MINING FREQUENT STRUCTURAL PATTERNS FROM XML DATASETS

BY

MOHAMMED MOHSIN ALI

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

MAY 2012

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis is written by **MOHAMMED MOHSIN ALI** under the direction of his thesis advisor and approved by his thesis committee members, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE**

Thesis Committee

Dr. Muhammed Salah Al-Mulhem (Chairman)

Dr. Salahadin Mohammed (Co-Chairman)

Dr. Moataz Ahmed (Member)

Dr. Adel Ahmed (Member)

Dr. Sajjad Mahmood (Member)

Dr. Adel Ahmed
(Department Chairman)

Dr. Salam A. Zummo
(Dean of Graduate Studies)

20/5/12

Date

DEDICATED TO

My parents

ACKNOWLEDGEMENT

First and foremost thanks to Allah for giving me strength, patience and ability to accomplish this thesis. Peace and blessing of Allah be upon his last messenger Mohammed (Sallallah-Alaihe-Wasallam), who guided us to the right path.

I would like to express my gratitude to my thesis Advisor Dr. Muhammed Salah Al-Mulhem and Co-advisor Dr. Salahadin Mohammed for their continuous support and encouragement. I would also like to thank the committee members Dr. Moataz Ahmed, Dr. Adel Ahmed, and Dr. Sajjad Mahmood for their advices.

I wish to thank my friends who deserve my warmest thanks for many of our discussions especially Imran-ul-haq and Zahid Ayar.

I am very grateful for the love and support of my parents. I would also like to give special thanks to my uncle Mr. Mohammed Hashim who constantly encouraged me to continue Higher Studies.

TABLE OF CONTENTS

Page

DEDICATED to i
ACKNOWLEDGEMENT..................... ii
Table of Contents..................... iii
List of Tables vi
List of Figures ix
Abstract..................... xi
الملخص..................... xiii
CHAPTER 1 INTRODUCTION..................... 1
1.1 General..................... 1
1.2 Problem Statement..................... 2
1.3 Thesis Objectives 3
1.4 Thesis Contributions 4
1.5 Research Methodology..................... 5
1.6 Thesis Outline 6
CHAPTER 2 BACKGROUND AND OVERVIEW..................... 8
2.1 Extensible Markup Language (XML) 8
2.2 Frequent Pattern Mining 10
2.3 The Apriori Algorithm 13
2.4 The FP-Growth Algorithm..................... 15
2.5 Frequent Structural Pattern Mining 18
CHAPTER 3 LITERATURE REVIEW..................... 21
3.1 Content-Based Algorithms..................... 21
3.2 Structure-Based Algorithms..................... 28
3.3 Content-Structure-Based Algorithms 34
3.4 Summary..................... 38
CHAPTER 4 PROPOSED ALGORITHM..................... 39
4.1 Introduction 39

4.2 Definitions 40
4.3 The XSC procedure 41
4.4 The Encoder Procedure 47
4.5 The Miner1 Procedure 49
4.6 The Miner2 Procedure 55
4.7 The ETEC procedure 60
4.8 Summary 63
CHAPTER 5 EXPERIMENTAL RESULTS 65
5.1 Introduction 65
5.2 Experimental Setup 66
5.2.1 The Machine and the software 66
5.2.2 The Datasets 66
5.2.3 The Performance measure 68
5.3 FSPM2 vs. MABERS 68
5.3.1 FSPM2 Vs. Mabers using The DBLP Dataset 69
5.3.2 FSPM2 Vs. Mabers using The LineItem Dataset 72
5.3.3 FSPM2 Vs. Mabers using The Synthetic Dataset 74
5.4 Scalibility of FSPM2 77
5.5 FSPM2 Vs. Mabers with non-uniform datasets 79
5.6 Comparison of FSPM1 Vs. FSPM2 80
5.6.1 FSPM1 Vs. FSPM2 using The DBLP Dataset 81
5.6.2 FSPM1 Vs. FSPM2 using The LineItem Dataset 84
5.6.3 FSPM1 Vs. FSPM2 using The SYNTHETIC Dataset 86
5.8 Summary 88
CHAPTER 6 CONCLUSION AND FUTURE WORKS 89
6.1 Thesis Summary 89
6.2 Future Work 91
Appendix A 93
FSPM2 Vs. Mabers using The DBLP Dataset D1 93
FSPM2 Vs. Mabers using The DBLP Dataset D2 95
FSPM2 Vs. Mabers using The DBLP Dataset D3 97
FSPM2 Vs. Mabers using The DBLP Dataset D4 99
Appendix B 101
FSPM2 Vs. Mabers using The Synthetic Dataset S1 101

FSPM2 Vs. Mabers using The Synthetic Dataset S2 103
FSPM2 Vs. Mabers using The Synthetic Dataset S3 105
FSPM2 Vs. Mabers using The Synthetic Dataset S4 107
Appendix C 109
FSPM1 Vs. FSPM2 using The DBLP Dataset D1 109
FSPM1 Vs. FSPM2 using The DBLP Dataset D2 111
FSPM1 Vs. FSPM2 using The DBLP Dataset D3 113
FSPM1 Vs. FSPM2 using The DBLP Dataset D4 115
Appendix D 117
FSPM1 Vs. FSPM2 using The Synthetic Dataset S1 117
FSPM1 Vs. FSPM2 using The Synthetic Dataset S2 119
FSPM1 Vs. FSPM2 using The Synthetic Dataset S3 121
FSPM1 Vs. FSPM2 using The Synthetic Dataset S4 123
REFERENCES 125
Vitae 128

LIST OF TABLES

Page

TABLE 2.1: TRANSACTION TABLE 12
TABLE 2.2: TRANSACTION TABLE 14
TABLE 2.3: SAMPLE TBD............ 17
TABLE 2.4: MINING FP-TREE 18
TABLE 4.1: THE STATE OF THE TAGS TABLE AFTER PROCESSING THE FIRST TRANSACTION IN THE DBLP DATASET OF FIGURE 4.1............ 45
TABLE 4.2: THE STATE OF THE LABELED-PATHS TABLE AFTER PROCESSING THE FIRST TRANSACTION IN THE DBLP DATASET OF FIGURE 4.1 45
TABLE 4.3: THE STATE OF CLUSTERS TABLE AFTER PROCESSING THE FIRST TRANSACTION IN THE DBLP DATASET OF FIGURE 4.1............ 45
TABLE 4.4: THE STATE OF THE TAGS TABLE AFTER PROCESSING THE LAST TRANSACTION IN THE DBLP DATASET OF FIGURE 4.1............ 46
TABLE 4.5: THE STATE OF THE LABELED-PATHS TABLE AFTER PROCESSING THE LAST TRANSACTION IN THE DBLP DATASET OF FIGURE 4.1 46
TABLE 4.6: THE STATE OF CLUSTERS TABLE AFTER PROCESSING THE LAST TRANSACTION IN THE DBLP DATASET OF FIGURE 4.1............ 46
TABLE 4.7: THE STATE OF THE TAG-CODES TABLE AFTER PROCESSING THE TAGS TABLE SHOWN IN TABLE 4.4............ 47
TABLE 4.8: THE STATE OF THE LABELED-PATHS TABLE AFTER THE ENCODER POPULATES THE LP-CODE COLUMN. 49
TABLE 4.9: LABELED-PATHS TABLE FOR DBLP DATASET............ 58
TABLE 4.10: CLUSTER TABLE FOR DBLP DATASET............ 58
TABLE 4.11: CLUSTER TABLE FOR DBLP DATASET AFTER REMOVING NON-FREQUENT LP-IDS 58
TABLE 4.12: CLUSTER TABLE AFTER 2-TREE-EXPRESSION............ 59
TABLE 4.13: CLUSTER TABLE AFTER 2-TREE-EXPRESSION FOR ALL CLUSTERS............ 59
TABLE 4.14: HASHTABLE REPRESENTATION OF 2-TREE-EXPRESSION 60
TABLE 4.15: CLUSTER TABLE AFTER REMOVING NON FREQUENT TREE- EXPRESSIONS............ 60
TABLE 5.1: THE DATASETS USED............ 67
TABLE 5.2: ELAPSED TIME: FSPM2 VS MABERS FOR D5 70
TABLE 5.3: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR D5 71
TABLE 5.4: ELAPSED TIME: FSPM2 VS MABERS FOR L1 – L5 73

TABLE 5.5: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR L1 – L5 74
TABLE 5.6: ELAPSED TIME: FSPM2 VS MABERS FOR S5 76
TABLE 5.7: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR S5 76
TABLE 5.8: ELAPSED TIME FOR D6-D10 78
TABLE 5.9: COUNT OF TREE-EXPRESSIONS FOR D6-D10 78
TABLE 5.10: ELAPSED TIME FSPM2 VS. MABERS 79
TABLE 5.11: FSPM1 VS FSPM2 MINING TIME FOR D5 83
TABLE 5.12: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR D5 83
TABLE 5.13: FSPM1 VS FSPM2 MINING TIME FOR L1 – L5 85
TABLE 5.14: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR L1 – L5 85
TABLE 5.15: FSPM1 VS FSPM2 MINING TIME FOR S5 87
TABLE 5.16: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR S5 88
TABLE A.1: ELAPSED TIME: FSPM2 VS MABERS FOR D1 94
TABLE A.2: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR D1 94
TABLE A.3: ELAPSED TIME: FSPM2 VS MABERS FOR D2 96
TABLE A.4: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR D2 96
TABLE A.5: ELAPSED TIME: FSPM2 VS MABERS FOR D3 98
TABLE A.6: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR D3 98
TABLE A.7: ELAPSED TIME: FSPM2 VS MABERS FOR D4 100
TABLE A.8: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR D4 100
TABLE A.9: ELAPSED TIME: FSPM2 VS MABERS FOR S1 102
TABLE A.10: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR S1 102
TABLE A.11: ELAPSED TIME: FSPM2 VS MABERS FOR S2 104
TABLE A.12: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR S2 104
TABLE A.13: ELAPSED TIME: FSPM2 VS MABERS FOR S3 106
TABLE A.14: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR S3 106
TABLE A.15: ELAPSED TIME: FSPM2 VS MABERS FOR S4 108
TABLE A.16: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR S4 108
TABLE A.17: FSPM1 VS FSPM2 MINING TIME FOR D1 110
TABLE A.18: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR D1 110
TABLE A.19: FSPM1 VS FSPM2 MINING TIME FOR D2 112
TABLE A.20: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVEL FOR D2 112
TABLE A.21: FSPM1 VS FSPM2 MINING TIME FOR D3 114
TABLE A.22: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVEL FOR D3 114
TABLE A.23: FSPM1 VS FSPM2 MINING TIME FOR D4 116
TABLE A.24: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR D4 116
TABLE A.25: FSPM1 VS FSPM2 MINING TIME FOR S1 118
TABLE A.26: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVEL FOR S1 118
TABLE A.27: FSPM1 VS FSPM2 MINING TIME FOR S2 120
TABLE A.28: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVEL FOR S2 120

TABLE A.29: FSPM1 VS FSPM2 MINING TIME FOR S3 122
TABLE A.30: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVEL FOR S3 122
TABLE A.31: FSPM1 VS FSPM2 MINING TIME FOR S4 124
TABLE A.32: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVEL FOR S4 124

LIST OF FIGURES

Page

Figure 2.1: An Example XML document. 9
Figure 2.2: Demonstrating Apriori principle 15
Figure 2.3: FP-Tree constructed from Table 2.3 17
Figure 2.4: Sample XML Tree 19
Figure 2.5: Association Rules on Values (1 and 2) and on Structure (3) 20
Figure 3.1: Example of right-and-left Tree joins 29
Figure 4.1: DBLP Dataset 44
Figure 4.4a: Join-compatibility of Path1 and Path2 52
Figure 4.4b: Join-compatibility of Path2 and Path3 52
Figure 4.5: Demonstrating join compatibility of 2 TE_2 54
Figure 4.6: Generating sub tree-expression 63
Figure 5.1: Elapsed Time: FSPM2 VS Mabers using D5 70
Figure 5.2: Elapsed Time: FSPM2 VS Mabers for L1 – L5 72
Figure 5.3: Elapsed Time: FSPM2 VS Mabers using S5 75
Figure 5.4: Elapsed Time FSPM2 VS Mabers using D6-D10 77
Figure 5.5: Elapsed Time FSPM2 VS mabers 80
Figure 5.6: FSPM1 VS FSPM2 Mining Time for D5 82
Figure 5.7: FSPM1 VS FSPM2 Mining Time for L1 – L5 84
Figure 5.8: FSPM1 VS FSPM2 Mining Time for S5 86
Figure A.1: Elapsed Time: FSPM2 VS Mabers using D1 93
Figure A.2: Elapsed Time: FSPM2 VS Mabers using D2 95
Figure A.3: Elapsed Time: FSPM2 VS Mabers using D3 97
Figure A.4: Elapsed Time: FSPM2 VS Mabers using D4 99
Figure A.5: Elapsed Time: FSPM2 VS Mabers using S1 101
Figure A.6: Elapsed Time: FSPM2 VS Mabers using S2 103
Figure A.7: Elapsed Time: FSPM2 VS Mabers using S3 105
Figure A.8: Elapsed Time: FSPM2 VS Mabers using S4 107
Figure A.9: FSPM1 VS FSPM2 Mining Time for D1 109
Figure A.10: FSPM1 VS FSPM2 Mining Time for D2 111
Figure A.11: FSPM1 VS FSPM2 Mining Time for D3 113
Figure A.12: FSPM1 VS FSPM2 Mining Time for D4 115
Figure A.13: FSPM1 VS FSPM2 Mining Time for S1 117

Figure A.14: FSPM1 VS FSPM2 Mining Time for S2 119
Figure A.15: FSPM1 VS FSPM2 Mining Time for S3 121
Figure A.16: FSPM1 VS FSPM2 Mining Time for S4 123

ABSTRACT

Name: Mohammed Mohsin Ali

Title: Mining Frequent Structural Patterns from XML Datasets

Degree: MASTER OF SCIENCE

Major Field: Information and Computer Science

Date of Degree: May, 2012

Due to its flexibility and capability for representing various kinds of data, XML has become a de facto standard for data exchange over the net. Recently, the use of XML has been increasing at tremendous pace. With the ever-increasing amount of data available in XML format, the ability to mine valuable information from them has become increasingly important. However mining useful information from the XML is difficult due to its hierarchical tree structure. In this thesis we are proposing a new and efficient algorithm for mining frequent structures from XML documents. Unlike general trees, XML trees have many repeated substructures. So the proposed algorithm exploits the presence of repeated substructures and does the following. First, it clusters the input XML dataset by structure; second, it encodes the XML dataset objects in order to minimize storage space and to

avoid string manipulation; and third, it applies Apriori algorithm on the clustered and encoded XML dataset to find the frequently repeated substructures. The experimental results show that the proposed algorithm significantly outperforms the Apriori based algorithms.

الملخص

الاسم:	محمد محسن علي
عنوان الرسالة:	إستكشاف الأنماط الهيكلية المتكررة من مجموعات البيانات في XML
الدرجة العلمية:	ماجستير العلوم
التخصص:	علم الحاسوب
التاريخ:	مايو، 2012

أصبح ال XML معياراً مقبولاً لتبادل المعلومات على شبكة الانترنت، نظراً لمرونته وقدرته على تمثيل أنواع مختلفة من البيانات. في الوقت الحاضر تزايد استخدام ال XML بشكل هائل. ونتيجة لتزايد كمية المعلومات المخزنة بهذا التنسيق، تزايدت أهمية إستكشاف معلومات قيمة منها. بكل الأحوال فإن إستكشاف معلومات مفيدة من البيانات المخزنة بتنسيق ال XML يعتبر مهمة صعبة، نتيجة لتمثيل البيانات في أنماط شجرية هرمية. سنقوم في هذه الأطروحة بتقديم خوارزمية جديدة و فعالة لإستكشاف الأنماط المتكررة للبيانات في ملفات ال XML. بخلاف أنماط البيانات الشجرية العامة، تمتلك الأنماط الشجرية في XML العديد من الأنماط الفرعية المتكررة. تستغل الخوارزمية المقترحة وجود الأنماط الفرعية المتكررة و تقوم بما يلي: أولا، تقوم بتجميع المدخلات المتمثلة بمجموعات بيانات XML بناء على الأنماط الهيكلية لها. ثانيا، تقوم بتشفير مجموعة بيانات ال XML من أجل تقليل مساحة التخزين اللازمة و من أجل تجنب التعامل مع النصوص. ثالثا، تقوم بتطبيق خوارزمية Apriori على مجموعة بيانات ال XML بعد إجراء التجميع والتشفير عليها، تقوم هذه

الخوارزمية بإيجاد الأنماط الهيكلية الفرعية المتكررة بشكل مستمر. أظهرت النتائج التجريبية أن الخوارزمية المقترحة تعطي نتائج أفضل بشكل كبير من الخوارزمية المبنية على خوارزمية Apriori.

CHAPTER 1

INTRODUCTION

1.1 GENERAL

Due to its flexibility and capability for representing various kinds of data, XML has become a de facto standard for data exchange over the net [1]. The ability of XML to represent structured, semi-structured, and unstructured data gives it flexibility to model a wide variety of datasets into XML documents. Recently, the use of XML has been increasing at a tremendous pace, especially in web applications [2].

With the ever-increasing amount of data available in the XML format, the ability to mine frequent patterns from them has become increasingly important. However, mining frequent patterns from the XML data is difficult due to its nested structure. The traditional frequent pattern mining algorithms cannot be applied directly to XML data [3].

There are four types of frequent pattern mining algorithms used with XML data, namely, frequent content mining, frequent element mining, frequent structures mining, and frequent content and structures mining. The frequent content mining algorithms mine only the values in XML documents; the frequent element mining algorithms mine only the tag names in XML documents; the frequent structures mining algorithms mine only the structural relationships among the elements in XML documents; and the frequent content and structures mining algorithms mine both content and structures in XML documents.

Finding frequent patterns has many applications, such as, querying/browsing information sources [4], indexing [5], and building wrappers [6]. It also plays an essential role in many data mining tasks such as associations, correlations, classification, clustering, and many other interesting relationships among data. Thus, frequent structures mining has become an important data mining task and a focused theme in data mining research.

1.2 PROBLEM STATEMENT

Recently, many frequent structures mining algorithms have been proposed to mine XML data. This is because many organizations have huge amount of data in XML format, and they need to discover rules and patterns

from the data to help them in decision making [7]. Due to the nested structure of the XML documents, the traditional frequent pattern mining algorithms for relational and transactional tables cannot be applied directly. Existing frequent structures mining algorithms are inefficient and are not scalable [8] [9]. Most of them suffer from high I/O cost when the input XML document is big. This thesis will address the problem of researching a new, efficient, and scalable frequent structures mining algorithm for XML datasets.

1.3 THESIS OBJECTIVES

The main objective of this thesis is to propose and implement a new, efficient, and scalable algorithm to mine frequent structures from XML datasets. In order to accomplish this objective the following tasks were performed.

1. Extensive survey of existing frequent structures mining algorithms was conducted.
2. A new, efficient, and scalable frequent structures mining algorithm was proposed.
3. The proposed algorithm was designed and implemented.
4. The proposed algorithm was tested using benchmark XML datasets.

5. Performance of the proposed algorithm was evaluated and the factors that affect its performance were identified.
6. The experimental results were analyzed and the drawn conclusions and future directions in the area of frequent structures mining are presented.

1.4 THESIS CONTRIBUTIONS

The contributions of our thesis are as follows:

1) Extensive literature survey of all the existing algorithms that mine frequent structural patterns from XML data.

2) Two new, efficient, and scalable algorithms to mine frequent structural patterns from XML dataset.

3) To the best of our knowledge, the proposed algorithms are the first algorithm to use clustering for mining frequent substructures from XML datasets.

4) An encoding scheme which improves the performance of the proposed algorithms by reducing their memory requirements and

by minimizing the number of string manipulations they need to perform.

5) Implementation of the two proposed algorithms.
6) Experimental results, analysis, and comparisons of the proposed algorithms.

1.5 RESEARCH METHODOLOGY

Our research methodology consisted of the following phases:

Phase 1: Literature review

In this phase, the existing algorithms which mine frequent structural patterns from XML datasets were studied and their limitations and strengths were identified. This phase helped us to thoroughly understand the area and to state the scope and the contributions of the thesis.

Phase 2: Proposition of the new algorithm

In this phase, we proposed the frequent structure mining algorithm which uses clustering, encoding, and mining using the principles of Apriori.

Phase 3: Implementation of the proposed algorithm

In this phase, the proposed frequent structural pattern mining algorithm was implemented using C#.

Phase 4: Performance analysis of the proposed algorithm

In this phase, the proposed algorithm was tested using benchmark and synthetic datasets. From the collected experimental results, the performance of the proposed algorithm was analyzed and compared.

Phase 5: Conclusions

In this phase, conclusions from the research work were drawn; and future directions in the research of mining frequent structural patterns from XML data were identified.

Thesis writing

The writing of this thesis was done during all the phases.

1.6 THESIS OUTLINE

The remaining of this thesis is organized as follows. Chapter 2 presents basic terminology, background information on XML, and frequent pattern mining. We reviewed the related works in Chapter 3. Chapter 4 presents the

proposed frequent structural pattern mining algorithms. In Chapter 5, we present the experimental results and analysis of the proposed algorithms. Finally, Chapter 6 concludes this thesis and suggests some future work.

CHAPTER 2

BACKGROUND AND OVERVIEW

This chapter presents some background information on XML and frequent pattern mining. The background information given is necessary to understand the rest of the chapter. Readers familiar with basic XML and frequent pattern mining can skip the chapter.

The chapter is organized as follows. In Section 2.1 a brief introduction to XML is presented. Frequent pattern mining is discussed in Section 2.2. In Sections 2.3 and 2.4, two most common frequent pattern mining algorithms, namely, the Apriori and the FP-growth are briefly presented. Section 2.5 gives an overview of frequent structural pattern mining in the context of XML.

2.1 EXTENSIBLE MARKUP LANGUAGE (XML)

XML stands for e**X**tensible **M**arkup **L**anguage which is a language for representing structured, semi-structured, and unstructured data. An example of a small XML document is shown in Figure 2.1.

```
<books>
        <book id="000-213">
            <title>XML<title>
            <allauthors>
                <author>
                        <fn>jane</fn>
                        <ln>poe</ln>
                </author>
                <author>
                        <fn>john</fn>
                        <ln>doe</ln>
                </author>
            </allauthors>
            <year>2000</year>
            <chapter>
                <title>XML</title>
                    <section>
                            <head>Origins</head>
                    </section>
            </chapter>
    </book>
</books>
```

Figure 2.1: An Example XML document.

An XML document basically consists of the following components:

- **Elements**: Each element represents a logical component of a document. Elements can contain other elements, attributes, and/or values. The boundary of each element is marked with a start tag and an end tag. A start tag starts with the ‘<‘character and ends with the ‘>‘character. An end tag starts with ‘</’ and ends with ‘>‘. The root element contains all the other elements in the document. In the sample XML document shown

in Figure 2.1, the root element of the document is the *"books"* element. The children of an element are elements that are directly contained in that element. For example, in the sample document, the "*title*" element is a child of the "*book*" element.

- **Attributes**: Attributes are descriptive information attached to elements. The values of attributes are set inside the start tag of an element. For example, in Figure 2.1, the expression <book id="000-213"> sets the value of the attribute *"id"*. The main differences between elements and attributes are that attributes cannot have their own "attributes" and they cannot contain elements. Further information on XML specification can be found in [10].

- **Values**: Values are sequences of characters which appear between elements' start-tag and end-tag. Like attributes, values cannot contain elements. In Figure 2.1, the expressions "XML", "jane", and "2000" are examples of values.

2.2 FREQUENT PATTERN MINING

Frequent patterns are itemsets, subsequences, or substructures that appear in a dataset with frequency greater than or equal to a user-specified

threshold. For example, a set of items, such as milk and bread that appear frequently together in a transaction dataset is a *frequent itemset*. A subsequence, such as first buying an iPhone, then an iPod, and then an iPad, if it occurs frequently in a shopping history database, is a *frequent sequential pattern*. A substructure can refer to different structural forms, such as sub-graphs, subtrees, or sub-lattices. In the context of XML, a substructure refers to a path or a twig. If a substructure occurs frequently in an XML dataset, then it is called a *frequent structural pattern*. Finding frequent patterns plays an essential role in many data mining operations, such as, classification, clustering, association rules, and correlations to mention a few. Among all these, mining of association rules is one of the most popular operations.

Frequent pattern mining was first introduced by Agrawal et al. [11] to analyze the customer buying habits in retail databases. It analyses customer buying habits by finding associations between the different items that customers place in their 'shopping baskets'. For instance, if customers are buying milk, how likely are they going to also buy bread on the same trip to the supermarket? Such information can lead to increased sales by helping retailers do selective marketing and arrange their shelf space.

Frequent pattern mining is closely related to mining association rules. The problem of mining association rules can be explained as follows: There is

the itemset $I=i_1, i_2, \ldots, i_n$, where I is a set of n distinct items, and a set of transactions D, where each transaction *t* is a set of items such that 't' $\subseteq$ 'I'. 'I' is all the items in the supermarket and 't' is the set of items purchased by a customer and D is the set of the transactions by all customers. Table 2.1 gives an example where a database D contains a set of transactions 'T', and each transaction consist of one or more items.

Transaction-id	Items bought
1	Bread, Milk
2	Bread, Coffee, Eggs, Sugar
3	Milk, Coffee, Coke, Sugar
4	Bread, Coffee, Milk, Sugar
5	Bread, Coke, Milk, Sugar

TABLE 2.1: TRANSACTION TABLE

An association rule is an implication of the form $X \Rightarrow Y$, where $X, Y \subset I$ and $X \cap Y = \varphi$. Before we demonstrate the example of finding the support and confidence, let us define these terms.

Definition 2.1: *Support Count*: The *support count* of X, denoted p(X), is equal to the number of transactions in D that contain X.

Definition 2.2: *Support*: The *support* of X, denoted s(X), is equal to $p(X)/|D|$, where $|D|$ is the number of transactions in D.

Definition 2.3: *Confidence*: The confidence of rule $X \Rightarrow Y$, denoted $c(X \Rightarrow Y)$, is defined as a fraction of transactions that contain X that also contain Y, and is equal to $s(X \cap Y)/ s(X)$.

Example 2.1: In this example we demonstrate the method of calculating the association rules. Consider the transaction database shown in Table 2.1. In this transaction database, the association rule {milk, sugar} ⇒ coffee has a support of 0.4 and a confidence of 0.66. This means that 40% of the customers bought milk, sugar, and coffee together; and only 66% of the customers who bought milk and sugar also bought coffee.

Support = p ({milk, sugar, coffee}) / Total transactions

= 2/5 = 0.4

Confidence = p ({milk, sugar, coffee}) / p ({milk, sugar})

= 2/3 = 0.66

□

2.3 THE APRIORI ALGORITHM

The Apriori algorithm was first introduced by Agrawal et al. in [12]. It was used to mine association rules. Given a set of transactions, the problem of mining association rules is to generate all the association rules that have support and confidence greater than the user-specified minimum support

(called *minsup*) and minimum confidence (called *minconf*) respectively. The algorithm makes many passes over the data. The supports of individual items are counted in the first pass to find the *frequent* itemsets. Frequent itemsets are those with support greater or equal to minsup. The next pass is started with the seed set of itemsets which are found to be *frequent* in the previous pass. The seed set is used to find the potentially frequent itemsets called *candidate* itemsets; the actual support of these candidates is counted during the pass over the data. At the end of the pass, the candidate itemsets which are frequent become the seed for the next pass. This process is repeated until no new frequent itemsets are found.

Example 2.2: In this example we demonstrate the working of the Apriori algorithm. Consider the transactions shown in Table 2.2.

Transaction-id	Items bought
1	Bread, Milk
2	Bread, Coffee, Eggs, Sugar
3	Milk, Coffee, Coke, Sugar
4	Bread, Coffee, Milk, Sugar
5	Bread, Coke, Milk, Sugar

TABLE 2.2: TRANSACTION TABLE

The user wants all the frequent patterns which have a minimum support count of 3. The working example of the Apriori algorithm is shown in Figure 2.2

1-Itemset	
Itemset	Count
{Bread}	4
~~{Coke}~~	~~2~~
{Milk}	4
{Coffee}	3
{Sugar}	4
~~{Eggs}~~	~~1~~

2-Itemset	
Itemset	Count
{Bread, Milk}	3
~~{Bread, Coffee}~~	~~2~~
{Bread, Sugar}	3
~~{Milk, Coffee}~~	~~2~~
{Milk, Sugar}	3
{Coffee, Sugar}	3

3-Itemset	
Itemset	Count
{Bread, Milk, Sugar}	3

Figure 2.2: Demonstrating Apriori principle

The itemsets which have a count less than the minimum support are not used for generating the large itemsets in the next pass. The itemsets "*coke*" and "*eggs*" are not used for generating 2-Itemset in the second pass. And the itemsets "*bread*, *coffee*" and "*milk*, *coffee*" are not used for generating 3-Itemset in the third pass.

□

2.4 THE FP-GROWTH ALGORITHM

The FP-growth algorithm was first proposed by Han et al. in [13]. The FP-tree structure is constructed first and then frequent patterns are mined by traversing the constructed FP-tree. The structure of an FP-tree consists of a

prefix-tree of frequent 1-itemset and a frequent header table. For every node in a prefix-tree there are three fields: *item-name, count,* and *node-link*.

- *Item-name* is the name of the item.
- *Count* is the number of transactions that consists of the frequent 1-items on the path from root to this node.
- *Node-link* is the link to the next same item-name node in the FP-tree. Each entry in the frequent item header table has two fields: *item-name* and *head of node-link*.
- *Item-name* is the name of the item.
- *Head of node-link* is the link to the first same item-name node in the prefix-tree.

An FP-tree is constructed by scanning the transactional database (TDB) twice. In the first scan it retrieves the frequent items and they are ordered according to their support count. In the second scan, a tree with a root labeled as null is created. When a new transaction is read it is checked to see whether it is present in the tree as a path. If it is present then its count is incremented otherwise a new path is created. An example of an FP-tree is shown in Figure 2.3. This FP-tree is constructed from Table 2.3 with minsup of 2 [14].

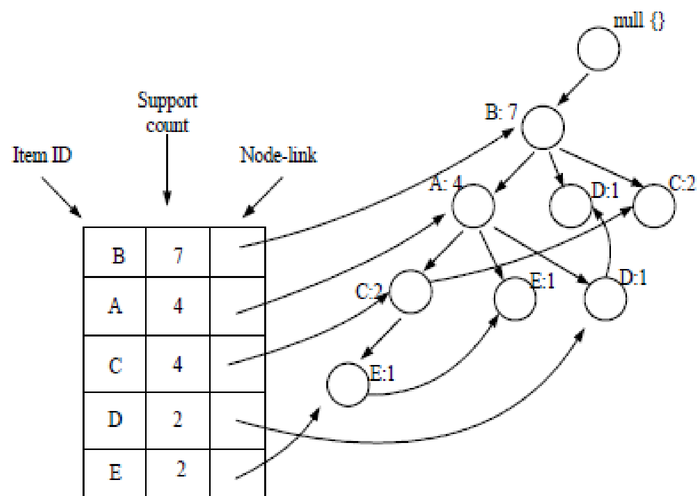


Figure 2.3: FP-Tree constructed from Table 2.3

TID	Items	Frequent Items
100	A, B, E	A, B, E
200	B, D	B, D
300	B, C	B, C
400	A, B, D	A, B, D
500	B, C	B, C
600	A, B, C, E	A, B, C, E
700	A, B, C	A, B, C

TABLE 2.3: SAMPLE TBD

The FP-growth algorithm then traverses all the node-links in the FP-tree's header table and mines the frequent patterns.

Example 2.3: We describe the process of mining all the frequent patterns including item A from the FP-tree shown in Figure 2.3. For node A, FP-growth mines a frequent pattern (A: 4) by traversing A's node-links through

node (A: 4). Then, it extracts A's prefix paths; <B: 7>. To study which items appear together with A, the transformed path <B: 4> is extracted from <B: 7> because the support value of A is 4. The path {(B: 4)} is called A's conditional pattern base. FP-growth then constructs A's conditional FP-tree containing only the paths in A's conditional pattern base as shown in Table 2.4. As only B is an item occurring more than minsup appearing in A's conditional pattern base, A's conditional FP-tree leads to only one branch (B: 7). Hence, only one frequent pattern (BA: 4) is mined.

Item	Conditional pattern base	Conditional FP-tree	Frequent pattern generated
E	{(BAC:1), (BA: 1)}	<B: 2, A: 2>	BE:2, AE:2, BAE: 2
D	{(B:1), (BA: 1)}	<B: 2>	BD:2
C	{(BA:2), (B: 2)}	<B: 4, A: 2>	BC: 2, AC:2, BAC: 2
A	{(B:4)}	<B: 4>	BA: 4

TABLE 2.4: MINING FP-TREE

□

2.5 FREQUENT STRUCTURAL PATTERN MINING

In this section we explain the frequent structural pattern mining in the context of XML. An XML document can be represented as a tree. Figure 2.4 shows an example of an XML document represented as a tree. Since an XML

document has a tree structure, mining XML association rules from XML documents is different than from traditional well-structured datasets. A transaction in an XML context is an XML fragment that defines the context in which the items must be counted. In other words, the transaction is a subtree. The root node of the subtree identifies the transaction.

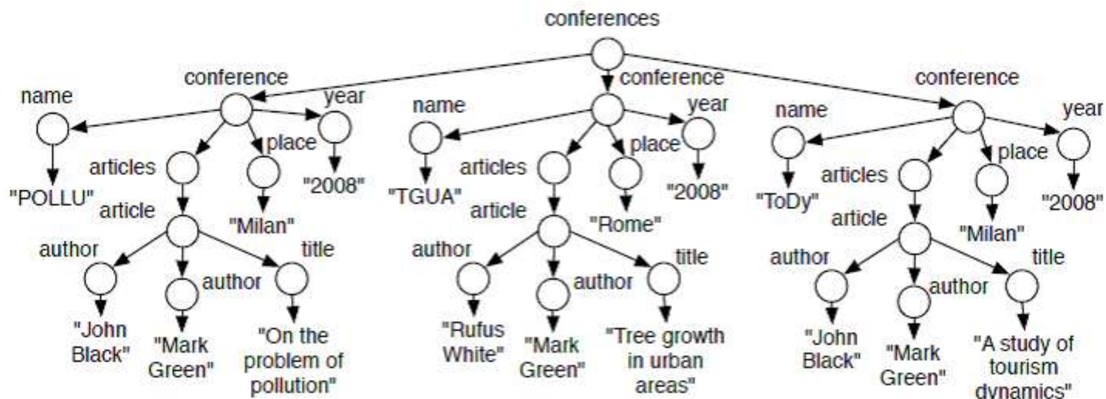


Figure 2.4: Sample XML Tree

Figure 2.5 shows some examples of association rules based on content (values) and structure.

Rule (1) states that, if there is a node labeled "*conference*" in the document, it probably has a child labeled "*year*" whose value is "*2008*".

Rule (2) states that, if there is a path composed by the following sequence of nodes: "*conference/articles/article/author*", and the content of author is "*Mark Green*", then the node "*authors*" probably has another child labeled "*author*" whose content is "*John Black*".

Finally, rule (3) describes the structural association rule mining, it states that, if there is a path composed of "*conference/articles/article*", then node "*conference*" probably has two other children labeled "*name*" and "*place*"[15]. The structural association rule does not contain values in the antecedent and precedent of the rule.

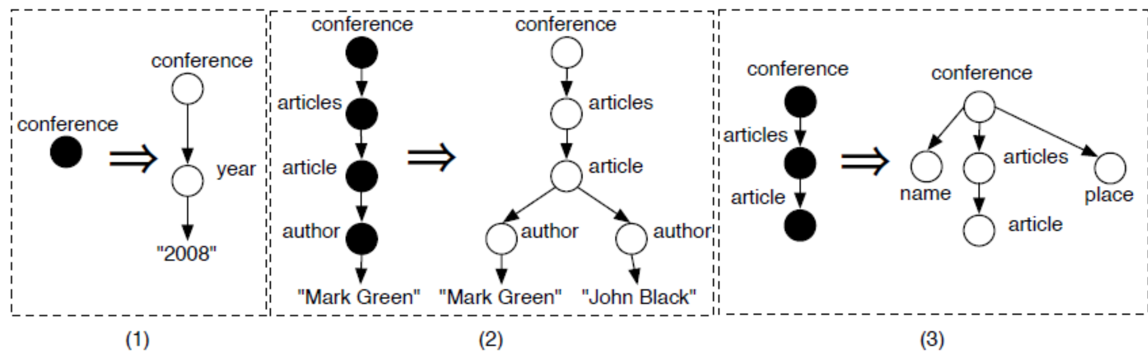


Figure 2.5: Association Rules on Values (1 and 2) and on Structure (3)

CHAPTER 3

LITERATURE REVIEW

Algorithms that mine association rules from XML documents can be classified into three, namely, content-based algorithms, structure-based algorithms, and content-structure-based algorithms. The content-based algorithms mine association rules only from the contents (values) of an XML dataset. These algorithms are discussed in Section 3.1. The structure-based algorithms mine association rules only from the structural relationships found in an XML dataset. The structure-based algorithms are discussed in Section 3.2. Finally, the content-structure-based algorithms mine association rules from both the contents and the structural relationships found in an XML dataset, and they are discussed in Section 3.3.

3.1 CONTENT-BASED ALGORITHMS

The content-based algorithms mine association rules from the contents of an XML dataset [7] [11] [16] [17] [18] [19] [20] [21] [22] . Nearly all of these algorithms are based on the Apriori algorithm [11].

Wang and Cao [18] algorithm starts by preprocessing the input XML document using the XSL and XSLT [23], which transforms the complex and irregular XML document into simple and regular XML document which helps in meeting the need of the mining algorithm. This type of preprocessing makes the algorithm more adaptable and universal and helps in identifying the mining context. The preprocessing goes through the following 3 steps.

1. A standard data source template called SDST is defined where the tag <transaction> forms the root node and is also used to identify the set of transactions. The transactions in the transaction set are represented by the tag <itemset> whereas the purchased item in each transaction is represented by the tag <item>.

2. The algorithm uses a modified Apriori algorithm to obtain large itemset to make it compatible with SDST.

3. The XSL and XSLT are applied to transform the complex and irregular XML document into a simple XML document with a different structure.

The association rules are then obtained from this transformed document using XQuery.

Khaing and Thein in [7] proposed an efficient association rule mining algorithm that mines association rules from large XML document. The XML data is represented as a binary table in which the value of 'one' for a particular item represents the existence of the item in the XML data and a value of 'zero' represents its absence. The algorithm uses Apriori like method to find the frequent itemsets and to generate the association rules.

The association rules are mined by converting the XML data into binary table format. The algorithm first computes the support count of the 1-itemsets. The items which do not satisfy the user defined threshold are filtered out. Candidate n-itemsets are obtained by applying logical AND operation on the frequent (n-1)-itemsets. This continues until the algorithm runs out of candidates. Interesting association rules are obtained by applying logical XOR operation. The obtained association rules are then displayed in XML format.

This algorithm cannot be applied to complex XML document and will be expensive when the XML document has numerous elements.

Li et al. in [19] made the task of mining association rules efficient; they gave a new definition to transaction and item in the context of XML. Their algorithm extracts the XML transactions from an index table. The index table is a collection of docID which represents the XML document number;

nodeEncode which is the encoding of a node *n* in a document tree and it is the encoding of its parent, augmented by the index of *n* among its siblings, adding a dot to separate them. The problem of checking 'include relation' between item and transaction is reduced to checking of ancestor-descendent relation between two element nodes. Since the transaction is a sub-tree and its leaf-node is an item, root is used to identify a transaction.

For extracting the transaction and item, the records (docID, nodeEncode) from the index table are selected where the given value is matched with the tag value. For each transaction (docID, nodeEncode) a transaction number (transID) is added. The relational table made up of transactions and items is generated. The columns are made up of XML items; rows are made up of XML transactions. If the ith transaction includes the jth item, then R(i,j)=1, otherwise, R(i,j)=0. Mining of association rules is done through XMLAssoMine algorithm which is based on plain Apriori algorithm.

The performance of the algorithm was demonstrated by the results obtained by applying the algorithm on a small 500KB Sigmod record real life data. This doesn't show the scalability of the algorithm.

Porkodi et.al in [20] presented an improved framework for mining association rules from XML data using XQuery and .NET based

implementation of the Apriori algorithm. The framework proposed in [20] consists of 3 phases.

1) **XQuery phase**: In this phase, XML transaction data files are stored in DB2 database. Each XML data file is uniquely identified using the transaction identifiers; using XQuery the items from the transaction database are accessed.

2) **Preprocessing Phase**: This phase is used to acquire distinct items from the XML data file. Then an encoded binary array is created. If an item occurs in the transaction then the corresponding column is encoded as 1, otherwise it is encoded as 0.

3) **Association Rule Mining Phase**: Finally, large itemsets are computed using Apriori algorithm, and association rules are generated for itemsets that satisfy the minimum support and the minimum confidence. The generated association rules are represented in the XML format.

The experimental results show that the algorithm works better for any size and number of XML data files. The algorithm also outperforms the existing java based implementation [24] in terms of CPU time by combining the features of XQuery and .Net based implementation of the Apriori algorithm. The performance of the algorithm is affected when the XML

dataset contains huge transactions because much time is spent in the communication with the DB2 database.

Zhang et al. in [21] proposed a framework called XAR-Miner, which can be used to mine association rules efficiently from XML documents. The preprocessing step transforms the XML document into an Indexed Content Tree (IX-tree) or Multi-relational databases (Multi-DB), depending on the memory constraints of the system and the size of XML document.

An indexed tree is a rooted tree $<V, E, A>$ where V is vertex, E is edge set and A is the indexed array set. The intermediate node stores the address of the immediate parent and children. An edge connects the two vertices using a bidirectional link. The set of indexed array stores the data in the leaf element or attribute nodes in the original XML document. When the size of the XML data exceeds then the XML document is transformed into relational database.

Data selection in IX-tree is facilitated by bidirectional link between parent and child nodes in a tree hierarchy. Nearest Common Ancestor Node (NCAN) is used to create the path connecting related concepts. In Multi-DB architecture the hierarchical information is maintained by creating SXS for each XML data and XPath of each relational database during data transformation. The SXS contain identical substrings of varied length. The

data is uniquely identified using the ordinal number of the NCAN of the concept.

The raw-XML data is generalized and meta-patterns are obtained. Based on the user specified minimum support and confidence, association rules are generated using Apriori algorithm.

Rahman et al. in [22] suggested a practical model which uses AR template for filtering data and generating virtual transactions which helps in efficiently finding the rules in which the user is interested. This work is an extension of XML-AR framework that was introduced by Feng [25].

The model consists of 5 steps namely Filtering, Generating Virtual Transactions, Finding Association Rules, Converting extracted rules to XML AR rules and Visualization. Filtering and Generating virtual transactions are the most important steps. XML-AR template is used in Filtering step to extract only those parts of XML in which the user is interested. Next, tag nesting in the XML document is used to define the transaction context; this is used to generate the virtual transactions which are used as input for association rule mining algorithm. The novel contribution of this algorithm is visualization of discovered association rules.

This work does not unveil all the association rules, the discovered rules are specific to the template used. There can be multiple templates for a single XML document based on the user's interest.

In some of works [14] [26] [27] the FP-Growth approach algorithm was used to mine the association rules.

Chit and Hla in [14] proposed an XQuery implementation for the efficient FP-tree based mining method which avoids preprocessing or post processing of XML documents. It overcomes the problem of costly candidate set generation by adopting pattern fragment method and divide-conquer method. It saves several database scans by constructing a highly compact FP-tree. The performance of the algorithm degrades significantly when applied on complex and irregular XML document.

3.2 STRUCTURE-BASED ALGORITHMS

The Structure-based algorithms mine frequent structures found in an XML dataset [3] [28] [29] [30] [28] [31] [6] [32] [33] [34]. Nearly all these algorithms are based on the Apriori Algorithm.

Hido and Kawano in [32] proposed AMIOT algorithm which uses candidate tree enumeration through right and left tree joins. Since two

frequent trees are joined, it is possible to enumerate efficiently only those trees with a high probability of being frequent. This ensures that the infrequent candidate trees and wasteful data scans are avoided in comparison to the conventional techniques of enumeration.

A partial tree excluding the leftmost leaf from a tree T is called the right tree 'Right (T)' and a partial tree excluding the rightmost leaf is called the left tree 'Left (T)'. The tree excluding both the left and the rightmost leaf is called the center tree 'Center (T)'. The tree expressed as a path from root vertex to the only leaf vertex is called a serial tree. Figure 3.1 shows the join operation of the right and left tree join. The serial tree cannot be enumerated with a right and left tree join. Therefore a new serial tree is obtained by adding a vertex to the only leaf vertex of T.

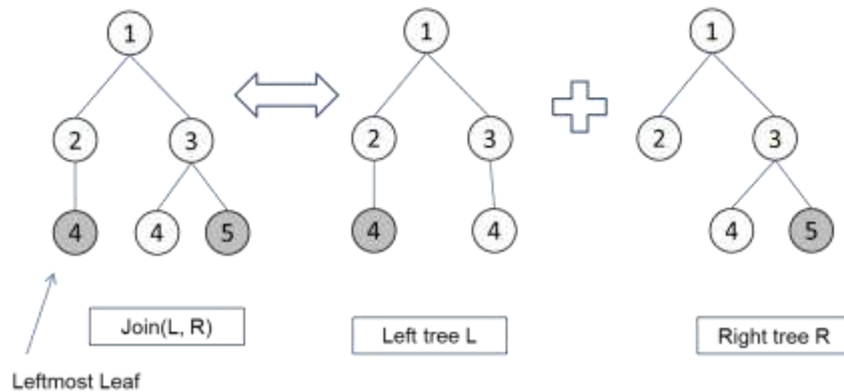


Figure 3.1: Example of right-and-left Tree joins

The frequent subtrees are obtained by data scanning and candidate tree are enumerated by right and left tree join and serial tree extension. This procedure is repeated until all the frequent subtrees are found. The execution time is the sum of candidate tree enumeration and time for calculating the number of occurrence of each candidate tree. More than half of the time is used for calculating the number of occurrences. When the size of dataset is large, the candidate enumeration time remains same but the time for calculating the number of occurrences increases significantly.

Experimental results showed that the AMIOT algorithm execution time was linear w.r.t to the data size. But, the enumeration technique with right-and-left tree joining needs a lot of memory space.

G. Cong et al. in [31] proposed a wild card mechanism which finds more complex and interesting substructures than existing techniques. Semi-structured objects are stored in a vertical data format. The algorithm has a powerful wildcard mechanism and overcomes the short coming of [6] by exploring the structure of irregular semi-structured data in a better way. Integers are used to code the labels of paths. The coding of path labels, introducing paths with wildcard and a special format of *tidlist* (tree id list) for paths with wildcard is done during the preprocessing phase. The algorithm uses the partitioning algorithm presented in [35] for association rules mining.

The algorithm has a downward closure property. The frequent subtrees are generated by using the r-path-structure (r>1) where *r* denotes a subtree composed of *r*-paths. F_r is used to denote the frequent r-path structure. The candidates are generated, pruning is done based on the downward closure property and if 'weaker than' relation exist between $f_1[1]$ and $f_2[1]$ for r=2. Finally all the frequent final substructures are generated. Support counting for 2-tree expression is a major performance bottleneck with this approach. Since it uses partition based algorithm it has an I/O issue.

Katsaros et al. in [33] proposed a fast mining of frequent tree structures by hashing and indexing. They identified the major performance bottleneck for WL algorithm [31] [6] which is that of support counting for 2-tree expressions. Repetitive tree-matching is avoided by using an efficient hashing scheme for ordered labeled trees.

The algorithm begins by computing the frequent 1-path expressions using the technique presented in [36]. The paths which were not frequent are removed. In the second stage, the 2-tree-expressions are bottleneck for the performance so a labeled tree encoding algorithm was applied and the tree was represented using the hash structures. To count the support of candidate 2-tree expression, the magic number of each 2-tree expression is calculated [31]. Hash structure is probed for each magic number, when a match is found

then a tree matching algorithm is used. When a match between corresponding trees is found then the count is increased by one. The WL algorithm for candidate support counting is followed for rest of the phases. The experiment was performed using synthetic and real data and it was observed that the proposed algorithm is better than the WL algorithm. The results can be further improved by using Clustering approach.

Paik et al. in [34] proposed a new algorithm called EFoX which is used to discover frequent subtrees. A special data structure called KidSet is used by the algorithm to manipulate frequent node labels and tree indexes. The algorithm does not use tree join operation to generate candidate sets. The algorithm consists of two steps. In the first step, the KidSet is created and maintained which avoids costly join operation and helps in reducing the number of candidates. In the second step, the data stored in the KidSet is used to extract the frequent subtrees.

A KidSet $[k]^d$ is a set of pairs (k_d,t_{id}) of keys, list of tree indexes where the key is a collection of node labels assigned on the nodes at depth *d* in every tree in D. D is the set of trees. In a KidSet, a pair is a frequent set if its key is frequent; otherwise it is called a Candidate Set. The Frequent Set and Candidate Set are represented by $[F]^d$ and $[C]^d$ respectively. The cross-reference operation consists of two phases. Phase 1 eliminates pairs from the

Candidate Set which are included in any previous Frequent Sets. Phase 2 is similar to Apriori style. In this phase, the Candidate Set is derived using union operation. Since the KidSet is a hierarchical structure, there is no need to generate candidate paths and additional candidate pairs by using join operations. Union operation is performed on the pairs which do not belong to any Frequent Set for all iterations. The processing of two consecutive Candidate Sets always produces frequent elements. The final sets of Frequent Sets contain all the keys which have frequency above the user specified threshold. The key of first non-empty frequent set is the root node of the frequent subtrees. Based on root nodes of the frequent subtrees, paths are formed with keys in the rest of Frequent Sets and the frequent subtrees are obtained incrementally.

The algorithm was evaluated using only synthetic data. Testing it using real data is important to show its effectiveness.

Paik et al. in [3] proposed EXiT-B which is a simple yet effective algorithm. According to the authors, this algorithm simplifies the process of mining maximal frequent subtrees. This was achieved in two distinct steps. All the string node labels are represented by some specified length of bits. Then, a PairSet, which is a specifically devised data structure, is used to avoid time-consuming tree join operation. The fundamental idea of the algorithm is

as follows: first, every node is mapped to an n-bit binary code. The bit sequences obtained by concatenation of each n-bit code are used to represent all the trees in the database. Each string label is transformed into an n-bit binary code by a hash function. Then, the algorithm mines the frequent subtrees using the binary code.

3.3 CONTENT-STRUCTURE-BASED ALGORITHMS

The content-structure-based algorithms mine association rules from both the structure and the contents of an XML dataset [15] [37] [38] [39] [40].

Paik et al. in [39] presented a framework for data structure-guided association rules extraction from XML trees. The use of a special data structure called Simple and Effective Lists Structure (SELS) avoids computationally intractable problem in the number of nodes, and it can represent simple and complex structured association relationships in XML data. With the use of SELS data structure, useless fragment generation is avoided, computational complexity is reduced, and fast extraction of desired fragments is enhanced. SELS is a set of lists which includes tree characteristic information such as label, tree id, node id, and parent/ancestor relationships among nodes.

For a tree database D, under each node label, the node ids and tree ids are members of a single list. The list is divided into two parts; one part is for identifying list from among a number of lists which deals with label of node and node ids; the other part stores all the relevant frequency information of the tree database. The complete list is called *Node and Tree List* (NaTL) as it is composed of node ids and tree ids. The leading part is named *head* of NaTL, ntl_{head} for distinguishing each NaTL; the trailing part is used for counting frequency of each label and is called *body* of NaTL, ntl_{body}.

Every ntl_{head} comprises of three fields which are label, node ids assigned the label, and pointer to corresponding ntl_{body}. The label is called the key of SELS for tree database. The corresponding ntl_{body} is a link list of several elements. Each element is composed of one pointer field for next element and two id fields; one is for tree which contains node(s) assigned the label in ntl_{head} and the other is for parent nodes of the node(s) in the tree. The count of the elements gives the information of the frequency of a label with respect to the database. The infrequent NaTLs whose body size is less than the threshold are filtered out. This filtered SELS is called *shallowly-frequent SELS* (*sfS)*. To refine the sfS a candidate hash table is created, the purpose of refinement is to deal with every parent node in elements and to make sfS contain all frequent nodes. The obtained SELS is called *deeply-frequent SELS,* abbreviated dfS.

Given a dfS, both ntl_{head} and ntl_{body} are associated together depending on ancestor-descendant relationship. Using this information, a *Minimum Support Satisfying Tree (*MSST*)* is derived. If the frequency of the edge is not frequent then that edge is deleted. Using this MSST and a given minimum support and minimum confidence, association rules can be generated.

Shin et al. in [40] proposed HILoP (Hierarchical layered structure of PairSet) which prevents multiple XML data scans to mine Association Rules from collection of XML documents. Also the number of candidate set is reduced by introducing Cross filtering algorithm. This approach avoids multiple data scans and simplifies the mining process.

The mining process consists of three phases: the first phase consists of constructing the tree structured data into a hierarchical structure called PairSets. A PairSet is a set of pair of element of XML tree and a tree in which the element appears. In second phase, the PairSets are operated according to minimum support. This minimum support is used to classify the PairSet into two class's namely frequent fragment set and candidate fragment set. In order to manipulate the data stored in the PairSets, cross-filtering algorithm is used. This cross-filtering algorithm consists of two steps; a pruning step which eliminates the current candidate sets which are already included in the frequent fragments sets previously; and a merging step which is used to

obtain the frequent fragment set from the current candidate fragment set without using the join operation. The third phase of the mining process mines the association rule measures. This algorithm can only be applied to XML trees with limited depth only.

Mazuran et al. in [15] extended the CMTTreeMiner which can be used to extract tree-based association rules from XML documents. The association rules are extracted without imposing any prior restriction on the structure and the content of the rules. The mined information is stored in XML format which can be queried later.

Mining tree based association rules are obtained in two steps. In the first step, frequent subtrees are obtained from the XML document and in the second step, interesting association rules are computed from the mined frequent subtrees. The algorithm mines association rules starting from the maximally frequent subtrees of the tree based representation of a document. The inputs given to the extended CMTTreeMiner algorithm are frequent subtrees and the minimal threshold for the confidence of the rules.

3.4 SUMMARY

Algorithms that mine association rules from XML documents can be classified into three, namely, content-based algorithms, structure-based algorithms, and content-structure-based algorithms. The content-based algorithms mine association rules only from the content (values) of XML dataset. The content based algorithms are not scalable, cannot be applied on the complex and irregular XML documents and some of them need XSLT according to the XML document structure. The structure-based algorithms, mine association rules only from the structural relationships found in an XML dataset. These algorithms have a major performance bottleneck for 2-tree expression, and the bit representation used by algorithm can be further enhanced. The performance of mining association rules on the structure can be improved by using clustering. Finally, the content-structure-based algorithms mine association rules from both the content and the structural relationships found in an XML dataset, these algorithms can be applied to the XML trees with limited depth only.

CHAPTER 4

PROPOSED ALGORITHM

4.1 INTRODUCTION

In this chapter the proposed two Frequent Structural Pattern Mining algorithms, namely, FSPM1 and FSPM2 are discussed in detail. Each of the two algorithms consists of four main procedures, namely, the XML Structural Clustering (XSC) procedure, the Encoder procedure, the Miner procedure, and the Embedded Tree-Expressions Counter (ETEC). The XSC, the Encoder, and the ETEC procedures are shared by both algorithms, but each has its own Miner procedure. Let the Miner procedure of FSPM1 be called Miner1 and that of FSMP2 be called Miner2. The XSC procedure clusters transactions by structure and is discussed in Section 4.3. The Encoder procedure maps each distinct element tag into a binary number. The Encoder is presented in Section 4.4. The Miner procedure mines the frequent structural patterns. Miner1 is discussed in Section 4.5 and Miner2 is discussed in Section 4.6. The ETEC procedure extracts tree-expressions embedded in the frequent tree-expressions and counts them. ETEC is explored in Section 4.7. But before we

discuss any of these procedures in detail, let us define some terms that are used in the rest of this thesis.

4.2 DEFINITIONS

Definition 4.1: A *transaction*: is a sub-tree rooted by an element specified by the user.

Definition 4.2: *A transaction element*: is the root element of a transaction.

Definition 4.3: A *tree-expression*: is a sub-tree of a transaction.

Definition 4.4: An *n-tree-expression*: is a tree-expression with n leaf nodes; where 0-tree-expression is a tree-expression with only one node. Let n-tree-expression be denoted as TE_n.

Definition 4.5: *Join-compatible*: Two n-tree-expressions are join-compatible if their first n-1 labeled paths are identical and they only differ in the last labeled path.

4.3 THE XSC PROCEDURE

The XSC procedure reads the input XML dataset and puts transactions with identical structure into the same cluster. To do this, the algorithm uses the following three tables.

The Labeled-Paths table: This table contains all the distinct labeled paths found in all the transactions. It has 4 columns, namely, LP, LP-ID, LP-count, and LP-code. LP contains distinct labeled paths; LP-ID contains IDs of each labeled path; LP-count contains the number of transactions with the same LP-ID, and LP-code contains the binary encoding of the labeled-path and is generated by the Encoder algorithm.

The Tags table: This table has 3 columns, namely, Tag-name, Tag-count, and Tag-code. XSC stores each distinct tag in the Tag-name column. The Tag-count column contains the number of transactions that contain the corresponding tag. The Tag-code is a bit string assigned to each tag by the Encoder.

The Clusters table: This table has two columns, namely, Cluster-ID and Cluster-Count. XSC maps transactions with identical structure into the same row and assigns them the same Cluster-ID. Cluster-Count contains the number of transactions with the same Cluster-ID.

Procedure 1 shows the XSC procedure. When the procedure reads a new transaction from the input XML dataset, it does the following:

- For each new tag, XSC inserts it into the Tags table and makes its Tag-count equals to 1. But if the tag is already in the Tags table and it is appearing for the first time in this transaction, then XSC increments its corresponding Tag-count by 1 (lines 3 to 9).
- For each labeled path of the transaction, if the labeled path is not in the Labeled-paths table, XSC assigns this labeled path a unique LP-ID number and inserts the labeled path, the LP-ID, and an LP-Count of 1 into the Labeled-paths table. But if the labeled path is already in the Labeled-paths table and this is the first time it is appearing in this transaction, then XSC increments its corresponding LP-count by 1 (lines 10 to 14).
- It forms a string called Cluster-ID by concatenating all the LP-IDs of the current transaction. The LP-IDs in the Cluster-ID are separated by a special character and they appear in the Cluster-ID in the same order they appear in the transaction (line 15).
- If the Cluster-ID is not in the Clusters Table, then XSC inserts it into the table and makes its corresponding Cluster-Count equal to 1. If the Cluster-

ID is already in the Clusters table, then it just increments its corresponding Cluster-Count by 1 (lines 17 to 21).

Procedure 1: XML Structural Clustering (XSC)

```
Begin
1:    foreach  transaction in the XML dataset
2:        Read all the labeled paths in the XML transaction
3:        foreach labeled path in the XML transaction
4:            Get the Tags present in the labeled path
5:            if the Tag present in Tags Table and first appearance in transaction
6:                Increment the Tag-count by '1'
7:            else
8:                Store the Tag-name and set Tag-count to '1'
9:            endif

10:           if a path is present in LP-Table and first appearance in transaction
11:               Increment LP-count by '1'
12:           else
13:               Store the path and set LP-count to '1'
14:           endif

15:           Concatenate the LP-IDs of a transaction by adding '*' as a separator
              //concatenated LP-IDs are called cluster
16:       endfor
17:           if Cluster-ID is not present in Cluster Table
18:               Store the Cluster-ID and set Cluster-Count to '1'
19:           else
20:               Increment the Cluster-Count by '1'
21:           endif
22:   endfor
End
```

Example 4.1: Let us assume that the transaction element in the DBLP dataset shown in Figure 4.1 is the 'inproceedings'. After XSC processes the first transaction, then the value of Cluster-ID will be equal to p1*p2*p3*p4*p5*p6*p7 and the states of the Tags table, Labeled-paths table, and Clusters table will be as shown in tables 4.1, 4.2, and 4.3 respectively.

```xml
<dblp>
 <inproceedings >
  <author>Philip D. MacKenzie</author>
  <title>Anonymous Investing: Hiding the Identities of Stockholders.<sub>n, n</sub></title>
  <pages>212-229</pages>
  <year>1999</year>
  <booktitle>Financial Cryptography</booktitle>
 </inproceedings>
 <inproceedings>
  <author>Gerhard Lakemeyer</author>
  <title>Belief Revision in a Nonclassical Logic.<sub>n, n</sub></title>
  <pages>199-211</pages>
  <year>1996</year>
  <booktitle>KI</booktitle>
 </inproceedings>
 <inproceedings>
  <author>Ioan Georgescu</author>
  <title>Constructive Theory Formation in Knowledge Based Systems.<sub>n, n</sub></title>
  <pages>300-312</pages>
  <year>1985</year>
  <booktitle>GWAI</booktitle>
 </inproceedings>
 <inproceedings>
  <author>Frank Gurski</author>
  <title>The Tree-Width of Clique-Width Bounded Graphs Without <i>K<sub>n, n</sub></i>.</title>
  <pages>196-205</pages>
  <year>2000</year>
  <crossref>conf/wg/2000</crossref>
 </inproceedings>
 <inproceedings>
  <author>Martin E. Hellman</author>
  <title>On the Difficulty of Computing Logarithms Over GF(q<sub>m</sub>).</title>
  <pages>83</pages>
  <year>1980</year>
  <crossref>conf/fc/1980</crossref>
  <booktitle>IEEE Symposium on Security and Privacy</booktitle>
 </inproceedings>
</dblp>
```


Figure 4.1: DBLP Dataset

Tag-name	Tag-count	Tag-code
inproceedings	1	
author	1	
title	1	
sub	1	
pages	1	
year	1	
booktitle	1	

TABLE 4.1: THE STATE OF THE TAGS TABLE AFTER PROCESSING THE FIRST TRANSACTION IN THE DBLP DATASET OF FIGURE 4.1

LP-ID	LP	LP-code	LP-count
p1	/inproceedings		1
p2	/inproceedings/author		1
p3	/inproceedings/title		1
p4	/inproceedings/title/sub		1
p5	/inproceedings/pages		1
p6	/inproceedings/year		1
P7	/inproceedings/booktitle		1

TABLE 4.2: THE STATE OF THE LABELED-PATHS TABLE AFTER PROCESSING THE FIRST TRANSACTION IN THE DBLP DATASET OF FIGURE 4.1

Cluster-ID	Cluster-Count
p1*p2*p3*p4*p5*p6*p7	1

TABLE 4.3: THE STATE OF CLUSTERS TABLE AFTER PROCESSING THE FIRST TRANSACTION IN THE DBLP DATASET OF FIGURE 4.1

□

Example 4.2: After processing all the five transactions in the DBLP dataset of Figure 4.1, the states of the Tags table, the Labeled-paths table, and the Clusters table will be as shown in tables 4.4, 4.5, and 4.6 respectively.

Tag-name	Tag-count	Tag-code
inproceedings	5	
author	5	
title	5	
sub	5	
pages	5	
year	5	
booktitle	4	
i	1	
crossref	2	

TABLE 4.4: THE STATE OF THE TAGS TABLE AFTER PROCESSING THE LAST TRANSACTION IN THE DBLP DATASET OF FIGURE 4.1

LP-ID	LP	LP-code	LP-count
p1	/inproceedings		5
p2	/inproceedings/author		5
p3	/inproceedings/title		5
p4	/inproceedings/title/sub		4
p5	/inproceedings/pages		5
p6	/inproceedings/year		5
P7	/inproceedings/booktitle		4
P8	/inproceedings/title/i		1
P9	/inproceedings/title/i/sub		1
P10	/inproceedings/crossref		2

TABLE 4.5: THE STATE OF THE LABELED-PATHS TABLE AFTER PROCESSING THE LAST TRANSACTION IN THE DBLP DATASET OF FIGURE 4.1

Cluster-ID	Cluster-Count
p1*p2*p3*p4*p5*p6*p7	3
p1*p2*p3*p8*p9*p5*p6*p10	1
p1*p2*p3*p4*p5*p6*p10*p7	1

TABLE 4.6: THE STATE OF CLUSTERS TABLE AFTER PROCESSING THE LAST TRANSACTION IN THE DBLP DATASET OF FIGURE 4.1

□

4.4 THE ENCODER PROCEDURE

The Encoder procedure takes the Tags table as input and populates its Tag-codes. The Tag-codes are binary numbers. The number of bits in a Tag-code is equal to the minimum number of bits needed to represent all the tags in the Tags table including the blank tag. The need for the blank tag will be explained later in this section.

Example 4.3: The number of tags required for Tag-names in Table 4.4 and the blank tag is 10. So the minimum number of bits needed to represent any Tag-code is 4. Table 4.7 shows the state of the Tag table after the Encoder populates its Tag-codes.

Tag-name	Tag-count	Tag-code
inproceedings	5	0001
author	5	0010
title	5	0011
sub	5	0100
pages	5	0101
year	5	0110
booktitle	4	0111
i	1	1000
crossref	2	1001

TABLE 4.7: THE STATE OF THE TAG-CODES TABLE AFTER PROCESSING THE TAGS TABLE SHOWN IN TABLE 4.4

□

The Encoder assigns the blank tag a Tag-code of 0. It assigns the rest of the tags sequentially the Tag-codes of 1, 2, 3, and so on.

After the Encoder finishes populating the Tag table with Tag-codes, it reads the Labeled-paths table to populate its LP-code column. The LP-code is also a binary number. The Encoder generates the LP-code of a labeled path by substituting each of its tags by its Tag-code and then by concatenating all its Tag-codes as shown in the equation below.

$$\text{LP-code}(LP_i) = \text{tag-code}(t_{i,0}) \;||\; \text{tag-code}(t_{i,1}) \;||\; \ldots \;||\; \text{tag-code}(t_{i,n-1})$$

where "||" is the concatenation operator, $t_{i,j}$ is the jth tag, from the root of the labeled path LP_i and n is the length of the longest labeled-path in the dataset. If the length of LP_i is less than n, then the Encoder adds a number of blank tags until the length of LP_i becomes n.

Example 4.4: The state of the Labeled-paths table after it is populated by the Encoder is shown in Tables 4.8. The length of the longest labeled path in the table is 4 and the length of p2 is 2. Hence, the Encoder changes the length of the LP-code of p2 into 4 by concatenating two more blank Tag-codes.

LP-ID	LP	LP-code	LP-count
p1	/inproceedings	0001000000000000	5
p2	/inproceedings/author	0001001000000000	5
p3	/inproceedings/title	0001001100000000	5
p4	/inproceedings/title/sub	0001001101000000	4
p5	/inproceedings/pages	0001010100000000	5
p6	/inproceedings/year	0001011000000000	5
P7	/inproceedings/booktitle	0001011100000000	4
P8	/inproceedings/title/i	0001001110000000	1
P9	/inproceedings/title/i/sub	0001001110000100	1
P10	/inproceedings/crossref	0001100100000000	2

TABLE 4.8: THE STATE OF THE LABELED-PATHS TABLE AFTER THE ENCODER POPULATES THE LP-CODE COLUMN.

□

Procedure 2: Encoder

```
begin
1:      Get the count of tags from the Tag Table
2:      Bits-required = ⌈log_2(Total number of Tags + 1)⌉

3:      foreach Tag-name in the Tag Table
4:              Assign Tag-code for each Tag-name
5:      endfor
6:      Get the length of longest LP
8:      foreach LP in the labeled path Table
9:              Substitute the Tag-code for each Tag
10:             if length of LP-code not equal to the Longest LP-code
11:                     Append blank tags
12:             endif
13:     endfor
End
```

4.5 THE MINER1 PROCEDURE

Miner1 mines the frequent structural patterns in the input XML dataset. The input to this procedure is the minimum support (minsup), the Tags table, the Labeled-Paths table, and the Clusters table and its output is a list of

frequent tree-expressions. The procedure uses the Apriori algorithm to generate the candidates and frequent tree-expressions. Let C(k) represent a set of candidate k-tree-expressions, F(k) a set of frequent k-tree-expressions, 'r' a row of the Clusters table, and 'r.C(k)' the C(k) of r.

Miner1 is shown in Procedure 3 and it goes through the following steps:

- It generates C(0) elements from the Tags table (line 1).
- It generates C(1) elements from the Labeled-Paths table (line 2).
- Then for each row r in the Clusters table it does the following three steps:
  1. Initializes k to 1.
  2. Generates r.C(k+1) from the join-compatible r.C(k) elements (line 5).
  3. If r.C(k+1) is not empty then it increments k by 1 and it goes back to step 2 (line 29).

The support count of each r.C(k) is equal to the Cluster-Count of r. So to generate the C(k) members, Miner1 adds the Cluster-Counts of all the r.C(k) members. Then a C(k) member whose support is higher than the minsup will be a member of F(k).

Procedure 3: MINER1

```
begin
1:    Generate 0-tree-expression from Tag Table
2:    Generate 1-tree-expression from Labeled Path Table
3:    foreach Cluster-ID
4:        foreach LP-code of LP-ID in Cluster-ID
              // Check if the path[i] can be joined with the all the paths appearing after it.
5:            Join-Compatible ( )
      //join compatible method
6:            If 'n' > 2
8:                If 'n-1' Labeled paths of n-tree expression are same
9:                    foreach Tag-code in LP-code
10:                    if Tag-code of LP1 and Tag-code of LP2 are same
11:                            continue
12:                    elseif Tag-code of LP1 and Tag-code of LP2 not same
13:                            if Tag-code of LP1 and Tag-code of LP2 not equal to zero
14:                                    RETURN joinable
15:                            else
16:                                    RETURN not joinable
17:                    endif

18:           elseif 'n' = 2
19:                   foreach Tag-code in LP-code
20:                    if Tag-code of LP1 and Tag-code of LP2 are same
21:                            continue
22:                    elseif Tag-code of LP1 and Tag-code of LP2 not same
23:                            if Tag-code of LP1 and Tag-code of LP2 not equal to zero
24:                                    RETURN joinable
25:                            else
26:                                    RETURN not joinable
27:                    endif
28:           endif
          endfor
29:       if C(k+1) not empty
30:               Goto step 5
31:       endif
32:   endfor
33:   Remove the non-frequent tree-expressions
end
```

In examples 4.5 and 4.6 we demonstrate the join compatibility of 1-tree-expressions and 2-tree-expressions respectively.

Example 4.5: The join-compatibility test of 1-tree-expressions is shown in Figure 4.4a and 4.4b. The join compatibility of Path1 and Path2 is shown in

Figure 4.4a and the join compatibility of Path2 and Path3 is shown in Figure 4.4b.

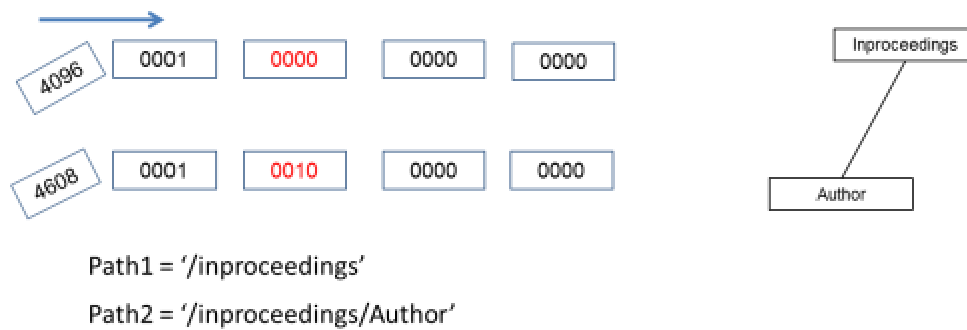


Figure 4.4a: Join-compatibility of Path1 and Path2

Path1 and Path2 differ in their second tags from the left. One of them has a Tag-code of '0000' and the other has a Tag-code '0010'. Since one of these Tag-codes is '0000', Path1 and Path2 cannot be joined to form a 2- tree-expression.

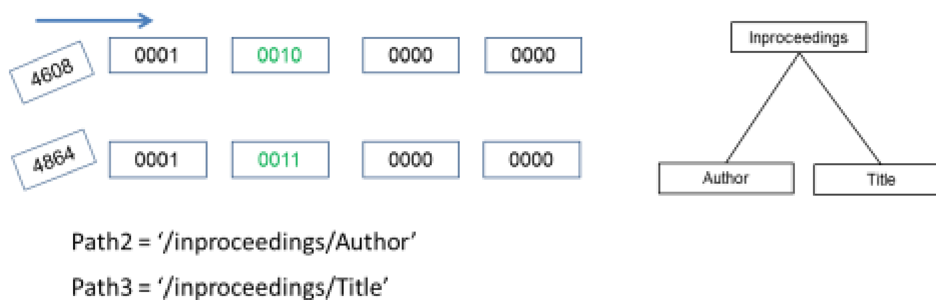


Figure 4.4b: Join-compatibility of Path2 and Path3

Path2 and Path3 differ in their second tag-codes which are '0010' and '0011' respectively. Since none of these two tags-codes is equal to '0000', Path2 and Path3 can be joined to form a 2-tree-expression. The LP-code of Path2 and Path3 are converted into integer as '4608*4864'. Where 4608 is the integer equivalent of the path '/inproceedings/author' and 4864 is the integer equivalent of path '/inproceedings/title'.

□

Example 4.6: Figure 4.5 shows the join-compatibility of 2-tree-expressions. The two paths '/inproceedings/author' and '/inproceedings/title' form 2-tree-expressions. Also the paths '/inproceedings/author' and '/inproceedings/pages' form 2-tree-expressions. The representation of these 2-tree-expression is '4608*4864' and '4608*5376' respectively.

4608*4864 = '/inproceedings/Author'* '/inproceedings/Title'

4608*88 = '/inproceedings/Author'* '/inproceedings/Pages'

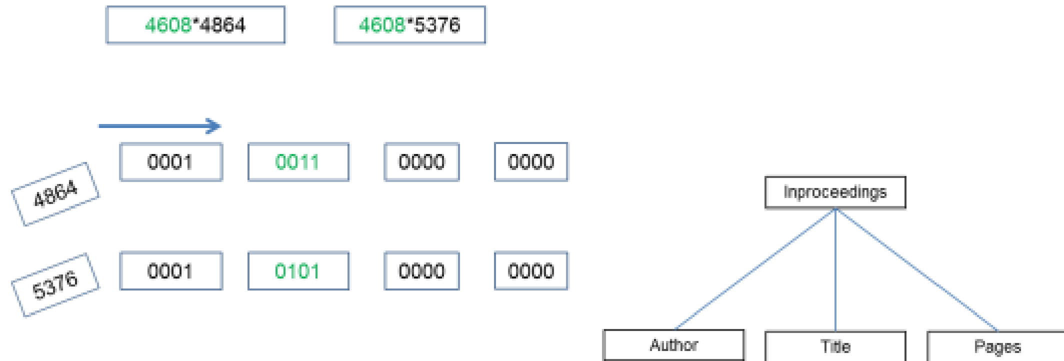


Figure 4.5: Demonstrating join compatibility of 2 TE_2

In Figure 4.5 the (n-1)-tree-expression of the two paths is the same, 'i.e.' 4608, '/inproceedings/Author'. The nth path of the first 2-tree-expression 'i.e.' 4864 and the nth path of second 2-tree-expression 'i.e.' 5376 is checked for join-compatibility. The join compatibility returns true because the Tag-codes of the second Tags are not the same and none of them is '0000'.

The new tree-expression is represented by joining the first 2-tree-expression with the nth labeled path of the second path to form '/inproceedings/Author*/inproceedings/Title*/inproceedings/Pages'. The integer representation of the new 3-tree-expression is '4608*4864*5376'.

□

4.6 THE MINER2 PROCEDURE

Miner2 mines the frequent structural patterns in the input XML dataset. The input to this procedure is the minsup, the Tags table, the Labeled-Paths table, and the Clusters table and its output is a list of frequent tree-expressions. Miner2 uses the Apriori algorithm to generate the candidate tree-expressions. Let C(k) be a set of candidate k-tree-expressions, F(k) a set of frequent k-tree-expression, 'r' a cursor which points to a row of the Clusters table, and 'r.C(k)' the C(k) members of r.

Miner2 is shown in Procedure 4, and it goes through the following steps:

1. It generates F(0) elements from the Tags table (line 1).
2. It generates F(1) elements from the Labeled-Paths table (line 2).
3. Then for each row 'r' in the Clusters table it does the following four steps:
    i. Initializes k to 1.
    ii. For each row 'r' it generates r.C(k+1) from the join-compatible r.F(k) members (line 5).

iii. Puts the C(k+1) members with count greater than minsup to F(k+1) (line 31).

iv. If C(k+1) is not empty, then it increments k by 1 and it goes back to step 2 (line 32).

The support count of each C(k) is equal to sum of the Cluster-Counts of all the r.C(K)s from which it is generated. So to generate the C(k) members, Miner2 adds the Cluster-Counts of all the r.C(k) members from the Cluster it is generated. Then a C(k) member whose support is higher than the minsup will be a member of F(k). The C(k+1) are always generated from the F(k).

Procedure 4: MINER2

```
begin
1:   Generate frequent 0-tree-expression from Tag Table
2:   Generate frequent 1-tree-expression from Labeled Path Table
3:   foreach Cluster-ID
4:     foreach LP-code of LP-ID in Cluster-ID
          // Check if the path[i] can be joined with the all the paths appearing after it.
5:        Join-Compatible ( )
     //join compatible method
6:          If 'n' > 2
8:               If 'n-1' Labeled paths of n-tree expression are same
9:                 foreach Tag-code in LP-code
10:                  if Tag-code of LP1 and Tag-code of LP2 are same
11:                        continue
12:                  elseif Tag-code of LP1 and Tag-code of LP2 not same
13:                        if Tag-code of LP1 and Tag-code of LP2 not equal to zero
14:                             Store the n-tree-expression in the Cluster-ID
15:                        else
16:                             RETURN not joinable
17:                  endif

18:         elseif 'n' = 2
19:                foreach Tag-code in LP-code
20:                  if Tag-code of LP1 and Tag-code of LP2 are same
21:                        continue
22:                  elseif Tag-code of LP1 and Tag-code of LP2 not same
23:                        if Tag-code of LP1 and Tag-code of LP2 not equal to zero
24:                             Store the n-tree-expression in the Cluster-ID
25:                        else
26:                             RETURN not joinable
27:                  endif
28:           endif
     //join compatible method
29:       endfor
30:  endfor
31:  Remove the non-frequent tree-expressions from Cluster-ID
32:    if r is not empty
33:          Generate C(k+1) using Join-compatible( )
34:          Remove the non-frequent tree-expressions from Cluster-ID
35:    else
36:          stop
37:    endif
End
```

In example 4.7 we explain the process of removing non-frequent LP-IDs from clusters table and example 4.8 illustrates the process of storing tree-expressions.

Example 4.7: The Labeled-Paths table is shown in Table 4.9. If the minimum support count, which is equal to the LP-count in this case, is 2, then 'P8' and 'P9' are pruned because their support count is less than 2.

LP-ID	LP	LP-code	LP-count
p1	/inproceedings	0001000000000000	5
p2	/inproceedings/author	0001001000000000	5
p3	/inproceedings/title	0001001100000000	5
p4	/inproceedings/title/sub	0001001101000000	4
p5	/inproceedings/pages	0001010100000000	5
p6	/inproceedings/year	0001011000000000	5
P7	/inproceedings/booktitle	0001011100000000	4
P8	/inproceedings/title/i	0001001110000000	1
P9	/inproceedings/title/i/sub	0001001110000100	1
P10	/inproceedings/crossref	0001100100000000	2

TABLE 4.9: LABELED-PATHS TABLE FOR DBLP DATASET

So the LP-IDs 'P8' and 'P9' are removed from the cluster table. The cluster table after removing the non-frequent LP-IDs is shown in Table 4.11.

Cluster-ID	Cluster-Count
p1*p2*p3*p4*p5*p6*p7	3
p1*p2*p3*p8*p9*p5*p6*p10	1
p1*p2*p3*p4*p5*p6*p10*p7	1

TABLE 4.10: CLUSTER TABLE FOR DBLP DATASET

Cluster-ID	Cluster-Count
p1*p2*p3*p4*p5*p6*p7	3
p1*p2*p3*p5*p6*p10	1
p1*p2*p3*p4*p5*p6*p10*p7	1

TABLE 4.11: CLUSTER TABLE FOR DBLP DATASET AFTER REMOVING NON-FREQUENT LP-IDS

□

Example 4.8: In this example we explain the process of storing tree-expressions. When the first Cluster-ID 'i.e.' 'p1*p2*p3*p4*p5*p6*p7' is processed the tree-expressions generated from this Cluster-ID 'i.e.' '4608*4864; 4608*4928; 4608*5376; 4608*5632; 4608*5888' are stored in the Cluster table. Table 4.12 shows the Cluster table after processing the 2-tree-expression for the first Cluster-ID.

Cluster-ID	Cluster-Count
4608*4864;4608*4928;4608*5376;4608*5632;4608*5888;	3
p1*p2*p3*p5*p6*p10	1
p1*p2*p3*p4*p5*p6*p10*p7	1

TABLE 4.12: CLUSTER TABLE AFTER 2-TREE-EXPRESSION

Table 4.13 shows the Cluster table after processing the 2-tree-expression for all the Cluster-IDs.

Cluster-ID	Cluster-Count
4608*4864;4608*4928;4608*5376;4608*5632;4608*5888;	3
4608*4864;4608*5376;4608*5632	1
4608*4864;4608*4928;4608*5376;4608*6400;4608*5888;6400*5888	1

TABLE 4.13: CLUSTER TABLE AFTER 2-TREE-EXPRESSION FOR ALL CLUSTERS

The generated 2-tree-expressions are also stored in the Hashtable (called C(2)). Table 4.14 shows the Hashtable containing the 2-tree-expressions.

2-tree-expression	Count
4608*4864	5
4608*4928	4
4608*5376	5
4608*5632	4
4608*5888	4
4608*6400	1
6400*5888	1

TABLE 4.14: HASHTABLE REPRESENTATION OF 2-TREE-EXPRESSION

It can be seen that the 2-tree-expressions '4608*6400' and '6400*5888' have a count of 1 which is less than our minimum support count of 2. So these tree-expressions are removed from the Hashtable (now called F(2)) shown in Table 4.14 and also from the Cluster table shown in Table 4.15.

Cluster-ID	Cluster-Count
4608*4864;4608*4928;4608*5376;4608*5632;4608*5888;	3
4608*4864;4608*5376;4608*5632	1
4608*4864;4608*4928;4608*5376;4608*5888;	1

TABLE 4.15: CLUSTER TABLE AFTER REMOVING NON FREQUENT TREE-EXPRESSIONS

□

4.7 THE ETEC PROCEDURE

The frequent tree-expressions mined by Miner1 and Miner2 are only those tree-expressions whose root is the transaction-element. The remaining frequent tree-expressions are embedded inside the frequent tree-expressions found by Miner1 and Miner2. To avoid counting the same embedded tree-expression many times, only embedded k-tree-expressions are considered

from an embedding k-tree-expression. The ETEC procedure mines all the embedded k-tree-expressions from each frequent k-tree-expression. The count of each embedded k-tree-expression is equal to that of the embedding frequent k-tree-expression.

Procedure 5 shows the ETEC algorithm. ETEC starts from the current (the embedding) k-tree-expression and goes through the following steps:

1. It removes the root node from the current k-tree-expression (line 2).
2. If the remaining nodes form a k-tree-expression, it makes the support count of the generated k-tree-expression to be equal to that of the embedding k-tree-expression (lines 6 and 7).
3. If in step 2 a new embedded k-tree-expression is discovered, it makes this new k-tree-expression the current k-tree-expression and it goes back to step 1, otherwise it goes to step 4 (lines 3 to 5).
4. Sums up the support count of the identical embedded k-tree-expressions and removes the duplicates (line 10).

Procedure 5: ETEC

```
begin
        // sub n-tree-expression where n > 1
1:      foreach k-tree-expression
2:              Remove the root node
3:              if new embedded k-tree-expression found
4:                      Make it the current k-tree-expression
5:                      Goto step 2
6:              elseif  nodes form a k-tree-expression
7:                      Increment the count of k-tree-expression with the count of embedding
        tree-expression
8:              endif
9:      endfor
10:     Remove duplicates
        // sub n-tree-expression where n = 1
11:     foreach Labeled-Path
12:             if length of Labeled-Path not less than 2
13:                     Remove the left Tag of the LP-code
14:                     Store the Labeled-path and form a sub 1-tree-expression
15:             else
16:                     Stop
end
```

Example 4.9: Figure 4.6 shows how to find the embedded 2-tree-expression of the 2-tree-expression 'P4*P5' of Table 4.14. To discover the corresponding embedded 2-tree-expressions, ETEC removes the current root element, 'inproceedings', and makes 'title' the new root. Since the sub-tree rooted at 'title' is a 2-tree-expression, ETEC considers it as an embedded 2-tree-expression.

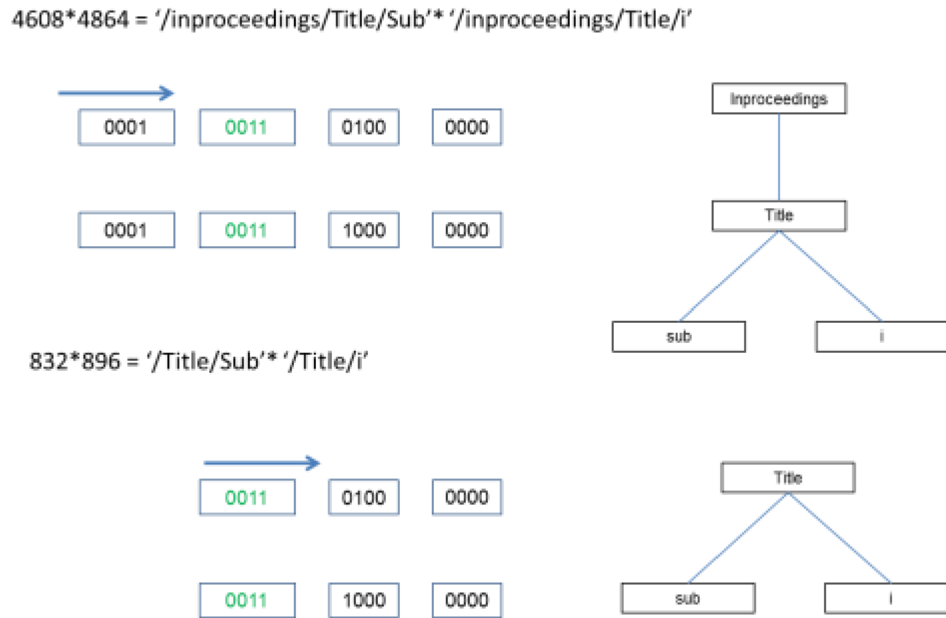


Figure 4.6: Generating sub tree-expression

□

4.8 SUMMARY

Our proposed algorithms, FSPM1 and FSPM2, can be divided into four main procedures, namely, the XML Structural Clustering (XSC) procedure, the Encoder procedure, the Miner procedure, and the Embedded Tree-Expressions Counter (ETEC). The XSC, the Encoder, and the ETEC procedures are shared by both algorithms, but each has its own Miner procedure. The FSPM1 algorithm mines all the candidates from the XML dataset and then removes the non-frequent tree-expressions. Whereas the FSPM2 algorithm removes the non-frequent tree expressions at each pass.

These procedures are explained using a small dataset. Experiments and analysis using the larger dataset is the topic of Chapter 5.

CHAPTER 5

EXPERIMENTAL RESULTS

5.1 INTRODUCTION

In this chapter we present the experimental results and analysis of the proposed algorithms. To study the performance of the proposed algorithms, four sets of experiments were conducted. The first set of experiments was done to compare the performance of FSPM2 with the mabers algorithm [33]. The second set of experiments was done to study the scalability of FSPM2. The third set of experiments was done to check the performance of FSPM2 when the average number of transactions per cluster varies. The last set of experiments was done to compare the performance of FSPM1 and FSPM2.

This chapter is organized as follows. Section 5.2 presents the experimental setup. The performance comparison of FSPM2 and mabers is presented in Section 5.3. Scalability analysis of FSMP2 is given in Section 5.4. The effect of the number of transactions on the performance of FSPM2 is discussed in Section 5.5. Section 5.6 presents the performance comparison of FSPM1 and FSPM2.

5.2 EXPERIMENTAL SETUP

This section presents the machine, the software, the datasets, and the performance measures used in the experiments.

5.2.1 THE MACHINE AND THE SOFTWARE

All the experiments were conducted using a Desktop computer with a Pentium IV processor 3.2 GHZ, 1 GB of RAM, and running Windows XP.

The proposed algorithms were implemented in C#. Visual Studio 2010 was used for executing the program.

5.2.2 THE DATASETS

In the experiments two benchmark (real) datasets and one synthetic dataset were used. The benchmark datasets used were the DBLP and the LineItem datasets obtained from the University of Washington repository [41]. A data generator code was developed in C# to produce the synthetic data.

To study the effect of the number of transactions on the performance of the proposed algorithms, we experimented with different numbers of transactions from each dataset.

Dataset Name	Dataset Segment	No of transactions
DBLP	D1	15000
	D2	25000
	D3	50000
	D4	75000
	D5	100000
	D6	200000
	D7	400000
	D8	600000
	D9	800000
	D10	1000000
LineItem	L1	10000
	L2	20000
	L3	30000
	L4	40000
	L5	50000
Synthetic	S1	10000
	S2	20000
	S3	30000
	S4	40000
	S5	50000

TABLE 5.1: THE DATASETS USED

Table 5.1 shows the number of transactions in each dataset. The datasets were selected based on their uniformity factor. By uniformity here we mean the percentage of transactions per cluster of identically structured transactions. A highly uniform dataset has a high percentage of transactions per cluster. A dataset with a medium uniformity has a medium percentage of transactions per cluster. A non-uniform dataset has a very low percentage of

transactions per cluster. The LineItem dataset is highly uniform whereas the DBLP dataset is of medium uniformity. We made the synthetic dataset non-uniform.

5.2.3 THE PERFORMANCE MEASURE

Elapsed time was used to evaluate the performance of the proposed algorithms. Elapsed Time is the total time spent to find the frequent structural patterns from an XML dataset. It is averaged over many runs. For FSPM1 and FSPM2, this time is the sum of the I/O time, the encoding time, the clustering time, and the mining time.

5.3 FSPM2 VS. MABERS

In this section we present the performance comparison of FSPM2 and the mabers algorithm. Three sets of experiments were done to compare the performance of the algorithms. These sets of experiments were done using the DBLP, the LineItem, and the synthetic datasets.

5.3.1 FSPM2 VS. MABERS USING THE DBLP DATASET

In the first set of experiments, FSPM2 and the mabers algorithms were run several times using the D5 dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure 5.1 shows the results of these experiments.

As it can be seen from Figure 5.1, FSPM2 showed significantly better performance than mabers in all the experiments. FSPM2 was faster by up to 182 times.

FSPM2 performed better than mabers because the DBLP dataset contains many transactions with similar structure. As a result many transactions were put into the same cluster. FSPM2 processes one transaction per cluster whereas the mabers algorithm processes each transaction individually. For example, if there are 50,000 transactions in a dataset and they were put into 100 clusters, FSPM2 will process only 100 transactions, one transaction from each cluster, whereas mabers will process all the 50,000 transaction individually. The time taken by FSPM2 to cluster transactions is low because the clustering is done in one scan.

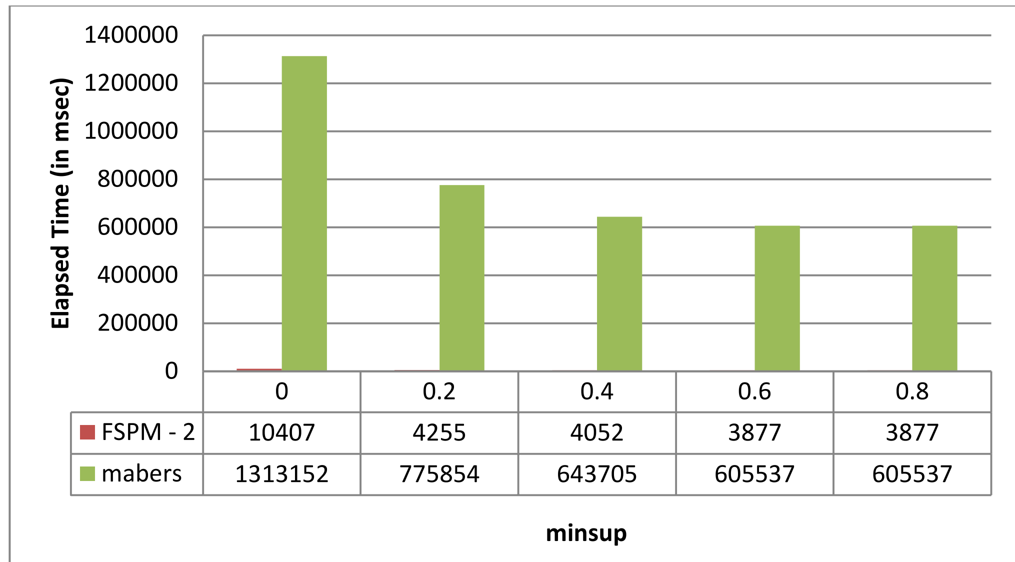


Figure 5.1: Elapsed Time: FSPM2 VS Mabers using D5

D5	FSPM2	mabers	
minsup	Elapsed Time (in msec)	Elapsed Time (in msec)	Gain
0	10407	1313152	**126.18**
10	4472	796830	**178.18**
20	4255	775854	**182.34**
30	4255	775854	**182.34**
40	4052	643705	**158.86**
50	3915	621831	**158.83**
60	3877	605537	**156.19**
70	3877	605537	**156.19**
80	3877	605537	**156.19**
90	3877	605537	**156.19**

TABLE 5.2: ELAPSED TIME: FSPM2 VS MABERS FOR D5

Table 5.2 shows the elapsed times for FSPM2 and mabers algorithms. The last column shows the *gain*, which is the elapsed time of mabers divided

by that of FSPM2. The gain was more when the minsup was 0.1, 0.2, or 0.3. This is because mabers was not able to prune many transactions earlier. The gain started decreasing when the minsup was 0.4 or 0.5. This is because the percentage of transactions in the dataset pruned earlier by mabers was less than the percentage of clusters of the dataset pruned by FSPM2. The mabers algorithm took less time when the minsup was 0.4 or 0.5 compared to the time it took when the minsup was 0.1, 0.2, or 0.3. The gain remained the same when the minsup was 0.6, 0.7, 0.8, and 0.9 as the count of tree-expressions remained the same at these minsups. The count of tree-expressions can be seen from Table 5.3.

minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6	TE_7	TE_8	TE_9	TE_{10}	TE_{11}
0	17	33	167	616	1307	1777	1626	1018	435	124	22	2
0.1	17	11	28	56	70	56	28	8	1	0	0	0
0.2	17	11	27	50	55	36	13	2	0	0	0	0
0.3	17	11	27	50	55	36	13	2	0	0	0	0
0.4	17	9	21	35	35	21	7	1	0	0	0	0
0.5	17	7	16	23	18	7	1	0	0	0	0	0
0.6	17	7	15	20	15	6	1	0	0	0	0	0
0.7	17	7	15	20	15	6	1	0	0	0	0	0
0.8	17	7	15	20	15	6	1	0	0	0	0	0
0.9	17	7	15	20	15	6	1	0	0	0	0	0

TABLE 5.3: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR D5

For the results of dataset segments D1 to D4 please refer to Appendix A.

5.3.2 FSPM2 VS. MABERS USING THE LINEITEM DATASET

In the second set of the experiments FSPM2 and the mabers algorithms were run several times using L1 to L5 LINEITEM dataset segments. Figure 5.2 shows the results of the experiments.

In all the experiments FSPM2 showed significantly better performance than mabers. FSPM2 was faster by up to 49,516 times.

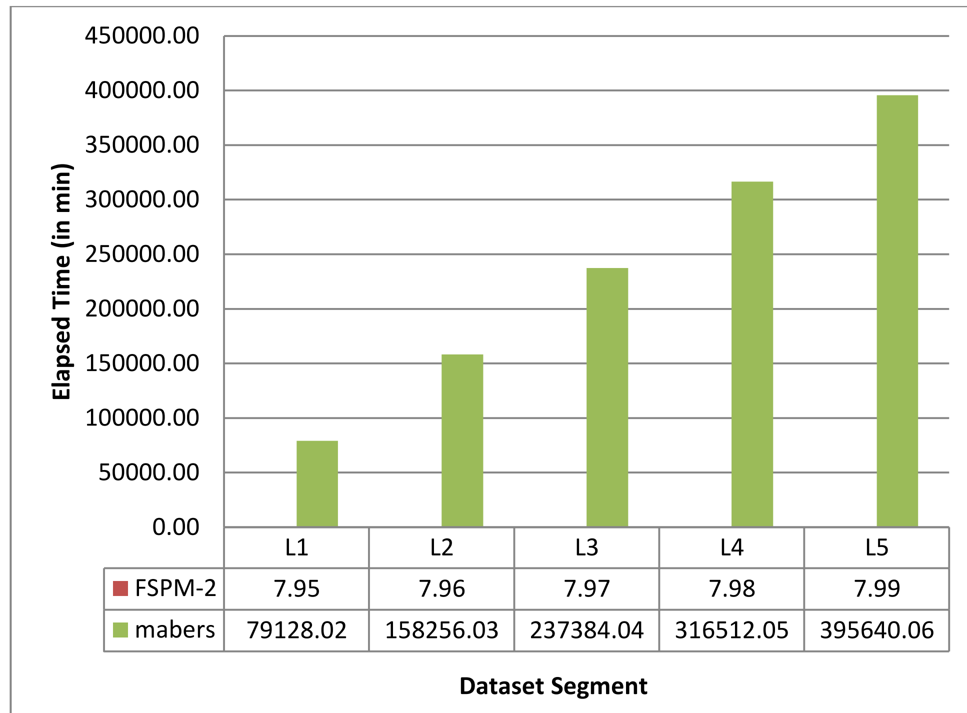


Figure 5.2: Elapsed Time: FSPM2 VS Mabers for L1 – L5

FSPM2 performed better than mabers because the LINEITEM dataset consists of transactions with similar structure. As a result, all transactions were put into the same cluster. FSPM2 processes one transaction per cluster while mabers processes individual transactions. The clustering of the transactions by FSPM2 doesn't take much time because it is achieved in one pass.

L1 – L5	FSPM2	mabers	Gain
Dataset Segments	Elapsed Time (in min)	Elapsed Time (in min)	
L1	7.95	79128.02	**9953.21**
L2	7.96	158256.03	**19881.41**
L3	7.97	237384.04	**29784.70**
L4	7.98	316512.05	**39663.16**
L5	7.99	395640.06	**49516.90**

TABLE 5.4: ELAPSED TIME: FSPM2 VS MABERS FOR L1 – L5

Table 5.4 shows elapsed time for FSPM2 and the mabers algorithms. The last column shows the *gain*, which is the elapsed time of mabers divided by that of FSPM2. As the time required for mining tree-expressions from one cluster is more than 7 minutes, it will be time-consuming for mining tree-expressions using the mabers algorithm. So for the mabers algorithm we took 100 transactions. The time obtained by running the algorithm for the 100 transactions is extrapolated to get the representative times for all the other dataset segments. The mining time for each of the dataset segments L1 to L5

remains the same as they all have the same structure. The clustering time is proportional to the size of the dataset segments. All the tree-expressions have a count equal to the number of transactions. The count of tree-expressions can be seen in Table 5.5.

Dataset Segment	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6	TE_7	TE_8
L1	17	17	120	560	1820	4368	8008	11440	12870
L2	17	17	120	560	1820	4368	8008	11440	12870
L3	17	17	120	560	1820	4368	8008	11440	12870
L4	17	17	120	560	1820	4368	8008	11440	12870
L5	17	17	120	560	1820	4368	8008	11440	12870

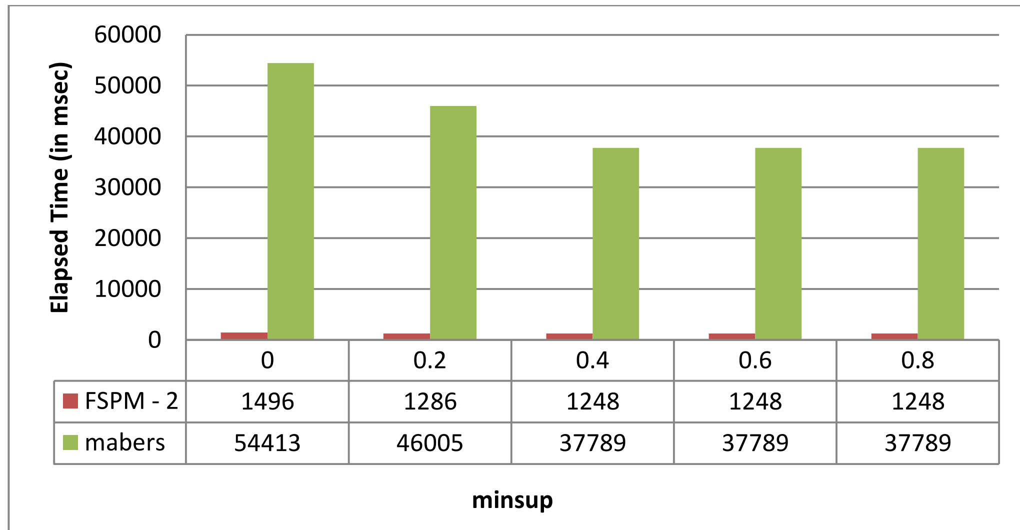
Dataset Segment	TE_9	TE_{10}	TE_{11}	TE_{12}	TE_{13}	TE_{14}	TE_{15}	TE_{16}
L1	11440	8008	4368	1820	560	120	16	1
L2	11440	8008	4368	1820	560	120	16	1
L3	11440	8008	4368	1820	560	120	16	1
L4	11440	8008	4368	1820	560	120	16	1
L5	11440	8008	4368	1820	560	120	16	1

TABLE 5.5: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR L1 – L5

5.3.3 FSPM2 VS. MABERS USING THE SYNTHETIC DATASET

In the third set of experiments, FSPM2 and the mabers algorithms were run several times using the S5 SYNTHETIC dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure 5.3 shows the results of the experiments.

In all the experiments, FSPM2 showed significantly better performance than mabers. FSPM2 was faster by up to 37 times.



	0	0.2	0.4	0.6	0.8
FSPM - 2	1496	1286	1248	1248	1248
mabers	54413	46005	37789	37789	37789

Figure 5.3: Elapsed Time: FSPM2 VS Mabers using S5

The SYNTHETIC dataset contains few transactions with similar structure. The similar transactions were put into the same cluster. FSPM2 processes one transaction per cluster while mabers processes individual transactions. This is because the percentage of transactions in the dataset pruned earlier by mabers was less than the percentage of clusters of the dataset pruned by FSPM2. The mabers algorithm took less time when the minsup was 0.2 or 0.3 compared to the time it took when the minsup was 0.1. The count of tree-expression can be seen from Table 5.7.

S5	FSPM2	mabers	
minsup	Elapsed Time (in msec)	Elapsed Time (in msec)	Gain
0	1496	54413	**36.37**
0.1	1302	48595	**37.32**
0.2	1286	46005	**35.77**
0.3	1267	44052	**34.77**
0.4	1248	37789	**30.28**
0.5	1248	37789	**30.28**
0.6	1248	37789	**30.28**
0.7	1248	37789	**30.28**
0.8	1248	37789	**30.28**
0.9	1248	37789	**30.28**

TABLE 5.6: ELAPSED TIME: FSPM2 VS MABERS FOR S5

minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6	TE_7
0	42	100	308	784	877	472	128	14
0.1	40	7	15	20	15	0	0	0
0.2	39	7	15	20	0	0	0	0
0.3	39	7	15	0	0	0	0	0
0.4	39	7	0	0	0	0	0	0
0.5	37	7	0	0	0	0	0	0
0.6	36	2	0	0	0	0	0	0
0.7	34	1	0	0	0	0	0	0
0.8	29	1	0	0	0	0	0	0
0.9	29	1	0	0	0	0	0	0

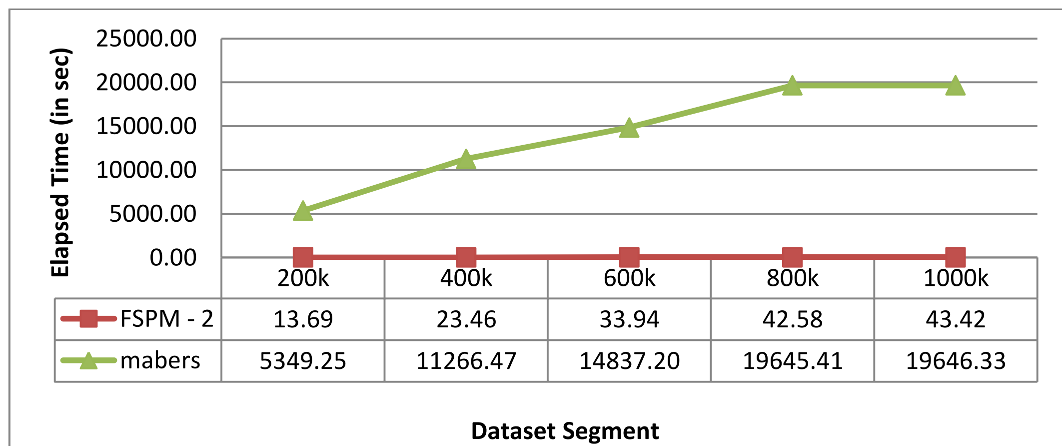
TABLE 5.7: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR S5

For the results of dataset segments S1 to S4 please refer to Appendix B.

5.4 SCALIBILITY OF FSPM2

In this set of experiments, FSPM2 and the mabers algorithms were run several times using the DBLP dataset segments D6 to D10. This DBLP dataset is available at [42]. For the scalability analysis we considered transactions of 200k and kept increasing the transactions till 1 million. The numbers of transactions in the dataset segments were 200k, 400k, 600k, 800k, and 1 million. Minsup of 0.1 was used while running the experiments. Figure 5.4 shows the results of the experiments.

In all the experiments, FSPM2 showed significantly better performance than mabers. FSPM2 was faster by up to 480 times.



	200k	400k	600k	800k	1000k
FSPM - 2	13.69	23.46	33.94	42.58	43.42
mabers	5349.25	11266.47	14837.20	19645.41	19646.33

Figure 5.4: Elapsed Time FSPM2 VS Mabers using D6-D10

Scalability	FSPM-2				Mabers			Gain
Dataset Segment	Clustering Time (in sec)	I/O Time (in sec)	Mining Time (in sec)	Total Time (in sec)	I/O Time (in sec)	Mining Time (in sec)	Total Time (in sec)	
D6	7.09	2.81	3.78	**13.69**	5.62	5337.63	**5349.25**	**390.88**
D7	12.17	5.69	5.60	**23.46**	11.37	11246.60	**11266.47**	**480.30**
D8	18.50	8.86	6.59	**33.94**	17.71	14808.29	**14837.20**	**437.15**
D9	24.25	11.09	7.23	**42.58**	22.18	19607.98	**19645.41**	**461.41**
D10	24.64	11.55	7.23	**43.42**	23.10	19607.98	**19646.33**	**452.46**

TABLE 5.8: ELAPSED TIME FOR D6-D10

Table 5.8 shows the elapsed time for FSPM2 and the mabers algorithms for Dataset segments D6 to D10. The last column shows the gain, which is the elapsed time of mabers divided by that of FSPM2. The clustering time increases from segments D6 to D10 as the clustering time is proportional to the dataset size. The elapsed time increases linearly with the size of the dataset segment. The count of tree-expressions can be seen in Table 5.9

Dataset Segment	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6	TE_7	TE_8	TE_9
D6	21	10	37	84	126	126	84	36	9	1
D7	22	10	37	84	126	126	84	36	9	1
D8	22	10	37	84	126	126	84	36	9	1
D9	22	10	37	84	126	126	84	36	9	1
D10	22	10	37	84	126	126	84	36	9	1

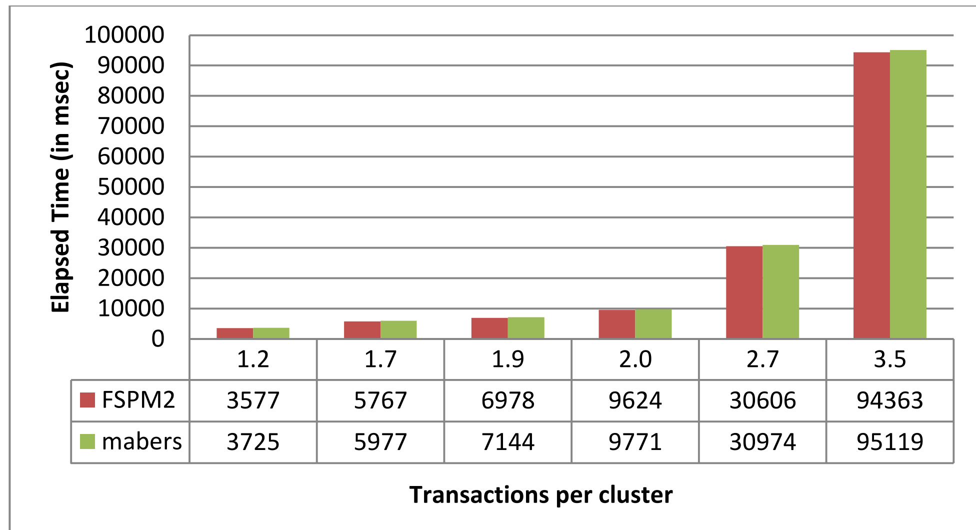
TABLE 5.9: COUNT OF TREE-EXPRESSIONS FOR D6-D10

5.5 FSPM2 VS. MABERS WITH NON-UNIFORM DATASETS

This section explains how the performance of FSPM2 is affected when the input dataset is non-uniform. In the experiments a number of non-uniform synthetic datasets were generated. Each dataset had 50,000 transactions. The number of transactions per cluster varied between 1 and 5, which is a small cluster size and also the worst case scenario for FSPM2. From Table 5.10 and Figure 5.5, we can conclude that in the worst case, the performance of FSPM2 is the same as that of mabers.

	FSPM2	Mabers
Transactions per cluster	Elapsed Time (in msec)	Elapsed Time (in msec)
1.2	3577	3725
1.7	5767	5977
1.9	6978	7144
2.0	9624	9771
2.7	30606	30974
3.5	94363	95119

TABLE 5.10: ELAPSED TIME FSPM2 VS. MABERS



	1.2	1.7	1.9	2.0	2.7	3.5
FSPM2	3577	5767	6978	9624	30606	94363
mabers	3725	5977	7144	9771	30974	95119

Figure 5.5: Elapsed Time FSPM2 VS mabers

5.6 COMPARISON OF FSPM1 VS. FSPM2

The description of the FSPM1 and FSPM2 algorithms are explained in Chapter 4. Both the algorithm work equally well. In this section we explain why the FSPM1 algorithm performs better than FSPM2 when the minsup is zero. The FSPM2 algorithm has a small overhead of storing the generated tree-expressions back into the cluster table. That is why we observe that when the minsup is 0.0, the FSPM1 algorithm generates tree-expressions in comparatively less time than FSPM2. The FSPM1 algorithm also works better than the FSPM2 algorithm when all the tree-expressions are frequent as we have seen in the case of the LINESITEM dataset. So the use of the FSPM1

algorithm is recommended when we need to mine tree-expressions for minsup 0.0; and when all the tree-expressions are frequent as in the case of the LINEITEM dataset.

We present the performance comparison of the FSPM1 and the FSPM2 algorithms. Three sets of experiments were done to compare the algorithms. These sets of experiments were done using the DBLP, the LineItem, and the synthetic datasets.

5.6.1 FSPM1 VS. FSPM2 USING THE DBLP DATASET

In the first set of the experiments, FSPM1 and FSPM2 were run several times using the D5 DBLP dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure 5.6 shows the results of these experiments.

In all the experiments, FSPM2 showed better performance than FSPM1 except when minsup was 0.0. FSPM2 was faster by up to 33 times.

FSPM2 performed better than FSPM1 when the minsup greater than zero as FSPM2 removes non-frequent tree-expressions at the end of each level whereas FSPM1 doesn't remove non-frequent tree-expressions between

levels. The tree expressions that don't satisfy the minsup are called non-frequent.

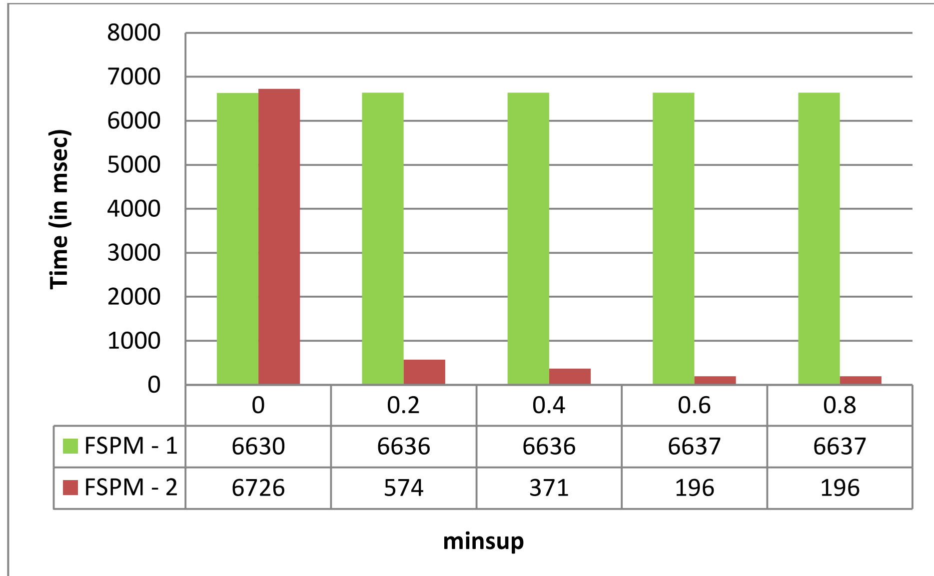


Figure 5.6: FSPM1 VS FSPM2 Mining Time for D5

FSPM2's methodology reduces the candidate tree-expressions for the next level. Whereas FSPM1 takes constant time for generating the tree-expressions and it takes some extra time for removing the non-frequent tree-expressions at the end of last level. Therefore gain improves as the minsup is increased. The count of the tree-expressions at each level can be seen in Table 5.12. It can be noticed that as the minsup increases the count of tree-expressions is reduced.

D5	FSPM1	FSPM2	Gain
minsup	Mining Time (in msec)	Mining Time (in msec)	
0	6630	6726	**0.99**
0.1	6636	791	**8.39**
0.2	6636	574	**11.56**
0.3	6636	574	**11.56**
0.4	6636	371	**17.89**
0.5	6636	234	**28.36**
0.6	6637	196	**33.86**
0.7	6637	196	**33.86**
0.8	6637	196	**33.86**
0.9	6637	196	**33.86**

TABLE 5.11: FSPM1 VS FSPM2 MINING TIME FOR D5

minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6	TE_7	TE_8	TE_9	TE_{10}	TE_{11}
0	17	21	33	167	616	1307	1777	1626	1018	435	124	22
0.1	17	11	28	56	70	56	28	8	1	0	0	0
0.2	17	11	27	50	55	36	13	2	0	0	0	0
0.3	17	11	27	50	55	36	13	2	0	0	0	0
0.4	17	9	21	35	35	21	7	1	0	0	0	0
0.5	17	7	16	23	18	7	1	0	0	0	0	0
0.6	17	7	15	20	15	6	1	0	0	0	0	0
0.7	17	7	15	20	15	6	1	0	0	0	0	0
0.8	17	7	15	20	15	6	1	0	0	0	0	0
0.9	17	7	15	20	15	6	1	0	0	0	0	0

TABLE 5.12: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR D5

For the results of dataset segments D1 to D4 please refer to Appendix C.

5.6.2 FSPM1 VS. FSPM2 USING THE LINEITEM DATASET

In the second set of experiments, FSPM1 and FSPM2 were run several times using the LINEITEM dataset segments L1 to L5. Figure 5.7 shows the results of these experiments.

In all the experiments, FSPM1 showed better performance than FSPM2. FSPM1 was faster by 1.05 times.

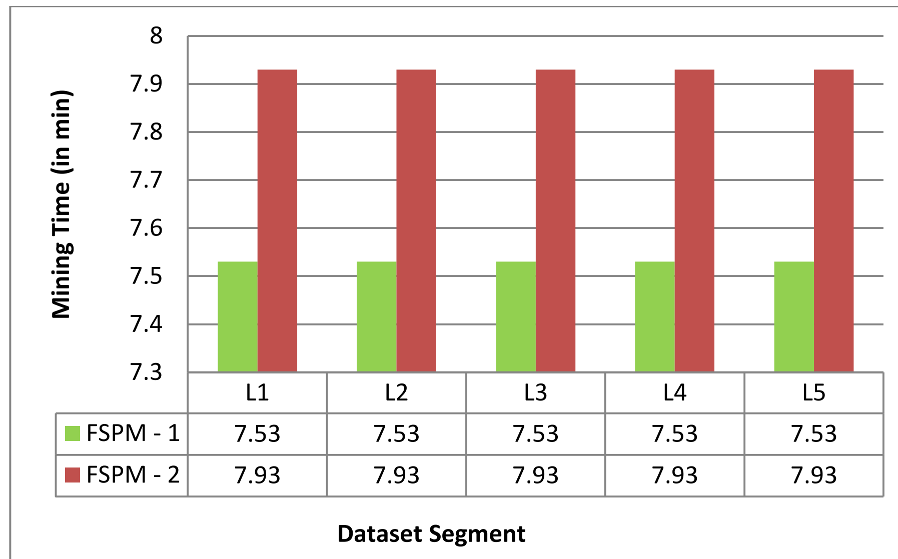


Figure 5.7: FSPM1 VS FSPM2 Mining Time for L1 – L5

FSPM1 performs better than FSPM2 as all the tree-expressions in the dataset segments L1 to L5 are frequent. The dataset segments L1 to L5 contain only one cluster. FSPM2 has a small overhead which is explained in Section 5.6.

L1 - L5 Dataset Segment	FSPM1 Mining Time (in min)	FSPM2 Mining Time (in min)	Gain
L1	7.53	7.93	**1.05**
L2	7.53	7.93	**1.05**
L3	7.53	7.93	**1.05**
L4	7.53	7.93	**1.05**
L5	7.53	7.93	**1.05**

TABLE 5.13: FSPM1 VS FSPM2 MINING TIME FOR L1 – L5

The count of the tree-expressions at each level can be seen in Table 5.14. It can be noticed that the count of tree-expressions for the dataset segments L1 to L5 is the same as all of them have the same structure. This table is divided into two parts for better visualization.

Dataset Segment	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6	TE_7	TE_8
L1	17	17	120	560	1820	4368	8008	11440	12870
L2	17	17	120	560	1820	4368	8008	11440	12870
L3	17	17	120	560	1820	4368	8008	11440	12870
L4	17	17	120	560	1820	4368	8008	11440	12870
L5	17	17	120	560	1820	4368	8008	11440	12870

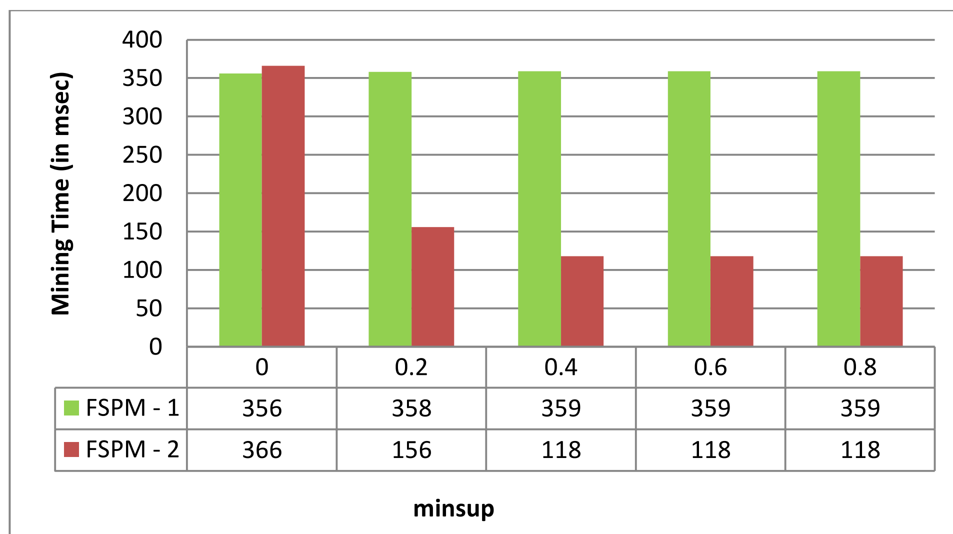
Dataset Segment	TE_9	TE_{10}	TE_{11}	TE_{12}	TE_{13}	TE_{14}	TE_{15}	TE_{16}
L1	11440	8008	4368	1820	560	120	16	1
L2	11440	8008	4368	1820	560	120	16	1
L3	11440	8008	4368	1820	560	120	16	1
L4	11440	8008	4368	1820	560	120	16	1
L5	11440	8008	4368	1820	560	120	16	1

TABLE 5.14: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR L1 – L5

5.6.3 FSPM1 VS. FSPM2 USING THE SYNTHETIC DATASET

In the third set of experiments, FSPM1 and FSPM2 were run several times using the S5 SYNTHETIC dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure 5.8 shows the results of these experiments.

In all the experiments, FSPM2 showed better performance than FSPM1 except when minsup was 0.0. FSPM2 was faster by up to 3 times.



	0	0.2	0.4	0.6	0.8
FSPM - 1	356	358	359	359	359
FSPM - 2	366	156	118	118	118

Figure 5.8: FSPM1 VS FSPM2 Mining Time for S5

FSPM2 performed better than FSPM1 when the minsup greater than zero as FSPM2 removes non-frequent tree-expressions at the end of each level whereas FSPM1 doesn't remove non-frequent tree-expressions between

levels. The tree expressions that don't satisfy the minsup are called non-frequent.

S5	FSPM1	FSPM2	Gain
minsup	Mining Time (in msec)	Mining Time (in msec)	
0	356	366	**0.97**
0.1	358	172	**2.08**
0.2	358	156	**2.29**
0.3	359	137	**2.62**
0.4	359	118	**3.04**
0.5	359	118	**3.04**
0.6	359	118	**3.04**
0.7	359	118	**3.04**
0.8	359	118	**3.04**
0.9	359	118	**3.04**

TABLE 5.15: FSPM1 VS FSPM2 MINING TIME FOR S5

FSPM2's methodology reduces the candidate tree-expressions for the next level. Whereas FSPM1 takes constant time for generating the tree-expressions and it takes some extra time for removing the non-frequent tree-expressions at the end of last level. Therefore gain improves as the minsup is increased.

The count of the tree-expressions at each level can be seen in Table 5.16. It can be noticed that as the minsup increases the count of the tree-expressions is reduced.

minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6	TE_7
0	42	100	308	784	877	472	128	14
0.1	40	7	15	20	15	0	0	0
0.2	39	7	15	20	0	0	0	0
0.3	39	7	15	0	0	0	0	0
0.4	39	7	0	0	0	0	0	0
0.5	37	7	0	0	0	0	0	0
0.6	36	2	0	0	0	0	0	0
0.7	34	1	0	0	0	0	0	0
0.8	29	1	0	0	0	0	0	0
0.9	29	1	0	0	0	0	0	0

TABLE 5.16: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR S5

For the results of dataset segments S1 to S4 please refer to Appendix D.

5.8 SUMMARY

In this section we presented the analysis of the proposed algorithms (FSPM1 and FSPM2) based on experiments on real XML benchmark datasets and synthetic datasets. Our proposed algorithms outperform the mabers algorithm. The proposed algorithms process one instance of several transactions. The gain obtained from running the algorithm is best for highly uniform datasets. The gain was about 49,516 times for the LineItem D5 dataset segment. In the worst case, the performance of FSPM2 was the same as mabers. The scalability analysis confirms that our proposed algorithm FSPM2 is scalable to a huge number of transactions.

CHAPTER 6

CONCLUSION AND FUTURE WORKS

In this chapter we summarize our thesis and propose ways in which this work can be extended in the future. The summary of our thesis is given in Section 6.1. Then, Section 6.2 lists the future work.

6.1 THESIS SUMMARY

In this section we summarize our thesis. The main objective of this thesis was to propose an efficient frequent structure mining algorithm. To achieve this objective, first we presented a literature review on frequent structural pattern mining in Chapter 3. The review showed there are a few algorithms to mine frequent structural patterns. Algorithms that mine association rules from XML documents can be classified into three, namely, content-based algorithms, structure-based algorithms, and content-structure-based algorithms. The content-based algorithms mine association rules only from the content (values) of an XML dataset. The structure-based algorithms, mine association rules only from the structural relationships found in an

XML dataset. Finally, the content-structure-based algorithms mine association rules from both the content and the structural relationships found in an XML dataset.

The proposed algorithms were designed to efficiently mine frequent structural patterns from XML datasets. Two Frequent Structural Pattern Mining algorithms, namely, FSPM1 and FSPM2 are proposed. Each of the two algorithms consists of four main procedures, namely, the XML Structural Clustering (XSC) procedure, the Encoder procedure, the Miner procedure, and the Embedded Tree-Expressions Counter (ETEC). The XSC, the Encoder, and the ETEC procedures are shared by both algorithms, but each has its own Miner procedure. To the best of our knowledge, our proposed algorithms are the first algorithms to use clustering for mining frequent substructures from XML datasets.

To validate the previous goals and solutions, experiments were conducted using synthetic and real life XML benchmark datasets. The results of these experiments are as follows:

- The Clustering phase takes one dataset scan.
- The Clustering time is proportional to the dataset size.

- The gain obtained from running the algorithm is best for highly uniform datasets. The gain was about 49,516 times.
- In the worst case the performance of FSPM2 is the same as mabers.

The proposed algorithms are used to find frequent structural patterns. Finding frequent patterns has many applications, such as, querying/browsing information sources, indexing, and building wrappers. It also plays an essential role in many data mining tasks such as associations, correlations, classification, clustering, and many other interesting relationships among data.

6.2 FUTURE WORK

Algorithms that mine association rules from XML documents can be classified into three, namely, content-based algorithms, structure-based algorithms, and content-structure-based algorithms. This thesis work focuses on mining frequent structural patterns using the structure of the XML document. The possible future work for this thesis is to come up with a new clustering algorithm which clusters the XML Dataset based on structure and content information. Another future work can be to use our algorithm to

mine the frequent patterns using content and structure information of an XML document.

APPENDIX A

FSPM2 VS. MABERS USING THE DBLP DATASET D1

In this set of experiments, FSPM2 and the mabers algorithms were run several times using the D1 DBLP dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure A.1 shows the results of the experiments.

In all the experiments, FSPM2 showed significantly better performance than mabers. FSPM2 was faster by up to 162 times.

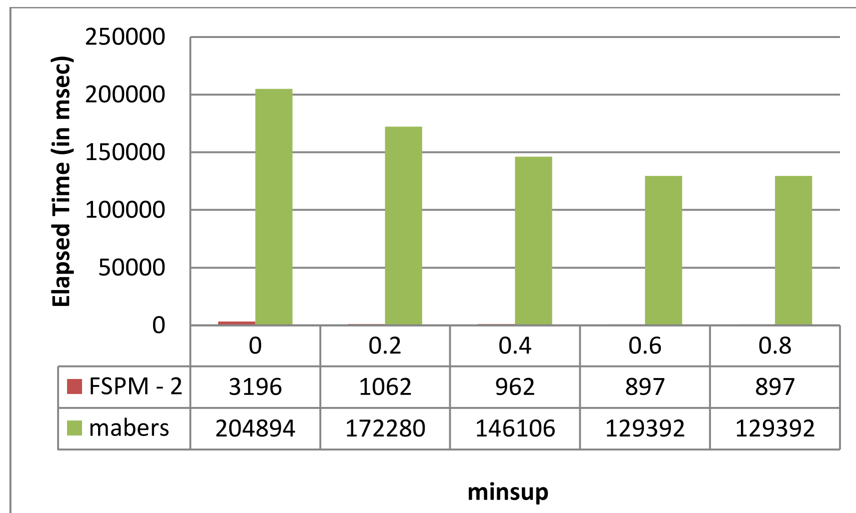


Figure A.1: Elapsed Time: FSPM2 VS Mabers using D1

D1	FSPM2	mabers	Gain
minsup	Elapsed Time (in msec)	Elapsed Time (in msec)	
0	3196	204894	**64.11**
0.1	1151	172863	**150.19**
0.2	1062	172280	**162.22**
0.3	1062	172280	**162.22**
0.4	962	146106	**151.88**
0.5	897	129392	**144.25**
0.6	897	129392	**144.25**
0.7	897	129392	**144.25**
0.8	897	129392	**144.25**
0.9	897	129392	**144.25**

TABLE A.1: ELAPSED TIME: FSPM2 VS MABERS FOR D1

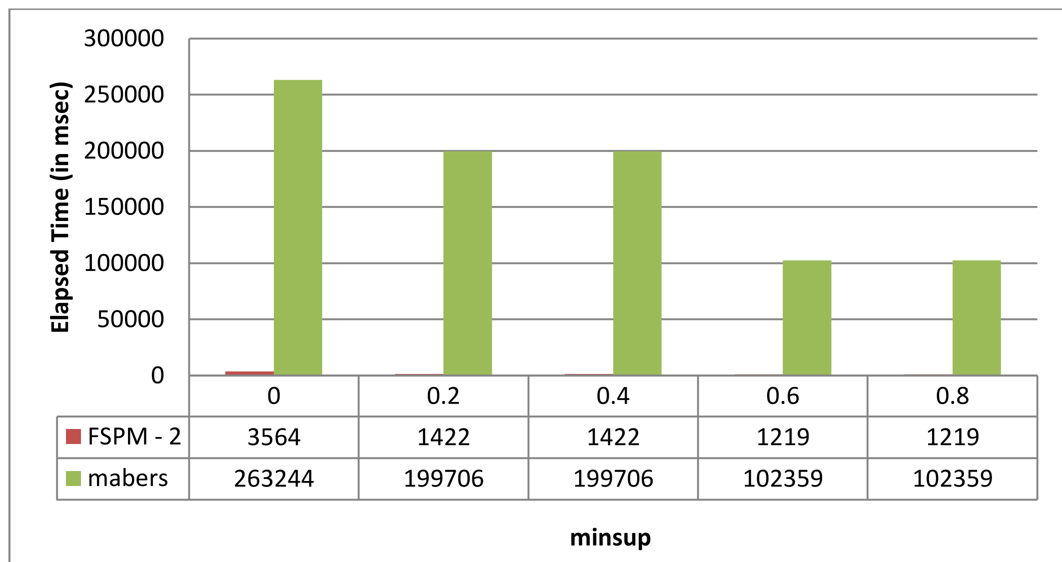
minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6	TE_7	TE_8	TE_9	TE_{10}
0	14	23	108	359	705	901	773	440	159	33	3
0.1	14	10	28	56	70	56	28	8	1	0	0
0.2	14	10	27	50	55	36	13	2	0	0	0
0.3	14	10	27	50	55	36	13	2	0	0	0
0.4	14	10	21	35	35	21	7	1	0	0	0
0.5	14	8	15	20	15	6	1	0	0	0	0
0.6	14	8	15	20	15	6	1	0	0	0	0
0.7	8	8	15	20	15	6	1	0	0	0	0
0.8	8	8	15	20	15	6	1	0	0	0	0
0.9	8	8	15	20	15	6	1	0	0	0	0

TABLE A.2: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR D1

FSPM2 VS. MABERS USING THE DBLP DATASET D2

In this set of experiments, FSPM2 and the mabers algorithms were run several times using the D2 DBLP dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure A.2 shows the results of the experiments.

In all the experiments, FSPM2 showed significantly better performance than mabers. FSPM2 was faster by up to 143 times.



	0	0.2	0.4	0.6	0.8
FSPM - 2	3564	1422	1422	1219	1219
mabers	263244	199706	199706	102359	102359

Figure A.2: Elapsed Time: FSPM2 VS Mabers using D2

D2	FSPM2	mabers	Gain
minsup	Elapsed Time (in msec)	Elapsed Time (in msec)	
0	3564	263244	**73.86**
0.1	1456	209079	**143.60**
0.2	1422	199706	**140.44**
0.3	1422	199706	**140.44**
0.4	1422	199706	**140.44**
0.5	1219	102359	**83.97**
0.6	1219	102359	**83.97**
0.7	1219	102359	**83.97**
0.8	1219	102359	**83.97**
0.9	1219	102359	**83.97**

TABLE A.3: ELAPSED TIME: FSPM2 VS MABERS FOR D2

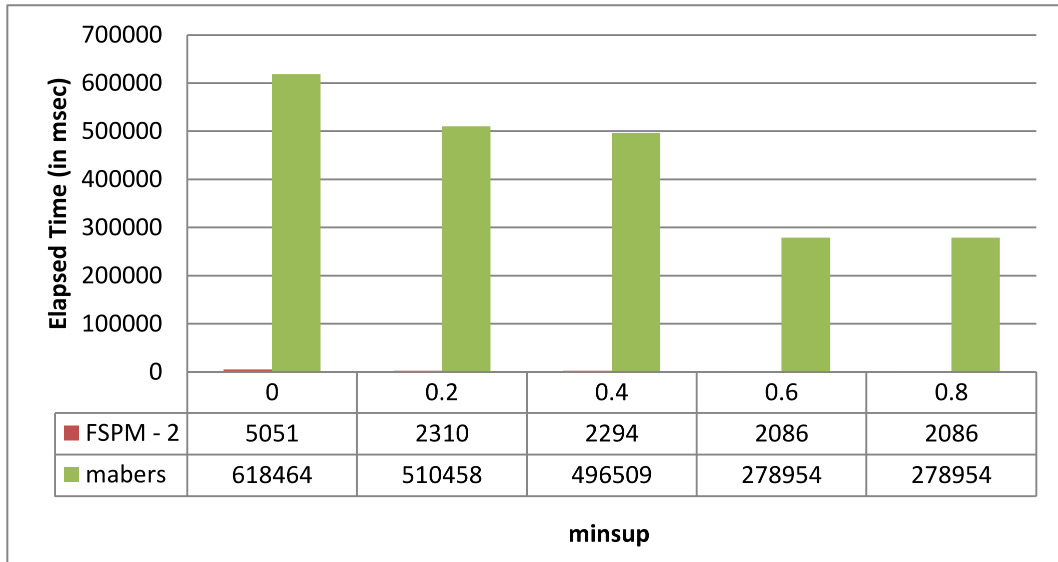
minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6	TE_7	TE_8	TE_9	TE_{10}
0	14	23	113	390	776	991	838	465	163	33	3
0.1	14	10	28	56	70	56	28	8	1	0	0
0.2	14	10	27	50	55	36	13	2	0	0	0
0.3	14	10	27	50	55	36	13	2	0	0	0
0.4	14	10	27	50	55	36	13	2	0	0	0
0.5	14	8	15	20	15	6	1	0	0	0	0
0.6	14	8	15	20	15	6	1	0	0	0	0
0.7	13	8	15	20	15	6	1	0	0	0	0
0.8	8	8	15	20	15	6	1	0	0	0	0
0.9	8	8	15	20	15	6	1	0	0	0	0

TABLE A.4: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR D2

FSPM2 VS. MABERS USING THE DBLP DATASET D3

In this set of experiments, FSPM2 and the mabers algorithms were run several times using the D3 DBLP dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure A.3 shows the results of the experiments.

In all the experiments, FSPM2 showed significantly better performance than mabers. FSPM2 was faster by up to 223 times.



	0	0.2	0.4	0.6	0.8
FSPM - 2	5051	2310	2294	2086	2086
mabers	618464	510458	496509	278954	278954

Figure A.3: Elapsed Time: FSPM2 VS Mabers using D3

D3	FSPM2	mabers	Gain
minsup	Elapsed Time (in msec)	Elapsed Time (in msec)	
0	5051	618464	**122.44**
0.1	2418	541173	**223.81**
0.2	2310	510458	**220.98**
0.3	2310	510458	**220.98**
0.4	2294	496509	**216.44**
0.5	2171	344657	**158.75**
0.6	2086	278954	**133.73**
0.7	2086	278954	**133.73**
0.8	2086	278954	**133.73**
0.9	2086	278954	**133.73**

TABLE A.5: ELAPSED TIME: FSPM2 VS MABERS FOR D3

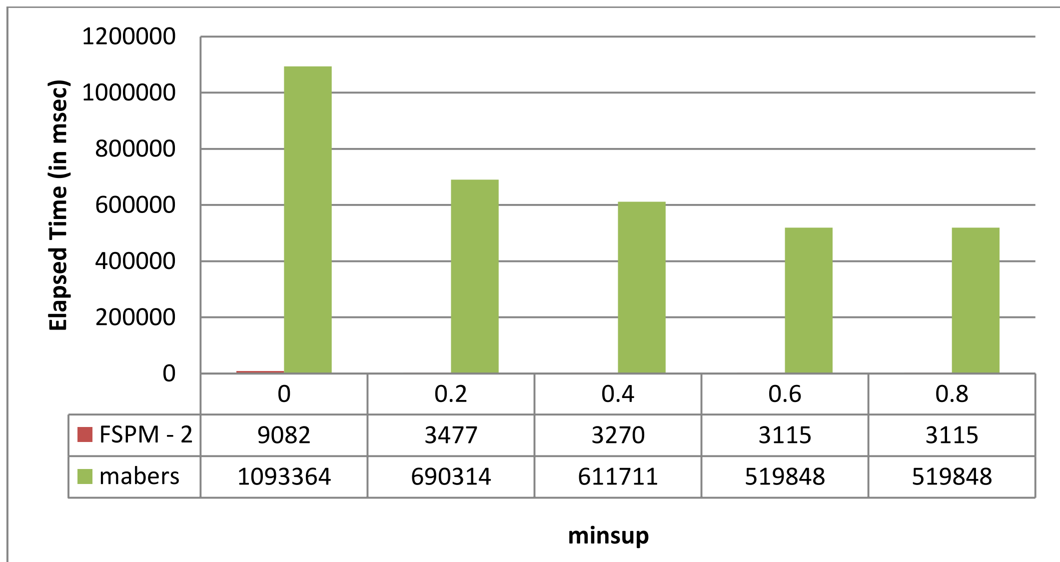
minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6	TE_7	TE_8	TE_9	TE_{10}
0	14	26	126	438	874	1118	944	520	179	35	3
0.1	14	10	28	56	70	56	28	8	1	0	0
0.2	14	10	27	50	55	36	13	2	0	0	0
0.3	14	9	27	50	55	36	13	2	0	0	0
0.4	14	9	25	47	54	36	13	2	0	0	0
0.5	14	8	21	35	35	21	7	1	0	0	0
0.6	14	8	15	20	15	6	1	0	0	0	0
0.7	8	8	15	20	15	6	1	0	0	0	0
0.8	8	8	15	20	15	6	1	0	0	0	0
0.9	8	8	15	20	15	6	1	0	0	0	0

TABLE A.6: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR D3

FSPM2 VS. MABERS USING THE DBLP DATASET D4

In this set of experiments, FSPM2 and the mabers algorithms were run several times using the D4 DBLP dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure A.4 shows the results of the experiments.

In all the experiments, FSPM2 showed significantly better performance than mabers. FSPM2 was faster by up to 237 times.



	0	0.2	0.4	0.6	0.8
FSPM - 2	9082	3477	3270	3115	3115
mabers	1093364	690314	611711	519848	519848

Figure A.4: Elapsed Time: FSPM2 VS Mabers using D4

D4	FSPM2	mabers	Gain
minsup	Elapsed Time (in msec)	Elapsed Time (in msec)	
0	9082	1093364	**120.39**
0.1	3615	856970	**237.06**
0.2	3477	690314	**198.54**
0.3	3477	690314	**198.54**
0.4	3270	611711	**187.07**
0.5	3270	611711	**187.07**
0.6	3115	519848	**166.89**
0.7	3115	519848	**166.89**
0.8	3115	519848	**166.89**
0.9	3115	519848	**166.89**

TABLE A.7: ELAPSED TIME: FSPM2 VS MABERS FOR D4

minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6	TE_7	TE_8	TE_9	TE_{10}	TE_{11}
0	16	31	156	567	1203	1654	1539	981	426	123	22	2
0.1	16	10	28	56	70	56	28	8	1	0	0	0
0.2	15	10	27	50	55	36	13	2	0	0	0	0
0.3	15	10	27	50	55	36	13	2	0	0	0	0
0.4	15	9	21	35	35	21	7	1	0	0	0	0
0.5	15	9	21	35	35	21	7	1	0	0	0	0
0.6	14	7	15	20	15	6	1	0	0	0	0	0
0.7	8	7	15	20	15	6	1	0	0	0	0	0
0.8	8	7	15	20	15	6	1	0	0	0	0	0
0.9	8	7	15	20	15	6	1	0	0	0	0	0

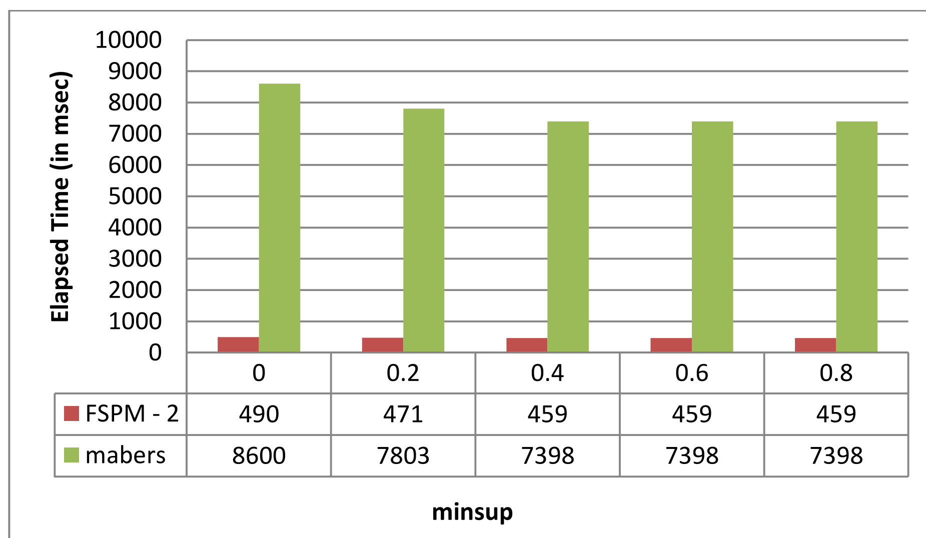
TABLE A.8: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR D4

APPENDIX B

FSPM2 VS. MABERS USING THE SYNTHETIC DATASET S1

In this set of experiments, FSPM2 and the mabers algorithms were run several times using the S1 SYNTHETIC dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure A.5 shows the results of the experiments.

In all the experiments, FSPM2 showed significantly better performance than mabers. FSPM2 was faster by up to 19 times.



	0	0.2	0.4	0.6	0.8
FSPM - 2	490	471	459	459	459
mabers	8600	7803	7398	7398	7398

Figure A.5: Elapsed Time: FSPM2 VS Mabers using S1

S1	FSPM2	mabers	
minsup	Elapsed Time (in msec)	Elapsed Time (in msec)	Gain
0	490	8600	**17.55**
0.1	474	7906	**16.68**
0.2	471	7803	**16.57**
0.3	461	7602	**16.49**
0.4	459	7398	**16.12**
0.5	459	7398	**16.12**
0.6	459	7398	**16.12**
0.7	459	7398	**16.12**
0.8	459	7398	**16.12**
0.9	459	7398	**16.12**

TABLE A.9: ELAPSED TIME: FSPM2 VS MABERS FOR S1

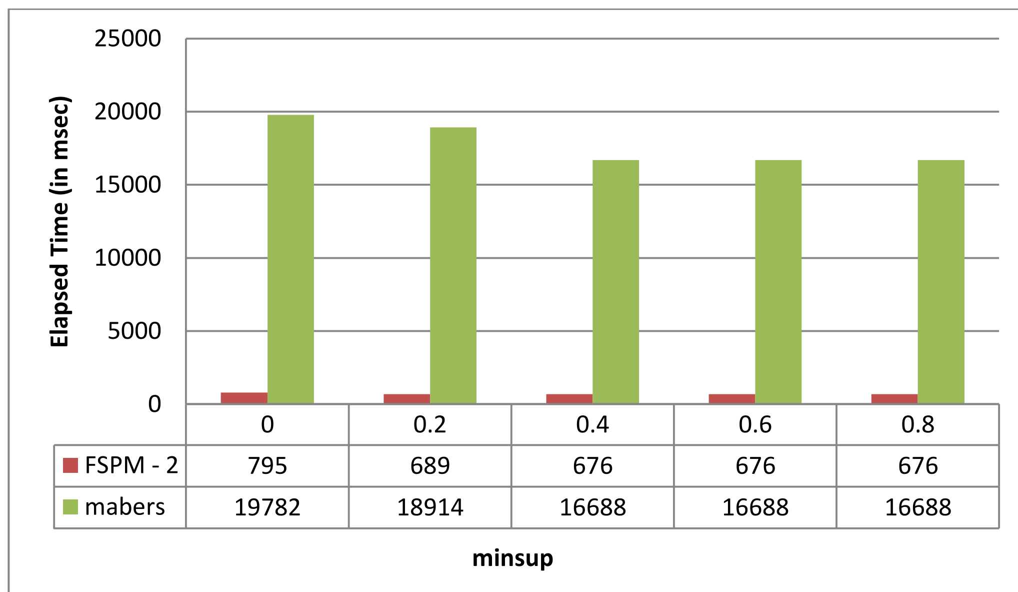
minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6
0	30	65	116	171	112	30	2
0.1	29	7	15	20	15	0	0
0.2	28	7	15	20	0	0	0
0.3	28	7	15	0	0	0	0
0.4	27	7	0	0	0	0	0
0.5	25	7	0	0	0	0	0
0.6	25	6	0	0	0	0	0
0.7	24	1	0	0	0	0	0
0.8	20	1	0	0	0	0	0
0.9	14	1	0	0	0	0	0

TABLE A.10: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR S1

FSPM2 VS. MABERS USING THE SYNTHETIC DATASET S2

In this set of experiments, FSPM2 and the mabers algorithms were run several times using the S2 SYNTHETIC dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure A.6 shows the results of the experiments.

In all the experiments, FSPM2 showed significantly better performance than mabers. FSPM2 was faster by up to 27 times.



	0	0.2	0.4	0.6	0.8
FSPM - 2	795	689	676	676	676
mabers	19782	18914	16688	16688	16688

Figure A.6: Elapsed Time: FSPM2 VS Mabers using S2

S2	FSPM2	mabers	Gain
minsup	Elapsed Time (in msec)	Elapsed Time (in msec)	
0	795	19782	**24.88**
0.1	697	19261	**27.63**
0.2	689	18914	**27.45**
0.3	687	18143	**26.41**
0.4	676	16688	**24.69**
0.5	676	16688	**24.69**
0.6	676	16688	**24.69**
0.7	676	16688	**24.69**
0.8	676	16688	**24.69**
0.9	676	16688	**24.69**

TABLE A.11: ELAPSED TIME: FSPM2 VS MABERS FOR S2

minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6
0	19	37	100	229	254	138	31
0.1	19	7	15	20	15	0	0
0.2	18	7	15	20	0	0	0
0.3	17	7	15	0	0	0	0
0.4	17	7	0	0	0	0	0
0.5	17	7	0	0	0	0	0
0.6	17	6	0	0	0	0	0
0.7	17	1	0	0	0	0	0
0.8	17	1	0	0	0	0	0
0.9	14	1	0	0	0	0	0

TABLE A.12: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR S2

FSPM2 VS. MABERS USING THE SYNTHETIC DATASET S3

In this set of experiments, FSPM2 and the mabers algorithms were run several times using the S3 SYNTHETIC dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure A.7 shows the results of the experiments.

In all the experiments, FSPM2 showed significantly better performance than mabers. FSPM2 was faster by up to 35 times.

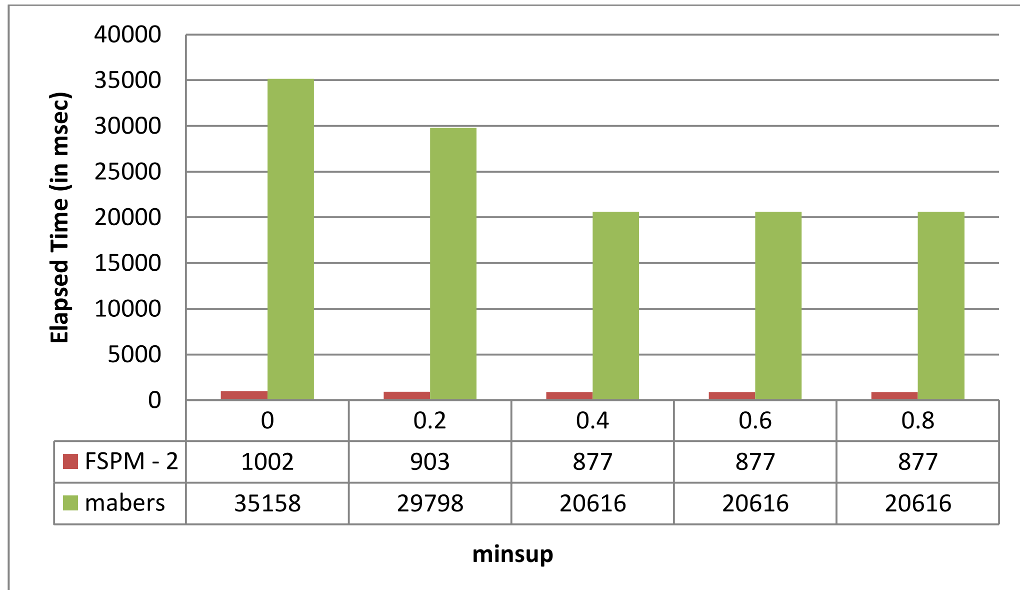


Figure A.7: Elapsed Time: FSPM2 VS Mabers using S3

S3	FSPM2	mabers	
minsup	Elapsed Time (in msec)	Elapsed Time (in msec)	Gain
0	1002	35158	**35.09**
0.1	911	32869	**36.08**
0.2	903	29798	**33.00**
0.3	893	28846	**32.30**
0.4	877	20616	**23.51**
0.5	877	20616	**23.51**
0.6	877	20616	**23.51**
0.7	877	20616	**23.51**
0.8	877	20616	**23.51**
0.9	877	20616	**23.51**

TABLE A.13: ELAPSED TIME: FSPM2 VS MABERS FOR S3

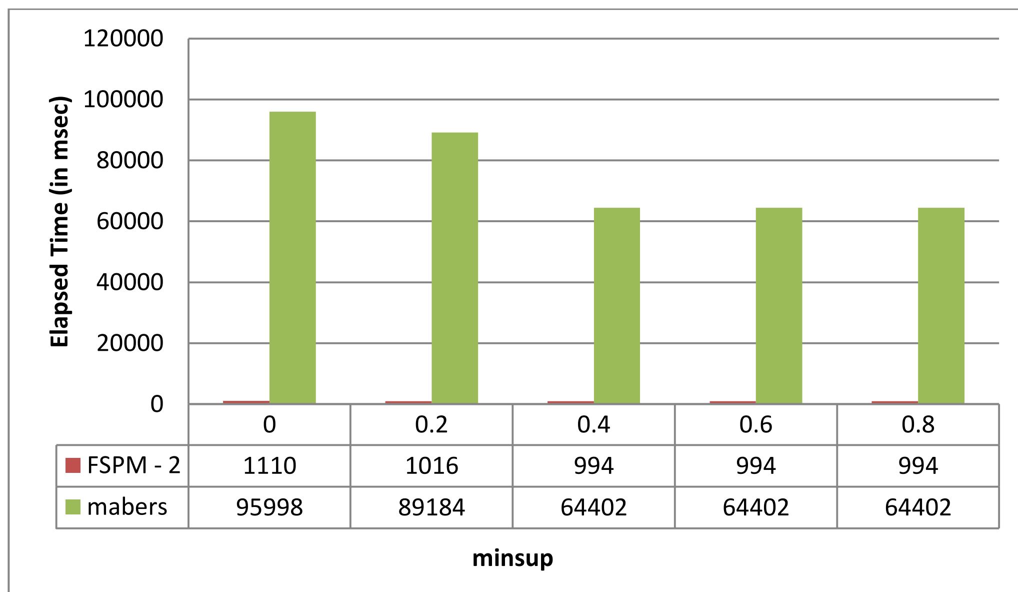
minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6
0	19	37	98	217	222	103	18
0.1	19	7	15	20	15	0	0
0.2	19	7	15	20	0	0	0
0.3	19	7	15	0	0	0	0
0.4	19	7	0	0	0	0	0
0.5	18	7	0	0	0	0	0
0.6	18	3	0	0	0	0	0
0.7	18	1	0	0	0	0	0
0.8	17	1	0	0	0	0	0
0.9	16	1	0	0	0	0	0

TABLE A.14: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR S3

FSPM2 VS. MABERS USING THE SYNTHETIC DATASET S4

In this set of experiments, FSPM2 and the mabers algorithms were run several times using the S4 SYNTHETIC dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure A.8 shows the results of the experiments.

In all the experiments, FSPM2 showed significantly better performance than mabers. FSPM2 was faster by up to 91 times.



	0	0.2	0.4	0.6	0.8
FSPM - 2	1110	1016	994	994	994
mabers	95998	89184	64402	64402	64402

Figure A.8: Elapsed Time: FSPM2 VS Mabers using S4

S4	FSPM2	mabers	Gain
minsup	Elapsed Time (in msec)	Elapsed Time (in msec)	
0	1110	95998	**86.48**
0.1	1022	93032	**91.03**
0.2	1016	89184	**87.78**
0.3	1013	88755	**87.62**
0.4	994	64402	**64.79**
0.5	994	64402	**64.79**
0.6	994	64402	**64.79**
0.7	994	64402	**64.79**
0.8	994	64402	**64.79**
0.9	994	64402	**64.79**

TABLE A.15: ELAPSED TIME: FSPM2 VS MABERS FOR S4

minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6
0	19	37	98	214	201	79	10
0.1	19	7	15	20	15	0	0
0.2	18	7	15	20	0	0	0
0.3	17	7	15	0	0	0	0
0.4	17	7	0	0	0	0	0
0.5	17	7	0	0	0	0	0
0.6	17	3	0	0	0	0	0
0.7	17	1	0	0	0	0	0
0.8	16	1	0	0	0	0	0
0.9	16	1	0	0	0	0	0

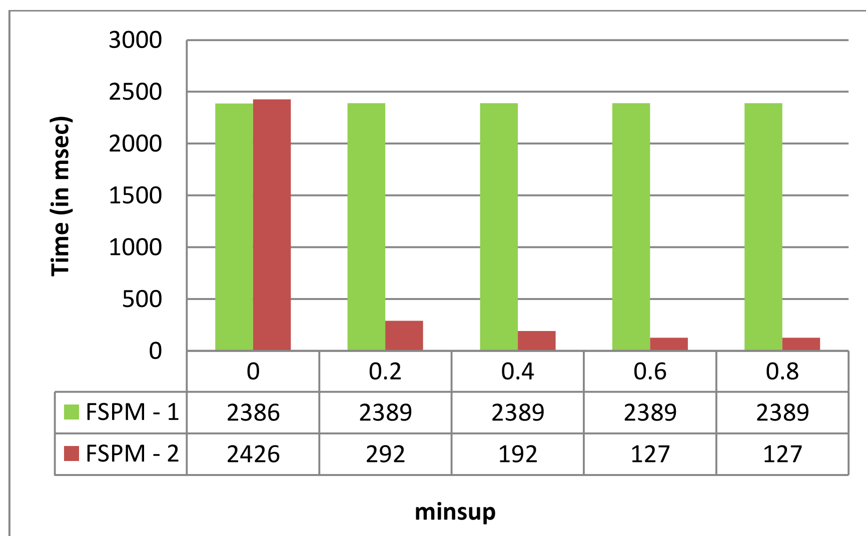
TABLE A.16: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR S4

APPENDIX C

FSPM1 VS. FSPM2 USING THE DBLP DATASET D1

In this set of experiments, FSPM1 and FSPM2 were run several times using the D1 DBLP dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure A.9 shows the results of these experiments.

In all the experiments, FSPM2 showed better performance than FSPM1 except when minsup was 0.0. FSPM2 was faster by up to 18 times.



	0	0.2	0.4	0.6	0.8
FSPM - 1	2386	2389	2389	2389	2389
FSPM - 2	2426	292	192	127	127

Figure A.9: FSPM1 VS FSPM2 Mining Time for D1

D1	FSPM1	FSPM2	Gain
minsup	Mining Time (in msec)	Mining Time (in msec)	
0	2386	2426	**0.98**
0.1	2389	381	**6.27**
0.2	2389	292	**8.18**
0.3	2389	292	**8.18**
0.4	2389	192	**12.44**
0.5	2389	129	**18.52**
0.6	2389	127	**18.81**
0.7	2389	127	**18.81**
0.8	2389	127	**18.81**
0.9	2389	127	**18.81**

TABLE A.17: FSPM1 VS FSPM2 MINING TIME FOR D1

minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6	TE_7	TE_8	TE_9	TE_{10}
0	14	23	108	359	705	901	773	440	159	33	3
0.1	14	10	28	56	70	56	28	8	1	0	0
0.2	14	10	27	50	55	36	13	2	0	0	0
0.3	14	10	27	50	55	36	13	2	0	0	0
0.4	14	10	21	35	35	21	7	1	0	0	0
0.5	14	8	15	20	15	6	1	0	0	0	0
0.6	14	8	15	20	15	6	1	0	0	0	0
0.7	8	8	15	20	15	6	1	0	0	0	0
0.8	8	8	15	20	15	6	1	0	0	0	0
0.9	8	8	15	20	15	6	1	0	0	0	0

TABLE A.18: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR D1

FSPM1 VS. FSPM2 USING THE DBLP DATASET D2

In this set of experiments, FSPM1 and FSPM2 were run several times using the D2 DBLP dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure A.10 shows the results of these experiments.

In all the experiments, FSPM2 showed better performance than FSPM1 except when minsup was 0.0. FSPM2 was faster by up to 18 times.

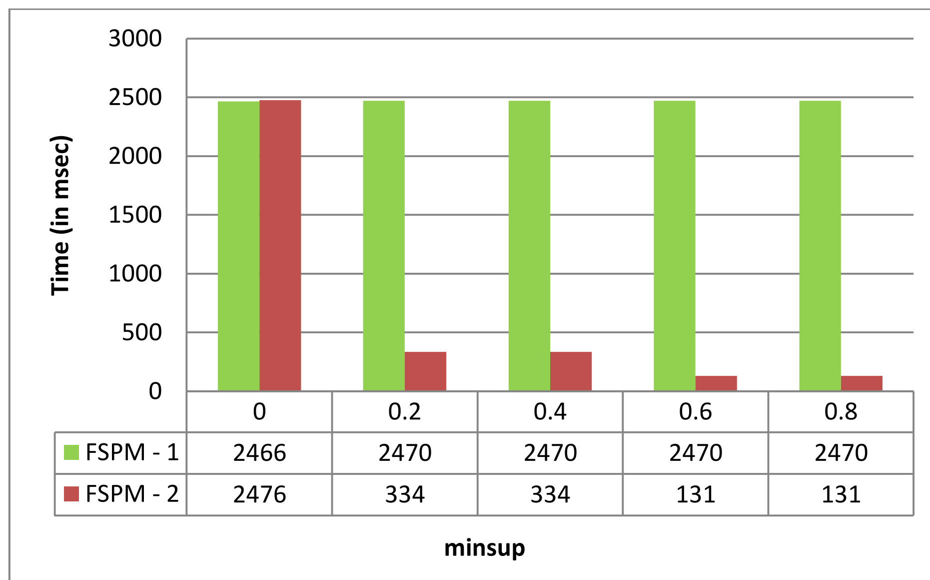


Figure A.10: FSPM1 VS FSPM2 Mining Time for D2

D2	FSPM1	FSPM2	Gain
minsup	Mining Time (in msec)	Mining Time (in msec)	
0	2466	2476	**0.996**
0.1	2470	368	**6.71**
0.2	2470	334	**7.40**
0.3	2470	334	**7.40**
0.4	2470	334	**7.40**
0.5	2470	131	**18.85**
0.6	2470	131	**18.85**
0.7	2470	131	**18.85**
0.8	2470	131	**18.85**
0.9	2470	131	**18.85**

TABLE A.19: FSPM1 VS FSPM2 MINING TIME FOR D2

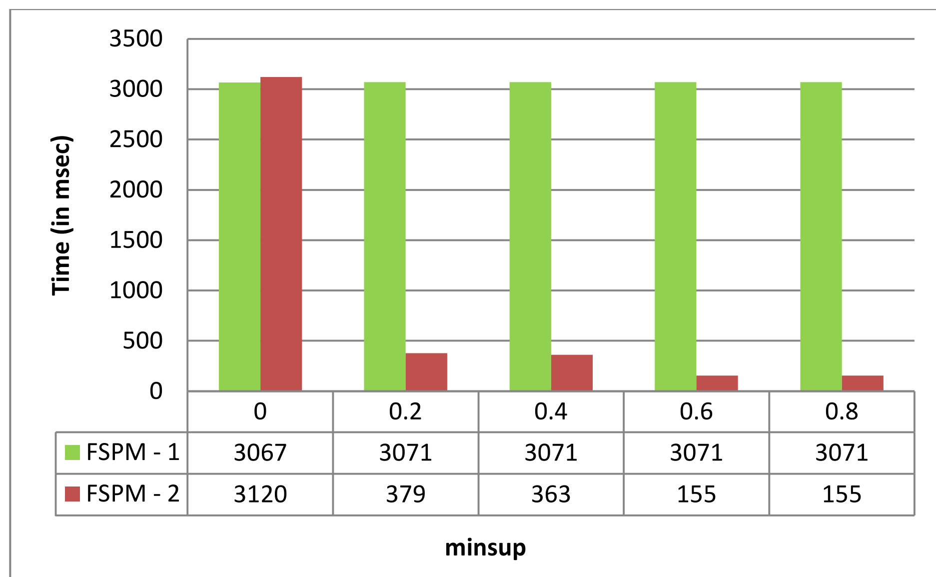
minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6	TE_7	TE_8	TE_9	TE_{10}
0	14	23	113	390	776	991	838	465	163	33	3
0.1	14	10	28	56	70	56	28	8	1	0	0
0.2	14	10	27	50	55	36	13	2	0	0	0
0.3	14	10	27	50	55	36	13	2	0	0	0
0.4	14	10	27	50	55	36	13	2	0	0	0
0.5	14	8	15	20	15	6	1	0	0	0	0
0.6	14	8	15	20	15	6	1	0	0	0	0
0.7	13	8	15	20	15	6	1	0	0	0	0
0.8	8	8	15	20	15	6	1	0	0	0	0
0.9	8	8	15	20	15	6	1	0	0	0	0

TABLE A.20: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVEL FOR D2

FSPM1 VS. FSPM2 USING THE DBLP DATASET D3

In this set of experiments, FSPM1 and FSPM2 were run several times using the D3 DBLP dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure A.11 shows the results of these experiments.

In all the experiments, FSPM2 showed better performance than FSPM1 except when minsup was 0.0. FSPM2 was faster by up to 19 times.



	0	0.2	0.4	0.6	0.8
FSPM - 1	3067	3071	3071	3071	3071
FSPM - 2	3120	379	363	155	155

Figure A.11: FSPM1 VS FSPM2 Mining Time for D3

D3	FSPM1	FSPM2	Gain
minsup	Mining Time (in msec)	Mining Time (in msec)	
0	3067	3120	**0.98**
0.1	3071	487	**6.31**
0.2	3071	379	**8.10**
0.3	3071	379	**8.10**
0.4	3071	363	**8.46**
0.5	3071	240	**12.80**
0.6	3071	155	**19.81**
0.7	3071	155	**19.81**
0.8	3071	155	**19.81**
0.9	3071	155	**19.81**

TABLE A.21: FSPM1 VS FSPM2 MINING TIME FOR D3

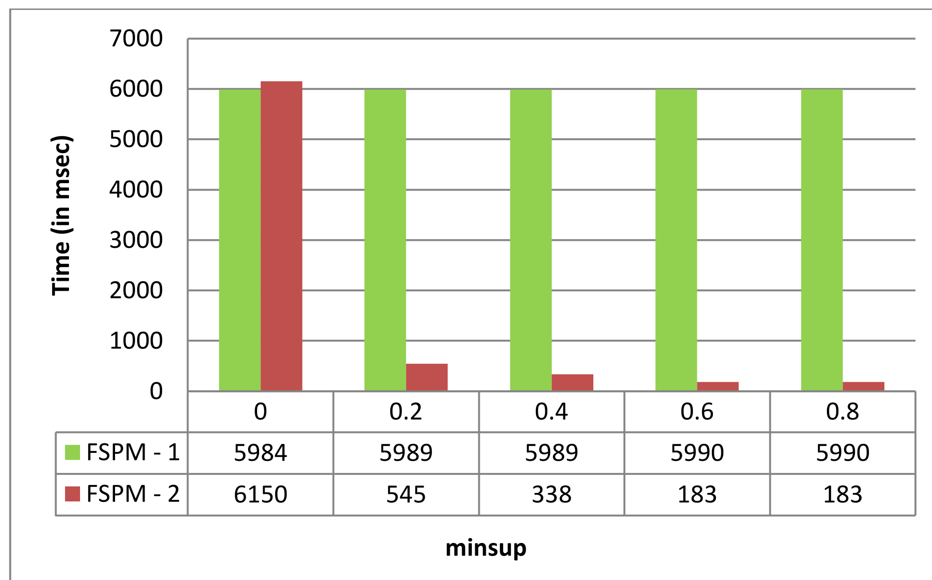
minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6	TE_7	TE_8	TE_9	TE_{10}
0	14	26	126	438	874	1118	944	520	179	35	3
0.1	14	10	28	56	70	56	28	8	1	0	0
0.2	14	10	27	50	55	36	13	2	0	0	0
0.3	14	9	27	50	55	36	13	2	0	0	0
0.4	14	9	25	47	54	36	13	2	0	0	0
0.5	14	8	21	35	35	21	7	1	0	0	0
0.6	14	8	15	20	15	6	1	0	0	0	0
0.7	8	8	15	20	15	6	1	0	0	0	0
0.8	8	8	15	20	15	6	1	0	0	0	0
0.9	8	8	15	20	15	6	1	0	0	0	0

TABLE A.22: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVEL FOR D3

FSPM1 VS. FSPM2 USING THE DBLP DATASET D4

In this set of experiments, FSPM1 and FSPM2 were run several times using the D4 DBLP dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure A.12 shows the results of these experiments.

In all the experiments, FSPM2 showed better performance than FSPM1 except when minsup was 0.0. FSPM2 was faster by up to 32 times.



	0	0.2	0.4	0.6	0.8
FSPM - 1	5984	5989	5989	5990	5990
FSPM - 2	6150	545	338	183	183

Figure A.12: FSPM1 VS FSPM2 Mining Time for D4

D4	FSPM1	FSPM2	Gain
minsup	Mining Time (in msec)	Mining Time (in msec)	
0	5984	6150	**0.97**
0.1	5989	683	**8.77**
0.2	5989	545	**10.99**
0.3	5989	545	**10.99**
0.4	5989	338	**17.72**
0.5	5989	338	**17.72**
0.6	5990	183	**32.73**
0.7	5990	183	**32.73**
0.8	5990	183	**32.73**
0.9	5990	183	**32.73**

TABLE A.23: FSPM1 VS FSPM2 MINING TIME FOR D4

minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6	TE_7	TE_8	TE_9	TE_{10}	TE_{11}
0	16	31	156	567	1203	1654	1539	981	426	123	22	2
0.1	16	10	28	56	70	56	28	8	1	0	0	0
0.2	15	10	27	50	55	36	13	2	0	0	0	0
0.3	15	10	27	50	55	36	13	2	0	0	0	0
0.4	15	9	21	35	35	21	7	1	0	0	0	0
0.5	15	9	21	35	35	21	7	1	0	0	0	0
0.6	14	7	15	20	15	6	1	0	0	0	0	0
0.7	8	7	15	20	15	6	1	0	0	0	0	0
0.8	8	7	15	20	15	6	1	0	0	0	0	0
0.9	8	7	15	20	15	6	1	0	0	0	0	0

TABLE A.24: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVELS FOR D4

APPENDIX D

FSPM1 VS. FSPM2 USING THE SYNTHETIC DATASET S1

In this set of experiments, FSPM1 and FSPM2 were run several times using the S1 SYNTHETIC dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure A.13 shows the results of these experiments.

In all the experiments, FSPM2 showed better performance than FSPM1 except when minsup was 0.0. FSPM2 was faster by up to 1.31 times.

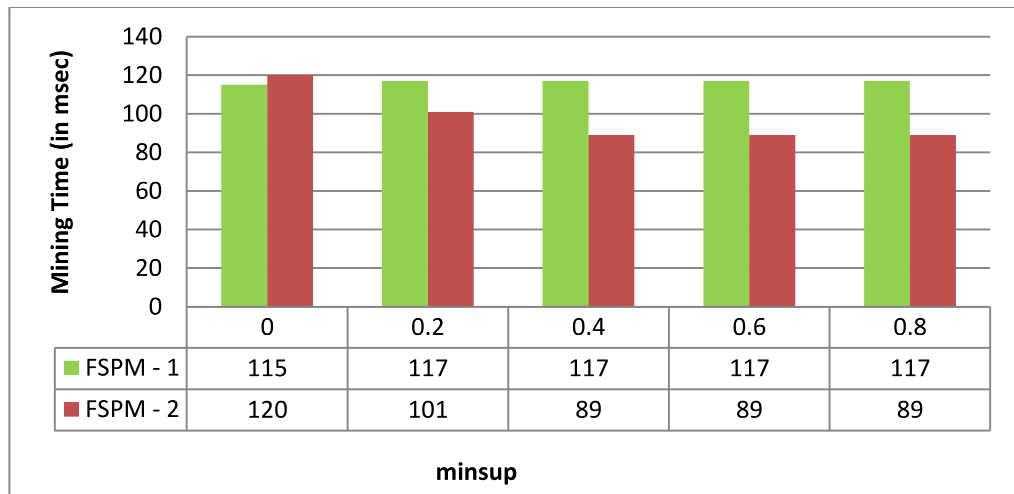


Figure A.13: FSPM1 VS FSPM2 Mining Time for S1

S1	FSPM1	FSPM2	Gain
minsup	Mining Time (in msec)	Mining Time (in msec)	
0	115	120	**0.96**
0.1	117	104	**1.13**
0.2	117	101	**1.16**
0.3	117	91	**1.29**
0.4	117	89	**1.31**
0.5	117	89	**1.31**
0.6	117	89	**1.31**
0.7	117	89	**1.31**
0.8	117	89	**1.31**
0.9	117	89	**1.31**

TABLE A.25: FSPM1 VS FSPM2 MINING TIME FOR S1

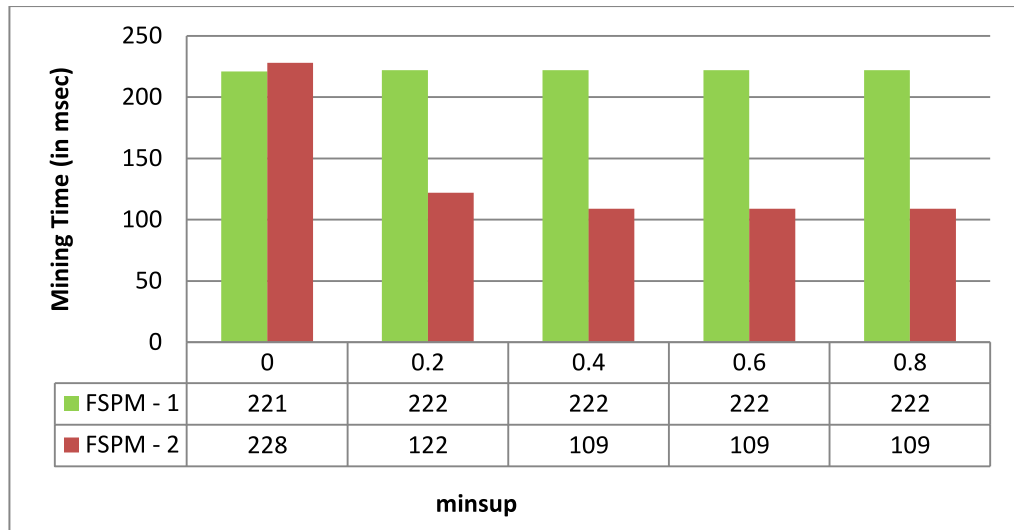
minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6
0	30	65	116	171	112	30	2
0.1	29	7	15	20	15	0	0
0.2	28	7	15	20	0	0	0
0.3	28	7	15	0	0	0	0
0.4	27	7	0	0	0	0	0
0.5	25	7	0	0	0	0	0
0.6	25	6	0	0	0	0	0
0.7	24	1	0	0	0	0	0
0.8	20	1	0	0	0	0	0
0.9	14	1	0	0	0	0	0

TABLE A.26: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVEL FOR S1

FSPM1 VS. FSPM2 USING THE SYNTHETIC DATASET S2

In this set of experiments, FSPM1 and FSPM2 were run several times using the S2 SYNTHETIC dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure A.14 shows the results of these experiments.

In all the experiments, FSPM2 showed better performance than FSPM1 except when minsup was 0.0. FSPM2 was faster by up to 2 times.



minsup	0	0.2	0.4	0.6	0.8
FSPM - 1	221	222	222	222	222
FSPM - 2	228	122	109	109	109

Figure A.14: FSPM1 VS FSPM2 Mining Time for S2

S2	FSPM1	FSPM2	Gain
minsup	Mining Time (in msec)	Mining Time (in msec)	
0	221	228	**0.97**
0.1	222	130	**1.71**
0.2	222	122	**1.82**
0.3	222	120	**1.85**
0.4	222	109	**2.04**
0.5	222	109	**2.04**
0.6	222	109	**2.04**
0.7	222	109	**2.04**
0.8	222	109	**2.04**
0.9	222	109	**2.04**

TABLE A.27: FSPM1 VS FSPM2 MINING TIME FOR S2

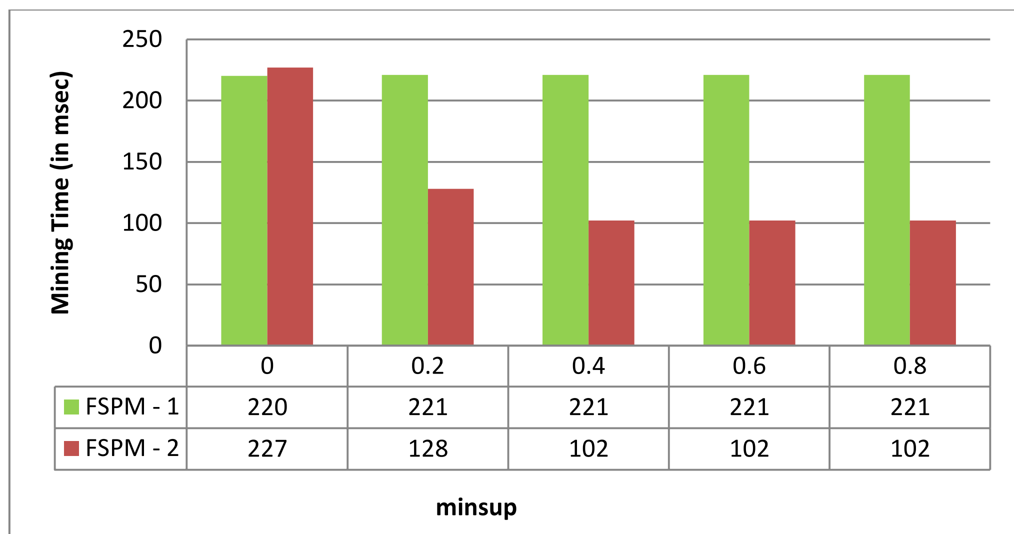
minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6
0	19	37	100	229	254	138	31
0.1	19	7	15	20	15	0	0
0.2	18	7	15	20	0	0	0
0.3	17	7	15	0	0	0	0
0.4	17	7	0	0	0	0	0
0.5	17	7	0	0	0	0	0
0.6	17	6	0	0	0	0	0
0.7	17	1	0	0	0	0	0
0.8	17	1	0	0	0	0	0
0.9	14	1	0	0	0	0	0

TABLE A.28: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVEL FOR S2

FSPM1 VS. FSPM2 USING THE SYNTHETIC DATASET S3

In this set of experiments, FSPM1 and FSPM2 were run several times using the S3 SYNTHETIC dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure A.15 shows the results of these experiments.

In all the experiments, FSPM2 showed better performance than FSPM1 except when minsup was 0.0. FSPM2 was faster by up to 2 times.



	0	0.2	0.4	0.6	0.8
FSPM - 1	220	221	221	221	221
FSPM - 2	227	128	102	102	102

Figure A.15: FSPM1 VS FSPM2 Mining Time for S3

S3	FSPM1	FSPM2	
minsup	**Mining Time (in msec)**	**Mining Time (in msec)**	**Gain**
0	220	227	**0.97**
0.1	221	136	**1.63**
0.2	221	128	**1.73**
0.3	221	118	**1.87**
0.4	221	102	**2.17**
0.5	221	102	**2.17**
0.6	221	102	**2.17**
0.7	221	102	**2.17**
0.8	221	102	**2.17**
0.9	221	102	**2.17**

TABLE A.29: FSPM1 VS FSPM2 MINING TIME FOR S3

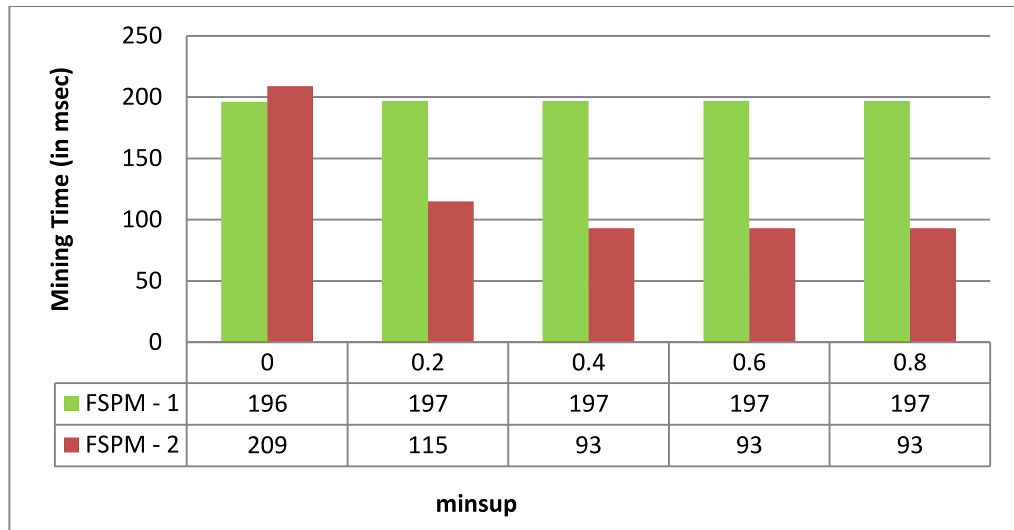
minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6
0	19	37	98	217	222	103	18
0.1	19	7	15	20	15	0	0
0.2	19	7	15	20	0	0	0
0.3	19	7	15	0	0	0	0
0.4	19	7	0	0	0	0	0
0.5	18	7	0	0	0	0	0
0.6	18	3	0	0	0	0	0
0.7	18	1	0	0	0	0	0
0.8	17	1	0	0	0	0	0
0.9	16	1	0	0	0	0	0

TABLE A.30: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVEL FOR S3

FSPM1 VS. FSPM2 USING THE SYNTHETIC DATASET S4

In this set of experiments, FSPM1 and FSPM2 were run several times using the S4 SYNTHETIC dataset segment. The experiments were done using a number of minsups varying from 0.0 to 0.9. Figure A.16 shows the results of these experiments.

In all the experiments, FSPM2 showed better performance than FSPM1 except when minsup was 0.0. FSPM2 was faster by up to 2 times.



	0	0.2	0.4	0.6	0.8
FSPM - 1	196	197	197	197	197
FSPM - 2	209	115	93	93	93

Figure A.16: FSPM1 VS FSPM2 Mining Time for S4

S4	FSPM1	FSPM2	Gain
minsup	Mining Time (in msec)	Mining Time (in msec)	
0	196	209	**0.94**
0.1	197	121	**1.63**
0.2	197	115	**1.71**
0.3	197	112	**1.76**
0.4	197	93	**2.12**
0.5	197	93	**2.12**
0.6	197	93	**2.12**
0.7	197	93	**2.12**
0.8	197	93	**2.12**
0.9	197	93	**2.12**

TABLE A.31: FSPM1 VS FSPM2 MINING TIME FOR S4

minsup	TE_0	TE_1	TE_2	TE_3	TE_4	TE_5	TE_6
0	19	37	98	214	201	79	10
0.1	19	7	15	20	15	0	0
0.2	18	7	15	20	0	0	0
0.3	17	7	15	0	0	0	0
0.4	17	7	0	0	0	0	0
0.5	17	7	0	0	0	0	0
0.6	17	3	0	0	0	0	0
0.7	17	1	0	0	0	0	0
0.8	16	1	0	0	0	0	0
0.9	16	1	0	0	0	0	0

TABLE A.32: COUNT OF TREE-EXPRESSIONS AT DIFFERENT LEVEL FOR S4

REFERENCES

[1] D. C. M. F. D. F. J. R. J. S. M. S. S. BOAG, "Xquery 1.0: An xml query language," April 2002.

[2] G. Gou and R. Chirkova, "Efficiently querying large XML data repositories: A survey," *IEEE Transactions on Knowledge and Data Engineering,* vol. 19, pp. 1381-1403, 2007.

[3] J. Paik*, et al.*, "Fast extraction of maximal frequent subtrees using bits representation," *Journal of Information Science and Engineering,* vol. 25, pp. 435-464, 2009.

[4] R. Goldman and J. Widom, "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases," presented at the Proceedings of the 23rd International Conference on Very Large Data Bases, 1997.

[5] T. Milo and D. Suciu, "Index Structures for Path Expressions," presented at the Proceedings of the 7th International Conference on Database Theory, 1999.

[6] K. Wang and H. Liu, "Discovering structural association of semistructured data," *IEEE Transactions on Knowledge and Data Engineering,* vol. 12, pp. 353-371, 2000.

[7] M. M. Khaing and N. Thein, "An efficient association rule mining for XML data," 2006, pp. 5782-5786.

[8] M. El-Hajj and O. R. Zaïane, "Non recursive generation of frequent K-itemsets from frequent pattern tree representations," vol. 2737, ed, 2003, pp. 371-380.

[9] J. Liu*, et al.*, "Mining frequent item sets by opportunistic projection," 2002, pp. 229-238.

[10] J. P. E.M. (eds) T. Bray, C.M. Sperberg-McQueen, "Extensible Markup Language (XML)," *http://www.w3.org/TR/REC-xml*.

[11] R. Agrawal*, et al.*, "Mining association rules between sets of items in large databases," presented at the Proceedings of the 1993 ACM SIGMOD international conference on Management of data, Washington, D.C., United States, 1993.

[12] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases," presented at the Proceedings of the 20th International Conference on Very Large Data Bases, 1994.

[13] J. Han*, et al.*, "Mining frequent patterns without candidate generation," *SIGMOD Record (ACM Special Interest Group on Management of Data),* vol. 29, pp. 1-12, 2000.

[14] C. N. Win and K. H. S. Hla, "Mining frequent patterns from XML data," 2005, pp. 208-212.

[15] M. Mazuran*, et al.*, "Mining tree-based frequent patterns from XML," vol. 5822 LNAI, ed, 2009, pp. 287-299.

[16] D. Braga*, et al.*, "A tool for extracting XML association rules," in *Tools with Artificial Intelligence, 2002. (ICTAI 2002). Proceedings. 14th IEEE International Conference on*, 2002, pp. 57-64.

[17] G. A. Potamias and V. S. Moustakis, "Knowledge discovery from distributed clinical data sources: the era for internet-based epidemiology," in *Engineering in Medicine and Biology Society, 2001. Proceedings of the 23rd Annual International Conference of the IEEE*, 2001, pp. 3638-3641 vol.4.

[18] X. Wang and C. Cao, "Mining association rules from complex and irregular XML documents using XSLT and XQuery," 2008, pp. 314-319.

[19] X. Y. Li*, et al.*, "Mining association rules from XML data with index table," 2007, pp. 3905-3910.

[20] R. Porkodi*, et al.*, "An improved association rule mining technique for xml data using Xquery and apriori algorithm," 2009, pp. 1510-1514.

[21] S. Zhang*, et al.*, "XAR-Miner: Efficient association rules mining for XML data," 2005, pp. 894-895.

[22] R. AliMohammadzadeh*, et al.*, "Template guided association rule mining from XML documents," 2006, pp. 963-964.

[23] "XSL Transformations (XSLT) Version 1.0 W3C Recommendation 16 November 1999," *http://www.w3.org/TR/1999/REC-xslt-19991116*.

[24] Q. D. a. G. Sundarraj, "Association rule mining from XML data " *Computer Science Program,* 2005.

[25] L. Feng and T. Dillon, "Mining interesting XML-enabled association rules with templates," 2005, pp. 66-88.

[26] L. Chen*, et al.*, "Mining association rules from structural deltas of historical XML documents," 2004, pp. 452-457.

[27] A. Termier, Rousset, M.-C., Sebag, M., "Mining XML data with frequent trees," *DBFusion Workshop,* pp. 87-96, 2002.

[28] T. Asai, Abe, K., Kawasoe, S., Arimura, H., Sakamoto, H., Arikawa, S, "Efficient substructure discovery from large semi-structured data " *Proc 2nd SIAM International Conference on Data Mining (SDM'02),* pp. 158-174.

[29] M. J. Zaki, "Efficiently mining frequent trees in a forest " *Proc 8th International Conference on Knowledge Discovery and Data Mining (ACM SIGKDD'02),* pp. 71-80.

[30] S. K. K. Abe, T. Asai, H. Arimura, S. Arikawa, "Optimized substructure discovery for semi-structured data," *in: Proceedings of the Sixth European Conference on Principles of Data Mining and Knowledge Discovery (PKDD), Lecture Notes in Artificial Intelligence,* vol. 2431, pp. 1–14, 2002.

[31] G. Cong*, et al.*, "Discovering Frequent Substructures from Hierarchical Semi-structured Data," *in Proc. SDM,* 2002.

[32] S. Hido and H. Kawano, "A fast algorithm for mining frequent ordered subtrees," *Systems and Computers in Japan,* vol. 38, pp. 34-43, 2007.

[33] D. Katsaros*, et al.*, "Fast mining of frequent tree structures by hashing and indexing," *Information and Software Technology,* vol. 47, pp. 129-140, 2005.

[34] J. Paik*, et al.*, "EFoX: A scalable method for extracting frequent subtrees," 2005, pp. 813-817.

[35] A. Savasere*, et al.*, "An Efficient Algorithm for Mining Association Rules in Large Databases," presented at the Proceedings of the 21th International Conference on Very Large Data Bases, 1995.

[36] R. Agrawal and R. Srikant, "Mining sequential patterns," 1995, pp. 3-14.

[37] H. Y. Youn, Paik, J., Kim, U.M., "A new method for mining association rules from a collection of XML documents " *Computational Science and Its Applications,* 2005.
[38] E. Quintarelli, Baralis, E., Garza, P., Tanca, L., "Answering XML queries by means of data summaries," *ACM Trans. of Information Systems,* 2007.
[39] J. Paik*, et al.*, "A framework for data structure-guided extraction of XML association rules," vol. 4489 LNCS, ed, 2007, pp. 709-716.
[40] S. Jun*, et al.*, "Mining association rules from a collection of XML documents using cross filtering algorithm," 2006, pp. 120-126.
[41] "UW XML repository," *http://www.cs.washington.edu/research/xmldatasets/www/repository.html*.
[42] "The DBLP Computer Science Bibliography," *http://dblp.uni-trier.de/xml/*.

VITAE

Mohammed Mohsin Ali was born in India. He earned his Bachelor of Engineering degree in Information Technology in June 2008 from Osmania University, India. Mohsin completed his Master of Science degree in Computer Science in May 2012 from KFUPM. Before joining KFUPM for the Master's program, he worked in ACCENTURE as a Business Intelligence Quality Analyst. He worked with two clients of ACCENTURE namely Safeco Insurance and SunTrust Bank.

Contact Details:

Present Address: KFUPM, P.O. Box 8650, Dhahran 31261, Saudi Arabia.

E-mail Address: g200905170@kfupm.edu.sa

Permanent Address: Hyderabad, India.

E-mail Address: mohsinali.mohammed@gmail.com

Phone: 050 441 9167