Discovering association rules from structural evolution of tree structured web data

Chen, Ling

2007

Chen, L. (2007). Discovering association rules from structural evolution of tree structured web data. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/35726>

<https://doi.org/10.32657/10356/35726>

a53263

DISCOVERING ASSOCIATION RULES FROM STRUCTURAL EVOLUTION OF TREE STRUCTURED WEB DATA

Chen Ling

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in fulfilment of the requirement for the degree of
Doctor of Philosophy

2007

Abstract

Today, the Web has been one of the most mammoth information storage. Its rapid evolution not only makes it more and more important for users to access information, but also poses challenging problems, such as maintaining search engines and updating web caching etc, in the Web community. In our research, we exploit the evolution of the Web further to discover novel knowledge from the Web history. We mine association rules from changes to different types of tree structured Web data and study how the extracted knowledge can be applied in corresponding applications.

We consider two types of data source as the mining context in this dissertation. One is a sequence of historical versions of a tree structure, which models some tree-structured Web data. The other is a set of historical tree sequences, where each tree models some semantic Web data. Given a sequence of historical versions of a tree structure, we are interested in the evolution association between subtrees. Particularly, we define two types of patterns as follows to capture intra-tree evolution associations.

- *FRACTURE* (**FR**erequently **And** **C**oncurrently **muT**ating substruct**URE**s), which is a set of subtrees that not only frequently change together but also frequently undergo significant changes together.
- *negative FRACTURE*, which consists of subtrees that are negatively correlated in undergoing significant changes.

Given a set of historical tree sequences, we are interested in the evolution association between trees. We propose the pattern *SEASON* (**SEm**Antic **S**equence of structural **ev**olutio**N**) to extract the inter-tree evolution association. Furthermore, while *FRACTURE* and *negative FRACTURE* consider changes to tree structures from the structural variation perspective, *SEASON* view changes from the semantic variation perspective. Hence, a

SEASON pattern is actually a frequent sequence which indicates how the semantics evolve along with the evolution of tree structures.

We designed evolution metrics to evaluate the interestingness of the proposed patterns and developed data mining algorithms to mine the proposed patterns efficiently. To demonstrate the usefulness of the patterns, we further applied them in corresponding applications. Particularly, we used *FRACTURE* patterns in a Web user clustering method which clusters Web users based on the evolution of their historical Web session trees. Our experimental results showed that given a set of Web users who visited similar pages at different time points, our approach discovered a set of clusters of high quality while existing Web user clustering methods failed to distinguished such users clearly. We developed Web site mirror maintenance strategies based on *negative FRACTUREs* discovered in historical Web site directory hierarchies. The experimental results demonstrated that the efficiency of mirror maintenance is improved without sacrificing too much in keeping the “freshness” of mirrors. Finally, we mine *SEASONs* from the historical XML query trees and utilize the discovered knowledge to design optimal cache replacement strategies. Experimental results verified that the XML queries can be answered more efficiently.

Acknowledgments

First and foremost I would like to acknowledge my supervisor, Associate Professor Sourav Saha Bhowmick, for his patient, tireless and prompt help. Dr. Sourav always provides me complete freedom to explore and work on the research topics that I have interests in. Although it was difficult for me to make quick progress at the beginning, I must appreciate the wisdom of his supervision when I started to think for myself and become relatively independent. At the same time, he is always available when I need his advises. It is my luck to study and work under his guidance. I also thank my co-supervisor, Associate Professor Chia Liang Tien, for many valuable discussions. Special acknowledgements go to Assistant Professor Li jinyan, my mentor during the eight-month attachment to the Institute for Infocomm Research (I²R), for his generous assistance and valuable suggestions.

During the past four years, my colleagues in the Centre for Multimedia and Networking Technologies (CeMNet) and the Centre for Advanced Information Systems (CAIS) have provided me much appreciated help in my daily work. I would like to thank all of them for their encouragements on my study and their valuable friendship.

I would like to thank the School of Computer Engineering of Nanyang Technological University, Singapore for the financial support and excellent working environment provided during my past years of study.

I cannot finish my thesis work without the strong support from my family. In the middle of my study, I gave birth to my first baby. In order to let me concentrate on my study, my dearest mother quitted her beloved job, which she has undertaken for 25 years, without any hesitation and took care of the baby for me day and night. I am deeply indebted to her for her unconditional support. Besides my mother, I have a great father as well. He teaches me all the good things that really matter in life. Special thanks must go to my lovely son – ZhiTan – who is the source of my happiness and the reason of my life. Together with him is my hubby, ATing, who is always there to provide me his support through both the highs and lows of my time.

Contents

Abstract	i
Acknowledgments	iii
List of Figures	viii
List of Tables	xi
1 Introduction	1
1.1 Key Features of Web Data	2
1.2 Limitation of State-of-the-art Web Mining	7
1.2.1 Existing Web Mining - A Bird's-Eye View	7
1.2.2 Limitations	8
1.3 Novel Research Issues	9
1.4 Research Overview	11
1.4.1 Architecture	11
1.4.2 Applications	13
1.4.3 Scope of Research	14
1.5 Contributions	15
1.6 Organization	16
2 Literature Survey	19
2.1 Association Rule Mining	19
2.1.1 Basic Definitions	19
2.1.2 Frequent Pattern	22
2.1.3 Infrequent Pattern	30
2.1.4 Concise Pattern	33
2.2 Web Mining	36
2.2.1 Web Content Mining	36

2.2.2	Web Structure Mining	40
2.2.3	Web Usage Mining	40
2.3	Web Change Detection	42
2.3.1	Detecting Changes to HTML Documents	43
2.3.2	Detecting Changes to XML Documents	44
2.4	Summary	47
3	Problem Definition	50
3.1	Problem Model	50
3.2	Examples of Associative Evolutionary Structures	54
3.2.1	FRACTURE	54
3.2.2	Negative FRACTURE	56
3.2.3	SEASON	58
3.3	Research Challenges	60
3.4	Related Work	62
3.5	Summary	63
4	FRACTURE Mining	65
4.1	Preliminaries	66
4.1.1	Background on Trees	66
4.1.2	Structural Delta Database	67
4.1.3	Evolution Metrics	71
4.1.4	Pattern Definition	73
4.2	FRACTURE Pattern Mining	73
4.2.1	Procedure Overview	73
4.2.2	Algorithm I: Apriori-FRACTURE	74
4.2.3	Algorithm II: FPG-FRACTURE	78
4.3	Maximal FRACTURE Pattern Mining	82
4.3.1	Maximal Pattern Definition	82
4.3.2	Maximal FRACTURE Mining	84
4.4	Experimental Results	90
4.4.1	Experiments on Synthetic Datasets	90
4.4.2	Experiments on Real-life Datasets	96

4.5	Applications of FRACTURES	99
4.6	Summary	102
5	COWES: FRACTURE-based Clustering	103
5.1	Introduction of COWES	104
5.1.1	Motivating Example	105
5.1.2	Applications	107
5.1.3	Overview of COWES	108
5.2	Clustering Feature	109
5.3	Similarity Measures	111
5.3.1	Types of Shared FRACTURES	112
5.3.2	FRACTURE Similarity	114
5.3.3	User Similarity	116
5.4	Framework of COWES	117
5.4.1	Similarity Computation	117
5.4.2	Cluster Generation	118
5.5	Experimental Results	120
5.5.1	Experiments on Synthetic Data	120
5.5.2	Experiments on Real Data	126
5.6	Related Work	131
5.7	Summary	132
6	Negative FRACTURE Mining and Application	134
6.1	Preliminaries	135
6.1.1	Evolution Metrics	135
6.1.2	Pattern Definition	138
6.2	Negative FRACTURE Mining	139
6.2.1	Procedure Overview	139
6.2.2	GDT Construction	140
6.2.3	Pattern Discovery	142
6.3	Applications of Negative FRACTURE	146
6.3.1	Potential Applications	146
6.3.2	Maintaining Autonomous Web Mirrors	148

6.4	Experimental Results	153
6.4.1	Performance of Negative FRACTURE Mining algorithm	154
6.4.2	Performance of Mirror Maintaining Strategy	156
6.5	Summary	161
7	SEASON Mining and Application	163
7.1	SEASON Definition	164
7.2	SEASON Mining	167
7.2.1	Procedure Overview	167
7.2.2	Semantic Tree Clustering	168
7.2.3	Frequent Cluster Sequence Discovery	175
7.3	Applications of SEASON	177
7.3.1	Potential Applications	177
7.3.2	Replacing XML Query Cache	178
7.4	Experimental Results	183
7.4.1	Performance of SEASON Mining	183
7.4.2	Performance of Replacement Strategy	187
7.5	Related Work on Caching	189
7.6	Summary	191
8	Conclusions	192
8.1	Summary of Results	192
8.2	Future Research Issues	195
A		198
B		200
	References	201

List of Figures

1.1	Types of Web data.	4
1.2	Examples of Inter-document data.	5
1.3	Examples of Intra-structure data.	6
1.4	Examples of access log data.	6
1.5	Examples of query log data.	7
1.6	Architecture of associative evolutionary structures mining.	12
1.7	Dissertation structure.	18
2.1	An example of FP-Tree.	26
2.2	OEM graph.	37
2.3	Comparison of existing association rule mining and our work.	47
2.4	Comparison of existing Web association rule mining and our work.	48
3.1	Evolutionary subtree database model.	51
3.2	Evolutionary tree database model.	52
3.3	Hierarchy of associative evolutionary structures mining problem.	53
3.4	Directory hierarchy.	54
3.5	Historical directory hierarchies I.	56
3.6	Historical directory hierarchies II.	57
3.7	XML query tree sequences.	58
3.8	Historical XML query trees.	60
4.1	Sequence of tree versions.	67
4.2	Algorithm of Apriori-FRACTURE.	76
4.3	<i>Optimized Apriori-FRACTURE</i> algorithm.	78
4.4	Transforming SDDb.	79

4.5	Signed-FPtree.	80
4.6	Algorithms of Signed-FPtree construction and FPG-FRACTURE.	82
4.7	Ordering subtrees in the head table of Signed-FPtree.	84
4.8	Mining process from s_1	87
4.9	<i>Maximal FRACTURE</i> mining algorithm.	89
4.10	Experiment results I.	92
4.11	Experiment results II.	93
4.12	Experiment results III.	94
4.13	Experiment results IV.	94
4.14	Experiment results V.	95
4.15	DBLP DTD structure.	97
4.16	Web log tree and discovered knowledge.	99
4.17	DoC values of Web log subtees.	100
5.1	Web session and document hierarchy.	105
5.2	Historical Web sessions.	106
5.3	Overview of COWES.	108
5.4	Historical Web session trees of u_2	110
5.5	Overall value of criterion functions versus number of clusters.	122
5.6	Similarity matrix ordered by clustering results.	124
5.7	<i>IS</i> and <i>ES</i> generated by partitional I_1 on different datasets.	125
5.8	Performance of different clustering methods on different datasets.	126
5.9	Similarity matrix ordered by clustering results I.	127
5.10	Similarity matrix ordered by clustering results II.	128
5.11	Comparison of clustering algorithms.	129
5.12	Comparison of similarity measures.	129
5.13	Shared <i>FRACTURES</i> of clustered hosts.	130
5.14	Performance variation with respect to thresholds.	131
6.1	Sequence of historical tree versions.	135
6.2	General delta tree and delta tree.	141
6.3	<i>Negative FRACTURE</i> mining algorithm.	143
6.4	Bloom histogram system.	148

6.5 *Negative FRACTURE* rule derivation and mirror maintenance algorithms. . 151

6.6 Experimental results I. 155

6.7 Experimental results II. 155

6.8 Experimental Results III. 156

6.9 Size evaluation of example sites. 157

6.10 Performance of mirror maintenance I. 159

6.11 Performance of mirror maintenance II. 159

6.12 Performance of mirror maintenance III. 161

7.1 Sequence of semantic trees. 164

7.2 Previous similarity measures. 169

7.3 Initial clusters. 171

7.4 Intra-cluster dissimilarity. 173

7.5 Clusters after disjointing. 174

7.6 Semantic clusters. 175

7.7 Mining *SEASONS* from evolutionary semantic cluster database. 177

7.8 Accuracy of clustering method. 185

7.9 Sensitivity to parameters. 186

7.10 Performance of clustering. 187

7.11 Performance of XML query caching strategy. 189

7.12 Average response time with respect to the training data size. 190

List of Tables

2.1	A sample transaction database.	20
4.1	Structural delta database.	71
4.2	Parameters list.	90
4.3	Results on DBLP data.	98
5.1	<i>FRACTUREs</i> discovered from Web session trees.	111
5.2	Parameters and Datasets.	120
5.3	Alternative similarity measures.	123
6.1	2×2 contingency table.	137
6.2	Parameters and Datasets.	153
7.1	An evolutionary semantic tree database.	165
7.2	Parameters and data sets.	183

Chapter 1

Introduction

The past decade has witnessed an explosive growth of the World Wide Web (WWW). This is caused by the widespread use of computers and internet, advances in hardware technologies, increased ease of use in computer software packages, tremendous opportunities the Web offers for all business, and so on.

“The World Wide Web has made great strides in the late 1990s. Its explosive growth is expected to continue into the next millennium” [29].

With the popularity of the Web as a global information system, it covers a huge amount of data from essentially every area such as news, business, entertainment, government, research, and many others. This fact makes the Web a rich source for various users such as Web surfers, business organizations, governments, etc. In turn, the huge amount of information online has flooded the users and made the searching of desired information a tedious and frustrating experience. Different Web users may encounter different problems when interacting with the Web. For example, when a Web surfer needs to find some relevant information, he usually uses a keyword based search tool. However, today’s search tools have problems in achieving high precision and recall because of the irrelevance of many of the search results and the inability to index all the information available on the Web. As another example, when a business organization needs to improve the effectiveness of their online marketing campaign by targeting the most potential customers, the problem of learning and analyzing mass online consumers may arise. Under such circumstances, **Web mining** techniques were proposed to assist searching for high quality information and uncovering knowledge buried in the World Wide Web. Web mining is defined in [58] as follows.

“Web mining is the use of data mining techniques to automatically discover and extract information from Web documents and services”.

Since its emergence in the late 1990s, Web mining has been a huge research area and the knowledge discovered by Web mining techniques has been used successfully in broad Web applications. For example, the link analysis technique [21], which is part of the Web mining techniques, has been used by Google¹ to rank the search results based on the quality of Web pages. However, we observed that most of the existing Web mining approaches mainly focus on the massiveness property of the Web data, ignoring other features such as the evolutionary nature of Web data. In this dissertation, we focus on the evolutionary nature of Web data and mine interesting patterns from the evolution of Web data. Particularly, we focus on *tree structured* Web data. We systematically study the problem of *associative evolutionary structures mining* by discussing the framework as well as techniques for extracting novel knowledge from changes to tree structured Web data.

In the remainder of this chapter, we begin by describing the key features of Web data in Section 1.1. In Section 1.2, we first provide an introduction to Web mining. Then, we motivate the need for our research by highlighting the limitations of existing Web mining techniques. In Section 1.3, we present an overview of this dissertation. We first discuss the architecture of *associative evolutionary structures* mining. Then, a list of applications of proposed associative evolutionary structures are illustrated. Contributions of this dissertation are summarized in Section 1.4. Finally, Section 1.5 outlines the organization of this dissertation.

1.1 Key Features of Web Data

Autonomous: One of the notable features of Web data is that Web data are autonomous. More and more companies are moving their business online. They create their own enterprise Web sites to attract online consumers. For example, Dell² is a company that has achieved success by selling products online. Governments are also providing information and services on the Web to facilitate daily routines. For example, the Inland Revenue Authority of Singapore provide an online portal³ so that taxpayers can transact electronically.

¹<http://www.google.com>

²<http://www.dell.com>

³<https://mytax.iras.gov.sg/ESVWeb/default.aspx>

Furthermore, each individual Web user can be an author of the Web by creating his home-pages, blogs etc. The autonomous nature of Web data can be regarded as the cause of the following three features.

Massive: The size of Web data is becoming more and more huge. The amount of Web data available has increased dramatically in the last decade. More importantly, massive volumes of new data are created everyday by different users with different purposes. According to the findings in [105], the new Web pages are created at the rate of 8% per week. Assuming the current Web has 4 billion pages [122], this result corresponds to 320 million new pages every week, which is roughly 3.8 terabytes in size⁴. It was predicted that the majority of human information will be available on the Web in the very near future in the Asilomar Report [15].

Diverse: Since everybody can be an author within the Web, the Web today provides a wealth of data in almost every field of science, technology, medicine etc. Besides the visible and available data to Web users, the Web contains other valuable information. As shown in Figure 1.1, Web data can be classified into three categories: Web content, Web structure, and Web usage [95]. Web content refer to explicit information on the Web. The Web content data can be decomposed into units of *Web documents*, which are named in the form of URL (Uniform Resource Locator). Hence, Web contents can be further divided into two categories: *intra-document* data and *inter-document data*. The former refers to the contents inside Web documents, such as text, graphics, and multimedia etc. The latter refers to the information among Web documents, such as the *directory hierarchy* of Web documents decided by their URLs. Web structures refer to structured information on the Web, which are usually classified as *intra-structure* and *inter-structure*. The former refers to the internal structure of the actual Web documents in HTML or XML format. The latter refers to the underlying structure of the Web formed by hyperlinks connecting Web documents. Web usages refer to the log information of Web users interacting with the Web, which can be broadly classified as *passive log* or *active log*. The former refers to the information generated when users access the Web in manner of browsing Web documents. The latter is generated when users access the Web by means of inputting profiles, issuing queries, such as keywords or XML queries etc. Different types of Web data can be modelled as different structures. For example, current research on textual intra-document contents

⁴The average page size in the data collected in their findings was around 12KB.

Figure 1.1: Types of Web data.

usually model the data as bags of words [159, 73, 56]. While, the inter-structure information is usually modelled as a graph [21, 83, 28]. In this dissertation, we focus our research on tree structured data⁵. For example, the data in bold blocks in Figure 1.1 can be modelled as trees, which are exemplified below.

Evolutionary: Another key feature of Web data which cannot be ignored is that Web data are dynamic. That is, Web data evolve over time. New data may be created at any time and existing data may be updated in different ways. We elaborate on the evolutionary feature in the context of tree structured Web data in the following.

Inter-document Data: Given the set of Web documents of a particular Web site, the inter-document information can be represented as a tree structure, called *directory hierarchy*, based on the URLs of the documents. A directory hierarchy is a partial ordering of the Web documents, in which a leaf node represents a Web page corresponding to a file in the server. A non-leaf node in a directory hierarchy represents a Web page corresponding to a directory in the server. A link from a parent node to a child node represents the consisting-of relationship between the corresponding pages. An example directory hierarchy constructed for Web documents in Figure 1.2 (a) is shown in Figure 1.2 (b). The directory hierarchy of Web documents changes along with the evolution of a Web site. For example, Figure 1.2 (c) shows a subsequent version of the directory hierarchy for the Web site at

⁵However, our research issues proposed in this dissertation can also be applied to other structures such as graph structured Web data.

Figure 1.2: Examples of Inter-document data.

`http://www.X.com`. It can be observed that the page *A* in the directory *previous* is deleted from the site, the page *C* is moved from the directory *latest* to the directory *previous*, a new page *E* is inserted in the directory *latest*, and a new directory *upcoming* and its child page *F* are inserted in the directory *products*.

Intra-structure Data: The intra-structure information of the actual Web documents in HTML or XML format are usually represented as trees as well. For example, Figure 1.3 (a) shows the structure of the XML document fragment, an excerpt of the XML version of DBLP [116], in Figure 1.3 (b). The XML document of DBLP data records the bibliographic information on major computer science journals and proceedings provided by the DBLP server. Hence, the XML document is evolutionary, e.g., more and more publications are collected by the DBLP server. Figure 1.3 (c) shows a later version of the document, from which it can be observed that a new publication with its key value as 193 is inserted.

Passive Log: The passive log of Web usage data are usually analyzed in the units of *Web session*. A Web session, which is an episode of interaction between a Web user and the Web server, consists of pages visited by a user in the episode [62]. For example, Figure 1.4 (a) shows four requests from one session. The first line means that the user at `foo.ntu.edu` accessed the page `www.uow.edu/sce/Jeffrey/pub.html` at 10 : 30 : 05 on January 01, 2005 with the protocol `HTTP/1.0`. The number 200 indicates that the request is successful and the number 3027 is the size of the requested page in bytes. The Web session of a user is evolutionary in nature. For example, Figure 1.4 (b) shows a Web session of the same user after one week. It can be observed that the user browsed different information under `www.uow.edu/sce/Jeffrey`. Web pages in a Web session can be organized into a

CHAPTER 1. INTRODUCTION

Figure 1.3: Examples of Intra-structure data.

Figure 1.4: Examples of access log data.

hierarchical structure, using the same method as that of extracting directory hierarchy for inter-document data. We, hence, omit showing the structure for Web sessions here.

Active Log: Active log of Web usage data contain queries raised by Web users, such as Web search engine query log and XML query log. Current Web search engines usually allow users to query via *keywords* [122, 124, 123]. Although keywords can be organized into tree structures based on their concept hierarchy [41], they are usually represented as bags or lists [80, 126]. Hence, we focus on the XML query data, which are typically modelled as trees [157]. For example, Figure 1.5 (a) shows the structure of the XML query, in Figure 1.5 (b), issued by some user at a point of time t_1 . Obviously, the tree structure reflects the semantic meaning of the query, i.e., it indicates that the user inquires the information of the book title, author, title of section and price. The information needs of a user may vary from time to time. For example, Figure 1.5 (c) shows another query issued by the same

Figure 1.5: Examples of query log data.

user at a subsequent time point t_2 . It can be observed that the user inquired about different information about the book.

1.2 Limitation of State-of-the-art Web Mining

Before motivating the need for our research, we first provide an introduction to the current research on Web mining. Then, we highlight the limitations of the existing works that fail to address the evolutionary feature of Web data.

1.2.1 Existing Web Mining - A Bird's-Eye View

Web mining is a converging research area from several research communities, such as data mining, information retrieval, artificial intelligence, statistics, Web technology, etc. The objectives of Web mining can be divided into four subtasks: *resource finding*, *information selection and pre-processing*, *generalization* and *analysis*. Resource finding refers to the task of retrieving intended Web documents. Information selection and pre-processing means automatically selecting and pre-processing specific information from retrieved Web resources. Generalization is to automatically discover general patterns at individual Web sites as well as across multiple sites. Finally, analysis refers to the validation and/or interpretation of mined patterns [84].

In the literature, many research efforts have been directed to the area of Web mining (Please refer to [84] for a survey of Web mining). Generally, Web mining can be classified into three categories with regard to the part of the Web to be mined: *Web content mining*, *Web structure mining*, and *Web usage mining*. *Web content mining* aims to discover knowledge from Web contents, particularly, the intra-document content, such as textual

data, image data, audio and video, etc. *Web structure mining* is the process of analyzing the structured information on the Web. Mining intra-structure is critical to extracting important information, such as schemas, for mediators and data warehouses [142]. Mining inter-structure is useful in identifying authoritative Web pages [83], ranking search results [21], constructing Web communities [59] and so on. The goal of *Web usage mining* is to discover meaningful information of Web users, such as access patterns and user profiles, from server side log files. The results of Web usage mining can be beneficial in applications such as Web caching and prefetching [158], Web information recommendation [155] and personalization [88] etc.

1.2.2 Limitations

Most of existing Web mining techniques focus on designing alternative data mining algorithms and applying Web mining results in specific Web applications. The massiveness feature is taken into account when developing data mining algorithms and the diversity feature is addressed when applying the discovered knowledge. However, very little work has been done concerning the evolutionary nature of Web data. Specifically, we observe the existing Web mining techniques have the following two limitations.

One of the key limitations is that, although Web data is evolutionary in nature, many existing research works ignore the temporal issue and analyze Web data without considering their associated timestamps. They typically mine knowledge from snapshots of the target data. For example, clustering Web users is one of the most important research topics in Web usage mining. Web user clustering aims to discover clusters of users that exhibit similar information needs, e.g., users that access similar pages [62]. By analyzing the characteristics of the clusters, Web designers may understand the users better and thus can provide more suitable, customized services to the users [152]. Existing Web user clustering methods [62, 146] cluster users based on the snapshots of their Web sessions. Consequently, generated clusters may not be very useful in subsequent analysis and applications since users accessing similar pages at a particular time window may access different pages at subsequent windows. As another example, caching XML queries has been recognized as an orthogonal approach to improve the performance of XML query engines [37, 157]. FastXMiner [157] is an algorithm proposed to mine frequent XML query patterns from a collection of users' query log. Discovered knowledge is used to design appropriate replacement policies, e.g., the

eviction of the discovered frequent queries will be delayed in expectation of higher caching hit rate. Nevertheless, frequent query patterns, discovered by ignoring the timestamps of queries, may not always be reliable in predicting the subsequent queries, because queries that are popular at a specific period may be unpopular at the subsequent periods. Therefore, the temporal information of Web data is a critical factor that cannot be ignored by Web mining technologies.

Another limitation of existing Web mining approaches is that, although the evolutionary nature of Web data has been considered by a few approaches [81, 57, 167, 166], research works addressing this concern are not complete. For example, due to the consideration that knowledge discovered from snapshots of Web data may not be valid when the data evolve, some efficient incremental Web mining technologies were proposed. Kamdar and Joshi [81] proposed to mine typical user profiles incrementally instead of creating new profiles, aiming to save time as well as physical memory requirements. FS-Miner [57] is a frequent access-pattern-mining algorithm which has the ability to adapt to changes in users' behavior over time. In the presence of new access data, FS-Miner responds incrementally without the need to perform full re-computation. Some approaches studying the statistics of Web evolution have been done as well. For example, Zhao et al. proposed to find individual substructures that change frequently in the history of an XML document in [167]. In [166], they further found individual substructures that change in increasing/decreasing manner in terms of structure size. However, ideas on Web mining research, addressing the evolutionary nature of Web data, can be explored further. For example, to the best of our knowledge, no work has been done to mine *frequent patterns* from *changes to Web data* to capture the *evolution association* of Web data. We present some novel research issues in this aspect below.

1.3 Novel Research Issues

Considering the evolutionary nature of Web data⁶, while knowledge extracted from snapshot of Web data is important, novel knowledge might be mined from changes to Web data. The underlying intuition is that, taking changes to Web data as the input of Web mining processes, we may find patterns that capture the evolutionary characteristics of Web

⁶Note that, in this dissertation, we consider the evolutionary nature of Web data from the perspective of sequential orders of historical versions only. Other issues, such as the time interval between two successive versions, are not the focus of this dissertation.

data, which cannot be discovered by mining snapshot Web data. While there are various data mining techniques, such as classification, prediction and clustering, can be applied on changes to Web data, and there are various types of Web data, we particularly discuss the issues of mining frequent patterns, the essential problem of association rule mining [5], from changes to tree structured Web data, called *structural delta*. We refer to the frequent patterns mined from structural delta of historical Web data as *associative evolutionary structures*. Then, the following issues are related to associative evolutionary structures.

- *Intra-tree or inter-tree*: Evolution associations can be discovered either from the substructures of trees (intra-tree) or from individual trees (inter-tree).
- *Positive association or negative association*: Similar to traditional frequent patterns [151], there are not only positive evolution associations but also negative evolution associations of structures.
- *Structural evolution or semantic evolution*: Since tree structures may have semantics when modelling semantic objects in the Web, associative evolutionary structures mining can be focused on either the evolution of structures (e.g., insertion/deletion of nodes/subtrees) or the evolution of semantics (e.g., modification of the meaning of the objects represented by tree structures).

With respect to the above issues, we now illustrate the specific types of novel knowledge, i.e., associative evolutionary structures, that we extract from structural deltas of historical Web data in this dissertation. Note that, the types of associative evolutionary structures that we consider in this dissertation are by no means complete. However, other types of associative evolutionary structures can be discovered by extending the framework we introduced here.

- ***FRACTURE*** (**F**requently **A**nd **C**urrently **mu**Tating substruct**U**REs): As indicated by the name, a *FRACTURE* pattern is a set of substructures that frequently undergo changes together. Hence, it captures positive evolution association in intra-tree and it considers changes from the structural evolution view. We discuss the details of *FRACTURE* pattern mining in Chapter 4 and one real application of *FRACTURE* in Chapter 5.

- **Negative FRACTURE**: While the positive evolution associations of substructures captured by *FRACTURE* patterns are interesting, the negative evolution association of substructures are also useful in some applications such as maintaining Web site mirrors. Hence, we further propose the *negative FRACTURE* pattern which refers to a set of subtrees that are negatively correlated in undergoing changes. Hence, *negative FRACTURE* captures negative evolution associations in intra-tree. It similarly considers changes from the structural evolution view. The details of *negative FRACTURE* are discussed in Chapter 6.
- **SEASON (SEmAntic Sequence of structural evOlutioN)**: While the previous two patterns consider changes from the structural evolution perspective, *SEASON* patterns are defined by viewing changes from the semantic evolution perspective. Furthermore, *SEASON* patterns are different from the previous two patterns in capturing evolution associations in inter-tree instead of intra-tree. A *SEASON* pattern refers to a frequent sequence of different semantics, represented by tree structures. In Chapter 7, we discuss the problem of *SEASON* pattern mining.

The technical challenges of mining the above patterns are presented in Section 3.3 of Chapter 3. The potential applications of the proposed patterns are discussed below.

1.4 Research Overview

Based on previous discussion, the objective of this dissertation is to discover novel knowledge from changes to historical tree-structured Web data. Specifically, we study the problem of mining *associative evolutionary structures* from structural delta of historical Web data. In this section, we first present the architecture of the associative evolutionary structure mining problem. To show the usefulness of the proposed associative evolutionary structures, we then present a list of applications where discovered patterns can be beneficial. Lastly, the scope of our research is defined.

1.4.1 Architecture

The general architecture of associative evolutionary structures mining is shown in Figure 1.6, which consists of three main modules: *change detection*, *evolution measurement* and *data*

Figure 1.6: Architecture of associative evolutionary structures mining.

mining. To verify the usefulness of the discovered patterns, we include another module: *applications*. We elaborate on the modules in turn as follows.

Firstly, given historical tree structured Web data associated with timestamps, a *change detection* module should be used here to detect changes, since we aim to mine novel knowledge from changes. Particularly, as we are interested in structural delta, the output of this component should contain only changes to structures.

With the detected structural delta, two modules function jointly: *evolution measurement* and *data mining*. The objective of the evolution measurement component is to use effective *evolution metrics* to evaluate the interestingness of structural deltas from different aspects, e.g., *frequency of change* and *significance of change*. Given a sequence of structural deltas, we use not only *local metrics*, which measure changes in two versions, but also *global metrics*, which measure changes from the perspective of the whole sequence of versions. Moreover, both metrics for singular substructure and metrics for multiple substructures are designed. In general, in order to discover interesting and useful associative evolutionary structures, meaningful evolution metrics should be defined soundly. We evaluate the effectiveness of proposed evolution metrics by mining associative evolutionary structures, which are defined based on these metrics, from real-life data sets to examine the discovered knowledge.

Various interesting associative evolutionary structures defined based on evolution metrics can be discovered by the data mining module. In this dissertation, we define three types of associative evolutionary structures, *FRACTURE*, *negative FRACTURE* and *SEASON*. Due to the significant difference between the proposed patterns and traditional frequent pattern [71], novel data mining algorithms need to be developed for mining the proposed patterns. The key issue of this module is to design efficient and scalable mining algorithms to discover desired patterns satisfying different constraints. We not only theoretically prove

the correctness of developed data mining algorithms but also evaluate the performance of the algorithms by conducting experiments on both synthetic data sets as well as real-life data sets.

Finally, the usefulness of associative evolutionary structures can be investigated by the *application* module. Different associative evolutionary structures have different applications. For example, since *FRACTURE* patterns capture structural evolution associations in intra-tree, they reflect the evolutionary characteristics of the tree to certain extent. Hence, *FRACTURE* patterns can be beneficial in applications where the knowledge of evolutionary features of tree structures is needed. On the contrary, since *SEASON* patterns capture semantic evolution associations, they can be used in applications where characteristics of semantic evolution are required. We present some representative applications of associative evolutionary structures in the following section.

1.4.2 Applications

Web user clustering: As we mentioned before, existing Web user clustering methods cluster users based on the snapshots of their Web sessions, although Web sessions are evolutionary in nature. Consequently, clusters generated in this manner may not be very useful in subsequent analysis since the information needs of users may vary over time. Hence, it makes sense to cluster users based on the evolution of their browsing behaviors, e.g., cluster users exhibiting similar variation on their information needs. Since *FRACTURE* patterns mined from changes to Web session trees can discover which information needs of a user are frequently changed together, *FRACTUREs* discovered from historical Web sessions of each user can be used as the clustering features. Then, clustering method can group Web users based on their *FRACTUREs*. Analyzing the characteristics of generated clusters, the browsing behaviors of Web users can be understood better, which benefits other Web applications such as Web caching, recommendations etc. The details of this application will be discussed in Chapter 5.

Maintenance of Web site mirrors: Web site mirroring is an important technique which can not only improve the availability of Web sites but also increase the speed with which Web sites can be accessed [16, 149]. For a fast evolving Web site directory hierarchy, it is straightforward that updating the directories frequently can keep the replica of the site on mirror servers more “fresh”. However, if the size of the directory hierarchy is

CHAPTER 1. INTRODUCTION

large, frequently updating all the directories results in heavy network load. Since *negative FRACTUREs* discovered from historical directory hierarchies indicate which directories rarely change together, the knowledge can be used in Web site mirror maintenance. For example, if a particular directory and one of its subdirectories are discovered as a *negative FRACTURE*, when maintaining the mirror of the directory, the subdirectory can be skipped to reduce some load without sacrificing too much in the “freshness” of the mirror. In Chapter 6, the details of applying *negative FRACTUREs* in Web site mirroring will be presented.

XML query caching: Caching XML queries has been recognized as an orthogonal approach to improve the performance of XML query engines [37, 157]. As discussed before, however, frequent query patterns discovered by ignoring the timestamps of queries may not always be reliable in predicting the subsequent queries, because queries that are popular in a particular period may not be popular at the subsequent periods. Hence, it is useful to discover how the semantics of queries evolve. Since a *SEASON* pattern, discovered from historical XML query trees, indicates that when a user inquires some information from an XML document, she/he will probably inquire some other information subsequently, the knowledge can be used in designing intelligent XML query caching replacement strategies. For example, we can delay the eviction of answers to queries predicted by *SEASONs* so that the query response efficiency can be improved. We discuss the details of applying *SEASONs* in XML query caching in Chapter 7.

1.4.3 Scope of Research

The scope of research in this dissertation is as follows.

- **Source Data.** As mentioned earlier, Web data is diverse. Different Web data can be represented as different models. In this dissertation, we focus our research on tree structured Web data. We address the evolutionary nature of Web data by associating timestamps with them. We then aim to mine novel knowledge from changes to historical Web data. Hence, the source data of our research are changes to tree structured Web data, i.e., structural delta. Note that, the problems related to collecting historical Web data, e.g., how to set the frequency of data collection etc., are excluded from our research. We assume historical Web data are available and complete.

- ***Desired Knowledge.*** Various knowledge may be discovered from changes to tree structured Web data, such as classifiers, prediction, and clusters and so on. In this dissertation, we focus on the problem of mining frequent patterns from structural delta, which are called *associative evolutionary structures* mining. Particularly, we mine three types of associative evolutionary structures: *FRACTURE*, *negative FRACTURE* and *SEASON*.
- ***Architecture Component.*** Since there exist effective algorithms on detecting changes to tree structures [147, 43], the module of *change detection* in the architecture will not be included in the research scope of this dissertation. That is, we focus our research issues in the modules in bold blocks in Figure 1.6, which are *evolution measurement*, *data mining*, and *applications*.

1.5 Contributions

The main contributions of this dissertation are fourfold.

- **The architecture of associative evolutionary structures mining.** To the best of our knowledge, this is the first effort to mine frequent patterns from changes to historical tree-structured Web data. Main modules of the associative evolutionary structures mining architecture and their respective functions are explained explicitly.
- **Evolution metrics.** Meaningful evolution metrics are defined to measure the structural deltas from different perspectives. Based on the evolution metrics, interesting and useful associative evolutionary structures, *FRACTURE*, *negative FRACTURE* and *SEASON*, can be defined.
- **Data mining algorithms.** We develop efficient and scalable data mining algorithms to discover desired patterns with different constraints. Properties of patterns are investigated to optimize the mining algorithms. Alternative mining algorithms are developed and implemented to compare performance.
- **Real applications.** To show the usefulness of defined associative evolutionary structures, we apply the discovered knowledge in real Web applications. Specifically, discovered *FRACTURE* patterns are used in Web user clustering, discovered *negative*

FRACTURE patterns are applied in the maintenance of Web site mirrors, and discovered *SEASON* patterns are used in designing appropriate XML caching replacement policies.

1.6 Organization

The remainder of this dissertation is organized as follows.

- In Chapter 2, we survey related works in the literature. Basically, we review two general areas: association rule mining, Web mining. Particularly, we focus on the problem of frequent pattern discovery in association rule mining and mining association rules from Web data. The comparison of our work with existing works is presented as well.
- In Chapter 3, we introduce the proposed associative evolutionary structures: *FRAC-TURE*, *negative FRACTURE* and *SEASON*. We formally define the research problem of this dissertation. We also discuss the main challenges in mining the proposed patterns.
- In Chapter 4, we discuss the problem of *FRACTURE* pattern mining. We define a set of evolution metrics, *DoC* (*Degree of Change*), *FoC* (*Frequency of Change*) and *SoC* (*Significance of Change*), to evaluate the interestingness of *FRACTURE* patterns. We design two algorithms, which employ the *level-wise* and *pattern-growth* strategies respectively, to mine *FRACTURE* patterns. We further optimize the problem to define the *maximal FRACTURE* patterns. A set of optimizing strategies are proposed for maximal pattern mining.
- In Chapter 5, we describe an application of *FRACTURE*. We present a new method, *COWES* (**C**lustering **O**f **W**eb users based on historical **wEb** **S**ession), which use the *FRACTUREs* mined from changes to Web session trees as clustering features. We define *FRACTURE*-based similarity metrics to measure the proximity of Web users. Different clustering algorithms are employed to compare the performance.
- In Chapter 6, we extend the model of *FRACTURE* to discover the negative evolution associations of substructures. We introduce the evolution metric, *CoC* (*Correlation of Change*), based on which the interestingness of *negative FRACTURE* patterns can

CHAPTER 1. INTRODUCTION

be evaluated. A *divide-and-conquer* algorithm is developed for *negative FRACTURE* mining. Finally, the application of *negative FRACTURE* in maintaining Web site mirrors is presented.

- In Chapter 7, we introduce the *SEASON* pattern which captures the semantic evolution associations in tree versions. We present a new algorithm to generalize the semantics of tree structures. We derive not only positive association rules but also negative association rules from *SEASON* patterns. Then, derived rules are applied in designing optimal XML query caching replacement strategies. Experiments are conducted to demonstrate the effectiveness of *SEASON*-based replacement strategies.
- In Chapter 8, we conclude this dissertation and point out the future research topics.

Figure 1.7 shows the structure of this dissertation.

CHAPTER 1. INTRODUCTION

Figure 1.7: Dissertation structure.

Chapter 2

Literature Survey

We present in this chapter prior research work that is closely related to the topics discussed in this dissertation. We begin by reviewing one of the major data mining techniques: *association rule mining* (Section 2.1). After giving some basic definitions of the problem, we introduce *frequent patterns*, *infrequent patterns*, *maximal* and *closed patterns* to illustrate a diversity of pattern types in this area. Then, we review the work of Web mining on the content data, the structure data and the usage data respectively (Section 2.2). Particularly, we will focus on the tree structured Web data. In addition, we describe some related work in detecting changes to tree-structured Web data (Section 2.3). Finally, we summarize this chapter by highlighting the relation between prior works and our research (Section 2.4).

2.1 Association Rule Mining

A major data mining task is to discover dependencies among data. Association rules [5] are proposed to achieve this goal. In the context of supermarket business, association rules are used to describe such a dependency that the presence of some items in a transaction implies, with a certain probability, the presence of other items in the same transaction. An example of such an association rule is the statement that 90% of transactions for the purchase of **bread** also include the purchase of **milk**. In the following, we first provide some preliminary definitions and then introduce different patterns and outline their discovering algorithms.

2.1.1 Basic Definitions

In relational databases, the most elementary term is called an **attribute**. An attribute has its domain values. The values can be **discrete** or **continuous**. For example, **colour** can be

Table 2.1: A sample transaction database.

TID	List of items
T_1	apple, bread, cabbage
T_2	bread, cabbage, egg
T_3	bread, cabbage, donut, egg
T_4	bread, donut

an attribute in some databases and the value of **colour** can be **red**, **yellow**, and **blue**. Another example is the attribute of **age** which can have continuous values in the range of $[0, 150]$. In transactional databases, a type of relational database, often the attributes are the names of merchandise such as **bread**, **milk**, and **egg**. Such attributes are binary with values of either 0 or 1. An attribute with the value 1 means that this merchandise was bought, otherwise it was not. We call the attribute-value pair an **item**. So, **colour**–**red** is an item, and **bread**–1 is also an item. A set of items is simply called an **itemset**. We define a set of items as a **transaction** if the attributes of the items represent the merchandise purchased by a customer in a transaction. A **database**, or a **data set**, is defined as a set of transactions. The **cardinality** of a database $D = \{T_1, T_2, \dots, T_n\}$, where T_i is a transaction, is the number of transactions in D , treating D as a normal set. The **dimension** of D is the number of attributes involved in D .

Table 2.1 is an example transactional database, which contains four transactions. There are five attributes involved in the database: **apple**, **bread**, **cabbage**, **donut** and **egg**. Note that only items with attribute values as 1 occur in transactions. For example, the first transaction indicates that **apple**, **bread** and **cabbage** were bought, but **donut** and **egg** were not.

Now, we turn to other basic definitions. We state that a transaction T contains an itemset X (or, X occurs in T), if $X \subseteq T$.

Definition 2.1 (Support) *Given a database D and an itemset X , the count of X , denoted as $\text{count}_D(X)$, is the number of transactions in D containing X . The support of X in D , denoted as $\text{supp}_D(X)$, is the percentage of transactions in D containing X .*

$$\text{supp}_D(X) = \frac{\text{count}_D(X)}{|D|}$$

where $|D|$ is the cardinality of D . Note that, the subscript D in both $\text{count}_D(X)$ and $\text{supp}_D(X)$ can be omitted if D is clear from the context. $\square$

Definition 2.2 (Frequent Itemset) Given a database D and a real number δ ($0 \leq \delta \leq 1$), an itemset X is defined as a frequent itemset if $\text{supp}_D(X) \geq \delta$. $\square$

Following Table 2.1, the support of itemset $\{\text{apple}\}$ is 25% because $\{\text{apple}\}$ occurs in one out of a total four transactions in D . Let $X = \{\text{bread}, \text{cabbage}\}$ and $\delta = 25\%$, then $\text{supp}(X) = \frac{3}{4} = 75\% > \delta$. Thus, X is a frequent itemset.

Definition 2.3 (Association Rules) Given a database D , an **association rule** is an implication of the form $X \rightarrow Y$, where X and Y are two itemsets in D and $X \cap Y = \emptyset$. The itemset X is called the **antecedent** of the rule, and the itemset Y is called the **consequent** of the rule. $\square$

Definition 2.4 (Support and Confidence of an Association Rule) Given a database D and an association rule $X \rightarrow Y$, the **support** of the rule is the percentage of the transactions in D that contain $X \cup Y$. The **confidence** of the rule is the percentage of the transactions in D that, containing X , also contain Y . $\square$

Note that the support of the rule $X \rightarrow Y$ is $\text{supp}(X \cup Y)$ and the confidence equals to $\frac{\text{count}(X \cup Y)}{\text{count}(X)}$. In Table 2.1, for the rule $\{\text{bread}, \text{cabbage}\} \rightarrow \{\text{egg}\}$, its support is $\text{supp}(\{\text{bread}, \text{cabbage}, \text{egg}\}) = 50\%$, and its confidence is $\frac{\text{count}(\{\text{bread}, \text{cabbage}, \text{egg}\})}{\text{count}(\{\text{bread}, \text{cabbage}\})} = \frac{2}{3} \approx 66.7\%$.

The problem of mining association rules is to generate all association rules that have support and confidence greater than or equal to the user-specified minimum support (called *minsup*) and minimum confidence (called *minconf*) respectively [5].

The problem of mining association rules can be decomposed into two subproblems:

- (i) Generate all frequent itemsets with respect to the support threshold *minsup*.
- (ii) For a given frequent itemset, generate all association rules whose confidence values satisfy the confidence threshold *minconf*.

The second subproblem is simple and straightforward. Therefore, the key efficiency issue of association rule mining is the first subproblem, namely discovering all the set of frequent itemsets.

A naïve approach to solving this problem is to generate all possible itemsets in the database D and then check whether their support values meet the threshold. Take the

database in Table 2.1 as an example. Since there are five attributes in the data set, one would check the support for $2^5 - 1$ itemsets (except the empty set). Obviously, this task would quickly become impossible as the number of attributes increases. This is because the number of candidate itemsets increases exponentially when the number of attributes increases.

The problem has been extensively studied by the data mining community for a long time. A lot of efficient algorithms have been proposed which partially overcome the difficulty imposed by the naïve algorithm. We describe some representative methods in the following.

2.1.2 Frequent Pattern

As discussed above, frequent itemset discovery plays an essential role in association rule mining on transactional database. During the past decade, the problem has been proved to be useful in other data sets, such as sequences, trees and graphs. The idea of frequent itemsets thus was extended to frequent sequences, frequent trees and frequent graphs. We call the combination of them as frequent patterns. In this subsection, we focus on the discovery of frequent itemsets first. Then, we briefly discuss the other patterns and their discovering algorithms.

2.1.2.1 Frequent Itemset

Many algorithms have been proposed to improve the efficiency of frequent itemset discovery. Generally, these algorithms fall into two categories: algorithms discovering frequent itemsets in *level-wise* manner and algorithms discovering frequent itemsets in *pattern-growth* manner. We describe two typical algorithms, Apriori [7] and FP-Growth [71], of this two categories respectively in the below.

Apriori Algorithm We first give an additional definition for describing the Apriori algorithm. A k -**itemset** is defined as an itemset consisting of k items. The itemset $\{a, b, c\}$ is a 3-itemset. The number k is called the **length** (or **cardinality**) of the itemset. Given a database D and a support threshold δ , obviously, the collection of the frequent itemsets in D is the union of all frequent 1-itemsets, all frequent 2-itemsets, ..., and all frequent m -itemsets, where m is the longest length of the frequent itemsets in D . In brief, the Apriori algorithm sequentially outputs the frequent k -itemsets with k from 1 to m . Hence, the Apriori algorithm is referred to as a *level-wise* algorithm.

The basic idea of the Apriori algorithm is that any subset of a frequent itemset is frequent. This property is called the apriori property. It is employed in the algorithm to narrow search spaces: generating a strict number of frequent itemset candidates which is much smaller than the number of candidates generated by the naïve method. For example, given all the frequent k -itemsets, the Apriori algorithm only generates those $(k + 1)$ -itemsets as candidates if their *immediate* proper subsets (having k items) are frequent. In other words, the Apriori algorithm does not generate any of those $(k + 1)$ -itemsets as candidates for which there exists any immediate proper subset which is not frequent.

Briefly, the algorithm consists of the following three main steps. The initial step is to set the iteration parameter k to 1.

- Step 1: Generating a collection of frequent itemset candidates which each has k items;
- Step 2: Scanning the database and calculating the supports of the candidates;
- Step 3: $k \leftarrow k + 1$.

The three steps are repeated until no new frequent itemsets are found.

The efficiency of the computation in Step 1 is important to the efficiency of the whole algorithm: generating more unnecessary candidates would apparently increase the complexity. Agrawal and Srikant [7] proposed the **Apriori-Gen** function, making use of the apriori property, for candidate generation. We use an example to illustrate the key points in the **Apriori-Gen** function. Suppose the collection of all the frequent 3-itemsets in some database be $\{\{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{a, c, e\}, \{b, c, d\}\}$. After a joint step in the **Apriori-Gen** function, the candidate collection of 4-itemsets will be $\{\{a, b, c, d\}, \{a, c, d, e\}\}$. After a pruning step, the itemset $\{a, c, d, e\}$ will be removed from the candidate set because the itemset $\{a, d, e\}$ is not the set of frequent 3-itemsets. As a result, only the itemset $\{a, b, c, d\}$ is a candidate and needs support calculation.

The Apriori algorithm achieves a good performance by reducing the size of candidate itemsets. Refinements to Apriori include Partition [119], DHP [108], sampling [136], DIC [20], and parallel Apriori [6]. Partition reads the database at most two times to generate all significant association rules while generating no false negatives. It is also inherently parallel in nature and can be parallelized with minimal communication and synchronization between partitions. DHP is a hash-based algorithm for generating candidate itemsets

that reduces the number of candidate 2-itemsets by an order of magnitude. Pruning the number of candidate 2-itemsets significantly reduces the number of frequent k -itemsets that need to be considered when $k > 2$. Sampling is used to take a random sample from a database to find all association rules that are probably valid in the entire database. A second pass of the database is used to verify the support for each potential frequent itemset. DIC also partitions the database into blocks. When scanning the first block, it only counts the 1-itemsets. When scanning the k -th block, it counts all the 2-, 3-, ..., k -itemsets. Usually, it can finish counting all the itemsets in two passes over the data. Parallel Apriori is a parallel version of Apriori that exhibits excellent scaleup behavior and requires minimal additional overhead compared to serial Apriori.

FP-Growth The most important contribution of Apriori is that the set of candidate itemsets is reduced significantly. However, there exist two bottlenecks of the Apriori algorithm. One is the complex candidate generation process which is time- and space- consuming. The other is the multiple scans of the database. Consequently, when the database contains long itemsets, the performance of Apriori deteriorates greatly. FP-Growth [71] is an algorithm proposed to break the two bottlenecks.

FP-Growth does not generate any candidate itemset. In order to avoid the multiple passes over the database, FP-Growth reconstructs the database into an information-condensed tree called *FP-Tree*, from which frequent itemsets are mined directly. The FP-tree, which consists of an extended prefix-tree structure and a header table, is constructed as follows.

- Step 1: Scan the database and find the set of frequent individual items. Sort the frequent items in support descending order as L .
- Step 2: Create the root node, labelled as *null*, of the FP-Tree *tree*. Scan the database for the second time. For each transaction T , select and sort frequent items in T according to the order of L . Let the sorted frequent item list in T be $\langle p|P \rangle$, where p is the first item and P is the remaining list. Call the function $insert_tree(\langle p|P \rangle, tree)$.

The function $insert_tree(\langle p|P \rangle, tree)$ is performed as follows. If *tree* has a child n such that n is labelled as p , the count of the n will be incremented by 1. Otherwise, a new node,

labelled as p , will be created as a child of $tree$. The process is recursively performed on P until it is empty.

In addition, when creating a new node in the FP-Tree, a node link, which indicates the previous node in the tree carrying the same label, is created simultaneously. The header table is used to maintain the frequent individual items and their first occurring nodes in FP-Tree. For example, let the support threshold be 30%. The FP-Tree constructed for the database in Table 2.1 is shown in Figure 2.1 (a), where b represents **bread**, c represents **cabbage**, d represents **donut** and e represents **egg** (a is not included because it is not frequent).

The frequent itemsets are then discovered from the FP-Tree with the *divide-and-conquer* strategy. For example, given the FP-Tree in Figure 2.1 (a), the frequent itemsets are discovered in the following order: itemsets involving item e , itemsets involving d but e , itemsets involving c but d and e , and so on and so forth. When discovering frequent itemsets involving a particular item x , all branches reaching the nodes labelled as x are collected by following the node links of the item and a *conditional FP-Tree* will be constructed. For example, in order to discover frequent itemsets involving item e , two branches are collected: $\langle b : 1, c : 1 \rangle$ and $\langle b : 1, c : 1, d : 1 \rangle$ (the number after colon is obtained from the count of node e in the branch and it indicates the number of times the items occur together with e). Obviously, d is infrequent with e so that it is pruned. Then, the conditional FP-Tree for item e can be created as in Figure 2.1 (b), from which we recursively discover frequent itemsets involving e and c , and frequent itemsets involving e and b but c . The process of mining frequent itemsets for other items is similar.

As a whole, the FP-Growth algorithm needs to scan the database for only two times and the cost of constructing a FP-Tree is attested to be less expensive than generating the set of candidate itemsets. It improves the efficiency by nearly an order of magnitude with respect to the Apriori algorithm. However, FP-Growth has some drawbacks in constructing a special data structure from original database, which makes FP-Growth unfit for dynamic or incremental mining. Once the support threshold or the original database changes, the FP-Tree should be reconstructed. Recently, several approaches have been proposed to improve the FP-Growth algorithm: CT-PRO [130], nonordfp [114] and FP-Growth-Tiny [107]. CT-PRO is a bottom-up non-recursive frequent itemset mining algorithm using compressed FP-Tree data structure. Nonordfp is a variation of FP-Growth which does not rebuild any conditional FP-Tree. FP-Growth-Tiny optimizes the FP-Growth algorithm in space consumption by eliminating the need for intermediate conditional pattern bases.

CHAPTER 2. LITERATURE SURVEY

Figure 2.1: An example of FP-Tree.

2.1.2.2 Frequent Sequence

Frequent itemsets are discovered from transactional database which ignores the order between the items. However, time stamp is an important dimension of the transactional database. For example, a transaction in the market basket data typically consists of not only the items purchased but also the date of the transaction. Thus, frequent sequential pattern mining [8] was introduced to address the time issue in transaction database. An example of a frequent sequential pattern is that *customers typically rent "Star Wars", then "Empire Strikes Back", and then "Return of the Jedi"*. While frequent itemsets indicate the intra transaction relationship, frequent sequences extract the inter transaction correlation. In other words, frequent itemsets are about which items are bought together frequently. These items must come from the same transactions. Frequent sequences are about which items are bought in a certain order by the same customer. These items come from different transactions.

A **sequence** s denoted by $\langle s_1 s_2 \dots s_l \rangle$ is an ordered list of itemsets, where s_j is an itemset. A sequence $\alpha = \langle a_1 a_2 \dots a_n \rangle$ is called a **subsequence** of another sequence $\beta = \langle b_1 b_2 \dots b_m \rangle$ and β a **supersequence** of α , denoted as $\alpha \subseteq \beta$, if there exist integers $1 \leq j_1 < j_2 < \dots < j_n \leq m$ such that $a_1 \subseteq b_{j_1}, a_2 \subseteq b_{j_2}, \dots, a_n \subseteq b_{j_n}$. A sequence database S is a set of tuple $\langle sid, s \rangle$, where sid is a sequence identity and s is a sequence. A tuple $\langle sid, s \rangle$ is said to contain a sequence α , if α is a subsequence of s .

Definition 2.5 (Support) Given a sequence database S and a sequence α , the support of

α , denoted as $\text{supp}_S(\alpha)$, is the percentage of tuples in S containing α .

$$\text{supp}_S(\alpha) = \frac{|\{ \langle \text{sid}, s \rangle | ((\text{sid}, s) \in S) \wedge (\alpha \sqsubseteq s) \}|}{|S|}$$

where $|S|$ is the number of tuples in S . The subscript S can be omitted if the sequence database is clear from the context. $\square$

Definition 2.6 (Frequent Sequence) Given a sequence database S and a support threshold δ ($0 \leq \delta \leq 1$), a sequence α is defined as a frequent sequence if $\text{supp}_S(\alpha) \geq \delta$. $\square$

Given a sequence database and a support threshold minsup , the problem of frequent sequential pattern mining is to find all frequent sequences, i.e., the sequences whose support is no less than minsup . Many previous studies contribute to the efficient mining of frequent sequences [8, 129, 161, 70, 111]. Similar to frequent itemset mining, the algorithms fall into two categories as well: one is the apriori-like algorithms, and the other is pattern-growth algorithms.

The apriori-like algorithm is based on the a priori property, which states the fact that *any supersequence of an infrequent sequence cannot be frequent*. Thus, frequent sequences can be mined based on the level-wise paradigm as in frequent itemset mining. For example, as a typical apriori-like frequent sequence mining algorithm, GSP [8] adopts a multiple-pass, candidate generation-and-test approach outlined as follows. The first scan finds all of the frequent items that form the set of single item frequent sequences. Each subsequent passes starts with a *seed set* of frequent sequences, which is *the set of frequent sequences found in the previous pass*. This seed set is used to generate new potential frequent sequences, called *candidate sequences*, based on the a priori property. Each candidate sequence contains one more item than a seed sequence. So, all the candidate sequences in a pass will have the same length (i.e., the number of items). The scan of the database in one pass finds the support for each candidate sequence. All the candidates with support no less than minsup in the database form the set of the newly found sequential patterns. This set is then used as the seed set for the next pass. The algorithm terminates when no new sequential pattern is found in a pass, or when no candidate sequence can be generated.

The pattern-growth algorithms adopt the divide-and-conquer strategy and discover frequent sequences as follows. The sequence database is recursively projected into a set of

CHAPTER 2. LITERATURE SURVEY

smaller projected databases based on the current frequent sequence(s), and frequent sequences are grown in each projected databases by exploring only locally frequent fragments. For example, FreeSpan [70] is a pattern-growth based algorithm which employs the above philosophy directly. PrefixSpan [111] offers ordered growth and reduced projected databases with the pseudoprojection technique.

By taking into account the order between items, frequent sequences extract more accurate and informative knowledge, which allows the retail organizations make more sound decisions. Apart from the analysis of customer purchase patterns, frequent sequences have broad applications in other domains such as mining sequential Web user access patterns, analyzing DNA sequences and using the history symptoms to predict certain kind of disease.

2.1.2.3 Frequent Structure

Frequent itemset mining and frequent sequence mining are one-dimensional structure mining. A recent trend in frequent pattern mining is to consider more complex cases which have two-dimensional or high-dimensional structures, such as tree databases and graph databases. The importance of high-dimensional structure mining is reflected in the large domain of applications in computer networking, Web mining, bioinformatics, multi-relational data mining, XML document mining, etc.

The problem of frequent structure mining can be generally defined as follows.

Definition 2.7 (Frequent Structure Mining) *Given a class of structures (trees or graphs) D , a support threshold minsup , and a transitive substructure relation $X \preceq T$ between two structures $X, T \in D$, the frequent structure mining problem is the problem of finding all structures $\mathcal{X} \subseteq D$ such that no two structures in $\mathcal{X}$ are isomorphic and for all $X \in \mathcal{X}$: $\text{supp}_D(X) = \frac{\sum_{T \in D} d(X, T)}{|D|} \geq \text{minsup}$, where d is an anti-monotone function such that $\forall T \in D : d(X', T) \geq d(X, T)$ if $X' \preceq X$. The substructure relation $X \preceq T$ defines whether a structure X occurs in a structure T . The simplest choice for function d is given by the indicator function:*

$$d(X, T) = \begin{cases} 1, & \text{if } X \preceq T \\ 0, & \text{otherwise.} \end{cases}$$

□

For frequent tree mining, many algorithms have been proposed to discover frequent subtrees from databases of labeled trees. These algorithms vary in the specific problem

formulations and their solution details. For example, there are many types of trees such as *free tree*, *rooted unordered tree* and *rooted ordered tree* etc. There are also different notions of subtrees such as *bottom-up subtree* [138], *induced subtree* and *embedded subtree*. As the frequent bottom-up subtree mining can be transformed to be the problem of string matching, most research efforts focus on the frequent embedded or induced subtree mining. For example, the TreeMiner algorithm developed by Zaki [162] is an algorithm which uses the combined depth-first and breadth-first traversal idea to discover all frequent embedded subtrees from a database of rooted ordered trees. The FREQT algorithm developed by Asai et al. [12] uses an extension-only approach to find all frequent induced subtrees in a database of one or more rooted ordered trees. Unot [13] extends the FREQT algorithm to mine unordered induced subtrees in the general case that siblings may have the same labels. Under the assumption that no two siblings are identically labeled, PathJoin [153] mines frequent unordered induced subtrees by joining root paths, which are paths between the root of a tree and leaf nodes in the tree. Generally, to deal with the potentially exponential number of frequent induced subtrees and embedded subtrees, proposed algorithms usually adopt the generate-and-test method, which can be summarized in the following two steps: 1) compute $\text{supp}(X)$ by determining all $T \in D$ s.t. $X \preceq T$; 2) let $X = \text{succ}(X)$, goto 1). Here, $\text{succ}(X)$ is a function which computes the successor of a tree X , such that all possible trees up to the maximum size are enumerated exactly once if one would repeatedly call $\text{succ}(X)$.

For frequent graph mining, there have also been studies in the literature. Inokuchi et al. [77] presented the AGM breadth-first algorithm for mining frequent unconnected graphs in a graph database. The AcGM algorithm [78] restricts AGM to only find connected subgraphs. Kuramochi et al. [85] presented another breadth-first algorithm for mining frequent subgraphs, FSG, which also uses the join operation to generate candidate subgraphs from the frequent subgraphs in the previous level. The AGM algorithm extends subgraphs by adding a vertex per level and the FSG algorithm by adding an edge. The frequent connected subgraph problem was also tackled by Yan et al. [156] in the pure depth-first algorithm gSpan and by Huan et al. [75] in the combined depth-first and breadth-first algorithm FFSM. However, to check if a structure supports a graph is an instance of the subgraph isomorphism problem which is NP-complete [65]; to check if two graphs are isomorphic (in order to avoid creating a candidate multiple times) is an instance of the graph isomorphism problem which is not known to be in either P or NP-complete [65].

2.1.3 Infrequent Pattern

The research work on mining frequent patterns has focused primarily on patterns that occur frequently in the data, i.e., patterns with high support. Any patterns that do not have sufficient support are assumed to be statistically insignificant and therefore, eliminated. However, there are situations in which such a filtering strategy may end up removing a huge amount of useful information. For example, patterns involving anti-correlated sets of items can be quite informative even though they seldom co-occur together. We now describe two types of associations, complementing the traditional association rules, which consider the infrequent patterns.

2.1.3.1 Negative Association

All the traditional association rule mining algorithms were developed to find positive associations between items. By positive association we refer to an association that implies the presence of some itemset Y depending on the occurrence of some other itemset X , i.e., a rule of the form $X \rightarrow Y$. In addition to the positive associations, the negative associations can provide added information about the relation between itemsets. An example of a negative association rule is the statement that 75% transactions for the purchase of **bread** do not include the purchase of **cake**. As positive associations, negative associations are equally important in revealing the buying behavior of customers and in devising market strategies.

Brin et. al [19] mentioned for the first time in the literature the notion of negative relationships. Their model is chi-square based. They use the statistical test to verify the independence between two variables. To determine the nature (positive or negative) of the relationship, a correlation metric was used. In [120], the authors presented a new idea to mine strong negative rules. They combined positive frequent itemsets with domain knowledge in the form of a taxonomy to mine negative associations. However, their algorithm is hard to generalize since it is domain dependant and requires a predefined taxonomy. A similar approach was employed in [160]. In [134], the authors used only negative associations of the type $X \rightarrow \neg Y$ to substitute items in market basket analysis. Hence, different models of negative associations exist in the literature, which results in the proposals of different discovering algorithms. We give a short description of two algorithms that can generate both positive and negative association rules in the following.

Wu et. al [151] added on top of the *support-confidence* framework another measure called *mininterest*. According to Piatetsky-Shapiro's argument that two itemsets X and Y are independent if $\text{supp}(X \cup Y) \approx \text{supp}(X) \times \text{supp}(Y)$, the *mininterest* is defined as $|\text{supp}(X \cup Y) - \text{supp}(X)\text{supp}(Y)|$. Similar to traditional association rule mining, the authors first generated both frequent itemsets and infrequent itemsets with the level-wise strategy. For example, after scanning the database for the k -th time, the support values of the candidate k -itemsets are calculated. If the support of a candidate satisfies the support threshold, it is a frequent k -itemset. Otherwise, it is an infrequent k -itemset. Frequent k -itemsets are used to generate candidates of the next turn. Generated frequent itemsets and infrequent itemsets are pruned if there exists no pair of itemsets, X and Y , satisfying the *mininterest* threshold. In the second phase of rule derivation, the authors revised the confidence measure so that the dependence between the antecedent and the consequent of a rule is integrated. By checking the revised confidence of possible rules, positive rules are generated from the set of frequent itemsets of interest while negative association rules are generated from the set of infrequent itemsets of interest.

It is well known that the more parameters a user is given, the more difficult it becomes to tune the system. Motivated by this fact, Antonie and Zaïane [11] designed a new model of negative association rule mining which avoids the addition of new parameters. Instead of adopting the Piatetsky-Shapiro's argument, the authors used the *correlation coefficient* as the measure of interestingness, which was thoroughly studied in the statistics community. For example, it is studied that the value 0.5 means the correlation is strong, the value 0.3 means the correlation is moderate. Thus, the threshold of correlation can slide progressively and automatically. The mining algorithm is different from the existing association rule mining algorithms in that the two phases are combined and rules are derived on-the-fly while analyzing the correlation with each candidate itemset. For each candidate itemset, correlations of all possible combinations of items are analyzed. If the correlation between X and Y of an itemset XY is negative, negative association rules, $X \rightarrow \neg Y$ and $\neg X \rightarrow Y$, are generated when the confidence is high enough (even if the support is insufficient). If the correlation is positive, a positive association rule with the classical support-confidence idea is generated. If the support is not adequate, a negative rule of the form $\neg X \rightarrow \neg Y$ is generated when its confidence and support are high enough.

2.1.3.2 Indirect Association

The notion of indirect association was originally introduced in [132] to capture the higher order dependencies in market-basket type of transactions. Indirect association is closely related to the concept of negative association rules in that both deal with itemsets that do not have sufficiently high support. Typically, the number of negative association rules in a data set can be prohibitively large, and the majority of them are uninteresting to the data analyst. In contrast, the model of indirect association assumes that an interesting negative association exists if the items share a common set of other items. Consider a pair of items, x and y , which are rarely present together in the same transaction. If both items are highly dependent on the presence of another itemset M , then the pair (x, y) is said to be indirectly associated via M . There are many potential applications for indirect associations, such as competitive product analysis and stock data analysis etc, as investigated in [132]. The idea of indirect association is also extended to sequential data in [131]. In the following, we also briefly review two typical algorithms for indirect association mining.

The first algorithm for indirect association mining was presented in [132]. Basically, there are two phases involved: 1) extract all frequent itemsets using standard frequent itemset mining algorithm Apriori; 2) discover valid indirect associations by candidate generation and candidate pruning. In the candidate generation step, the set of frequent itemsets L_k is used to generate candidate indirect associations for pass $k + 1$, i.e., C_{k+1} . Each candidate in C_{k+1} is a triplet $\langle x, y, M \rangle$, where x and y are the items that are indirectly associated via the mediator M . C_{k+1} is generated by joining the frequent itemsets in L_k . During the join, a pair of frequent k -itemsets, $\{x_1, x_2, \dots, x_k\}$ and $\{y_1, y_2, \dots, y_k\}$, are joinable if the two itemsets have exactly $k - 1$ items in common and thus produce a candidate indirect association $\langle x, y, M \rangle$, where x and y are the distinct items, one from the each k -itemset, and M is the set of common items. For example, two frequent itemsets, $\{a, b, c, d\}$ and $\{a, b, d, e\}$, can be joined together to produce a candidate indirect association, $\langle c, e, \{a, b, d\} \rangle$. In the steps for candidate pruning, the support between x and y and the dependence between x and M , y and M are checked.

There are two join steps in the above algorithm, which makes it computationally expensive. The HI-Mine [140] algorithm was proposed to address this problem. HI-Mine constructs a special data structure, HI-Struct, for the data set. The HI-Struct is a dynamic structure which progressively projects the related transactions for frequent items. For each

frequent item x , HI-Mine generates the *infrequent itempair set* $IIS(x) = \{y\}$ such that x and y are infrequent and the *mediator support set* $MSS(x) = \{M\}$ such that x is dependent on M , based on the HI-Struct. After the two sets are computed for each item, the second phase of HI-Mine is to generate the set of mediators $\{M\}$ for each infrequent itempair $\{x, y\}$. This can be done by joining the $MSS(x)$ and $MSS(y)$. Then, all indirect association patterns can be discovered.

2.1.4 Concise Pattern

The number of frequent itemsets (**FI**) increases exponentially with the increase of frequent itemsets' length. So large length of frequent itemsets leads to no feasible of frequent itemset mining. However, since frequent itemsets are upward closed, it is sufficient to discover only **MFI** (maximal frequent itemsets) [14]. A frequent itemset is maximal if it has no frequent superset. Another recent promising direction is to mine only **FCI** (frequent closed itemsets) [109]. A frequent itemset is closed if it has no superset with the same frequency. It is straightforward to see that the following holds: $\text{MFI} \subseteq \text{FCI} \subseteq \text{FI}$. In practice, the set **MFI** is orders of magnitude smaller than the set **FCI**, which itself is orders of magnitude smaller than the set **FI**. There is much research on methods for maximal frequent itemset mining and closed frequent itemset mining. We now review some typical mining algorithms for the two patterns respectively.

2.1.4.1 Maximal Pattern

The set of **MFI** is the smallest possible representation of the data that can still be used to generate the set of **FI**. Once the set **FI** is generated, the support information can be easily recomputed from the transactional database. Also, there are applications where the set of maximal patterns is adequate, such as combinatorial pattern discovery in biological applications [117]. Therefore, maximal frequent itemset discovering has become a fundamental and important problem in association rule mining.

Definition 2.8 (Maximal Frequent Itemset) *Given a database D and a support threshold δ ($0 \leq \delta \leq 1$), an itemset X is defined as a maximal frequent itemset if 1) $\text{supp}(X) \geq \delta$; 2) $\forall Y \supset X, \text{supp}(Y) < \delta$.* $\square$

Most of the existing algorithms of maximal frequent itemset mining are based on two pruning techniques, which are stated as follows.

Lemma 2.1 (Subset Infrequency) *A proper superset of any infrequent itemset is not a frequent itemset.*

Lemma 2.2 (Superset Frequency) *A proper subset of any frequent itemset is not a maximal frequent itemset.*

The MaxMiner [14] employs a breadth-first traversal policy for the searching. To reduce the search space, it performs not only subset infrequency pruning to skip over the itemsets that have any infrequent subsets, but also superset frequency pruning (also called *lookahead* pruning). To increase the effectiveness of superset frequency pruning, MaxMiner dynamically reorders the child nodes. The strategy was used by all the other MFI mining algorithms after it [4,6,7,9,10]. Normally depth-first approaches have better performance by the lookahead strategy, but MaxMiner uses a breadthfirst approach instead to limit the number of passes over the database.

DepthProject [4] performs a mixed depth-first traversal and does the subset infrequency pruning and a variation of superset frequency pruning to the set enumeration tree. Also it uses an improved counting method based on transaction projections along its branches. The original database and the projections are represented as a bitmap. The experiment results in [4] show that DepthProject outperforms MaxMiner by more than one time.

Mafia [24] is another depth-first algorithm. Mafia also uses a vector bitmap representation, where the count of an itemset is based on the column in the bitmap. Besides the two pruning methods mentioned above, another novel pruning technique called PEP (Parent Equivalence Pruning) is also used. The PEP strategy states that if the set of transactions supporting itemset X equals to the set of transactions supporting item y , which belongs to the tail of X in the set enumeration tree, then it is not necessary to consider further itemsets containing X but y . Hence, y can be moved from the tail to the head of node X . The experiments in [24] show that PEP prunes the search space greatly.

Both DepthProject and Mafia mine a superset of the MFIs, and require a postpruning to eliminate non-maximal frequent itemsets. GenMax [66] integrates the pruning with the mining and finds the exact MFIs by using two strategies. First, just like transaction database is projected on current itemset, the discovered MFI set can also be projected on the local MFI, which yields fast superset checking. Second, GenMax uses *diffset* [163] propagation to do fast frequency computation.

FPM_{ax}* [67] is an extension of the FP-Growth method, for MFIs mining only. It uses a FP-Tree to store the transaction projection of the original database for each node in the tree. In order to test whether a frequent itemset is the subset of any discovered MFI with the lookahead pruning strategy, another tree structure (MFI-Tree) is utilized to keep the track of all discovered MFI, which makes effective superset checking. FPM_{ax}* uses an array for each node to store the counts of all 2-itemsets that are subsets of the frequent itemsets, which allows the algorithm to scan each FP-Tree only once for each recursive call emanating from it. The experiment results [67] show that FPM_{ax}* has the best performance compared with GenMax and Mafia.

2.1.4.2 Closed Pattern

As a smaller alternative to FI, the set of FCI retain complete information, i.e., from this concise set, it is straightforward to generate all the frequent itemsets with accurate support counts. In the last several years, extensive studies have proposed fast algorithms for mining frequent closed itemsets.

Definition 2.9 (Frequent Closed Itemset) *Given a database D and a support threshold δ ($0 \leq \delta \leq 1$), an itemset X is defined as a frequent closed itemset if 1) $\text{supp}(X) \geq \delta$; 2) $\nexists X' \supset X, \text{supp}(X') = \text{supp}(X)$.* $\square$

The first algorithm proposed for mining closed itemsets was A-CLOSE [109]. A-CLOSE first browses level-wise the frequent itemsets lattice by means of an Apriori-like strategy, and mines all the *minimal itemsets* of each *equivalence class*. Two itemsets belong to the same equivalence class if they are supported by the same set of transactions. A k -itemset is a minimal itemset if and only if no one of its $(k - 1)$ -subsets has the same support. Obviously, minimal elements are discovered with an intensive subset checking. In its second step, A-CLOSE calculates the closure of all the minimal itemsets previously found. Since a single equivalence class may have more than one minimal itemsets, redundant closures may be computed. A-CLOSE performance suffers from the high cost of the off-line closure calculation and the huge number of subset searches.

The authors of FP-Growth [71] proposed CLOSET [110] and CLOSET+ [141]. These two algorithms inherit from FP-Growth the compact FP-Tree data structure and the exploration technique based on recursive conditional projections of the FP-Tree. With a depth

first browsing of the FP-Tree and recursive conditional FP-Tree projections, CLOSET mines closed itemsets by *closure climbing*, which uses supersets of the discovered closed itemset as new closure generators and extends discovered closed itemsets with items having the same support in the conditional dataset. All closed sets previously discovered are kept in the main memory, indexed by a two level hash for duplicate closed itemset checking. CLOSET+ is similar to CLOSET. But it exploits an adaptive behavior in order to fit both sparse and dense datasets.

CHARM [164] performs a bottom-up depth-first browsing of a prefix tree of frequent itemsets built incrementally. As soon as a frequent itemset is generated, its tid-list is compared with those of the other itemsets having the same parent. If one tid-list includes another one, the associated nodes are merged since both itemsets surely belong to the same equivalence class. Itemset tid-lists are stored in each node of the tree by using the *diffset* technique. Since different paths can however lead to the same closed itemset, a duplicates pruning strategy is implemented as well. CHARM adopts a technique similar to that of CLOSET, by storing in the main memory the closed itemsets indexed by single level hash.

2.2 Web Mining

The World Wide Web has grown explosively since it emerged in the early 1990s. Nowadays, as a global information system, the Web flooded us with a substantial amount of data and information. Thus, data mining was introduced into this fertile area to help find useful information. As we discussed in the previous section, data mining on the Web can be roughly classified into three categories: *Web content mining*, *Web structure mining* and *Web usage mining*. We review representative problems in each category in the following. We are particularly interested in association rule mining on tree structured Web data.

2.2.1 Web Content Mining

As stated in [84], Web content mining can be viewed from perspective of either *information retrieval* or *database*. From the information retrieval view, the Web content is usually modelled as *unstructured* data, i.e., bags of words. From the database view, the Web content is often represented as *semi-structured* data. Thus, we review problems of association rule mining on Web content data from the database view.

Figure 2.2: OEM graph.

The database view of Web content mining mainly tries to model the data on the Web and to integrate them so that more sophisticated queries other than the keywords based search could be performed. These could be achieved by finding the schema of Web contents, building a Web warehouse or a Web knowledge base or a virtual database. Association rule mining is frequently used in the first approach of finding frequent subschema in semi-structured data.

2.2.1.1 HTML Web Document

Many online documents are similarly structured, though not identically structured. The work done in [142] tried to discover “typical” structures of a collection of Web documents. They adopted the *Object Exchange Model* (OEM) to represent semi-structured documents. Then they defined a *tree-expression* to describe objects in OEM graph. For example, Figure 2.2 presents a cyclic graph about object $\&O$, which is transformed to be noncyclic by introducing a leaf node named by a special symbol $\perp_i$, $i > 0$. A leaf node name $\perp_i$ is the alias of the ancestor that is i nodes above it. Then, the *tree-expression* of $\&O$ is phrased as: $te = \{Award : \perp, Cast : \{Actor : \{film : \perp_3\}\}\}$. A *tree-expression* is **weaker** than another *tree-expression* if it is less *informative* [142]. Then the problem of “typical” structure discovery can be stated as follows:

Definition 2.10 (Typical Structure Discovery) *Given a collection of transaction objects in an OEM graph and a support threshold δ ($0 \leq \delta \leq 1$). The support of a tree-expression te , $supp(te)$, is the percentage of transaction objects $\&t$ such that te is weaker*

than $\&t$. te is **frequent** if $\text{supp}(te) \geq \delta$. The typical structure discovery problem is to find all frequent tree-expressions. $\square$

In order to discover frequent *tree-expressions*, *tree-expressions* are represented by a sequence of *label paths*. A k -*tree-expression* represents a tree with k leaf nodes and can be denoted by a k -sequence *label path*. Each *label path* has the form of $[\top, \ell_1, \dots, \ell_n, \perp]$ or $[\top, \ell_1, \dots, \ell_n, \perp_i]$, where symbol $\top$ stands for the transaction object, ℓ_i are the labels on the path from the transaction object to one of its leaf node, and $\perp$ or $\perp_i$ stands for the end of the path. For the *tree-expression* shown in above example, it can be represented by 2-sequence of (p_1, p_2) , where $p_1 = [\top, \text{Award}, \perp]$ and $p_2 = [\top, \text{Cast}, \text{Actor}, \text{Film}, \perp_3]$.

The discovery algorithm contains two phases. In the first phase, all transaction objects are traversed and frequent 1-*tree-expressions* are found. In other words, all frequent individual *label paths* are discovered. In the second phase, every pair of frequent $(k-1)$ -sequences, $(p_1, \dots, p_{k-2}, p_{k-1})$ and $(p_1, \dots, p_{k-1}, p_k)$, will be merged to generate the candidate k -*tree-expressions* $(p_1, \dots, p_{k-2}, p_{k-1}, p_k)$. Transaction objects are traversed for the k th pass and for each transaction object. If the candidate k -*tree-expression* is weaker than the *tree-expression* of the object, the count of the candidate is incremented. If the support of the k -*tree-expression* is no less than user-defined threshold, then it is frequent. The process iterates until there's no candidate *tree-expression* generated. The generation of candidate *tree-expression* is based on the property that if one k -*tree-expression* is frequent, any of its $(k-1)$ -*tree-expression* is frequent as well. To facilitate the counting of support, candidate k -*tree-expressions* are stored in a structure of *trie*. Some optimization strategies are proposed by pruning the search space.

2.2.1.2 XML Web Document

Recently, XML emerges as a new standard for data representation and exchange over the Web. XML documents consist of two parts of data: XML structure and XML content. The former refers to the hierarchical structured formed by the embedding relationship between XML elements. The latter means the XML element values. There exist both research work on XML structure mining and work on XML content mining.

The main objective of XML structure mining is to find frequent substructures in XML documents as XML are semi-structured data as well. TreeFinder[135] is one of such algorithms. Their work is distinct from other work of frequent substructure mining in that they

CHAPTER 2. LITERATURE SURVEY

tried to find trees which are *approximately* embedded in several input trees rather than *exactly* embedded in input trees. Thus, when XML documents vary in the nesting relationship of labels, discovered frequent trees are robust enough. To this end, they first define a subsumption relationship between two trees to represent that a tree is approximately embedded in the other. Each labelled tree is described in *relaxed relational representation*. For example, given a labelled tree t , the relaxed relational description $Rel^+(t)$ is the conjunction of all atoms $a * b(u, v)$, where u and v are two nodes in t , a and b are labels of u and v respectively. Additionally u is an ancestor node of v . Then each input tree can be viewed as a transaction in traditional association mining and each atom of a relational description can be regarded as an item. TreeFinder contains two steps to find frequent trees. The first step is to cluster input trees in which same atoms of relational description occur together frequently enough. The second step is to find the maximal common trees in each cluster. This is done by using a proposed algorithm of *least general generalization* to generalize each relational description of trees in a cluster. After finding the least general generalizers of relational descriptions for trees in a cluster, it is encoded into a tree, i.e., reconstructing parent-child relation from the ancestor-descendant relation which is stated in least generalized relational descriptions. The advantage of the algorithm is that by representing labelled tree in relational descriptions, finding trees in subsumption relation is NP complete, which can be implemented efficiently. The disadvantage is that TreeFinder is not complete.

Meisels et al. [101] also mined frequent substructures from XML documents. The main feature which differentiates it from previous work is that frequent substructures are composed by paths which are not only frequent but also interesting. A path is frequent if the ratio of the occurrence of the last node on the path to the occurrence of its parent node is greater than some support threshold. A path is interesting if the attribute value of the last node on the path has a frequency which exceeds some interesting threshold. A path containing any uninteresting node will be pruned except the node has any interesting descendants. By this way, discovered structural association is supposed to be more meaningful because both the frequency of XML tags and the content of tags, which is in the form of attribute values of tags, are considered.

Maruyama and Uehara [99] aimed to mine association rules from the XML content. They first borrowed the algorithm proposed by Wang and Liu [142] to find frequent substructures from XML documents first. Discovered frequent structures will be regarded as

schema patterns. Association rules of data values will be mined based on the schema. For example, if a discovered schema contains attributes like “journal”, “keyword”, etc., then a probable rule may be $\{journal = “IEEE”, year = “1986”\} \Rightarrow \{keywords = “geometricmodeling”, keywords = “vision”\}$. To find more interesting and informative rules, concept hierarchies are considered to be constructed on the data values. They adopted an Apriori-like mining algorithm.

2.2.2 Web Structure Mining

Web content mining from database view is focused on mining structure within each Web document, which is called intra-document structure. There is a contrary inter-document structure mining which pays attention to the hyperlinks among Web documents. Typical algorithms about this type of Web structure mining include PageRank [21] and HITS [83]. These algorithms aim to enhance the performance of search engines by calculating the quality rank or relevancy of each Web page. Although there’s some implication between the weights of pages and the given topics, this problem has no similar form of classical association rules.

The work done by Madria et al. [96] also mines the structure of links among Web documents. An example of discovered association rules is that 10% of links starting from *www.a.com* terminate at *www.b.com*. The problem and algorithm are defined under a specific Web warehouse environment, WHODWEDA, where Web data are organized as *Web tuples* in *Web tables*. Each Web tuple contains two kinds of Web objects: nodes (Web pages) and edges (hyperlinks). The schema of a Web table represents the query graph provided by user. There are two types of association rules that can be mined from Web tables: association rule of nodes and association rules of links. The discovered node association rules imply how data in Web tuples is related, and discovered link association rules reflect the luminosity of a Web page.

2.2.3 Web Usage Mining

Web usage information captures activities from online users and exhibits a wide range of different behavioral patterns. The behavioral patterns can shed light on user browsing preferences, online customer behavior and future Web site improvement directions. Association rule mining can be used in Web usage mining to relate pages that are most often

referenced together in a single Web user session. An example of such association rules may reveal a correlation between a page containing electronic products to a page about sporting equipment. Aside from being applicable for business and marketing applications, the presence or absence of such rules can help Web designers to restructure their Web site. The association rules may also serve as a heuristic for prefetching documents in order to reduce user-perceived latency when loading a page from a remote site.

Chen et al. [39] assumed that a backward reference is made for changing a traversal path, which can not be considered as user access patterns. Hence, they proposed to transform the original access data into a set of *maximal forward references*. A maximal forward reference is the sequence of pages requested by a user up to the last page before backtracking occurs during a particular server session. After preprocessing the data, they mined frequent traversal patterns from the set of maximal forward references. Frequent traversal patterns are those forward references with their occurring times greater than some support threshold. A similar technique as DHP[108] is applied to generate candidate frequent traversal pattern. Thus, web usage data are represented as sequential data.

Recently, there are some proposals on modelling the Web usage data as tree structures by looking at the entire forward accesses of users [162, 153]. After representing the Web usage data as tree structures, the problem of frequent pattern mining is actually discovering frequent subtrees. Then, algorithms for frequent subtree mining, which we reviewed previously, can be employed directly. We omit the discussion of these algorithms here.

Unlike the sequential data model and the tree structured data model, Borges and Levene [18] organized Web usage data into a weighted directed graph. Each node in the graph represents a Web page and each arc represents a link between pages. The weight of an arc corresponds to the times it is traversed. A simple association rule can be mined from two adjacent nodes A and B , which can be derived into a composite association rule considering a path of more than one link. For example, a composite association rule may have the statement as *when a user goes through the links from A to B to C to D , he will go to node E with some probability*. Two algorithms are proposed to mine composite association rules. One is to traverse each link in the graph and form a depth-first search tree. Each branch of the tree is a candidate rule. The extension of a branch terminates when either the confidence of the rule lower than minimum confidence threshold or it reaches a node whose links have been visited already. The other algorithm is based on the property that if

a sequence of links has its confidence greater than some threshold, subsequences of the links will also have their confidence greater than the threshold. Hence, a candidate sequence of k links can be formed by combining two sequences of $(k - 1)$ links.

2.3 Web Change Detection

Data on the Web may change at any time and in any way. The rapid and often unpredictable changes to the information on the Web result in the important research problem of Web change detection. Basically, changes to the Web can be categorized into four classes, *content changes*, *presentation changes*, *structural changes* and *behavioral changes*, which are explained as follows.

- *Content changes* refer to modification of the page contents from the reader's point of view. For example, a page created for a tennis tournament might be continuously updated as the tournament progresses. After the tournament has ended, the page might change to a presentation about the tournament results.
- *Presentation changes* are changes related to the document representation that do not reflect changes in the topic presented in the document. For instance, changes to HTML tags can modify the appearance of a Web page while it otherwise remains the same.
- *Structural changes* refer to the underlying connection of the document to other documents. As an example, consider changes in the link destinations of a "Weekly Hot Links" Web pages. While this page might be conceptually the same, the fact that the destination of the links have changed might be relevant, even if the text of the links has not. Structural changes are also important to detect, as they often might not be visually perceptible.
- *Behavioral changes* refer to modifications to the active components of a document. For Web pages this includes scripts, plug-ins and applets.

The behavioral changes are hard to detect or predict since many pages hide the script code in other files. Hence, most of existing algorithms of Web change detection focus on the first three types of changes. For example, WebGUIDE (Web Graphical User Interface to a

Difference Engine) [54] is a tool that computes not only the differences between the pages with respect to two dates but also the differences in link structures which are shown via graphical representations. As we are interested in tree-structured Web data, we review some related works of detecting changes to Web documents in the following.

2.3.1 Detecting Changes to HTML Documents

The AT&T Internet Difference Engine (AIDE) [55] is a system that finds and displays changes to pages on the Web. The system consists of several components, including a web crawler that detects changes, an archive of past versions of pages, a tool called *HtmlDiff* to highlight changes between versions of a page, and a graphical interface to view the relationship between pages over time. *HtmlDiff* takes two versions of a page as input and produces a new syntactically correct new HTML page that highlights the differences between the two pages. In *HtmlDiff*, an HTML document is viewed as a sequence of sentences and “sentence-breaking” markups (such as $\langle p \rangle$, $\langle HR \rangle$, $\langle LI \rangle$) where a sentence is a sequence of works and certain (non-sentence-breaking) markups (such as $\langle B \rangle$ or $\langle A \rangle$). Then, Hirschberg’s solution [72] to the longest common subsequence (LCS) problem is applied to compare HTML documents. As an efficient implement of *HtmlDiff* in java, TopBlend [40] used a more recent comparison algorithm for the heaviest common subsequence problem. The experimental results showed that TopBlend significantly outperforms the old *HtmlDiff* in the most time consuming jobs.

WebCQ [94] is a prototype system for Web information monitoring and delivery. It provides a personalized notification of what and how Web pages of interest have been changed, and personalized summarization of Web page changes. WebCQ contains a change detection robot which first locates and extracts the objects being monitored from the page. In order to generate the difference from two versions of a Web page, WebCQ used a simplified difference generation algorithm which reduce the overhead of the general *HtmlDiff*. That is, WebCQ tracks changes to a page fragment rather than the entire page. A page fragment can be a specific HTML object (such as phrase, list, and table), an arbitrary text fragment, or a specific component of the page (such as links, images, and words).

HTML documents are usually modelled as semistructured data. In [30], a *snapshot-delta* approach was used for representing changes in semistructured data. The authors present a simple and general model called *DOEM* for representing changes and also present

a language called *Chorel* for querying changes represented in *DOEM*. This model is founded on the OEM data model and the Lorel query language [3]. It uses *annotations* on the nodes and arcs of an OEM graph to represent changes. Intuitively, the set of annotations on a node or arc represents the history of that node or arc. An important feature of this approach is that it represents and queries changes directly as annotation on the affected area instead of representing them indirectly as the difference between database states. Furthermore, they describe the design and implementation of an application of change management called a *Query subscription Service* (QSS). QSS can be used to notify subscribers of relevant changes in semistructured information sources.

Although HTML documents can be represented as hierarchical data, in [93], the authors pointed out that existing change-detection algorithms for hierarchical data are not suitable in dealing with HTML documents, since the hierarchy of markups and data contents in an HTML document is often meaningless or not informative. This is because HTML was designed to “display” data, not to “define” data. Hence, they proposed the algorithm called SCD to detect semantic changes in HTML documents. The novelty of SCD is the adopting the notion of *semantic hierarchy* instead of the hierarchy of HTML markups. Basically, SCD consists of three steps: *i*) construction of the semantic hierarchies of any two given HTML documents; *ii*) identification of branches in the resulting trees and removal of branches that appear in both; and *iii*) determination of changes from one tree to the other by comparing each remaining branch in one tree to every remaining branch in the other.

The CDWeb system [60] also represents the HTML document as a tree. Instead of looking for the exact sequence of changes that permits the new document to be produced from the old version, CDWeb pays attention to how much the changes have modified the document under observation. CDWeb permits the users to focus on specific regions of the document tree, i.e., subtrees, and the items inside the region, i.e., subtrees of the previously selected subtree, and the conditions on the type of changes which must be detected. Other features of CDWeb include allowing users to specify the type of changes they consider relevant and measure the differences in a quantitative way.

2.3.2 Detecting Changes to XML Documents

Recently, the XML research community has recognized the importance of change management problem. An XML document can be modelled as either an *ordered tree* or an *unordered*

tree. In an ordered XML tree, both the parent-child relationship and the left-to-right order among the sibling are important. In an unordered XML tree, the parent-child relationship is significant, while the left-to-right order among the siblings is not important. By adopting different XML tree models, change-detection algorithms for XML documents are different as well. We review some typical algorithms in each category as follows.

2.3.2.1 Ordered XML Documents

Chawathe et. al [32] formulated the change detection problem on hierarchically structured documents, and proposed an efficient algorithm, *LaDiff*, based on the following key assumption: Given two labeled trees, T_1 and T_2 , there is a “good” matching function *compare*, so that given any leaf s in T_1 , there is at most one leaf in T_2 that is “close” enough to match s . *LaDiff* runs in time $O(ne + e^2)$, where n is the number of tree leaves and e is “weighted edit distance” (typically, $e \ll n$). However, as observed in [147], the assumption holds well for many SGML documents that do not contain duplicate or similar objects, but it does not hold for many XML documents.

Chawathe et. al [31] also presented a heuristic algorithm, *MH-Diff*, to detect change in unordered structured documents. *MH-Diff* is based on representing an edit script between two trees as an edge cover of a bipartite graph. They introduced new edit operators such as “subtree copy” and “subtree glue” in the algorithm. However, the worst case of algorithm is in $O(n^3)$.

XMLTreeDiff [48] computes the difference between two XML documents. First, it computes hash values for the nodes of both documents using DOMHash [98] and reduces the size of the two trees by removing identical subtrees (i.e., ones with identical hash values). Second, it uses Zhang and Shasha’s algorithm [165] to generate the difference between the two simplified trees. While using DOMHash to filter out identical subtrees can reduce the size of the two trees, its use conflicts with the cost model employed by Zhang and Shasha’s algorithm. Thus, *XMLTreeDiff* may not generate an optimal result.

Recently, Cobena et. al [43] proposed *XyDiff*, an algorithm for detecting changes in XML documents. The algorithm first computes a signature (i.e., hash value) and a weight (i.e., subtree size) for every node in both documents in a bottom-up fashion (the root nodes of the two documents end up with the largest weights). Next, starting with the root nodes of the two documents, *XyDiff* compares the signatures of the two nodes. If they are equal,

the two nodes are matched; otherwise, their child nodes will be inserted into a priority queue in which the subtrees with the largest weights are always compared first. Whenever *XyDiff* finds an exact match between two subtrees, it attempts to propagate the match to the respective parents of the two nodes with the weight of each subtree determining how many levels the matching is propagated. Whenever there is more than one potential candidate for matching, *XyDiff* uses a few simple heuristic rules to pick one in order to avoid having to perform a full evaluation of the alternatives. This algorithm achieves $O(n \log n)$ complexity in execution time and generates fairly good results in many cases. However, *XyDiff* cannot guarantee any form of optimal or near-optimal result because of the greedy rules used in the algorithm.

Both *XMLTreeDiff* and *XyDiff* are main-memory algorithms for detecting changes in ordered XML documents. Recently, Leonardi et al [90] proposed to detect changes to XML documents by using relational database. By storing XML documents in relational database and detecting changes with SQL queries, their approach gave the opportunities to have more scalable change detection system. However, their approach detects only the changes to leaf nodes of XML tree while state-of-art approaches detect changes to both leaf nodes as well as internal nodes.

2.3.2.2 Unordered XML Documents

It is argued in [147] that an unordered tree model is more suitable for most XML based applications. However, using an unordered model, change detection is substantially harder than using the ordered model. Change detection on unordered trees has been shown to be NP-Complete in general case.

In [147], the authors presented *X-Diff*, an algorithm for detecting changes on unordered XML documents. *X-Diff* assigns XHash values for each node in the XML tree. The XHash value of a node is calculated from the XHash values of its descendants. The algorithm tries to find a minimum-cost matching between two documents before generating the edit script in order to find the minimum edit script. *X-Diff* achieves polynomial time in complexity.

Since *X-Diff* only reduce the the size of tree by removing equivalent second-level subtrees, the filtering is not efficient. Besides, in some cases, the results generated by *X-Diff* may not be satisfactory to users. In [154], a highly efficient algorithm named *KF-Diff+*

		Data Structure	Data Temporal Feature	Association Nature	Knowledge Type
Frequent Pattern	Frequent Itemset	itemset	static	positive	complete
	Frequent Sequence	sequence			
	Frequent Structure	tree/graph			
Infrequent Pattern	Negative Association	itemset		negative	
	Indirect Association	itemset			
Concise Pattern	Maximal Pattern	itemset, sequence, tree, graph		positive	concise
	Closed Pattern				
Structural Evolution Pattern		tree	evolutionary	positive/negative	complete/concise

Figure 2.3: Comparison of existing association rule mining and our work.

was proposed. The key property of the *KF-Diff+* is that the algorithm transforms the traditional tree-to-tree correction into the comparing of the key trees which are substantially labelled trees without duplicate paths. Thus, *KF-Diff+* achieves the high efficiency with the complexity of $O(n)$, where n is the number of nodes in the trees.

Motivated by the same reason that main memory based change-detection algorithm may not be scalable well to handle large XML documents, Leonardi et al [91] also proposed their version of secondary memory based algorithm for detecting changes on unordered XML documents. In order to store XML documents in relational database, a relational schema which supports efficient insertion and extraction of XML documents and efficient execution of path expression queries is necessary. Thus, the author adopted the SUCXENT [112] relational schema as it outperforms significantly the current state-of-the-art model mapping approaches. The first phase of their algorithm is to find a best matching of subtrees. Then, changes are detected in the order of inserted and deleted internal nodes, inserted and deleted leaf nodes and finally updated leaf nodes. Their experimental results show that the relational approach has better scalability compared to the algorithm *X-Diff*. The result quality is also comparable to the *X-Diff* algorithm.

2.4 Summary

In this chapter, we surveyed three related research fields: *association rule mining* and *Web mining* and *Web change detection*.

In association rule mining, we first present one set of basic definitions which include items, itemsets, databases, support and confidence. They serve as a background to the

CHAPTER 2. LITERATURE SURVEY

	Data Structure	Data Temporal Feature	Association Nature	Knowledge Type
Web Content AR Mining	bags of words, tree	static	positive	frequent pattern
Web Structure AR Mining	graph			
Web Usage AR Mining	sequence, tree, graph			
Structural Evolution Pattern	tree	evolutionary	positive/negative	evolution association

Figure 2.4: Comparison of existing Web association rule mining and our work.

dissertation. Two representative approaches to association rule mining: Apriori and FP-Growth, have been described. In order to show the diversity of frequent patterns, we have also reviewed patterns of other forms, such as frequent sequences and structures, infrequent patterns, maximal and closed patterns. We compare our work and existing association rule mining from the following four aspects: *data structure*, *data temporal feature*, *association nature* and *knowledge type*. The differences have been summarized in the Figure 2.3. Existing association rule mining addresses different data structures such as itemsets, sequences, tree and graph structures. In our research, we focus on tree structured data. The significant difference between our work and existing association rule mining is that we consider evolutionary data while existing work considers static data only. Existing association rule mining can be divided into works mining either positive association or negative association. In our associative evolutionary structures mining, we consider both types of associations. Finally, existing association rule mining work discovers either the complete set of patterns or the concise pattern set from which complete patterns can be inferred. In our research, we consider the two types of knowledge as well.

In the field of Web mining, we classified the works into categories of Web content mining, Web usage mining and Web structure mining. Since Web structure is usually represented as a graph, we focus on the first two categories. In each category, we describe the typical tasks of association rule mining on this type of Web data. We compare our work with existing Web association rule mining from the four aspects as above. The differences are summarized in Figure 2.4. Existing Web association rule mining models different data as different data structure. For example, the Web content is modelled as bags of words from the information retrieval view and semi-structured data from the database view. In our research, we consider different Web data which can be modelled as tree structures. Existing Web association rule mining usually mines knowledge from the snapshot of Web data even though Web data is evolutionary. However, in our research, we address the evolution nature

CHAPTER 2. LITERATURE SURVEY

of Web data. Moreover, existing Web association rule mining focus on discovering positive associations, while we consider both positive and negative associations. Finally, the main objective of existing Web association rule mining is to discover frequent patterns. Whereas, we aim to discover evolution association between structures of Web data.

In Web change detection, we mainly discuss the works on detecting changes to Web contents, HTML documents and XML documents, which are usually modelled as tree structures. Algorithms for detecting changes to both ordered tree structures and unordered tree structures are reviewed. In our research, we mainly consider Web data which can be modelled as unordered tree structures. Furthermore, we consider basic edit operations (as shown in Section 4.1.2 of Chapter 4) defined in the literature. Hence, we borrow existing change detection algorithms based on unordered tree structures and focus on other modules of our associative evolutionary structure mining framework.

Chapter 3

Problem Definition

In this chapter, we formalize the research problem of this dissertation: *associative evolutionary structure mining* on changes to tree structured Web data. Given historical tree structured Web data, different types of associative evolutionary structures can be defined by considering changes from different aspects, such as *intra-tree evolution v.s. inter-tree evolution* and *structural variation v.s. semantic variation*. Particularly, we propose three types of associative evolutionary structures: *FRACTURE*, *negative FRACTURE* and *SEASON*. The first two patterns capture evolution associations in intra-tree by viewing changes from the structural variation perspective, while the third one captures evolution associations in inter-tree and considers changes from the semantic variation view.

In the remainder of this chapter, we begin by presenting the problem model of associative evolutionary structure mining. Examples of specific associative evolutionary structures investigated in this dissertation, *FRACTURE*, *negative FRACTURE* and *SEASON*, are described in Section 3.2. The main challenges and research issues are discussed in Section 3.3. We conclude this chapter with a brief review of related work and a summary.

3.1 Problem Model

The Web is evolutionary in nature. However, as discussed in Chapter 1, existing Web mining approaches usually mine knowledge from snapshots of Web data. While knowledge extracted from snapshots of Web data is important and useful, interesting knowledge might be buried in changes to historical Web data. In this dissertation, we focus our study on mining frequent patterns, called *associative evolutionary structures*, from changes to Web data.

CHAPTER 3. PROBLEM DEFINITION

Figure 3.1: Evolutionary subtree database model.

We focus on evolutionary tree-structured Web data. Particularly, we consider two types of database as the mining context of *associative evolutionary structures*. One is an *evolutionary subtree database* which is constructed from a sequence of historical tree versions. The other is an *evolutionary tree database* which is defined on a set of tree sequences. For the former, we focus on studying the evolution association between subtrees, which is called *intra-tree evolution association*. For the latter, we are interested in the evolution association between trees, which is called *inter-tree evolution association*. The two types of databases are respectively defined in the below.¹

Given a sequence of historical versions of a tree, representing some particular Web data, an evolutionary subtree database can be defined by viewing each tree version as a set of subtrees.

Definition 3.1 (Evolutionary Subtree Database) Let S be a tree structure, $T = \langle t_1, t_2, \dots, t_n \rangle$ be a sequence of time points with some particular time granularity, S^{t_i} be the version of S at time point t_i , and s_i be a subtree of S , denoted as $s_i \prec S$. Given a sequence of versions of S on T , $\Sigma = \langle S^{t_1}, S^{t_2}, \dots, S^{t_n} \rangle$, it can be represented as a sequence of subtree sets $\Omega = \langle V^{t_1}, V^{t_2}, \dots, V^{t_n} \rangle$, where $V^{t_j} = \{s_i | s_i \prec S^{t_j}\}$. Then, an **evolutionary subtree database** $\mathcal{D}$ is a database in the form of (t_i, V^{t_i}) , where $t_i \in T$ and $V^{t_i} \in \Omega$. $\square$

For example, Figure 3.1 models an evolutionary subtree database, where each triangle denotes a tree version and the horizontal axis represents the ordinal time property of the database. An evolutionary subtree database considers each tree version in terms of a set of subtrees, which are denoted by inside triangles filled with patterns in Figure 3.1. (Note that,

¹The two types of databases are defined based on the assumption that historical versions of tree-structured Web data are obtained appropriately so that changes to tree structures can be detected.

CHAPTER 3. PROBLEM DEFINITION

Figure 3.2: Evolutionary tree database model.

without loss of generality, we did not define the type of subtrees, e.g., embedded subtrees or induced subtrees. We specify the type of subtrees when defining specific associative evolutionary structures.) Subtrees evolve over time. For example, in Figure 3.1, the variations of triangles' size denote the structural changes to subtrees. Associative evolutionary structures mined from an evolutionary subtree database aim to discover evolution associations between subtrees (e.g., which subtrees frequently change together).

Given a set of historical tree sequences, an evolutionary tree database can be defined as follows.

Definition 3.2 (Evolutionary Tree Database) Let S be a tree structure, $\Gamma = \{S_1, S_2, \dots, S_n\}$ be a set of distinct trees. A tree sequence is an ordered list of trees, denoted as $Q = \langle S_1 S_2 \dots S_m \rangle$. An **evolutionary tree database** $\tilde{D}$ is a set of tree sequences in the form of (Qid, Q) , where Qid is a tree sequence identifier, and $Q = \langle S_1 S_2 \dots S_m \rangle$ s.t. $S_i \in \Gamma$ ($1 \leq i \leq m$). $\square$

For example, Figure 3.2 models an evolutionary tree database. Similarly, we use triangles to represent tree structures and horizontal axes to represent the ordinal time property. However, instead of considering the changes to subtrees in a sequence of tree versions, we focus on the transitions between tree structures. Hence, as shown in Figure 3.2, each triangle with fixed size represents a particular tree with fixed structures. Thus, an evolutionary tree database is a sequential database of tree structures. Associative evolutionary structures

CHAPTER 3. PROBLEM DEFINITION

Figure 3.3: Hierarchy of associative evolutionary structures mining problem.

mined from an evolutionary tree database aim to discover evolution associations between trees (e.g., a tree frequently evolves into another tree).

Then, generally, the problem of associative evolutionary structures mining can be stated as follows.

Definition 3.3 (Associative Evolutionary Structures Mining) Let $\mathcal{D}$ be an evolutionary subtree database on a set of subtrees $\Omega = \{s_1, s_2, \dots, s_m\}$ and $\tilde{\mathcal{D}}$ be an evolutionary tree database on a set of trees $\Gamma = \{S_1, S_2, \dots, S_n\}$. Let $\mathbb{R}$ and $\mathbb{S}$ be two sets of evolution measures on subtrees and trees respectively. The **problem of intra-tree associative evolutionary structures mining** is to find a set of patterns such that each pattern is a set of **subtrees** and the evolution measures of the pattern satisfy some constraints λ , $\{P = \{s_1 \dots s_x\} | P \subseteq \Omega, \mathbb{R}_{\mathcal{D}}(P) \models \lambda\}$. The **problem of associative evolutionary inter-tree structures mining** is to find a set of patterns such that each pattern is a sequence of **trees** and evolution measures of the pattern satisfy some constraints ψ , $\{Q = \langle S_1 \dots S_y \rangle | Q \subseteq \Gamma, \mathbb{S}_{\tilde{\mathcal{D}}}(Q) \models \psi\}$. The **problem of associative evolutionary structures mining** is to find both intra-tree associative evolutionary structures and inter-tree associative evolutionary structures. $\square$

Obviously, different types of specific associative evolutionary structures can be defined based on different evolution measures. In this dissertation, we particularly define the following three types of associative evolutionary structures: *FRACTURE*, *negative FRACTURE* and *SEASON*. As shown in Figure 3.3, the first two patterns belong to the intra-tree associative evolutionary structures and the third one belongs to the inter-tree associative evolutionary structures. Mined from an evolutionary subtree database, a *FRACTURE*

Figure 3.4: Directory hierarchy.

pattern is defined as a set of subtrees that frequently change together. On the contrary, a *negative FRACTURE* is defined as a set of subtrees that are negatively correlated in undergoing changes. *SEASON* patterns are mined from an evolutionary tree database and each *SEASON* pattern is a frequent subsequence of trees (or tree clusters).

3.2 Examples of Associative Evolutionary Structures

We now present several examples to illustrate the *FRACTURE*, *negative FRACTURE* and *SEASON* patterns in respective mining context.

3.2.1 FRACTURE

As shown in Chapter 1, Web content includes both intra-document data and inter-document data. The inter-document data refer to the information between Web documents of a Web site, which can be modelled as a tree structure, called *directory hierarchy*, based on the URLs of Web documents. An example directory hierarchy constructed for documents in Figure 3.4 (a) is shown in Figure 3.4 (b).

When a Web site evolves, some new Web directories and documents are inserted while some obsolete ones are removed. Hence, the structure of a directory hierarchy changes along with the evolution of a Web site. For example, Figure 3.5 shows four historical versions of the above directory hierarchy at a sequence of continuous time points, t_1 , t_2 , t_3 , and t_4 , with some particular time granularity. Gray nodes in the figure depict the directories and documents that are deleted in the next version and black nodes depict the directories and

documents that are inserted in the current version².

Since *FRACTURE* belongs to intra-tree associative evolutionary structures, we are interested in changes to subtrees. Each tree version consists of a set of subtrees. In the literature, there are different notions of subtrees, such as embedded subtrees and induced subtrees. In our work, we consider *bottom-up subtrees* [139]. A tree x is a bottom-up subtree of another tree y if there exists a node in y such that it is the root of x , all its descendants in y exist in x as well. Then, a tree with n nodes has n bottom-up subtrees. For example, consider the first tree version in Figure 3.5. The two subtrees highlighted by dotted lines are two bottom-up subtrees of the first tree version.

Given a sequence of historical tree versions, we study the evolution of subtrees. A subtree is *changed* in two successive versions if at least one node is inserted to it or deleted from it. For example, the subtree w/p (we use the root path arriving at the root of a subtree to identify the subtree) in Figure 3.5 is changed from t_1 to t_2 because the node a was removed, so does the subtree w/t .

Changes to subtrees can be evaluated from different aspects. For example, we can measure how significantly a subtree changes in two versions, how frequently a subtree changes in the history, how frequently a set of subtrees undergo significant changes together etc. In order to capture interesting knowledge, we require that each subtree of a *FRACTURE* pattern changes **frequently** enough and undergoes **significant** changes frequently enough. Furthermore, all subtrees of a *FRACTURE* pattern should change together **frequently** enough and undergo **significant** changes together frequently enough (we will formally define the frequency and significance of changes in Chapter 4). That is, subtrees that rarely change together or rarely undergo significant changes together will not be extracted as a *FRACTURE* pattern. The formal definition of *FRACTURE* will also be presented in Chapter 4. We use the following example to give some ideas on *FRACTURE*.

Example 3.1 (FRACTURE) Given a sequence of 4 historical versions of a directory hierarchy as in Figure 3.5, we consider the two subtrees w/p and w/t . Since both subtrees changed in every subsequent version, they changed together frequently. We now observe how significantly the two subtrees changed each time. Without explaining our formal definition of the metric which measures the degree of changes to subtrees, we temporarily use the

²We do not consider the modification of documents here because we are interested in structural changes. Additionally, modification of a document can be regarded as deleting a document and inserting the document with new content.

CHAPTER 3. PROBLEM DEFINITION

Figure 3.5: Historical directory hierarchies I.

number of inserted/deleted nodes to evaluate the significance of changes. Suppose users are interested in changes involving no less than 2 inserted/deleted nodes. The subtree w/p changed significantly enough in the versions at time t_3 and t_4 . Similarly, the subtree w/t changed significantly in the two versions as well. Hence, the two subtrees underwent significant changes together in two subsequent versions. If users think it is frequent enough, the two subtrees will be discovered as a *FRACTURE* pattern. ■

The *FRACTURE* pattern discovered in Example 3.1 indicates that the directory *products* and the directory *training* frequently change together. The obtained knowledge can be used by Web masters to identify related Web directories, maintain Web sites and reengineer Web sites etc. For example, a Web site is likely created and maintained by different authors. Keeping the site consistent is the critical step toward providing a reliable Web site of high quality. When some directories are adjusted, with discovered *FRACTUREs*, the authors realize that some other directories probably need to be maintained as well. We discuss the potential applications of *FRACTURE* in Chapter 4 and present a real application of *FRACTURE* patterns in Web user clustering in Chapter 5.

3.2.2 Negative FRACTURE

While knowledge indicated by *FRACTURE* patterns, which discover the *positive evolution associations* of subtrees, is interesting and important, the knowledge about the *negative evolution association* of subtrees (e.g., a set of subtrees that are negatively correlated in undergoing changes.) may be useful also in some other applications. For example, consider the evolution of a Web directory hierarchy again. Suppose we aim to maintain an

CHAPTER 3. PROBLEM DEFINITION

Figure 3.6: Historical directory hierarchies II.

autonomous mirror site of an external Web site. Naïvely updating the whole directory tree of the Web site every time results in heavy network load. However, if the directory has *negative evolution associations* with some of its subdirectories, i.e., the directory and some of its subdirectories rarely change significantly together, we can skip updating the mirrors of these subdirectories to save some network load. Hence, we further propose the pattern *negative FRACTURE* to extract the negative evolution associations of subtrees.

Similar to *FRACTURE* mining, we study the evolution of the subtrees as well. We are interested in subtrees sets that are negatively correlated in undergoing significant changes. That is, when one set of subtrees undergo significant changes, the other set of subtrees usually do not change or change slightly. The following example shows what a *negative FRACTURE* looks like.

Example 3.2 (Negative FRACTURE) Suppose Figure 3.6 shows another sequence of 4 historical versions of a Web site directory hierarchy. We observe the evolution of the two subtrees w/p and $w/p/r$. Similar to the Example 3.1, we temporarily use the number of inserted/deleted nodes to evaluate the change degree of subtrees. Suppose users are interested in changes involving no less than 2 inserted/deleted nodes. Then, the subtree w/p changed significantly in every subsequent version. However, the subtree $w/p/r$ never changed significantly in the subsequent versions. Then, the correlation between the two subtrees in terms of undergoing significant changes is negative (according to some correlation measure we define formally in Chapter 6). Thus, the two subtree sets, $\langle \{w/p\}, \{w/p/r\} \rangle$, will be discovered as a *negative FRACTURE*. ■

Figure 3.7: XML query tree sequences.

Knowledge can be inferred from *negative FRACTURE* that when one subtree (e.g., w/p in Example 3.2) changes, the other subtree (e.g., $w/p/r$ in Example 3.2) rarely undergoes significant changes. When applying obtained knowledge in the application of Web site mirror maintenance, we can optimally skip updating the directory *previous* when maintaining the mirror of the directory *products*. Other potential applications of *negative FRACTURE* are described in Chapter 6.

3.2.3 SEASON

Besides *FRACTURE* and *negative FRACTURE*, we propose another type of associative evolutionary structure: *SEASON*. *SEASON* belongs to the inter-tree associative evolutionary structures. While we consider changes to trees from the *structural variation* perspective when mining *FRACTURES* and *negative FRACTURES*, we study the evolution of tree structures by viewing changes from the *semantic variation* perspective in *SEASON* mining. For example, given a set of tree sequences where each tree conveys certain meaning, we are interested in the frequent transitions between different meanings reflected by different trees (or tree clusters).

Particularly, we mine *SEASON* patterns from an evolutionary tree database where each tree structure models some Web object such that it is semantically meaningful³. As

³We say a tree is semantically meaningful if the semantic hierarchy of labels of nodes complies with the structural hierarchy of nodes. For example, a tree with a parent-child node pairs which are labelled as “wheel” and “car” is unmeaningful. While, it is meaningful if the pair of nodes are labelled as “car” and “wheel”.

discussed in Chapter 1, a tree modeling an XML query is semantically meaningful and it reflects the semantics of the query. Thus, we use sequential XML query database as an example. For instance, Figure 3.7 (a) shows a set of XML query sequences, where each query is expressed in XPath [46] language and modelled as a tree structure in Figure 3.7 (b). Each sequence records the sequential queries issued by some user. Obviously, regarding each query in Figure 3.7 (a) as its corresponding tree in Figure 3.7 (b), we get an evolutionary tree database. Then, *SEASONS* can be mined from the database as frequent subsequences of tree structures. Each frequent subsequence indicates that when a user inquires about some information, she/he will inquire about some other information subsequently. However, different users may inquire about similar information by issuing different XML queries. For example, both queries S_2 and S_4 in Figure 3.7 inquire about the information of *book author*. Hence, directly mining *SEASONS* as frequent subsequence of trees may not be able to discover all frequent transitions between users' information needs. We thus optimize the solution by clustering query trees with similar semantics first and then mine *SEASONS* as frequent subsequences of tree clusters. The following example shows how to find *SEASONS* from the evolutionary tree database in Figure 3.7.

Example 3.3 (SEASON) Given the database of XML query tree sequences in Figure 3.7, we categorize each query tree to certain cluster by grouping query trees representing similar semantics. For example, the query trees of S_1 and S_3 will be grouped as they similarly represent the information of *book title*. Suppose the clustering results of the five query trees in Figure 3.7 are shown in Figure 3.8 (a), where the cluster C_1 represents the information of *book title*, the cluster C_2 represents the information about *book author*, and the cluster C_3 represents the information about *book section*. Then, the database of XML query tree sequences can be transformed into a set of cluster sequences, which are shown in Figure 3.8 (b). *SEASONS* then can be discovered from the transformed database as frequent subsequences of tree clusters with traditional frequent sequential pattern mining techniques. For example, suppose the threshold of support be 0.4. Then, the sequence $\langle C_1, C_2 \rangle$, in Figure 3.8 (b), will be discovered as a *SEASON* pattern because its support is $2/5 \geq 0.4$. ■

SEASONS reflect the frequent evolutions between different semantics represented by different tree clusters. For example, the discovered *SEASON* pattern in Example 3.3 indicates that when a user inquires about the information of book title, she/he very likely issues

Figure 3.8: Historical XML query trees.

another query about the author of the book sequentially. The knowledge can be used in designing optimal XML query caching replacement strategies. For example, we can delay the eviction of answers to queries about book author, if they exist in cache already, once the queries about book title are issued. In Chapter 7, we will formally define the problem of *SEASON* pattern mining and present how to apply it in XML query caching.

3.3 Research Challenges

After giving examples of the three types of specific associative evolutionary structures which we will investigate in this dissertation, we now highlight the main research challenges in each particular pattern mining. Basically, we focus on the problems of how to define the interestingness of the patterns, how to discover the desired patterns and how the resulting patterns are useful in various Web applications. There are many research issues involved in these problems and we limit our study to the following research challenges. In Chapter 8, we present other issues that we would like to address in future work.

- ***FRACTURE* definition and discovery.**

As illustrated in Example 3.1, in order to identify interesting patterns, a set of subtrees should not only change together frequently but also change significantly together frequently to be a *FRACTURE* pattern. Hence, evolution metrics which measure the change frequency and change significance of subtrees respectively should be designed. Nevertheless, with such particular constraints, the *apriori* property [7] does not hold for *FRACTUREs*, i.e., a proper subset of a known *FRACTURE* is not necessarily a *FRACTURE* pattern. This property partially accounts for the difficulty of the problem of mining all *FRACTUREs*. Then, the development of efficient data mining

algorithms may be achieved by considering the following two problems. First, is there any properties, other than the apriori property, of *FRACTUREs*? If yes, how to integrate the properties effectively in the mining process? Second, is there any concise representations, such as maximal patterns [66], of *FRACTUREs* such that the concise patterns can be discovered efficiently?

- ***FRACTURE* application.**

As exemplified in Chapter 1, *FRACTURE* patterns can be useful in clustering Web users exhibiting similar variations in their information needs. The basic idea is mining *FRACTUREs* from historical Web sessions for users and clustering users sharing similar *FRACTUREs*. Then the critical problem is to define effective similarity measures based on shared *FRACTUREs*. As *FRACTURE* patterns differentiate each other in not only its elements, i.e., the subtrees, but also its strength, i.e., the associated frequency of change and significance of change, the designed similarity measures should take both into account. Furthermore, the novelty of discovered clusters should be analyzed.

- ***negative FRACTURE* definition, discovery and application.**

In order to identify the negative nature of subtrees in undergoing significant changes, a correlation metric should be designed soundly before the *negative FRACTURE* can be formally defined. Moreover, *negative FRACTUREs* can be either defined on subtrees structurally independent or defined on subtree structurally related. For example, for applications such as Web site mirror maintenance, *negative FRACTUREs* should be defined on a set of subtrees with ancestor-descendant relationship. Thus, in the mining process, we need to reserve the ancestor-descendant relationship of changed subtrees.

- ***SEASON* definition, discovery and application.**

When defining *SEASON* patterns, it should be noticed that *SEASONs* are discovered from an evolutionary tree database while its form is a sequence of tree clusters. Then, the critical problem of *SEASON* mining is to generate tree clusters which collectively represent the meaning of tree structures. Clustering trees of similar semantics is a challenging problem as simple similarity measures may fail to capture the semantic proximity while complex similarity measures may lead to high complexity. When applying *SEASON* patterns in XML query caching, appropriate caching replacement strategies should be designed based on discovered *SEASONs*.

The above research challenges will be thoroughly studied in the following chapters.

3.4 Related Work

In this section, we discuss some related works which also mine patterns (associations) from changes.

TAR [145] is a type of temporal association rules mined from evolving numerical attributes. It aims to capture the correlation among numerical attribute evolutions. The mining context (database) is a set of objects where each object has a unique ID and a set of numerical attributes $A_1, A_2, \dots, A_m$. The database is viewed as a sequence of snapshots $S_1, S_2, \dots, S_n$ of objects and their attribute values taken at some frequency. An example of TAR is $(salary \in [40000, 45000]) \rightarrow (salary \in [47500, 55000]) \Leftrightarrow (housing_expense \in [10000, 15000]) \rightarrow (housing_expense \in [15000, 17500])$. Such a rule indicates that when the attribute *salary* evolves from the interval of $[40000, 45000]$ to the interval of $[47500, 55000]$, the attribute *housing_expense* evolves from the interval of $[10000, 15000]$ to the interval of $[15000, 17500]$. TAR is totally different from our intra-tree associative evolutionary structures in the pattern forms, interestingness measures, data mining algorithms etc. However, although TAR studies the evolution of numerical attributes while our *SEASON* pattern focuses on the evolution of tree structures, they are similar in the way that TAR needs to categorize the attribute values into meaningful intervals, e.g., $[40000, 45000]$ and *SEASON* needs to categorize the tree structures into meaningful semantic clusters before mining the evolution associations. However, the *SEASON* patterns focus on the evolution of one tree structure while TAR considers the evolution of multiple attributes.

Emerging pattern [53] was proposed to capture significant changes and differences between datasets. Basically, emerging patterns are defined as itemsets whose supports increase significantly from one dataset to another dataset. Thus, when applied to timestamped databases, emerging patterns can capture emerging trends in business or demographic data. Our study is different from emerging patterns in that we consider the changes in a sequence of snapshots of the data while emerging patterns consider only two snapshots. Consequently, emerging patterns consider merely the significance of changes in two snapshots, while our work needs to take into account other characteristics of changes, such as the frequency of changes. Obviously, the knowledge discovered by our work and emerging patterns is different as well. For example, emerging patterns capture useful contrasts between two snapshots

while our associative evolutionary structures capture the evolutionary characteristics of tree structure.

Temporal Text Mining (TTM) is also concerned with discovering temporal patterns in text information collected over time. Recently, a particular TTM task – discovering and summarizing the evolutionary patterns of themes in a text stream – was proposed by Mei and Zhai [100]. The evolutionary theme patterns (ETP) discovery problem aims to discover the evolution of themes, i.e., the happening of the Asian tsunami disaster, the statistics of victims and damage, the aids from the world and the lessons from the tsunami. Similar to TAR, ETP refers to patterns of objects (themes) evolving from one status (subtopic) to another status. Thus, ETP has some relations with our *SEASON* patterns as well. However, they are different in the ways they generate topics (semantics) from the input data. For example, the input of ETP is a stream of documents. Then, ETP extracts multiple topics from each documents. Whereas, the input of *SEASON* is sequences of tree structures and *SEASON* mining assigns multiple tree structures into one semantic cluster. Then, after extracting topics from documents, the discovery step of ETP needs to handle the multiple dimensional data. Whereas, the discovery step of *SEASON* needs to handle one dimensional data after transforming each tree version into a semantic cluster.

Our patterns are also related to works on regularities in time series. These previous works include partial periodic patterns [69, 68], cyclic association rules [106], sequential patterns in single-variable numerical time series [8], frequent episode [97], inter-transaction association rules [137], and the work [89] which studied the problem of efficiently mining the phrases whose historical frequency curves match a given shape (trend) in time-stamped text databases. The basic differences between our work and the above works are the datasets and the data mining objectives. For example, most of the above works consider sets of items or sequences of items while our work considers the tree structures. Whereas, most of the data mining objectives of the above works can be introduced into an evolutionary tree (subtree) database as well.

3.5 Summary

We have defined the research problems of this dissertation by classifying associative evolutionary structures as intra-tree associative evolutionary structures and inter-tree associative evolutionary structures. We have introduced the ideas of *FRACTURE*, *negative*

CHAPTER 3. PROBLEM DEFINITION

FRACTURE and *SEASON* patterns. The first two capture the evolution associations from the structural variation view while the last extracts the semantic evolution associations by viewing the changes to tree structure from the semantic change perspective. Examples of the patterns have been given to highlight the potential usefulness of the proposed patterns. We also have discussed the challenges in discovering proposed patterns and reviewed some related research problems. In the remainder of this dissertation, we will investigate the algorithms on mining *FRACTURE*, *negative FRACTURE* and *SEASON* patterns, and will study the usefulness of these patterns in according applications.

Chapter 4

FRACTURE Mining

We have introduced *FRACTURE* as a type of *intra-tree associative evolutionary structures* in the previous chapter. Mined from a sequence of historical tree versions, basically, a *FRACTURE* pattern is a set of subtrees that frequently change together and frequently undergo significant changes together. Thus, evolution metrics that measure the change frequency and the change significance of individual subtrees as well as subtree sets should be defined. Due to its significant difference with the traditional frequent patterns, we develop novel data mining algorithms for *FRACTURE* mining. Moreover, in order to make the resulting patterns concise so that domain experts can comprehend them easily, we further prune the redundancy in *FRACTURES* based on a “subsumption” relationship between patterns. The notion of *maximal FRACTURE* is defined based on the subsumption relationship¹. We also design a set of optimizing strategies for *maximal FRACTURE* pattern mining. *FRACTURE* patterns can capture the evolutionary characteristics of an evolutionary tree structure from certain aspect. Hence, they can be applied in domains where data have tree structures and temporal features. The content in this chapter appeared in our previous publications [33, 34, 38].

We organize the remainder of this chapter as follows. We begin with some preliminary definitions. In Section 4.2, we describe the algorithms for *FRACTURE* mining. In Section 4.3, we propose the notion of *maximal FRACTURE* and the optimizing strategies for *maximal FRACTURE* mining. We evaluate the performance of the developed algorithms in Section 4.4. We then present the potential applications of *FRACTURE* patterns in Section 4.5. A summary is provided at the end of the chapter.

¹ Although we use the name *maximal FRACTURE*, it is defined completely differently from the maximal pattern in traditional frequent pattern mining. Hence, other ideas of concise patterns in traditional frequent pattern mining, such as closed pattern, cannot be applied here either.

4.1 Preliminaries

In this section, we formally define the *FRACTURE* pattern mined from a sequence of historical versions of a tree structure. We first introduce some preliminaries of the tree structure model. Then, a *structural delta database* is defined to record changed subtrees and their corresponding change degrees. Next, we define evolution metrics to measure the change frequency and change significance of a set of subtrees. Finally, the *FRACTURE* pattern is defined based on these metrics.

4.1.1 Background on Trees

A tree is an acyclic connected directed graph. There are many types of trees, such as unrooted unordered trees (free trees), rooted unordered trees and rooted ordered trees [139]. Tree-structured Web data are usually modelled as rooted trees. Particularly, in our study, we focus on the model of rooted unordered trees. However, our work can be extended to rooted ordered trees easily.

Rooted Unordered Subtree. A *rooted unordered tree* is a triplet $S = \langle N, E, L \rangle$, where N is the set of nodes, E is the set of directed edges and L is the set of labels on the nodes. Particularly, a node $r \in N$ is defined as the root of the tree. For each edge $(n_1, n_2) \in E$, where $n_1, n_2 \in N$, we call n_1 the *parent* of n_2 and n_2 a *child* of n_1 . Each node has one and only parent. Nodes that share the same parent are *siblings*. A *path* is a sequence of connected edges, i.e., $p = \langle \langle n_1, n_2 \rangle, \langle n_2, n_3 \rangle, \dots, \langle n_{k-1}, n_k \rangle \rangle$, where $n_i \in N$ ($1 \leq i \leq k$), and k is the number of nodes on the path. Briefly, we represent the path just by the nodes on the path $\langle n_1, n_2, \dots, n_k \rangle$. A node x is called an *ancestor* of another node y if there exists a path starting from x to y , and accordingly y is called a *descendent* of x . A path starting from the root node is called a *root path*. A node that has no descendant other than itself is called a *leaf node*; otherwise, it is called an *internal node*. The *depth* or *level* of a node is defined as the length of the root path arriving at that node. The *size* of a tree S , denoted as $|S|$, is defined as the number of nodes the tree has.

There are also different notions of subtrees, such as bottom-up subtrees, induced subtrees and embedded subtrees. In this dissertation, we consider the bottom-up subtrees.

Bottom-up Subtree. For a rooted unordered tree $S = \langle N, E, L \rangle$, we say that an unordered tree $s_i = \langle N_i, E_i, L_i \rangle$, rooted at the node i , is a bottom-up subtree of S , denoted as

CHAPTER 4. FRACTURE MINING

Figure 4.1: Sequence of tree versions.

$s_i \preceq S$, if and only if (1) $i \in N$ (2) $N_i \subseteq N$, (3) $E_i \subseteq E$, (4) the labeling of N_i is preserved in s_i , (5) for a node $x \in N$, if $x \in N_i$ then all descendants of x (if any) must be in N_i .

Intuitively, a bottom-up subtree s_i of S can be obtained by taking a node i of S together with all i 's descendants and the corresponding edges. Thus, a tree S , where $|S| = m$, has total m bottom-up subtrees. Since there is only one root path to any node in the tree S , each subtree can be uniquely identified by the root path arriving at the root of the subtree. In this dissertation, we use the node labels on the root path to identify a subtree. For example, consider the leftmost tree in Figure 4.1. Since the size of the tree is 8, it has 8 bottom-up subtrees: a , a/b , $a/b/d$, $a/b/d/e$, $a/b/d/f$, a/c , $a/c/g$, $a/c/h$.

4.1.2 Structural Delta Database

Let $\mathcal{T} = \langle t_1, t_2, \dots, t_n \rangle$ be a sequence of time points with a particular time granularity, e.g., week, month, quarter, year etc. Given a tree structure S , we study the sequence of historical versions of S on $\mathcal{T}$, which is denoted as $\langle S^{t_1}, S^{t_2}, \dots, S^{t_n} \rangle$. Particularly, we assume all tree versions are rooted at the same node. An example sequence of 4 historical tree versions, $\langle S^{t_1}, S^{t_2}, S^{t_3}, S^{t_4} \rangle$, is shown in Figure 4.1.

Given two versions of a tree structure, many types of *edit operations* have been defined in traditional change detection algorithms. An edit operation is an operation e that can be applied on a tree $S_1 = \langle N_1, E_1, L_1 \rangle$ to produce another tree $S_2 = \langle N_2, E_2, L_2 \rangle$, denoted as $S_1 \xrightarrow{e} S_2$. We review some typical edit operations in the literature [31, 32] before defining those we consider in this dissertation.

- **Insertion:** Intuitively, an insertion operation, $INS(x, p, C)$, creates a new tree node with a given label, and places it at a given position in the tree. The position of the new node x in the tree is specified by giving its parent node p and a subset C of the

CHAPTER 4. FRACTURE MINING

children of p . The result of this operation is that x is a child of p , and the nodes C , that were originally children of p , are now children of the newly inserted node x .

- **Deletion:** This operation is the inverse of the insertion operation. Intuitively, deletion of node x , $DEL(x)$, causes x to disappear from the tree; the children of x are now the children of the (old) parent of x . The root of the tree cannot be deleted.
- **Update:** The operation $UPD(x, l)$ changes the label of the node x to a new label l .

The above edit operations are defined based on nodes. In traditional change detection algorithms, there are also some operations defined on subtrees, such as *move a subtree*, *copy a subtree* and *delete a subtree* etc. Usually, an edit operation defined on a subtree can be decomposed into a sequence of edit operations defined on nodes.

The insertion and deletion operations presented above are also decomposable because they allow the insertion and deletion of internal nodes. For example, the insertion of an internal node can be decomposed into deleting a set of sibling nodes C and all their descendants first, inserting a new node as a child of the (old) parent of C and inserting C and all its descendants under the new node.

In our study, we consider the most basic edit operations, i.e., atomic operations, defined on leaf nodes. Hence, we define only two edit operations as follows. Note that, we do not consider the update operation because it does not incur structural changes. Certainly, the update operation defined above can be regarded as deleting a node and inserting a node with a new label.

- **INS**($x(name), p$): This operation creates a new leaf node x , with node label “name”, as a child node of node p in a tree.
- **DEL**(x): This operation is the inverse of the insertion one. It removes leaf node x from a tree.

For example, consider the first two tree versions, S^{t_1} and S^{t_2} , in Figure 4.1. The version S^{t_2} can be produced from S^{t_1} by applying two edit operations: inserting a node labeled i as a child of node b and deleting a node labeled f .

Given two versions of a tree (subtree), *edit distance* [165] is usually adopted in traditional change detection algorithms to measure the distance between the two versions. Edit distance

CHAPTER 4. FRACTURE MINING

is defined based on edit script, which is a sequence of edit operations that converts one tree into another. Note that, in traditional change detection algorithms there may exist more than one valid edit script since the edit operations are decomposable. However, since we consider atomic edit operations, there exists only one valid edit script. Then, edit distance can be defined as follows.

Definition 4.1 (Edit Distance) *Given two versions of a tree S at time points t_j and t_{j+1} , denoted as S^{t_j} and $S^{t_{j+1}}$, let $E(S^{t_j} \rightarrow S^{t_{j+1}})$ be a sequence of basic edit operations, $\langle e_1, e_2, \dots, e_n \rangle$, which converts S^{t_j} to $S^{t_{j+1}}$, then, the edit distance between the two versions, denoted as $d(S^{t_j}, S^{t_{j+1}})$, is the number of edit operations in the edit script.*

$$d(S^{t_j}, S^{t_{j+1}}) = |E|$$

□

Example 4.1 Let the subtree a/b in Figure 4.1 be s . The following sequence of edit operations convert s^{t_1} to s^{t_2} : $E(s^{t_1} \rightarrow s^{t_2}) = \langle DEL(f), INS(b, i) \rangle$. Then, the edit distance between s^{t_1} and s^{t_2} , $d(s^{t_1}, s^{t_2})$, is 2. ■

Considering that an edit distance is an absolute value which measures the difference between two tree structures, we define the Degree of Change (DoC) of a tree in two versions by normalizing the edit distance by the size of the *consolidate tree* of the two tree versions.

Definition 4.2 (Consolidate Tree) *Given two tree versions $S^{t_j} = \langle N^{t_j}, E^{t_j}, L^{t_j} \rangle$ and $S^{t_{j+1}} = \langle N^{t_{j+1}}, E^{t_{j+1}}, L^{t_{j+1}} \rangle$, the consolidate tree of them, denoted as $S^{t_j} \uplus S^{t_{j+1}}$, is $\langle N, E, L \rangle$, where i) $N = N^{t_j} \cup N^{t_{j+1}}$, ii) $e = (x, y) \in E$, if and only if x is the parent of y in E^{t_j} or $E^{t_{j+1}}$.*

□

For example, the consolidate tree of the subtree a/c in the first two versions in Figure 4.1 is the subtree a/c in the second version.

Definition 4.3 (Degree of Change) *Given two versions of a tree S^{t_j} and $S^{t_{j+1}}$, let $d(S^{t_j}, S^{t_{j+1}})$ be the edit distance between the two versions, then the Degree of Change of S from t_j to t_{j+1} , denoted as $DoC(S, t_j, t_{j+1})$, is:*

$$DoC(S, t_j, t_{j+1}) = \frac{d(S^{t_j}, S^{t_{j+1}})}{|S^{t_j} \uplus S^{t_{j+1}}|}$$

where $|S^{t_j} \uplus S^{t_{j+1}}|$ is the size of the consolidate tree of S^{t_j} and $S^{t_{j+1}}$.

□

CHAPTER 4. FRACTURE MINING

The value of DoC ranges from 0 to 1. If a tree does not change in two versions, then its DoC is zero. If a tree is totally removed or newly inserted, then the DoC of the tree is one. The greater the value of DoC , the more significantly the tree changed.

Example 4.2 Let the subtree a/b in Figure 4.1 be subtree s_1 . Then the DoC of s_1 in the first two versions is $DoC(s_1, t_1, t_2) = 2/5$. Let the subtree $a/b/d$ be subtree s_2 , then $DoC(s_2, t_1, t_2) = 1/3$. ■

When a tree S evolves from one version to another version, some of its subtrees change (i.e., their DoC values are greater than zero), while some of its subtrees remain unchanged (i.e., their DoC values equal to zero). We are interested in the set of changed subtrees. Particularly, we refer to the set of changed subtrees, in two successive versions of S , as the *Structural Delta*.

Definition 4.4 (Structural Delta) Given two versions of a tree S at time t_j and t_{j+1} , the structural delta of S from t_j to t_{j+1} , denoted as $\Delta_S(t_j, t_{j+1})$, refers to the set of changed subtrees in the two versions.

$$\Delta_S(t_j, t_{j+1}) = \{s | s \prec S^{t_j} \text{ \& } DoC(s, t_j, t_{j+1}) > 0\} \cup \{s | s \prec S^{t_{j+1}} \text{ \& } DoC(s, t_j, t_{j+1}) > 0\}.$$

□

For example, consider the first two tree versions in Figure 4.1. $\Delta_S(t_1, t_2) = \{a/b, a/b/i, a/b/d, a/b/d/f, a/c, a/c/j, a/c/j/k\}$. Here, the subscript S can be omitted if it is clear from the context.

Given a sequence of tree versions, correspondingly, there is a sequence of structural deltas. Then, a *Structural Delta DataBase* ($SDDB$) can be defined on the sequence of structural deltas.

Definition 4.5 (Structural Delta DataBase) Given a sequence of historical versions of tree S on $\mathcal{T} = \langle t_1, t_2, \dots, t_n \rangle$, a sequence of structural deltas can be obtained $\langle \Delta(t_1, t_2), \Delta(t_2, t_3), \dots, \Delta(t_{n-1}, t_n) \rangle$. A Structural Delta DataBase $SDDB$ is a set of (DID, SID, DoC) , where DID is the identifier of a structural delta, SID denotes the identifier of a changed subtree and DoC records the Degree of Change of the subtree in the corresponding structural delta. □

For example, the $SDDB$ generated from the four tree versions in Figure 4.1 is shown in Table 4.1.

Table 4.1: Structural delta database.

<i>DID</i>	<i>SID</i>	<i>DoC</i>		<i>DID</i>	<i>SID</i>	<i>DoC</i>
1	a/b	0.40		2	a/c/g/m	1.00
1	a/b/d	0.33		3	a/b	0.43
1	a/b/d/f	1.00		3	a/b/d	0.50
1	a/b/d/i	1.00		3	a/b/d/e	1.00
1	a/c	0.50		3	a/b/d/n	1.00
1	a/c/j	1.00		3	a/b/i	0.50
1	a/c/j/k	1.00		3	a/b/i/o	1.00
2	a/b	0.20		3	a/c	0.43
2	a/b/d	0.33		3	a/c/g	0.33
2	a/b/d/l	1.00		3	a/c/g/q	1.00
2	a/c	0.40		3	a/c/j	0.50
2	a/c/h	1.00		3	a/c/j/k	1.00
2	a/c/g	0.50		3	a/c/p	1.00

4.1.3 Evolution Metrics

We now introduce the metrics defined to measure the *change frequency* and *change significance* for a set of subtrees. Particularly, we use the following notations for the subsequent definitions. Let $\mathcal{T} = \langle t_1, t_2, \dots, t_n \rangle$ be a sequence of time points, $\Sigma = \langle S^{t_1}, S^{t_2}, \dots, S^{t_n} \rangle$ be a sequence of versions of tree S on $\mathcal{T}$, $\langle \Delta(t_1, t_2), \dots, \Delta(t_{n-1}, t_n) \rangle$ be the sequence of corresponding structural deltas, $\Omega = \Delta(t_1, t_2) \cup \dots \cup \Delta(t_{n-1}, t_n)$ be the set of all changed subtrees.

Change frequency measures how frequently a set of subtrees changed together. Hence, we define the *Frequency of Change (FoC)* of a set of subtrees as the fraction of the structural deltas in which all subtrees in the set changed.

Definition 4.6 (Frequency of Change) Given a structural delta database $SDD\mathcal{B}$, let $X = \{s_1, s_2, \dots, s_m\}$ be a set of changed subtrees, $X \subseteq \Omega$, the *Frequency of Change (FoC)* of the set X is:

$$FoC(X) = \frac{\sum_{j=1}^{n-1} V_j}{n-1}$$

$$\text{where } V_j = \prod_{i=1}^m V_{j_i} \text{ and } V_{j_i} = \begin{cases} 1, & \text{if } DoC(s_i, t_j, t_{j+1}) \neq 0 \\ 0, & \text{if } DoC(s_i, t_j, t_{j+1}) = 0 \end{cases} \quad 1 \leq i \leq m \quad \square$$

CHAPTER 4. FRACTURE MINING

The value of FoC ranges from zero to one. When subtrees in a set never changed together, FoC of the set is zero. When subtrees in a set occur together in every structural delta, FoC of the set is one. The greater the value of FoC , the more frequently the set of subtrees changed together.

Example 4.3 Consider the $SDDB$ in Table 4.1. Let X be a set of two subtrees, $X = \{a/b/d, a/c/g\}$, $FoC(X) = 2/3 \approx 0.67$ as the two subtrees occur together for two times in three structural deltas. ■

FoC only reflects the frequency that a set of subtrees changed together, the significance of the changes to the subtrees is not taken into account. We then define another metric *Significance of Change* (SoC) to measure how frequently a set of subtrees underwent significant changes together. We say a subtree underwent significant changes if its DoC is no less than some DoC threshold α . Hence, SoC of a set of subtrees is defined with respect to the DoC threshold α .

Definition 4.7 (Significance of Chance) Given a structural delta database $SDDB$, let $X = \{s_1, s_2, \dots, s_m\}$ be a set of changed subtrees, $X \subseteq \Omega$, let the threshold of DoC be α , the *Significance of Change* (SoC) of the set X , denoted as $SoC_\alpha(X)$, is defined as:

$$SoC_\alpha(X) = \frac{\sum_{j=1}^{n-1} D_j}{(n-1) * FoC(X)}$$

$$\text{where } D_j = \prod_{i=1}^m D_{j_i} \text{ and } D_{j_i} = \begin{cases} 1, & \text{if } DoC(s_i, t_j, t_{j+1}) \geq \alpha \\ 0, & \text{otherwise} \end{cases} \quad 1 \leq i \leq m \quad \square$$

That is, the SoC of a set of subtrees is the ratio of the number of structural deltas where subtrees underwent significant changes together to the number of structural deltas where subtrees in the set changed together. Therefore, if all subtrees in a set changed significantly every time when they changed together, then the SoC of the set will be one; if subtrees in a set never changed significantly when they changed together, then the SoC of the set will be zero.

Example 4.4 Consider the $SDDB$ in Table 4.1 again. Let the threshold of DoC α be 0.3. Let $X = \{a/b, a/c\}$. Then, $SoC_{0.3}(X) = 2/3 \approx 0.66$ because the two subtrees changed together in three structural deltas, while both of them changed significantly in two structural deltas. ■

4.1.4 Pattern Definition

Based on the metrics defined above, **F**requently **A**nd **C**oncurrently **mu**Tating substruct**U**RE (*FRACTURE*) patterns can be defined as follows.

Definition 4.8 (FRACTURE) Let $X = \{s_1, s_2, \dots, s_m\}$ be a set of changed subtrees in a structural delta database *SDDB* on Ω , where $X \subseteq \Omega$. Let α be the threshold of *DoC*, β be the threshold of *FoC* and γ be the threshold of *SoC*. Then, X is a *FRACTURE* if it satisfies the two conditions: 1) $FoC(X) \geq \beta$, 2) $SoC_\alpha(X) \geq \gamma$. $\square$

According to the definitions of *FoC* and *SoC*, the semantics of a *FRACTURE* pattern can be understood as a set of subtrees that not only changed together frequently but also underwent significant changes together frequently in a sequence of historical versions of a tree structure.

Example 4.5 Let us consider the structural database in Table 4.1 again. Let $\alpha = 0.4$, $\beta = 0.6$ and $\gamma = 0.6$. The subtree set $X = \{a/b/d, a/c\}$ is a *FRACTURE* because $FoC(X) = 1 \geq \beta$ and $SoC_{0.4}(X) = 2/3 \approx 0.66 \geq \gamma$. $\blacksquare$

Then, the problem of *FRACTURE* pattern mining can be formally stated as follows: Given a structural delta database *SDDB*, the *DoC* threshold α , the *FoC* threshold β and the *SoC* threshold γ , the **problem of FRACTURE mining** is to find the set of all *FRACTURES*: $\{X | X \subseteq \Omega, FoC(X) \geq \beta, SoC_\alpha(X) \geq \gamma\}$.

4.2 FRACTURE Pattern Mining

To the best of our knowledge, *FRACTURE* pattern is defined differently from any other existing frequent patterns in the literature, such as frequent itemsets [5] and frequent sequences [8] etc. Hence, no existing data mining algorithms can be employed directly to mine *FRACTURES*. In this section, we propose a novel method to mine *FRACTURE* patterns. We first describe the basic phases in the mining procedure and then discuss the designed algorithms.

4.2.1 Procedure Overview

Given a sequence of historical versions of a tree structure, two phases are involved in the mining procedure.

- **Phase I: *SDDB* construction.** This phase takes the sequence of historical versions of a tree structure as input and generates the structural delta database *SDDB*. Since existing change detection systems such as [147] can detect defined basic change operations (*insertion* and *deletion* of leaf nodes), changed subtrees in each pair of successive versions can be identified easily. That is, if a node has any inserted or deleted descendants, then it is the root of a changed subtree. The *DoC* of the subtree can be calculated directly according to its definition. Thus, the *SDDB* can be constructed accordingly.
- **Phase II: Pattern Discovery.** In this phase, we mine the set of *FRACTURE* patterns with the input of the constructed *SDDB* and the set of thresholds α , β , γ of *DoC*, *FoC* and *SoC*.

Since the first phase can be handled in a straightforward manner as described above, subsequent discussions will be focused on the *Phase II* to mine the *FRACTURE* patterns from the *SDDB*.

We developed two algorithms, *Apriori-FRACTURE* and *FPG-FRACTURE*, to mine the set of *FRACTURE* patterns. *Apriori-FRACTURE* is an apriori-like algorithm that searches the set of *FRACTURE*s with the *level-wise* strategy; while *FPG-FRACTURE* is developed from the well known algorithm FP-growth [71] which employs the *divide-and-conquer* strategy.

4.2.2 Algorithm I: Apriori-FRACTURE

Apriori-FRACTURE is developed based on the “downward closure” property of the metric *FoC*. That is, given a subtree set which satisfies the threshold of *FoC*, all of its nonempty subsets satisfy the threshold of *FoC* as well.

Property 4.1 *Given a *SDDB* database $\mathcal{D}$ and the threshold of *FoC* β , if a subtree set X' satisfies the threshold, $FoC(X') \geq \beta$, then, $\forall X \subseteq X', FoC(X) \geq \beta$. $\square$*

Unfortunately, the “downward closure” property does not hold with respect to the metric *SoC*. That is, even if a subtree set satisfies the threshold of *SoC*, it is not necessary that all of its subsets satisfy the threshold as well (this can be simply obtained from the definition of *SoC*). Hence, only the metric *FoC*, rather than the metric *SoC*, can be utilized to prune candidates.

CHAPTER 4. FRACTURE MINING

According to Property 4.1, candidate *FRACTURE* patterns can be generated in the similar way as the algorithm *Apriori* [7]. We call a subtree set containing k subtrees as a k -*subtree-set*. Then two k -*subtree-sets* that satisfy the threshold of *FoC* and share a prefix of $k - 1$ subtrees can be joined to generate a candidate $(k + 1)$ -*subtree-set*. For each generated candidate set, we need to check not only its *FoC* but also its *SoC*. If both the *FoC* and the *SoC* of the candidate set satisfy the respective thresholds, then it is a *FRACTURE*. If only the *FoC* of the candidate set satisfies the threshold, then the candidate set will be preserved for generating candidate sets in the subsequent round. If neither the *FoC* nor the *SoC* of the candidate set satisfies the respective thresholds, the candidate set will be discarded. Note that *Apriori* [7] generates candidate patterns of the subsequent round only from the frequent patterns discovered in the current round. By contrast, we need to generate candidate subtree sets not only from the *FRACTURE*s discovered in the current round but also the candidate sets satisfying the threshold of *FoC*. The reason is that even if a subtree set does not satisfy the threshold of *SoC*, it is possible that some of its supersets satisfy the threshold.

In order to further prune the search space, we utilize the following lemma which is based on the product of *FoC* and *SoC*².

Lemma 4.1 *Given a structural delta database $SDD\mathcal{B}$, a *DoC* threshold α , *FoC* threshold β and a *SoC* threshold γ , if a subtree set X satisfies the condition that $FoC(X) \times SoC_\alpha(X) < \beta \times \gamma$, then 1) X is not a *FRACTURE*; 2) any superset of X is not a *FRACTURE* as well.*

□

Proof: The first conclusion is obvious. If X is a *FRACTURE* pattern, then $FoC(X) \geq \beta$ and $SoC_\alpha(X) \geq \gamma$. Thus, $FoC(X) \times SoC_\alpha(X) \geq \beta \times \gamma$, which contradicts the condition. We then prove the second conclusion. Let $|SDD\mathcal{B}|$ be the size of the structural delta database. Let $|\Delta_g(X)|$ be the number of structural deltas in which subtrees in X changed significantly. According to the definition of *FoC* and *SoC*, $FoC(X) \times SoC_\alpha(X) = \frac{|\Delta_g(X)|}{|SDD\mathcal{B}|}$. Let X' be a subtree set s.t. $X' \supseteq X$, $|\Delta_g(X')| \leq |\Delta_g(X)|$. Then $FoC(X') \times SoC_\alpha(X') = \frac{|\Delta_g(X')|}{|SDD\mathcal{B}|} \leq \frac{|\Delta_g(X)|}{|SDD\mathcal{B}|} = FoC(X) \times SoC_\alpha(X) \leq \beta \times \gamma$. Thus, X' is not a *FRACTURE*. ■

Therefore, we do not need to generate candidate $(k + 1)$ -*subtree-sets* by joining all k -*subtree-sets* that satisfy the threshold of *FoC*. Given a k -*subtree-set* X , it will be used to

²Based on the definitions of *FoC* and *SoC*, the product of *FoC* and *SoC* actually equals to the fraction of structural deltas in which all subtrees in a set changed significantly.

CHAPTER 4. FRACTURE MINING

(a) Apriori-FRACTURE	(b) GenCandidatePatterns
Input: $SDDB$, thresholds α , β and γ Output: The set of <i>FRACTURES</i> P Description: <pre> 1: Q_1 = all individual subtrees with $FoC \geq \beta$ && $(FoC \times SoC_\alpha) \geq (\beta \times \gamma)$ 2: P_1 = all individual subtrees with $FoC \geq \beta$ and $SoC_\alpha \geq \gamma$ 3: for ($k = 2$; $Q_{k-1} \neq \emptyset$; $k++$) do 4: $C_k = \text{GenCandidatePatterns}(Q_{k-1})$ 5: for ($i=1$; $i \leq SDDB$; $i++$) do 6: for each candidate pattern $c_k \in C_k$ do 7: if (all subtrees in c_k changed in Δ_i) then 8: $c_k.FoC_count++$ 9: end if 10: if (all subtrees in c_k changed significantly in Δ_i) then 11: $c_k.SoC_count++$ 12: end if 13: end for 14: end for 15: $Q_k = \{c_k \in C_k \mid c_k.FoC_count \geq (\beta \times SDDB) \ \& \ (c_k.FoC_count \times c_k.SoC_count) \geq (\beta \times \gamma)\}$ 16: $P_k = \{c_k \in C_k \mid c_k.FoC_count \geq (\beta \times SDDB) \ \& \ (c_k.SoC_count/c_k.FoC_count) \geq \gamma\}$ 17: end for 18: return $\bigcup_k P_k$ </pre>	Input: $(k-1)$ -subtree-sets Q_{k-1} , β Output: The set of k -subtree-sets C_k Description: <pre> 1: for each $(k-1)$-subtree-set $\{m_1, m_2, \dots, m_{k-1}\} \in Q_{k-1}$ do 2: for each $(k-1)$-subtree-set $\{n_1, n_2, \dots, n_{k-1}\} \in Q_{k-1}$ do 3: if $(m_1=n_1) \wedge \dots \wedge (m_{k-2}=n_{k-2}) \wedge (m_{k-1} < n_{k-1})$ then 4: $c_k = (m_1, \dots, m_{k-2}, m_{k-1}, n_{k-1})$ 5: if c_k has any subset with $FoC < \beta$ then 6: remove c_k 7: else 8: set $c_k.FoC_count=0$; 9: set $c_k.SoC_count=0$; 10: add c_k to C_k 11: end if 12: end if 13: end for 14: end for </pre>

Figure 4.2: Algorithm of Apriori-FRACTURE.

generate candidate $(k+1)$ -subtree-sets only if it satisfies the following two conditions: 1) its FoC is no less than β ; 2) the product of $FoC(X) \times SoC_\alpha(X)$ is no less than $\beta \times \gamma$. The algorithm of *Apriori-FRACTURE* is shown in Figure 4.2. We scan the $SDDB$ for the first time to find the set of individual subtrees, Q_1 , where each satisfies the threshold of FoC and the condition stated in Lemma 4.1. The function *GenCandidatePatterns* is called to generate candidate 2-subtree-sets C_2 from Q_1 . For each candidate set, we scan the $SDDB$ again to compute its FoC and SoC . Then, we find the *FRACTURES* and the set of subtree sets, Q_2 , which will be used to generate candidate sets in the subsequent round. The algorithm iteratively generates the candidate sets and finds *FRACTURES* until the set of Q_{k-1} is empty.

Theorem 4.1 *The algorithm Apriori-FRACTURE discovers the complete set of FRACTURES.* □

The completeness of *Apriori-FRACTURE* follows the Property 4.1 and the Lemma 4.1.

Theorem 4.2 *The complexity of Apriori-FRACTURE is $O(\sum_k \{k \cdot |Q_{k-1}|^3, m \cdot |SDDb| \cdot |C_k|\})$, where m is the cost of checking whether all subtrees in c_k changed (significantly) in each structural delta.* $\square$

Proof: Consider the function *GenCandidatePatterns*. The complexity of examining each pair of sets in Q_{k-1} is $|Q_{k-1}|^2$. For each generated candidate set, we need to check whether it has k subsets in Q_{k-1} , which is of complexity $k \cdot |Q_{k-1}|$. Hence, the complexity of *GenCandidatePatterns* is $k \cdot |Q_{k-1}|^3$. Suppose m is the cost of checking whether all subtrees in c_k changed (significantly) in each structural delta (basically, m depends on the length of each structural delta and the number of candidate sets in the structural delta). Then, the complexity of *Apriori-FRACTURE* from line 5 in Figure 4.2 (a) is $m \cdot |SDDb| \cdot |C_k|$. Therefore, the total complexity of *Apriori-FRACTURE* is $O(\sum_k \{k \cdot |Q_{k-1}|^3, m \cdot |SDDb| \cdot |C_k|\})$. $\blacksquare$

Optimization of Apriori-FRACTURE

The algorithm *Apriori-FRACTURE* can be optimized as follows.

Optimization 4.1 *Computing FoC and SoC of candidate subtree sets in separate rounds.*

$\square$

After generating the set of candidate k -subtree-sets, *Apriori-FRACTURE* scans the *SDDb* to calculate the *FoC* and the *SoC* of them. However, if a candidate subtree set does not satisfy the threshold of *FoC*, we waste resource to compute its *SoC*. Hence, we can count the *FoC* and the *SoC* of candidate sets in separate rounds. In other words, we calculate only the *FoC* of the candidate k -subtree-sets in the k th round. Only if a candidate k -subtree-set satisfies the threshold of *FoC*, its *SoC* will be calculated in the subsequent round, together with the calculating of the *FoC* of the candidate $(k+1)$ -subtree-sets. Referring to Figure 4.2 (a), when line 7 computes the *FoC* of the candidate k -subtree-sets, line 10 computes the *SoC* of the candidate $(k-1)$ -subtree-sets. Efficiency can be improved by computing the *SoC* for a tightened set of candidate sets. However, Lemma 4.1 cannot be utilized to prune search space as the *SoC* of the k -subtree-sets is unavailable until the $(k+1)$ th round. Moreover, an extra scan of database is required to calculate the *SoC* of the candidate sets generated in the last round. The detailed algorithm that integrates Optimization 4.1 with *Apriori-FRACTURE* is given in Figure 4.3. We refer to this algorithm as *Apriori-FRACTURE-I* and evaluate its performance in Section 4.4.

CHAPTER 4. FRACTURE MINING

Apriori-FRACTURE-I

Input: $SDD\mathcal{B}$, thresholds α , β and γ

Output: The set of *FRACTURE*s P

Description:

```

1:  $Q_1$  = all individual subtrees with  $FoC \geq \beta$ 
2: for ( $k=2$ ;  $Q_{k-1} \neq \emptyset$ ;  $k++$ ) do
3:    $C_k$  = GenCandidatePatterns( $Q_{k-1}$ )
4:   for ( $i=1$ ;  $i \leq |SDD\mathcal{B}|$ ;  $i++$ ) do
5:     for each candidate pattern  $c_k \in C_k$  do
6:       if (all subtrees in  $c_k$  changed in  $\Delta_i$ ) then
7:          $c_k.FoC\_count++$ 
8:       end if
9:     end for
10:    for each subtree set  $q_{k-1} \in Q_{k-1}$  do
11:      if (all subtrees in  $q_{k-1}$  changed significantly in  $\Delta_i$ ) then
12:         $q_{k-1}.SoC\_count++$ 
13:      end if
14:    end for
15:    end for
16:     $P_{k-1} = \{q_{k-1} \in Q_{k-1} \mid (q_{k-1}.SoC\_count / q_{k-1}.FoC\_count) \geq \gamma\}$ 
17:     $Q_k = \{c_k \in C_k \mid c_k.FoC\_count \geq (\beta \times |SDD\mathcal{B}|)\}$ 
18:  end for
19: return  $\bigcup_{k=1} P_{k-1}$ 

```

Figure 4.3: *Optimized Apriori-FRACTURE* algorithm.

4.2.3 Algorithm II: FPG-FRACTURE

Apriori-FRACTURE, similar to the algorithm *Apriori* [7], has the bottleneck in generating the candidate sets and scanning database for multiple times. To address the problem, we develop a *divide-and-conquer* algorithm, *FPG-FRACTURE*, which is based on the algorithm FP-growth [71].

Data Structure

FP-growth constructs a special data structure *FP-tree* which contains compact information of frequent itemsets. The fundamentals of FP-tree can be briefly described as follows. First of all, only frequent individual items will have corresponding nodes in the *FP-tree*. Transactions sharing common items share prefix paths in the *FP-tree*. Each node registers the number of transactions in which it occurs, together with nodes in its prefix path. Node links are used to indicate the occurrences of an item in different paths. A head table maintains the set of frequent individual items and pointers to their node links.

Obviously, an *FP-tree* can be used here directly to record the information of *FoC* for subtree sets. That is, the information of which subtrees changed together in each structural delta. However, it cannot simultaneously maintain the information of *SoC* for subtree sets, such as the information of which subtrees changed significantly together in each structural

CHAPTER 4. FRACTURE MINING

DID	SID	DoC	DID	SID	DoC	DID	SID	DoC	DID	SID	DoC	DID	SIDs
1	s_1	0.2	2	s_1	0.3	3	s_1	0.22	4	s_2	0.24	1	$s_1, -s_2, -s_3, s_4, s_5$
1	s_2	0.05	2	s_2	0.25	3	s_2	0.25	4	s_3	0.22	2	$s_1, s_2, -s_3, s_4, s_5$
1	s_3	0.1	2	s_3	0.1	3	s_4	0.3	5	s_1	0.05	3	$s_1, s_2, s_4, -s_5$
1	s_4	0.25	2	s_4	0.3	3	s_5	0.05	5	s_2	0.1	4	$-s_1, s_2, s_3$
1	s_5	0.3	2	s_5	0.22	4	s_1	0.1	5	s_3	0.35	5	$-s_1, -s_2, s_3$

(a)

(b)

Figure 4.4: Transforming SDDB.

delta. A naïve solution may be to construct two *FP-trees* to record the information of *FoC* and *SoC* respectively and intersect the results mined from the two *FP-trees*. Clearly, this is not space-economical. Therefore, we study how to record the information of both the *FoC* and the *SoC* of subtree sets in one *FP-tree*.

Consider that each subtree in a structural delta has two states: its *DoC* is either less than the threshold of *DoC* α or no less than α . We use a pair of identifiers to represent the two states of a subtree. Given a subtree s , when its *DoC* is less than α , we use an identifier $-s$ to represent it in this structural delta; otherwise, we use the original identifier s . For example, given a *SDDB* shown in Figure 4.4 (a). Let $\alpha = 0.15$, it can be transformed into the one shown in Figure 4.4 (b) (for ease of exposition, we also transformed the schema of the table so that each structural delta has one tuple in the transformed table).

Now if we construct an *FP-tree* from the transformed *SDDB* by creating different nodes for subtrees with different identifiers, the information of both the *FoC* and the *SoC* of subtree sets can be preserved. We call the resulting tree *Signed-FPtree*. The method of constructing a *Signed-FPtree* is similar to the construction of an *FP-tree*. The key difference between the *Signed-FPtree* and the original *FP-tree* is that in a *Signed-FPtree*, node links should connect nodes with identifiers as s and $-s$ since they actually represent the same changed subtree.

When deciding which subtrees will have corresponding nodes in the *Signed-FPtree*, we recall the Property 4.1 and the Lemma 4.1. Thus, given the threshold of *DoC* α , the threshold of *FoC* β and the threshold of *SoC* γ , a subtree s will be constructed in the *Signed-FPtree* if $FoC(s) \geq \beta$ and $(FoC(s) \times SoC_\alpha(s)) \geq (\beta \times \gamma)$. For example, let $\alpha = 0.15$, $\beta = 0.4$ and $\gamma = 0.5$. The *Signed-FPtree* constructed from Figure 4.4 (b) is presented in Figure 4.5 (a).

Figure 4.5: Signed-FPtree.

The algorithm of constructing a *Signed-FPtree* is given in Figure 4.6 (a). The completeness of the *Signed-FPtree* can be justified by the following theorem.

Theorem 4.3 *Given a structural delta database $SDD\mathcal{B}$, a threshold of DoC α , a threshold of FoC β and a threshold of SoC γ , the constructed *Signed-FPtree* contains the complete information of $SDD\mathcal{B}$ in relevance to FRACTURE mining.* $\square$

Proof: According to the construction of the *Signed-FPtree*, each structural delta in the $SDD\mathcal{B}$ is mapped to one path in the *Signed-FPtree*. The two states of a subtree in each structural delta is preserved by using nodes with different identifiers. Hence, the information of FRACTURES in each structural delta is completely stored in the *Signed-FPtree*. ■

Mining Algorithm

We now explain how to mine the set of FRACTURES from the *Signed-FPtree*. The algorithm of *FPG-FRACTURE* is shown in Figure 4.6 (b). The critical differences between *FPG-FRACTURE* and the original *FP-growth* are the way we calculate the FoC and the SoC for subtree sets and the way we construct the *conditional Signed-FPtree*. In the following, we illustrate the algorithm and the differences with an example. Consider the last subtree, s_5 , in header table of the *Signed-FPtree* in Figure 4.5 (Line 10). $FoC(s_5) = 3/5 = 0.6$ since the total number of occurrences of s_5 and $-s_5$ in the *Signed-FPtree* is three. $SoC(s_5) = 2/3 \approx 0.66$ since s_5 occurs twice. Hence, $\{s_5\}$ is a FRACTURE (Line 11 – 14). There are three paths related to s_5 : $\langle s_1 : 1, -s_2 : 1, -s_3 : 1, s_4 : 1, s_5 : 1 \rangle$, $\langle s_1 : 1, s_2 : 1, -s_3 : 1, s_4 : 1, s_5 : 1 \rangle$ and $\langle s_1 : 1, s_2 : 1, s_4 : 1, -s_5 : 1 \rangle$ (each number after

CHAPTER 4. FRACTURE MINING

a colon indicates the number of structural deltas in which the subtree changed together with s_5). In order to mine *FRACTURE*s related to s_5 , we need to construct its *conditional Signed-FPtree*. Since the β is 0.4 and the γ is 0.5, subtree s_3 should not be included in s_5 's *conditional Signed-FPtree* as $(FoC(s_3) \times SoC(s_3)) = 0 < (\beta \times \gamma)$. Hence, we construct s_5 's *conditional Signed-FPtree* from the following three prefix paths: $\langle s_1 : 1, -s_2 : 1, s_4 : 1 \rangle$, $\langle s_1 : 1, s_2 : 1, s_4 : 1 \rangle$ and $\langle s_1 : 1, s_2 : 1, s_4 : 1 \rangle$ (Line 15). Note that in the third prefix path, subtree s_5 occurs as $-s_5$, which means subtree s_5 did not change significantly with the subtrees s_1 , s_2 and s_4 in this structural delta. To record this fact, we need to replace the identifiers of the three subtrees with $-s_1$, $-s_2$ and $-s_4$. We shall justify the correctness of this operation in the Lemma 4.2 in the below. Then the *conditional Signed-FPtree* of s_5 is shown in Figure 4.5 (b). Mining from it (Line 16 – 17), we firstly generate the pattern $\{s_4, s_5\}$. Considering both occurrences of s_4 and $-s_4$ in Figure 4.5 (b), the *FoC* of $\{s_4, s_5\}$ is 0.6. Considering the occurrences of s_4 only, its *SoC* is 0.66. Then the pattern $\{s_4, s_5\}$ is a *FRACTURE*. Other *FRACTURE*s can be mined similarly.

Lemma 4.2 Let $X = \{s_1, s_2, \dots, s_n\}$ be a *FRACTURE* such that s_n is the last subtree that was added to X . Let A be the (conditional) *Signed-FPtree* from which X is discovered. When constructing conditional *Signed-FPtree* B from A to mine *FRACTURE*s related to X , any subtree occurs as s_i , on paths where subtree s_n occurs as $-s_n$, should be replaced with $-s_i$. The replacement does not affect the correctness when recursively constructing conditional *Signed-FPtrees* from B . $\square$

Proof: On the paths in A where s_n occurs as $-s_n$, any subtree s_i occurring as s_i did not change significantly together with s_n . Hence, replacing s_i with $-s_i$ records the correct information of the *SoC* of $X \cup \{s_i\}$ without affecting the *FoC* of $X \cup \{s_i\}$. Suppose $\{X \cup s_{n+1}\}$ is the *FRACTURE* mined from the *conditional Signed-FPtree* B . Then, for a *conditional Signed-FPtree* C constructed from B , where *FRACTURE*s related to $\{X \cup s_{n+1}\}$ will be mined, the replacement does not affect the correctness of C because of the following reason. If s_i did not change significantly together with X in this structural delta, it also did not change significantly together with $\{X \cup s_{n+1}\}$. Thus, we have the lemma. $\blacksquare$

Theorem 4.4 The algorithm *FPG-FRACTURE* discovers the set of *FRACTURE* completely. $\square$

The correctness of *FPG-FRACTURE* comes from the completeness of *Signed-FPtree*, the proved correctness of *FP-growth* and the Lemma 4.2.

CHAPTER 4. FRACTURE MINING

(a) Signed-FPtree Construction	(b) FPG-FRACTURE
Input: A transformed $SDDB'$, β, γ Output: Constructed Signed-FPtree A Description: <ol style="list-style-type: none"> 1: scan $SDDB'$ once to find the set of $Q_1 =$ all individual subtrees with $FoC \geq \beta$ && $(FoC \times SoC) \geq (\beta \times \gamma)$ 2: Sort subtrees in Q_1 in descending order of their FoC as L, the list of potential $FRACTUREs$. 3: Create the root of a Signed-FPtree A, and label it as "null". 4: for each $\Delta_i \in SDDB'$ do 5: Select subtrees in L from Δ_i and sort them according to L. Represent the list of selected and sorted subtrees in the form of $[m M]$, where m is the first subtree and M is the remaining list. Call $INS_TREE([m M], A)$ 6: end for 7: function $INS_TREE([m M], A)$ 8: if A has a child node n such that $n.identifier = m.identifier$ then 9: increment n's count by 1 10: else 11: create a new node n, initialize its count as 1, its parent node be A and its node-link linking with nodes with identifier as either $m.identifier$ or $-m.identifier$ 12: end if 13: if M is not empty then 14: call $INS_TREE(M, n)$ 15: end if 16: end function 	Input: Signed - FPtree A , β, γ Output: P : A set of $FRACTUREs$ Description: <ol style="list-style-type: none"> call $FPG-FRACTURE(A, null)$ 1: function $FPG-FRACTURE(A, a)$ 2: if A contains a single path P then 3: for each combination (denoted as b) of the nodes in the path P do 4: generate pattern $b \cup a$ with $FoC(b \cup a) = \text{minimum } FoC \text{ of nodes in } b$ and $SoC(b \cup a) = \text{minimum } SoC \text{ of nodes in } b$ 5: if $SoC(b \cup a) \geq \gamma$ then 6: $P = P \cup (b \cup a)$ 7: end if 8: end for 9: else 10: for each s in the header of tree do 11: generate pattern $b = s \cup a$ with $FoC(b) = FoC(s)$ and $SoC(b) = SoC(s)$ 12: if $SoC(b) \geq \gamma$ then 13: $P = P \cup b$ 14: end if 15: construct b's conditional Signed-FPtree $tree_b$ 16: if $tree_b \neq \emptyset$ then 17: $FPG-FRACTURE(tree_b, b)$ 18: end if 19: end for 20: end if 21: end function

Figure 4.6: Algorithms of Signed-FPtree construction and FPG-FRACTURE.

4.3 Maximal FRACTURE Pattern Mining

Similar to traditional frequent pattern mining which introduces concise patterns, such as maximal frequent pattern [14], to represent the complete set of frequent patterns with a tightened set of patterns, we present in this section the mining of a tightened set of $FRACTUREs$, called *maximal FRACTUREs*, from which the complete set of $FRACTUREs$ can be inferred. We first define the *maximal FRACTURE* patterns and then discuss the data mining algorithm.

4.3.1 Maximal Pattern Definition

Traditional definition on maximal frequent pattern is that a frequent pattern is *maximal* if none of its superset is frequent. The complete set of frequent patterns then can be

CHAPTER 4. FRACTURE MINING

inferred from the maximal ones in the way that all subsets of a maximal frequent pattern are frequent as well. Unfortunately, *maximal FRACTUREs* cannot be defined in the similar way because a subset of a *FRACTURE* may not necessarily be a *FRACTURE* as well. This can be understood easily as the “downward closure” does not hold for the metric *SoC*.

However, we observed a redundancy existing in some specific pair of subtree sets such that if one of the subtree sets is a *FRACTURE*, the other must be a *FRACTURE* as well. We say the two subtree sets in such a specific pair have a *subsumption* relationship, which is defined as follows.

Definition 4.9 (Subsumption) *Given a structural delta database $SDD\mathcal{B}$ on $\mathcal{T} = \langle t_1, t_2, \dots, t_n \rangle$, let Ω be the set of changed subtrees, X and X' be two subtree sets s.t. $X \subseteq \Omega$, $X' \subseteq \Omega$, $X' = X \cup \{s_1, s_2, \dots, s_n\}$ and $X \cap \{s_1, s_2, \dots, s_n\} = \emptyset$. If $\forall i (1 \leq i \leq n), \exists s_j \in X$ s.t. $s_j^{t_k} \prec s_i^{t_k}$ ($t_1 \leq t_k \leq t_n$), we say X is subsumed by X' , or X' subsumes X , denoted as $X \prec X'$.* $\square$

Example 4.6 Consider the ancestor relationship between subtrees in Figure 4.7 (b). Let $X = \{s_1, s_4\}$ and $X' = \{s_1, s_4, s_5\}$. Then, $X' \succ X$ because $s_5 \prec s_4$. $\square$

Then we have the following lemma between a pair of subtree sets that have the subsumption relationship.

Lemma 4.3 *Given two subtree sets X and X' s.t. $X \prec X'$, if X' is a *FRACTURE*, then X is a *FRACTURE* as well.* $\square$

Proof: Let $|SDD\mathcal{B}|$ be the size of the structural delta database, $|\Delta(X)|$ be the number of structural deltas in which subtrees in X changed together and $|\Delta_g(X)|$ be the number of structural deltas in which subtrees in X changed significantly together. Suppose the threshold of *FoC* is β and the threshold of *SoC* is γ . Since $X \prec X'$, $|\Delta(X)| = |\Delta(X')|$, $FoC(X) = \frac{|\Delta(X)|}{|SDD\mathcal{B}|} = \frac{|\Delta(X')|}{|SDD\mathcal{B}|} = FoC(X') \geq \beta$. Since $X \prec X'$, then $X \subset X'$, $|\Delta_g(X)| \geq |\Delta_g(X')|$. Hence, $SoC(X) = \frac{|\Delta_g(X)|}{|\Delta(X)|} \geq \frac{|\Delta_g(X')|}{|\Delta(X')|} = SoC(X') \geq \gamma$. Then X is also a *FRACTURE* and we have the lemma. $\blacksquare$

Example 4.7 Consider X and X' in **Example 4.6**. Since $X' \succ X$, if X' is a *FRACTURE*, then X must be a *FRACTURE* as well according to **Lemma 4.3**. $\square$

According to the Lemma 4.3, if a subtree set is a *FRACTURE*, we can infer directly that all its subsumed subsets are *FRACTUREs* as well. In other words, a *FRACTURE* can be represented by another *FRACTURE* that subsumes it. Then the notion of *maximal FRACTURE* can be defined as follows.

Figure 4.7: Ordering subtrees in the head table of Signed-FPtree.

Definition 4.10 (maximal FRACTURE) A set of subtrees is a *maximal FRACTURE*, if it is a *FRACTURE* and it is not subsumed by any other *FRACTURE*. $\square$

Obviously, the set of *maximal FRACTURE* patterns is a tightened set of the complete set of *FRACTURES*, $\{\text{maximal FRACTURE}\} \subseteq \{\text{FRACTURE}\}$, and the complete set of *FRACTURES* can be inferred from the set of *maximal FRACTURES*.

4.3.2 Maximal FRACTURE Mining

In this subsection, we discuss the problem of mining the set of *maximal FRACTURES*. It is prohibitively expensive to discover all the *FRACTURES* first and then prune the non-maximal ones. Hence, we study how to integrate the pruning techniques in the mining process. Note that the *Apriori-FRACTURE* algorithm searches the *FRACTURES* in the breath-first way, whereas the *FPG-FRACTURE* adopts the depth-first manner. Consider that if a *FRACTURE* X is subsumed by a *maximal FRACTURE* X' , then the cardinality of X' must be larger than that of X . In order to efficiently discover the *maximal FRACTURES* without generating the non-maximal ones, subtree sets with larger cardinality should be checked first. Hence, a depth-first algorithm is more appropriate to be optimized for the discovery the *maximal FRACTURES*. Thus, we focus on modifying the algorithm *FPG-FRACTURE* to discover the *maximal FRACTURES*.

4.3.2.1 Optimization of Subtree Ordering

As discussed above, in order to efficiently discover the *maximal FRACTURES*, it is ideal to generate a subtree set X' before another subtree set X if $X \prec X'$. Then, if the subtree

CHAPTER 4. FRACTURE MINING

set X' is a *FRACTURE*, we do not need to examine the subtree set X as it will not be maximal. According to the definition of the subsumption relationship, we have the following optimizing strategy.

Optimization 4.2 *Given an ancestor relationship between changed subtrees, a subtree set X should be generated earlier than those containing descendant subtrees of subtrees in X .*

□

Optimization 4.2 guides how to order the list of subtrees in the head table of *Signed-FPtree*. For example, consider the transformed *SDDB* in Figure 4.4 (b), which is redrawn in Figure 4.7 (a). Suppose the ancestor relationship between the changed subtrees is shown in Figure 4.7 (b), which means s_1 is an ancestor subtree of s_2 , s_2 is an ancestor subtree of s_3 and so on. We can arrange them either in descending order of the number of their ancestor subtrees (i.e., $\{s_3 : 2, s_2 : 1, s_5 : 1, s_1 : 0, s_4 : 0\}$, each number after a colon is the number of ancestor subtrees of this subtree) or in the reverse order of depth-first traversal of the ancestor relationship (i.e., $\{s_5, s_4, s_3, s_2, s_1\}$). Note that the former is a partial order while the latter is a total order. With either ordering scheme, a subtree set will be mined earlier than those subsumed by it. For example, with the former ordering scheme, all *FRACTURE*s related to s_4 will be mined before those related to s_5 but s_4 , which are probably subsumed by the *FRACTURE*s related to s_4 and then are not maximal. Before deciding which ordering policy should be employed, we examine the following property first.

Property 4.2 *Given two subtree sets, X and X' s.t. $X \prec X'$, the projected structural delta sets of X' are same as the projected structural delta sets of X .*

□

The projected structural delta sets of a subtree set X is the set of structural deltas in which subtrees in X changed. If $X \prec X'$, then $FoC(X) = FoC(X')$. That is, every structural delta containing the subtree set X contains the subtree set X' as well, vice versa.

According to the Property 4.2, we are presented with the opportunity to mine X and X' from the same *conditional Signed-FPtree*. Furthermore, *FRACTURE*s related to X and X' can also be mined from the same data structure. In order to utilize the same *conditional Signed-FPtree*, *FRACTURE*s related to subtree set X should be examined right after the *FRACTURE*s related to subtree set X' . Then, we need to arrange the individual subtrees in the head table in reverse order of a depth first traversal. For example, given the ancestor

relationship shown in Figure 4.7 (b), the *Signed-FPtree* is shown in Figure 4.7 (c). Let X be the set $\{s_2\}$ and X' be the set $\{s_1, s_2\}$. We can examine both X and X' from the *conditional Signed-FPtree* constructed for $\{s_1\}$ as $X \prec X'$. Then, we can skip the examination of X if X' is discovered to be a *FRACTURE*. Furthermore, the *conditional Signed-FPtree* constructed for $\{s_1, s_2\}$ can be used to mine not only all *FRACTUREs* related to X' but also all *FRACTUREs* related to X .

As explained in the algorithm *FPG-FRACTURE*, labels of some nodes in *conditional Signed-FPtree* need to be shifted to record the correct information. However, this may results in the problem that a *conditional Signed-FPtree* cannot be sharable. For example, suppose we examine the *FRACTUREs* related to $\{s_2\}$ from the *conditional Signed-FPtree* constructed for $\{s_1, s_2\}$. Consider the second path from the right in the *Signed-FPtree* shown in Figure 4.7 (c). Since the subtree s_1 occurs as $-s_1$, the subtree s_3 should be replaced as $-s_3$. Nevertheless, s_3 changed significantly with s_2 in the current structural delta. Then, the *SoC* of $\{s_2, s_3\}$ will be computed wrongly from the *conditional Signed-FPtree* constructed for $\{s_1, s_2\}$. Therefore, we propose an alternative technique of shifting node labels. When constructing a *conditional Signed-FPtree* for a subtree set X , we append a tag to each path in the *conditional Signed-FPtree*, which indicates the states of subtrees in X . For example, Figure 4.8 (a) shows the *conditional Signed-FPtree* for $\{s_1\}$. Each path is appended with a tag: “1” indicates that s_1 changed significantly in this path while “-1” indicates it changed insignificantly in this path. Obviously, the modified *conditional Signed-FPtree* records the complete information of *FoC* and *SoC* of subtree sets without affecting the states of any subtree in the paths. Thus, the modified *conditional Signed-FPtree* is sharable. For example, consider the *conditional Signed-FPtree* constructed for $\{s_1, s_2\}$ which is shown in Figure 4.8 (b). When mining *FRACTUREs* related to $\{s_1, s_2\}$, we consider every bit in each path’s tag. When mining *FRACTUREs* related to $\{s_2\}$ but $\{s_1\}$, we only consider the last bit in the tags.

4.3.2.2 Optimization of Selectively Examining Subtrees

Based on the *Property 4.2* and the technique making the *conditional Signed-FPtree* sharable, we do not need to construct the *conditional Signed-FPtree* for a subtree set X to mine *FRACTUREs* related to it if it is subsumed by any subtree set X' , since *FRACTUREs* related to X can be mined from *conditional Signed-FPtree* where *FRACTUREs* related to X' can be mined. Then we have the following optimization.

CHAPTER 4. FRACTURE MINING

Figure 4.8: Mining process from s_1 .

Optimization 4.3 From the head table of (conditional) Signed-FPtree, only the last subtree and subtrees which are not descendants of the subtree, whose patterns are just examined in the preceding round, need to be examined. $\square$

For example, from the original Signed-FPtree where the head table contains the subtrees in the list as $\langle s_5, s_4, s_3, s_2, s_1 \rangle$, we only need to construct conditional Signed-FPtree for subtree s_1 and s_4 since $s_2 \prec s_1$, $s_3 \prec s_1$, $s_4 \not\prec s_1$ and $s_5 \prec s_4$. The set $\{s_2\}$ can be examined at the same time when examining the set $\{s_1, s_2\}$. The set $\{s_3\}$ can be examined at the same time when examining the set $\{s_1, s_2, s_3\}$. The sets related to s_2 but s_1 can be mined from the conditional Signed-FPtree constructed for $\{s_1, s_2\}$ and the sets related to s_3 but s_1 or s_2 can be mined from the conditional Signed-FPtree constructed for $\{s_1, s_2, s_3\}$. Similarly, from the conditional Signed-FPtree of $\{s_1\}$ in Figure 4.8 (a), we only need to mine s_2 and s_4 .

Recall that in the algorithm *FPG-FRACTURE*, we need to maintain a *SoC* count for each subtree in the head table of the (conditional) Signed-FPtree. The *SoC* count records the number of times the subtree changed significantly with subtrees in the set that the data structure is constructed for. Now, since we examine subtree sets related to both subtree set X' and subtree set X s.t. $X \prec X'$ from the same data structure, we need to maintain more than one *SoC* count for some subtrees in the head table. For example, consider the

CHAPTER 4. FRACTURE MINING

conditional Signed-FPtree for $\{s_1\}$ as in Figure 4.8 (a), we need to maintain two *SoC* counts for s_2 , one records the *SoC* count for the set $\{s_1, s_2\}$ and the other records the *SoC* count for the set $\{s_2\}$. Similarly, for the *conditional Signed-FPtree* constructed for $\{s_1, s_2\}$ as in Figure 4.8 (b), consider the last subtree s_3 . Besides examining the set $\{s_1, s_2, s_3\}$, we examine the other three sets subsumed by it, $\{s_1, s_3\}$, $\{s_2, s_3\}$ and $\{s_3\}$. Hence, there are four *SoC* counts should be maintained for subtree s_3 .

As an induction, the number of *SoC* counts that each subtree in the head table of a *conditional Signed-FPtree* should maintain can be calculated as follows.

Definition 4.11 (Number of SoC Counts) Let $X_i = \langle s_1, s_2, \dots, s_n \rangle$ be a sequence of subtrees s.t. $\forall j \in [1, n-1], s_j \prec s_{j+1}$. Let $FT(X_i)$ be the first subtree and $LT(X_i)$ be the last subtree in X_i . Let $P = X_1 \cup X_2 \cup \dots \cup X_m$, where $\forall i, j (1 \leq i, j \leq m, i \neq j), X_i \cap X_j = \emptyset$ and $\forall k (1 \leq k \leq m-1), LT(X_k) \not\prec FT(X_{k+1})$. Suppose subtree sets related to P are mined from current *conditional Signed-FPtree*. The number of *SoC* counts we need to maintain for subtree s in the head table is as follows.

$$\text{Number of SoC Counts} = \begin{cases} \prod_{i=1}^{m-1} 2^{|X_i|-1} \cdot 2^{|X_m|} & \text{if } s \prec LT(X_m) \\ \prod_{i=1}^m 2^{|X_i|-1} & \text{if } s \not\prec LT(X_m) \end{cases} \quad \square$$

Example 4.8 Consider the *conditional Signed-FPtree* in Figure 4.8 (c). $P = X_1 = \langle s_1, s_2, s_3 \rangle$. Since $s_4 \not\prec s_3$, the number of *SoC* count we need to maintain for s_4 is $2^{3-1} = 4$. The four *SoC* counts record the *SoC* information for subtree sets $\{s_1, s_2, s_3, s_4\}$, $\{s_1, s_3, s_4\}$, $\{s_2, s_3, s_4\}$ and $\{s_3, s_4\}$ respectively. Consider the *conditional Signed-FPtree* in Figure 4.8 (d). $P = X_1 \cup X_2 = \langle s_1, s_2, s_3 \rangle \cup \langle s_4 \rangle$. Since $s_5 \prec s_4$, we need to maintain $2^{3-1} \cdot 2^1 = 8$ *SoC* counts for subtree s_5 . ■

Note that, the checking for *maximal FRACTUREs* can be performed after examining each subtree in the head table of a (conditional) *Signed-FPtree*. For example, after examining the subtree s_3 in the head table of the *conditional Signed-FPtree* in Figure 4.8 (b), we generate four subtree sets: $\{s_1, s_2, s_3\}$, $\{s_1, s_3\}$, $\{s_2, s_3\}$ and $\{s_3\}$. Then, we only need to find *maximal FRACTUREs* from the four subtree sets rather than compare each subtree set with all previously discovered *FRACTUREs* and incoming *FRACTUREs* to verify whether it is maximal.

CHAPTER 4. FRACTURE MINING

Figure 4.9: Maximal FRACTURE mining algorithm.

4.3.2.3 Optimization of Mining Signed-FPtree of Single Path

When the data structure contains a single path, we have the following optimization strategy.

Optimization 4.4 *If the (conditional) Signed-FPtree contains a single path, maximal FRACTURES can be generated directly from the subtrees in the head table which are 1) with their node identifier as s_i rather than $-s_i$ in the path and 2) either last subtree or not descendant of the subtree mined in the preceding round.* $\square$

For a (conditional) Signed-FPtree with a single path, if a subtree occurs with the identifier $-s_i$, every set related to this subtree will have its SoC be zero. Thus, we have the first condition. The second condition is similar to Optimization 4.3.

According to the FPG-FRACTURE algorithm in Figure 4.6 (b), the three optimization techniques can be employed in Line 15, Line 11 and Line 2 respectively. The detailed algorithm is called Max-FRACTURE and given in Figure 4.9.

Table 4.2: Parameters list.

$ D $	Number of structural deltas	10000
W	Average size of each structural delta	20
L	Average size of subtree sets potentially satisfying minimum FoC	6
P	Number of subtree sets potentially satisfying minimum FoC	2000
M	Mean value of the fraction of subtrees satisfying minimum SoC in a structural delta	0.75
I	Number of changed subtrees	1000
D	Maximum depth of each ancestor relationship	10
F	Maximum fanout of each ancestor relationship	10

4.4 Experimental Results

In this section, we first evaluate the performance of the developed algorithms for mining *FRACTUREs* and *maximal FRACTUREs* by conducting experiments over the synthetically generated structural deltas in Section 4.4.1. Then, we examine the novel knowledge that can be discovered by *FRACTUREs* with experiments on real-life datasets in Section 4.4.2. Algorithms are implemented in the Java programming language. All experiments are conducted on a Pentium IV 2.8GHz PC with 512 MB memory. The operating system is Windows 2000 professional.

4.4.1 Experiments on Synthetic Datasets

We first describe the process of generating synthetic structural deltas of tree versions. Then, we study the performance of the algorithms for mining *FRACTUREs* and *maximal FRACTUREs* respectively.

4.4.1.1 Datasets

In order to evaluate the algorithms of mining *FRACTUREs* and *maximal FRACTUREs*, a structural delta database *SDDb* is required. We implemented a structural delta generator by extending the one that is used to generate transaction datasets in [7]. Parameters of the synthetic structural delta generating process are shown in Table 4.2, with default values in the third column. Four steps are involved in the process of generating synthetic structural deltas.

- Organizing all I subtrees into ancestor relationships with the given maximum depth D and maximum fanout F . At a given node, the number of children is sampled uniformly at random from the range 0 to F . For each child, the process is performed

recursively if the depth of the tree is less than or equal to D or the total number of nodes reaches I .

- Generating subtree sets which potentially satisfy minimum FoC . The size of each potential subtree set is picked from a Poisson distribution with mean equal to L . Subtrees in the first potential set are selected randomly. To model the situation that some *FRACTURES* may have common subtrees, some fraction of subtrees in subsequent potential sets are chosen from the previously generated potential set. We use an exponentially distributed random variable with mean equal to 0.5 to decide this fraction of subtrees.
- Each potential set is associated with a weight, which corresponds to the probability that it will be selected to generate a structural delta. The weight is picked from an exponential distribution with unit mean. Weights of each potential set is normalized so that the sum of weights is 1. The next potential set to be used to generate a structural delta is chosen by tossing an I -sided weighted coin, where the weight of a side is the probability of picking the corresponding potential set. To model the phenomenon that all subtrees in a potential set do not always occur together, we associate each potential set a corruption level, which is obtained from a normal distribution with mean 0.5 and variance 0.1. Then, when adding the potential set to the structural delta, we keep dropping a subtree from the set as long as a uniformly distributed random number between 0 and 1 is less than the corruption level of the potential set. To model the phenomenon that whenever a subtree changed, all its ancestor subtree changed as well, we add all ancestor subtrees of each selected subtrees from selected potential sets to a structural delta as well.
- Assigning DoC to subtrees in each structural delta. In each delta, the number of subtrees whose DoC is no less than the minimum DoC is picked from a Poisson distribution with a specified mean value M . In all experiments, we set the minimum DoC α as 0.15.

4.4.1.2 Methodology & Results for Algorithms of FRACTURE mining

In evaluating the algorithms of mining *FRACTURES*, we carried out four experiments for algorithms: *Apriori-FRACTURE*, *Apriori-FRACTURE-I* (optimized *Apriori-FRACTURE*) and *FPG-FRACTURE*.

CHAPTER 4. FRACTURE MINING

Figure 4.10: Experiment results I.

- **Scalability Study:** We test the scale-up features of all the three algorithms against the number of structural deltas, which is varied from 1K to 30K. The thresholds for FoC β and SoC γ are set as 0.75% and 60% respectively. Figure 4.10 (a) shows the results of the experiment. The performance of *Apriori-FRACTURE* degrades quickly when the number of structural deltas increases while the algorithm *FPG-FRACTURE* scales well with the increasing of the number of structural deltas. The scalability of the optimized *Apriori-FRACTURE-I* is better than the algorithm *Apriori-FRACTURE*.
- **Efficiency Study I:** We compare the execution time of each algorithm to discover *FRACTURE*s by varying the minimum FoC β from 0.35% to 2%. The threshold γ is fixed at 60%. The results are shown in Figure 4.10 (b). Again, the efficiency of the algorithm *FPG-FRACTURE* outperforms the algorithms *Apriori-FRACTURE* and *Apriori-FRACTURE-I*. And the optimized *Apriori-FRACTURE-I* is more efficient than the *Apriori-FRACTURE* although it scans database one more time than the *Apriori-FRACTURE* does.
- **Efficiency Study II:** We measure the execution time of each algorithm to discover *FRACTURE*s by varying the γ from 30% to 80%. The threshold β is fixed at 0.75%. As shown in Figure 4.11 (a), the variation of the minimum SoC does not affect the performance of the algorithms. This is true because none of the algorithms utilize this constraint to prune search space.
- **Optimizing Strategy for *Apriori-FRACTURE*:** We measure the effectiveness of the optimizing strategy for *Apriori-FRACTURE* by comparing the gap between the execution time of the *Apriori-FRACTURE* and the *Apriori-FRACTURE-I* against both

Figure 4.11: Experiment results II.

the number of structural deltas and minimum FoC β . Since *Apriori-FRACTURE-I* tries to gain efficiency by counting SoC for a tightened set of candidate sets, both the size of database and the β may affect its performance. Three sets of data are used: *DBI*(1K), *DBII*(5K) and *DBIII*(10K). The β ranges from 0.5% to 3%. As shown in Figure 4.11 (b), when the size of the dataset turns to be larger and the β turns to be smaller, where the *Apriori-FRACTURE* algorithm cannot perform well, the gap between the *Apriori-FRACTURE-I* and *Apriori-FRACTURE* increases.

- **Effect of Thresholds:** We also conduct experiments to investigate how the thresholds α , β and γ affect the number of discovered *FRACTUREs*. We generate a set of 1000 structural deltas with other default parameters. The default values of the three thresholds are 0.3, 0.1 and 0.6 respectively. The results are shown in the second rows of the three tables in Figure 4.12 (please ignore the third rows of the tables temporarily). We observed that when the threshold (α , β or γ) is increased, the fewer *FRACTURE* patterns will be discovered. Hence, users can adjust the thresholds to discover appropriate number of *FRACTUREs* based on the inverse relationship.

4.4.1.3 Methodology & Results for Algorithms of maximal FRACTURE Mining

In this section, we first carried out experiments to show how the set of *maximal FRACTUREs* is more concise than the complete set of *FRACTUREs*. Subsequently, we evaluated the performance of the algorithm *Max-FRACTURE*.

CHAPTER 4. FRACTURE MINING

Figure 4.12: Experiment results III.

Figure 4.13: Experiment results IV.

- Conciseness of *maximal FRACTUREs*: Firstly, we contrast the size of the set of *maximal FRACTUREs* with the size of the complete set of *FRACTUREs* by adjusting the average depth and fanout of the ancestor relationships. As shown in Figure 4.13 (a), the set of *maximal FRACTUREs* is apparently tighter than the set of *FRACTUREs*. When the average depth and fanout of ancestor relationships are larger, more *FRACTUREs* might be subsumed by their supersets. Hence, the compression ratio turns to be greater.
- Efficiency Study: We evaluate the efficiency of *Max-FRACTURE* with respect to a naïve algorithm which mines *FRACTUREs* using *FPG-FRACTURE* first and then prunes the ones which are not *maximal FRACTUREs*. In addition, since the first optimization applied in *Max-FRACTURE*, Optimization 4.3, can be implemented

CHAPTER 4. FRACTURE MINING

Figure 4.14: Experiment results V.

with *FPG-FRACTURE* independently, we also study the performance of this optimizing strategy. We call the algorithm which applies Optimization 4.3 with *FPG-FRACTURE* *Max-FRACTURE-I*. As the threshold γ does not affect the efficiency of the mining algorithms, as shown in Figure 4.11 (a), we conducted this experiment by varying the threshold β from 2% to 10%. The results are shown in Figure 4.13 (b). When the threshold is smaller, the *Max-FRACTURE* is more efficient. This is because when the threshold is smaller, more *FRACTURES* will be generated. Hence, the naïve algorithm needs to check more *FRACTURES* to verify whether they are maximal or not. It can be observed that the Optimization 4.3 is an effective optimizing strategy because only newly found patterns need to be checked against existing ones to see whether they are non-maximal. Existing patterns do not need to be checked against the new ones.

- Scalability Study I: We test the scale-up features of the naïve algorithm and *Max-FRACTURE* against the number of structural deltas, which is varied from 8K to 80K. Figure 4.14 (a) shows that the *Max-FRACTURE* has the better scalability than the naïve one. Moreover, when the number of structural deltas is larger, the gap between the two algorithms is greater. The reason is that the more subtree sets which possibly are *FRACTURES*, the more efficiency the *Max-FRACTURE* can obtain by filtering non-maximal *FRACTURES* in small sets of *FRACTURES*.
- Scalability Study II: We also observe the scalability of the two algorithms with respect to the number of discovered maximal *FRACTURES*. As presented in Figure 4.14 (b),

when mining the same number of *maximal FRACTURES*, the *Max-FRACTURE* is faster than the naïve algorithm. Furthermore, when the size of the set of *maximal FRACTURES* increases, the *Max-FRACTURE* scales even better.

- **Effect of thresholds:** We conduct experiments to examine the effect of thresholds on the number of *Maximal FRACTURES* with the same dataset which is used to examine the effect of thresholds on the number of *FRACTURES*. The default values of α , β and γ are 0.3, 0.1 and 0.6 respectively. The results are shown in the third rows of the three tables in Figure 4.12. The similar observation can be obtained that the number of *Maximal FRACTURES* decreases when the thresholds increase. Furthermore, we notice that when the thresholds are higher, the gap between the number of *FRACTURES* and the number of *Maximal FRACTURES* is smaller. It means that when the thresholds are higher, there is less redundancy in the set of *FRACTURES*.

4.4.2 Experiments on Real-life Datasets

In this section, we conduct experiments on real-life data sets. Note that, since the performance of the approaches has been evaluated with the experiments on synthetic data, we focus on examining the *knowledge* that can be discovered by *FRACTURES* with the experiments on real-life data sets.

Given a *FRACTURE* pattern, two types of knowledge can be inferred from it. Firstly, we can infer directly that the set of subtrees in a *FRACTURE* frequently undergo structural changes together, which is called *structural knowledge*. Secondly, when applied to tree structures representing objects which are semantically meaningful (e.g., the Object Exchange Model (OEM) representation of objects [3]), we can infer from a *FRACTURE* pattern that the set of objects, represented by the subtrees in the *FRACTURE*, have some hidden association in their semantics as they frequently changed together. We call this type of knowledge as *semantic knowledge*.

We conduct experiments on two sets of real-life data, XML documents of DBLP and Web access log, to investigate the two types of knowledge respectively.

Figure 4.15: DBLP DTD structure.

4.4.2.1 Methodology & Results on DBLP Data

The XML document of DBLP data records the bibliographic information on major computer science journals and proceedings provided by the DBLP server [116]. An XML document can be represented as a tree structure based on the embedded elements. For example, the basic DTD structure of the document *dblp.xml* is shown in Figure 4.15 (a). It can be observed that *dblp.xml* has eight distinct elements under the root: *article*, *inproceedings*, *proceedings*, *book*, *incollection*, *phdthesis*, *masterthesis*, and *www*. With the evolution of *dblp.xml*, new instances of these elements will be added incrementally. Hence, we may discover some *structural evolution associations* from these elements, e.g., new instances of *inproceedings* and *proceedings* may be frequently added together. In order to discover *FRACTUREs* indicating such *structural knowledge*, we reorganized the *dblp.xml* file in the following two steps: (1) all instances of each child element of the root are organized under a newly inserted element (black nodes in Figure 4.15 (b)) corresponding to the original element. The resulted DTD structure is shown in Figure 4.15 (b); (2) historical versions of *dblp.xml* are generated according to the element *year* (gray node in Figure 4.15 (b)) of the instances. For example, a resulted historical version *dblp1970.xml* contains all instances whose element *year* has a value no greater than “1970”.

We totally generated 30 historical versions of *dblp.xml* from year 1971 to year 2000. Experiments are conducted to learn the evolution associations in total 30 years as well as in every 10 years. For experiments on the whole 30 versions, we set the threshold of *DoC* α , the threshold of *FoC* β and the threshold of *SoC* γ as 0.15, 0.2 and 0.4 respectively. For experiments on every 10 versions, the three thresholds are set as 0.15, 0.3 and 0.5 respectively. The

Table 4.3: Results on DBLP data.

1971-2000	1971-1980
dblp/total-proceedings/ dblp/total-article/ dblp/total-phdthesis/ dblp/total-inproceedings/ {dblp/total-inproceedings/, dblp/total-article/}	dblp/total-proceedings/ dblp/total-inproceedings/ dblp/total-article/ {dblp/total-proceedings/, dblp/total-inproceedings/} {dblp/total-inproceedings/, dblp/total-article/}
1981-1990	1991-2000
dblp/total-proceedings dblp/total-phdthesis {dblp/total-proceedings, dblp/total-phdthesis}	dblp/total-proceedings dblp/total-phdthesis dblp/total-masterthesis dblp/total-www {dblp/total-masterthesis, dblp/total-www}

results are shown in Table 4.3. It can be observed that discovered *FRACTURE*s contain not only individual subtrees but also pairs of subtrees. For example, from the *FRACTURE*s discovered from the whole 30 versions, we noticed that the subtrees *dblp/total-inproceedings* and *dblp/total-article* frequently and concurrently changed together. We may infer from the *FRACTURE* that from year 1971 to 2000, new instances of *inproceedings* (conference papers) and *articles* (journal papers) are frequently added together. The usefulness of such *structural knowledge* are presented in the next section with some potential applications.

Although the *FRACTURE*s discovered from the *dblp.xml* indicates some *structural knowledge*, the *semantic associations* are not obvious. For example, it is hard to explain the semantic relation in the *FRACTURE* $\{dblp/total-masterthesis, dblp/total-www\}$ discovered from year 1991 to year 2000. Furthermore, since the depth of *dblp.xml* is small, we only discover *FRACTURE*s from the child elements of the root. To overcome these two deficiencies, we conducted experiments on another set of real-life dataset in the next subsection.

4.4.2.2 Methodology & Results on Web Log Data

As described in Chapter 1, Web access log data can be organized into a hierarchical structure using the way of constructing a directory hierarchy of a Web site. For example, given the access logs of a web user in Figure 4.16 (a), it can be represented as a tree in Figure 4.16 (b). Then, given a sequence of historical versions of a Web log tree representing a web user's historical access logs, we can mine *FRACTURE*s to discover the associated information needs of the user.

We collected the historical Web logs of a particular web user visiting an E-learning Web site [128] during September 2004. For pages accessed by the user in each day, we generate

Figure 4.16: Web log tree and discovered knowledge.

a Web log tree. Since the user did not access the Web site every day, there are totally 23 constructed historical versions of the Web log tree. The maximal depth of the generated Web log trees is 5. We then discover *FRACTUREs* from the subtrees rooted at different levels of the Web log trees.

We conducted the experiments by varying the thresholds, *minimum DoC* α , *minimum FoC* β and *minimum SoC* γ , to find the set of meaningful *FRACTUREs* (Note that, setting loose thresholds results in too many *FRACTUREs* while setting strict thresholds returns *FRACTUREs* containing only individual subtrees). Figure 4.16 (c) shows the *FRACTUREs* when setting the α , β and γ as 0.5, 0.6 and 0.7 respectively. Figure 4.17 shows the two sequences of *DoC* values of “Elearning/images/drseries” and “Elearning/Learning_Objects/x186”. We observed from the figure that the two Web log subtrees changed together for 14 times. Hence, *FoC* of the two subtrees is $14/22 \approx 0.63 > \beta$. Given $\alpha = 0.5$, we observed that *SoC* of the two subtrees is $10/14 \approx 0.71 > \gamma$. Domain experts with the knowledge of the Web site may be able to infer the *semantic knowledge* from the *FRACTURE*. For example, the web user may frequently visited different learning objects under the node “x186”, where most of the learning objects invoke various images under the node “drseries”.

4.5 Applications of FRACTUREs

FRACTUREs can be applied in different applications where data are evolutionary and can be modelled as tree structures. In this section, we enumerate some applications of *FRACTURE* patterns based on the different knowledge that can be inferred from *FRACTUREs*.

Figure 4.17: DoC values of Web log subtees.

Based on the *structural knowledge* inferred from *FRACTURE* patterns, *FRACTURES* can be used in the following applications:

- **Native XML Storage.** Native XML storage usually views an XML document as a tree and partitions the XML tree into distinct records containing disjoint connected subtrees [121] [82]. These distinct records are then stored in disk pages. Due to the dynamic nature of XML documents, XML records need to be updated according to changes. Then the knowledge of *evolution association* inferred from *FRACTURES* can be used as a guide to partition the XML tree. For example, if two subtrees, s_1 and s_2 , are discovered as a *FRACTURE*, then we can store the two subtrees in the same disk page or adjacent pages. Thus, the updating process can be expected to be more efficient in locating the positions of changed subtrees as they frequently change together and they locate adjacent to each other.
- **XML Query Caching.** XML queries can be represented as tree structures as well. Caching XML queries has been recognized as an orthogonal algorithm to improve the efficiency of XML query processing [157]. As shown in the experiment on the real-life data about DBLP XML documents, *FRACTURE* patterns can be used to discover XML paths that are associated in the increase of their *counts* (or *supports*). For example, the pattern $\{dblp/total-proceedings, dblp/total-articles\}$ indicates that the counts of the two paths, $dblp/proceedings$ and $dblp/articles$, frequently increase together. Hence, by mining *FRACTURE* patterns from XML users' query logs, we

can infer that when some query paths are getting popular, some other paths may probably be the interests of users as well. Thus, once a particular path query is issued by a user, both the *weight* of the path and the *weight* of its associated path are promoted so that we delay the eviction of the path and its associated path (assuming the the paths are cached already).

Based on the *semantic knowledge* inferred from *FRACTUREs*, *FRACTUREs* can be used in the following applications:

- Web Site Maintenance.** Nowadays, Web sites are becoming important assets of companies, whose business strongly depend on their presence on the internet. Consequently, maintenance arises as a crucial issue to the production of high quality and reliable Web sites. One of the challenges of maintaining the consistency of a Web site is that the site is very likely written by different authors. Thus, new authors may not have the knowledge of semantically related Web pages such that when some of the pages are modified, the others should be updated accordingly, particularly, when the size of the Web site is getting larger and larger. *FRACTURE* patterns can be used here to address this problem. For example, Web pages in a site can be organized into a *directory hierarchy* as explained in Chapter 3. Mining *FRACTURE* patterns from the historical versions of the directory hierarchy, we can discover which Web directories are frequently changed together. Then, we can infer that such Web directories are probably semantically related. Thus, when some of them are updated, the others should be checked to see whether they need to be modified as well to keep the content of the Web site consistent.
- Web User Clustering.** Discovering clusters of Web users that exhibit similar information needs, e.g., users that access similar pages, is an important topic in Web usage mining. By analyzing the characteristics of the clusters, web designers may understand the users better and thus can provide more suitable, customized services to the users. Existing work on Web user clustering considers the snapshots of Web user sessions. However, Web user sessions are dynamic. The knowledge obtained by clustering Web users based on their snapshot Web sessions may be limitedly useful. For example, users accessing similar pages at this time point may access different pages

at the next time point. Hence, it is interesting to cluster users based on their historical Web sessions such that clusters of users exhibit similar variations of information needs. Since *FRACTURE* patterns can capture the evolutionary characteristics of the dynamic Web session trees, they can be used as the features to cluster Web users. We will describe a Web user clustering algorithm based on *FRACTURE* patterns in the next chapter.

4.6 Summary

In this chapter, we have introduced a new pattern *FRACTURE*. To the best of knowledge, this is the first work which discovers evolution associations from the changes to tree structures. Basically, a *FRACTURE* pattern is a set of subtrees which not only change together frequently but also undergo significant changes together frequently.

A set of evolution metrics, *DoC*, *FoC* and *SoC*, which measure the degree of change, the frequency of change and the significance of change respectively for individual subtrees as well as subtree sets are defined to evaluate the interestingness of *FRACTURE* patterns. Two algorithms, *Apriori-FRACTURE* and *FPG-FRACTURE*, were designed to discover the *FRACTURES* with different mining strategies. In order to make the resulting patterns concise, we further defined the notion of *maximal FRACTURE*. We then optimized the algorithm *FPG-FRACTURE* to handle the problem of *maximal FRACTURE* mining.

We have conducted experiments on synthetic data sets to study the performance of developed algorithms. The experimental results demonstrated that the proposed data mining algorithms can discover *FRACTURES* with certain efficiency and scalability, the optimizing strategies work effectively in improving the performance of the algorithms, and the modified version of *FPG-FRACTURE* can discover the set of *maximal FRACTURE* patterns efficiently. We also conducted experiments on real-life data sets to investigate the knowledge that can be discovered by *FRACTURE* patterns. Based on the different types of knowledge inferred from *FRACTURES*, we also discussed the potential applications of *FRACTURE* patterns in Web problems.

Chapter 5

COWES: FRACTURE-based Clustering

FRACTURE patterns extract evolution associations of substructures of a tree. Hence, they reflect the evolutionary characteristics of the tree from certain aspect. Consequently, they are beneficial in applications where the evolutionary characteristics of a tree structure is important. In this chapter, we introduce a new Web user clustering algorithm, *COWES* (Clustering Of Web users based on historical wEb Session), to show the usefulness of *FRACTURE* patterns in clustering tree-structured data which are evolutionary. A preliminary version of this chapter appeared in our previous publication [36].

Clustering Web users is one of the most important research topics in Web usage mining. Web user clustering aims to discover clusters of users that exhibit similar information needs, e.g., users that access similar pages. By analyzing the characteristics of the clusters, Web designers may understand the users better and thus can provide more suitable, customized services to the users [152]. Existing Web user clustering methods [152, 146] cluster users based on different features of Web usage data, such as the pages accessed by users, the visiting time spent on pages, and the order between visited pages etc. One of the notable features of Web usage data is that Web usage data are *evolutionary*. Web users may vary their information needs over time. However, to the best of our knowledge, none of existing Web user clustering methods take into account this feature to discover clusters of users that exhibit similar variation patterns in their information needs. In this chapter, we present the *COWES* algorithm, which clusters Web users based on the evolution of their usage data. That is, users sharing similar evolutionary characteristics of their Web usage data will be grouped together. Particularly, we use the *FRACTURE* patterns, mined from historical

Web user sessions, as the clustering feature to capture certain evolutionary characteristics of Web usage data¹. Obviously, *COWES* discovers novel clusters of Web users which cannot be discovered by existing approaches. As exemplified later, discovered knowledge are useful in applications such as intelligent Web advertisement and proxy cache management etc. The content in this chapter appeared in our previous publication [86].

In the rest of this chapter, we begin with an example to illustrate the motivation of *COWES*. In Section 5.2, we discuss the clustering feature used by *COWES*. The similarity metric and the clustering framework of *COWES* are described in Sections 5.3 and 5.4 respectively. We evaluate the performance of *COWES* in Section 5.5 and conclude this chapter with a summary and some discussions in Section 5.6.

5.1 Introduction of COWES

Before presenting our motivating example, we briefly introduce existing Web user clustering techniques. Existing Web user clustering consists of three phases: *data preparation*, *cluster discovery*, and *cluster analysis*. Since the last phase is application-dependent, we describe the first two phases as follows.

In the first phase, *Web sessions* of users are extracted from the Web server log by using some user identification and session identification techniques [47]. A Web session, which is an episode of interaction between a Web user and the Web server, consists of pages visited by a user in the episode [62]. For example, Figure 5.1 (a) shows four requests from one session. The first line means that the user at *foo.cs.ntu.edu* accessed the page *www.uow.edu/sce/Jeffrey/pub.html* at 10 : 30 : 05 on January 01, 2005. The number 200 indicates that the request is successful and the number 3027 is the size of the requested page in bytes.

In the second phase, clustering techniques are applied to generate user clusters. For example, consider the three users, u_1 , u_2 and u_3 , and their Web session data about visiting the Web site rooted at the home page a in a particular time period p_1 , as shown in the left part of Figure 5.2 (a) (Note that, only accessed pages are presented in the figure, where other information in Web sessions are ignored). Existing Web user clustering methods which use commonly accessed Web pages as the clustering feature, such as the work in [62], will group the three users together as their Web sessions share common pages.

¹Alternative patterns which capture other evolutionary characteristics of Web usage data can be used as clustering features as well.

Figure 5.1: Web session and document hierarchy.

5.1.1 Motivating Example

Existing Web user clustering methods cluster users based on Web sessions collected in a certain time period. However, Web usage data is *evolutionary* in nature. The information needs of users may vary over time. Consequently, pages accessed by users in different time periods may be different as well. For example, Figures 5.2 (b), (c) and (d) (left parts) show another three sets of Web sessions of users u_1 , u_2 and u_3 visiting the Web site at subsequent time periods p_2 , p_3 and p_4 respectively. It can be observed that users accessed different pages in different periods.

Such evolutionary nature of the Web usage data poses both challenges and opportunities to the Web user clustering. In particular, the evolutionary nature of Web usage data leads to the following two challenging problems:

- **Maintenance of Web user clustering results:** Take the Web sessions in Figure 5.2 as an example. Although users u_1 , u_2 and u_3 accessed similar Web pages in the period p_1 , they visited different pages in p_2 and beyond. Hence, the clustering results, generated by existing algorithms in a particular time period, have to be updated along with the changes to Web usage data. This requires development of efficient incremental Web user clustering techniques.
- **Discovery of novel Web user clusters:** Considering the evolutionary characteristics of Web usage data, interesting and novel Web user clusters might be discovered. For example, clusters of users that exhibit similar characteristics in the evolution of their usage data may be discovered. Examples of such clusters will be illustrated later.

CHAPTER 5. COWES: FRACTURE-BASED CLUSTERING

Figure 5.2: Historical Web sessions.

The first problem has been addressed by some incremental clustering algorithms proposed in the literature [148][127]. Our focus here is the second problem – discovering novel user clusters based on the evolutionary characteristics of Web usage data.

We now show the novel clusters that can be discovered based on evolutionary characteristics of Web usage data with the motivating example in Figure 5.2. Pages accessed in a Web session can be organized into a hierarchical structure, called Web session tree, based on the URLs of the pages [62]. For example, Figure 5.1 (b) is the Web session tree constructed for the pages in the Web session shown in Figure 5.1 (a). A Web session tree represents the information needs of a user. Then, the sequences of historical Web sessions of Web users u_1 , u_2 and u_3 are represented as sequences of Web session trees in the right part of Figure 5.2. Each gray node in the figure represents a page that was not accessed

in the next Web session and each black node denotes a page that was not accessed in the previous session. Obviously, changes to the structures of Web session trees of a user, i.e., gray and black nodes, reflect the variation of the user's information needs. Observing the sequence of Web session trees of user u_1 , we noticed that u_1 frequently varies his information needs in the Web session subtree a/b^2 , which is highlighted by dotted lines in Figure 5.2 (*right part*). Similarly, users u_2 and u_3 frequently vary their information needs in the Web session subtrees a/c and a/d . Hence, based on such evolutionary characteristics of their usage data, our clustering algorithm, *COWES*, will cluster u_2 and u_3 together while leave u_1 in another cluster. Note that, such clusters cannot be discovered by existing approaches based on the Web usage data collected in a particular period. Knowledge can be inferred from such clusters that users in the same cluster exhibit similar variation patterns in their information needs.

5.1.2 Applications

Novel Web user clusters generated based on evolutionary characteristics of Web usage data are useful at least in the following two applications:

- **Intelligent Web Advertisement:** 99% of all Web sites offer standard banner advertisements [23]. This shows the importance of this form of online advertising. The most commonly used pricing schemes in banner advertisements are the *cost-per-thousand impressions* (CPM) model and the *cost-per-click* (CPC) model, where the cost is associated with the amount of exposure of the advertisement and the times the advertisement banner is clicked on respectively. One of the ways to maximize revenues for the party who owns the advertising space is to design intelligent techniques for the selection of an appropriate set of advertisements to display in appropriate Web pages. Web user clusters generated by *COWES* can be beneficial for designing intelligent advertisement placement strategies. For example, analyzing the clusters generated by *COWES* in Figure 5.2, we know that the variation patterns of information needs of u_1 is different from those of users u_2 and u_3 . Suppose an advertisement is targeting all the three users. Although all of them accessed the page $a/b/e$ in the current period p_4 , the advertisement banner should be put in the page a/b instead of $a/b/e$ for user u_1 because he frequently changes his information needs under a/b .

²Similar to Chapter 4, we use the root path arriving at the root of a Web session subtree to denote the subtree.

Figure 5.3: Overview of COWES.

- Proxy Cache Management:** Web caching is an interesting problem in Web research area [25] [158] as Web caches can reduce not only network traffic and but also downloading latency. Because of the limited size of cache region, it is important to design effective replacement strategies to maximize hit rates. One of the frequently used replacement strategies is LRU (Least-Recently-Used), which assigns priorities to the most recently accessed pages. Web user clusters generated by *COWES* can be used with LRU to manage the caching region more optimally. For example, analyzing the clusters generated by *COWES* in Figure 5.2, we know that users u_2 and u_3 frequently vary their information needs under a/c and a/d together. That is, once user u_2 (u_3) varies information needs under a/c , he probably will vary his information needs under a/d as well. Thus, if u_2 (u_3) accesses different pages under a/c in the next time period, such as p_5 , we can degrade the priority of cached pages under a/d so that their eviction from the cache can be hasten.

5.1.3 Overview of COWES

The overview of *COWES* is presented in Figure 5.3. The input is a collection of Web users $\{u_1, u_2, \dots, u_n\}$. Each user is associated with a sequence of historical Web sessions that are collected at a sequence of time periods $\langle p_1, p_2, \dots, p_m \rangle$ with some particular time granularity. Each sequence of Web sessions of user u_i can be represented as a sequence of Web session trees $\langle S_i^{p_1}, S_i^{p_2}, \dots, S_i^{p_m} \rangle$.

The first step is to extract *FRACTUREs* from Web session trees. As discussed before, we aim to cluster users based on the evolutionary characteristics of their usage data. Since *FRACTURE*, as presented in the previous chapter, is a type of pattern that can capture the evolutionary characteristics of a tree from certain aspect, we mine *FRACTUREs* from historical Web session trees for each user and use *FRACTUREs* as the clustering feature. Then, each Web user can be represented as a set of *FRACTUREs* $u_i = \{X_i^1, X_i^2, \dots, X_i^k\}$.

Given a set of users represented by the *FRACTUREs* mined from their historical Web session trees, the second step is to measure the similarity between users based on their *FRACTUREs*. We define a similarity measure which considers the proximity of shared *FRACTUREs* in terms of not only their elements (e.g., subtrees contained by *FRACTUREs*) but also their strength (e.g., the *FoC* and *SoC* values of *FRACTUREs*). After computing the similarity between each pairs of users, a similarity matrix of users can be generated.

Finally, given a similarity matrix of Web users, existing appropriate clustering algorithms, such as partitional clustering algorithms or agglomerative clustering algorithms, can be performed on the similarity matrix to generate the clusters of Web users.

5.2 Clustering Feature

In this section, we discuss the clustering feature used by *COWES*. That is, the *FRACTUREs* discovered from historical Web session trees

As shown in [62], pages in a Web session can be organized into a hierarchical structure based on their URLs, which is called a *Web session tree*. Formally, a Web session tree is an unordered tree $S = \langle N, E, L \rangle$, where N is the set of nodes where a leaf node represents a Web page corresponding to a file in the Web server and a non-leaf node represents a Web page corresponding to a directory in the server, E is the set of edges where each edge from a parent node to a child node represents the consisting-of relationship between the corresponding pages, and L is the set of labels on the nodes where each leaf node label corresponds to a filename in the Web server and each non-leaf node label corresponds to a directory name in the server. Particularly, a node r , $r \in N$, is the root of the tree which represents the home page of a Web site. Accordingly, a tree $s_i = \langle N_i, E_i, L_i \rangle$ is a Web session subtree, denoted as $s_i \prec S$, if s_i is a bottom-up subtree of S .

Figure 5.4: Historical Web session trees of u_2 .

Given two versions of a Web session tree, we are interested in the structural changes which reflect the variation of the user's information needs. Particularly, we consider the following two basic operations defined in Chapter 4: $INS(x(name), y)$ and $DEL(x)$. The former creates a new leaf node x , with node name "name" as a child node of node y in a Web session tree. This operation corresponds to user's accessing of a new page x with filename "name" in the directory y . The latter removes leaf node x from a Web session tree, which means that the user no longer visits the page x .

Then, given a sequence of historical Web session trees of a Web user, other definitions related to *FRACTURE* mining, such as *DoC*, *FoC* and *SoC*, can be applied here directly. We omit redefining *FRACTURE* in the context of historical Web session trees. We use the following example to show *FRACTUREs* mined from Web session trees.

Example 5.1 The sequence of Web session trees of user u_2 in Figure 5.2 is redrawn in Figure 5.4. Let the subtree a/c be subtree s . The *DoC* of s in the first two versions is $DoC(s, p_1, p_2) = 1/3$. Similarly, $DoC(s, p_2, p_3) = 2/4$. Let X be a set of two subtrees, $X = \{a/c, a/d\}$, $FoC(X) = 3/3 = 1$ as the two subtrees changed together in all the three subsequent versions (after the first version). Suppose the threshold of *DoC* α be $1/2$. $SoC_{1/2}(X) = 2/3$ because the two subtrees changed together in three subsequent versions, while they both changed significantly together in the last two versions. Then, suppose both the thresholds β and γ are $2/3$. The subtree set X is a *FRACTURE* of user u_2 . ■

For each user, a set of *FRACTUREs* can be mined from his historical Web session trees by using the data mining algorithms proposed in the previous chapter, *Apriori-FRACTURE* and *FPG-FRACTURE*, with respect to the set of given thresholds. Consider the Web users and their historical Web session trees in Figure 5.2. Suppose the thresholds α , β and γ are 0.5, 0.6 and 0.6 respectively. Table 5.1 shows the discovered *FRACTUREs* of each user. We aim to cluster users based on their *FRACTUREs*.

³In this column, we use the subscript to denote the user and the superscript to denote the pattern.

Table 5.1: *FRACTUREs* discovered from Web session trees.

<i>User ID</i>	<i>FRACTURE ID</i> ³	<i>FRACTURE</i>	<i>FoC</i>	<i>SoC</i>
u_1	X_1^1	{a/b}	1.00	1.00
u_1	X_1^2	{a/c/i}	0.66	1.00
u_1	X_1^3	{a/b, a/c/i}	0.66	1.00
u_2	X_2^1	{a/c}	1.00	0.66
u_2	X_2^2	{a/d}	1.00	0.66
u_2	X_2^3	{a/c, a/d}	1.00	0.66
u_3	X_3^1	{a/c}	1.00	1.00
u_3	X_3^2	{a/d}	1.00	0.66
u_3	X_3^3	{a/c, a/d}	1.00	0.66

5.3 Similarity Measures

After representing each user as a set of *FRACTUREs*, we need to measure the proximity of users. In this section, we present the similarity measure *User Similarity*, which is defined based on the similarity measure *FRACTURE Similarity*.

Given two users represented as two sets of *FRACTUREs* respectively, they may share some of their *FRACTUREs*. Then, the complete set of features is the combination of the two set of *FRACTUREs*. For example, suppose user u_1 has a set of *FRACTUREs* $\{\{a/b, a/c\}, \{a/b/d, a/c\}\}$ and user u_2 has a set of *FRACTUREs* $\{\{a/b, a/c\}, \{a/b/d, a/e\}\}$. The complete set of features are $\{\{a/b, a/c\}, \{a/b/d, a/c\}, \{a/b/d, a/e\}\}$. Users u_1 and u_2 share the first *FRACTURE* only. Suppose a similarity measure *FRACTURE Similarity* (*FS*) can be computed on a pair of shared *FRACTUREs*. Then, according to the algorithm introduced in Section 2.4.6 in [133], the measure *User Similarity* (*US*) can be computed by the following three steps:

- For the k^{th} *FRACTURE*, compute a similarity, $FS(u_1, u_2)$, in the range $[0, 1]$.
- Define an indicator variable δ_k , such that
$$\delta_k = \begin{cases} 0, & \text{if one of the users has a missing value for the } k^{th} \text{ FRACTURE} \\ 1, & \text{otherwise} \end{cases}$$
- Compute the overall similarity between the two users using the following formula:

$$US(u_1, u_2) = \frac{\sum_{k=1}^n \delta_k FS_k(u_1, u_2)}{\sum_{k=1}^n \delta_k} \quad (\text{Eq. 5.1})$$

However, in our application, we observed that such a similarity measure has the following two limitations.

Firstly, the denominator in Eq. 5.1 counts only the number of shared *FRACTUREs*. Actually, the number of *FRACTUREs* which are not shared by the two users affects the proximity of the users as well. Consider two users who have m and n *FRACTUREs* respectively ($m \gg 1$ and $n \gg 1$) and share only one *FRACTURE* with FS as 1. Then, according to Eq. 5.1, the two users are completely similar even if they have so many different *FRACTUREs*. Hence, we address this problem by revising the denominator in Eq. 5.1 as $\frac{m+n}{2}$ so that the total number of *FRACTUREs* of the two users are considered.

Secondly, the numerator in Eq. 5.1 treats all shared *FRACTUREs* equally. However, if there exist different types of shared *FRACTUREs* (e.g., we will define two types of shared *FRACTUREs*, *Identical FRACTURE* and *Approximate FRACTURE*, in the next subsection), which contribute different importance to the proximity of the users, the numerator of Eq. 5.1 should be modified by introducing weights to different types of shared *FRACTUREs*. Consequently, the numerator of Eq. 5.1 becomes:

$$\sum_{k=1}^n \omega_k \delta_k FS_k(u_1, u_2) \quad (\text{Eq. 5.2})$$

In the following, we first introduce the two types of shared *FRACTUREs* and the way to weigh their contribution to the proximity of users (ω). Then, we define the similarity measure FS . Finally, the similarity measure US is formally defined.

5.3.1 Types of Shared FRACTUREs

Recall that a *FRACTURE* discovered from historical Web session trees is a set of Web session subtrees (Web directories). If two *FRACTUREs* have the exactly same Web session subtrees, they certainly contribute to the similarity of the two users. We refer to such type of *FRACTUREs* shared by two users as *Identical FRACTUREs*.

Definition 5.1 (Identical FRACTUREs) Let $X_1 = \{s_1, \dots, s_m\}$, $X_2 = \{s_1, \dots, s_n\}$ be two *FRACTUREs*. Let $L(s)$ be the root path arriving at the root of Web session subtree s . X_1 and X_2 is a pair of *Identical FRACTUREs*, denoted as $X_1 = X_2$, if 1) $m = n$, and 2) $\forall i(1 \leq i \leq m)$, $\exists j(1 \leq j \leq n)$ s.t. $L(s_i) = L(s_j)$ and 3) $\forall j(1 \leq j \leq n)$, $\exists i(1 \leq i \leq m)$ s.t. $L(s_j) = L(s_i)$. $\square$

That is, two *FRACTURES* are *Identical FRACTURES* if there is a one-to-one mapping between the subtrees of the two *FRACTURES* and the corresponding subtrees have same labels. For example, consider the *FRACTURES* in Table 5.1. The two *FRACTURES* X_2^1 and X_3^1 is a pair of *Identical FRACTURES*. The two users u_2 and u_3 share another two pairs of *Identical FRACTURES*, $X_2^2 = X_3^2$ and $X_2^3 = X_3^3$.

It was observed in [62] that Web directories (pages) are not randomly organized. Normally, a Web directory (page) is stored in another Web directory which semantically contains it. Thus, Web session subtrees, whose labels have prefix relationships, are semantically related. For example, given two Web session subtrees s_1 and s_2 . Suppose $L(s_1) = a/b$ and $L(s_2) = a/b/c$. Since $L(s_1)$ is a prefix of $L(s_2)$, the two Web session trees are semantically related, e.g., s_1 semantically contains s_2 . Then, a pair of *FRACTURES* whose corresponding subtrees are semantically related contribute to the similarity of users as well. Hence, besides *Identical FRACTURES*, we define another type of shared *FRACTURES*, which is called *Approximate FRACTURES*, as follows.

Definition 5.2 (Approximate FRACTURES) Let $X_1 = \{s_1, \dots, s_m\}$ and $X_2 = \{s_1, \dots, s_n\}$ be two *FRACTURES*. Let $L(s)$ be the root path arriving at the root of Web session subtree s . X_1 and X_2 is a pair of *Approximate FRACTURES*, denoted as $X_1 \approx X_2$, if 1) $m = n$, and 2) $\forall i(1 \leq i \leq m), \exists j(1 \leq j \leq n)$ s.t. $L(s_i)$ is a prefix of $L(s_j)$ or $L(s_j)$ is a prefix of $L(s_i)$ and 3) $\forall j(1 \leq j \leq n), \exists i(1 \leq i \leq m)$ s.t. $L(s_j)$ is a prefix of $L(s_i)$ or $L(s_i)$ is a prefix of $L(s_j)$. $\square$

That is, two *FRACTURES* are *Approximate FRACTURES* if there is a one-to-one mapping between the subtrees of the two *FRACTURES* and labels of corresponding subtrees have prefix relationships. For example, the two *FRACTURES* X_1^2 and X_2^1 in Table 5.1 is a pair of *Approximate FRACTURES*.

We now explain how to assign different weights to different types of shared *FRACTURES*. Given two Web session subtrees whose labels have prefix relationship, we define *Prefix Level* to measure the distance between the two subtrees.

Definition 5.3 (Prefix Level) Let s_i and s_j be two Web session subtrees s.t. $L(s_i)$ is a prefix of $L(s_j)$. The length of the path $L(s_i)$, denoted as $|L(s_i)|$, is the number of edges traversed by the path. Then, the *Prefix Level* between s_i and s_j , denoted as $PL(s_i, s_j)$, can be computed as

$$PL(s_i, s_j) = |L(s_j)| - |L(s_i)| \quad \square$$

For example, consider the two subtrees in the pair of *Approximate FRACTURES*, $X_1^2 \approx X_2^1$, in Table 5.1 again. Since the labels of the subtrees are $a/c/i$ and a/c , the *Prefix Level* between the two subtrees is 1. Note that, *Identical FRACTURES* can be viewed as a special case of *Approximate FRACTURES* from the perspective that the *Prefix Level* between each pair of corresponding subtrees of *Identical FRACTURES* is 0.

According to the observation in [62], the less the *Prefix Level* between two Web session subtrees, the more semantically related the two subtrees. Given a pair of *Identical/Approximate FRACTURES*, the sum of the *Prefix Levels* between corresponding subtrees can be computed, which takes value in $[0, +\infty)$. Particularly, the sum is zero for a pair of *Identical FRACTURES*. Obviously, the less the sum, the more importance the shared *FRACTURES* contribute to the similarity of users. Thus, we use some monotonic decreasing function³ to transform the sum to some weight value, ranging between $(0, 1]$. Then, the weight assigned to a pair of shared *FRACTURES* can be computed as follows.

Definition 5.4 (Weight) Given a pair of shared *FRACTURES* $X_1 = \{s_1^1, \dots, s_1^m\}$ and $X_2 = \{s_2^1, \dots, s_2^m\}$ s.t. $PL(s_1^i, s_2^i) \geq 0$ ($1 \leq i \leq m$). The weight assigned to the pair of *FRACTURES*, denoted as $\omega(X_1, X_2)$, is defined as,

$$\omega(X_1, X_2) = e^{-\sum_{i=1}^m PL(s_1^i, s_2^i)} \quad \square$$

Thus, when the pair of shared *FRACTURES* are *Identical FRACTURES*, they are assigned the weight 1. The more similar the two *FRACTURES* in terms of their subtrees, the more weight will be assigned. For example, the weight assigned to the *Approximate FRACTURES*, $X_1^2 \approx X_2^1$, in Table 5.1 can be computed as follows. $\omega(X_1^2, X_2^1) = e^{-1} \approx 0.37$.

5.3.2 FRACTURE Similarity

A pair of shared *FRACTURES* may have different *FoC* and *SoC* values. Hence, we define the metric *FRACTURE Similarity (FS)* to measure the similarity of shared *FRACTURES* in terms of their *FoC* and *SoC* values.

Note that, although *SoC* is defined based on *FoC*, the correlation between *FoC* and *SoC* is uncertain. That is, for different data set, the correlation between the two variables

³A variety of monotonic decreasing functions can be used here. For example, let Σ be the overall prefix levels between a pair of shared *FRACTURES* and ω be the weight. Then functions $\omega = \frac{1}{a\Sigma+1}$ and $\omega = a^{-\Sigma}$ can be used, where $a > 0$. The goodness of a function depends on the data. We evaluate the performance of different functions in Section 5.5.

is different as well. Hence, we adopt the *Mahalanobis distance* to measure the distance between the *FoC* and *SoC* values of a pair of *Identical/Approximate FRACTURES* X_1 and X_2 .

$$d(X_1, X_2) = (X_1 - X_2)^T \Sigma^{-1} (X_1 - X_2) = \begin{pmatrix} FoC(X_1) - FoC(X_2) & SoC(X_1) - SoC(X_2) \end{pmatrix} \Sigma^{-1} \begin{pmatrix} FoC(X_1) - FoC(X_2) \\ SoC(X_1) - SoC(X_2) \end{pmatrix} \quad (\text{Eq. 5.3})$$

where Σ^{-1} is the inverse of the covariance matrix of the complete set of *FoC* and *SoC* values of all users. Particularly, when *FoC* and *SoC* are uncorrelated for the data set, the *Mahalanobis distance* is actually the *Euclidean distance* between the *FoC* and *SoC* values of a pair of shared *FRACTURES* X_1 and X_2 .

$$d(X_1, X_2) = \sqrt{(FoC(X_1) - FoC(X_2))^2 + (SoC(X_1) - SoC(X_2))^2} \quad (\text{Eq. 5.4})$$

Then, in order to define the *FRACTURE Similarity* based on the distance between *FoC* and *SoC* values of *shared FRACTURES*, we use a monotonic decreasing function to convert the distance to a similarity⁴.

Definition 5.5 (FRACTURE Similarity) Let X_1 and X_2 be a pair of *Identical/ Approximate FRACTURES*. Suppose $d(X_1, X_2)$ is the distance between the *FoC* and *SoC* values of X_1 and X_2 . The *FRACTURE Similarity* of the pair of *FRACTURES*, denoted as $FS(X_1, X_2)$, is defined as,

$$FS(X_1, X_2) = \frac{1}{1 + d(X_1, X_2)} \quad \square$$

Since the *Mahalanobis distance* takes values in $[0, +\infty)$, *FS* ranges in $(0, 1]$. The closer the *FoC* and *SoC* values of the two *FRACTURES*, the higher the *FS*. *FS* have the following two properties.

Property 5.1 (Positivity Property) $FS(X_1, X_2) = 1$ only if $X_1 = X_2$ ($0 < FS \leq 1$).

That is, given two *FRACTURES* X_1 and X_2 , $FS(X_1, X_2) = 1$ only if $FoC(X_1) = FoC(X_2)$ and $SoC(X_1) = SoC(X_2)$. Since *Mahalanobis distance* is a metric, which has the positivity property, $d(X_1, X_2) = 0$ only if $FoC(X_1) = FoC(X_2)$ and $SoC(X_1) = SoC(X_2)$. As $FS = \frac{1}{1+d(X_1, X_2)}$ is a monotonic function, $FS = 1$ only if $d = 0$. The property holds for *FS*.

⁴Again, other monotonic decreasing functions can be used as well. The performance of some alternative functions is evaluated in Section 5.5.

Property 5.2 (Symmetry Property) $FS(X_1, X_2) = FS(X_2, X_1)$ for all X_1 and X_2 .

Again, since *Mahalanobis distance* is a metric, $d(X_1, X_2) = d(X_2, X_1)$. Thus, $FS(X_1, X_2) = \frac{1}{1+d(X_1, X_2)} = \frac{1}{1+d(X_2, X_1)} = FS(X_2, X_1)$.

Example 5.2 Consider the *FoC* and *SoC* of the *FRACTUREs* in Table 5.1. The covariance matrix between the complete set of *FoC* and *SoC* is $\Sigma = \begin{pmatrix} 0.0225 & -0.0161 \\ -0.0161 & 0.0321 \end{pmatrix}$. For the pair of *Identical FRACTUREs*, $X_2^3 = X_3^3$, $d(X_2^3, X_3^3) = 0$ and $FS(X_2^3, X_3^3) = 1$, since the two *FRACTUREs* have the same *FoC* and *SoC* values. While, for the pair of *Approximate FRACTUREs*, $X_1^2 \approx X_2^1$, $d(X_1^2, X_2^1) = \begin{pmatrix} -0.34 & 0.34 \end{pmatrix} \Sigma^{-1} \begin{pmatrix} -0.34 \\ 0.34 \end{pmatrix} = 5.6$. Hence, $FS(X_1^2, X_2^1) = \frac{1}{1+5.6} \approx 0.15$. ■

5.3.3 User Similarity

After discussing the two limitations of the Eq 5.1, the similarity measure FS , and the way to assign different weights to different types of shared *FRACTUREs*, the *User Similarity* can be formally defined as follows.

Definition 5.6 (User Similarity) Let $u_1 = \{X_1^1, X_1^2, \dots, X_1^m\}$ and $u_2 = \{X_2^1, X_2^2, \dots, X_2^n\}$ be two Web users that are represented as two sets of *FRACTUREs*. Suppose there exists $k (0 \leq k \leq m \leq n)$ s.t. $X_1^1 = X_2^1$ or $X_1^1 \approx X_2^1$, ..., $X_1^k = X_2^k$ or $X_1^k \approx X_2^k$. The Similarity of Users, denoted as $US(u_1, u_2)$, is defined as,

$$US(u_1, u_2) = \frac{\sum_{i=1}^k \omega(X_1^i, X_2^i) FS(X_1^i, X_2^i)}{\frac{m+n}{2}} \quad \square$$

If each pair of corresponding *FRACTUREs* of the two users are *Identical FRACTUREs* with the FS value 1, then US has the maximum value of 1. Otherwise, if the two users share no *FRACTURE*, US is 0. The positivity property and the symmetry property hold for US as well.

Property 5.3 $US(u_1, u_2) = 1$ only if $u_1 = u_2$. ($0 \leq US \leq 1$).

Firstly, if $u_1 = u_2$, which means u_1 and u_2 share all of their *FRACTUREs* and all shared *FRACTUREs* are *Identical FRACTUREs* with FS as 1, then it is obvious $US(u_1, u_2) = 1$. Secondly, if $u_1 \neq u_2$, there exists at least one *FRACTURE* of one user which cannot be shared by the other user. Thus, for k, m, n in Definition 5.6, $k < m$ or $k < n$. Let's suppose

$k < m$. k is either less than n or equal to n . If $k < n$, $k + k < m + n$, $k < \frac{m+n}{2}$, $US < 1$, that is $US \neq 1$. If $k = n$, similarly, $k + k < m + k$, $2k < m + n$, and $US \neq 1$. Thus, the positivity property holds for US .

Property 5.4 (Symmetry Property) $US(u_1, u_2) = US(u_2, u_1)$ for all u_1 and u_2 .

According to Definition 5.4, $\omega(X_1^i, X_2^i) = \omega(X_2^i, X_1^i)$. Moreover, it is shown above that FS has the symmetry property. Hence, $US(u_1, u_2) = US(u_2, u_1)$.

Example 5.3 Consider the users and their *FRACTURES* in Table 5.1 again. Since the two users u_2 and u_3 share three pairs of *Identical FRACTURES*, where each pair has the FS value 1, $US(u_2, u_3) = \frac{3}{\frac{3+3}{2}} = 1$. While, users u_1 and u_2 share a pair of *Approximate FRACTURES*, which has the FS 0.15 and assigned the weight 0.37. $US(u_1, u_2) = \frac{0.37 \times 0.15}{\frac{3+3}{2}} = 0.0185$. ■

5.4 Framework of COWES

In this section, we describe the framework of *COWES*. Given a collection of Web users, where each user is associated with a sequence of his historical Web sessions, *COWES* generates the clusters of users in the follows phases:

- Phase I. From the historical Web sessions of each user, we extract a set of *FRACTURES*, which will be treated as a vector of features for clustering.
- Phase II. Compute the similarity matrix between pairs of Web users based on the defined similarity measure US .
- Phase III. Perform clustering on the generated similarity matrix of Web users.

In Chapter 4, we have discussed the algorithms of mining *FRACTURE* patterns from a sequence of historical tree structures. Thus, we discuss the Phases II and III in the following subsections.

5.4.1 Similarity Computation

As the output of Phase I, each Web user is represented as a set of *FRACTURES*, where each *FRACTURE* is a set of Web session subtrees and associated with two values: FoC and SoC . We need to compute the similarity between each pair of users in the second phase.

Given two Web users represented as two sets of *FRACTUREs*, we first compute an optimal assignment of their *FRACTUREs* so that the total weight between shared *FRACTUREs* is maximized (i.e., the total *Prefix Level* between matched subtrees of the two *FRACTUREs* is minimized). For example, suppose $u_1 = \{X_1^1\}$ where $X_1^1 = \{a/b, a/c\}$, and $u_2 = \{X_2^1, X_2^2\}$ where $X_2^1 = \{a/b, a/c/i\}$ and $X_2^2 = \{a/b/e, a/c/i\}$. Although X_1^1 is approximate with both X_2^1 and X_2^2 , we assign X_1^1 to X_2^1 so that the total weight between shared *FRACTUREs* of the two users is maximized. This problem is referred to as *Assignment Problem*, in the branch of optimization or operations research in mathematics [113], which can be defined as follows [51]:

Any problem involving minimizing the sum of $C(a, b)$ over a set P of pairs (a, b) where a is an element of some set A and b is an element of set B , and C is some function, under constraints such as “each element of A must appear exactly once in P ” or similarly for B , or both.

In our problem, A is the set of *FRACTUREs* of one user, B is the set of *FRACTUREs* of the other user, C is the total *Prefix Level* between matched subtrees from a pair of shared *FRACTUREs*⁵, and P is the assignment solution. Many algorithms have been developed to handle the assignment problem [113]. Here, we use the Hungarian algorithm [9], which solves assignment problems in $O(n^3)$ time⁶. An implemented version of Hungarian algorithm can be freely downloaded from [10].

After getting the optimal assignment, the *FS* of shared *FRACTUREs* and the *US* of the two users can be computed according to Definition 5.5 and Definition 5.6.

5.4.2 Cluster Generation

After Phase II, we get a similarity matrix of Web users. Then, many appropriate clustering methods can be used to generate the clusters. The suitability of different methods are in general application dependent. In our work, we compare the performance of 8 clustering methods provided by CLUTO (CLustering TOolkit), which is a freely distributed clustering package created by Karypis [61]. Particularly, we experimented with four partitional

⁵Note that, finding matched subtrees from a pair of *Approximate FRACTUREs* is an assignment problem also. For example, given a pair of *Approximate FRACTUREs*, $\{a/b, a/b/c\}$ and $\{a/b, a/b/c/d\}$, the minimum sum of *Prefix Level* should be 1. The sum of *Prefix Level* can be 3 if assigning subtrees differently.

⁶Although the complexity is polynomial, the number n (the number of *FRACTUREs* of a user or the number of subtrees of a *FRACTURE*) in our problem is usually not high.

methods using different clustering criterion functions, three agglomerative methods using different merging schemes and one graph-based clustering algorithm.

We briefly describe the employed clustering methods as follows. For more details, please refer to the document provided by CLUTO [61]. The partitional clustering methods we used generate the desired k -way clustering solutions by performing a sequence of $k - 1$ repeated bisections. During each step, the cluster is bisected so that the resulting 2-way clustering solution optimizes a particular clustering criterion function. The mathematical definitions of the four criterion functions used by us are shown in Eq. 5.5 through Eq. 5.8, where k is the number of clusters, n_i is the number of users in the i th cluster, u and v are two individual users, C_i is the i th cluster and C represents clusters other than C_i .

$$I_1 : \text{maximize} \sum_{i=1}^k \frac{1}{n_i} \left(\sum_{v, u \in C_i} \text{sim}(u, v) \right) \quad (\text{Eq. 5.5})$$

$$I_2 : \text{maximize} \sum_{i=1}^k \sqrt{\sum_{v, u \in C_i} \text{sim}(u, v)} \quad (\text{Eq. 5.6})$$

$$E_1 : \text{minimize} \sum_{i=1}^k n_i \frac{\sum_{v \in C_i, u \in C} \text{sim}(v, u)}{\sqrt{\sum_{v, u \in C_i} \text{sim}(u, v)}} \quad (\text{Eq. 5.7})$$

$$G_1 : \text{minimize} \sum_{i=1}^k \frac{\sum_{v \in C_i, u \in C} \text{sim}(v, u)}{\sum_{v, u \in C_i} \text{sim}(u, v)} \quad (\text{Eq. 5.8})$$

The agglomerative clustering methods we used generate the desired k -way clustering solution based on the agglomerative paradigm, which starts with points (users) as individual clusters and, at each step, merge the closest pair of clusters. Three different merging schemes are used: *single-link*, *complete-link* and *UPGMA* (Unweighted Pair Group Method with Arithmetic mean) [79]. The three merging schemes respectively define the cluster proximity based on the closest two points in different clusters, the farthest two points in different clusters, and the average pairwise proximities of all pairs of points from different clusters.

The graph-based clustering method computes the desired k -way clustering solution by first modelling the points (users) using a nearest-neighbor graph, and then splitting the graph into k -clusters using a min-cut graph partitioning algorithm.

Table 5.2: Parameters and Datasets.

U	Number of Web users	1000	Data	Feature Parameters
W	Average number of <i>FRACTURE</i> s per user	3	D_1	$R = 0 - 1$
G	Number of <i>FRACTURE</i> groups	6	D_2	$R = 0 - 5$
F	Average number of <i>FRACTURE</i> s of each group	6	D_3	$R = 0 - 10$
P	Number of <i>FRACTURE</i> s	36	D_4	$W = 5, F = 4$
S	Average number of subtrees of each <i>FRACTURE</i>	3	D_5	$W = 4, F = 5$
N	Number of nodes of Web session tree	1000	D_6	$W = 3, F = 6$
R	Range of overall prefix levels	0-5	D_7	$W = 2, F = 7$

(a) Parameter List

(b) Dataset List

5.5 Experimental Results

In this section, we evaluate the performance of *COWES* via experiments on both synthetic and real data sets. We point out here that the novelty of *COWES* is not the clustering method, but the extraction of appropriate information from historical Web sessions as a base for clustering and the similarity measures we defined to measure the proximity of Web users in terms of their characteristics in usage data evolution. The first two steps of *COWES* are implemented in the Java programming language. All experiments are carried out on a Pentium IV 2.8GHz PC with 512MB memory. The operating system is Windows 2000 professional.

5.5.1 Experiments on Synthetic Data

We conduct two experiments on the synthetic data. The first experiment is carried out to evaluate alternative similarity measures. The second experiment is used to study the performance of different clustering methods in our application.

5.5.1.1 Synthetic Data Generator

We implemented a synthetic data generator which generates *FRACTURE*s for users with the following steps. Parameters used by the generator are shown in Table 5.2 (a), where the third column shows the default values of the parameters.

- Firstly, we generate a general Web session tree with the given number of nodes N . At a given node, the number of children is sampled uniformly at random from the range 0 to 10. For each child, the process is performed recursively if the depth of the tree is less than or equal to R or the total number of nodes reaches N .

- We then select subtrees from the general Web session tree to compose P *FRACTURES*. The size of each *FRACTURE* is picked from a Poisson distribution with mean equal to S . The subtrees are sampled uniformly at random from the general Web session tree.
- Then, we organize *FRACTURES* into *FRACTURE* groups. We sample the size of a *FRACTURE* group from a uniform distribution in the range of 0 to F . The group to which a *FRACTURE* will be assigned is sampled uniformly at random from the range 0 to G .
- Finally, we generate *FRACTURES* for Web users. For each Web user, we decide the number of *FRACTURES* from a Poisson distribution with mean equal to W . Each *FRACTURE* group is associated with a weight, which corresponds to the probability that it will be selected for a Web user. The weight is picked from an exponential distribution with unit mean. Weights of each group is normalized so that the sum of weights is 1. The next group to be used for a user is chosen by tossing an G -sided weighted coin, where the weight of a side is the probability of picking the corresponding group. To model the phenomenon that all *FRACTURES* in a group do not always occur together, we associate each group a corruption level, which is obtained from a normal distribution with mean 0.5 and variance 0.1. Then, when adding *FRACTURES* for a user, we keep dropping a *FRACTURE* from the group as long as a uniformly distributed random number between 0 and 1 is less than the corruption level of the group. To model the phenomenon of *Approximate FRACTURE*, for each *FRACTURE*, we replace some of its subtrees with its ancestor/descendant subtrees. The number of replaced subtrees is decided randomly from a uniform distribution in the range of 0 and the size of the *FRACTURE*. The prefix level between the original subtree and the replacing subtree is also sampled randomly from a uniform distribution in the range of 0 and R . Finally, we assign *FoC* and *SoC* to each selected *FRACTURE* from a uniform distribution in the range of $[0, 1]$.

Totally, there are 7 datasets used in our experiments. As shown in Table 5.2 (b), the first three datasets are generated with the default parameter values except that the value of parameter R is varied. We use the three datasets to conduct the first experiment which evaluates the alternative similarity measures. Hence, datasets generated with different R

Figure 5.5: Overall value of criterion functions versus number of clusters.

values will affect the weight of shared *FRACTURES*, which in turn affect the performance of alternative similarity measures. The last four datasets are generated by varying the parameters W and F . The four datasets are used in the second experiment which studies the performance of different clustering methods. Datasets generated with different W and F values exhibit different degrees of data dense, which affects the performance of clustering methods.

5.5.1.2 Result Analysis

Before conducting any experiment to evaluate alternative similarity measures or clustering methods, we first carried out an initial experiment to see whether the natural number of clusters can be decided clearly. We then performed clustering on the dataset D_2 using the partitional methods with criterion functions I_1 , I_2 and E_1 . The plots of overall value of criterion functions versus the number of clusters are shown in Figure 5.5. Since there exists no distinct knees or peaks in the overall value of criterion function, according to [133], the number of clusters can not be decided clearly. Then, we simply generate 10 clusters in the remaining experiments.

As we mentioned in Section 5.3, different monotonic decreasing functions can be used in transforming a distance measure to a similarity measure. Hence, the similarity measure US can be defined alternatively. We now conduct experiments to study the performance of 6 alternative similarity measures as in Table 5.3. For example, for US_1 , we use the function e^{-d} for both FS and ω . While, for US_2 , we use the function e^{-d} for FS and $\frac{1}{1+d}$ for ω .

Table 5.3: Alternative similarity measures.

US	ω	FS
US_1	e^{-d}	e^{-d}
US_2	$\frac{1}{1+d}$	e^{-d}
US_3	$0/1$	e^{-d}
US_4	e^{-d}	$\frac{1}{1+d}$
US_5	$\frac{1}{1+d}$	$\frac{1}{1+d}$
US_6	$0/1$	$\frac{1}{1+d}$

In addition, we use similarity measure US_3 (and US_6) to consider *Identical FRACTURES* only. That is, ω is either 1 for *Identical FRACTURES*, or 0 for other situations.

We perform clustering with the six similarity measures on datasets D_1 , D_2 and D_3 . The results are shown in Figures 5.6 (a), (b) and (c) respectively. In Figure 5.6, we use the two measures IS and ES to compare the clustering qualities. The IS refers to the average internal similarity of clusters, which can computed as follows.

$$IS = \frac{1}{k} \sum_{i=1}^k \frac{1}{n_i} \left(\sum_{v,u \in C_i} sim(v,u) \right) \quad (\text{Eq. 5.9})$$

While, ES refers to the average external similarity of clusters, which is defined as Eq. 5.10.

$$ES = \frac{1}{k} \sum_{i=1}^k \frac{1}{n_i} \sum_{v \in C_i, u \in C} sim(v,u) \quad (\text{Eq. 5.10})$$

Basically, for a good clustering, the IS should be large while the ES should be small. Then, from the experimental results in Figure 5.6, we have the following observations:

- There is a tradeoff between IS and ES . That is, similarity measures which works well in IS do not perform well in ES , vice versa. For example, US_5 can achieve high IS value, while its ES is quite high as well. Similarly, although US_3 (or US_6) generates low ES , it sacrifices too much in IS . Generally, US_4 , which is the user similarity measure defined in Definition 5.6, generates well IS without sacrificing too much in ES .
- From each table, it can be discovered that the performance of a similarity measure is independent of the clustering methods. That is, no matter what kind of clustering

CHAPTER 5. COWES: FRACTURE-BASED CLUSTERING

		partitional				agglomerative			graph-based
		I1	I2	E1	G1	slink	clink	upgma	
US ₁	IS	0.487	0.494	0.490	0.486	0.063	0.518	0.428	0.201
	ES	0.131	0.130	0.131	0.131	0.159	0.123	0.123	0.005
US ₂	IS	0.516	0.506	0.511	0.502	0.067	0.535	0.445	0.205
	ES	0.139	0.139	0.139	0.139	0.169	0.131	0.133	0.005
US ₃	IS	0.454	0.463	0.414	0.441	0.053	0.394	0.374	0.183
	ES	0.104	0.104	0.116	0.107	0.120	0.125	0.101	0.005
US ₄	IS	0.514	0.514	0.517	0.514	0.068	0.534	0.452	0.209
	ES	0.142	0.142	0.123	0.141	0.172	0.133	0.134	0.006
US ₅	IS	0.537	0.538	0.533	0.536	0.182	0.553	0.529	0.215
	ES	0.152	0.152	0.153	0.151	0.153	0.141	0.153	0.006
US ₆	IS	0.456	0.481	0.445	0.459	0.056	0.434	0.396	0.187
	ES	0.112	0.110	0.117	0.114	0.132	0.127	0.107	0.005

(a) Performance of alternative similarity measures on D_1

		partitional				agglomerative			graph-based
		I1	I2	E1	G1	slink	clink	upgma	
US ₁	IS	0.319	0.319	0.301	0.327	0.067	0.275	0.301	0.150
	ES	0.090	0.090	0.091	0.079	0.075	0.079	0.060	0.004
US ₂	IS	0.373	0.369	0.358	0.368	0.015	0.352	0.353	0.149
	ES	0.108	0.111	0.112	0.114	0.102	0.096	0.077	0.004
US ₃	IS	0.277	0.261	0.267	0.268	0.049	0.262	0.255	0.115
	ES	0.064	0.062	0.075	0.062	0.047	0.073	0.040	0.005
US ₄	IS	0.323	0.322	0.319	0.332	0.067	0.276	0.266	0.133
	ES	0.092	0.090	0.103	0.081	0.076	0.085	0.064	0.004
US ₅	IS	0.381	0.381	0.371	0.381	0.381	0.376	0.376	0.152
	ES	0.116	0.116	0.118	0.119	0.119	0.119	0.077	0.004
US ₆	IS	0.266	0.263	0.268	0.270	0.049	0.265	0.262	0.116
	ES	0.065	0.062	0.075	0.062	0.047	0.073	0.038	0.005

(b) Performance of alternative similarity measures on D_2

		partitional				agglomerative			graph-based
		I1	I2	E1	G1	slink	clink	upgma	
US ₁	IS	0.286	0.286	0.284	0.282	0.186	0.226	0.311	0.109
	ES	0.085	0.085	0.085	0.082	0.057	0.091	0.078	0.004
US ₂	IS	0.345	0.286	0.344	0.341	0.114	0.326	0.300	0.127
	ES	0.112	0.085	0.112	0.108	0.090	0.110	0.105	0.004
US ₃	IS	0.225	0.217	0.215	0.220	0.209	0.256	0.213	0.092
	ES	0.051	0.051	0.051	0.051	0.040	0.062	0.047	0.004
US ₄	IS	0.290	0.290	0.287	0.286	0.186	0.229	0.286	0.108
	ES	0.087	0.087	0.086	0.084	0.059	0.089	0.073	0.004
US ₅	IS	0.356	0.356	0.355	0.352	0.114	0.342	0.277	0.129
	ES	0.117	0.117	0.117	0.113	0.095	0.108	0.107	0.004
US ₆	IS	0.226	0.218	0.213	0.221	0.209	0.312	0.220	0.092
	ES	0.052	0.051	0.051	0.052	0.037	0.062	0.044	0.004

(a) Performance of alternative similarity measures on D_3

Figure 5.6: Similarity matrix ordered by clustering results.

Figure 5.7: IS and ES generated by partitional I_1 on different datasets.

method is used, the performance of a similarity measure is consistent (note that, the agglomerative clustering method which uses the simple-link merging scheme is not a suitable method here because the generated IS is lower than ES in many cases).

- From the tables, it can be noticed that when the dataset changes so that the overall prefix levels between *Approximate FRACTURES* increases, the goodness of the similarity measures is consistent. For example, Figures 5.7 (a) and (b) respectively plot the IS and ES values generated by the partitional clustering method with the I_1 function and the six similarity measures on the three datasets. We learned that US_2 and US_5 always achieve the highest IS (ES) while US_3 and US_6 persistently have the lowest IS (ES).

We now describe the second experiment conducted to evaluate the performance of different clustering methods. Different methods have different performance with respect to characteristics of the datasets. In our experiment, we use the four synthetic datasets D_4 , D_5 , D_6 and D_7 , which are generated by using different values for parameters W and F . According to the way we generate synthetic datasets, the higher the W and the lower the F , the more overlapped the user groups. Consequently, the dataset D_4 is more overlapped than D_7 . Figure 5.8 shows the results of the experiments, where the defined similarity measure US_4 is used. It can be observed that the partitional clustering method using the criterion function I_1 usually achieves the highest IS value while the graph-based clustering method works best in ES . Figure 5.9 and Figure 5.10 show the gray scale images of the same similarity matrix ordered by the clusters generated by the eight algorithms. The shade of each

		partitional				agglomerative			graph-based
		I1	I2	E1	G1	slink	clink	upgma	
D ₄	IS	0.268	0.275	0.270	0.288	0.014	0.238	0.311	0.091
	ES	0.134	0.120	0.121	0.124	0.139	0.122	0.106	0.007
D ₅	IS	0.402	0.379	0.371	0.385	0.182	0.298	0.252	0.156
	ES	0.201	0.183	0.184	0.142	0.023	0.158	0.099	0.010
D ₆	IS	0.363	0.327	0.320	0.309	0.063	0.289	0.276	0.138
	ES	0.117	0.122	0.121	0.115	0.094	0.112	0.083	0.006
D ₇	IS	0.417	0.411	0.364	0.408	0.054	0.327	0.382	0.229
	ES	0.079	0.079	0.076	0.079	0.107	0.071	0.068	0.003

Figure 5.8: Performance of different clustering methods on different datasets.

point in the images represents the value of the corresponding entry in similarity matrix. In extreme cases, white and black correspond to the similarity values of 1 and 0 respectively. Hence, for a good clustering, the rectangles on the diagonal should be as white as possible as they represent the Web users in the same clusters, while the remaining areas should be as black as possible. We observed that generally, the partitional clustering methods perform well not only in achieving the better accuracy but also in controlling the balance of the cardinality of the clusters.

5.5.2 Experiments on Real Data

We also conducted three experiments on real-life data. In the first experiment, we compare the accuracy of *COWES* and an existing algorithm which clusters tree-structured data without considering its evolution. The second experiment is conducted to compare the effectiveness of our similarity measure and alternative similarity measures. The third experiment shows the performance variation with respect to the parameters of *COWES*.

5.5.2.1 DataSets

The real-life datasets are collected from Internet Traffic Archive (<http://ita.ee.lbl.gov>), sponsored by ACM SIGCOMM. We use the trace that contains a day’s worth of all HTTP requests to the EPA WWW server located at Research Triangle Park, NC. In considering the evolution of Web usage data, the requests of a host are grouped with a time interval of one hour. All the requests of all 2333 hosts in the trace form the Dataset I. In order to study the novelty of the knowledge that can be discovered by *COWES*, we collect the requests of 57 hosts that browse the subtree with two paths, “/docs/WhatsNew.html” and “/docs/WhatsHot.html” to form the Dataset II. Since hosts in the Dataset II are similar

CHAPTER 5. COWES: FRACTURE-BASED CLUSTERING

Figure 5.9: Similarity matrix ordered by clustering results I.

Figure 5.10: Similarity matrix ordered by clustering results II.

in their requests, without considering the evolutionary characteristics of the requests, they may not be distinguished by existing cluster algorithms. We study to see whether *COWES* can generate clusters of high quality based on evolutionary features of the requests.

5.5.2.2 Result Analysis

We first conduct experiments to evaluate the accuracy of *COWES* on real-life data. The defined similarity measure US_4 and the partitional clustering method with criterion function I_1 are used. Similarly, we use IS and ES to evaluate the quality of the clustering results. We compare the results of *COWES* with those of an existing clustering method, referred to as *STRUCTURE* in Figure 5.11, which clusters hosts based on their a day's worth all HTTP requests without considering the evolution of the requests. The details of *STRUCTURE* can be found in [92]. The reason we compare *COWES* against *STRUCTURE* instead of other classical Web user clustering methods [152, 146] is as follows. Classical Web user clustering methods usually represent each Web session as a bag pages. However, *STRUCTURE* considers Web sessions as tree structures, which capture more semantic similarity between pages. For example, given two web sessions, $\{a/b, a/c\}$ and $\{a/b/d, a/c\}$, *STRUCTURE* defines similarity based on shared edges. Then, the Jaccard similarity between the two Web sessions is $2/3$. However, if we consider the two Web sessions as bags of pages, the Jaccard similarity between the two sessions is $1/3$. Since *COWES* represents Web sessions as trees also, it is fair to compare *COWES* with *STRUCTURE* instead of other classical Web user clustering algorithms.

CHAPTER 5. COWES: FRACTURE-BASED CLUSTERING

Num of Clusters	Dataset I				Num of Clusters	Dataset II			
	COWES		STRUCTURE			COWES		STRUCTURE	
	IS	ES	IS	ES		IS	ES	IS	ES
5	0.36	0.013	0.09	0.007	3	0.67	0.24	0.35	0.24
6	0.36	0.014	0.08	0.006	4	0.72	0.39	0.37	0.24
7	0.38	0.017	0.21	0.006	5	0.73	0.34	0.38	0.23
8	0.39	0.019	0.18	0.008	6	0.72	0.32	0.40	0.22

Figure 5.11: Comparison of clustering algorithms.

Num of Clusters	Dataset I				Num of Clusters	Dataset II			
	US_4		US_6			US_4		US_6	
	IS	ES	IS	ES		IS	ES	IS	ES
5	0.36	0.013	0.21	0.015	3	0.67	0.24	0.59	0.21
6	0.36	0.014	0.22	0.015	4	0.72	0.39	0.67	0.34
7	0.38	0.017	0.38	0.019	5	0.73	0.34	0.65	0.31
8	0.39	0.019	0.30	0.024	6	0.72	0.32	0.65	0.29

Figure 5.12: Comparison of similarity measures.

The experimental results are shown in Figure 5.11. The *DoC*, *FoC* and *SoC* thresholds for *FRACTURE* mining are fixed at 0.5, 0.2 and 0.6 respectively. On Dataset I, the number of generated clusters ranges from 5 to 8. Since there are fewer hosts in Dataset II, we generate 3 through 6 clusters. We observed that for Dataset I, the accuracy achieved by *COWES* is compatible with respect to that of *STRUCTURE*. However, for Dataset II, *COWES* achieves much better *IS* and competitive *ES*. The reason that *STRUCTURE* cannot perform as well as *COWES* on Dataset II is as follows. *STRUCTURE* clusters hosts based on all pages accessed by each host in one day. Hence, it cannot distinguish hosts which accessed similar pages (e.g., “/docs/WhatsNew.html” and “/docs/WhatsHot.html”). Nevertheless, *COWES* clusters hosts based on the evolutionary features of Web sessions. Although hosts accessed similar pages, *COWES* can distinguish them if they share similar *FRACTUREs*. For example, we observed that in all the clustering results on Dataset II when the cluster number is varied from 3 to 6, the two hosts, “e659229.boeing.com” and “key-hole.es.dupont.com”, are consistently clustered together. After analyzing the historical Web sessions of the two hosts, we observed that although they share similar pages with other hosts in Dataset II, they share a particular pair of *FRACTUREs* $\{EPA-AIR/1995, docs/EPA-AIR/1995\}$ ⁷, whose sequential *DoC* values are shown in Figures 5.13 (a) and (b) respec-

⁷The two hosts share some other *FRACTUREs* which are shared by other hosts in the cluster also. For clarity, we do not show them all.

Figure 5.13: Shared *FRACTUREs* of clustered hosts.

tively. For “e659229.boeing.com”, *FoC* of the *FRACTURE* is 0.22 and *SoC* of the *FRACTURE* is 1.0. For “keyhole.es.dupont.com”, the *FoC* and *SoC* values of the *FRACTURE* are 0.26 and 1.0. *COWES* successfully distinguished them from other hosts in Dataset II because they frequently vary their information needs under the two Web session subtrees together.

We conduct another experiment to evaluate the effectiveness of our similarity measure in the real-life data. Particularly, we compare the two similarity measures US_4 and US_6 described in the previous subsection. The former is the defined similarity measure while the latter considers *Identical FRACTUREs* only. The experimental results are shown in Figure 5.12. We noticed that although both similarity measures have similar performance in *ES*, the defined similarity measure works much better in *IS*.

We then examine how the performance of *COWES* varies with respect to the variations of *DoC*, *FoC* and *SoC* thresholds. The experiments are conducted on the Dataset I. The cluster number is fixed at 5 (Note that, our objective in this experiment is to study the variation trend of the performance. Hence, the number of clusters will not affect the trend.). When varying one of the thresholds, the other two are set as default values. The default values of the three thresholds are 0.4, 0.6, and 0.7 respectively. The normalized *IS* and *ES* values with respect to each threshold are shown in Figures 5.14 (a), (b) and (c) respectively. We observed that when the threshold values are increased, both the *IS* and *ES* values decreased. The reason is that when the threshold values are low, more *FRACTURE* patterns can be discovered for each Web user. Hence, users probably share more *FRACTUREs* and

Figure 5.14: Performance variation with respect to thresholds.

both the average internal similarity and average external similarity increase. On the contrary, when the thresholds are high, fewer *FRACTURES* can be discovered and Web users probably share fewer *FRACTURES*. Consequently, both *IS* and *ES* values decrease. We also observed that that our approach on this dataset is more sensitive to the thresholds of *DoC* and *SoC* than the threshold of *FoC*. The reason is that even if the threshold of *FoC* is lowered, it is not necessary more *FRACTURES* can be found for each user, because the threshold of *SoC* is fixed. Generally, the performance of *COWES* depends on the thresholds. Users of *COWES* need to adjust the thresholds until acceptable *IS* and *ES* values are reached.

5.6 Related Work

Clustering of web users is an important task of web usage mining. Existing works on web user clustering usually extract access patterns of users from web server log files and organize them into web sessions. J. Xiao et al. [152] clustered web user sessions based on various similarity measures, such as the number of shared web pages, the frequency of accessing the shared web pages, the visiting time spent on shared pages and the order of visiting shared paged. Rather than clustering the web users based on web sessions directly, Y. Fu et al. [63] first generalized the sessions so that pages representing the similar semantics are collapsed. By this manner, the dimension of clustering feature can be reduced significantly. W. Wang and O. R. Zaiane [146] also cluster web users based on snapshots of web sessions. They represented web sessions as vectors of encoded page IDs and then a clustering algorithm handling categorical data was employed. The critical difference between existing works on clustering web users and our effort is that we address the dynamic nature of web usage data.

Existing works cluster web users by the snapshots of web sessions, whereas we cluster web users by their historical web sessions. Furthermore, our similarity measure is also different from existing ones. We measure the proximity of web users based on the characteristics of their usage data evolution. Existing works measure the likeness between web users based on the information in snapshot web sessions. Consequently, the clusters generated by our algorithm indicate different knowledge and thus have different applications.

Recently, the evolutionary feature of data was addressed by some data mining works which aim to maintain consistent mining results over time. For example, D. Chakrabarti et al. [27] proposed to cluster timestamped data such that a clustering at each timestep is similar to the clustering at the previous timestep. Furthermore, each clustering should accurately reflect the data arrived during that timestep. The dynamic nature of Web access patterns were also observed by O. Nasraoui et al. [103] [102]. As an alternative to locking the state of the web access patterns in a frozen state depending on when the web log data was collected and preprocessed, they proposed to consider the web usage data as a reflection of a dynamic environment. However, our works are different in the way that they cope with the dynamic nature of web usage data by designing evolutionary approaches to update knowledge, whereas we mine the historical web usage data to obtain novel knowledge.

5.7 Summary

In this chapter, we have presented an application of *FRACTUREs*, *COWES*. To the best of our knowledge, this is the first work which clusters Web users based on the evolution of Web usage data. In order to capture the evolutionary characteristics of Web usage data, we mine *FRACTUREs* from users' historical Web sessions and use discovered *FRACTUREs* as the clustering features.

We identified two types of *FRACTUREs* that can be shared by users. Since the two types of *FRACTUREs* contribute different importance to the proximity of users, we assign different weight to them. The proximity of a pair of shared *FRACTUREs* is measured based on their *strength* (e.g., *FoC* and *SoC* values). Finally, the similarity between users is defined by considering the *number* of shared *FRACTUREs* as well as the *similarity* of shared *FRACTUREs*.

Various clustering methods are employed to generate clusters of Web users. The experimental results show that (i) partitional clustering methods are preferable to agglomerative

CHAPTER 5. COWES: FRACTURE-BASED CLUSTERING

clustering methods and graph-based methods in our application; (ii) the defined similarity measure works well in achieving high average internal similarity of clusters, without sacrificing too much in average external similarity of clusters; (iii) *COWES* is effective in discovering high quality clusters when users accessing similar pages exhibit different characteristics in usage evolution.

Chapter 6

Negative FRACTURE Mining and Application

Given a sequence of historical versions of a tree structure, a *FRACTURE* pattern is a set of subtrees that frequently change together. If we refer to this type of relation between subtrees as *positive evolution association*, then there exists another type of relation between subtrees, which can be referred as *negative evolution association*. Contrary to the positive evolution association, the *negative evolution association* between subtrees indicates that subtrees are negatively correlated in undergoing structural changes. For example, when a set of subtrees changes (significantly), another set of subtrees rarely changes (significantly). In this chapter, we extend the model of *FRACTURE* pattern to discover negative evolution association between subtrees. We refer to this type of patterns as *negative FRACTURES*. *Negative FRACTURES* are useful in applications such as maintaining mirrors of Web site, maintaining XML path selectivity estimation [144] etc.

The rest of this chapter is organized as follows. We begin with some preliminary definitions. In Section 6.2, we describe the data mining algorithm we developed for *negative FRACTURE* mining. In Section 6.3, we first present some applications of *negative FRACTURES* and then applied *negative FRACTURES* in autonomous Web mirror site maintenance. We evaluate the performance of the developed *negative FRACTURE* mining algorithm and the effectiveness of *negative FRACTURE*-based mirror maintenance strategies in Section 6.4. Finally, we conclude this chapter with a summary.

Figure 6.1: Sequence of historical tree versions.

6.1 Preliminaries

In this section, we first define some related evolution metrics. Then, the notion of *negative FRACTURE* is defined formally based on the metrics. Finally, we formalize the problem of *negative FRACTURE* mining.

6.1.1 Evolution Metrics

Similar to *FRACTURES*, *negative FRACTURES* belong to the type of *intra-tree associative evolutionary structures* which are mined from a sequence of historical versions of a tree structure. Hence, in the subsequent discussion, we use the following notations as described in Chapter 4. Let $\mathcal{T} = \langle t_1, t_2, \dots, t_n \rangle$ be a sequence of time points with a particular time granularity, $\Sigma = \langle S^{t_1}, S^{t_2}, \dots, S^{t_n} \rangle$ be a sequence of versions of tree S on $\mathcal{T}$, $\langle \Delta(t_1, t_2), \dots, \Delta(t_{n-1}, t_n) \rangle$ be the sequence of corresponding structural deltas, where each structural delta consists of the set of changed subtrees in two successive tree versions, and $\Omega = \Delta(t_1, t_2) \cup \dots \cup \Delta(t_{n-1}, t_n)$ be the set of all changed subtrees.

As defined in Chapter 4, the *DoC* (*Degree of Change*) of a subtree¹ measures how significantly a subtree changed in two successive versions. For example, consider the sequence of four tree versions in Figure 6.1, $\langle S^{t_1}, S^{t_2}, S^{t_3}, S^{t_4} \rangle$. The *DoC* value of the subtree a/b in the first two versions can be computed as $DoC(a/b, t_1, t_2) = 4/12 \approx 0.33$. The *DoC* value of a subtree is zero if it did not undergo any structural change in two versions.

In order to justify whether a set of subtrees are correlated in undergoing significant changes, we need a metric to measure how frequently the set of subtrees undergoes significant

¹Similar to *FRACTURE* mining, we consider bottom-up subtrees as well.

CHAPTER 6. NEGATIVE FRACTURE MINING AND APPLICATION

changes together. Hence, we introduce the metric *Frequency of Significant Change* ($FoSC$) for a set of subtrees. Note that, the metric $FoSC$ is defined with respect to a given threshold of DoC because we say a subtree undergoes significant changes only if its DoC value is no less than the DoC threshold.

Definition 6.1 (FoSC) Let $X = \{s_1, s_2, \dots, s_m\}$ be a set of changed subtrees, $X \subseteq \Omega$, and the threshold of DoC be α . The Frequency of Significant Change for the set X , with respect to α , denoted as $FoSC_\alpha(X)$, is:

$$FoSC_\alpha(X) = \frac{\sum_{j=1}^{n-1} D_j}{(n-1)}$$

$$\text{where } D_j = \prod_{i=1}^m D_{j_i} \text{ and } D_{j_i} = \begin{cases} 1, & \text{if } DoC(s_i, t_j, t_{j+1}) \geq \alpha \\ 0, & \text{otherwise} \end{cases} \quad 1 \leq i \leq m \quad \square$$

That is, $FoSC_\alpha$ of a set of subtrees is the fraction of subsequent versions (after the first version) where the set of subtrees undergoes significant changes together. The value of $FoSC_\alpha$ ranges in $[0, 1]$. If all subtrees in the set undergo significant changes together in each subsequent version, then the value of $FoSC_\alpha$ equals to one. If subtrees in the set never undergo significant changes together in subsequent versions, then the value of $FoSC_\alpha$ is zero. Obviously, the $FoSC$ of a set equals to the product of its FoC and SoC_α , defined in Chapter 4. That is, given a subtree set X , $FoSC_\alpha(X) = FoC(X) \times SoC_\alpha(X)$.

Example 6.1 Consider the sequence of historical tree versions in Figure 6.1. Let $X = \{a/b, a/b/e\}$ and the threshold of DoC α be $1/4$. Then, $FoSC_{1/4}(X) = 2/3$ as both subtrees undergo significant changes in two subsequent versions S^{t_2} and S^{t_4} . Let $Y = \{a/b/e, a/b/e/i\}$. Then $FoSC_{1/4}(Y) = 1/3$ as the two subtrees undergo significant changes together only in the version S^{t_3} . ■

We now define the metric to measure the correlation between subtrees in undergoing significant changes. Given a DoC threshold α , in each transition between two successive versions, a set of subtrees either undergoes significant changes together or not. This could be considered as a binary variable. Furthermore, it is a symmetric binary variable because we are interested in both the occurrences and the nonoccurrences of significant changes to subtrees. Hence, correlation measures which are suitable for analyzing symmetric binary variables can be used, such as ϕ -coefficient, odds ratio, and the Kappa statistic etc. [133]. In our analysis, we use the ϕ -coefficient. Given the contingency table in Table 6.1, where

Table 6.1: 2×2 contingency table.

	Y	$\neg Y$	Σ_{row}
X	f_{11}	f_{10}	f_{1+}
$\neg X$	f_{01}	f_{00}	f_{0+}
Σ_{col}	f_{+1}	f_{+0}	M

X (Y) represents that a set of subtrees X (Y) undergoes significant changes together and $\neg X$ ($\neg Y$) represents subtrees in X (Y) do not undergo significant changes together, the ϕ -coefficient between variables X and Y can be computed by the following equation.

$$\phi(X, Y) = \frac{M f_{11} - f_{1+} f_{+1}}{\sqrt{f_{1+}(M - f_{1+}) f_{+1}(M - f_{+1})}} \quad (\text{Eq. 6.1})$$

Particularly, in our mining context, the value of M in Table 6.1 equals to $n - 1$, which is the total number of transitions between successive historical versions. Furthermore, f_{11} refers to the number of versions where all subtrees in X and Y undergo significant changes together. Hence, f_{11} equals to $FoSC_\alpha(X \cup Y) \times (n - 1)$. Similarly, f_{1+} equals to $FoSC_\alpha(X) \times (n - 1)$ and f_{+1} equals to $FoSC_\alpha(Y) \times (n - 1)$. Thus, the equation Eq. 6.1 can be transformed as the following one, which is formally defined as *Correlation of Change (CoC)*.

Definition 6.2 (CoC) Let X and Y be two sets of subtrees, s.t. $X \subseteq \Omega$, $Y \subseteq \Omega$, and $X \cap Y = \emptyset$. Given a DoC threshold α , the Correlation of Change of X and Y , with respect to α , denoted as $CoC_\alpha(X, Y)$, is:

$$CoC_\alpha(X, Y) = \frac{FoSC_\alpha(X \cup Y) - FoSC_\alpha(X) * FoSC_\alpha(Y)}{\sqrt{FoSC_\alpha(X)(1 - FoSC_\alpha(X))FoSC_\alpha(Y)(1 - FoSC_\alpha(Y))}}$$

□

According to the definition of ϕ -coefficient, if $CoC_\alpha(X, Y)$ is greater than zero, two sets of subtrees X and Y are positively correlated in undergoing significant changes. Otherwise, they are negatively correlated in undergoing significant changes. In the following discussion, the subscript α in both $FoSC_\alpha$ and CoC_α is omitted if α is understood in the context. **Example 6.2** Consider the sequence of historical versions of tree S in Figure 6.1 again. Let the threshold of DoC α be $1/3$. Let $X = \{a/b, a/b/e\}$ and $Y = \{a/b/e/i\}$. $CoC_{1/3}(X, Y) = \frac{3 \times 0 - 2 \times 1}{\sqrt{2 \times (3 - 2) \times 1 \times (3 - 1)}} = -1$. Hence, $\{a/b, a/b/e\}$ and $\{a/b/e/i\}$ are negatively correlated in undergoing significant changes. ■

6.1.2 Pattern Definition

Based on the evolution metrics introduced above, various negatively correlated subtree patterns can be defined. For example, we can define the pattern as two sets of subtrees with a negative *CoC* value. The subtrees may or may not be structurally independent. We discuss applications of this kind of pattern in Section 6.3. Particularly, in our research, we define *negative FRACTURE* on subtrees with ancestor-descendant relationships. The usefulness of *negative FRACTURE* will be presented in Section 6.3 also.

Let s_i be an ancestor subtree of s_{i+1} (conversely, s_{i+1} be a descendant subtree of s_i), denoted as $s_i \succ s_{i+1}$, if there is a path from the root of s_i to the root of s_{i+1} . Then, *negative FRACTURES* can be defined as follows.

Definition 6.3 (negative FRACTURE) Given the *DoC* threshold α , the *FoSC* threshold η , and the *CoC* threshold ζ ($0 \leq \alpha, \eta \leq 1, \zeta \geq 0$), $P = \langle X, Y \rangle$, where $X = \langle s_1, s_2, \dots, s_{m-1} \rangle$, $Y = \langle s_m \rangle$, and $s_i \succ s_{i+1}$ ($1 \leq i < m$), is a *negative FRACTURE* if it satisfies the following conditions: (i) $FoSC_\alpha(X) \geq \eta$; (ii) $FoSC_\alpha(P) < \eta$; (iii) $CoC_\alpha(X, Y) \leq -\zeta$. $\square$

Example 6.3 Consider the sequence of historical versions of tree S in Figure 6.1 again. Suppose the threshold α is $1/3$, the threshold η is $2/3$ and the threshold ζ is $1/2$. $\langle \langle a/b, a/b/e \rangle, \langle a/b/e/i \rangle \rangle$ is a *negative FRACTURE* because $FoSC(\{a/b, a/b/e\}) = 2/3 \geq \eta$, while, $FoSC(\{a/b, a/b/e, a/b/e/i\}) = 0 < \eta$ and $CoC(\{a/b, a/b/e\}, \{a/b/e/i\}) = -1 < -\zeta$. $\blacksquare$

A *negative FRACTURE* $\langle \langle s_1, s_2, \dots, s_{m-1} \rangle, \langle s_m \rangle \rangle$ indicates that the sequence of subtrees $\langle s_1, s_2, \dots, s_{m-1} \rangle$ frequently undergo significant change together. Whereas, they rarely undergo significant changes together with a descendant subtree s_m . Based on the inferred knowledge, we identified two types of subsumption relationships between *negative FRACTURES* as follows.

Definition 6.4 (Tail Subsumption) Given two *negative FRACTURES* $P_1 = \langle X, Y_1 \rangle$ and $P_2 = \langle X, Y_2 \rangle$, where $Y_1 = \langle s_x \rangle$ and $Y_2 = \langle s_y \rangle$, we say P_1 *tail-subsumes* P_2 (or P_2 is *tail-subsumed* by P_1), denoted as $P_1 \sqsupset_t P_2$, if $s_x \succ s_y$. $\square$

Definition 6.5 (Head Subsumption) Given a *negative FRACTURE* $P_1 = \langle X_1, Y_1 \rangle$ and a subtree sequence X_2 s.t. $FoSC(X_2) \geq \eta$, where $X_1 = \langle s_1 s_2 \dots s_m \rangle$ and $X_2 = \langle v_1 v_2 \dots v_n \rangle$, we say P_1 is *head-subsumed* by X_2 , denoted as $P_1 \sqsubset_h X_2$, if $\exists i (1 < i \leq m \leq n)$ such that $s_1 = v_1, \dots, s_{i-1} = v_{i-1}$, and $s_i \prec v_i$. $\square$

Example 6.4 Consider example in Figure 6.1 again. If both $\langle\langle a/b \rangle, \langle a/b/e \rangle\rangle$ and $\langle\langle a/b \rangle, \langle a/b/e/i \rangle\rangle$ are *negative FRACTURES*, then the former tail-subsumes the latter. If the subtree sequence $S = \langle a/b, a/b/e \rangle$ satisfies the *FoSC* threshold, then all *negative FRACTURES* $\langle\langle a/b, a/b/e/i \rangle, Y \rangle$ are head-subsumed by the sequence S , where Y can be any descendant subtree of $a/b/e/i$. ■

We will show in subsection 6.3.2 that subsumed *negative FRACTURES* add no further value to applications of *negative FRACTURES*. Hence, we exclude the subsumed *negative FRACTURES* from the task of *negative FRACTURE mining*. Thus, the problem of *negative FRACTURE mining* can be formally defined as follows.

Definition 6.6 (Negative FRACTURE Mining) *Given a sequence of historical versions of a tree structure, the DoC threshold α , the FoSC threshold η and the CoC threshold ζ , the **problem of negative FRACTURE pattern mining** is to find the complete set of negative FRACTURES where each satisfies the specified thresholds and is neither tail-subsumed or head-subsumed.*

6.2 Negative FRACTURE Mining

Negative FRACTURES are different from *FRACTURES* in at least the following two aspects. First, since the former focus on the negative evolution associations while the latter capture the positive evolution associations, different evolution metrics are used. Second, the former are defined on a sequence of subtrees with ancestor-descendant relationships, while the latter are defined on subtrees which may or may not be structurally related. Hence, novel data mining algorithm should be developed for *negative FRACTURE mining*. In this section, we first present basic phases of the mining procedure. Then, we discuss the details of the data mining algorithm.

6.2.1 Procedure Overview

Given a sequence of historical versions of a tree structure and thresholds of *DoC* and *FoSC*, the general algorithm *negative_FRACTURE_Miner* is shown in Figure 6.3 (a). Note that, the threshold of *CoC* does not need to be given manually. Cohen [45] discussed the strength of the ϕ -coefficient value. It is stated that the correlation is strong if $|\phi|$ is above 0.5, moderate if $|\phi|$ is around 0.3, and weak if $|\phi|$ is around 0.1. Based on this argument,

the *CoC* threshold can be set automatically and progressively. In our algorithm, we set the initial threshold of *CoC* ζ as 0.5. If no *negative FRACTURE* is discovered with respect to this value and ζ is greater than 0.3, ζ is decreased successively. With a given ζ , basically, there are two phases involved in the mining procedure.

- **Phase I: *GDT Constructor*.** Given the input as a sequence of historical versions of a tree structure, the first phase has two subtasks. Firstly, structural deltas, which contain changed subtrees in each two successive versions, should be detected. Secondly, structural deltas should be organized into an appropriate data structure. Particularly, we propose a data structure, called *General Delta Tree (GDT)*, which not only records the changed subtrees but also preserves the ancestor-descendant relationships between subtrees. We discuss the details of this phase in the next subsection.
- **Phase II: *Pattern Discovery*.** The input of this phase is the *GDT* constructed in the previous phase. We mine *negative FRACTURES* with respect to a set of thresholds of *DoC*, *FoSC* and *CoC*. *Negative FRACTURES* are discovered with the *divide-and-conquer* strategy. The details of this phase are presented in the subsection 6.2.3.

6.2.2 *GDT Construction*

As mentioned above, given a sequence of historical versions of a tree structure as the input, the objective of the first phase is to detect structural deltas and organize structural deltas into a data structure which is appropriate for pattern mining. Recall that in *FRACTURE* mining, we organize changed subtrees into a table called *structural delta database*. However, such a data structure is insufficient here because it cannot preserve the ancestor-descendant relationships between subtrees. Hence, we propose another data structure, called *General Delta Tree (GDT)*. A *GDT* not only registers change information of subtrees, such as their *DoC* values, but also preserves the structural information of subtrees, such as the ancestor-descendant relationships.

Definition 6.7 (General Delta Tree) Given a sequence of historical tree versions Σ , a sequence of consolidate trees² of each two successive versions can be obtained, $\langle S'_1 = \langle N_1, E_1 \rangle, S'_2 = \langle N_2, E_2 \rangle, \dots, S'_{n-1} = \langle N_{n-1}, E_{n-1} \rangle \rangle$, where $S'_i = S^{t_i} \uplus S^{t_{i+1}}$. A *General*

²Consolidate tree is defined in Definition 4.2 of Chapter 4.

Figure 6.2: General delta tree and delta tree.

Delta Tree $GDT = \langle N, E \rangle$, where $N = N_1 \cup N_2 \cup \dots \cup N_{n-1}$ and $e = (x, y) \in E$ only if x is a parent of y in any $E_i (1 \leq i \leq n-1)$. Each node x is associated with a vector where the i th element corresponds to $DoC(s_x^{t_i}, s_x^{t_{i+1}})$. $\square$

Example 6.5 Consider the sequence of historical tree versions in Figure 6.1. The constructed general delta tree is shown in Figure 6.2 (a) (Note that, for clarity, we show the vector of DoC values for some of the nodes only). The node b is associated with a sequence $\langle 1/3, 2/11, 5/11 \rangle$, which indicates that the DoC value of the subtree a/b in the first two versions is $1/3$, $DoC(a/b, t_2, t_3) = 2/11$, and $DoC(a/b, t_3, t_4) = 5/11$. ■

The algorithm of this phase is shown in Figure 6.3 (b). At first, we initialize the root node of GDT . Then, for each two successive tree versions, we employ the algorithm X-Diff [147] to detect changes to tree structures. X-Diff is an algorithm for detecting changes to XML documents. It parses XML document into unordered trees before detecting changes. We revise the algorithm slightly so that it directly takes tree structures as input. Given two tree versions, X-Diff generates a *result tree* which combines the nodes in both versions. For example, Figure 6.2 (b) shows the example *result tree* generated by X-Diff after detecting changes between the first two tree versions in Figure 6.1. Here, we use gray nodes to represent the deleted nodes and black nodes to represent the inserted nodes.

For each *result tree* generated by X-Diff, we employ the algorithm *Depth.Traverse* to generate a *delta tree* which contains change information of subtrees. The *Depth.Traverse* algorithm is shown in Figure 6.3 (e). Basically, *Depth.Traverse* traverses the *result tree* in the depth-first order. While reaching a node x in the *result tree*, we create a corresponding node in the *delta tree* and maintain two counters for the node: *descendants* and

changed_descendants. The former records the number of descendants of x while the latter records the number of inserted/deleted descendants of x . Then, the *DoC* value of the subtree rooted at x can be computed by dividing *changed_descendants* by *descendants*. If the *DoC* value equals to zero, we remove the node from the *delta tree* because the subtree rooted at x is not a changed subtree. Otherwise, we associate the *DoC* value with the node. For example, after performing *Depth_Traverse* on the *result tree* in Figure 6.2 (b), the generated *delta tree* is shown in Figure 6.2 (c).

Then, for each generated *delta tree*, we merge it into the *GDT*. The algorithm is shown in Figure 6.3 (d). When merging the i th *delta tree* (subtree) rooted at node x' with the *GDT* (subtree) rooted at node x , we only need to set the i th element of the *DoC* sequence of x with the *DoC* value of $s_{x'}$. Then, the algorithm iteratively merges their children with the same label. For a child y' of x' , if x does not have any matching child, we create a child node of x labeled as y . We associate a *DoC* sequence with the new child node such that the i th element of the sequence is set as the *DoC* value of y' .

Let $|S|$ be the maximum size of the historical tree versions and $|\Delta S|$ be the maximum size of the *result trees*. Then, the complexity of the first phase can be computed as follows. The initialization of *GDT* is $O(|\Delta S|)$. According to X-Diff, the complexity of generating a *delta tree* is $O(|S|^2 \times \deg(S) \times \log_2(\deg(S)))$, where $\deg(S)$ is the average out-degree of nodes in tree versions. For each *result tree*, we traverse it to calculate the *DoC* for each node and then merge it to the *GDT* if its *DoC* is greater than zero. Thus, the complexity of this operation is $O(2|\Delta S|)$. Hence, the dominant cost is the change detection step, which is iterated $n - 1$ times. Hence, the complexity of phase I is $O((n - 1) \times (|S|^2 \times \deg(S) \times \log_2(\deg(S))))$.

6.2.3 Pattern Discovery

We now discuss the second phase of *negative FRACTURE* mining. The input of this phase is the *GDT* and the thresholds α , η and ζ of *DoC*, *FoSC* and *CoC*. We aim to discover all non-subsumed *negative FRACTURES* satisfying the thresholds from the *GDT*.

The complete set of *negative FRACTURES* $\mathcal{P}$ can be divided into disjoint partitions, $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2 \cup \dots \cup \mathcal{P}_m$, such that each partition contains *negative FRACTURES* starting from the same subtree. That is, $\forall P_x = \langle \langle s_x s_{x+1} \dots s_{x+m} \rangle, \langle s_{x+m+1} \rangle \rangle \in \mathcal{P}, P_y = \langle \langle s_y s_{y+1} \dots s_{y+n} \rangle, \langle s_{y+n+1} \rangle \rangle \in \mathcal{P}$, if $s_x = s_y$ then $P_x, P_y \in \mathcal{P}_i$; otherwise, $P_x \in \mathcal{P}_i, P_y \in \mathcal{P}_j$ ($i \neq j, 1 \leq i, j \leq m$).

CHAPTER 6. NEGATIVE FRACTURE MINING AND APPLICATION

Figure 6.3: *Negative FRACTURE* mining algorithm.

Therefore, the discovery of the complete set of *negative FRACTURES* can be divided into subproblems of mining patterns for each partition, where each pattern begins with the same subtree. The algorithm of the second phase is shown in Figure 6.3 (c). We mine patterns by traversing the *GDT* with the depth-first order. For each node x , which represents the subtree s_x , if the value of $FoSC(s_x)$ is no less than the threshold η , we call the function *PartPatternMiner* to discover *negative FRACTURES* starting from s_x . Note that, the $FoSC(s_x)$ can be computed with the vector of *DoC* values associated with node x . For example, consider the *GDT* in Figure 6.2 (a). Let $\alpha = 0.3$ and $\eta = 0.3$. The vector of *DoC* values of node b is $\langle 2/5, 2/11, 5/11 \rangle \approx \langle 0.40, 0.18, 0.45 \rangle$. Then, $FoSC(a/b) = 2/3 \approx 0.67 > \eta$. Thus, we need to find *negative FRACTURES* starting from subtree a/b .

Before mining *negative FRACTURES* starting from some subtree s_x , we initialize a pattern $P = \langle s_x \rangle$ and associate a bitmap with the pattern. In a bitmap, there is one bit for each element of the *DoC* sequence of node x . If the i th element is greater than α , then bit i is set to one. Otherwise, bit i is set to zero. For example, before mining *negative FRACTURES* starting from a/b in the *GDT* in Figure 6.2 (a), the bitmap associated with $P = \langle a/b \rangle$ is $\langle 101 \rangle$ if α is set as 0.3.

The algorithm *Part_Pattern_Miner* is shown in Figure 6.3 (f). For a particular node x in *GDT*, it generates all non-subsumed *negative FRACTURES* starting from s_x . Note that, we directly mine non-subsumed *negative FRACTURES* instead of performing filtering process afterwards. For this purpose, we employed the following three strategies in *Part_Pattern_Miner*. *Firstly, when mining negative FRACTURES starting from a subtree s_x , candidate patterns are generated and examined by traversing s_x with the depth-first manner.* As shown in Figure 6.3 (f), *Part_Pattern_Miner* performs a depth-first traversal on the subtree rooted at node x . This strategy facilitates the performing of the following two strategies. *Secondly, once a sequence of subtrees satisfying the $FoSC$ threshold is discovered, candidate patterns must be extended from this sequence.* For example, as shown in the Lines 4 to 6 of *Part_Pattern_Miner*, given the subtree rooted at node x and current pattern $\langle s_x \rangle$, for each child y of x , if $FoSC(\{s_x, s_y\})$ satisfies the threshold η , the pattern is updated as $P = \langle s_x, s_y \rangle$. Simultaneously, the bitmap of P is also updated with the *DoC* vector of node y . Then, *Part_Pattern_Miner* is recursively called for the updated pattern. This strategy prevents generating *negative FRACTURES* which are head-subsumed. *Thirdly, once a negative FRACTURE is discovered, we stop traversing nodes deeper than the current one.*

CHAPTER 6. NEGATIVE FRACTURE MINING AND APPLICATION

For example, as shown in Lines 8 and 9 of *Part_Pattern_Miner*, if $\text{CoC}(\{s_x\}, \{s_y\}) < -\zeta$, a *negative FRACTURE* $\langle \langle s_x \rangle, \langle s_y \rangle \rangle$ is discovered. *Part_Pattern_Miner* is not recursively called any more. This strategy keeps from generating *negative FRACTURES* that are tail-subsumed.

For example, consider the *GDT* in Figure 6.2 (a) again. Let the thresholds $\alpha = 0.3$, $\eta = 0.3$ and $\zeta = 0.5$. *Part_Pattern_Miner* mines *negative FRACTURES* starting from subtree a/b as follows. The initial pattern $P = \langle a/b \rangle$ and its bitmap is $\langle 101 \rangle$. For the child node e of b , since DoC vector of e is $\langle 1/3, 1/3, 1/3 \rangle \approx \langle 0.33, 0.33, 0.33 \rangle$, then $\text{FoSC}(\{a/b, a/b/e\}) = 2/3 > \eta$. Hence, the pattern is updated as $P = \langle a/b, a/b/e \rangle$ and its updated bitmap is $\langle 101 \rangle$. Then, we further find *negative FRACTURES* starting from $\langle a/b, a/b/e \rangle$. For the child node i of e , since $\text{FoSC}(\{a/b, a/b/e, a/b/e/i\}) = 0 < \eta$ and $\text{CoC}_{0.3}(\{a/b, a/b/e\}, \{a/b/e/i\}) = -1 < -\zeta$, then $\langle \langle a/b, a/b/e \rangle, \langle a/b/e/i \rangle \rangle$ is discovered as a *negative FRACTURE*. For the child node l of e , since $\text{FoSC}(\{a/b, a/b/e, a/b/e/l\}) = 1/3 > \eta$, the pattern, together with its bitmap, is updated. Then, we further search *negative FRACTURES* starting from $\langle a/b, a/b/e, a/b/e/l \rangle$ and discover the *negative FRACTURE* $\langle \langle a/b, a/b/e, a/b/e/l \rangle, \langle a/b/e/l/p \rangle \rangle$. The algorithm terminates when all descendants of node b are visited.

Lemma 6.4 *No subsumed negative FRACTURE will be discovered Negative_FRACTURE_Miner.* □

Proof: Suppose a *negative FRACTURE* $P_1 = \langle \langle s_1 s_2 \cdots s_{m-1} s_m \rangle, \langle s_{m+1} \rangle \rangle$ is head-subsumed by a subtree sequence $X_2 = \langle s_1 s_2 \cdots s_{m-1} s_n \rangle$ with $\text{FoSC}(X_2) \geq \eta$. Since $P_1 \sqsubset_h X_2$, $s_m \prec s_n$. According the first strategy, X_2 is examined earlier than $\langle s_1 s_2 \cdots s_m \rangle$. And since $\text{FoSC}(X_2) \geq \eta$, according to the second strategy, candidate pattern will be extended from X_2 . Hence, *negative FRACTURES* starting with $\langle s_1 s_2 \cdots s_{m-1} s_m \rangle$ will not be generated. Otherwise, it contradicts to Strategy 2. That is, no head-subsumed *negative FRACTURES* will be generated. Given a *negative FRACTURE* $P_1 = \langle X, \langle s_m \rangle \rangle$, if there exists another *negative FRACTURE* $P_2 = \langle X, \langle s_{m+1} \rangle \rangle$ s.t. $P_1 \sqsubset_t P_2$, then $s_m \prec s_{m+1}$. Then, it contradicts with the third strategy. Hence, there is neither head-subsumed nor tail-subsumed *negative FRACTURES* will be generated by *Negative_FRACTURE_Miner*. ■

The complexity of the second phase can be computed as follows. We traverse each node in the *GDT*. For each interesting subtree rooted at node x , we call the *Part_Negative_Miner*

method which needs to traverse each node in the subtree s_x . Hence, each node in GDT is traversed at most m times, where m is the depth of the node. Then, let the depth of GDT be $dep(GDT)$, the complexity of this phase is $O(|GDT| \times dep(GDT))$.

Theorem 6.5 *The algorithm `Negative_FRACTURE_Miner` discovers the set of non-subsumed negative FRACTURES completely.* $\square$

The correctness of `Negative_FRACTURE_Miner` comes from the completeness of the GDT and the correctness of `Part_FRACTURE_Miner`. Since GDT not only records each changed subtree and its DoC in each successive versions but also preserves the ancestor-descendant relationship between subtrees, all information in relevance to *negative FRACTURE* mining is maintained. The three strategies employed by `Part_FRACTURE_Miner` ensure that each potential pattern is either examined or skipped to avoid generate subsumed *negative FRACTURES*. Hence, `Negative_FRACTURE_Miner` is complete in discovering non-subsumed *negative FRACTURES*.

6.3 Applications of Negative FRACTURE

In this section, we first show the usefulness of *negative FRACTURES* with some potential applications. Then, we discuss the details of applying *negative FRACTURE* in maintaining autonomous Web mirrors.

6.3.1 Potential Applications

As we discussed before, *negative FRACTURE* can be defined in different ways. Generally, it refers to two sets of subtrees that are negatively correlated in undergoing significant changes. We first discuss applications of *negative FRACTURE* defined in the general way. Next, we present some applications of the specific *negative FRACTURE* defined in this chapter.

- **Applications of general *negative FRACTURE*.** Similar to *FRACTURES*, *negative FRACTURES* of general form capture the evolution associations of subtrees. Hence, *negative FRACTURES* can be useful in similar applications of *FRACTURES*, whereas, in the opposite way. For example, as shown in Chapter 4, the knowledge inferred from *FRACTURES* can be used in partitioning an XML document for native XML storage. For example, if two XML subtrees s_x and s_y are discovered as a

FRACTURE, we can optimally partition the XML document so that s_x and s_y are stored in the same disk page or adjacent disk page. Thus, the updating process can be more efficient in locating the positions of changed subtrees. Similarly, the general form of *negative FRACTURE* can be used in this application in the opposite way. For example, if $\langle\langle s_x \rangle, \langle s_y \rangle\rangle$ is discovered as a *negative FRACTURE*, we can optimally partition the XML document so that the two subtrees are not stored in the same disk page.

- Application of specific *negative FRACTURE* — Maintaining autonomous Web mirrors.** In this chapter, we define a particular type of *negative FRACTURE* where subtrees have ancestor-descendant relationships. We find that such type of *negative FRACTURE* are very useful in maintaining autonomous Web mirrors, which means the original Web site and mirror sites are maintained by different parties. In order to keep the “freshness” of a mirror site, a mirror site should be updated frequently with respect to the original Web site. However, if the size of the Web site directory is large, updating the complete directory frequently may incur heavy workload. Suppose when a target Web directory changes significantly, some subdirectory rarely undergoes significant changes together with it. Then, we can optimally skip updating the files in the subdirectory when maintaining the mirrors of the target directory. In order to obtain such knowledge, we can mine *negative FRACTURES* from historical directory hierarchies of Web sites. Discovered patterns are then used to design optimal strategies for autonomous Web mirror maintenance. The details of this application will be illustrated later.
- Application of specific *negative FRACTURE* — Maintaining XML path selectivity estimation.** Cost-based XML query optimization calls for accurate estimation of the selectivity of *path expressions*, which are commonly used in XML query languages to locate a subset of XML data. Bloom Histogram [144] is a framework for XML path selectivity estimation in a dynamic context. Figure 6.4 (a) shows the overview of the system used by Bloom Histogram. There are three basic components in the system: the data file, the dynamic summary and the estimator. The data file refers to the XML file which is usually large. The dynamic summary keeps the

Figure 6.4: Bloom histogram system.

necessary information of the data and the estimator is responsible for efficiently estimating the selectivity of path expression queries. The dynamic summary usually uses a path table to record the occurrences of paths. Figure 6.4 (b) shows an example path table. When an XML file evolves, the system needs to rebuild the path table before updating the estimator. Actually, some paths, in the path table, do not need to be updated if their occurrences change insignificantly. For example, we can mine *negative FRACTURES* from historical XML tree versions to discover when the support³ of some paths change significantly, the support of some path rarely undergoes significant changes together. Then, when updating the occurrence for a path, e.g., `//a/b`, whose support changes significantly, we check whether any child of node *b* forms a *negative FRACTURE* with nodes on the path. Suppose a child *c* of *b* forms a *negative FRACTURE* with *a* and *b*, we do not update the path `//a/b/c` in the path table as the support of the path may not change or change insignificantly.

6.3.2 Maintaining Autonomous Web Mirrors

We now discuss the details of applying *negative FRACTURES* in maintaining autonomous Web mirrors. Mirroring Web sites is a well-known technique that is commonly used in the Web community. As defined in [125], a mirror is a Web site or set of files on a computer server that has been copied to another computer server so that the site or files are available from more than one place. That is, a mirror site is a replica of the original site and is often placed in a different location throughout the Internet. As stated in previous studies [16] [149], mirroring of Web sites are useful at least in the following aspects:

³The support of a path refers to the number of occurrences of the path in the document. The definition of *DoC* can be adjusted easily to measure the degree of support change.

CHAPTER 6. NEGATIVE FRACTURE MINING AND APPLICATION

- **Load Reduction.** If one server is extremely popular, then a mirror may help relieve server load. For example, when a sever is overloaded with demand, its mirror sites allow the total number of requests to be spread among several servers.
- **Better Availability.** If one server is suffering hardware failure, then a mirror can serve as a backup as well as a means of disaster recovery. The availability of the original site is ensured.
- **Faster Access.** Mirror sites also increase the speed with which files or Web sites can be accessed. For example, users can access files more quickly from a server that is geographically closer to them.

There are a lot of tools for mirroring Web sites, such as rsync [118], ftpmirror [2], and CVSup [1] etc. All these tools can be used to synchronize files and directories from one location to another. They are different from each other in the data update algorithms and the communication protocols.

Web mirroring can be generally classified as *internal mirroring* and *external mirroring*. The former refers to the situation that both the original site and mirror sites are set up and maintained by the same organization, such as *www.akamai.com*. The latter means that the original site and mirror sites are autonomous. For example, many Linux user groups in various countries, such as *www.linuxforum.net* in China, mirror the site *www.samba.org* to provide quicker access for Linux fans in their own countries.

Since a mirror site is a replica of an original Web site, it is expected that it reflects the content of the original site. An obsolete mirror site leads to *stale* content, wasting users' time and causing frustration. Hence, a mirror site should be updated regularly with respect to the original site. For internal mirroring, this process can be performed efficiently by synchronizing mirror sites only if the original site evolves. However, for external mirroring where the original site and mirror sites are independent, the server holding the mirror site has to scan the original site frequently to detect the changes and update its local copy (To maintain the freshness of mirror sites, *www.samba.org* suggests external mirror sites scan it once per day). Obviously, frequent scan of the whole original site is not a light workload for the server holding the mirror site, especially when the server holds more than one mirror sites. Hence, optimized mirror maintenance strategies are needed to improve the efficiency of external mirror maintenance.

Recall that a *negative FRACTURE* indicates that a subtree rarely undergoes significant changes with its ancestor subtrees. Mining *negative FRACTURES* from a sequence of historical directory hierarchies of a Web site, we can infer which subfolders rarely change together with their ancestor folders. Since existing mirroring tools, such as rsync and Wget, allow to exclude some particular directories in mirroring, we can maintain the mirror of a Web site optimally by skipping some subdirectories according to discovered *negative FRACTURES*. For example, our experiment on the real historical directory hierarchies of the Web site “www.freebsd.org” showed that using *negative FRACTURES* mined from the historical directories hierarchies from the 13th of October, 2002 through the 15th of December, 2002 can skip 45.3% pages while only miss 0.02% pages in maintaining the directory hierarch on 22nd of December, 2002. More experiments in Section 6.4 show promising results of using *negative FRACTURES* in maintaining external mirrors.

6.3.2.1 Rule Derivation

Before applying *negative FRACTURES* in maintaining autonomous Web mirrors, we need to derive association rules from *negative FRACTURES* first. Note that, a *negative FRACTURE* indicates a subtree is negatively correlated with its ancestor subtrees in undergoing significant changes. While, a negative association rule derived from *negative FRACTURE* further indicates that when a set of ancestor subtrees changes significantly, the descendant subtree very like does not change significantly. Hence, we describe the derivation of negative association rule from *negative FRACTURE* first.

We use the symbol $\neg s_x$ to denote that subtree s_x does not change significantly. Then, given a *negative FRACTURE* $\langle\langle s_1 s_2 \dots s_{k-1} \rangle, \langle s_k \rangle\rangle$, the derived association rule has the form as $\langle s_1 s_2 \dots s_{k-1} \rangle \Rightarrow \neg s_k$. Similar to traditional association rules, the *confidence* of the rule can be defined as follows.

Definition 6.8 (Confidence of negative FRACTURE rule) Let $\langle\langle s_1 s_2 \dots s_{k-1} \rangle, \langle s_k \rangle\rangle$ be a *negative FRACTURE* satisfying the thresholds of DoC, FoSC and CoC. The confidence of the rule $\langle s_1 s_2 \dots s_{k-1} \rangle \Rightarrow \neg s_k$, denoted as $Conf(\langle s_1 s_2 \dots s_{k-1} \rangle \Rightarrow \neg s_k)$, is defined as follows:

$$Conf(\langle s_1 s_2 \dots s_{k-1} \rangle \Rightarrow \neg s_k) = \frac{FoSC(\{s_1 s_2 \dots s_{k-1}\}) - FoSC(\{s_1 s_2 \dots s_{k-1} s_k\})}{FoSC(\{s_1 s_2 \dots s_{k-1}\})}$$

□

CHAPTER 6. NEGATIVE FRACTURE MINING AND APPLICATION

Figure 6.5: *Negative FRACTURE* rule derivation and mirror maintenance algorithms.

That is, the confidence of a *negative FRACTURE* rule measures how frequently the subtree s_k changes insignificantly when all its ancestor subtrees $s_1, s_2, \dots, s_{k-1}$ change significantly. The value of confidence ranges from zero to one. The greater the value, the more probably the rule holds.

Given some threshold of confidence θ , $\langle s_1 s_2 \dots s_{k-1} \rangle \Rightarrow \neg s_k$ is a valid *negative FRACTURE* rule if $\langle \langle s_1 s_2 \dots s_{k-1} \rangle, \langle s_k \rangle \rangle$ is a *negative FRACTURE* pattern and the confidence of the rule is no less than θ . The derivation of association rules can be integrated with the *negative FRACTURE* mining. For example, Figure 6.5 (a) shows the revised *Part_Pattern_Miner* algorithm, where we derive a candidate *negative FRACTURE* rule once a *negative FRACTURE* is discovered (Line 7). Note that, the confidence of the candidate rule can be computed directly with the bitmap of the current pattern P and the vector of *DoC* values associated with the current node.

6.3.2.2 Mirror Maintenance Strategy

We now discuss our optimized mirror maintenance strategies based on discovered *negative FRACTURE*s. Recall that, as an approximate maintenance strategy, it aims to achieve high mirror maintenance efficiency while keeping the loss of freshness of mirrors as little as possible. Hence, the basic strategy is that when maintaining the mirror of a Web directory s_x , we do not skip any of its subdirectories which have positive evolution associations with it. We also do not skip any subdirectories which have neither positive nor negative evolution

association with it. Thus, only subdirectories which appear on the right-side of *negative FRACTURE* rules starting with s_x will be skipped. However, *negative FRACTURE* rules starting from s_x are not equally useful. Some of them are redundant with respect to others while some of them may even cause detriment to the quality of mirrors. Particularly, we identify the following two situations. *First, if a negative FRACTURE rule is tail-subsumed by another negative FRACTURE rule, then the subsumed rule adds no value to optimized mirror maintenance.*⁴ For example, if both $s_x \Rightarrow \neg s_y$ and $s_x \Rightarrow \neg s_z$ are valid *negative FRACTURE* rules, where $s_z \prec s_y$, then the subdirectory s_z will be skipped based on the first rule as well as the second rule. *Second, if a negative FRACTURE rule is head-subsumed by a subtree sequence satisfying FoSC threshold, then the subsumed rule is either redundant or excessive with respect to other rules in terms of mirror maintenance.* For example, given a *negative FRACTURE* rule $s_x \Rightarrow \neg s_z$ and a subtree sequence $S = \langle s_x s_y \rangle$ such that $\text{FoSC}(S) \geq \eta$ and $s_z \prec s_y$, if $s_x s_y \Rightarrow \neg s_z$ is a valid *negative FRACTURE* also, then the first rule is redundant. Otherwise, mirror maintenance strategies based on the first rule maybe too excessive because the subdirectory s_y , which contains s_z , should not be skipped considering $\text{FoSC}(S) \geq \eta$ (Our experimental results in next section investigate the excessiveness of such rules). Since our *negative FRACTURE* (i.e., *negative FRACTURE* rule) mining algorithm discovers rules which are neither tail-subsumed nor head-subsumed, the mining results can be used in maintenance directly.

Existing Web mirroring tools focus on *page-level* synchronization. For example, “rsync” tool is based on the “rsync” algorithm [118] which efficiently updates remote file by exchanging block signature information and hash search algorithm for block matching. Most of existing mirroring tools offer the option which allows to skip mirroring a particular subdirectory. Hence, our optimized strategies can be used by existing mirroring tools on top of page-level maintenance. For example, Figure 6.5 (b) shows a mirror site maintenance algorithm which uses existing mirroring tools together with *negative FRACTURE* rules. Basically, it first discovers *negative FRACTURE* rules starting with the target Web directory. Then, it performs a top-down traversal on the directory and either skips a subdirectory based on *negative FRACTURE* rules or mirror pages using existing tools. Note that, in the algorithm, we assume every parent directory is changed when traversing the directory tree in consideration of minimizing the loss in freshness.

⁴The subsumption relationship between two *negative FRACTURE* rules comes from the subsumption relationship between the corresponding *negative FRACTURES*.

Table 6.2: Parameters and Datasets.

F	Fanout of the first tree version	10
D	Depth of the first tree version	10
N	Size of the first tree version	1K
V	Number of tree versions	50
C	Change percent between tree versions	3%

(a) Parameter List

Data	Feature Parameters
D_1	$V = 50, C = 3\%$
D_2	$V = 50, C = 6\%$
D_3	$V = 50, C = 12\%$
D_4	$N = 1K, C = 3\%$
D_5	$N = 1K, C = 6\%$
D_6	$N = 1K, C = 12\%$
D_7	$N = 10K, V = 200, C = 3\%$
D_8	$N = 10K, V = 200, C = 6\%$
D_9	$N = 10K, V = 200, C = 12\%$

(b) Dataset List

Lemma 6.5 *At each node, there exists no contradictable negative FRACTURE rules that can be used.*

Two *negative FRACTURE* rules are contradictable if one rule indicates some subdirectory should be mirrored while the other indicates that it should be skipped.

Proof: *This equals to the situation that a subtree occurs in the head of a rule as well as in the tail of another rule. However, this is impossible. Suppose s_x is the current subtree and there is a subtree sequence $S = \langle s_1 \cdots s_{x-1} \rangle$ along the path arriving at s_x , with $FoSC(S) \geq \eta$. According to the algorithm of *Negative_FRACTURE_Miner*, descendant subtrees of s_{x-1} should be examined by extending S . If s_x occurs in the head of a rule, $FoSC(s_1 \cdots s_{x-1} s_x) \geq \eta$. If s_x occur in the tail of a rule, then $FoSC(s_1 \cdots s_{x-1} s_x) < \eta$. Hence, a subtree does not occur both in the head of a rule and in the tail of another rule. Thus, we have the lemma.* ■

6.4 Experimental Results

In this section, we first evaluate the performance of the *negative FRACTURE* mining algorithm with synthetic data in Section 6.4.1. Then, we examine the effectiveness of *negative FRACTURE* rule based mirror maintaining strategy with real life data in Section 6.4.2. Algorithms are implemented in the Java programming language. All experiments are conducted on a Pentium IV 2.8GHz PC with 512 MB memory. The operating system is Windows 2000 professional.

6.4.1 Performance of Negative FRACTURE Mining algorithm

Synthetic Datasets. We implemented a synthetic data generator which generates sequential tree versions. The set of parameters used by the generator are shown in Table 6.2 (a). At first, the generator constructs the first historical tree version, S^{t_1} , based on parameters F , D and N . The first tree version is generated using the following recursive process. At a given node in the tree S^{t_1} , we decide how many children to generate. The number of children is sampled uniformly at random from 0 to F . Before processing children nodes, we assign random probabilities to decide whether a child node is a leaf node, as long as tree depth is less than the maximum depth D . If a child node is not a leaf node and the total number of nodes is less than N , the process continues recursively.

Once the first tree version has been created, we create as many subsequent versions as specified by the parameter V . The tree version S^{t_i} is generated based on the version $S^{t_{i-1}}$ with parameter C . For example, if the value of C is 3% and the number of nodes in the tree S^{t_i} is N_i , then the number of inserted and deleted nodes between trees S^{t_i} and $S^{t_{i-1}}$ is $3\% \times N_i$. We randomly decide the positions of inserted and deleted nodes. Basically, there are 7 datasets used in the following experiments. The key parameter values used in generating the datasets are shown in Table 6.2 (b).

Scaleup Evaluation. As analyzed before, the two dominant factors that affect the complexity of the algorithm are the average size of trees and the number of historical versions. Hence, we evaluate the scalability of the algorithm by varying the two parameters.

We conduct the first experiment on the three data sets D_1 , D_2 and D_3 . For each data set, we vary the size of the first tree version from 1K nodes to 16K nodes. The thresholds of DoC , $FoSC$ and CoC are set as 0.001, 0.01 and 0.3 respectively. The experimental results are shown in Figure 6.6 (a). It can be observed that the scale-up feature of the mining algorithm is basically linear to the size of trees. Furthermore, the smaller the change percent, the faster the mining algorithm.

We conduct the second experiment on the three data sets D_4 , D_5 and D_6 . For each data set, we vary the number of versions from 50 to 800. The thresholds of DoC , $FoSC$ and CoC are set as 0.001, 0.01 and 0.3 respectively. The experimental results are shown in Figure 6.6 (b). We observed that when the number of versions increases, the run time increases quickly because the sizes of subsequent trees increase as well.

CHAPTER 6. NEGATIVE FRACTURE MINING AND APPLICATION

Figure 6.6: Experimental results I.

Figure 6.7: Experimental results II.

The respective scaleup feature of the two phases of the mining algorithm was studied as well on D_4 with 50 versions. Figure 6.7 (a) shows the results. Both phases take more time when the tree size increases. However, the first phase dominates the efficiency of the algorithm.

Efficiency Evaluation. We now study how the thresholds of DoC , FoC , and CoC affect the efficiency of the algorithm. We show the run time of the second phase as the thresholds do not interfere the performance of the first phase. Experiments are conducted on the data sets D_7 , D_8 and D_9 .

The results of varying the threshold of DoC are shown in Figure 6.7 (b). The $FoSC$ threshold and CoC threshold are set as 0.01 and 0.3 respectively. It can be observed that the greater the DoC threshold value, the more efficient the algorithm. This is because when the DoC threshold value is greater, less subtrees are interesting enough so that we need to

Figure 6.8: Experimental Results III.

mine less negative evolution patterns. Figure 6.8 (a) shows the results of the experiments on varying the threshold of *FoSC*. The thresholds of *DoC* and *CoC* are set as 0.001 and 0.3 respectively. We noticed that the greater the *FoSC* threshold, the more efficient the algorithm. Finally, Figure 6.8 (b) shows that the variation on the *CoC* threshold has no affection on the performance of the mining algorithm, as illustrated by the complexity analysis.

6.4.2 Performance of Mirror Maintaining Strategy

We now study the performance of our optimized mirror maintenance approach. We first present some empirical experimental results on several real-life Web sites with different features of Web directory evolution. Then, we evaluate our mirror maintenance approach from different aspects based on a large set of real Web sites.

6.4.2.1 Empirical Examples

Data set. A. Ntoulas et al. [105] collected Web pages of 154 “popular” Web sites every week from October 2002 until October 2003, for a total of 51 weeks. In their paper, they explained how the sites are selected so that it is a “representative” yet “interesting” sample of the Web. We first conduct experiments on some example sites, which have different evolutionary features of Web directories. Basically, the group of example sites, shown in Figure 6.9, are different in the evolution of their directory tree size. For each site, Figure 6.9 shows the relative weekly-size, which is normalized by the size of its first version. We observed that the evolutions of the two sites *www.sec.gov* and *www.ksc.nasa.gov* exhibit certain patterns,

Figure 6.9: Size evaluation of example sites.

e.g., generally, the size of both sites increase gradually. On the other hand, the evolutions of the two sites *www.fags.org* and *www.oracle.com* are relatively irregular. In addition, the size of the site *www.biosis.org* does not change frequently or significantly. We aim to study the effectiveness of our approach in maintaining mirrors of sites with different evolutionary characteristics.

Methodology and Metrics. For each Web site, we have total 51 historical versions of Web directory trees. We conduct experiments by using k successive versions as the training data to mine *negative FRACTURE* rules. With discovered rules, we test the performance of our approach with respect to the subsequent $(k + 1)th$ version. Basically, we follow the mirror maintenance algorithm in Figure 6.5 (b). However, instead of really mirroring a page or skipping a subdirectory, we count the number of mirrored pages, the number of skipped pages as well as the number of changed pages (both mirrored and skipped). Let $N(V_t)$ and $C(V_t)$ be the set of pages and the set of changed pages in the test version respectively, $N(V_m)$ and $C(V_m)$ be the set of pages in the mirrored version and the set of changed pages updated by our approach respectively, and $S(V_m)$ be the set of pages skipped by our approach. The following four metrics are used to measure the performance:

$$Bypass\ Ratio(BR) = \frac{|S(V_m)|}{|N(V_t)|}$$

$$Overall\ Precision(OP) = \frac{|N(V_t) \cap N(V_m)|}{|N(V_m)|}$$

$$Overall\ Recall\ (OR) = \frac{|N(V_t) \cap N(V_m)|}{|N(V_t)|}$$

$$\text{Change Recall}(CR) = \frac{|C(V_t) \cap C(V_m)|}{|C(V_t)|}$$

The *Bypass Ratio* evaluates how much efficiency can be obtained by our approach. The *Overall Precision* and *Overall Recall* measures the accuracy of our approach with respect to the whole test version, and the *Change Recall* measures the accuracy of our approach with respect to the changes to the test version. Note that, since all changed pages updated by our approach belong to the real changes to the test version, the change precision of our approach is always one. As a good approximate mirror maintenance approach, values of all the four metrics should be as high as possible.

Results. Figure 6.9 (b) shows the results of the five example sites by using 20 successive versions as the training data and the subsequent version as the test data. The results are averaged by running the experiment for 10 times on sequential 20 versions. From the results, we have the following observations. First of all, with respect to the size of the whole test version, our mirror maintenance approach loses slight freshness. For example, both the *Overall Precision* and the *Overall Recall* are high for all sites. Secondly, our approach probably gains more efficiency improvement on sites with certain patterns in their evolution than sites with irregular evolution. For example, the *Bypass Ratio* values of the sites *www.sec.gov* and *www.ksc.nasa.gov* are higher than those of the sites *www.fags.org* and *www.oracle.com*. Similarly, the *Change Recall* value of the former two sites are higher than those of the latter two sites. Thirdly, for a site which does not change frequently, i.e. *www.biosis.org*, our approach achieves a high *BR* value but a low *CR* value probably because it is difficult to predict the location of changes for a site which rarely changes.

6.4.2.2 Performance of Mirror Maintenance

Data Set. In this experiment, we evaluate the performance of our mirror maintenance approach from different aspects based on the whole set of Web sites collected by A. Ntoulas et al. However, we noticed that data of some Web sites are incomplete. For example, the Web server accommodating the site *www.interplay.com* blocked Web crawling. Thus, we perform a preprocess on the set the of Web sites collected by A. Ntoulas et al. to filter those sites with more than 5 blocked historical versions. The list of remaining 102 sites used in our study is given in Appendix B.

Methodology and Metrics. The following experiments are conducted on the set of Web sites. (a) We fix the size of the training data and vary the *distance* between training

CHAPTER 6. NEGATIVE FRACTURE MINING AND APPLICATION

Figure 6.10: Performance of mirror maintenance I.

Figure 6.11: Performance of mirror maintenance II.

data and test data. For example, *negative FRACTURE* rules are mined from the first k versions and performance is evaluated with respect to the $(k + 1)th$ version, the $(k + 2)th$ version, and so on. This experiment aims to study the temporal validity of discovered rules. (b) We vary the size of the training data and use the subsequent version as the test data to study how the training size affects the performance. (c) We evaluate the effectiveness of our approach on sites from different domains, such as *.com*, *.gov*, *.org* etc. (d) We study how the thresholds (e.g., α , η , θ) affect the performance of our approach. (e) We evaluate the performance of mirror maintenance strategies based on subsumed *negative FRACTURE* rules. The same four metrics, *Bypass Ratio*, *Overall Precision*, *Overall Recall* and *Change Recall* are used to evaluate the performance.

Results. Figure 6.10 (a) shows the results of the first experiment which varies the distance between the training data and the test data. Similarly, the results corresponding to each distance are averaged by running experiments 10 times on sequential data. It can be observed that our approach skips around one third pages of a site on average and loses slight “freshness”, e.g., values of *OP*, *OR* and *CR* are high (Note that, the results are obtained by

using the same thresholds on all sites. As we show later, the performance on each site can be tuned by setting appropriate thresholds). Furthermore, the performance of our approach does not deteriorate obviously with the increase of the distance. This probably because that discovered evolution associations hold for most of sites in the tested period. Based on the results, we can save some expenditure by mining rules for once and utilizing them for multiple times. Figure 6.10 (b) shows the results of the experiment which varies the size of the training data. We noticed that when the size of the training data increases, the *BR* value of our approach slightly decreases. It's probably because we fix the thresholds for training data of different size. As a result, when the size of training data increases, different sets of rules, based on which fewer pages are skipped, are discovered. Accordingly, fewer changed pages are missed by our approach, which results in the improvement of the *CR* value. In Figure 6.10 (c), we present the performance of our approach with respect to sites from different domains. According to the results, it indicates that our approach works better on sites from domains like *.org* and *.gov* than on sites from domains such as *.com* and *.edu*. The reason may be that Web sites from the latter two domains evolves more irregularly.

Figure 6.11 (a) shows the resulting *BR* value of sites on variation of the *DoC* threshold. Each point in the figure corresponds to a site which achieves highest *BR* value at current threshold (relatively highest in the *DoC* threshold range of [0.01, 0.05]). It indicates that as traditional association rule mining, appropriate thresholds are site-dependent. The relationship between other metrics and thresholds are similar. We also conduct this experiment on a particular Web site www.freebsd.org. The default values of *DoC*, *FoSC* and *confidence* are 0.01, 0.1, and 0.6 respectively. We first vary the threshold of *DoC* from 0.01 to 0.06. Figures 6.11 (b) and (c) respectively show the *BR* and *CR* on four training data sets which have 10 versions, 11 versions, 12 versions and 13 versions respectively. We observed that when the *DoC* threshold increases, the *BR* increases as well, while the *CR* value does not decrease too much in a certain *DoC* threshold range. The possible reason is that when *DoC* threshold is low, subtrees are easy to undergo significant changes and hard to form the negative change correlations. Hence, less rules may be found and the maintenance method skips less pages. On the contrary, when the *DoC* threshold is large, more rules may be found and more pages are skipped by mirror maintenance algorithm, while more changed pages are missed. That's the reason that the *CR* decreases when *DoC* threshold is too

Figure 6.12: Performance of mirror maintenance III.

large. We also conduct experiments to vary the threshold of *FoSC* and *Confidence*. Since the length of the training data is limited, the variation of these thresholds do not effect the *BR* or *CR* obviously. Actually, the effect on *BR* and *CR* is too slight to be shown clearly in figures. For example, when the threshold of *FoSC* is varied from 0.2 to 0.8, both *BR* and *CR* on the four training data sets keep same.

Figure 6.12 (a) shows the performance of mirror maintenance based on tail-subsumed rules. We normalize the results with respect to the performance of our approach. It can be observed that maintenance strategies based on subsumed rules gain less efficiency than our approach does (e.g., $BR1/BR < 1$). Correspondingly, it sacrifices less freshness (e.g., $CR1/CR > 1$). However, the magnitude of the former is larger than the latter. Similarly, Figure 6.12 (b) shows the normalized performance of maintenance strategies based on head-subsumed rules. Compared to our approach, such strategies skip more pages and miss more changed pages. However, the magnitude of the efficiency they improve is less than that of the freshness they lose. Therefore, as an approximate mirror maintenance strategy which aims to achieve high efficiency as well as sacrifice slight freshness, mirror maintenance based on non-subsumed rules works better than approaches based on subsumed ones.

6.5 Summary

In this chapter, we extend the *FRACTURE* mining model to discover subtrees that are negatively correlated in undergoing significant changes, which is called *negative FRACTURE*. Similar to *FRACTURE* mining, *negative FRACTURE* mining focuses on discovering intra-tree structural evolution associations. While *FRACTURE* is defined on subtrees which may

CHAPTER 6. NEGATIVE FRACTURE MINING AND APPLICATION

or may not be structurally related, we particularly discuss *negative FRACTUREs* which are defined on subtrees with ancestor-descendant relationships.

Two evolution metrics, *FoSC* and *CoC*, which measure the frequency of significant changes and the correlation of significant changes respectively are defined to evaluate the interestingness of *negative FRACTURE* patterns. A data mining algorithm, which employs a data structure *GDT* to maintain structural delta information, is developed to discover *negative FRACTUREs* with the divide-and-conquer strategy. Potential applications of *negative FRACTURE* patterns are discussed also. Particularly, we apply *negative FRACTUREs*, mined from historical Web directory hierarchies, in maintaining autonomous Web mirrors.

The performance of the proposed data mining algorithm and the effectiveness of *negative FRACTURE* rule based mirror maintenance method are studied extensively based on experiments conducted on both synthetic datasets and real-life datasets. The experimental results demonstrate that the developed algorithm can discover the set of *negative FRACTUREs* efficiently. The *negative FRACTURE* rule based mirror maintenance method effectively reduces the workload without sacrificing too much “freshness” of the mirrors.

Chapter 7

SEASON Mining and Application

So far, we have introduced two types of intra-tree associative evolutionary structures — *FRACTURE* and *negative FRACTURE*, which consider the evolution associations of subtrees. As discussed in Chapter 3, associative evolutionary structures can also be mined by considering the evolution association among trees. In this chapter, we propose an inter-tree associative evolutionary structures called *SEASON* (**SEmAntic Sequence of structural evOlutioN**), which extracts sequential associations from historical tree versions. While *FRACTURE* and *negative FRACTURE* patterns view changes to tree structures from the *structural variation* perspective, *SEASON* patterns consider changes from the *semantic variation* perspective. Given an evolutionary tree database which consists of a set of historical tree sequences, where each tree represents some semantic meaning, a *SEASON* pattern captures the frequent transitions between semantics. *SEASON* patterns are beneficial in applications where the knowledge of semantic evolution is required, such as XML query caching, prefetching, XML user clustering etc. A preliminary version of this chapter appeared in our previous publication [35].

In the rest of this chapter, we begin with some preliminary definitions of *SEASON* pattern. In Section 7.2, we describe the algorithm for *SEASON* pattern mining. In Section 7.3, we discuss the potential applications of *SEASON* patterns and the details of applying *SEASONS* in designing appropriate XML query caching replacement strategies. We evaluate the performance of the algorithm for *SEASON* mining and the effectiveness of *SEASON* based replacement strategies in Section 7.4. Finally, we conclude this chapter with a summary in Section 7.5.

Figure 7.1: Sequence of semantic trees.

7.1 SEASON Definition

In this section, some preliminary definitions are described first. Then, we formally define the notion of *SEASON* and the problem of *SEASON* mining. As exemplified in Chapter 3, *SEASON* patterns are mined from an evolutionary tree database where each tree represents some semantic meaning. Particularly, we refer to a tree with semantics as a semantic tree.

Definition 7.1 (Semantic tree) A semantic tree is a node-labeled rooted tree, which is a 3-tuple $S = \langle N, E, L \rangle$, where N is a set of nodes, $E \subseteq N \times N$ is the set of edges and $L : N \rightarrow \Lambda$ is a node labelling function which assign each node $n \in N$ a label $L(n) \in \Lambda$. For each edge $(n_1, n_2) \in E$, $L(n_1)$ semantically contains $L(n_2)$, i.e., $L(n_2)$ is either an attribute or a subobject of $L(n_1)$. $\square$

That is, a node-labeled tree is a semantic tree if the label of each node semantically contains the label of any of its child nodes. As shown in Chapter 3, XML queries can be modeled as semantic trees which represent inquired XML information. Figure 7.1 shows five semantic trees. For example, the tree S_1 represents the information of book title and book author.

Correspondingly, an evolutionary semantic tree database can be defined as a set of semantic tree sequences¹.

Definition 7.2 (Evolutionary Semantic Tree Database) Let $\Sigma = \{S_1, S_2, \dots, S_n\}$ be a set of semantic trees. A semantic tree sequence Q is an ordered list of semantic trees, denoted as $Q = \langle S_1 S_2 \dots S_l \rangle$, where $S_i \in \Sigma$ ($1 \leq i \leq l$). $\mathcal{D}$ is an evolutionary semantic tree database on Σ if $\mathcal{D} = \{Q = \langle S_1 S_2 \dots S_l \rangle \mid \forall S_i \in Q, S_i \in \Sigma\}$. $\square$

¹An Evolutionary Semantic Tree Database is similar to the Evolutionary Tree Database defined in Definition 3.2 in Chapter 3, except that we request every tree should represent meaningful semantics here.

Table 7.1: An evolutionary semantic tree database.

<i>ID</i>	<i>Semantic tree sequence</i>
Q_1	$\langle S_1 S_4 S_5 \rangle$
Q_2	$\langle S_2 S_4 S_5 \rangle$
Q_3	$\langle S_2 S_4 \rangle$
Q_4	$\langle S_1 S_3 \rangle$
Q_5	$\langle S_1 S_4 S_2 \rangle$

For example, let $\Sigma = \{S_1, S_2, S_3, S_4, S_5\}$ be the set of semantic trees as shown in Figure 7.1. Then, Table 7.1 is an evolutionary semantic tree database on Σ , where the first column *ID* contains the identities of semantic tree sequences.

Note that, different semantic trees may have similar semantics. For example, both semantic trees S_1 and S_3 in Figure 7.1 are related to the information of book author. And both semantic trees S_2 and S_5 are about the information of book publisher. Then, given a collection of semantic trees, we can group them based on their semantics to generate a set of representative semantic clusters. After categorizing each tree to a semantic cluster, a semantic tree sequence can be transformed to be a semantic cluster sequence.

Definition 7.3 (Semantic cluster sequence) Let $\Sigma = \{S_1, S_2, \dots, S_n\}$ be a set of semantic trees. Suppose there exists a function $f : \Sigma \rightarrow \Upsilon$ that categorizes each tree in Σ to a semantic cluster in $\Upsilon = \{C_1, C_2, \dots, C_m\}$ (e.g., for any $i \in [1, n]$, there exists one and only $j \in [1, m]$ s.t. $f(S_i) = C_j$). Then, given any semantic tree sequence $Q = \langle S_1 S_2 \dots S_l \rangle$ where $S_i \in \Sigma$, it can be transformed into a semantic cluster sequence $C(Q) = \langle C_1 C_2 \dots C_l \rangle$, where $C_1 = f(S_1), \dots, C_l = f(S_l)$. $\square$

Example 7.1 Consider the set of semantic trees in Figure 7.1. Suppose they can be categorized into three semantic clusters as follows: $f(S_1) = C_1$, $f(S_2) = C_2$, $f(S_3) = C_1$, $f(S_4) = C_3$, $f(S_5) = C_2$, where C_1 represents the information of book author, C_2 represents the information of book publisher and C_3 represents the information of book chapter. Thus, for the first semantic tree sequence Q_1 in Table 7.1, it can be transformed to be the semantic cluster sequence $C(Q_1) = \langle f(S_1)f(S_4)f(S_5) \rangle = \langle C_1 C_3 C_2 \rangle$. $\blacksquare$

Given a set of semantic trees Σ and a function $f : \Sigma \rightarrow \Upsilon$ that categorizes each tree in Σ to a semantic clusters in Υ , a semantic tree sequence $Q = \langle S_1 S_2 \dots S_l \rangle$ is said to support a

CHAPTER 7. SEASON MINING AND APPLICATION

a semantic cluster sequence $C = \langle C_1 C_2 \dots C_k \rangle$, denoted as $Q \sqsubseteq C$, if there exists an integer $i (1 \leq i \leq l - k + 1)$ such that $f(S_i) = C_1, f(S_{i+1}) = C_2, \dots, f(S_{i+k-1}) = C_k$. Then the support of a semantic cluster sequence in an evolutionary semantic tree database can be defined as follows.

Definition 7.4 (Support) Let $\mathcal{D}$ be an evolutionary semantic tree database on Σ . Suppose there exists $f : \Sigma \rightarrow \Upsilon$. Then, given a semantic cluster sequence C on Υ , the support of C in $\mathcal{D}$, denoted as $\text{supp}_{\mathcal{D}}(C)$, is

$$\text{supp}_{\mathcal{D}}(C) = \frac{|\{Q | Q \sqsubseteq C, Q \in \mathcal{D}\}|}{|\{Q | Q \in \mathcal{D}\}|}$$

□

Example 7.2 Suppose the semantic trees in Figure 7.1 are clustered as in Example 7.1. Then, given a semantic cluster sequence, $C = \langle C_1 C_3 \rangle$, its support in the evolutionary semantic tree database $\mathcal{D}$ in Table 7.1 is $2/5$, since two semantic tree sequences, $Q_1 = \langle S_1 S_4 S_5 \rangle$ and $Q_5 = \langle S_1 S_4 S_2 \rangle$, support it. ■

Definition 7.5 (SEASON) Given an evolutionary semantic tree database $\mathcal{D}$ on Σ , $f : \Sigma \rightarrow \Upsilon$, and a real number ξ ($0 \leq \xi \leq 1$) as the threshold of support, a semantic cluster sequence $C = \langle C_1 C_2 \dots C_k \rangle$ ($C_i \in \Upsilon$) is called a *SEASON* pattern with respect to $\mathcal{D}$ if $\text{supp}_{\mathcal{D}}(C) \geq \xi$. □

Example 7.3 Again, suppose the semantic trees in Figure 7.1 are clustered as in Example 7.1. Given the evolutionary semantic tree database in Table 7.1 and the support threshold $\xi = 0.2$, the semantic cluster sequence $\langle C_1 C_3 \rangle$ is a *SEASON* pattern as its support is $2/5 = 0.4 > \xi$.

The *SEASON* pattern in **Example 7.3** reflects the frequent transition between the two kinds of semantics: book publisher (C_2) and book chapter (C_3). The usefulness of such knowledge will be discussed with some potential applications in Section 7.3. Then, the problem of *SEASON* mining can be formally stated as follows: Let $\mathcal{D}$ be an evolutionary semantic tree database on a set of semantic trees Σ . Given a support threshold ξ ($0 \leq \xi \leq 1$), the **problem of SEASON mining** is first to find the function $f : \Sigma \rightarrow \Upsilon$ that categorizes trees in Σ to semantic clusters in $\Upsilon = \{C_1, C_2, \dots, C_m\}$ such that the semantics of trees within a cluster are similar to one another and different from the semantics of trees in other

clusters, and then to find the set of semantic cluster sequences $\{C = \langle C_1 C_2 \dots C_k \rangle \mid \forall C_i \in \Upsilon, \text{supp}_{\mathcal{D}}(C) \geq \xi\}$ ².

The problem of *SEASON* mining is different from traditional frequent sequential pattern mining in the following two critical aspects. First, the data type we considered is different. Traditional frequent sequential pattern mining considers one-dimensional data, such as items, while *SEASON* mining considers tree structured data. Second, the definition of *support* is different. In traditional frequent sequential pattern mining, a sequence supports another sequence if the latter is embedded in the former. While, in *SEASON* mining, a semantic tree sequence supports a semantic cluster sequence only if the latter is embedded in a semantic cluster sequence that is transformed from the former. Consequently, existing frequent sequential pattern mining algorithms cannot be applied here directly to discover *SEASONS*. We propose a new data mining algorithm for *SEASON* mining in the following section.

7.2 SEASON Mining

In this section, we first present the overview of *SEASON* mining. Then, we discuss the details of respective mining phases.

7.2.1 Procedure Overview

Given a semantic tree sequence database $\mathcal{D}$ and some user-specified support threshold ξ , the discovery of *SEASON* patterns consists of the following two phases.

- **Phase I: semantic tree clustering.** In order to transform semantic tree sequences to calculate the support of semantic cluster sequences, we need to cluster the set of semantic trees in $\mathcal{D}$ first. Hence, a clustering algorithm which clusters tree structures based on semantics is needed in this phase.
- **Phase II: pattern discovery.** After getting the results of Phase I, each semantic tree sequence can be transformed to be a semantic cluster sequence. Then, the second phase mines *SEASONS* from the transformed database with respect to the support threshold ξ .

²Note that, different sets of *SEASON* patterns will be found if the function $f : \Sigma \rightarrow \Upsilon$ is implemented differently. The *goodness* of different sets of *SEASON* patterns depends on the applications (e.g., a set of patterns is more beneficial than the others in some particular application).

7.2.2 Semantic Tree Clustering

We discuss the phase of semantic tree clustering first. Given a semantic tree sequence database $\mathcal{D}$ on a set of semantic trees Σ , the objective of this phase is to implement the function $f : \Sigma \rightarrow \Upsilon$ which categorizes each semantic tree in Σ to a semantic cluster in Υ . The function f can be implemented by clustering semantic trees in Σ based on semantics. In other words, given a collection of semantic trees $\{S_1, S_2, \dots, S_n\}$, we aim to find a clustering f that forms a partition $\{C_1, C_2, \dots, C_m\}$ of $\{S_1, S_2, \dots, S_n\}$ such that trees in each C_i are semantically as close as possible.

7.2.2.1 Clustering Criteria

There are several tree structure clustering methods [92, 49] proposed in the literature. One of the factors which differentiate them from each other is the clustering criteria. We review some existing similarity measures defined on tree structures by dividing them into two basic categories.

- **Node-based Similarity.** Node-based similarity measures the proximity of two trees based on shared nodes. For example, the work in [49] proposed to measure the similarity between two trees using *tree distance*, which is a sequence of node insertion, node deletion and node relabeling operations etc. Then, the less operations in the tree distance, the closer the two trees are. However, as observed by [92], tree distance based similarity may not be able to distinguish trees of different semantics. For example, as shown in Figure 7.2 (a), the tree distance between S_1 and S_2 is one (a node relabeling), while the tree distance between S_2 and S_3 is one also. Although S_1 and S_2 should be semantically closer as they represent the information of the same object A , the tree distance based similarity cannot identify the correct clustering.
- **Edge-based Similarity.** Edge-based similarity measures the proximity of two trees based on shared edges. For example, the work in [92] proposed an adapted *Jaccard coefficient* defined on edges, $dist(S_1, S_2) = 1 - \frac{|sg(S_1) \cap sg(S_2)|}{\max\{|sg(S_1)|, |sg(S_2)|\}}$, where $sg(S_i)$ is the set of edges in tree S_i . Thus, the more edges the two trees share, the closer the two trees are. Edge-based similarity measure is more accurate than node-based measure as it considers not only the nodes but also the edges connecting the nodes. For example, using the adapted *Jaccard coefficient* to measure the distance between

Figure 7.2: Previous similarity measures.

the trees in Figure 7.2 (a), we will get that the distance between S_1 and S_2 is 0.5, while the distance between S_2 and S_3 is 1. However, we observed that edge-based measure also failed to distinguish trees of different semantics in some cases. For example, as shown in Figure 7.2 (b), using the adapted *Jaccard coefficient*, the distance between S_1 and S_2 is as same as the distance between S_2 and S_3 . But, S_1 and S_2 should be semantically closer as they represent the information of an object A which has two sub-objects B and C .

Based on the above discussion, we notice that both nodes and edges provide limited information such that similarities based on these elements cannot achieve good accuracy in clustering semantic trees. Hence, we consider to measure similarity based on some elements which contain more information than nodes and edges. Particularly, in our method, we base our clustering criteria on *rooted subtree* of semantic trees, which is defined as follows.

Definition 7.6 (Rooted Subtree) Given a semantic tree $S = \langle N, E, L \rangle$, let $\text{Root}(S)$ be the root of S , a rooted subtree $RS = \langle N', E', L' \rangle$ is a subtree of S if it satisfies the following conditions: 1) $\text{Root}(RS) = \text{Root}(S)$; 2) $N' \subseteq N$, $E' \subseteq E$. $\square$

For example, in the Figure 7.2 (b), the tree S_1 is a rooted subtree of S_2 , while S_3 is not a rooted subtree of S_2 .

Note that, rooted-subtree-based similarity scheme has certain advantages over the node-based and edge-based ones. For example, consider the trees in Figure 7.2 (a) again. S_1 and S_2 share the common rooted subtree RS with a root node A and a leaf node B , while S_3

does not share any rooted subtree with S_1 or S_2 . Hence, S_1 and S_2 should be more similar. Consider the trees in Figure 7.2 (b) again. S_1 and S_2 share the rooted subtree RS with the root A and two child nodes B and C . S_3 share the rooted subtree RS with the root A and a leaf node B . Since the size of the rooted subtree shared by S_1 and S_2 is larger, S_1 and S_2 should be closer.

7.2.2.2 Clustering Method

We now discuss the clustering method. Existing tree structure clustering approaches usually employ the agglomerative clustering technique [92, 49]. They compute the similarity between all pairs of clusters and then merge the most similar pair. Recently, the type of “cluster-centered” method was proposed in the area of document clustering [143, 64]. A “clustered-centered” method constructs clusters by measuring the “cohesiveness” of clusters directly using frequent patterns. For example, FIHC [64] mined frequent terms of a collection of documents first and then clustered the documents according to the frequent terms they contain. The motivation is that there are some frequent terms for each cluster (topic) in the document set, and different clusters share few frequent terms. We employ such a clustering strategy here to cluster semantic trees as it is revealed in [64] that the “cluster-centered” method can distinguish documents of different semantics better and achieve higher clustering accuracy.

In order to cluster semantic trees with the “cluster-centered” method which uses shared rooted subtrees as the similarity scheme, we need to discover frequent rooted subtrees from a collection of semantic trees first. We borrow the algorithm FastXMiner [157] to discover frequent rooted subtrees from semantic trees. According to [157], frequent rooted subtrees can be defined as follows.

Definition 7.7 (Frequent Rooted Subtree) *Given a set of semantic trees $\Sigma = \{S_1, S_2, \dots, S_n\}$, and a real number δ ($0 \leq \delta \leq 1$), which is called minimum rooted subtree support, the support of a rooted subtree RS , denoted as $\text{supp}(RS)$, is the fraction of semantic trees that include it. RS is a frequent rooted subtree if $\text{supp}(RS) \geq \delta$. $\square$*

Example 7.4 In order to show our clustering method clearly, we use another set of semantic trees, in the upper part of Figure 7.3, as the running example. If the threshold δ is 0.2, then four frequent rooted subtrees, RS_1 , RS_2 , RS_3 and RS_4 , will be discovered as in

Figure 7.3: Initial clusters.

the lower part of Figure 7.3. Each arrowed line from a semantic tree to a frequent rooted subtree indicates that the semantic tree supports the rooted subtree. ■

After discovering frequent rooted subtrees from a collection of semantic trees, the clustering method constructs clusters in the following three steps: initializing clusters, disjointing clusters and pruning clusters.

Initializing. In this step, we construct initial clusters for each mined frequent rooted subtree. We use the frequent rooted subtrees as the labels of the initial clusters. A semantic tree is assigned to an initial cluster if the label of the cluster is the maximal frequent rooted subtree supported by the semantic tree³. For example, consider the semantic tree S_8 in Figure 7.3. Two frequent rooted subtrees are supported by it, RS_2 and RS_3 . We assign S_8 to the initial cluster RS_3 since the label of RS_2 is not a maximal frequent rooted subtree supported by S_8 . Note that, if a semantic tree does not support any frequent rooted subtree, e.g., the semantic tree S_4 in Figure 7.3, then it will be treated as an outlier because the semantics of the tree does not close to any cluster. Given the set of semantic trees and discovered frequent rooted subtrees in Figure 7.3, the results of the initializing step are shown in the bottom of Figure 7.3, where four initial clusters, C_1 , C_2 , C_3 and C_4 , are created.

Initial clusters may not be disjoint because a semantic tree may support more than one maximal frequent rooted subtrees. For example, the semantic tree S_9 supports two maximal

³A frequent rooted subtree is not maximal w.r.t. a semantic tree if it is included by another frequent rooted subtree supported by the semantic tree.

frequent rooted subtrees, RS_1 and RS_3 . Thus, S_9 is assigned to two corresponding initial clusters C_1 and C_3 . We discuss how to make the initial clusters disjoint in the next step.

Disjointing. For each semantic tree, we identify the best initial cluster and keep the tree only in the best cluster. We define the *goodness* of a cluster for a semantic tree based on the intra-cluster dissimilarity. That is, we remove a semantic tree from all the clusters but the one to which adding the tree results in the minimal intra-cluster dissimilarity. We measure the intra-cluster dissimilarity based on the number of infrequent edges in the cluster⁴. That is, we merge all semantic trees in a cluster into a tree structure. For the merged tree, each edge e is associated with a *cluster support*, denoted as $supp_C(e)$, which is the fraction of semantic trees containing it. For example, Figure 7.4 (a) shows the merged tree structure for the initial cluster C_1 in Figure 7.3. The numbers on the edges are their absolute cluster support values. Given a *minimum edge support* χ ($0 \leq \chi \leq 1$), an edge in a merged tree is infrequent if its *cluster support* is less than χ . Then, we define the intra-cluster dissimilarity as follows.

Definition 7.8 (Intra-cluster dissimilarity) Given a merged tree M_i of a cluster C_i and a minimum edge support χ , the intra-cluster dissimilarity of C_i , denoted as $Intra(C_i)$, is

$$Intra(C_i) = \frac{|\{e \in E(M_i) | supp_{C_i}(e) < \chi\}|}{|\{e \in E(M_i)\}|}$$

where $E(M_i)$ is the set of edges of M_i . □

The value of $Intra(C_i)$ ranges from 0 to 1. The higher the $Intra(C_i)$, the more dissimilar the semantic trees in cluster C_i .

Example 7.5 Figure 7.4 shows the merged trees for initial clusters C_1 and C_3 in Figure 7.3. Suppose the *minimum edge support* χ be 0.6. In the merged tree of C_1 , there are three infrequent edges: $(book, year)$, $(book, author)$ and $(author, ln)$. Hence, $Intra(C_1) = 3/4 = 0.75$. Similarly, the edge $(book, title)$ is an infrequent edge in the merged tree of C_3 . Hence, $Intra(C_3) = 1/3 \approx 0.33$. ■

We then make the clusters disjoint by assigning a semantic tree to a cluster which has the smallest intra-cluster dissimilarity value. That is, a semantic tree S_i is kept in cluster C_j if

$$C_j = \operatorname{argmin}_{C_j \in C, S_i \in C_j} Intra(C_j)$$

⁴We highlight that our fall-back on measuring dissimilarity based on edges in this step will not damage the clustering quality significantly since the initializing step based on rooted subtrees basically decides the assignment of semantic trees to clusters.

Figure 7.4: Intra-cluster dissimilarity.

Example 7.6 For example, given the set of semantic trees and initial clusters in Figure 7.3, S_9 is assigned to both clusters C_1 and C_3 . As shown in Example 7.5, grouping S_9 in C_1 generates the $Intra(C_1) = 0.75$, whereas grouping S_9 in C_3 results in the $Intra(C_3) = 0.33$. Hence, we remove S_9 from cluster C_1 . ■

After this step, clusters are not overlapping any longer. The initial clusters in Figure 7.3 are adjusted as shown in Figure 7.5, where each cluster is represented as a merged tree of all semantic trees in it.

Pruning. If the *minimum rooted subtree support* δ is small, many frequent rooted subtrees will be mined from the collection of semantic trees. Then, some of the rooted subtrees are semantically close, e.g., RS_2 and RS_3 in Figure 7.3, which results in semantically close clusters, e.g., C_2 and C_3 in Figure 7.5. Hence, in this step, we perform cluster pruning to merge close clusters.

We measure the similarity of a cluster to another cluster based on the number of frequent edges shared by the two clusters. Given a *minimum edge support* χ , the set of frequent edges of cluster C_i , denoted as F_{C_i} , are the edges in the merged tree of C_i with their *cluster support* no less than χ . That is, $F_{C_i} = \{e | e \in E(M_i) \ \& \ supp_{C_i}(e) \geq \chi\}$.

Example 7.7 Consider the cluster C_3 in Figure 7.5. Let the χ be 0.6. Then the two edges, $(book, author)$ and $(author, ln)$, are frequent because each edge has the cluster support $2/2 = 1 \geq \chi$. Hence, $F_{C_3} = \{(book, author), (author, ln)\}$. ■

Then, the similarity of one cluster to another cluster can be defined as follows.

Definition 7.9 (Cluster Similarity) Given two clusters C_i and C_j , the cluster similarity

CHAPTER 7. SEASON MINING AND APPLICATION

Figure 7.5: Clusters after disjointing.

of C_i to C_j , denoted as $Sim(C_i \rightarrow C_j)$, is,

$$Sim(C_i \rightarrow C_j) = \frac{|\{e|e \in F_{C_i}, e \in F_{C_j}\}| - |\{e|e \in F_{C_i}, e \notin F_{C_j}\}|}{|\{e|e \in F_{C_i}\}|} + 1$$

where F_{C_i} is the set of frequent edges of C_i . □

That is, the more frequent edges of C_i are frequent in C_j , the closer C_i is to C_j . The value of the first term, $\frac{|\{e|e \in F_{C_i}, e \in F_{C_j}\}| - |\{e|e \in F_{C_i}, e \notin F_{C_j}\}|}{|\{e|e \in F_{C_i}\}|}$, ranges from -1 to 1 . If all frequent edges in cluster C_i are frequent as well in cluster C_j , then the value of $Sim(C_i \rightarrow C_j)$ is 1 . If none of the frequent edge in cluster C_i is frequent in cluster C_j , then the value of $Sim(C_i \rightarrow C_j)$ is -1 . To avoid negative similarity values, we add the term $+1$. As a result, the range of $Sim(C_i \rightarrow C_j)$ is $[0, 2]$. If $Sim(C_i \rightarrow C_j)$ is greater than 1 , cluster C_i is semantically close to cluster C_j . Note that, the cluster similarity is asymmetrical.

Then, we define the inter-cluster similarity between C_i and C_j as the geometric mean of the two similarities: $Sim(C_i \rightarrow C_j)$ and $Sim(C_j \rightarrow C_i)$.

Definition 7.10 (Inter-cluster Similarity) Given two clusters C_i and C_j , the inter-cluster similarity between C_i and C_j , denoted as $Inter(C_i, C_j)$, is,

$$Inter(C_i, C_j) = \sqrt{Sim(C_i \rightarrow C_j) \times Sim(C_j \rightarrow C_i)}$$

□

According to [64], the advantage of the geometric mean is that the inter-cluster similarity will be high only if both values of $Sim(C_i \rightarrow C_j)$ and $Sim(C_j \rightarrow C_i)$ are high. Since the range of cluster similarity is $[0, 2]$, the range of $Inter(C_i, C_j)$ is also $[0, 2]$. Obviously, the inter-cluster similarity has the symmetry property.

Figure 7.6: Semantic clusters.

Higher value of inter-cluster similarity imply higher similarity between two clusters. An $Inter(C_i, C_j)$ value below 1 implies that the weight of dissimilar item (e.g., $Sim(C_i \rightarrow C_j)$ or $Sim(C_j \rightarrow C_i)$) has exceeded the weight of similar item. Hence, we merge two clusters C_i and C_j if $Inter(C_i, C_j)$ is greater than 1.

Example 7.8 Consider the cluster C_2 and C_3 in Figure 7.5 again. Let the χ be 0.6, $F_{C_2} = \{(book, author)\}$ and $F_{C_3} = \{(book, author), (author, ln)\}$. Thus, $Sim(C_2 \rightarrow C_3) = (1 - 0)/1 + 1 = 2$ because the frequent edge $(book, author)$ in C_2 is frequent as well in C_3 . Whereas, $Sim(C_3 \rightarrow C_2) = (2 - 1)/2 + 1 = 1.5$ because the frequent edge $(author, ln)$ in C_3 is infrequent in C_2 . Then, $Inter(C_2, C_3) = \sqrt{2 \times 1.5} \approx 1.73$ and we merge the two clusters C_2 and C_3 . ■

The final clustering result is shown in Figure 7.6. That is, given the set of ten semantic trees as shown in the upper part of Figure 7.3, we implement the following function f to categorize each semantic tree to a collectively represented semantic cluster.

$$f(S_i) = \begin{cases} C_1, & \text{if } i = 1, 3, 7 \\ C_2, & \text{if } i = 2, 8, 9 \\ C_3, & \text{if } i = 5, 6, 10 \\ Null, & \text{if } i = 4 \end{cases}$$

7.2.3 Frequent Cluster Sequence Discovery

After implementing the function $f : \Sigma \rightarrow \Upsilon$, the semantic tree sequence database $\mathcal{D}$ can be transformed to be a database of semantic cluster sequences. Then, the objective of the second phase is to discover *SEASON* patterns from the transformed database with respect to some given support threshold ξ . Since *SEASON* patterns are actually frequent semantic cluster sequences, the objective of this phase is similar to the problem of traditional frequent

sequential pattern mining. As we reviewed in Chapter 2, many data mining approaches have been proposed in the literature for frequent sequential pattern mining [8, 129, 161, 70, 111]. However, we adopt the *FS-Miner* [57] algorithm because of the following reason. Most of existing frequent sequential pattern mining algorithms discover frequent *embedded* subsequences. For example, $\langle ace \rangle$ will be regarded as a subsequence of $\langle abcde \rangle$. While, both *FS-Miner* and our *SEASON* mining are interested in frequent *induced* subsequences only, such as $\langle abc \rangle$ of $\langle abcde \rangle$.

We illustrate how *FS-Miner* discovers *SEASONs* from the transformed evolutionary semantic cluster database with an example in the following. Readers can refer to [57] for the details of *FS-Miner*. For example, given the evolutionary semantic tree database in Table 7.1 and the clustering in Example 7.1, Figure 7.7 (a) shows the transformed evolutionary semantic cluster database.

Given an evolutionary semantic cluster database and a support threshold ξ , *FS-Miner* discovers *SEASONs* from a constructed data structure called *FS-tree*, which is shown in Figure 7.7 (b). Similar to *FP-tree* [71], *FS-tree* contains a header table and a tree structure. Each entry in the header table has three fields: edge (a frequent sequence of length 2), count (count of the edge) and link (pointing to the first occurrence of the edge in the tree structure). For example, suppose the threshold ξ is 0.4, edges are frequent if their counts are no less than 2. After scanning the table in Figure 7.7 for the first time, three frequent edges are discovered and created in the header table. Then, a second scan of the database is performed to construct the tree structure. For example, we construct a root path $\langle C_1 C_3 C_2 \rangle$ in the tree structure for the first semantic cluster sequence. Similarly, another root path is created for the second semantic cluster sequence. For the third sequence, since it can share the edge $\langle C_2 C_3 \rangle$ with the second root path, we simply increment the count of the edge in the tree. Other sequences are inserted into the tree with the same logic.

For each edge in the header table, *FS-Miner* extracts derived paths. For example, for the edge $\langle C_3 C_2 \rangle$, two derived paths, highlighted with bold lines in Figure 7.7, are extracted. Then, the conditional sequence base of $\langle C_3 C_2 \rangle$ can be obtained by setting the frequency count of each edge in the paths to the count of the removed $\langle C_3 C_2 \rangle$, as shown in Figure 7.7 (c). We create the conditional *FS-tree* of $\langle C_3 C_2 \rangle$ by inserting each path of conditional sequence base into the tree in a backward manner. Finally, a depth first traversal is performed to discover all sequences satisfying the threshold ξ .

Figure 7.7: Mining SEASONs from evolutionary semantic cluster database.

We would like to highlight here that the novelty of the SEASON mining algorithm lies in the first phase, where we developed a “cluster-centered” method to construct clusters from a set of trees. To the best of our knowledge, this is the first method that clusters tree structures by measuring the “semantic cohesiveness” of clusters directly using frequent substructures.

7.3 Applications of SEASON

In this section, we first show the usefulness of SEASON patterns with some potential applications. Then, we discuss the details of applying SEASON in designing optimal replacement strategies in XML query caching.

7.3.1 Potential Applications

Since SEASON patterns discover knowledge about semantic evolution associations, they are useful in applications where tree structures are dynamic and semantically meaningful. Typically, Web usage data can be modelled as tree structures with semantics. For example, both XML query trees and Web session trees represent the semantics of user queries and evolve along with the variation of users’ information needs. Hence, we enumerate some applications of SEASON patterns mined from Web usage data as follows.

- **Replacing XML query cache.** Efficient processing of XML queries is an important issue in the XML research community [115] [150] [22]. Recently, caching XML queries

has been recognized as an orthogonal approach to improve the performance of XML query engines. For example, FastXMiner [157] proposed to mine frequent XML query patterns from the user queries and discard infrequent query patterns first once the cache is full. However, frequent query patterns may not always be reliable in predicting the subsequent queries, as the frequent query pattern based technique in [157] ignores the evolutionary feature of user queries. Instead, we can mine *SEASON* patterns from XML queries to predicate the subsequent information needs of users based on their current queries. The details are discussed in the next subsection.

- **Prefetching XML data.** Besides caching, another XML query performance improvement approach is prefetching. For example, Vincent Ng et al. [104] proposed to mine association rules from XML query sequences and prefetch answers to queries which are predicated to be issued subsequently. Considering that users may not issue exactly same queries in sequence, they proposed the idea of *abstract rules*, which are association rules between semantically similar queries. Queries are semantically similar if they are only different in constraint values (predicate). However, the form of abstract rules is not flexible enough such that many sequential associations between semantically similar queries may not be discovered. For example, two XPath queries, `//book[title = "XML"]/publisher/` and `//book[title = "WWW"]/publisher/`, will be regarded as similar queries as they are different in predicates only (e.g., XML and WWW). However, two queries, `//book[title = "XML"]/publisher/` and `//book[title = "WWW"]/publisher/address` will not be treated as similar queries although both inquire the information of book publisher. Since *SEASON* mining cluster similar queries before mining sequential patterns, *SEASON* based prefetching system should be more effective in improving the hit ratio and precision of XML cache.

7.3.2 Replacing XML Query Cache

In this section, we discuss the details of applying *SEASONs* in designing optimal XML cache replacement strategies. We first present deriving useful association rules, called *SEASON rules*, from *SEASONs* mined from sequential XML queries. Then, we discuss the designing of caching replacement strategies based on derived rules.

7.3.2.1 SEASON Rules

Given an XML query sequence database, where each sequence represents the set of XML queries sequentially issued by some user, a support threshold ξ , a minimum rooted subtree support δ and a minimum edge support χ , we can mine *SEASONS* of any length. That is, *SEASONS* with any number of semantic clusters. Particularly, in the application of XML query caching, we consider *SEASONS* of length 2 only (e.g., $\langle C_1 C_2 \rangle$), because this type of short patterns significantly reduce the complexity of the mining process and any *SEASON* with more than 2 clusters can be decomposed into a set of *SEASONS* of length 2.

Given a *SEASON* $\langle C_1 C_2 \rangle$ mined from an XML query sequence database, we may derive an association rule with the form of $C_1 \Rightarrow C_2$. The rule indicates that when a user issues a query that is semantically contained by or close to the semantics of cluster C_1 , she/he will probably issue a subsequent query that is semantically related to the semantics of cluster C_2 . Based on such kind of knowledge, we may optimally delay the eviction of answers to the predicated queries from the cache. On the contrary, we observe the following kind of knowledge is potentially useful as well. When a user issues a query semantically related to the cluster C_1 , she/he will probably not issue a query semantically related to cluster C_2 subsequently. Thus, we can optimally hasten the purge of answers to queries related to the semantics of C_2 . Thus, besides $\langle C_1 C_2 \rangle$, *SEASONS* including negative clusters, such as $\langle C_1 \neg C_2 \rangle$, are beneficial (the symbol $\neg$ is used to represent the nonoccurrence of a semantic cluster). We call the former *positive SEASON* and the latter *negative SEASON* respectively.

To determine the positive or negative association between variables, some correlation measures should be used. As we are equally interested whether a user inquires some information or not, the occurrence of a cluster C_i is a symmetric binary value. Hence, correlation measures which are suitable for analyzing symmetric binary variables can be used, such as ϕ -coefficient, odds ratio, and the Kappa statistic etc. [133]. In our analyses, we use the ϕ -coefficient. Then, if we use $\text{supp}(\langle C_i, * \rangle)$ to denote the support of any 2-sequence starting with C_i and $\text{supp}(\langle *, C_i \rangle)$ to denote the support of any 2-sequence ending with C_i , then the correlation of a *SEASON* of length 2 can be defined as follows.

Definition 7.11 (Correlation of SEASON) Given a *SEASON* $\langle C_i C_j \rangle$, the correlation of the pattern, denoted as $\text{corr}(\langle C_i C_j \rangle)$, is

$$\text{corr}(\langle C_i C_j \rangle) = \frac{\text{supp}(\langle C_i C_j \rangle) - \text{supp}(\langle C_i, * \rangle)\text{supp}(\langle *, C_j \rangle)}{\sqrt{\text{supp}(\langle C_i, * \rangle)(1 - \text{supp}(\langle C_i, * \rangle))\text{supp}(\langle *, C_j \rangle)(1 - \text{supp}(\langle *, C_j \rangle))}}$$

□

CHAPTER 7. SEASON MINING AND APPLICATION

The strength and the meaning of the correlation of *SEASON* are same as those of the ϕ -coefficient. Hence, if $\langle C_i C_j \rangle$ is greater than zero, it is a *positive SEASON* pattern. Otherwise, it is a *negative SEASON* pattern. The confidence of association rules derived from a *positive/negative SEASON* can be defined accordingly.

Definition 7.12 (Confidence of SEASON rule) *Given a SEASON $\langle C_i C_j^* \rangle$, the derived association rule has the form of $C_i \Rightarrow C_j^*$. The confidence of rule, denoted as $\text{conf}(C_i \Rightarrow C_j^*)$, is,*

$$\text{conf}(C_i \Rightarrow C_j^*) = \frac{\text{support}(\langle C_i C_j^* \rangle)}{\text{support}(\langle C_i, \rangle)}$$

where C_j^* represents either C_j or $\neg C_j$. □

Then, positive and negative *SEASON* rules, which can be mined from XML query sequences, can be defined based on these metrics.

Definition 7.13 (Positive and negative SEASON rules) *Let $\mathcal{D}$ be an XML query tree sequence database on $\Sigma = \{S_1, S_2, \dots, S_n\}$, where each tree in Σ represents an XML query. Let $f : \Sigma \rightarrow \Upsilon$ be the function that categorizes each tree in Σ to a semantic cluster in $\Upsilon = \{C_1, C_2, \dots, C_m\}$. Given the threshold of support ξ , the threshold of correlation μ and the threshold of confidence θ ,*

- $C_i \Rightarrow C_j$ is a *positive SEASON* rule if 1) $\text{corr}(\langle C_i C_j \rangle) \geq \mu$; 2) $\text{supp}(\langle C_i C_j \rangle) \geq \xi$; 3) $\text{conf}(\langle C_i \Rightarrow C_j \rangle) \geq \theta$.
 - $C_i \Rightarrow \neg C_j$ is a *negative SEASON* rule if 1) $\text{corr}(\langle C_i C_j \rangle) \leq -\mu$; 2) $\text{conf}(\langle C_i \Rightarrow \neg C_j \rangle) \geq \theta$.
-

7.3.2.2 Mining SEASON Rules

Given an XML query sequence database, the function $f : \Sigma \rightarrow \Upsilon$ can be implemented with the clustering algorithm discussed in the previous section. Then, the database can be transformed to be an evolutionary semantic cluster database. Rather than discovering *SEASONS* first and then derive possible positive and negative rules as commonly done by traditional association rule mining algorithm, we can discover positive and negative *SEASON* rules directly from the transformed evolutionary semantic cluster database.

CHAPTER 7. SEASON MINING AND APPLICATION

Algorithm 7.1 Positive and Negative SEASON Rule Generation**Input:** $\mathcal{D}, f : \Sigma \rightarrow \Upsilon, \xi, \mu, \theta$ **Output:** PR : A set of positive SEASON rules, NR : A set of negative SEASON rules**Description:**

```

1: scan  $\mathcal{D}$  and find the frequent 1-sequence ( $F_1$ ) /* $support(\langle C_i, \rangle)$  or  $support(\langle, C_i \rangle) \geq \xi$ */
2:  $P_2 = F_1 \bowtie F_1$  /*candidate frequent 2-cluster sequence*/
3: for each  $\langle C_i, C_j \rangle \in P_2$  do
4:   if  $corr(\langle C_i, C_j \rangle) \geq \mu$  then
5:     if ( $supp(\langle C_i, C_j \rangle) \geq \xi$ ) && ( $conf(C_i \Rightarrow C_j) \geq \theta$ ) then
6:        $PR = PR \cup \{C_i \Rightarrow C_j\}$ 
7:     end if
8:   else
9:     if ( $corr(\langle C_i, C_j \rangle) \leq -\mu$ ) && ( $conf(C_i \Rightarrow \neg C_j) \geq \theta$ ) then
10:       $NR = NR \cup \{C_i \Rightarrow \neg C_j\}$ 
11:    end if
12:  end if
13: end for

```

The algorithm is presented in Algorithm 7.2. Initially, we scan the transformed database to find the set of frequent 1-cluster sequences (Line 1). Then we join the frequent 1-cluster sequences to generate candidate *SEASONS* (Line 2). Note that, only a pair of 1-cluster sequences in the form of $\langle C_i, \rangle$ and $\langle, C_j \rangle$ can be joined. After that, we verify the correlation of candidate *SEASONS*. If the correlation is greater than the threshold μ (Line 4), we consider to derive a positive *SEASON* rule and check the *support* and *confidence* of the possible rule (Line 5). If both support and confidence of the derived rule are greater than the thresholds ξ and θ , it is added into the set of positive *SEASON* rules PR (Line 6). Otherwise, if the correlation of a candidate *SEASON* is not greater than the threshold $-\mu$, we check whether valid negative *SEASON* rules can be derived (Line 9 – 11).

7.3.2.3 Designing Replacement Strategies

After deriving positive and negative *SEASON* rules, we design replacement strategies with discovered rules⁵. As we cluster XML query (trees) into semantic clusters, our replacement scheme has two levels. The upper level applies replacement functions on clusters while the lower level decides the replacement value for queries in each cluster.

We incorporate the discovered association rules between clusters with LRU [44], which is a classic cache replacement algorithm that discards the least recently used items first,

⁵Although *SEASONS* are defined on semantic trees with specifically specified node labels, our strategy can be extended straightforwardly to handle XML queries with wildcards “*” and relative paths “//”, given the partial ordering defined in [157]. That is, a specifically specified node label is semantically contained by “*”, which is semantically contained by “//”.

CHAPTER 7. SEASON MINING AND APPLICATION

Algorithm 7.2 SEASON Rule based Cache Replacement Strategy**Input:** PR - the set of *positive SEASON* rules, NR - the set of *negative SEASON* rules, S_x - the new XML query**Description:**

```

1: let  $V_{top}$  be the most recent values for clusters
2: if  $Sim(S_x \rightarrow C_i) > 1$  and  $C_i \Rightarrow C_j \in PR$  then
3:   select the  $C_i \Rightarrow C_j$  with the highest  $Sim(S_x \rightarrow C_i)$ 
4:   if  $C_j$  exists in cache then
5:      $V(C_j) = V(C_j) + (V_{top} - V(C_j)) \times Conf(C_i \Rightarrow C_j)$ 
6:   end if
7: end if
8: if  $Sim(S_x \rightarrow C_i) > 1$  and  $C_i \Rightarrow \neg C_j \in NR$  then
9:   select the  $C_i \Rightarrow \neg C_j$  with the highest  $Sim(S_x \rightarrow C_i)$ 
10:  if  $C_j$  exists in cache then
11:     $V(C_j) = V(C_j) + (V(C_j) - V_{top}) \times Conf(C_i \Rightarrow \neg C_j)$ 
12:  end if
13: end if
14: for there is insufficient space for  $S_x$  do
15:   select the cluster  $C_i$  with lowest  $V(C_i)$ 
16:   remove queries in  $C_i$  according to some existing query level caching strategy
17: end for
18: admit  $S_x$ 
19: if there is a cluster  $C_i$  in the cache with the highest  $Sim(S_x \rightarrow C_i)$  then
20:    $V(C_i) = V_{top} + 1$ 
21: else
22:   create a cluster  $C_x$  for  $S_x$ ,  $V(C_x) = V_{top} + 1$ 
23: end if

```

in the upper level. Without loss of generality, we assume that “the most recent value for clusters”, V_{top} , is incremented by one, each time a new query S_x is issued. When a new query S_x is issued, we need to justify whether S_x is semantically contained by or close to an existing cluster C_i . Basically, we treat S_x as a singular cluster and compute the cluster similarity $Sim(S_x \rightarrow C_i)$ according to Definition 7.9. If there’s more than one cluster C_i such that $Sim(S_x \rightarrow C_i)$ is greater than one, we select the cluster with the highest cluster similarity. Then, we examine whether a positive *SEASON* rule $C_i \Rightarrow C_j$ was discovered and C_j is cached. If yes, we calculate a new replacement value for C_j as $V(C_j) = V(C_j) + (V_{top} - V(C_j)) \times conf(C_i \Rightarrow C_j)$. Obviously, $V(C_j)$ is increased and we delayed the eviction of queries in cluster C_j based on the rule. It is similar for negative *SEASON* rules. For example, with a negative rule $C_i \Rightarrow \neg C_j$, we update $V(C_j) = V(C_j) + (V(C_j) - V_{top}) \times conf(C_i \Rightarrow \neg C_j)$. Since $V(C_j)$ is decreased, we actually hasten the the purge of queries in cluster C_j .

For the replacement function for queries in the lower level, some other parameters, such as the processing cost of the query and the size of the query region, should be considered. These issues are out of the scope of our discussion on this application. Interested readers can refers to [157] for the details.

N	Number of semantic trees	1K
L	Number of potential frequent rooted subtrees	8
D	Maximum depth of rooted subtrees	10
F	Maximum fanout of rooted subtrees	10
M	Average number of nodes of rooted subtrees	20
P	Maximum overlap between frequent rooted subtrees	0.2
O	The ratio of outliers	0.05

(a)

Data	Feature parameters
d_1	$P = 0.05$
d_2	$P = 0.1$
d_3	$P = 0.2$
d_4	$O = 0.01$
d_5	$O = 0.05$
d_6	$O = 0.08$
d_7	$N = 10K$

(b)

Table 7.2: Parameters and data sets.

7.4 Experimental Results

In this section, we evaluate the performance of the *SEASON* mining algorithm and the *SEASON* based cache replacement strategies. We implemented our approach in the Java programming language. Experiments are carried out on a Pentium IV 2.8GHz PC with 512 MB memory. The operating system is Windows 2000 professional.

7.4.1 Performance of SEASON Mining

Firstly, we investigate the performance of the *SEASON* mining algorithm. As discussed in Section 7.2, the mining algorithm has two phases. The first one clusters the semantic trees while the second one discovers *SEASONs* from the transformed evolutionary semantic cluster database. Since we borrow the *FS-Miner* [57] algorithm in the second phase, we focus on evaluating the performance of the first phase.

In order to evaluate our semantic tree clustering method in the first phase, we implemented a synthetic tree generator. The set of parameters used by the generator are presented in Table 7.2. Basically, synthetic trees are generated with the following steps ⁶:

- We generate a set of L potential frequent rooted subtrees by controlling the overlap P between them. The structure of each potential frequent rooted subtree are decided by D , F and M . For example, at a given node, the number of children is sampled uniformly at random from the range 0 to F . For each child, the process is performed recursively if the depth of the tree is less than or equal to D or the total number of nodes reaches a value sampled from a Poisson distribution with mean of M . After

⁶Note that, the generated synthetic trees do not really convey meaningful semantics. But it does not affect the evaluation of our algorithm by assuming the generated labelled trees are semantically meaningful.

generating the first potential frequent rooted subtree, we generate subsequent subtrees based on the previous one with respect to the parameter P . For example, if the number of nodes in the previous tree is M_i and the number of nodes in the current tree is M_{i+1} , then the number of changed nodes between trees $(1 - p) * (M_i + M_{i+1})$. We randomly decide the positions of inserted and deleted nodes.

- We then generate the set of $N * (1 - O)$ trees based on potential frequent rooted subtrees produced in the first step. We generate $N * O$ trees randomly with respect to the parameters D , F and M . Each potential frequent rooted subtree is assigned a weight, which corresponds to the probability that it will be selected to generate a synthetic tree. The weight is picked from an exponential distribution with unit mean. Weights of each frequent rooted subtree is normalized so that the sum of weights is 1. The next frequent rooted subtree to be used to generate a synthetic tree is chosen by tossing an L -sided weighted coin, where the weight of a side is the probability of picking the corresponding frequent rooted subtree. We also associate each potential frequent rooted subtree a corruption level, which is obtained from a normal distribution with mean 0.5 and variance 0.1. Then, when constructing a synthetic tree based on the potential frequent rooted subtree, we keep deleting nodes randomly as long as a uniformly distributed random number between 0 and 1 is less than the corruption level of the potential frequent rooted subtree.

The third column of Table 7.2 shows the default values. Basically, there are 7 datasets used in the following experiments. The key parameter values used in generating the datasets are shown in Table 7.2 (b).

Accuracy Study. We first conduct experiments to study the accuracy of the clustering method. We evaluate the accuracy of the clustering with two indexes, *average intra-cluster dissimilarity* and *average inter-cluster similarity*, which are defined as follows.

$$Avg_Intra = \frac{1}{k} \sum_{i=1}^k Intra(C_i) \quad (\text{Eq. 7.1})$$

$$Avg_Inter = \frac{2 \times Inter(C_i, C_j)}{k \times (k - 1)} \quad (\text{Eq. 7.2})$$

where $Intra(C_i)$ is the intra-cluster dissimilarity as defined in Definition 7.8 and $Inter(C_i, C_j)$ is the inter-cluster similarity as defined in Definition 7.10, k is total number of clusters. For

	d_1	d_2	d_3	d_4	d_5	d_6
<i>Avg_Intra</i>	0.312	0.339	0.386	0.380	0.386	0.388
<i>Avg_Inter</i>	0.142	0.181	0.205	0.199	0.205	0.204

Figure 7.8: Accuracy of clustering method.

a good clustering, both values should be low. We conduct experiments on the set of data sets d_1 through d_6 . The data sets d_1 , d_2 and d_3 are generated with different overlap values. While, the data sets d_4 , d_5 and d_6 are generated with different outlier ratios. Both the *minimum rooted subtree support* and the *minimum edge support* are set as 25%. Figure 7.8 shows the accuracy of the produced clusters. We have the following observations:

- Generally, our clustering method can achieve both small average intra-cluster dissimilarity and small average inter-cluster similarity.
- When the overlap value between potential frequent rooted subtrees decreases (e.g., from d_3 to d_1), our method can achieve better accuracy.
- The variation of the outlier ratio, which is used to generate data sets (d_4, d_5, d_6) , does not affect the performance of our method obviously.

Sensitivity to Parameters. We conduct the second experiment to study how the variation of parameters affect the accuracy of the clustering. There are two parameters used in the clustering method: *minimum rooted subtree support* (δ) and *minimum edge support* (χ). We conduct experiments on data sets d_1 , d_2 and d_3 to study the effect of the two parameters respectively. Figures 7.9 (a) and (b) shows the *Avg_Intra* and the *Avg_Inter* by varying δ respectively. We noticed that when δ is larger, the *Avg_Intra* value increases while the *Avg_Inter* decreases. This is because when δ is larger, fewer frequent rooted subtrees are generated and fewer clusters are found accordingly. Figures 7.9 (c) and (d) show the *Avg_Intra* and the *Avg_Inter* by varying χ respectively. We observed that when χ increases, although the *Avg_Intra* and *Avg_Inter* values vary in the same trend as when δ increases, they vary within smaller range. That is, the parameter δ has more effect on the accuracy of the clustering.

Efficiency Study. We evaluated the efficiency of our clustering method by conducting experiments on data set d_7 and varying the number of trees from 1K to 10K. We first

Figure 7.9: Sensitivity to parameters.

examine the time cost of each clustering step. Both the minimum rooted subtree support and the minimum edge support are set as 25%. The experimental results are shown in Figure 7.10 (a). The main cost of our clustering method is the disjointing step as it needs to construct the merged trees for clusters. We further evaluate the efficiency of the clustering method with respect to the variation of the two parameters: minimum rooted subtree support (δ) and minimum edge support (χ). The experimental results are shown in Figures 7.10 (b) and (c) respectively. We notice that when the threshold of rooted subtree support is increased, the clustering method is more efficient. The reason is that when the minimum rooted subtree support is high, fewer clusters will be generated. Thus, both the disjointing step and the merging step need to handle fewer initial clusters. However, the minimum edge support does not affect the efficiency of our clustering method obviously. This is because the computation efficiency of each clustering step does not depend on this parameter.

Figure 7.10: Performance of clustering.

Scalability Study. We evaluate the scalability of the clustering method by duplicating the trees in data set d_7 until we get 100K semantic trees. Since the parameter of minimum edge support χ does not affect the execution time of the clustering method, we fix it at 25%. Three different minimum rooted subtree support values are used: $\delta = 10\%$, $\delta = 30\%$ and $\delta = 50\%$. The experimental results are shown in Figure 7.10 (d). It verifies that our clustering method scales well with respect to the number of trees.

7.4.2 Performance of Replacement Strategy

We then show the effectiveness of our replacement strategy with *SEASON* rules mined from XML query sequence database.

We used a simple XQuery processor [76] that proceeds queries directly from the source XML file. Hence, no underlying storage strategy or indexing techniques will be involved in affecting the query response time. However, such a processor is not very scalable with

respect to the size of the source XML document. Consequently, in our experiment, we generated a fragment of DBLP data as the source XML document. The file size is 10.5M and there are totally 248,215 nodes.

Synthetic XML queries are generated with the similar method described in the previous subsection. Firstly, we generate a set of potential frequent XML queries based on the DTD of DBLP. Secondly, we generate a set of potential frequent sequences such that each sequence contain two potential frequent XML queries. Thirdly, a sequence of 1000 XML queries are generated based on the frequent queries and frequent sequences generated in the first two steps. We then use the first M queries in the sequence as the training data to discover SEASON rules and the remaining N queries as the test to evaluate the performance of caching.

Three sets of experiments were carried out to investigate the effect of varying the number of queries, varying the size of cache, and varying the size of training data set respectively. We compared our SEASON association rule based LRU replacement strategy (denoted as *LRU_AR*) with another two strategies, *LRU* and LRU integrated with frequent query patterns mined by [157] (denoted *LRU_FQPT*). We use the *Average Response Time*, which is the ratio of total execution time for answering a set of queries to the total number of queries in this set, as the metric.

Variation of Query Numbers. Because of the limited power of the query processor, we vary the number of test queries N from 100 to 500 (Note that, we fix the training queries at 900 and duplicate the test queries). The cache size is fixed at 2.5MB, which is nearly one quarter of the source XML document. The set of thresholds ξ , δ and χ are 0.1, 0.25 and 0.25 respectively. The experimental results are shown in Figure 7.11 (a). We observed that when the number of queries is less than 200, *LRU_AR* does not have much advantage over *LRU_FQPT*. However, when the number of queries is larger than 200, the strategy *LRU_AR* works best. Particularly, when the number of queries is 500, *LRU_AR* works nearly 1s faster than *LRU_FQPT*. This improvement is not slight considering that the query processor averagely takes 0.005s to process an XML query. Furthermore, according to the trend shown in Figure 7.11, when more test queries are used, more efficiency gains will be achieved by *LRU_AR*.

Variation of Cache Size. In this experiment, we vary the size of cache from 1.5M to 5.5M, the number of test queries N is fixed at 300. The set of thresholds are same above the

Figure 7.11: Performance of XML query caching strategy.

above experiment. As shown in Figure 7.11 (b), the more limited the cache size, the greater gap in average response time between *LRU_AR* and the other two competitors. However, when the cache size is large enough, the simple *LRU* strategy works best as it updates the replacement values of query regions most efficiently.

Variation of Training Data Size. In this experiment, we evaluate the performance of our caching strategy with respect to the variation of the training data size. We vary the number of training queries M from 500 to 900 and use the subsequent 100 queries as the test data (In order to show the difference clearly, the test data is duplicated until we get 300 testing queries). We fix the cache size at 2.5M. The experimental results are shown in Figure 7.12. It can be observed that our XML caching strategy works better with a larger training data size. The reason is that when the number of training queries increases, the discovered *SEASON* rules are more accurate and the corresponding caching strategy is more effective.

7.5 Related Work on Caching

Caching has played a key role in the client-server databases, distributed databases and Web-based information systems because network traffic and slow remote servers can lead to long delays in the delivery of answers [157]. Semantic caching was proposed in [50] as a more flexible model than the previous tuple [52] and page [26] caching systems. Semantic caching manages the cache at the data granularity of group of tuples. Hence, it incurs less space overhead than tuple caching for cache management (such as buffer control blocks, hash

Figure 7.12: Average response time with respect to the training data size.

table entries etc). Semantic caching allows the grouping of tuples to be adjusted to the needs of the current queries, thus, it is more flexible than the page caching which statically defines the groups of tuples (pages).

Due to its flexibility, semantic caching was popular in Web query caching [42] and recently XML query caching [74] [37]. For example, Chidlovskii and Borghoff [42] discussed using semantic caching to group together semantically related Web documents covered by a boolean Web user query (i.e., the documents dynamically generated by cgi-scripts after the user filling out a search form). Hristidis and Petropoulos [74] proposed a compact structure, *modified incomplete tree (MIT)*, to represent the semantic regions of XML queries. ACE-XQ [37] is a holistic XQuery-based semantic caching system. They discussed how to judge whether a new query is contained by any cached query and how to rewrite the new query with respect to the cached queries. They also proposed a fine-grained replacement strategy rather than replacing a complete query region at a time. However, this work did not consider using the knowledge mined from historical user queries to design the replacement function.

Recently, intelligence has been incorporated into Web/XML query caching by constructing predictive models of user requests with the knowledge mined from historical queries [87] [17] [157]. Lan et al. [87] mined association rules from Web user access patterns. Then they prefetched Web documents based on discovered associations and current requested documents. They focused on the placement strategy (fetching and prefetching) while we focused on the replacement strategy. Bonchi et al. [17] mined association rules from Web log data to extend the traditional LRU replacement strategy. For example, given

a discovered association rule $URL\ A \Rightarrow URL\ B$, when a user queries the resource at $URL\ A$, if the resource at $URL\ B$ is in the cache, its eviction should be delayed. However, their work cannot be applied in XML query caching directly because answers to XML query do not have explicit identifiers such as URL. Hence, our work is different from this one in that we mine association rules between query groups in which queries are semantically close. Furthermore, we also use negative association rules to demote the replacement values of corresponding query regions.

7.6 Summary

In this chapter, we have introduced another associative evolutionary structure, *SEASON*, which views changes to tree structures from the semantic variation perspective. Given an evolutionary semantic tree database, where each tree represents some semantic meaning, a *SEASON* pattern is a frequent pattern about semantic evolution.

Two phases are involved in discovering *SEASONs* from an evolutionary semantic tree database. First, each semantic tree should be categorized to a collectively represented semantic cluster. To this end, we performed a clustering algorithm on the set of tree structures in the database so that trees in the same cluster represent similar semantics. After this phase, the semantic tree sequence database can be transformed to be an evolutionary semantic cluster database. Then, *SEASON* patterns are discovered from the transformed database easily with some existing frequent sequential pattern mining algorithm with respect to the threshold of support.

SEASONs mined from XML query sequence database infer that when a user issues a query of some particular information, she/he will query about some other information of the XML document subsequently. Then, *SEASONs* can be used in designing optimal XML query cache replacement strategies. We first derive both positive and negative association rules from discovered *SEASONs*. Then, we promote the replacement value for queries in the consequences of positive rules and demote the replacement value for those of negative rules. As verified by the experimental results, *SEASON*-based replacement strategy improved the cache performance.

Chapter 8

Conclusions

In this doctoral dissertation, we have studied two basic problems in the field of data mining: the definition of new knowledge patterns and the discovery of knowledge patterns. We have also investigated other problems such as how to concisely represent the knowledge patterns (with maximal patterns) and how to apply the discovered patterns in other research problems. In this chapter, we summarize in Section 8.1 the results that we have obtained in this dissertation. In Section 8.2, we discuss some future research issues on mining association rules from structural evolution of tree structured Web data.

8.1 Summary of Results

We have proposed (in Chapter 3) the idea of mining structural evolution patterns from the evolution of tree structured data and formalized it as a data mining problem. The mining context is a temporal tree (subtree) database which contains sequential versions of a tree structure. The problem of *structural evolution pattern mining* aims to discover interesting associations in the evolution of the tree structure. We have proposed two special patterns: intra-tree structural evolution pattern and inter-tree structural evolution pattern. The former focuses on discovering associations between subtrees, i.e., a set of subtrees that frequently change together in the sequence of historical versions; while the latter considers the associations between sequential trees, i.e., a tree (representing some particular semantics) frequently evolves into another tree (representing some other semantics). These proposals narrow our research space and facilitate us in defining interesting and meaningful knowledge patterns.

We have proposed the *FRACTURE* pattern (in Chapter 4) as an intra-tree structural evolution pattern. Considering the two attributes of changes to trees (subtrees): the frequency of change and the significance of change, we defined a *FRACTURE* as a set of subtrees which not only frequently change together but also frequently undergo significant changes together. Such a definition excludes some meaningless patterns. For example, a subtree always changes together with its ancestor subtrees. However, they may not undergo significant changes together frequently. We have defined two interestingness measures, *Frequency of Change (FoC)* and *Significance of Change (SoC)*, to measure the how frequently a set of subtrees change together and how frequently a set of subtrees undergo significant change together respectively. We have investigated the properties of *FRACTUREs* such as the downward closure on the *FoC* and the downward closure on the product of *FoC* and *SoC*. Based on these properties, we have proposed efficient *level-wise* based algorithm and *pattern-growth* based algorithm. To concisely represent the set of resulting patterns, we have observed a redundancy in pairs of *FRACTURE* patterns having the *subsumption* relationship. That is, if a set of subtrees is subsumed by a known *FRACTURE*, then the set of subtrees is a *FRACTURE* pattern as well. Based on the observation, we have proposed the *maximal FRACTURE* pattern and proved that the complete set of *FRACTURE* patterns can be generated from the set of maximal ones. Without post-processing the mined *FRACTURE* patterns to discover the maximal ones, we have proposed a set of optimizing strategies to mine the *maximal FRACTUREs* based on the pattern-growth algorithm. We have shown with experimental results that the proposed algorithm is superior to the naïve algorithm. In addition, the cardinality of the set of *maximal FRACTUREs* is substantially less than that of the complete set of *FRACTUREs*.

We have introduced (in Chapter 5) a new Web user clustering algorithm by making use of the *FRACTURE* patterns' capability to capture some evolutionary characteristics of a evolutionary tree. The clustering algorithm is called *COWES* – Clustering Of Web users by historical wEb Sessions. The basic idea of *COWES* is to use *FRACTURE* patterns mined from historical Web session trees of users as the clustering features. Considering that Web sessions of users are evolutionary, for example, users may vary their information needs from time to time, it makes sense to generate clusters of users who exhibit similar accessing behaviors in their historical Web sessions (e.g., they vary their information needs with the similar patterns). We then mine *FRACTUREs* from historical Web session trees for each

CHAPTER 8. CONCLUSIONS

user and cluster Web users based on their shared *FRACTURE* patterns. We have defined a set of similarity measures which measure the proximity of users based on their shared *FRACTURES* in terms of *element similarity* as well as *strength similarity*. After generating the similarity matrix of Web users, different clustering algorithms, such as partitional clustering, agglomerative hierarchical clustering and graph-based clustering, are employed to generate the final clusters. Our experimental results showed that *COWES* discovered novel clusters which cannot be discovered by existing Web user clustering methods.

We have defined (in Chapter 6) the *negative FRACTURE* as an extension to *FRACTURE* patterns. *FRACTURE* patterns capture the positive correlation in changed subtrees, while *negative FRACTURE* refers to a set of subtrees that are negatively correlated in undergoing significant changes. Hence, we have defined the metric *Correlation of Change (CoC)* to identify the negative nature between subtrees in undergoing significant changes. Since *negative FRACTURES* are particularly defined on a sequence of subtrees with parent-child relationships, we have developed a data structure that not only records the changed subtrees but also preserves the parent-child relationship between subtrees for pattern mining. The developed pattern mining algorithm employs the *divide-and-conquer* strategy to discover the complete set of *negative FRACTURES* efficiently. We have applied the *negative FRACTURES*, discovered from historical Web directory hierarchies, in designing optimal autonomous Web mirror maintenance strategies. The experimental results demonstrated that *negative FRACTURE* based mirror maintenance strategy can improve the maintenance efficiency significantly without sacrificing too much in keeping the “freshness” of the mirror.

We have proposed the *SEASON* pattern (in Chapter 7) as an inter-tree structural evolution pattern. Complementary to *FRACTURES* and *negative FRACTURES* which consider the changes to tree structures from the structural variation perspective, *SEASON* patterns view the changes from semantic variation perspective. Particularly, *SEASON* patterns describe the frequent semantic evolution along with the tree structure evolution. In order to discover the evolution of semantics, we proposed to categorize each tree version into a cluster representing some semantics. Due to the intuition that there are some frequent structures for each tree cluster and different clusters share few frequent structures, we have developed a “cluster-centered” clustering method. The method first mines frequent rooted subtrees from the set of trees and assigns each tree to the cluster representing the maximal

frequent rooted subtree it supports. Two subsequent steps which make the clusters disjoint and prune the semantically close clusters respectively are performed. After transforming the evolutionary tree database into an evolutionary semantic cluster database, the *SEASON* patterns of frequent cluster sequences can be discovered directly. We have applied the *SEASON* patterns mined from XML query sequence database in designing replacement strategies for XML query caching. Experiment results have verified the effectiveness of the *SEASON* pattern based caching replacement strategy.

8.2 Future Research Issues

We list below some potential improvements to our work and some new challenges to be addressed as future research topics.

- (i) In *FRACTURE* pattern mining, we ignore the lifespan of subtrees and consider the subtrees that existing in the whole sequence of historical versions. However, some subtrees may appear from the middle of the sequence while some subtrees may be deleted before the last version. Considering subtrees with different lifespans, the current set of interestingness measures are biased. For example, the *FoC* normalizes the times the set of subtrees changed together with the length of the sequence. Thus, for a subtree that occurs from the middle of the sequence, the current *FoC* underestimates the frequency of change for subtree sets containing this subtree. Consequently, the current definition of *FRACTURE* may uncover some interesting subtree sets. In order to address the lifespan of subtrees, the current interestingness measures should be extended to be temporal. Novel discovering algorithms should be developed accordingly.
- (ii) From discovered *FRACTURE* patterns, we can derive association rules such that each rule indicates that when a set of subtrees changes significantly, another set of subtrees frequently undergoes significant changes as well. However, in some applications, it is more useful if we can further infer the degree of change for the set of subtrees on the right hand of the rule. For example, in numbering XML indexing schemes which assign a pair of numbers $\langle order, size \rangle$ to each node to represent the position of the node and the ranges of its descendants respectively, it has been suggested to reserve some space in the *size* of the node to allow future insertions of its descendant. Then, if the *FRACTURE* patterns can indicate the degree of change for subtrees,

CHAPTER 8. CONCLUSIONS

the knowledge can be used to guide the decision on the reserved size for each node. Therefore, it is meaningful to extend the current form of *FRACTURE* to be $\{s_1 : d_1, s_2 : d_2, \dots, s_m : d_m\}$, where the d_i after s_i is some meaningful range the *DoC* of subtree s_i belongs to.

- (iii) In *COWES*, we used all the set of *FRACTURE* patterns mined from historical Web session trees for each user as the clustering feature. Then, the feature dimension is large and the computation of similarity matrix is expensive, as we need to find the optimal assignment of each two sets of *FRACTUREs* such that the total weight between matching *FRACTUREs* can be maximized. However, including all the set of *FRACTUREs* of each user as the clustering feature does not necessary improve the clustering accuracy. For example, *FRACTURE* patterns shared by all users do not help in discriminating users. Consequently, issues related to selecting *FRACTURE* patterns with high discriminative power such that the clustering accuracy and efficiency can be improved should be considered.
- (iv) In this dissertation, we discussed the particular *negative FRACTURE* which is defined on a sequence of subtrees with ancestor-descendant relationships. Actually, *negative FRACTUREs* defined on any subtrees, which are either structurally related or not, may be useful as well in some applications. Furthermore, we considered the negative evolution association between a set of subtrees and one particular subtree. That is, the association rule derived from a *negative FRACTURE* has only one subtree on the right hand side. In fact, rules which have more than one subtrees on the right hand side should be beneficial. For example, when applying *negative FRACTUREs* in maintaining autonomous Web mirrors, we skip the whole subdirectory represented by the subtree on the right hand side of the *negative FRACTURE* rule. However, even if a set of subtrees X are negatively correlated with its descendant subtree s_b , they might be positively correlated with some descendant subtree of s_b , e.g., s_c . Thus, in maintaining the directories represented by X , if we skip all the files in the directory represented by s_b , we missed the changed information in the directory represented by s_c . Hence, it is beneficial if a *negative FRACTURE* is discovered between X and all the set of descendant subtrees of s_b .

CHAPTER 8. CONCLUSIONS

- (v) In order to discover *SEASON* patterns, we proposed an algorithm that clusters tree structures based on their semantics. However, the clustering algorithm is static. That is, when a set of new trees come in (which is a frequent situation in XML query database), we need to perform the clustering algorithm from scratch to generate the semantic clusters. Then, the *SEASON* patterns should be rediscovered from the set of new clusters. Obviously, it is computationally expensive. This necessitates the development of evolutionary clustering algorithms which are updatable with the new incoming trees. This is a challenging problem as the incoming trees may affect the existing clusters. Furthermore, considering that some previous tree versions need to be reassigned to their closest clusters, the maintenance of the *SEASON* patterns is even difficult since existing incremental sequential pattern mining algorithms may not be applied directly.

Appendix A

Publications

- **Ling Chen**, S S Bhowmick and W. Nejdl. Mirror Site Maintenance Based on Evolution Associations of Web Directories. *In proceedings of the 13th International Conference on World Wide Web (WWW)*, 2007.
- **Ling Chen**, S S Bhowmick and L-T Chia. FRACTURE-Mining: Mining Frequently and Concurrently Mutating Structures from Historical XML Documents. *Data and Knowledge Engineering Journal (DKE)*, Elsevier Science, 2006.
- Q Zhao, **Ling Chen**, S S Bhowmick, S K Madria. XML Structural Delta Mining: Issues and Challenges. *Data and Knowledge Engineering Journal (DKE)*, Elsevier Science, 2006.
- **Ling Chen**, S S Bhowmick and J Li. COWES: Clustering web users based on historical web sessions. *To appear in proceedings of the 11th International Conference on Database Systems for Advanced Applications (DASFAA)*, 2006.
- **Ling Chen**, S S Bhowmick and J Li. Mining temporal indirect associations. *To appear in proceedings of the 10th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, 2006.
- **Ling Chen**, S S Bhowmick and L-T Chia. Mining Positive and Negative Association Rules from XML Query Patterns for Caching. *In proceedings of the 10th International Conference on Database Systems for Advanced Applications (DASFAA)*, 2005.
- **Ling Chen**, S S Bhowmick and L-T Chia. Mining Association Rules from Structural Deltas of Historical XML Documents. *In proceedings of the 8th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, 2004.

APPENDIX A

- **Ling Chen**, S S Bhowmick and L-T Chia. Mining Maximal Frequently Changing Subtree Patterns from XML Documents. *In proceedings of the 6th International Conference on Data Warehousing and Knowledge Discovery (DaWak)*, 2004.
- **Ling Chen**, S S Bhowmick and L-T Chia. VRules: An Effective Association-based Classifier for Videos. *In proceedings of the 2nd ACM International Workshop on Multimedia Database (MMDB)*, 2004.

Appendix B

List of Web sites

adam.ac.uk; cbs.sportsline.com; ccrma-www.stanford.edu; elib.cs.berkeley.edu; eserver.org; etext.lib.virginia.edu; java.sun.com; memory.loc.gov; nccam.nih.gov; softwaredev.earth-web.com; start.earthlink.net; thomas.loc.gov; wdcn.nig.ac.jp; www.acm.org; www.acsh.org; www.ai.mit.edu; www.ala.org; www.ama-assn.org; www.amstat.org; www.archives.gov; www.artifice.com; www.bayarea.com; www.berkeley.edu; www.biosis.org; www.biostat.wisc.edu; www.bls.gov; www.boston.com; www.cancerbacup.org.uk; www.cancercenter.com; www.census.gov; www.charm.net; www.chron.com; www.cia.gov; www.cmp.com; www.colorado.edu; www.columbia.edu; www.contemporaryjazz.com; www.cs.berkeley.edu; www.discovery.com; www.dotmusic.com; www.dtic.mil; www.dtv.dk; www.duke.edu; www.eff.org; www.eonline.com; www.faqs.org; www.fcc.gov; www.fda.gov; www.findlaw.com; www.freebsd.org; www.georgetown.edu; www.getty.edu; www.golfweb.com; www.gospelcom.net; www.house.gov; www.hp.com; www.infomine.com; www.intel.com; www.intelihealth.com; www.internetnews.com; www.intranetjournal.com; www.ipl.org; www.irs.gov; www.jpl.nasa.gov; www.ksc.nasa.gov; www.llnl.gov; www.macromedia.com; www.metrokc.gov; www.nal.usda.gov; www.nba.com; www.ncaa.org; www.ncsa.uiuc.edu; www.nfl.com; www.novell.com; www.npr.org; www.nps.gov; www.nsf.gov; www.nws.noaa.gov; www.oracle.com; www.oregonlive.com; www.oreilly.com; www.osha.gov; www.osti.gov; www.pbs.org; www.perseus.tufts.edu; www.planetquake.com; www.sec.gov; www.si.edu; www.sigmaxi.org; www.symantec.com; www.time.com; www.tomshardware.com; www.usda.gov; www.usgs.gov; www.usps.com; www.w3.org; www.washingtonpost.com; www.webopedia.com; www.well.com; www.worldbank.org; www.wunder-ground.com; www-csli.stanford.edu

References

- [1] Cvsup. In <http://www.cvsup.org/>.
- [2] Ftpmirror. In <http://packages.debian.org/unstable/net/ftpmirror>.
- [3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1):68–88, 1997.
- [4] R. Agarwal, C. Aggarwal, and V. Prasad. A tree projection algorithm for generation of frequent item sets. *J. Parallel Distrib. Comput.*, 61(3):350–371, 2001.
- [5] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD Conference*, pages 207–216, 1993.
- [6] R. Agrawal and J. C. Shafer. Parallel mining of association rules. *IEEE Trans. Knowl. Data Eng.*, 8(6):962–969, 1996.
- [7] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.
- [8] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE*, pages 3–14, 1995.
- [9] Hungarian algorithm. http://en.wikipedia.org/wiki/Hungarian_algorithm.
- [10] Munkres' (Hungarian) Algorithm. <http://www.spatial.maine.edu/~kostas/dev/soft/munkres.htm>.
- [11] M. Antonie and O. R. Zaïane. Mining positive and negative association rules: An approach for confined rules. In *PKDD*, pages 27–38, 2004.
- [12] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *SDM*, 2002.
- [13] T. Asai, H. Arimura, T. Uno, and S.-I. Nakano. Discovering frequent substructures in large unordered trees. In *Discovery Science*, pages 47–61, 2003.
- [14] R. J. Bayardo. Efficiently mining long patterns from databases. In *SIGMOD Conference*, pages 85–93, 1998.

REFERENCES

- [15] P. A. Bernstein, M. L. Brodie, S. Ceri, D. J. DeWitt, M. J. Franklin, H. Garcia-Molina, J. Gray, G. Held, J. M. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. F. Naughton, H. Pirahesh, M. Stonebraker, and J. D. Ullman. The asilomar report on database research. *SIGMOD Record*, 27(4):74–80, 1998.
- [16] K. Bharat and A. Z. Broder. Mirror, mirror on the web: A study of host pairs with replicated content. *Computer Networks*, 31(11-16):1579–1590, 1999.
- [17] F. Bonchi, F. Giannotti, C. Gozzi, G. Manco, M. Nanni, D. Pedreschi, C. Renso, and S. Ruggieri. Web log data warehousing and mining for intelligent web caching. *Data Knowl. Eng.*, 39(2):165–189, 2001.
- [18] J. Borges and M. Levene. Mining association rules in hypertext databases. In *KDD*, pages 149–153, 1998.
- [19] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *SIGMOD Conference*, pages 265–276, 1997.
- [20] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *SIGMOD Conference*, pages 255–264, 1997.
- [21] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.
- [22] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD Conference*, pages 310–321, 2002.
- [23] C. Buchwalter, M. Ryan, and D. Martin. The state of online advertising: data covering 4thQ 2000. In *TR Adrelevance*, 2001.
- [24] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *ICDE*, pages 443–452, 2001.
- [25] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *USENIX Symposium on Internet Technologies and Systems*, 1997.
- [26] M. J. Carey, M. J. Franklin, and M. Zaharioudakis. Fine-grained sharing in a page server *oodbms*. In *ACM SIGMOD*, 1994.
- [27] Deepayan Chakrabarti, Ravi Kumar, and Andrew Tomkins. Evolutionary clustering. In *KDD*, pages 554–560, 2006.
- [28] S. Chakrabarti, B. Dom, R. Kumar, P. Raghavan, S. Rajagopalan, A. Tomkins, D. Gibson, and J. M. Kleinberg. Mining the web’s link structure. *IEEE Computer*, 32(8):60–67, 1999.

REFERENCES

-
- [29] G. Change, M. J. Healey, J. McHugh, and J. Wang. *Mining the World Wide Web: An Information Search Approach*. Kluwer Academic Publishers, 2001.
 - [30] S. S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. In *ICDE*, pages 4–13, 1998.
 - [31] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *SIGMOD Conference*, pages 26–37, 1997.
 - [32] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *SIGMOD Conference*, pages 493–504, 1996.
 - [33] L. Chen, S. S. Bhowmick, and L-T Chia. Association rules from structural deltas of historical xml documents. In *PAKDD*, 2004.
 - [34] L. Chen, S. S. Bhowmick, and L-T Chia. Mining maximal frequently changing subtree patterns from xml documents. In *Dawak*, 2004.
 - [35] L. Chen, S. S. Bhowmick, and L-T Chia. Mining positive and negative association rules from xml query patterns for caching. In *DASFAA*, 2005.
 - [36] L. Chen, S. S. Bhowmick, and J. Li. Cowes: Clustering web users based on historical web sessions. In *DASFAA*, 2006.
 - [37] L. Chen, E. A. Rundensteiner, and S. Wang. Xcache: a semantic caching system for xml queries. In *SIGMOD Conference*, page 618, 2002.
 - [38] Ling Chen, Sourav S. Bhowmick, and Liang-Tien Chia. Fracture mining: Mining frequently and concurrently mutating structures from historical xml documents. *Data Knowl. Eng.*, 59(2):320–347, 2006.
 - [39] M.-S. Chen, J. S. Park, and P. S. Yu. Efficient data mining for path traversal patterns. *IEEE Trans. Knowl. Data Eng.*, 10(2):209–221, 1998.
 - [40] Y.-F. Chen, F. Douglass, H. Huang, and K.-P. Vo. Topblend: An efficient implementation of htldiff in java. In *WebNet*, pages 88–94, 2000.
 - [41] Z. Chen, F. Lin, H. Liu, Y. Liu, W.-Y. Ma, and L. Wenyan. User intention modeling in web applications using data mining. *WWW*, 5(3):181–191, 2002.
 - [42] B. Chidlovskii and U. M. Borghoff. Semantic caching of web queries. *VLDB J.*, 9(1):2–17, 2000.
 - [43] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in xml documents. In *ICDE*, pages 41–52, 2002.
 - [44] E. G. Coffman and P. J. Denning. Operating systems theory. In *Prentice-Hall International Editions*, 1973.

REFERENCES

- [45] J. Cohen. Statistical power analysis for the behavioral sciences (2nd ed.). In *Lawrence Erlbaum, New Jersey*, 1988.
- [46] World Wide Web Consortium. Xpath, <http://www.w3.org/tr/xpath/>.
- [47] R. Cooley, B. Mobasher, and J. Srivastava. Data preparation for mining world wide web browsing patterns. *Knowl. Inf. Syst.*, 1(1):5–32, 1999.
- [48] Curbera and D. A. Epstein. Fast difference and update of xml documents. In *XTech*, 1999.
- [49] T. Dalamagas, T. Cheng, K. Winkel, and T. K. Sellis. Clustering xml documents by structure. In *Proc. of SETN*, 2004.
- [50] S. Dar, M. J. Franklin, B. THór Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB*, pages 330–341, 1996.
- [51] Assignment Problem Definition. <http://www.definethat.com/define/3622.htm>.
- [52] D. DeWitt, P. Futersack, D. Maier, and F. Velez. A study of three alternative workstation-server architectures for object-oriented database systems. In *VLDB*, 1990.
- [53] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *KDD*, pages 43–52, 1999.
- [54] F. Dougliis, T. Ball, Y.-F. Chen, and E. Koutsofios. Webguide: Querying and navigating changes in web repositories. *Computer Networks*, 28(7-11):1335–1344, 1996.
- [55] F. Dougliis, T. Ball, Y.-F. Chen, and E. Koutsofios. The at&t internet difference engine: Tracking and viewing changes on the web. *World Wide Web*, 1(1):27–44, 1998.
- [56] S. T. Dumais, J. C. Platt, D. Hecherman, and M. Sahami. Inductive learning algorithms and representations for text categorization. In *CIKM*, pages 148–155, 1998.
- [57] M. El-Sayed, C. Ruiz, and E. A. Rundensteiner. Fs-miner: efficient and incremental mining of frequent sequence patterns in web logs. In *WIDM*, pages 128–135, 2004.
- [58] O. Etzioni. The world-wide web: Quagmire or gold mine? *Commun. ACM*, 39(11):65–68, 1996.
- [59] G. W. Flake, S. Lawrence, and C. L. Giles. Efficient identification of web communities. In *KDD*, pages 150–160, 2000.
- [60] S. Flesca, F. Furfaro, and E. Masciari. Meaningful change detection on the web. In *DEXA*, pages 22–31, 2001.
- [61] CLUTO 2.1.1: Software for Clustering High-Dimensional Datasets. <http://www.cs.umn.edu/~karypis>.

REFERENCES

- [62] Y. Fu, K. Sandhu, and M. Shih. A generalization-based approach to clustering of web usage sessions. In *WEBKDD*, pages 21–38, 1999.
- [63] Y. Fu, K. Sandhu, and M. Shih. A generalization-based approach to clustering of web usage sessions. In *Proc. of WEBKDD'99*, 1999.
- [64] B. C. M. Fung, K. Wang, and M. Ester. Hierarchical document clustering using frequent itemsets. In *SDM*, 2003.
- [65] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.
- [66] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *ICDM*, pages 163–170, 2001.
- [67] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI*, 2003.
- [68] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. In *ICDE*, pages 106–115, 1999.
- [69] J. Han, W. Gong, and Y. Yin. Mining segment-wise periodic patterns in time-related databases. In *KDD*, pages 214–218, 1998.
- [70] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M. Hsu. Freespan: frequent pattern-projected sequential pattern mining. In *KDD*, pages 355–359, 2000.
- [71] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD Conference*, pages 1–12, 2000.
- [72] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, 1977.
- [73] T. Hofmann. The cluster-abstraction model: Unsupervised learning of topic hierarchies from text data. In *IJCAI*, pages 682–687, 1999.
- [74] V. Hristidis and M. Petropoulos. Semantic caching of xml databases. In *WebDB*, 2002.
- [75] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *ICDM*, pages 549–552, 2003.
- [76] XP: A Simple XQuery Processor Implemented in Java. <http://www.cs.wisc.edu/~mcilwain/classwork/cs764/>.
- [77] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD*, pages 13–23, 2000.

REFERENCES

- [78] A. Inokuchi, T. Washio, K. Nishimura, and H. Motoda. A fast algorithm for mining frequent connected subgraphs. In *Technical Report, IBM Research, Tokyo Research Laboratory*, 2002.
- [79] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.
- [80] B. J. Jansen, A. Spink, J. Bateman, and T. Saracevic. Real life information retrieval: A study of user queries on the web. *SIGIR Forum*, 32(1):5–17, 1998.
- [81] T. Kamdar and A. Joshi. Using incremental web log mining to create adaptive web servers. *Int. J. on Digital Libraries*, 5(2):133–150, 2005.
- [82] C. Kanne and G. Moerkotte. Efficient storage of XML data. In *ICDE*, page 198, 2000.
- [83] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. In *SODA*, pages 668–677, 1998.
- [84] R. Kosala and H. Blockeel. Web mining research: A survey. *SIGKDD Explorations*, 2(1):1–15, 2000.
- [85] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM*, pages 313–320, 2001.
- [86] S. S. Bhowmick L. Chen and W. Nejdl. Mirror site maintenance based on evolution associations of web directories. In *WWW*, 2007.
- [87] B. Lan, S. Bressan, B. C. Ooi, and K.-L. Tan. Rule-assisted prefetching in web-server caching. In *CIKM*, pages 504–511, 2000.
- [88] P. Langley. User modeling in adaptive interfaces, 1992.
- [89] B. Lent, R. Agrawal, and R. Srikant. Discovering trends in text databases. In *KDD*, pages 227–230, 1997.
- [90] E. Leonardi, S. S. Bhowmick, T. S. Dharma, and S. K. Madria. Detecting content changes on ordered xml documents using relational databases. In *DEXA*, pages 580–590, 2004.
- [91] E. Leonardi, S. S. Bhowmick, and S. K. Madria. Xandy: Detecting changes on large unordered xml documents using relational databases. In *DASFAA*, pages 711–723, 2005.
- [92] W. Lian, D. W.-L. Cheung, N. Mamoulis, and S.-M. Yiu. An efficient and scalable algorithm for clustering xml documents by structure. *IEEE Trans. Knowl. Data Eng.*, 16(1):82–96, 2004.
- [93] S. J. Lim and Y.-K. Ng. An automated change detection algorithm for html documents based on semantic hierarchies. In *ICDE*, pages 303–312, 2001.

REFERENCES

- [94] L. Liu, C. Pu, and W. Tang. Webcq: Detecting and delivering information changes on the web. In *CIKM*, pages 512–519, 2000.
- [95] S. K. Madria, S. S. Bhowmick, W. K. Ng, and E.-P. Lim. Research issues in web data mining. In *DaWaK*, pages 303–312, 1999.
- [96] S. K. Madria, C. Raymond, S. S. Bhowmick, and M. K. Mohania. Association rules for web data mining in whoweda. In *Kyoto International Conference on Digital Libraries*, pages 127–139, 2000.
- [97] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *KDD*, pages 210–215, 1995.
- [98] H. Maruyama, K. Tamura, and R. Uramoto. Digest values for dom (domhash) proposal. In *IBM Tokyo Research Laboratory*, 1998.
- [99] K. Maruyama and K. Uehara. Mining association rules from semi-structured data. In *ICDCS Workshop of Knowledge Discovery and Data Mining in the World-Wide Web*, pages F23–F30, 2000.
- [100] Q. Mei and C. Zhai. Discovering evolutionary theme patterns from text: an exploration of temporal text mining. In *KDD*, pages 198–207, 2005.
- [101] A. Meisels, M. Orlov, and T. Maor. Discovering associations in xml data. In *WISE Workshops*, pages 178–183, 2002.
- [102] O. Nasraoui, C. Cardona-Urbe, and C. Rojas-Coronel. Tecno-streams: Tracking evolving clusters in noisy data streams with a scalable immune system learning model. In *ICDM*, 2003.
- [103] O. Nasraoui, R. Krishnapuram, H. Frigui, and A. Joshi. One step evolutionary mining of context sensitive associations and web navigation patterns. In *SDM*, 2002.
- [104] V. Ng, C. C. Kong, and S. H. Wang. Prefetching xml data with abstract query mining. In *ITCC*, page 428, 2004.
- [105] A. Ntoulas, J. Cho, and C. Olston. What’s new on the web? the evolution of the web from a search engine perspective. In *Proceedings of the 13th international conference on World Wide Web*, pages 1–12, 2004.
- [106] B. Özden, S. Ramaswamy, and A. Silberschatz. Cyclic association rules. In *ICDE*, pages 412–421, 1998.
- [107] E. Özkural and C. Aykanat. A space optimization for fp-growth. In *FIMI*, 2004.
- [108] J. S. Park, M.-S. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. In *SIGMOD Conference*, pages 175–186, 1995.

REFERENCES

- [109] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT*, pages 398–416, 1999.
- [110] J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.
- [111] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: Mining sequential patterns by prefix-projected growth. In *ICDE*, pages 215–224, 2001.
- [112] S. Prakash, S. S. Bhowmick, and S. K. Madria. Sucxent: An efficient path-based approach to store and query xml documents. In *DEXA*, pages 285–295, 2004.
- [113] Assignment Problem. http://en.wikipedia.org/wiki/Assignment_problem.
- [114] B. Rácz. nonordfp: An fp-growth variation without rebuilding the fp-tree. In *FIMI*, 2004.
- [115] P. Rao and B. Moon. Prix: Indexing and querying xml using prüfer sequences. In *ICDE*, pages 288–300, 2004.
- [116] UW XML Repository. Dblp computer science bibliography, <http://www.cs.washington.edu/research/xmldatasets/www/repository.html>.
- [117] I. Rigoutsos and A. Floratos. Combinatorial pattern discovery in biological sequences: The teiresias algorithm [published erratum appears in bioinformatics 1998;14(2): 229]. *Bioinformatics*, 14(1):55–67, 1998.
- [118] rsync. <http://samba.anu.edu.au/rsync/>.
- [119] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *VLDB*, pages 432–444, 1995.
- [120] A. Savasere, E. Omiecinski, and S. B. Navathe. Mining for strong negative associations in a large database of customer transactions. In *ICDE*, pages 494–502, 1998.
- [121] H. Schöning and J. Wäsch. Tamino - an internet database system. In *EDBT*, pages 383–387, 2000.
- [122] Google Search. <http://www.google.com>.
- [123] MSN Web Search. <http://search.msn.com>.
- [124] Yahoo Search. <http://search.yahoo.com>.
- [125] searchStorage.com Definitions. Mirror, mirror site, mirroring.
- [126] C. Silverstein, H. Marais, M. Henzinger, and M. Moricz. Analysis of a very large web search engine query log. *SIGIR Forum*, 33(1):6–12, 1999.

REFERENCES

- [127] D. Simovici, N. Singla, and M. Kuperberg. Metric incremental clustering of nominal data. In *ICDM*, pages 523–526, 2004.
- [128] Elearning Web Site. <http://elearn.litespeed.com.sg>.
- [129] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *EDBT*, pages 3–17, 1996.
- [130] Y. G. Sucahyo and R. P. Gopalan. Ct-pro: A bottom-up non recursive frequent itemset mining algorithm using compressed fp-tree data structure. In *FIMI*, 2004.
- [131] P. Tan and V. Kumar. Mining indirect associations in web data. In *WEBKDD*, pages 145–166, 2001.
- [132] P. Tan, V. Kumar, and J. Srivastava. Indirect association: Mining higher order dependencies in data. In *PKDD*, pages 632–637, 2000.
- [133] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison Wesley, 2006.
- [134] W.-G. Teng, M.-J. Hsieh, and M.-S. Chen. On the mining of substitution rules for statistically dependent items. In *ICDM*, pages 442–449, 2002.
- [135] A. Termier, M.-C. Rousset, and M. Sebag. Treefinder: a first step towards xml data mining. In *ICDM*, pages 450–457, 2002.
- [136] H. Toivonen. Sampling large databases for association rules. In *VLDB*, pages 134–145, 1996.
- [137] A. Tung, H. Lu, J. Han, and L. Feng. Breaking the barrier of transactions: Mining inter-transaction association rules. In *KDD*, pages 297–301, 1999.
- [138] G. Valiente. *Algorithms on Trees and Graphs*. Springer-Verlag, 2002.
- [139] G. Valiente. Algorithms on trees and graphs. In *Springer*, 2002.
- [140] Q. Wan and A. An. Efficient mining of indirect associations using hi-mine. In *Canadian Conference on AI*, pages 206–221, 2003.
- [141] J. Wang, J. Han, and J. Pei. Closet+: searching for the best strategies for mining frequent closed itemsets. In *KDD*, pages 236–245, 2003.
- [142] K. Wang and H. Liu. Discovering structural association of semistructured data. *IEEE Trans. Knowl. Data Eng.*, 12(2):353–371, 2000.
- [143] K. Wang, C. Xu, and B. Liu. Clustering transactions using large items. In *CIKM*, pages 483–490, 1999.
- [144] W. Wang, H. Jiang, H. Lu, and J. X. Yu. Bloom histogram: Path selectivity estimation for XML data with updates. In *VLDB*, pages 240–251, 2004.

REFERENCES

- [145] W. Wang, J. Yang, and R. R. Muntz. Tar: Temporal association rules on evolving numerical attributes. In *ICDE*, pages 283–292, 2001.
- [146] W. Wang and O. R. Zaïane. Clustering web sessions by sequence alignment. In *DEXA Workshops*, pages 394–398, 2002.
- [147] Y. Wang, D. J. DeWitt, and J.-Y. Cai. X-diff: An effective change detection algorithm for xml documents. In *ICDE*, pages 519–530, 2003.
- [148] D. Widyantoro, T. Ioerger, and J. Yen. An incremental approach to building a cluster hierarchy. In *ICDM*, pages 705–708, 2002.
- [149] 2005. Answers.com 11 . 2006. Wikipedia. Wikipedia. mirror site.
- [150] X. Wu, M.-L. Lee, and W. Hsu. A prime number labeling scheme for dynamic ordered xml trees. In *ICDE*, pages 66–78, 2004.
- [151] X. Wu, C. Zhang, and S. Zhang. Mining both positive and negative association rules. In *ICML*, pages 658–665, 2002.
- [152] J. Xiao, Y. Zhang, X. Jia, and T. Li. Measuring similarity of interests for clustering web-users. In *ADC*, pages 107–114, 2001.
- [153] Y. Xiao, J. Yao, Z. Li, and M. H. Dunham. Efficient data mining for maximal frequent subtrees. In *ICDM*, pages 379–386, 2003.
- [154] H. Xu, Q. Wu, H. Wang, G. Yang, and Y. Jia. Kf-diff+: Highly efficient change detection algorithm for xml documents. In *CoopIS/DOA/ODBASE*, pages 1273–1286, 2002.
- [155] T. W. Yan, M. Jacobsen, H. Garcia-Molina, and U. Dayal. From user access patterns to dynamic hypertext linking. *Computer Networks*, 28(7-11):1007–1014, 1996.
- [156] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.
- [157] L. Yang, M.-L. Lee, and W. Hsu. Efficient mining of XML query patterns for caching. In *VLDB*, pages 69–80, 2003.
- [158] Q. Yang, H. H. Zhang, and T. Li. Mining web logs for prediction models in www caching and prefetching. In *KDD*, pages 473–478, 2001.
- [159] Y. Yang, J. Carbonell, R. Brown, T. Pierce, B. T. Archibald, and X. Liu. Learning approaches for detecting and tracking news events, 1999.
- [160] X. Yuan, B. P. Buckles, Z. Yuan, and J. Zhang. Mining negative association rules. In *ISCC*, pages 623–628, 2002.

REFERENCES

- [161] M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1/2):31–60, 2001.
- [162] M. J. Zaki. Efficiently mining frequent trees in a forest. In *KDD*, pages 71–80, 2002.
- [163] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *KDD*, pages 326–335, 2003.
- [164] M. J. Zaki and C.-J. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *SDM*, 2002.
- [165] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.
- [166] Q. Zhao, S. S. Bhowmick, and S. K. Madria. Discovering pattern-based dynamic structures from versions of unordered xml documents. In *DaWaK*, pages 77–86, 2004.
- [167] Q. Zhao, S. S. Bhowmick, M. K. Mohania, and Y. Kambayashi. Discovering frequently changing structures from historical structural deltas of unordered xml. In *CIKM*, pages 188–197, 2004.