a large amount of unimportant data needs to be read in the memory and the overhead of deriving relations between the data is quite high.

Process mining focuses on the analysis of processes using event data. One of the key aspects of process mining is to generate process models that can be used for analysis purposes in the growing business needs [2]. Thus process mining is basically an analytical application. Other broad aspect of process mining is Organizational Mining. Organizational Mining also focuses on inferring relationships between actors and they use various metrics to bind actors under one hood or the other. Thus graph database proves to be an able fit to model and study organizational sociogram.

Query language like SQL (Structured Query Language) has been growing tremendously over the years and have become a standard way of interacting with the database. Unlike SQL, CYPHER query language is quite new but focuses on area where SQL doesn’t perform well, particularly joins. This paper attempts to model organizational mining algorithms in these database languages to the extent possible.

The specific research aim of this work can be summed up as follows:
1.3 Research Aim

1. To investigate the intersection of Process Mining and Graph Database(s) for detecting social, hierarchical structures.

2. To understand applications needs that can be modelled into this new domain.

3. To implement organizational mining algorithms *viz.* Similar-Task algorithm and Sub-Contract algorithm in row oriented data store MySQL.

4. To implement Similar-Task algorithm and Sub-Contract algorithm in graph oriented database Neo4j.

5. To compare the performance of Similar-Task algorithm in MySQL and Neo4j.

6. To compare the performance of Sub-Contract algorithm in MySQL and Neo4j.
2

Related Work

This section reviews closely the related work that are presented in this paper. It lists the novel contributions that this paper puts forward.

2.1 Implementation of Mining Algorithms in Relational Databases

Ordonez et al. did an extensive work on implementing k-means clustering algorithm in SQL [5]. They came up three different SQL implementations of k-means algorithm to integrate it with RDBMS. Experiments were performed on large clusters, efficient indexing and with queries optimized and re-written. Ordonez et al. also presented SQL implementations of EM Algorithm that worked with high dimensional data, high number of clusters and very large datasets [6]. They came up with three different strategies viz. Horizontal, Vertical and Hybrid. Ordonez et al. came up with another SQL implementation of clustering algorithm which merges Markov Chain Monte Carlo with EM algorithm [7]. Sattler et al. described primitives for applying and building decision tree classifiers which were directly coupled on commercial databases used in various classification problems [8].

2.2 Implementation of Mining Algorithms in Graph Databases

Wang et al. presented papers that studied structural pattern mining for large disk based graph databases. They presented a novel ADI index structure and efficient algorithms for mining frequent patterns [9]. Wang et al again came up with novel techniques to obtain scalable mining on large disk based graph databases [10]. Huan et al. also
2.3 Performance comparison between Relational Databases and Graph Databases.

presented techniques to find maximal frequent sub-graphs from Graph Databases [11]. Ozaki came up with the concept of hyperclique pattern in graph databases to detect highly correlated sub-graph in graph structured databases. It considers general ordering of sub-graphs and employed techniques like breadth-first search/depth-first search with powerful pruning techniques based on various measures [12].

2.3 Performance comparison between Relational Databases and Graph Databases.

Vicknair et al. performed comparisons between Relational Databases and Graph Databases. Their work included recording and querying data provenance information [13]. McColl et al. evaluated performance for a series of open-source graph databases. They used four different graph algorithms to evaluate performance for graph setup consisting up to 256 million nodes [14]. Ciglan et al. came up with benchmarking of graph databases over graph traversal algorithms [15]. Macko et al. presented a performance introspection framework for graph databases, PIG. PIG provided techniques and tools to understand performance of graph databases [16].

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 study organizational mining algorithms in relational databases.

2. While graph databases have been used to implement data mining algorithms, they have never been used to harvest the power of process mining perspective like organizational mining. We present, in this paper, the implementation of two different organizational mining algorithms which belongs to different categories of social analytic metrics.

3. We present a performance benchmarking of organizational mining algorithms on both relational and graph database.
Algorithm Description

Process mining is a process management technique that allows for the analysis of business processes based on event logs. The basic idea is to extract knowledge from event logs recorded by an information system. Process mining aims at improving this by providing techniques and tools for discovering process, control, data, organizational, and social structures from event logs.

Organizational Mining is a field of Process Mining that helps in mining the organizational model for the event logs. It enables one to understand the information flow in an organization, the hierarchy in-place at the workplace. It also provides enough information to understand the social interaction in an organization and group the actors in one way or the other.

Organizational Mining can be best studied using different metrics. Some of these metrics consider events on case basis, whereas some are irrespective of such cases. Four broad categories of metrics [4] are

1. metrics based on possible causality. It monitors how work moves among actors. Handover and Sub-contract are example of this case.

2. metrics based on joint cases. It gives an idea of actors working together for a particular case.

3. metrics based on joint activities. It focuses on actors working together irrespective of case. Similar-Task algorithm is an example of this case.

4. metrics based on special event types. It focuses on events like reassignment, transfer, etc. that is not a regular happening at a workplace.
In this paper, we implement Similar-Task algorithm and Sub-Contract Algorithm that are representative of two different metrics.

### 3.1 Similar-Task Algorithm

#### 3.1.1 Description

Similar-Task is sociogram metric based on joint activities. The idea is that individuals performing similar tasks are more closely related to each other than individuals performing different tasks. Similarity calculation could be achieved using Cosine-Similarity, Pearson Correlation Coefficient, Hamming Distance, etc. [4].

In this paper, we use Cosine-Similarity as a metric of measuring similarity of tasks between actors. The input to the algorithm is an Actor-Activity Matrix. Each row in
3.1 Similar-Task Algorithm

The actor-activity matrix represents an actor and each column indicates the frequency of each task performed by the actor. Table-3.2 shows an Actor-Activity Matrix for the event log shown in Table 3.1.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matt</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Britney</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Brad</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Joan</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>George</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3.2: Actor-Activity Matrix

3.1.2 Algorithm

The Similar-Task algorithm is given in Algorithm 1. Here each actor is compared with every other actor to compute the extent of similarity between them. The similarity values is collected in a 2-dimensional matrix. Table 3.3 gives similarity values between actors based on Algorithm-1.

**Algorithm 1:** Similar-Task Algorithm (Matrix M)

- **Data:** Actor-Activity Matrix (M)
- **Result:** Matrix-Similarity Values between Actors

1. Get the number of rows of M into \( m \).
2. Get the number of columns of M into \( n \).
3. \( D[m][m] \) = Declare square matrix to store results.
4. \textbf{foreach} \( i = 1 \) to \( m - 1 \) \textbf{do}
5. \( P=\text{Vector corresponding to } i^{\text{th}} \text{ row.} \)
6. \textbf{foreach} \( j = i + 1 \) to \( m \) \textbf{do}
7. \( Q=\text{Vector corresponding to } j^{\text{th}} \text{ row.} \)
8. Apply Cosine Similarity between \( i^{\text{th}} \) and \( j^{\text{th}} \) row

\[
\cos(P, Q) = \frac{P \cdot Q}{\| P \| \| Q \|}
\]  

(3.1)

9. Set \( D[i][j]=\text{similarity value obtained in the step above.} \)
3.2 Subcontract Algorithm

Description of some important steps of Similar-Task Algorithm is as follows:

1. The first step is to collect the number of Actors in the log. This is equal to number of rows in the Actor-Activity Matrix. Call it $m$.

2. The second step collects the number of distinct activities in the log. This is equal to number of columns in the Actor-Activity Matrix. Call it $n$.

3. The third step declares a 2D Matrix of size $m \times m$ to store the similarity result.

4. The eighth step calculates cosine-similarity between actors depicted by $P$ and $Q$ in Step-6 and Step-7 respectively.

5. The ninth step stores the similarity value between actor $P$ and $Q$.

<table>
<thead>
<tr>
<th></th>
<th>Matt</th>
<th>Britney</th>
<th>Brad</th>
<th>Joan</th>
<th>George</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matt</td>
<td>–</td>
<td>0.263</td>
<td>0.719</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Britney</td>
<td>–</td>
<td>–</td>
<td>0.298</td>
<td>0.671</td>
<td>0.00</td>
</tr>
<tr>
<td>Brad</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0.167</td>
<td>0.00</td>
</tr>
<tr>
<td>Joan</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0.00</td>
</tr>
<tr>
<td>George</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 3.3: Cosine-Similarity Values

3.2 Subcontract Algorithm

3.2.1 Description

Sub-Contract Algorithms tries to find out the number of times individual $j$ executes it’s task in between two activities performed by individual $i$. Considering causality($\beta$), direct/indirect succession($depth$) and multiplicity within cases can result upto eight different types of sub-contraction [4].

Each ProcessInstance refers to a CaseID; AuditTrailEntryList constitutes all the events pertaining to a particular CaseID; an AuditTrailEntry refers to an individual event [17]. An example of event log is shown in the Figure 3.1.

The algorithm for Sub-Contract is presented in Algorithm 2. Here for each Case identifiers, sub-contraction is detected if any.
3.2 Subcontract Algorithm

**Algorithm 2:** Subcontract Algorithm \((\beta, \text{depth}, \text{Len}, \text{Log})\)

**Data:** \(\beta, \text{depth}, \text{Len}, \text{Log}\)

**Result:** Normalized 2D Matrix \(D\) with subcontract values between Actors

1. Declare Square Matrix \(D\) of size \(\text{Len} \times \text{Len}\). Initialize all elements to 0
2. Declare and initialize variable \(\text{normal}\) to 0
3. \textbf{foreach} ProcessInstance \(pi\) in the log \textbf{do}

   - Get AuditTrailEntryList \(ates\) for \(pi\).
   - if \(\text{size}_{ates} < 3\) then
     - continue to the next ProcessInstance, \(pi\)
   - Declare and initialize \(\text{minK}\) to 0.
   - if \(\text{size}_{ates} < \text{depth}\) then
     - set \(\text{minK} = \text{size}_{ates}\)
   - else
     - set \(\text{minK} = \text{depth} + 1\).
   - if \(\text{minK} < 3\) then
     - set \(\text{minK} = 3\).
   - \textbf{foreach} \(k:=2\) to \(\text{minK}\) \textbf{do}

     - Update \(\text{normal}\) by \(\beta^{k-2}\).
     - \(m\) = Square matrix of \(\text{Len} \times \text{Len}\).
     - \textbf{foreach} \(i:=0\) to \(\text{size}_{ates} - k\) \textbf{do}
       - \(\text{ate}_i\) = get AuditTrailEntry at position \(i\).
       - \(\text{ate}_{ik}\) = get AuditTrailEntry at position \(i+k\)
       - if \(\text{Actor}_{ate_i} = \text{Actor}_{ate_{ik}}\) then
         - \textbf{foreach} \(j:=i+1\) to \(i+k\) \textbf{do}
           - \(\text{ate}_j\) = get AuditTrailEntry at position \(j\).
           - \(\text{row}\) = get row-position for \(\text{Actor}_{ate_i}\)
           - \(\text{col}\) = get column-position for \(\text{Actor}_{ate_j}\)
           - For valid \((\text{row}, \text{col})\) set \(m[\text{row}][\text{col}] = 1\).
     - \textbf{foreach} \(i:=0\) to \(\text{Len}\) \textbf{do}
       - \textbf{foreach} \(j:=0\) to \(\text{Len}\) \textbf{do}
         - set \(D[i][j] = D[i][j] + m[i][j]*\beta^{k-2}\).
   - Return \(\text{NormalizedMatrix}D\). //divide each value by \(\text{normal}\).
3.2 Subcontract Algorithm

\[
\beta \text{ is the causality factor that explains dependency of tasks which can be obtained using Process Model. } \text{depth} \text{ denotes the level of indirect succession between activities performed by same actor. } \text{minK} \text{ is the minimum distance between two activities performed by same actor. The number of distinct actors in the Log is denoted by } \text{Len}. \\
\]

The stepwise description of the algorithm is as follows:

1. In the first step, declare a 2-dimensional matrix of \( \text{Len} \times \text{Len} \) to store the results.

2. Step two initializes a variable \( \text{normal} \) which is used to normalize the subcontraction result. It is so because, direct succession between actors would always have higher contribution to the sub-contraction values than between actors involving indirect succession.
3.2 Subcontract Algorithm

<table>
<thead>
<tr>
<th></th>
<th>Matt</th>
<th>Britney</th>
<th>George</th>
<th>Brad</th>
<th>Joan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matt</td>
<td>0</td>
<td>0.22</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Britney</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>George</td>
<td>0.22</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Brad</td>
<td>0</td>
<td>0.22</td>
<td>0</td>
<td>0</td>
<td>0.22</td>
</tr>
<tr>
<td>Joan</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.4: Sub-contraction Values

3. Step three iterates over the process instances, $pi$ in the event log.

4. Step four gets all the AuditTrailEntry for the process instance, $pi$.

5. Step five checks if the number of entries is valid or not. For a $pi$ to be valid (or for chances of subcontraction existing in a $pi$), there has to be at least three activities in the $pi$.

6. Step eight to thirteen determines the minimum number of hops, $minK$ between two subcontracting actors.

7. Step fourteenth to twenty-fifth checks if there is/are any Actor(s) who execute their task in between the same Actor performing different tasks. If such Actor is found, then corresponding entry in the temporary matrix is set to 1.

8. Step twenty-sixth to twenty-eighth updates the resultant matrix $D$ after each iteration of valid $pi$.

9. Step twenty-ninth normalizes the resultant matrix $D$ by $normal$.

We consider indirect succession and multiple occurrences within a case, while ignoring causality. For example, considering events pertaining to Case1 in Table 3.1 (shaded rows), we have a sub-contraction between Matt and Britney. Matrix entry corresponding to Matt and Britney is updated in $m$ followed by an update in $D$. Final result shown in Table 3.4 is obtained after all such sub-contractions are identified from all cases in the event log.
4

Implementation of Algorithms on RDBMS

4.1 Similar-Task Algorithm

4.1.1 Pre-Processing

1. Create Table for importing data. *dataset* is the default table which if exists, has been dropped.

   ```sql
   DROP TABLE IF EXISTS dataset;
   CREATE TABLE dataset (ID int auto_increment PRIMARY KEY, Task varchar(50), Team varchar(20));
   ```

2. Load data from files into *dataset* table. Values not required in the current context are ignored using session variables.

   ```sql
   LOAD DATA LOCAL INFILE 'Dataset.csv' INTO TABLE dataset
   FIELDS TERMINATED BY ','
   IGNORE 1 lines
   (@id1,@id2,@id3,Task,Team,@id4,@id5);
   ```

3. The imported data is to be converted into an Actor-Activity Matrix (referred in the SQL code as AAMatrix). This step is embedded in the stored procedure `createTable` used to calculate initial similarity measure between the Originators.
4.1 Similar-Task Algorithm

4.1.2 Initial Similarity Calculation

1. CALL stored procedure `createTable`.

```sql
CREATE DEFINER='root'@'localhost' PROCEDURE 'createTable'(
    IN tblName varchar(20),
    IN tblName2 varchar(20))
BEGIN

    DECLARE taskname varchar(50) DEFAULT "";
    DECLARE exit_loop BOOLEAN;
    DECLARE qry LONGTEXT;
    DECLARE task_curs CURSOR FOR SELECT DISTINCT task FROM dataset;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET exit_loop = TRUE;
    SET @dropqry=CONCAT("DROP TABLE IF exists ",tblName);
    PREPARE stmt from @dropqry;
    EXECUTE stmt;
    SET qry = CONCAT("CREATE TABLE ",tblname," ( TEAM varchar(20) PRIMARY
    KEY ," ");
    SET @insertqry=CONCAT("INSERT INTO ",tblName," SELECT team");
    SET @insertSim=CONCAT("INSERT INTO ",tblName2," SELECT T1.TEAM, T2.
    TEAM,");
    SET @numerator="(";
    SET @denominator1="SQRT(";
    SET @denominator2="SQRT(";
    OPEN task_curs;
    my_loop: loop
        FETCH task_curs into taskname;
        IF exit_loop THEN
            CLOSE task_curs;
            LEAVE my_loop;
        END IF;
        SET qry=CONCAT(qry,',',taskname, ", INT DEFAULT 0, ");
        SET @insertqry=CONCAT(@insertqry, ", COUNT(IF(Task = ",taskname," 
        , 1, NULL))")
        @numerator=CONCAT(@numerator," T1.",taskname," * T2."
        taskname," +");
        SET @denominator1=CONCAT(@denominator1," T1.",taskname," * T1."
        ,taskname," +");
        SET @denominator2=CONCAT(@denominator2," T2.",taskname," * T2."
        ,taskname," +");
    END LOOP my_loop;
    SET qry=TRIM(TRAILING ', ' FROM qry);
    SET @insertqry=TRIM(TRAILING ', ' FROM @insertqry);
```
The procedure `createTable` fetches distinct Tasks from `dataset` table and uses it to define a new table that acts as Actor-Activity Matrix (AAMatrix). Also applied herein is the Cosine-Similarity measure between the rows of the AAMatrix where each row corresponds to a different actor.

Joins are a way to find relationships in MySQL. Similarly in this case too, joins were required to find similar actors. Self-joins were imposed on the table `dataset` to find similarity values between the actors. The self-join is implemented in the stored procedure `createTable` as part of `insertSim` query.
4.1 Similar-Task Algorithm

4.1.3 Final Similarity Calculation

1. CALL stored procedure `insertData`.

```sql
CREATE DEFINER='root'@'localhost' PROCEDURE 'insertData' (IN tblName varchar(20), IN tblName2 varchar(20))
BEGIN

DECLARE teamname varchar(20) DEFAULT "";
DECLARE teamexit_loop BOOLEAN;
DECLARE qrys LONGTEXT;
DECLARE team_curs CURSOR FOR SELECT DISTINCT Team FROM dataset;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET teamexit_loop = TRUE;
SET @dropqrys=CONCAT("DROP TABLE IF exists ",tblName);
PREPARE stmt from @dropqrys;
EXECUTE stmt;
SET qrys = CONCAT("CREATE TABLE ",tblname," ( SOURCE_TEAM varchar(20) PRIMARY KEY,");
SET @finalinsert=CONCAT("INSERT INTO ",tblname," SELECT Source, ");
OPEN team_curs;
team_loop: loop
    FETCH team_curs into teamname;
    IF teamexit_loop THEN
        CLOSE team_curs;
        LEAVE team_loop;
    END IF;
    SET qrys=CONCAT(qrys,teamname, ' DOUBLE(10,8) DEFAULT 0.00, ');
    SET @finalinsert=CONCAT(@finalinsert," GROUP_CONCAT(if(Target = '",teamname,"', Similarity, NULL)),");
END LOOP team_loop;
SET qrys=TRIM(TRAILING ', ' FROM qrys);
SET @finalinsert=TRIM(TRAILING ', ' FROM @finalinsert);
SET @qrys1=CONCAT(qrys, ");
SET @finalinsert=CONCAT(@finalinsert," FROM ",tblName2," GROUP BY Source;";
PREPARE stmt2 from @qrys1;
EXECUTE stmt2;
PREPARE stmt3 from @finalinsert;
EXECUTE stmt3;
DEALLOCATE PREPARE stmt;
DEALLOCATE PREPARE stmt2;
DEALLOCATE PREPARE stmt3;
END
```
The stored procedure `insertData` takes values from the table that stores initial similarity in the procedure `createTable`. It populates the final table with the similarity values and stores it in the form of a matrix.

The procedure `insertData` thus brings an end to the Similar-Task Algorithm implementation in SQL. The code is a generic implementation provided the initial table where the records are imported is `dataset` and does not exists in the current database. It is dropped if present.

## 4.2 Subcontract Algorithm

### 4.2.1 Native SQL Implementation

#### 4.2.1.1 Pre-Processing

1. Create Table for importing data. `dataset` is the default table which if exists, has been dropped.

   ```sql
   DROP TABLE IF EXISTS dataset;
   CREATE TABLE dataset (ID INT AUTO_INCREMENT PRIMARY KEY, CaseID VARCHAR(20), Activity VARCHAR(100), Actor VARCHAR(20));
   ```

2. Load data from files into `dataset` table. Values not required in the current context are ignored using session variables.

   ```sql
   LOAD DATA LOCAL INFILE 'Dataset.csv' INTO TABLE dataset
   FIELDS TERMINATED BY ','
   IGNORE 1 lines
   (CaseID,@id2,@id3,Activity,Actor,@id4,@id5);
   ```

The imported data has to be processed such that events corresponding to a particular caseID are together and in the same order as they were imported. The auto incrementing key ID for the table `dataset` is used for ordering the records/events in the said fashion. The data is inserted into a newly created table `organiseddata` which has the same structure as the original `dataset`. Also a secondary index `caseindex` is created on columns CaseID, Actor, Activity and ID for better efficiency.
4.2 Subcontract Algorithm

3. Create table `organiseddata` with same structure as `dataset`.

```sql
CREATE TABLE organiseddata LIKE dataset;
```

4. Populate `organiseddata` with records from `dataset`.

```sql
INSERT INTO organiseddata(CaseID,Activity,Actor) SELECT CaseID, Activity, Actor FROM dataset ORDER BY ID, CaseID;
```

5. Create secondary index on columns of `organiseddata`.

```sql
CREATE INDEX caseindex ON organiseddata(CaseID, Actor, Activity,ID);
```

6. Define a global variable `normal`. This variable would be used to normalize the sub-contraction values obtained after the complete execution of the stored procedure `ExecuteCase`.

```sql
SET @normal = 0.0;
```

7. Define `\beta`. `\beta` is used to take into consideration the dependency on activities that the sub-contracting actors performs. This dependency can be obtained from the process model obtained by applying any Process Mining algorithm like \( \alpha \)-Algorithm.

```sql
SET @beta = 0.5;
```

Since causality or dependency on activities is not considered here, the value of \( \beta \) has been set to default value of 0.5. However, more appropriate values can be used for \( \beta \) as and when some process mining algorithm is implemented.

4.2.1.2 Create Initial Matrix

1. CALL stored procedure `createMatrix`.

```sql
CREATE DEFINER='root'@'localhost' PROCEDURE 'createMatrix'(IN matrixName varchar(20))
BEGIN
DECLARE distinctActor varchar() DEFAULT "";
DECLARE distinctActorExit_Loop BOOLEAN;

DELIMITER $%

CALL createMatrix('organiseddata');

DELIMITER ;
```

---

93 CREATE DEFINER='root'@'localhost' PROCEDURE 'createMatrix'(IN matrixName varchar(20))
94 BEGIN
95 DECLARE distinctActor varchar() DEFAULT "";
96 DECLARE distinctActorExit_Loop BOOLEAN;

---

29
The procedure `createMatrix` takes in all distinct actors and builds a table (matrix). This matrix tracks progress of the sub-contraction values between originators.

### 4.2.1.3 Get Distinct Cases.

1. CALL stored procedure to `getDistinctCaseIDs`.

```sql
CREATE DEFINER='root'@'localhost' PROCEDURE `getDistinctCaseIDs`(IN matrixName varchar(20))
BEGIN
    DECLARE distinctCaseName varchar(20) DEFAULT "";
    DECLARE totalCases INT DEFAULT 0;
    DECLARE distinctCaseExit_Loop BOOLEAN;
```
DECLARE distinctcase_curs CURSOR FOR SELECT DISTINCT CaseID, COUNT(*)
FROM organiseddata GROUP BY CaseID HAVING COUNT(CaseID)>=3;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET distinctcaseexit_loop = TRUE;
OPEN distinctcase_curs;
distinctcase_loop: loop
  FETCH distinctcase_curs into distinctCaseName, totalCases;
  IF distinctcaseexit_loop THEN
    CLOSE distinctcase_curs;
    LEAVE distinctcase_loop;
  END IF;
  CALL ExecuteCase(distinctCaseName, totalCases, matrixName);
END LOOP distinctcase_loop;

Distinct Cases are collected in this procedure. For each such case, another procedure ExecuteCase is called which detects sub-contraction.

4.2.1.4 Find Subcontraction.

1. CALL stored procedure ExecuteCase.

CREATE DEFINER='root'@'localhost' PROCEDURE 'ExecuteCase'(INOUT currCase varchar(20), IN caseCount INT, IN matrixName varchar(20))
BEGIN
  DECLARE CommActor_ID1 INT DEFAULT 0;
  DECLARE CommActor_ID2 INT DEFAULT 0;
  DECLARE Diff INT DEFAULT 0;
  DECLARE CommActor_Name varchar(20) DEFAULT ""
  DECLARE CommActorExit_Loop BOOLEAN;
  DECLARE CASE_FLAG INT;
  DECLARE K INT;
  DECLARE CommActor_curs CURSOR FOR SELECT T1.ID, T2.ID, (T2.ID-T1.ID), T1.Actor
FROM organiseddata as T1
JOIN organiseddata as T2
ON T2.ID>=T1.ID+2
AND T1.Actor=T2.Actor
AND T1.Activity <> T2.Activity
AND T1.CaseID = currCase

FROM organiseddata as T1
JOIN organiseddata as T2
ON T2.ID>=T1.ID+2
AND T1.Actor=T2.Actor
AND T1.Activity <> T2.Activity
AND T1.CaseID = currCase
AND T2.CaseID = currCase
WHERE T1.CaseID=currCase ORDER BY Diff ASC;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET CommActorExit_Loop = TRUE;
OPEN CommActor_curs;
SET CASE_FLAG=0;
SET K=2;
CommActor_Loop: LOOP
  FETCH CommActor_curs INTO CommActor_ID1,CommActor_ID2,Diff,
                  CommActor_Name;
  IF CommActorExit_Loop THEN
    CLOSE CommActor_curs;
    LEAVE CommActor_Loop;
  END IF;
  WHILE(K<caseCount) DO
    SET @normal=@normal+POW(@beta,K-2);
    SET K=K+1;
  END WHILE;
  IF(CASE_FLAG<>Diff) THEN
  END IF;

BLOCK2: BEGIN
  DECLARE InBetweenActors_ID INT DEFAULT 0;
  DECLARE InBetweenActors_Name varchar(15) DEFAULT "";
  DECLARE InBetweenActorExit_Loop BOOLEAN;
  DECLARE InBetweenActor_curs CURSOR FOR SELECT ID, Actor FROM
      organiseddata WHERE ID> CommActor_ID1 AND ID < CommActor_ID2;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET
      InBetweenActorExit_Loop = TRUE;
  OPEN InBetweenActor_curs;
  InBetweenActor_Loop: LOOP
    FETCH InBetweenActor_curs INTO InBetweenActors_ID,
                 InbetweenActors_Name;
    IF InBetweenActorExit_Loop THEN
      CLOSE InBetweenActor_curs;
      LEAVE InBetweenActor_Loop;
    END IF;
    SET @tempQuery = "INSERT INTO";
This is the heart of Sub-Contract Algorithm. For each case, all sub-contraction are detected. normal is updated for each case and final values are updated in the table.

Alike Similar-Task algorithm, joins were also necessary to find sub-contracting actors. For each case in the event log, joins were applied to find any sub-contracting actors, if any. A prior join would have required to store the huge result. But joining on case basis did although not require storing the results, but is compute intensive task. It is so because, results had to fetch segregated for each case and then join be applied with the required condition(s).

### 4.2.1.5 Normalize the Final Matrix

1. CALL stored procedure `normalizedMatrix`.

   ```sql
   CREATE DEFINER='root'@'localhost' PROCEDURE 'normalizedMatrix'(in finalMatrixName VARCHAR(25))
   BEGIN
   DECLARE normalizedCols varchar(20) DEFAULT ";
   DECLARE normalizedColsExit_Loop BOOLEAN;
   DECLARE normalizedCols_curs CURSOR FOR SELECT DISTINCT Actor FROM organiseddata;
   DECLARE CONTINUE HANDLER FOR NOT FOUND SET normalizedColsExit_Loop = TRUE;
   DECLARE normalizedMainTableQuery="UPDATE ";
   SET @normalizedMainTableQuery=CONCAT(@normalizedMainTableQuery,finalMatrixName, ", SET ");
   ```
4.2 Subcontract Algorithm

```
normalizedCols_Loop: LOOP
    FETCH normalizedCols_curs INTO normalizedCols;
    IF normalizedColsExit_Loop THEN
        CLOSE normalizedCols_curs;
        LEAVE normalizedCols_Loop;
    END IF;
    SET @normalizedMainTableQuery=CONCAT(@normalizedMainTableQuery,
        normalizedCols," = ",normalizedCols,"/",@normal," ");
END LOOP normalizedCols_Loop;
SET @normalizedMainTableQuery =TRIM(TRAILING ', ' FROM
    @normalizedMainTableQuery);
SET @normalizedMainTableQuery=CONCAT(@normalizedMainTableQuery,"' FROM
    @normalizedMainTableQuery);
PREPARE normalizedMainTableQuery_Stmt from
    @normalizedMainTableQuery;
EXECUTE normalizedMainTableQuery_Stmt;
DEALLOCATE PREPARE normalizedMainTableQuery_Stmt;
END
```

normalizedMatrix is the final step in the algorithm. Here the final table obtained at the end of ExecuteCase is normalized by normal.

4.2.2 Memoization

Memoization is an optimisation technique in computer programming where results are cached to avoid any sort of functions calls happening again. The approach involves storing intermediate results that can be used for other following calculations. Memoization is commonly a used approach in Dynamic Programming Paradigm.

4.2.2.1 Bottleneck

Here, though not used in the same context as memoization is generally used, it has been found that storing intermediate results rather than pushing them immediately to databases tables caused a bottleneck in disk I/O. It was observed that even in the execution of the sub-contract SQL implementation on a dataset of 65000 records, there were over ninety lacs (and counting) number of intermediate transactions.

1. Joins. Joins are usually considered to be an expensive operations in databases. The below code which forms an integral part of the the stored procedure ExecuteCase achieves the required join.
4.2 Subcontract Algorithm

```
SELECT T1.ID, T2.ID, (T2.ID-T1.ID), T1.Actor
FROM organiseddata as T1
JOIN organiseddata as T2
ON T2.ID>=T1.ID+2
AND T1.Actor=T2.Actor
AND T1.Activity <> T2.Activity
AND T1.CaseID = currCase
AND T2.CaseID = currCase
WHERE T1.CaseID=currCase ORDER BY Diff ASC;
```

Here joins are performed for each CaseID to detect sub-contraction between actors, if any. However the join was called for for each CaseID present in the dataset and that increased the number of procedural calls to a large extent. Another approach to calculate join only once and then get results for each CaseID was not considered because of the amount of memory that would be required to store the join result of a dataset would be massive.

2. **Update Database Tables.** The major contributing factor for the bottleneck was not join though. It was the amount of writes to the tables that needed to be performed. The following code in the stored procedure `ExecuteCase` was the main reason for sloppy performance.

```
INSERT INTO matrixName(Actor,InBetweenActors_Name) VALUES (?,?) ON
DUPLICATE KEY UPDATE InBetweenActors_Name = InBetweenActors_Name
+POW(@beta","Diff","-2);
```

### 4.2.2.2 Improved alternative approach

The alternative approach comprised of achieving the same task but by caching the intermediate results.
4.2 Subcontract Algorithm

Algorithm 3: Outline of Similar-Task Algorithm

1. Get distinct CaseID from organised data.
2. Get distinct Actors from organised data.
3. Assign unique identifier to each Actor.
4. \( m = \) Get number of distinct actors.
5. Declare ResultMatrix of size \( m \times m \) to store the sub-contraction result.
6. \textbf{foreach} CaseID \( ci \) \textbf{do}
   7. ResultSet \( rs = \) Declare a ResultSet to collect results.
   8. \( rs = \) Get join result for \( ci \).
   9. \textbf{foreach} record \( r \) in \( rs \) \textbf{do}
      10. Identify sub-contracting actors. Let the sub-contraction be from Actor with unique identifier \( i \) to Actor with unique identifier \( j \).
      11. Set ResultMatrix\([i][j]\) = ResultMatrix\([i][j]\) + \( \beta^{n-2} \). // The caching happens here
12. \textbf{foreach} row \( \text{ResultMatrix} \) \textbf{do}
      13. Create and insert the entire row back to the database table.

Rather than updating tables after every iteration, the values were updated once and only once. All intermediate results were cached in the ResultMatrix matrix. The implementation was bridged through java programming interface. The procedure of the improved implementation is given in Algorithm 3. However this may not be taken as an alternative to original sub-contract concept defined in [17].

The implementation of the above algorithm was found to perform extensively better than the native SQL code. It was so because of the memoization of intermediate results at Step 11 of the algorithm. The remaining steps were performed in the same manner as the native SQL implementation.
5

Implementations of Algorithms on Neo4j- NoSQL

5.1 Similar-Task Algorithm

5.1.1 Schema Definition

Defining Schema is an important task in Neo4j. Although they are basically vertices and edges but maintaining proper attributes for proper and quick execution of the algorithm is a challenging aspect. Figure 5.1 below depicts the schema used for Similar-Task algorithm. All nodes are unique and each store the count of their occurrences. [:PERFORMS] connects ACTOR to ACTIVITY. It also has a property times that stores the frequency of the ACTIVITY performed by that ACTOR.

5.1.2 Native CYPHER Implementation

(a) Load data from file.

```
LOAD CSV with HEADERS FROM 'Dataset.csv' AS line
FIELDTERMINATOR ‘;’
MERGE (a:Actor {name: line.Team, count: 0})
MERGE (b:Activity {work: line.Task, count: 0})
CREATE UNIQUE a-[rel:performs{times: 0}]-b
SET rel.times=rel.times+1
SET a.count=a.count+1
```
5.1 Similar-Task Algorithm

Importing the data was a crucial task as it had to make sure that no duplicates nodes were found. But if they are found, then their count be incremented accordingly rather than adding new nodes and further relationship. This was accomplished using MERGE function.

(b) Find Intersection of common task and calculate Cosine-Similarity.

```
MATCH (p1:Actor)-[x:PERFORMS]->(m:Activity)<=-[y:PERFORMS]-(p2:Actor)
WITH SUM(x.times * y.times) AS xyDotProduct,
    SQRT(REDUCE(xDot = 0.0, a IN COLLECT(x.times) | xDot + a^2)) AS xLength,
    SQRT(REDUCE(yDot = 0.0, b IN COLLECT(y.times) | yDot + b^2)) AS yLength,
    p1, p2
MERGE (p1)-[:SIMILARITY]->(p2)
SET s.similarity = xyDotProduct / (xLength * yLength)
```

This step does the purpose of for loops in the algorithm. Accomplishing the for loop for finding intersecting tasks between originators was a matter or selecting MATCH query. These intersected list were then used for calculating the similarity based on their property values.
5.2 Subcontract Algorithm

5.2.1 Schema Definition

Like Similar-Task algorithm, defining schema for Sub – Contract algorithm was equally important. Here each CASE had to be taken care of and within each case, various ACTOR performing various activities played an important role. As such, the ACTOR nodes were redundant whereas the CASE nodes were maintained unique. CASE node only contained a Name property whereas ACTOR node contained properties like Name, OccID and Activity. It denotes that an ACTOR with Name has occurred at position OccID within that Case and performs an activity. The CASE node is connected to the ACTOR node via the [: CONTAINS] relationship. The organization of nodes is shown in the Figure 5.2.

Once such a setup of nodes is traversed, sub-contraction is identified within each case. The contributing ACTOR nodes are then connected by [:RELATED_TO] relationship with properties ‘value’ always set to 1 and another property ’length’ set to the distance between the actor whose occurrence was repeated.

In the final step, the start and end ACTOR nodes of each and every [:RELATED_TO] relationship are found out and sub-contraction values are determined and as-
5.2 Subcontract Algorithm

signed between the UNIQUEACTOR nodes corresponding to the start and end ACTOR node found above. The UNIQUEACTOR nodes are connected by [: SUBCONTRACT] relationship with a property 'strength' denoting the sub-contraction value between them. The 'strength' property are finally normalised by normal to remove any bias that may have crept up because of direct and indirect succession.

5.2.2 Native CYPHER Implementation

5.2.2.1 Identify Sub-contracting Actors

```
MATCH (n:CASE)
MATCH commActorPath=(Actor1)<-(n)-->(Actor2)
WHERE Actor1.name = Actor2.name
AND Actor1.OccID - Actor2.OccID >= 2
AND Actor1.activity <> Actor2.activity
WITH commActorPath,n, (Actor2.OccID - Actor1.OccID) as sepDist
WITH RANGE(head(nodes(commActorPath)).OccID+1, last(nodes(commActorPath)).OccID-1) as intermediateIDs,
    n,
    head(nodes(commActorPath)).OccID as startID,
    sepDist
UNWIND intermediateIDs as endID
MATCH (person1:PERSON {OccID:startID})<-(n)-->(person2:PERSON {OccID:endID})
MERGE (person1)-[:RELATED_TO {value:1, length:sepDist}]->(person2)
```

The code iterates case-wise and within each case, sub-contraction between actors are detected according to the criteria. For each such pair, their sub-contraction value is set to 1 and length is used to specify the distance between actor nodes (same actor performing different tasks) responsible for that sub-contraction.

5.2.2.2 Collect distinct Actors

```
MATCH (n:ACTOR)
WITH collect(DISTINCT (n.name)) AS distinctNames
```
5.2 Subcontract Algorithm

It creates a new set of distinct nodes UNIQUEACTOR from the existing graph. Their uniqueness is identified by their names.

5.2.2.3 Set Sub-contraction values

It takes values between actors from relation :RELATED_TO identified in step 5.2.2.1 to calculate sub-contraction values between distinct users found in step 5.2.2.2.

5.2.2.4 Calculate normal

It calculates the value of normal. The same is required for normalization of sub-contraction values obtained.

5.2.2.5 Normalize the result
5.2 Subcontract Algorithm

The code normalizes the sub-contraction values between actors.

5.2.3 Memoization

Yet again, the native CYPHER implementation of sub-contract algorithm suffered from having to match large graphs and making them available in the memory at runtime. So, memoization was used yet again to store the references of nodes and storing the intermediate properties.

5.2.3.1 Bottleneck

Neo4j requires nodes, relationships and properties to be present in memory during query execution. Thus, this can be a bottleneck in system where memory configuration is limited. Also Neo4j nodes take 14 bytes, relationships take 33 bytes, properties take 41 bytes besides other memory requirements like index creation, etc. So for considerably larger dataset, the amount of memory may become a bottleneck.

(a) MATCH Query. MATCH are usually used when a reference of a node is to be obtained. It can also be used to map to a set of relationships or properties, which may be quite large. In the native CYPHER implementation, the following MATCH queries forms a bottleneck.

- MATCH query in procedure Identify Sub-contracting actors.

\[
\text{MATCH (n:CASE)} \\
\text{MATCH commActorPath=(Actor1)<--(n)-->(Actor2)}
\]

The above code is equivalent to finding each CASE node in the graph and then for each CASE nodes, identify the sub-contracting ACTOR nodes. So, the entire graph is loaded into the memory at any point of time which results in bottleneck.

- MATCH query in procedure Set Sub-Contraction values.
5.2 Subcontract Algorithm

MATCH (n:CASE)
MATCH (n)-[:CONTAINS]->()-[r:RELATED_TO]->()-[:CONTAINS]-(n)

It identifies all subcontracting actors within each case. So in the worst case, the entire graph may again be loaded into the memory. This again is another factor for bottleneck besides the one found above in the code above.

(b) Properties Properties are means of storing information. Properties can be found on nodes and relationships. The properties which contributes heavily to higher memory usage are:

- **Set Properties** in procedure Identify Sub-contracting actors.

MERGE (person1)-[:RELATED_TO {value:1, length:sepDist}]->(person2)

The properties `value` and `length` are updated for every sub-contraction found in the graph. This can be equivalent to setting properties for a complete graph of all unique actors and for each case. So setting properties can result in \(O(n^3)\) computation time and a space complexity of loading the entire graph into the memory.

- **Set Properties** in procedure Set Sub-Contraction values.

SET rf.strength = CASE WHEN rf.strength IS NULL THEN r.value ELSE rf.strength + (0.5^(l-2))*r.value END

This again has the effect of loading a major part of the graph in the memory.

5.2.3.2 Improved alternative approach

Having found the bottlenecks, the improved implementation stores the reference of each nodes on creation. Also instead of setting properties after each iteration, the results are again stored in a matrix and then the final properties are set using references of the nodes. The algorithmic procedure uses Neo4j API and caches the intermediate results. The procedure is shown below.
Algorithm 4: Outline of Sub-Contract Algorithm

1. Create distinct CASE nodes and store their reference.
2. \textbf{foreach} CASE node \textbf{do}
   3. Create ACTOR node with occurrence ID (OccID) of the node, name and activity as properties.
   4. Store reference of ACTOR node along with reference of CASE node.
   5. Create UNIQUEACTOR node for the above ACTOR (if not already created) and store its reference.

6. \textbf{m} = Get number of UNIQUEACTOR nodes.
7. Declare ResultMatrix of size \textbf{m}\*\textbf{m} to store the sub-contraction result.
8. \textbf{foreach} CASE node \textbf{cn} \textbf{do}
   9. Identify sub-contracting ACTORS. Let those two ACTORS correspond to UNIQUEACTOR \textbf{i} and UNIQUEACTOR \textbf{j}.
   10. Set ResultMatrix[\textbf{i}][\textbf{j}] = ResultMatrix[\textbf{i}][\textbf{j}] + \beta^{\textbf{n}-2}.

11. \textbf{foreach} \textbf{p} := 0 to \textbf{m} \textbf{do}
   12. \textbf{foreach} \textbf{q} := 0 to \textbf{m} \textbf{do}
   14. Set rf.'strength' = ResultMatrix[\textbf{p}][\textbf{q}].

The terminologies can be best understood by studying the Schema Description given in section 5.2.1. Meanwhile, the results drastically improved on memoizing references of nodes. This resulted in $O(1)$ seek time for references rather than searching for the node in the whole graph setup. Also avoiding frequent property writes by simply setting properties at the end lessened memory requirements and thus gave better performance.
Experimental Dataset

We use Business Process Intelligence 2014 (BPI 2014) dataset to conduct our experiments. The log contains events from an incident and problem management system called of Rabobank Group ICT. The dataset selected for the purpose of this study is Detail Incident Record. The data is related to the process of managing requests from Rabobank customers. These requests may be in the form of mail or call. The process describes how a request is handled in the Service Department operated by Rabobank Group ICT. The dataset is provided in CSV format. We use the Detail Incident Record which contains 466737 records to conduct our experiments. The various fields in the dataset can be explained as:

(a) **Incident ID**: The unique ID of an Incident-record in the Service Management tool.

(b) **DateStamp**: Date and time when this specific Incident Activity started

(c) **Incident Activity Number**: Unique ID for an Incident Activity.

(d) **Incident Activity Type**: Short code to identify which type of Incident Activity took place

(e) **Interaction ID**: The unique ID of an Interaction-record in the Service Management tool.

(f) **Assignment Group**: The team responsible for this Incident Activity.

(g) **KM Number**: A Knowledge Document contains default attribute values for the Interaction-record and a set of questions for a Service Desk Agent to derive which Configuration Item is disrupted and to determine Impact and Urgency for the customer.
Figure 6.1 shows the dataset that is used in our experimentation. Since Organizational Mining algorithm is concerned only with the relationships between performer of activities, and we need find similarity between actors, so we consider only two fields - IncidentActivity_Type which represents the activity performed and Assignment_Group which represents the performer of the activity. Also for sub-contraction algorithm, we consider three fields Incident_ID which represents the identifier for various cases, IncidentActivity_Type which is the nature of the work performed and Assignment_Group which represents the performer of the activity. All these used columns are highlighted in Figure 6.1.

Figure 6.2: Number of Events per Case
Fig. 6.2 shows the graph between the total number of cases and the number of events for each case. As can be seen from the figure, the total number of records in the event log are 466737, the total number of cases i.e. process instances are 46616 and the total number of activities in the event log are 39. The frequency of some of the most frequent cases is shown in Figure 6.3.

<table>
<thead>
<tr>
<th>Case ID</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM0000428</td>
<td>178</td>
</tr>
<tr>
<td>IM0000382</td>
<td>175</td>
</tr>
<tr>
<td>IM0004038</td>
<td>171</td>
</tr>
<tr>
<td>IM000220</td>
<td>170</td>
</tr>
<tr>
<td>IM005837</td>
<td>165</td>
</tr>
<tr>
<td>IM014171</td>
<td>156</td>
</tr>
<tr>
<td>IM000078</td>
<td>149</td>
</tr>
<tr>
<td>IM000515</td>
<td>146</td>
</tr>
<tr>
<td>IM004400</td>
<td>142</td>
</tr>
<tr>
<td>IM000391</td>
<td>141</td>
</tr>
<tr>
<td>IM0008030</td>
<td>138</td>
</tr>
<tr>
<td>IM000088</td>
<td>137</td>
</tr>
<tr>
<td>IM000080</td>
<td>130</td>
</tr>
</tbody>
</table>

**Figure 6.3:** Top Cases ordered by number of Events

Also, since the algorithm had to define and find relationships between Actors taking into consideration the activities that they perform, so presented in Figure 6.4 is a graphical representation of the number of actors and the frequency of their occurrence. Figure 6.5 presents the same information for some of the actors who have higher presence in the log. Figure 6.4 and 6.5 shows that there were actors that performed only once in the entire log. On the other hand, some actors were as frequent as 84143.

**Figure 6.4:** Frequency of Actors
Activities also formed a major data for both the algorithms. For Similar-Task algorithm, these activities were stored as a node, whereas in Sub-Contract algorithm, these activities were embedded into actor nodes as property. Hence, it is important to have an insight into the number of activities in the event log. Figure 6.6 gives a relative distribution of the activities in the event log. Also included in Figure 6.7 are some of the activities with higher frequency.

As can be seen from Figure 6.6 and Figure 6.7 that frequency of activities varied from 2 to 88502.

This brings forward the amount of data that needs to be available for any Neo4j script execution. Since one node takes 14 bytes, relationships take 33 bytes and properties take 41 bytes, so the amount of memory needed for proper execution may increase with increase in the number of elements of property graph. Prior
Figure 6.7: Highest Frequency Activities

analysis of the event log proved useful during program execution.
Performance Comparison

The implementations were tested on our benchmarking system. The system had 64-bit Windows OS which ran on top of Intel Core-2 Duo processor. The system had 4GB of RAM and 3MB cache and backed up by 320GB of secondary memory. The values recorded are an average of multiple runs. The experiments were conducted on single node MySQL and single node Neo4j instance of the database.

7.1 Similar-Task Algorithm

Implementation of Similar-Task algorithm required the actor-activity matrix as an input. The dataset was broken into different sizes. Since number of unique actors play an important role in Similar-Task algorithm, dataset of various sizes are compared to find the number of unique actors. The result is shown in Table 7.1.

Table 7.2 shows the time taken to load datasets of different sizes on MySQL and Neo4j database and for a single node setup. The time taken is directly related to the number of unique actors present in each dataset size. For each dataset size,

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>Unique Actors</th>
</tr>
</thead>
<tbody>
<tr>
<td>65000</td>
<td>150</td>
</tr>
<tr>
<td>1,01,000</td>
<td>158</td>
</tr>
<tr>
<td>2,19,500</td>
<td>220</td>
</tr>
<tr>
<td>3,00,000</td>
<td>229</td>
</tr>
<tr>
<td>4,66,731</td>
<td>242</td>
</tr>
</tbody>
</table>

**Table 7.1:** Number of Unique Actors per dataset size.
7.1 Similar-Task Algorithm

the data was initially imported and actor-activity matrix (tables in MySQL and relationships in Neo4j) being created. It is observed that both the databases gave similar performance. However with increase in number of unique actors, Neo4j started giving better load time performance. Figure 7.1 shows the data load time

<table>
<thead>
<tr>
<th>Unique Actors</th>
<th>Load Time (msec)</th>
<th>MySQL</th>
<th>Neo4j</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>2467</td>
<td>3413</td>
<td></td>
</tr>
<tr>
<td>158</td>
<td>2875</td>
<td>3362</td>
<td></td>
</tr>
<tr>
<td>220</td>
<td>5966</td>
<td>4354</td>
<td></td>
</tr>
<tr>
<td>229</td>
<td>5850</td>
<td>5877</td>
<td></td>
</tr>
<tr>
<td>242</td>
<td>7819</td>
<td>6875</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.2: Data Load Time (Similar – Task)

comparison on a graph plotted for various dataset sizes whose corresponding values can be referred from Table 7.2.

Figure 7.1: Data Load Time for Similar-Task Algorithm

The reason Neo4j performs better is down to the fact that only unique Actor and Activity nodes are imported into the graph setup. On the other hand, MySQL had a predefined schema equivalent to the number of distinct activities in the dataset. So even if an actor had not performed an activity, value (though zero) had to be set at that respective column. This was easily avoided in case of Neo4j.
7.1 Similar-Task Algorithm

The heart and soul of Similar-Task algorithm is similarity calculation (Step-8 of the algorithm) and update the result table (Step-9 of the algorithm). Table 7.3 records the time taken to execute Step-8 and Step-9 of Similar-Task Algorithm as a function of the number of unique actors for different dataset size as given in Table 7.1.

<table>
<thead>
<tr>
<th>Unique Actors</th>
<th>Execution Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Step-8</td>
</tr>
<tr>
<td>150</td>
<td>225</td>
</tr>
<tr>
<td>158</td>
<td>372</td>
</tr>
<tr>
<td>220</td>
<td>713</td>
</tr>
<tr>
<td>229</td>
<td>903</td>
</tr>
<tr>
<td>242</td>
<td>1403</td>
</tr>
</tbody>
</table>

Table 7.3: Execution Time for Step-8 and Step-9 (Similar – Task)

It can be observed that calculation based on node properties and relationship properties are time taking as compared to retrieving from tables. Also Step-9 comparison shows that \textit{INSERT} in MySQL performs better than \textit{SET} properties in Neo4j. Figure 7.2 presents execution-time comparison for cosine-similarity calculation.

The difference in execution time for two different implementations is quite large. This is so because, data for cosine similarity calculation in MySQL was readily available in tables. No computation had to be performed to get data as those computations were done as part of pre-processing step where Actor-Activity Matrix was carved out of initially imported dataset. Using some sort of similar matrix in Neo4j would have defeated the whole purpose of graph database.

Equivalent values of the form of Actor-Activity Matrix were incorporated into relationship properties in Neo4j. So for any computation, Neo4j requires matching intersecting activities between the two actors in concern, followed by the actual computation. Also Figure 7.3 gives an idea of the time required to update results.

It can be observed that setting relationship properties in Neo4j is more time consuming because existing relationships needed to be merged with the updated properties or new ones be created if such relationship doesn’t exist.
7.1 Similar-Task Algorithm

Figure 7.2: Cosine-Similarity calculation in Step-8

Table 7.4 and 7.5 shows the disk space taken by tables (MySQL) or nodes, properties, relationships (Neo4j). These data includes all intermediate memory consumption, indexes besides initial data loading and results. Figure 7.4 and 7.5 shows the disk space usage for various elements of MySQL and Neo4j.

<table>
<thead>
<tr>
<th>Tables</th>
<th>Dataset Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>65000</td>
</tr>
<tr>
<td>Dataset</td>
<td>3686400</td>
</tr>
<tr>
<td>OTMatrix</td>
<td>65536</td>
</tr>
<tr>
<td>InitSim</td>
<td>1589248</td>
</tr>
<tr>
<td>FinalSim</td>
<td>229376</td>
</tr>
</tbody>
</table>

Table 7.4: Disk Space Usage (bytes) for MySQL tables (Similar – Task)

Table 7.4 indicates the various tables sizes. The table Dataset is where the initial data is imported. From the imported data, Actor-Activity Matrix (referred herewith as OTMatrix) is constructed. Tables InitSim and FinalSim stores the initial similarity values and final similarity values respectively.

Table 7.5 shows disk space taken by graph elements. The results shows that Neo4j
gains upperhand when it comes to disk space usage. These can be attributed to the fact that Neo4j nodes and relationships are only created when needed unlike MySQL which has to have a proper schema defined beforehand.

### 7.2 Sub Contract Algorithm

Table 7.6 shows the time taken to load datasets of different sizes on MySQL and Neo4j database and for a single node setup, we observe that both the databases gave similar performance. However with increase in dataset size, Neo4j started giving better load time performance.

Figure 7.6 shows the data load time comparison on a graph plotted for various dataset sizes whose corresponding values can be referred from Table 7.6.
The data load time in MySQL is the summation of loading the dataset from the file into initial table \textit{dataset} and inserting them into second table \textit{organiseddata} with all records being ordered by their \textit{ID} and \textit{CaseID}. This is done to bring uniformity in load time statistics because loading data in Neo4j, by default means incorporating these values on graph elements like relationships and properties.

Alike Similar-Task algorithm, data load time exhibited the same pattern in Sub-Contract algorithm too. However it must be noted that no unique actors is of concern here because sub-contraction can only be calculated within cases. With increase in dataset size, Neo4j started giving better load time. This is due to
the fact that with increase in dataset size, ordering in MySQL takes longer as compared to creating straight forward relationships in Neo4j.

Table 7.7 shows the execution time of Sub-Contract algorithm using the improvised alternative approach. As can be seen from Table 7.7, MySQL gave almost identical performance for various dataset sizes.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>Update Normal</th>
<th>Sub-Contract Detection</th>
<th>Update Result</th>
<th>Normalize Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>65,000</td>
<td>32</td>
<td>11712</td>
<td>8296</td>
<td>16</td>
</tr>
<tr>
<td>1,01,000</td>
<td>32</td>
<td>11782</td>
<td>8138</td>
<td>16</td>
</tr>
<tr>
<td>2,19,500</td>
<td>35</td>
<td>11713</td>
<td>7940</td>
<td>17</td>
</tr>
<tr>
<td>3,00,000</td>
<td>70</td>
<td>11736</td>
<td>8094</td>
<td>17</td>
</tr>
<tr>
<td>4,66,737</td>
<td>73</td>
<td>11747</td>
<td>7754</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 7.7: Execution Time for Sub-Contract Algorithm in MySQL

Figure 7.7 presents the graph corresponding to Table 7.7. The execution time has been noted over multiple iterations of the implementation.
7.2 Sub Contract Algorithm

As seen in Figure 7.7, Sub-Contract algorithm implemented in MySQL have identical performance for various dataset sizes. It can be ascertained that not many sub-contracting actors were detected beyond a certain dataset size. Also locality of reference played a big part as most of the data for a larger dataset were already accessed.

Table 7.8 puts forward the execution time noted for Sub-Contract algorithm implemented in Neo4j using the improvised approach.

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>Update Normal</th>
<th>Sub-Contract Detection</th>
<th>Update Result</th>
<th>Normalize Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>65,000</td>
<td>118</td>
<td>1542</td>
<td>2077</td>
<td>5</td>
</tr>
<tr>
<td>1,01,000</td>
<td>140</td>
<td>1704</td>
<td>2173</td>
<td>5</td>
</tr>
<tr>
<td>2,19,500</td>
<td>202</td>
<td>2534</td>
<td>2369</td>
<td>6</td>
</tr>
<tr>
<td>3,00,000</td>
<td>330</td>
<td>3442</td>
<td>5261</td>
<td>9</td>
</tr>
<tr>
<td>4,66,737</td>
<td>560</td>
<td>4149</td>
<td>5334</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 7.8: Execution Time for Sub-Contract Algorithm in Neo4j

Figure 7.8 depicts the execution time of Sub-Contract implementation in Neo4j using a radar graph. The colored lines represents different tasks of the algorithm. The convergence of these lines towards the vertices of the radar gives an estimate
of the execution time of each of these tasks. Dotted points at the centre of the radar indicates faster execution and hence smaller time value. Values farther from the center of the graph indicates higher time duration.

Comparing Figure 7.7 and Figure 7.8, it is observed that, Neo4j has better performance than MySQL in case of finding sub-contracting actors. It is so because, joins in MySQL are high intensive tasks as compared to defining relationships in Neo4j. Also in this case, joins were expensive because they had to be performed repeatedly for each CaseID.

Another aspect that Table 7.7 and Table 7.8 brings forward is that write operation in MySQL is a painful task as compared to Neo4j. It is so because MySQL needs to write values, albeit zero or some default value, for all those relations that doesn’t even exist. This however can be avoided in Neo4j by only creating relationship between nodes that are part of the satisfying relation. Although a gradual increasing trend in UpdateResult (or write operation) in Neo4j is seen, it still outperforms MySQL write operation by almost a factor of 1.5.

Table 7.9 presents the disk space taken by tables in MySQL. These statistics in-
include both initial tables, intermediate tables, final tables and index if any.

Table 7.9: Disk Space Usage (bytes) for MySQL tables (Sub – Contract)

<table>
<thead>
<tr>
<th>Tables</th>
<th>Dataset Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset</td>
<td>4734976</td>
</tr>
<tr>
<td>Organised Data</td>
<td>4734976</td>
</tr>
<tr>
<td>Result Matrix</td>
<td>1589248</td>
</tr>
</tbody>
</table>

Figure 7.9 shows the the variance of disk space usage in MySQL for five different dataset sizes. Initial table dataset and table that has the data ordered, organiseddata are of the same size because organiseddata doesn’t add any new data to it. It just organises data so any overhead on selecting and organising data for each case during join computation may be avoided.

The disk usage of ResultMatrix in MySQL is quite less as compared to other tables because it only stores the sub-contraction values for all unique actors. Extra disk space for storing Activity, CaseID are avoided in this case.

Table 7.10 shows the disk usage of graph elements for the implementation of
sub-contraction algorithm for different dataset sizes.

<table>
<thead>
<tr>
<th>Tables</th>
<th>Dataset Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>65000</td>
</tr>
<tr>
<td>Nodes</td>
<td>982212</td>
</tr>
<tr>
<td>Relationships</td>
<td>153477291</td>
</tr>
<tr>
<td>Result Matrix</td>
<td>384189475</td>
</tr>
</tbody>
</table>

Figure 7.10: Disk Space Usage (bytes) for Neo4j elements (Sub – Contract)

Figure 7.10 shows the the variance of disk space usage in Neo4j for graph elements like nodes, relationship and properties. The disk space for nodes is contributed by three different nodes type viz. Case nodes, actor nodes and unique actor nodes. There are three relationships that contribute to relationship disk space viz. [:CONTAINS] relationships that connects case node to actor nodes, [:RELATED_TO] connects actor to actor who satisfy the sub-contraction criteria and [:SUBCONTRACT] connects unique actor nodes to unique actor nodes with the actual sub-contraction value between the unique actors. Properties take higher amount of disk space because there are eight properties in the implementation - five of them are part of nodes and three of them are embedded in relationships.
It can be concluded that disk space for Neo4j graph elements is comparatively higher due to the fact that there are a lot of redundant information spread across nodes.
Limitations and Future Work

In our work we have used the BPI challenge 2014 dataset that consists of only 466737 records. Though five different size of this dataset was used to study scalability over two algorithms, further work should include different dataset to study the impact of dataset change in the overall evaluation.

Both implementations were done of a single node setup. Evaluating organizational mining algorithms over a distributed setup would be an interesting work to look forward too. Such implementation would not only require careful examination of disk/memory requirements on a single node but also studying them on distributed setup. Besides, other factor like network I/O and validation of result would be of much importance.

Organizational Mining is a broad topic in itself. Generalizing results of two algorithms for an entire community of relational database and graph database is misleading. However with more metrics of organizational mining and social analysis and studying use case feasibilities for all other graph databases would form an important research topic. Study of performance of these algorithms and other organizational mining algorithms with different graph databases like OrientDB, ArangoDB, InfiniteGraph, etc. would also be an important contribution.

Recommendation Systems are built using graph databases. And process discovery, conformance and enhancement are part and parcel of process mining. So studying the intersection of process enhancement with enhancement of graph
database like recommendation system would be a major contribution. To study how process mining can assist in building useful recommendation system can be useful contribution to the field of process mining and graph database.
Conclusion

This paper introduced the implementation of two different organizational mining algorithms in Structured Query Language and Cypher Query Language. It compares the performance of these algorithms on MySQL and Neo4j. Implementations were carried out on both native SQL client and as well as through java api’s using memoization. The work concentrated on defining suitable tables for MySQL and defining proper node and relationship structure in Neo4j. Writing efficient queries, stored procedures or Neo4j scripts for two different algorithms was an important challenge.

Implementation of Similar-Task algorithm were done on native SQL and Cypher clients. Reading database, processing and storing results back to the database were entirely query language specific. Implementation of Sub-Contract algorithm was done using api’s for respective databases. Experiments for Similar-Task and Sub-Contract algorithm showed that Neo4j performs better on when loading dataset of larger size. Thus, we can conclude that Neo4j is optimized to be really fast on a single node as compared to Cassandra. Also, the time taken to read from the database is more in Cassandra as compared to MySQL for read operations.

Performance comparison of Similar-Task implementation showed that MySQL gave better performance than Neo4j. MySQL gave better performance because there were only two hundred and forty two unique actors and achieving join for such smaller set was easy for MySQL. Also Neo4j needed to find intersection of activities between two actors before computing similarity which contributed to
greater execution time in Neo4j. However in case of Sub-Contract algorithm, Neo4j gave better performance as compared to MySQL. This is due to the fact that MySQL had to implement joins for all cases (CaseID) that were almost close to fifty thousand. Whereas graph databases like Neo4j are optimized to avoid join intensive queries by using index-free adjacency.

Write performance of relational and graph databases differ greatly. With respect to organizational mining algorithm implemented here, it was observed that for a smaller set of actors, MySQL performs better. But with increase in dataset size and the size of overall data to be pushed back to the database, Neo4j approach to writing properties only for valid relations proved to be the key difference. In MySQL, values had to set for those pair of actors between whom the relation didn’t even exist which increased both write overhead and disk usage.

Thus, with regards to organizational mining, it can be concluded that graph database performs better than relational database as size of dataset increases. Also as far as storage in concerned, Neo4j gave better performance when only distinct values are to be stored but in case when duplicate data is to be maintained, disk space usage varies with requirements.
References


[9] Yongtai Zhu Baile Shi Jian Pei Xifeng Yan Jiawei Han Wei Wang, Chen Wang. GraphMiner: A Structural Pattern-Mining System


