2005

Mining frequent sequential patterns in data streams using SSM-algorithm.

Mostafa Monwar
University of Windsor

Mining Frequent Sequential Patterns in Data Streams using SSM-Algorithm

By

Mostafa Monwar

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science in
Partial Fulfillment of the Requirements for
The Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2005

Your file Votre référence
ISBN: 0-494-09835-X
Our file Notre référence
ISBN: 0-494-09835-X

Mostafa Monwar 2005

Abstract

Frequent sequential mining is the process of discovering frequent sequential patterns in data sequences as found in applications like web log access sequences. In data stream applications, data arrive at high speed rates in a continuous flow. Data stream mining is an online process different from traditional mining. Traditional mining algorithms work on an entire static dataset in order to obtain results while data stream mining algorithms work with continuously arriving data streams. With rapid change in technology, there are many applications that take data as continuous streams. Examples include stock tickers, network traffic measurements, click stream data, data feeds from sensor networks, and telecom call records. Mining frequent sequential patterns on data stream applications contend with many challenges such as limited memory for unlimited data, inability of algorithms to scan infinitely flowing original dataset more than once and to deliver current and accurate result on demand.

Recent work on mining data streams exists for classification of stream data, clustering data streams, mining frequent patterns over data streams, time series analysis on data stream. However, we rarely see a work done on mining frequent sequential patterns in data streams.

This thesis proposes SSM-Algorithm (sequential stream mining–algorithm) that delivers frequent sequential patterns in data streams. The concept of this work came from FP-Stream algorithm that delivers time sensitive frequent patterns. Proposed SSM-Algorithm outperforms FP-Stream algorithm by the use of a hash based and two efficient tree based data structures. All incoming streams are handled dynamically to improve memory usage. SSM-Algorithm maintains frequent sequences incrementally and delivers most current result on demand. The introduced algorithm can be deployed to analyze e-commerce data where the primary source of the data is click stream data.

Keywords: Data Mining, Web Mining, Large items, Candidate Sequence, Customer Access Sequence, Batch, Frequent Sequential Patterns, Buffer, Click Stream Data.

Dedication

To my mother, my brother and my wife

Acknowledgement

I would like to express my gratitude to the people who helped me while writing this thesis.

First of all, I would like to thank to my supervisor Dr. Christie Ezeife for her invaluable comments, patience, guidance, encouragement, and support during the thesis work and all my graduate studies period. This thesis would not have been completed without her help.

I would also like to thank my internal reader Dr. Randa El-Marakby and external reader Dr. Ihsan S. Al-Aasm for their comments, constructive criticisms, and valuable pieces of advice that helped to improve my thesis. Many thanks go to Dr. Ziad Kobti for chairing my thesis committee.

I would like to thank my mother and brother for their continuous encouragement and moral support.

My special thanks go to my beloved wife Naeema for her enormous understanding and everlasting support that helped me to complete my graduate studies.

At the end, I like to thank my friends for their help and support during my M.Sc study period.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Data Mining

Data mining is the process of extracting knowledge from large amounts of data. This extraction or discovery process has to be automatic and the patterns discovered must be meaningful, so that it can guide decisions about future activities. Data mining discovers which data are important. There is no need to make assumptions about which data are relevant to analyze [Han & Kamber2000].

Data mining tasks can be classified as descriptive and predictive [Han & Kamber2000]. A descriptive model is used to discover interesting patterns based on the general properties in the database. On the other hand, the predictive model is used to make predictions based on data mining result. People can benefit from data mining in various ways. End users can extract business information from large databases, and discover interesting patterns to help make business decisions.

Various analysis or data mining functionalities are available: Concept/Class description, Association Analysis, Classification and Prediction, Cluster Analysis, Outlier Analysis, and Evolution Analysis [Han & Kamber2000]. Below is a brief description of each analysis.

Concept/Class Description: Concept description analysis generates descriptions for characterization and comparisons of data. Sometimes it is called Class Description Analysis. For example, a sales manager of a company may want to view the data from generalized to higher levels (summarized by customer groups according to geographic regions, frequency of products per group, and customer income) instead of examining individual customer transactions [Han & Kamber2000].

Association Analysis: Association rule mining finds interesting association or correlation relationships among a large set of data. Sometimes association rule mining is referred to as market basket analysis. This process analyzes customers' buying habits by finding correlation between the different items in the customers' shopping basket. For example, if customers are buying milk, how likely they also buy bread on the same trip to the supermarket. Association Rule is being used in various sectors besides super market basket analysis now-a-days. For example, proposes to use association rule discovery methods for determining associations among expression levels of different genes

[Tuzhilin & Adomavicius2002]. Section 2 provides more details on Association Rule mining.

Classification and Prediction Analysis: Classification is the process of finding a set of models (or functions) that describe and distinguish data classes or concepts. Later, these models are used to predict the class of objects whose class labels are unknown (i.e. training data). The derived model may be represented in various forms, such as decision trees, mathematical formulae, or neural networks [Han & Kamber2000].

Cluster Analysis: Clustering is the process of grouping a set of physical or abstract objects into classes of similar objects. Cluster analysis has numerous uses in various applications. For example, [Gunduz & Ozsu2003] introduced a model that takes the sequences of visiting pages and time spent on each page by the users as input. It is a two dimensional mining approach. One dimension is page sequences and another dimension is time. Based on the similarities (sequences and time) of users, the proposed model clusters user sessions. Clustering of sessions enables to reduce search spaces. Clustering can be used for other purposes such as pattern recognition, data analysis, image processing, and market research.

Outlier Analysis: A database may contain data objects that do not fulfill with the general behavior (model) of the data. These data objects are outliers. Usually, data mining method discards outliers as noise or exceptions. However, in some applications, these outliers could be more interesting, for example, fraud detection. The analysis of outlier data is called outlier mining [Han & Kamber2000].

Evolution Analysis: This type of analysis describes regularities or trends for objects whose behavior changes over time. For example, an evolution analysis on stock exchange data may identify stock evolution regularities for overall stocks and for the stocks of particular companies [Han & Kamber2000].

1.2 Web Mining

Web mining is similar to data mining. When data mining techniques are applied to the web, data mining is called web mining [Xie & Phoha2001]. In other words, web mining is used to discover a set of interesting patterns from the web by applying automatic mining techniques [Dutta et al.2001]. Interesting patterns can be discovered from web

contents or web pages, web links in the web pages and web logs. In web mining, the source data can be collected at the server-side, client-side, proxy-server or organization's database [Srivastava et al.2000]. The benefit behind mining the web is to improve the web site by better understanding the users' interests, typical paths, and the correlation between customer behavior and the products [Theusinger & Huber2000]. Yang et al. [Yang, Haining, & Li2001] have suggested an interesting method to reduce network latency by applying web mining techniques. The method finds frequent access patterns from web logs by analyzing click stream data. Once frequent access pattern is found, it can be used in web caching and pre-fetching system in order to improve the hit rate and reduce unnecessary network traffic.

Web mining are classified into 1) Web content mining, 2) Web structure mining, 3) Web usage mining [Cooley, Mobasher & Srivastava1997] [Liu, Chen & Song2002] [Zhang & Chang2002] [Woon, Ng, & Lim2002] [Mah, Hoek & Li2001] [Srivastavaetal2000] [Kosala & Blockeel2000].

Web Content Mining: Web content is the content or information of a web page. This content information can be in an HTML or XML file format. They are usually in unstructured or semi structured format. Basically, web content mining aims to extract / mine useful information or knowledge from the content of the web pages [Liu & Chang2004].

The basic framework for mining web content is to structure the data from the unstructured or semi structured data format and discover interesting relationships from the structured data. Web content mining is still a new area in knowledge discovery and a little work has been done so far.

Web Structure Mining: Web pages are connected to each other by hyperlinks. Web structure mining is the process of discovering interesting patterns from hyperlinks structure within the web itself. The hyperlink structure in the web can be used by web structure mining to apply social-network analysis [Kosala & Blockeel2000]. It is possible to discover specific types of pages with social-network analysis based on incoming and outgoing links.

Web Usage Mining: Web usage mining is the process of discovering interesting relationships and global patterns in large access-log files [Masseglia, Teisseire &

Poncelet2002]. When a user visits a web site, the web server usually keeps some important information about the user in the log file, for example, the client's IP address, the URL requested by the client, and the date and time for that request [Masseglia, Teisseire & Poncelet2001]. This log file is a flat file. Three types of log files are available on a web server: server-logs, error-logs, cookie-logs [Buchner & Mulvenna1998]. Analyzing data on the web server-logs is called "click stream analysis" [Dutta et al.2001]. E-commerce or e-business can highly benefit from this click stream analysis. It provides important information to understand better the marketing and merchandising efforts, for example, how customers find the store, what products they search for, and what products they purchase [Lee & Podlaseck2000]. There are more discussion is ahead about various techniques of click stream data analysis.

The possible sources of Web Usage Data are below:

- Click stream Data: when a user visits a web site, it is possible to keep the record of IP-Address, requested URL from users, time stamp of the event from each click. This record is stored in common log format in web server log [Dutta et a.2001].
- Cookie Logs Data: The use of click stream data alone is not enough for mining for all of the cases. Dynamic IP-Address changes from session to session. The same user may visit several times with different IP-Addresses, which may reduce the accuracy of mining result. By placing a cookie on a user's computer, it is possible to trace the same user from the cookie logs in web server when the user uses dynamic IP-Addresses [Buchner & Mulvenna1998] [Srivastava et al.2000].
- TCP / IP packets are an alternative source of web usage data. Packet sniffers monitor incoming traffic to the web servers and extract data from the TCP/IP packets [Srivastava et al.2000].
- Query Data: Query data are usually generated when a customer searches for products on an e-commerce site [Buchner & Mulvenna1998].

1.2.1 Phases of Web Usages Mining

Web usage mining has three phases: preprocessing, knowledge or pattern discovery and patterns analysis [Zhang & Chang2002] [Srivastava et al.2000].

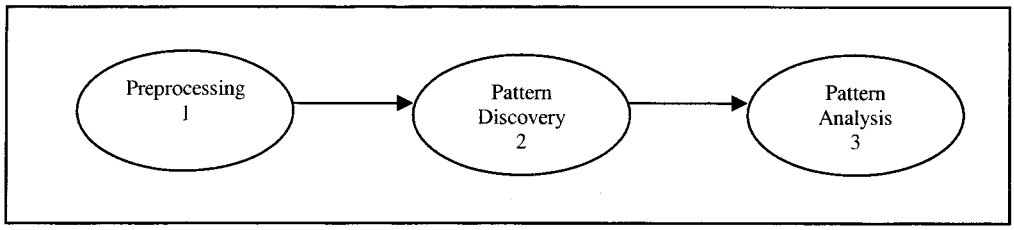

Figure 1.1.3- 1: The phases of Web Usage-mining

1.2.1.1 Preprocessing

Ansari et al. [Ansari et al.2001] state that preprocessing constitutes about 80% of the work of data mining tasks. The following tasks may be included in the preprocessing phase: data-cleaning, user-identification, session-identification, path-completion, transaction-identification, formatting [Cooley, Mobasher & Srivastava1999] [Woon, Ng, & Lim2002] [Nahm, Bilenko & Mooney2002]. However, the tasks will be included in preprocessing phase based on the mining interest. Let's discuss a few tasks of the preprocessing phase.

1.2.1.2 Data Cleaning

Data-Cleaning is the task of removing irrelevant and noisy data from the server-logs for mining purposes. For instance, the HTTP protocol requires a separate connection (for stateless connection) for every file that is requested from the server [Cooley, Mobasher & Srivastava1997]. In this case, when a user requests a page to view, all of the scripts and files that are embedded with this particular page get recorded in the log file during the download process. As a matter of fact, only the HTML page should be listed in the navigational path for mining interest, not the graphics, music or scripts. Therefore, it is necessary to eliminate the entries with file name suffixes (e.g. gif, jpeg, jpg, mpg, etc).

1.2.1.3 User Identification

Users are identified by their IP-Addresses. However, this simple task becomes complicated when users share machines or when users are connected to Internet through proxy servers. A proxy server sits in between a client application (i.e. web browser) and a real web server. All requests from clients to the real server go via proxy server. The proxy server interprets all requests and check whether it can fulfill the requests itself before it forwards the requests to the web server. If it cannot fulfill the requests, it forwards the request to the real server. Proxy servers have two main purposes: improve performance and filter requests. To resolve this kind of issues, Cooley et al. [Cooley, Mobasher & Srivastava1999] have proposed log/site based method to identify users heuristically.

1.2.1.4 Formatting

Formatting is the end part of preprocessing phase after taking care of all necessary steps for cleaning the data. In this stage, a final preparation module can be used to properly format or transform the data for mining to be accomplished. In other words, structured data is used in order to run mining algorithm.

1.2.2 Knowledge Discovery

The following mining techniques or analysis can be applied during the knowledge discovery phase on web usage data [Srivastava et al.2000][Woon, Ng, & Lim2002].

1.2.2.1 Mining Techniques

- Statistical Analysis is used to find statistical information, for example, the most frequent accessed pages and average page view duration [Woon, Ng, & Lim2002].
- Path Analysis is used to find the most frequent traversal paths in site for efficient web site design [Berkhin, Becher, & Randall2001] [Keahey & Eick2002].

- Association Rule Mining is used to discover correlations among the web pages for structuring web sites [Agrawal, Imielinski, & Swami1993] [Agrawal & Srikant1994].
- Sequential Pattern Mining is used to discover sequential patterns for predicting the users visit patterns or buying habit [Agrawal & Srikanth1995] [Srikanth & Agrawal1996] [Hin2002] [Lu & Ezeife2003] [Ezeife & Chen2004a] [Ezeife & Chen2004b] [Ezeife & Lu2005](More discussion on section 2.3).
- Clustering is used to group together the users based on their similar interest for web content personalization [Xiao & Zhang2001].
- Classification is used to group together into predefined classes based on their similar interests for customer profiling [Hu & Cercone2002].

Association Rules Mining, Sequential Pattern Mining and Clustering are most widely used in web usage mining [Cooley, Mobasher & Srivastava1999].

1.2.3 Pattern Analysis

This is the last stage of web usage mining. After applying all of the mining techniques or algorithms, it is the time to analyze the patterns and filter out uninteresting rules or patterns that are discovered during the knowledge discovery phase.

1.3 The motivation for this thesis

A data stream is a continuous, unbounded, and high-speed flow of data items. Many applications generate a large amount of data streams, for example, network monitoring, traffic telecom, call detail records (CDR), ATM operations in banks, sensor networks, web logs and web click streams, transactions in retail chains and many others. Mining data in such a data mode is referred to as stream mining. Stream mining adds many complexities to traditional mining requirements. 1) The primary problem with stream mining is that there is a massive volume of data arriving everyday, every hour and we have to look for knowledge in it. As the volume of the data is too high, therefore, it is

difficult to store all of them. 2) The system will run out of memory if we apply our traditional mining algorithm on such high volume data. 3) As data come in a continuous fashion, the mining algorithm would not have enough time to scan the original dataset more than once. 4) A method to deliver considerably accurate result on demand when needed. 5) As we are specifically interested in finding sequential patterns in click stream data, we need to keep Customer Access Sequences (CAS) intact. CAS is the access order by a customer. Keeping CAS intact for each transaction and mining sequential patterns in it is a cumbersome process. So, we envision a continuous process that extracts frequent CAS from incoming stream, mines sequential patterns and store the patterns compactly and quickly. It also important to handle everything dynamically because we assume that we do not know how many click stream items we have to deal with before we start mining. Therefore, we need a data structure that grows with incoming stream dynamically.

A busy website generates a huge amount of click stream data everyday. Each click stream data series reflects a customer's buying interest. For an e-commerce company, detecting future customers based on their sequential mouse movements on the content page would help significantly to generate more revenue.

There are some recent studies on mining data streams, classification of stream data [Domingos & Hulten2000], online classification of data streams [Las2002], clustering data streams [Guna et al.2000] [Guna et al.2003], web session clustering [Gunduz & Ozsu2003], approximate frequency counts over data streams [Manku & Motwani2002], mining frequent patterns in data stream at multiple time granularities [Giannella et al.2003], a symbolic representation of time series analysis [Lin et al.2003], and multi-dimensional time series analysis [Chen et al.2002], Temporal pattern mining in data streams [Teng, Chen & Yu2003] but there is still an open area to mine frequent sequential patterns in data streams.

Considering the importance of sequential mining in data streams, we have developed SSM-Algorithm that uses three data structures to generate frequent sequences continuously. Our method uses a technique to process the stream data on arrival, SSM-Algorithm (sequential stream mining-algorithm) forms dynamic sized batches by taking

data from the data stream, performs batch mining and constantly stores result on a compact tree that is able to deliver results on demand.

1.4 Contribution of Thesis

This thesis proposes SSM-Algorithm that uses three types of data structures (d_list, PLWAP and FSP-tree) to handle and solve the complexities of mining frequent sequential patterns in data streams.

The concept of this work came from FP-Stream [Giannella et al.2003] algorithm. FP-Stream [Giannella et al.2003] delivers time sensitive frequent patterns in data stream. Please note here, frequent patterns and frequent sequential patterns are two different techniques. Frequent sequential patterns require to keep customer access order intact prior to mining. On the other hand, frequent pattern does not care about the customer accessing order. Data structure d_list uses hash chain indexing in order to maintain incoming elements and their frequency. D_list was not introduced in FP-Stream in order to make the mining process fully dynamic and quick. It is very efficient if there are thousands of items are used in e-commerce, brand new items get posted to the e-commerce site and unpopular items get discontinued from the site on regular basis.

Proposed algorithm handles dynamic sized batches. FP-Stream [Giannella et al.2003] or Lossy Counting algorithm [Manku & Motwani2002] use fixed size batches. The advantages of using dynamic sized batches over fixed size batches are the size of the batch can grow bigger and smaller with incoming streams. If the environment is a heavy stream environment, then it would be efficient to have bigger batches. On the other hand, if the environment is a low stream environment, then it would be efficient to have smaller batches to avoid long time waiting period to form a batch.

SSM-Algorithm uses previously introduced PLWAP-tree algorithm to find frequent sequential patterns. Proposed algorithm takes the advantage of preordered linkage and position coding of PLWAP-tree in order to eliminate the cost of reconstruction and computation time of intermediate trees unlike FP-tree [Han, Pie & Yin2000] [Han et al.2004]. FP-Stream [Giannella et al.2003] uses FP-tree to generate frequent patterns.

FSP-tree (frequent sequential pattern-tree) is used in SSM to store obtained frequent sequential patterns. FSP-tree is a simple form of pattern-tree [Giannella et al.2003] that is

introduced in FP-Stream algorithm. FSP-tree stores and maintains frequent sequential patterns constantly and that is capable of delivering results on demand. On the other hand, pattern-tree [Giannella et al.2003] maintains and delivers frequent patterns that is capable of delivering time sensitive frequent queries.

The produced result by SSM-Algorithm does not cross pre-defined error threshold and delivers all frequent sequential patterns that have user defined minimum support. The use of buffer mechanism in SSM restricts memory usage to a specific size. In other words, we use a small fixed size buffer in the memory that handles continuous incoming data streams.

1.5 Outline of the Thesis Proposal

The remainder of the thesis is organized as follows: Chapter 2 reviews most related work. Chapter 3 describes detailed discussion of the new algorithm sequential stream mining algorithm (SSM-Algorithm) and the data structures that are used by SSM-Algorithm to discover sequential patterns in data stream. Chapter 4 presents the analysis on SSM-Algorithm and its implementation and testing result. Chapter 5 draws the conclusions of this research and discusses future work in this direction.

2. Related Work

Association Rules

An association rule has the form of X →Y. Here X is the left side of the rule and it is called antecedent and Y is right side of the rule, it is called consequent. Say, X and Y are the sets of items (attributes) of a transaction database *D*. Two important features of association rule are: support and confidence. The support of an itemset X is the fraction of database transaction that contains X. The support of a rule of the form X → Y is then the same as the support of X ∪ Y and its confidence is the ratio of the supports of X ∪ Y and X. Association rules are used to find the correlation between antecedent and consequent.

2.1 Apriori

Agrawal et al. [Agrawal, Imielinski, & Swami1993] have introduced the AIS algorithm to apply association rules to basket data and to discover interesting patterns within the basket items. AIS is a time consuming algorithm because it scans the database many times in the process of creating large itemsets. Later, Agrawal and Srikant [Agrawal & Srikant1994] have introduced the Apriori algorithm as a faster algorithm for mining association rules in a large database. Over time, the Apriori algorithm became very popular algorithm for mining association rules. [Agrawal & Srikant1994] introduced an Apriori series, composed of three algorithms: Apriori, AprioriTid, and AprioriHybrid.

The idea behind that series is to use Apriori-gen [Agrawal & Srikant1994] function to reduce the candidate itemsets. Apriori algorithm finds all the large itemsets iteratively. In the first iteration, it generates all candidate 1-itemsets (denoted as C_1). Then the database is scanned and only those items equal to or greater than minimum support are selected as large 1-itemsets (denoted as L_1).

In next iteration, C_2 is generated by using apriori-gen function [Agrawal & Srikant1994]. Apriori-gen function joins L_1 and L_1 conditionally. The conditions are as follows: 1) Conditional join L_1 and L_1 means when one itemset X is chosen from the first L_1, another itemset Y is chosen from the second L_1, the last item of X must be less than the last item of Y, the rest of the items of X must be same as Y. Then, the function inserts those itemsets into C_2. The last step of this function is to prune out all candidate

itemsets in C_2 that have any of their subsets not in L_1. The large 2-itemset L_2 is again generated from C_2 by selecting items that meet the minimum support threshold. This process stops when either C_i or L_i is empty set. An example of the Apriori algorithm is given below. Assume Table 2.1-1 is fraction of a transactional database *D*, user specified minimum-support is 2.

- In the first iteration, it generates all candidate 1-itemsets listed with their support counts as C_1= {Bread:3, Butter:1, Cheese:3, Egg:2, Milk:3}. Next task or step

TID	Items Purchased
1	Butter, Egg, Milk
2	Bread, Cheese, Milk
3	Bread, Cheese, Egg, Milk
4	Bread, Cheese

Table 2.1- 1: A sample of transactional database *D*

in first iteration is to generate large 1-itemsets L_1 from C_1 by eliminating the items that do not have minimum support (2) in C_1. From Table 2.1-1, we find L_1={Bread:3, Cheese:3, Egg:2, Milk:3}. Item butter appeared in only one transaction (its support is 1) and that is less than minimum support threshold. So, we omited item butter in 1–large itemsets (L_1).

Itemset	**Suport**	**Itemset**	**Support**	**Itemset**	**Support**
{Bread}	3	{Bread, Cheese}	3	{Bread, Cheese, Milk}	2
{Cheese}	3	{Bread, Milk}	2		
{Egg}	2	{Egg, Milk}	2		
{Milk}	3	{Cheese, Milk}	2		
L_1		**L_2**		**L_3**	

Table 2.1- 2: Large Itemsets generation

- During the second iteration, the first task will be to generate all candidate 2-itemsets C_2 by joining L_1 with L_1 using Apriori-Gen function [Agrawal & Srikant1994]. After joining, found C_2 = {(Bread, Cheese):3, (Bread, Egg):1, (Bread, Milk):2, (Cheese, Egg):1, (Cheese, Milk):2, (Egg, Milk):2}. The next task or step in second iteration is to generate L_2 itemset by filtering the items that have up to minimum-support of 2 in C_2. In our example, we omit itemsets (Bread, Egg) and (Cheese, Egg) because they do not meet minimum support threshold. After filtering, we find L_2 = {(Bread, Cheese):3, (Bread, Milk):2, (Cheese, Milk):2, (Egg, Milk):2}.
- During the third iteration, the first task will be to generate all candidate 3-itemsets (C_3) by joining L_2 with L_2 using Apriori-Gen function [Agrawal & Srikant1994]. After joining, we find C_3 = {(Bread, Cheese, Milk):2}. Next step is in third iteration, to generate L_3 itemsets by filtering the items that have support greater than or equal to minimum-support in C_3. We find L_3 = {(Bread, Cheese, Milk):2}. We try to generate C_4, but it turns out to be empty and the process stops here.

The final Large itemsets $L = L_1 \cup L_2 \cup L_3$ (Table 2.1-2).

L = { Bread:3, Cheese:3, Egg:2, Milk:3, (Bread, Cheese):3, (Bread, Milk):2, (Cheese, Milk):2, (Egg, Milk):2, (Bread, Cheese, Milk):2}

Apriori and AprioriTid use the same algorithm to generate large itemsets. The difference between Apriori and AprioriTid is that AprioriTid keeps an auxiliary dataset after first pass over (transaction from C_1 to L_1). The auxiliary datasets keep the record of candidate-list for every transaction. Unlike Apriori, AprioriTid uses auxiliary database to count the support instead of rescanning the database. It is proven that AprioriTid is faster than Apriori. However, AprioriTid requires a good amount of memory and in some cases, it may exceed the main memory. To solve this problem, Agrawal and Srikant [Agrawal & Srikant1994] proposed an algorithm called AprioriHybrid. AprioriHybrid is a combination of Apriori and AprioriTid which takes advantages of both methods. AprioriHybrid starts with Apriori method and switches to AprioriTid as soon as the system determines that adequate memory is available for data.

2.2 Apriori-Growth

The Apriori series [Agrawal & Srikant1994] requires the generation of a good number of candidate itemsets. Generating candidate itemsets is costly. In order to reduce the cost and candidate itemsets, Han et al. [Han et al.2004] [Han, Pie & Yin2000] [Han & Kamber2000] proposed the FP-Tree algorithm to generate frequent pattern itemsets.

To review, the FP-Tree algorithm [Han et al.2004] [Han, Pie & Yin2000] assumes a transactional database *D* has the following records (Table 2.2-1, columns' 1 and 2) and user specified minimum-support is 3. The transaction items need to be scanned once because only the frequent items will play a role during the

TID	Items Purchased	(Ordered) Frequent Items
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	f, c, e, l, p, m, n	f, c, a, m, p

Table 2.2- 1: Ordered Frequent Items

construction of FP-Tree. After scanning *D* once, we find the frequent 1-items in frequency descending order are {(f:4), (c:4), (a:3), (b:3),(m:3),(p:3)}. Then, the items of each transaction need to be ordered in frequency descending order (Table 2.2-1, column 3). Next task will be to create the virtual root labeled 'null'. Now, we scan the database again and start constructing the branch of the tree with the first transaction with TID 100. For TID 100, we take items <(f:1), (c:1), (a:1), (m:1), (p:1)> as they are ordered in the terms of frequency rather than the order of the items appear in the transaction. In our example, we insert 'f' as first node under the root and the initial count of 'f' is 1. Now 'c' goes under 'f' node as child node and put initial count as 1. Similarly, we place 'a', 'm', and 'p' (Figure 2.2-1). We are done with TID 100 and our next TID is 200. As per rule, we take ordered items <(f:1), (c:1), (a:1), (b:1), (m:1)> (see Table 2.2-1, column 3) to construct another branch. First 3 items (f, c, a) of TID 200 are matching with the first three items of TID 100. In this case, we increment the count of (f, c, a) by one in the branch and create one new node (b:1) as a new branch and link it as a child of (a:2). Similarly, we create a new node (m:1) and link as a child of (b:1) node. The rest of the

items of the transactions will be inserted accordingly. The final constructed tree looks like Figure 2.2-1.

Once the tree construction is completed, mining algorithm FP-Growth starts from the bottom of the header table. For node p, it derives a frequent pattern (p:3) from the path (f:4, c:3, a:3, m:2, p:2) and (c:1, b:1, p:1) of the tree. The prefix (f,c,a,m) happened together two times with the event "p" and (c,b) happened once with "p". Here, p's sub-patterns (f:2,c:2,a:2,m:2) and (c:1,b:1) form the conditional pattern bases of p. After

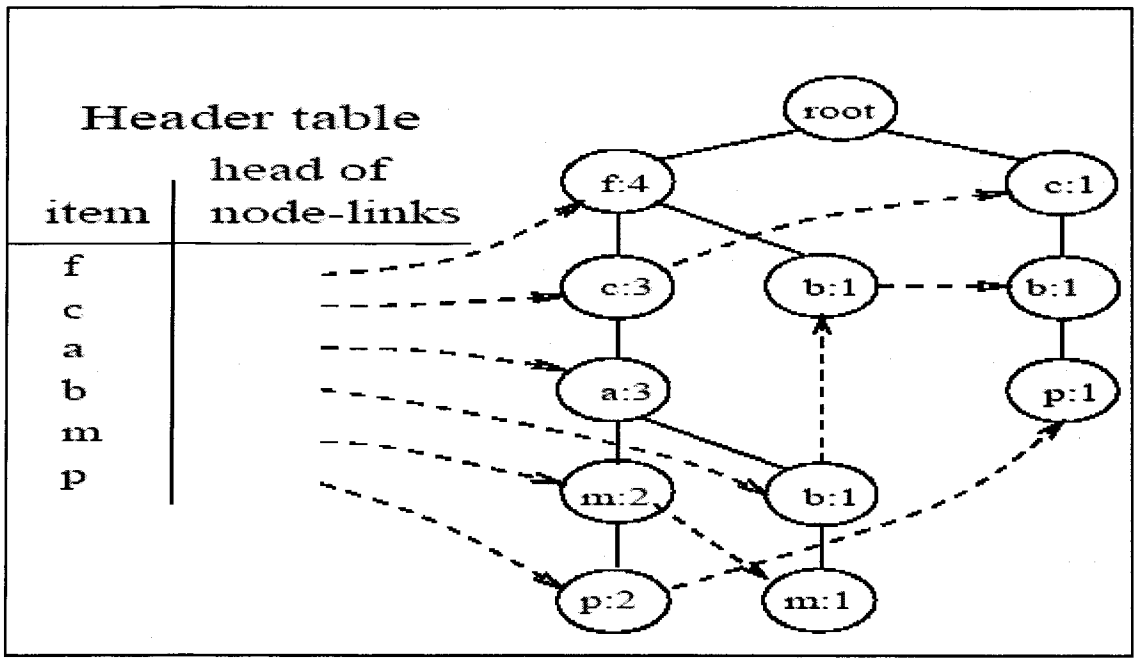

Figure 2.2- 1: FP-Tree constructed from TID 100, TID 200, TID 300, TID 400, TID 500

Item	Conditional Pattern Base	Conditional FP-tree
P	{(f:2,c:2,a:2,m:2),(c:1,b:1)}	{(c:3)}|P
M	{(f:2, c:2, a:2), (f:1,c:1,a:1,b:1)}	{(f:3,c:3,a:3)}|m
B	{(f:1,c:1,a:1,b:1),(f:1,b:1),(c:1,b:1)}	0
A	{(f:3,c:3)}	{(f:3,c:3)}|a
C	{(f:3)}	{(f:3)}|c
F	0	0

Table 2.2- 2: Complete conditional patterns

constructing a p-conditional FP-tree, we can find frequent pattern (cp:3) from only one branch (c:3). The p-conditional search stops and we start with the next item (m) from bottom to top in the header table. For item m, the frequent pattern (m:3) derived from two paths are (f:4, c:3, a:3, m:2) and (f:4, c:3, a:3, b:1, m:1). Similarly, m's sub-patterns (f:2,c:2,a:2) and (f:1,c:1,a:1b:1) form the conditional pattern bases of m. Constructing m-conditional FP-tree, a single frequent pattern (f:3, c:3, a:3) is obtained. From this single pattern all m-conditional frequent patterns can be found by combining all of its items <(m:3), (am:3), (cm:3), (fm:3), (cam:3), (fam:3), (fcm:3)>. The conditional pattern base and conditional FP-tree for other items in the header table could be obtained in similar way. Table 2.2-2 shows the complete conditional pattern. The maximal frequent patterns {cp}, {fcam}, {b} can be obtained recursively.

Sequential Mining

Sequential patterns are used to predict the sequential order of items. For example, typically if a customer rents "Start Wars", then rents "Empire Strike Back", then "Return of the Jedi". AprioriAll and AprioriSome algorithms were introduced in [Agrawal & Srikanth1995] for the first time to extract sequential patterns in a transaction database. Below is the discussion on AprioriAll and AprioriSome.

2.3 AprioriALL

The algorithm of AprioriALL is similar to basic Apriori algorithm. The only difference is AprioriALL is used for mining sequential patterns. AprioriALL finds maximal large sequences among all sequences that have a certain user-specified minimum support (or large sequences). In AprioriALL the term large sequences are used unlike large itemsets in Apriori. In large sequences a sequence is maximal if that particular sequence is not contained in any other sequences. Let's go through an example to find maximal sequences using AprioriALL algorithm. Consider Table 2.3-1 as a portion of a database *D*. In Table 2.3-1, there are records of items that are purchased by five customers. The minimum user-specified support is 40% or 2 customer sequences. We follow same procedure as Apriori [Agrawal & Srikant1994]. In Table 2.3-3, we generate large 1-

sequences or L_1 from candidate 1-sequences or C_1 (Table 2.3-2). In our example, all of C_1 have minimum-support (Table 2.3-2). Therefore, they are in L_1 as well. Next task to generate candidate 2-sequences (C_2) by joining L_1 with L_1 using apriori-generate function [Agrawal & Srikant1994] (Table 2.3-4). Sequence (2 5) is filter out from C_2 (because it does not have minimum-support) to generate large 2-sequences or L_2 (Table 2.3-5).

TID	ITEMS
100	({1 5} {2} {3} {4})
200	({1} {3} {4} {3 5})
300	({1} {2} {3} {4})
400	({1} {3} {5})
500	({4} {5})

Table 2.3- 1: Customer Sequences

Sequence	Support
(1)	4
(2)	2
(3)	4
(4)	4
(5)	2

Table 2.3- 2: C_1

Sequence	Support
(1)	4
(1)	4
(2)	2
(3)	4
(4)	4

Table 2.3- 3: L_1

Sequence	Support
(1 2)	2
(1 3)	4
(1 4)	3
(1 5)	3
(2 3)	2
(2 4)	2
(2 5)	0
(3 4)	3
(3 5)	2
(4 5)	2

Sequence	Support
(1 2)	2
(1 3)	4
(1 4)	3
(1 5)	3
(2 3)	2
(2 4)	2
(3 4)	3
(3 5)	2
(4 5)	2

Table 2.3- 4: C_2

Table 2.3- 5: L_2

Sequence	Support
(1 2 3)	2
(1 2 4)	2
(1 3 4)	3
(1 3 5)	2
(1 4 5)	1
(2 3 4)	2
(3 4 5)	1

Table 2.3- 6: C_3

Sequence	Support
(1 2 3)	2
(1 2 4)	2
(1 3 4)	3
(1 3 5)	2
(2 3 4)	2

Table 2.3-7: L_3

Sequence	Support
(1 2 3 4)	2

Table 2.3-8: C_4

We generate candidate 3-sequences (C_3) by joining L_2 with L_2 using apriori-generate function (Table 2.3-6). Sequence (1 2 5) is pruned out in C_3 because its subsequence (2 5)

Sequence	Support
(1 2 3 4)	2

Table 2.3-9: L_4

Sequence	Support
(1 2 3 4)	2
(1 3 5)	2
(4 5)	2

Table 2.3-10: Maximal Large Sequences

is not in L_2. Sequences (1 4 5) and (3 4 5) are filtered out to generate L_3 in C_3 because they do not have minimum-support (Table 2.3-7).

Next generate candidate 4-sequences (C_4) by joining L_3 with L_3 using apriori-gen function (Table 2.3-8). Sequence (1 3 4 5) is pruned out in C_4 because its subsequence (3 4 5) is not in L_3. The only sequence (1 2 3 4) is found as candidate 4-sequence or C_4 (Table 2.3-8). Sequence (1 2 3 4) is also considered as large 4-sequence (L_4) because it has minimum-support (Table 2.3-9). The process stops generating sequences at C_5 because it is an empty set.

Now is the time to generate maximal sequences among all large sequences that we have generated so far. Sequence (1 2 3 4) is maximal sequence in L_4 because it is not a subsequence of any other sequences, the only sequence (1 3 5) is selected in L_3 as maximal large sequence because it is not a subsequence of (1 2 3 4). Similarly, sequence (4 5) is maximal sequence in L_2 because other sequences in L_2 are the subsequences of (1 2 3 4) and (1 3 5) that are selected as maximal large sequences previously. The final maximal large sequences are (1 2 3 4), (1 3 5) and (4 5).

2.4 AprioriSome

This algorithm works through two phases: forward pass phase and backward pass phase [Srikanth & Agrawal1996]. We will go through sample example that is used in AprioriALL to understand AprioriSome. In the forward pass phase, first task to generate C_1 and compute L_1 from C_1 (Table 2.3-3). Second, task to generate C_2 by joining L_1 and L_1 using apriori-gen function and we compute L_2 from C_2 (Table 2.3-5).

Sequence	Support
(1 2 3 4)	2
(1 3 4 5)	1

Table 2.4- 1: C_4

Sequence	Support
(1 2 3 4)	2

Table 2.4- 2: L_4

Sequence	Support
(1 3 5)	2
(1 4 5)	1
(3 4 5)	1

Table 2.4- 3: C_3 (after pruning)

C_3 is generated by joining L_2 and L_2 using apriori-gen function (Table 2.3-6). Now, we do not compute L_3 from C_3 because we do not generate L_3. Instead we generate C_4 by joining C_3 and C_3 using Apriori-gen function (Table 2.4-1). L_4 is computed from C_4 (Table 2.4-2). Next, C_5 turns out to be an empty set. The process starts backward phase, nothing gets deleted from L_4 because sequence (1 2 3 4) is the only sequence in L_4 and it counted as maximal large 4-sequence. As per rule, the sequences get deleted in C_3 if they are the subsequences of (1 2 3 4) (Table 2.4-3). In AprioriALL, a total of seven candidate 3-sequences are generated but in AprioriSome, a total of three candidate 3-sequences are generated. Sequence (1 3 5) get selected as maximal large 3-sequence in C_3 (Table 2.4-3). All of the sequences are deleted in L_2 except (4 5) sequence because the deleted sequences are contained in maximal large-4 sequence and maximal large 3-sequence. For the same reason all sequences in L_1 are also deleted. The final maximal large sequences are (1 2 3 4), (1 3 5) and (4 5).

2.5 GSP-Algorithm (Generalized Sequential Patterns-Algorithm)

Srikanth & Agrawal presented GSP algorithm [Srikanth & Agrawal1996] that incorporates three important issues in sequential patterns:

1) Absence of time constraints: a minimum and a maximum time gap require for a particular sequence. For example, if a customer bought a book named "Foundation", followed by another book named "Foundation and Empire" three years later, then, there should be a time constraint that the above customer sequence would be counted if adjacent elements occur within a specified time interval, say four months.

2) Sliding time window: for some applications, it is acceptable if items in an element of a sequential pattern were present in two different transactions as long as the

difference between the maximum and minimum transaction times is less than the size of a sliding time window or sliding time frame. For example, if a bookstore specifies a time window of a week, a customer who purchased "Foundation" on Monday "Ringworld" on Saturday and in few weeks later, the same customer purchased at the same time "Foundation and Empire" and "Ringworld Engineers" would still support the pattern "Foundation and Ringworld" followed by "Foundation and Empire" and "Ringworld Engineers".

3) Absence of taxonomies: many datasets have a user-defined taxonomy (hierarchy) over the items in the data and users may want to find patterns that include items across different levels of the hierarchy. For example, if a customer bought "Coke" followed by "Potato Chips", would support the patterns, "Coke", followed by "Potato Chips", "Soft Drinks" followed by "Potato Chips", "Drinks" followed by "Crackers" (Figure 2.5-1 and 2.5-2).

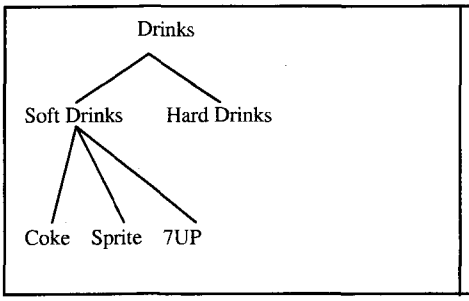

Figure 2.5- 1: Taxonomy 1

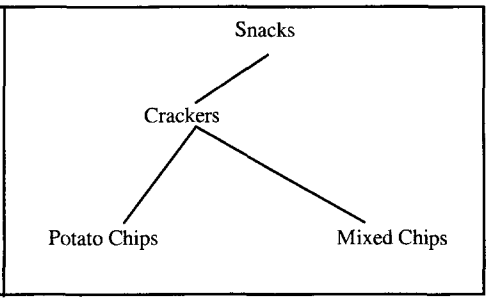

Figure 2.5- 1: Taxonomy 2

GSP algorithm's main advantage over AprioriALL is that GSP counts fewer candidates than AprioriALL. AprioriALL prunes candidate sequences by checking if the subsequences obtained by dropping an element have minimum support. On the other hand, GSP checks if the subsequences obtained by dropping an item have minimum support. Therefore, GSP always counts fewer candidates than AprioriALL. For the same reason, GSP performs several times faster than AprioriALL.

2.6 WAP Tree

Pei at al. [Pei et al.2000] stated that GSP [Srikanth & Agrawal1996] is efficient if sequences and transactions are not too long. When the length of the sequences increases

and the transactions are large, the number of candidate sequences may increase very substantially and GSP may meet difficulties.

Pei et al. have introduced Web Access Pattern tree or WAP-tree [Pei et al.2000] for mining web logs efficiently. This tree stores highly compressed and critical information for access pattern mining. The construction of WAP-tree is similar to FP-tree, but WAP mine algorithm is incorporated with WAP-tree algorithm for sequential mining from large web logs. Moreover, the performance of WAP-tree is faster than its Apriori based counterparts for mining web access patterns. The steps for WAP-mine algorithm are:

1) Find all frequent events. 2) Construct a WAP-tree over the set of frequent events.

3) Mine the WAP-tree using conditional search recursively.

Below given a web-log sequence:

<100,a><100,b><200,a><300,b><200,b><400,a><100,a><400,b><300,a><100,c>
<200,c><400,a><200,a><300,b><200,c><400,c><400,c><300,a><300,c>

User ID	Web Access Sequences	Frequent Subsequences
100	abdac	abac
200	eaebcac	abcac
300	babfaec	babac
400	afbacfc	abacc

Table 2.6- 1: A transformed database of web access sequences (WAS)

After the preprocessing phase, we find Candidate 1- Sequences are <a:4, b:4, c:4, d: 1, e:2, f:2>. The minimum-support is 3. So, the Large -1 sequences are (a:4, b:4, c:4) and the transformed database will look like Table 2.6-1.

A WAP tree can be constructed from Table 2.6-1. Only the frequent subsequences data will be the input of WAP tree. The others will be discarded. The frequent subsequences are filtered from web access sequences by predefined support threshold. The construction of WAP tree is similar to FP tree. It starts with a virtual root and all of

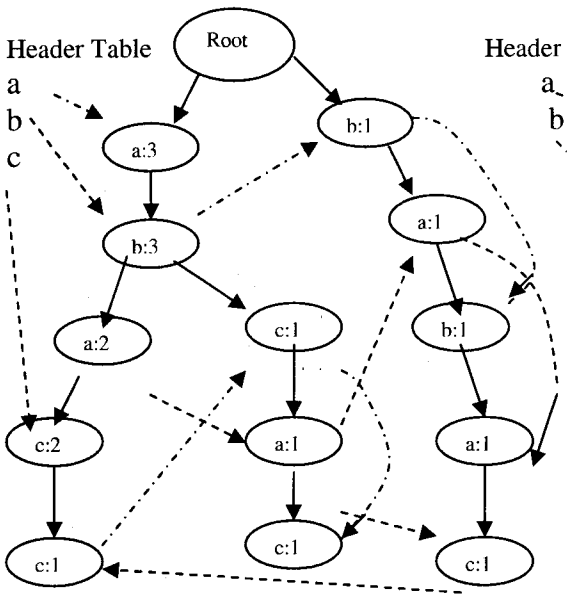

Figure 2.6- 1: WAP tree

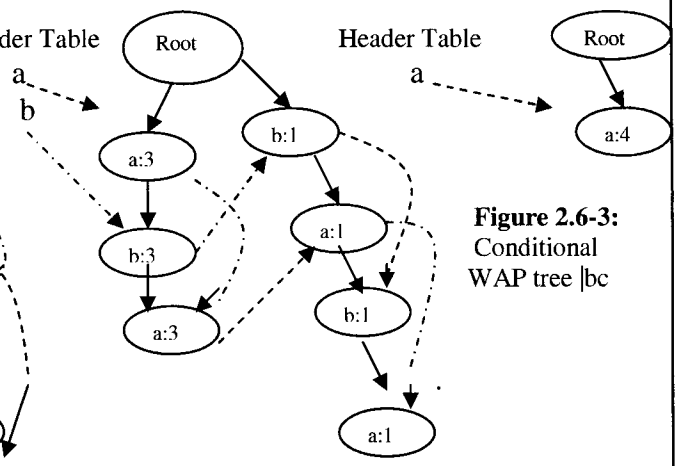

Figure 2.6-3: Conditional WAP tree |bc

Figure 2.6-2: Conditional WAP tree | c

the frequent subsequences go under the Root as children. For example, if we insert abac sequence (column 3, Table 2.6-1) , we will insert "a" as first node under the Root and the count for "a" will be 1, then (b: 1), then (a: 1) and (c: 1). Next, for the sequence abcac, we will start with node "a". Node "a" is already created under Root on the tree and first sequence "a" of the sequence (abcac) is matching with the existing node, we increment the count of "a" by 1 (total 2 at this point). We repeat same task for next sequence "b" in (abcac), we increment the count of node "b" by 1. But we need to create a new branch for node (c :1) under (b:2) because it is not matching with the previous order. Node (a :1)

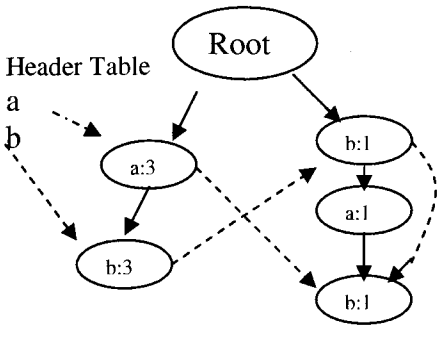

Figure 2.6-4:Conditional WAP tree | ac

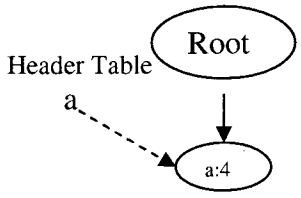

Figure 2.6-5:Conditional WAP tree |bac

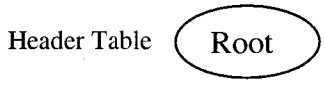

Figure 2.6-6: Conditional WAP tree |aac

and (c : 1) will fall under new "c" node. For the sequence babac, a new node (b:1) will be created under Root and the rest of the sequence abac will go under node (b:1) accordingly. The final tree looks like Figure 2.6-1. So far we have completed step 1 and

2. We calculated frequent subsequences and constructed a WAP tree using frequent subsequences. Below is the discussion of how to mine the tree based on its conditional search recursively.

The process starts with the lowest entry 'c' from the header table and mine the tree (Figure 2.6-1). We first find all sequences that have last event c. It is called the conditional sequence base of c. We find the following sequences: aba : 2; ab : 1; abca : 1; ab : -1; baba:1; abac : 1; aba : -1. Here, sequences ab and aba have count –1 because aba is the prefix sequence of conditional sequence abac and ab is the prefix sequence of conditional sequence abca. Therefore, we deduce the count of this subsequence. The count we need to deduct from the subsequences is same as the count we add of super-sequence. After deducting, new counts will be: a:4, b:4, c:2. The minimum-support is set to 3. So, 'c' will be deleted from the sequences. The rest of the sequences are: aba: 2; ab : 1; aba : 1; ab : -1; baba:1; aba : 1; aba : -1. We construct c conditional WAP tree or WAP-tree|c using above sequences (Figure 2.6-2) and obtain pattern "c".

In this intermediate tree (WAP-tree|c), prefix sequences based on b (lowest element in the header of the intermediate tree) are: a:3; ba:1. The counts are a:4, b:1. We delete b and construct WAP-tree|bc (Figure 2.6-3) and obtain pattern "bc". Now only one branch is left in Figure 2.6-3, we obtain pattern "abc". Here, the mining process stops because only Root is left. Thus, the mining process goes back to WAP-tree|c to count prefix sequences based on a from header table. They are ab:3, b:1, bab:1, b:-1. Here, the counts are a:4, b:4. WAP-tree|ac (Figure 2.6-4) is constructed using above sequences (ab:3, bab:1) and obtain frequent sequence "ac".

We use the previous procedure, prefix sequences based on b on WAP-tree|ac are a:3, ba:1. The counts are a:4, b:1. We construct WAP-tree|bac (Figure 2.6-5) and obtain "bac". Now, only one branch is left, we obtain pattern "abac", it stops sub mining process. The mining process goes back to WAP-tree|ac, prefix sequence based on a is b:1. We delete b because it does not have minimum support. We construct a WAP-tree|aac (Figure 2.6-6), now only root is left, the output is "aac". Here stops the sub mining process.

The mining process goes back to WAP-tree|ac and finds nothing left for next sub mining. The process goes back to WAP-tree|c and finds that c conditional mining have

been completed {c, bc, abc, ac, bac, abac, aac}. It goes back to main WAP-tree and constructs another conditional WAP-tree|b, the process goes through the sub mining process and completes mining. The obtained result is {b, ab} from WAP-tree|b.

The process goes back to the main WAP-tree and construct next conditional tree (WAP-tree|a). It goes through sub mining process and obtains frequent sequences {a, ba, aba, aa} from WAP-tree|a. Then, it goes back to main WAP-tree and finds that the job is completed. The final pattern is {c, bc, abc, ac, bac, abac, aac, b, ab, a, ba, aba, aa}.

2.7 PLWAP TREE

Pre-ordered Linked WAP-tree was proposed by Ezeife and Lu in [Ezeife & Lu2005][Lu & Ezeife2003]. The basic idea behind the PLWAP tree algorithm is to add pre-ordered link on a WAP-tree. In that way, several intermediate WAP-tree constructions can be avoided and reduce memory load that usually happens during the WAP-tree mining. We discuss the PLWAP-tree in details next.

Step 1: First PLWAP scans DB (Table 2.6-1, we use same table as WAP-tree for our example) to generate all Candidate 1-sequences. As per Table 2.6-1, candidate 1-sequences are (a:4, b:4, c:4, d:1, e:2, f:2). The minimum-support is set to 3. The large 1-sequences are (a:4, b:4, c:4).

Step 2: Second PLWAP scans DB for another time to discover frequent subsequences of each transaction (column 3 of Table 2.6-1). In the mean time, PLWAP tree is constructed in Figure 2.7-1 based on WAP-tree algorithm. In addition, a position code is added to each node to indicate the position of the nodes in the PLWAP- tree.

The rules for defining position for each node are if n is a node of the WAP tree and if node n is the root, then n has null position code. If n is the left most child of a parent node, then it obtains position code 1 that is equal to appending "1" to the position code of its parent. The position code of any other node is same as appending "0" to the position code of the node's closest left sibling. For example, if we apply frequent sequences (a b a c) from Table 2.6-1 on PLWAP tree, then 'a' will be the child of ROOT and the value of 'a' is (a: 1 : 1), here a's initial count is 1 and the position code 1 is obtained because 'a' is the left most child of root (Figure 2.7-1). Then, create node (b: 1: 11) as a child of 'a'.

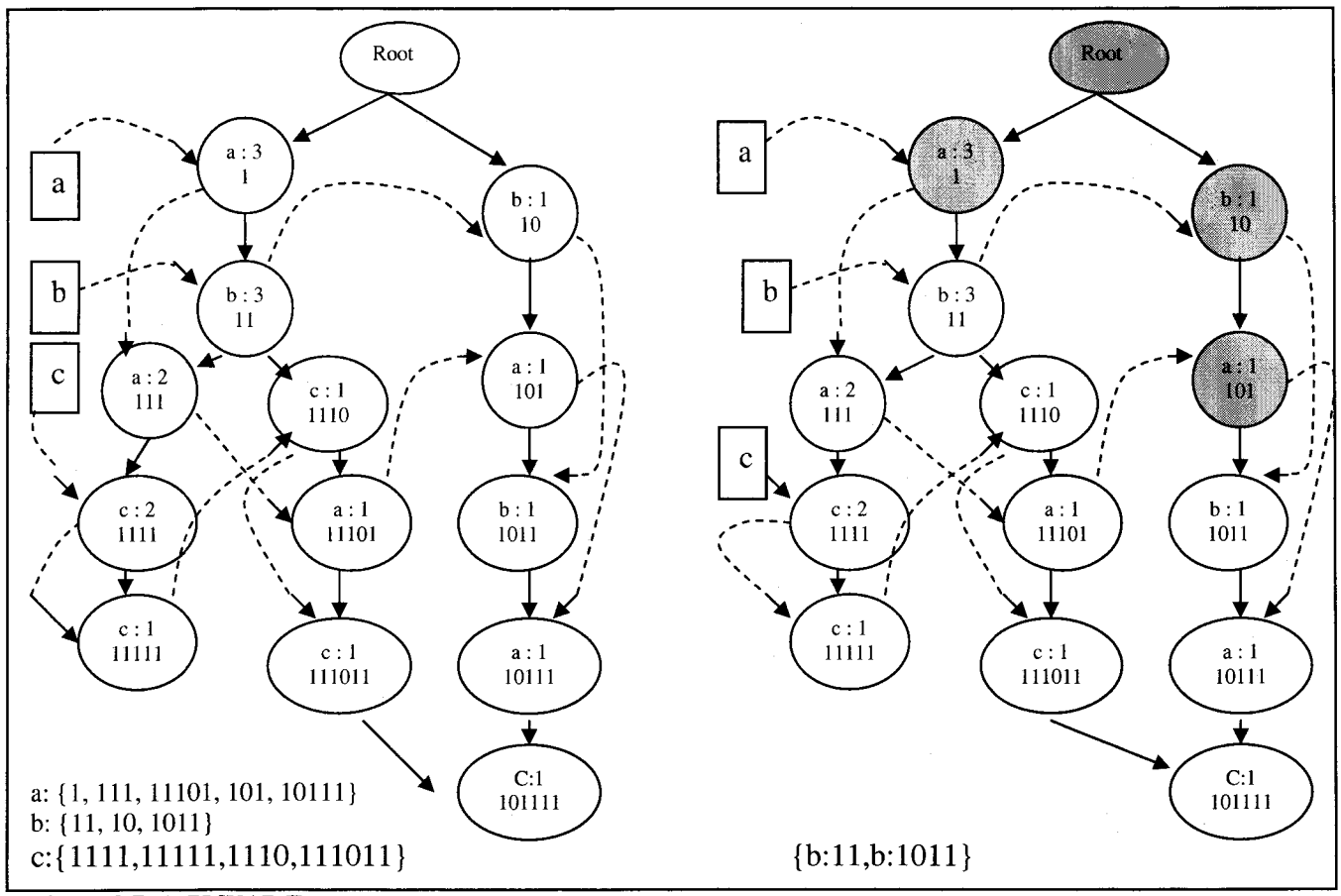

Figure 2.7- 1: PLWAP-Tree

Figure 2.7- 2: a | suffix tree

Here, the initial count is 1 and position code is 11. 'b' is the left most child of 'a', therefore, 'b' has obtained 11. It obtained position code 1 from parent and another 1 for being parent's left most children. Then create (a: 1: 111) as the child of 'b'. Similarly, we insert (c: 1: 1111) as the left most child of node (a: 1: 111).

Next we can move on to second frequent subsequences (a b c a c) to insert on PLWAP tree. Since node (a) and (b) are existing, therefore, we just increment the count of them by 1 as (a: 2: 1) and (b: 2: 11) respectively. Then, we insert sequence 'c' on the tree. Sequence 'c' will create new branch under (b: 2: 11) as per rules and it will obtain (c: 1: 1110). Here the count is 1 because it is inserted for the first time on the tree but it obtained position code 1110 because it is the sibling of node (a: 1: 111). Sibling 'a' has position code 111. As per rule, additional 0 will add at the end for the closest leftmost sibling. Next insert (a:1:11101) and (c:111011) under (c: 1: 1110) respectively. Similarly, we insert (b a b a) and (a b a c c) sequences and build the tree (Figure 2.7-1).

Once the construction of PLWAP is completed, a pre-order traversal mechanism is used to add a pre-order linkage on the tree. In pre-order traversal mechanism, the mining process visit root first, visit left subtree and visit right subtree. Starting from the root, travel the left most node first (Figure 2.7-1) and find (a : 3 :1) that is a root of its suffix tree. So, linkage with same event label of 'a' is created. Continues to travel and find (b:3:11) and create the linkage too. After that, travel to 'b's left child and find (a : 2: 111). A's linkage is already created that points to (a : 3: 1). So, create a link from (a: 3: 1) to (a : 2 : 111) and also create a linkage from linkage table to (a : 2 : 111). It continues the

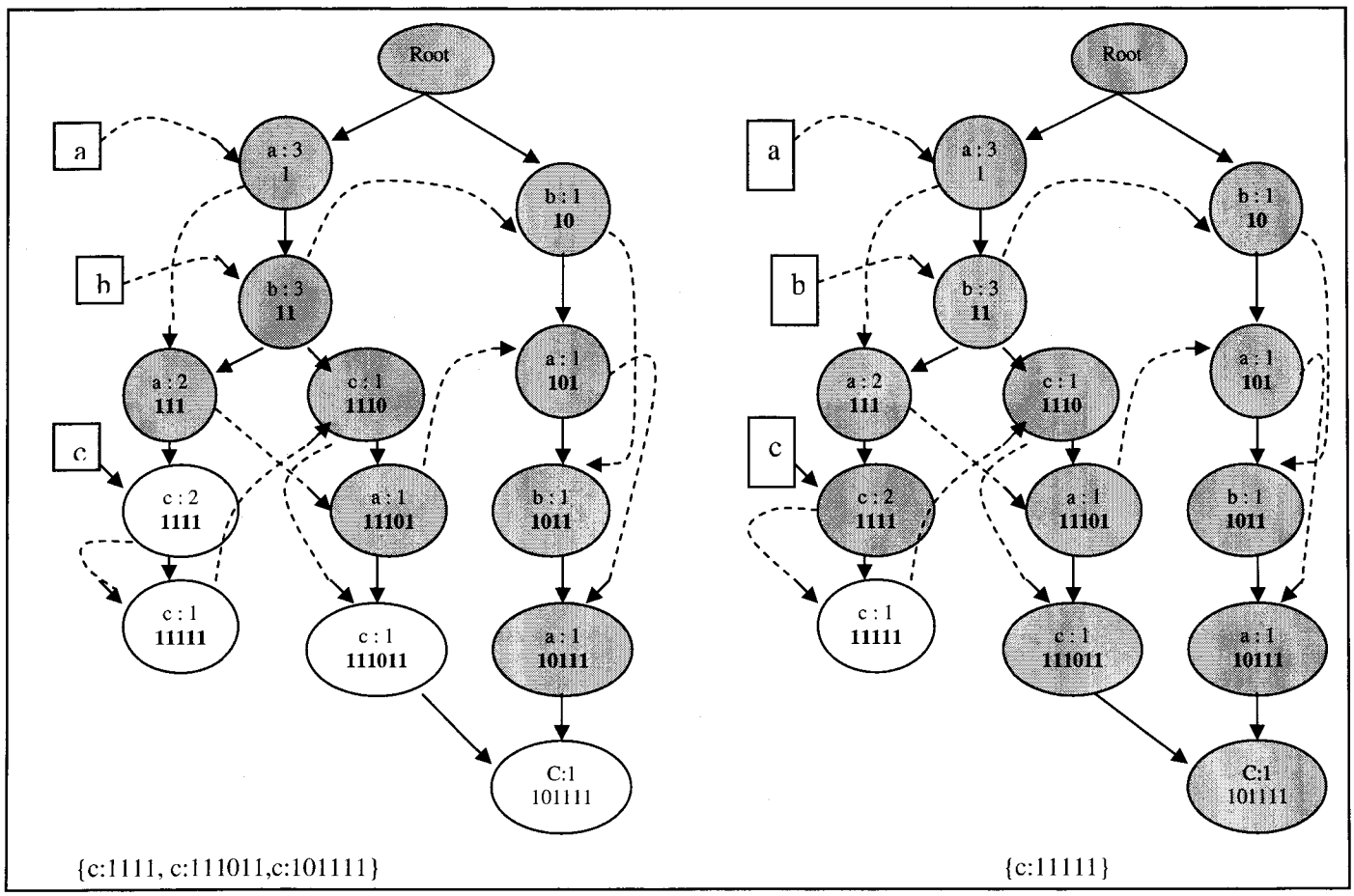

Figure 2.7- 3: aa | suffix tree

Figure 2.7- 4: aac | suffix tree

travel, at node (c:2:1111) and creates the linkage for the table. It continues travel and creates a link from (c:2:1111) to (c:2:11111). (c:2:1111) is located at the leaf node because there is nothing after this node. So, it looks for siblings and finds nothing. It goes back to its parent (c:2:1111), looks for siblings, finds nothing. It goes back to

(c:2:1111)'s parent (a:2:111) and finds its sibling is (c:1:1110). It creates a link from (c:1:11111) to (c:1:1110) and repeat this process to create linkage on the entire PLWAP-tree.

Step 3: In this stage, PLWAP starts finding frequent sequences with the frequent 1-sequence in the set of events {a b c}. It starts following the header linkage of the first element 'a'. Since the two suffix trees of PLWAP tree are rooted at a:3:1 and b:1:10 respectively (Figure 2.7-2) have the first occurrence of 'a' node with support of 4 (sum of a:3:1 and a:1:101 counts). Minimum support is set to 3, therefore 'a' is considered as frequent 1-sequence and it will be listed in frequent sequence list. The PLWAP process looks for 2-seqeunces that start with event 'a'. Using the position codes, the process continues to mine all frequent events in the suffix tree of a:3:1 and a:1:101 that are rooted at b:3:11 and b:1:1011 respectively (Figure 2.7-1). From Figure 2.7-1, we keep finding the first occurrence of 'a' for each suffix tree. We find (a:2:111, a:1:11101 and a:1:10111) that give us 'a' as a sequence event (Figure 2.7-3). Now, 'a' is added to the last list of frequent sequence 'a' to form the new frequent sequence 'aa'. The process continues to mine the conditional PLWAP tree in Figure 2.7-3. The suffix trees of these nodes that are rooted at c:2:1111, c:1:111011 and c:1:101111 give another 'c' frequent to obtain the sequence 'aac'. The last suffix tree Figure 2.7-4 is no longer frequent and form 'aac'. It terminates this leg of recursive search.

PLWAP searches for other frequent events by backtracking in the order of previous conditional suffix tree mined. Since no more frequent events are found in the conditional PLWAP tree in Figure 2.7-3, it further backtracks to Figure 2.7-1, b:3:11, b:1:1011 and yields a frequent event for 'b' to give the next frequent sequence as 'ab'. The algorithm keeps moving and finds next frequent sequences as aba, abac, abc. The algorithm does not find any more frequent sequences, it terminates this leg of recursive search. The algorithm backtracks to Figure 2.7-3, c:2:1111, c:1:111011 and c:1:101111 and yields a frequent event for 'c' to give the next frequent sequence as 'ac'. So far, we found all frequent sequences starting with 'a'. Similarly, we find frequent sequences starting with 'b' {b, ba, bac, bc} and 'c' {c}. PLWAP gives the final mining result as {a, aa, aac, ab, aba, abac, abc, ac, b, ba, bac, bc, c}and that is same as WAP-tree [Pei et al.2000] result.

2.8 Revised-PL4UP

Ezeife and Chen [Ezeife & Chen2004a] introduced Re-PL4UP for mining frequent sequential patterns incrementally using PLWAP tree structure. In the paper, old database denoted as DB, new database as db, updated database as U = (DB + db). F and S represent previous large items and previous small items in the DB respectively; F´ and S´ represent updated large items and updated small items in the updated database. Each event in U belongs to one of the six categories (or cases) below:

TID	Web Access Sequence	Frequent Subseq. with s = 50%
100	A b d a c	a b a c
200	A e b c a c e	a b c a c
300	B a b a	b a b a
400	A f b a c f c	a b a c c
500	A b e g f h	a b

Table 2.8- 1: Original Database DB

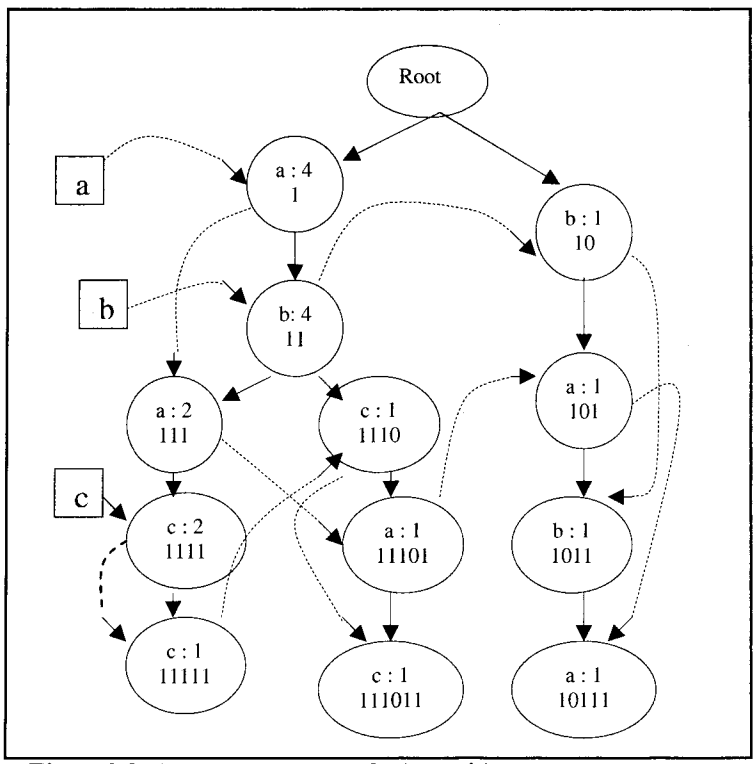

Figure2.8- 1: Re-PLWAP tree for DB with support s

Category 1: The items that were large in 'DB' are still large in 'U' (F→F′).

Category 2: The items that were large in 'DB' became small in 'U' (F→S′).

Category 3: The items that were small in 'DB' became large in 'U' (S→F′).

Category 4: The items that were small in 'DB' are still small in 'U' (S→S′).

Category 5: New and frequent in 'U' (Ø→F′).

Category 6: New and small in 'U' (Ø→S′).

Re-PL4UP algorithm scans only the incremental part of database (db), then, it uses frequent items that are generated from db to update the old pattern. The significant changes will occur on old PLWAP tree if cases 2 (F→S′) and cases 3 (S→F′) happen.

For case 2, old frequent items will be deleted and for case 3, previous small items will be inserted on the tree. Re-PL4UP takes the advantages of the position code of the PLWAP tree. It stores the list of all small items' position code. During the update process, the unique codes are used to re-insert the previous small items in the proper positions in the tree without rescanning the old database. The Re-PL4WAP algorithm is discussed next with an example. Assume original or old database DB (Table 2.8-1) with the set of items I = {a, b, c, d, e, f, g, h} and minimum support is set to 50%. The first process is to build initial Re-PL4WAP^{DB} tree using the frequent subsequences (support is $\geq$ 50%). The construction of Re-PL4UP^{DB} tree is the same as PLWAP tree construction. To build the tree, we scan the DB and generate candidate 1-itemsets as C_1 = {a:5, b:5, c:3, d:1, e:2, f:2, g;1, h:1}. Here, F_1 = {a:5, b:5, c:3} and S_1 = {e:2, f:2, d:1, g:1, h:1}. Then, scan the DB second time to generate frequent subsequences (column 2 in Table 2.8-1). Insert each frequent subsequence in the tree, count, position code and construct the tree (Figure 2.8-1). While inserting frequent items in the sequence, the algorithm checks the original transaction to mark location of small items in the transaction. For example, small event 'd' in the first transaction 'abdac' would have the position (d:1:111) in the created branch if 'd' were frequent. This will write this position code in the small item code profile for item d as S-code^d={111}. The complete small item code profiles for all small items will be S-Code^d={111} or {3}, S-code^e={110, 11011111, 1110} or {6, 223, 14}, S-code^f={1100, 11001111, 111011} or {12, 207, 59}, S-code^g={11101} or

{29}, S-code^{h}={1110111} or {119}. After building the tree a pre-order traversal mechanism (visit root, visit left sub-tree, visit right sub-tree) is used to add a pre-order linkage on the tree for all frequent 1-items, F_1 = {a, b, c}. The broken lines are used to show the linkage of the frequent items in Figure 2.8- 1.

TID	Web Access Sequence	Frequent Subseq. with s = 50%	Frequent subseq. with t=0.8s
700	B a h e f g	b a h e f g	b a h e f g
800	A e g f h	a e g f h	a e g f h

Table 2.8- 2: Inserted Database db

Once the linkage is created, Re-PL4UP^{DB} begins mining process. By applying the same mining process as PLWAP algorithm and support threshold 50% (3 transactions), we find frequent patterns on old database FP^{DB} = {a:5, aa:4, aac:3, ab:4, ac:3, abc:3, abcc:2, ac:3, acc:2, b:4, ba:4, bac:3, bc:3, bcc:2, c:3, cc:2}

Now by assuming that original database DB (Table 2.8-1) is updated with inserted db (records are in Table 2.8-2). 1). The objective of Re-PL4UP is to reuse previous FP^{DB} as much as possible with the new changes to the database. Re-PL4UP algorithm works as follows:

1). Update all intermediate candidate lists: $C'_1 = C_1 \cup C_1^{db}$. C_1 = {a:5, b:5, c:3, d:1, e:2, f:2, g;1, h:1}. C_1^{db} = {a:2, b:1, e:2, f:2, g:2, h:2}, thus C'_1 = {a:7, b:6, c:3, d:1, e:4, f:4, g:3, h:3}. F'_1 ={a:7, b:6, e:4, f:4} and $F_1^{db} = C_1^{db} \cap F'_1$= {a:2, b:1, e:2, f:2}. S´={c:3, d:1, g:3, h:3}. $S_1^{db}=C_1^{db} \cap S'_1$= {g:2}.

2) Classify items in U (updated database), into one of the defined six categories as 1) $F_1 \cap F'_1$ = {a, b}, 2) $F_1 \cap S'_1$= {c}, 3) $S_1 \cap F'_1$= {e, f}, 4) $S_1 \cap S'_1$ = {d, g, h}, 5) F'_1-F_1= Ø, 6) S'_1-S_1= Ø.

3) Modify the old Re-PL4UP^{DB} tree by deleting the items for category 2 (F→S´) = {c} and insert items into the tree for category 3 (S→F´) = {e, f} using the small code profile. The position of small item in the current tree will be determined by any matching prefix of its binary position code that is listed in the profile. For example, if an item has the position profile code is 111011 and we find a node position in the current tree for the prefix 1110, then, we insert small to large item there. The new code profile of the item is

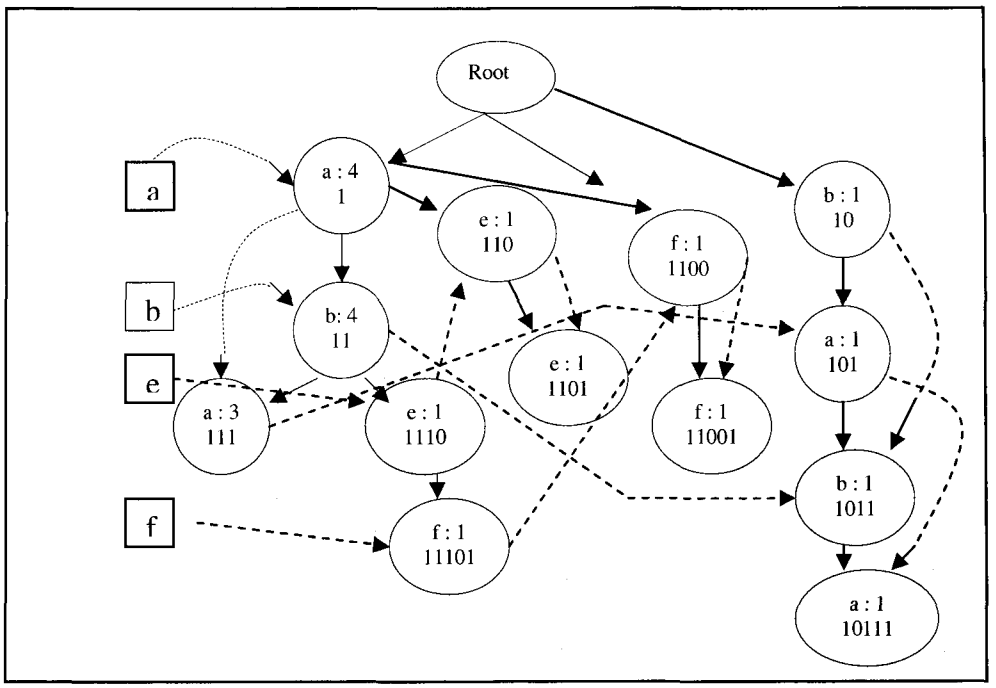

Figure2.8- 2 : Modified Revised PL4WAP tree

the physical code in the tree. Then, the small code profile is updated. Thus, after tree modification, the updated codes for the small-to-large items {e, f} will be S-code^{e}={110, 1101, 1110} and S-code^{f} = {1100, 11001, 11101}. The last task will be to re-construct the frequent header linkage.

4) Mine only the modified branches of the Re-PL4UP tree. We only modified the left branch of the tree. We obtain frequent sequences called Re-FP^{DB} as {aee:1, abef:1, aff:1, ae:2, af:2, be:1, bf:1, bef:1, ef:1, ee:1, ff:1, e:2, f:2}.

5) Construct Re-PL4UP^{db} using only changes to the database (db). The construction of the tree is based on $F_1^{db}=C_1^{db} \cap F'_1$ = {a, b, e, f}. Mine Re-PL4UP^{db} to obtain FP^{db} patterns. We find frequent patterns FP^{db}={b:1, a:2, h:2, e:2, f:2, g:2, ba:1, be:1, bf;1, aef:2, bef:1, ae:2, af:2, ef:2, bf:1}.

6) Combine old frequent pattern FP^{DB}, pattern of the modified branches of the old tree Re-FP^{DB} and the pattern of the changes to the database FP^{db}. Keep those items that have new support (s´) greater or equal to 4. Thus, FP´ = FP^{DB} ∩ Re-FP^{DB} ∩ FP^{db} ≥ s´. FP´={a:7, aa:4, ab:5, aba:4, b:6, ba:5, ae:4, af:4, e:4, f:4}.

7) Finally, the frequent sequences in the incremental part (db) that are {baef, aef} are inserted into the main Re-PL4UP tree to keep it up-to-date. Small code profiles are updated by adding new small items and the codes of the deleted nodes in the tree.

2.9 Lossy Counting Algorithm

Lossy counting algorithm [Manku & Motwani2002] is very prominent frequency counting algorithm in data streams. The proposed algorithm finds frequent items in data streams when a maximum acceptable error $\varepsilon \in (0, 1)$ as well as maximum support $s \in (0, 1)$ are given where $\varepsilon << s$. Let N denote the current length of a stream. The authors used a landmark model that counts frequent items in data streams by assuming that patterns are measured from the start of the stream up to current moment. The process waits until the fixed sized buffer gets filled with the frequent data stream. Once the buffer is filled, the incoming stream is conceptually divided into buckets. The buckets are labeled with bucket ids, starting from 1. The width of each bucket $w = [1/\varepsilon]$ transactions. Then, run Lossy Counting-algorithm on buckets one after another to find frequent items.

The algorithm uses support s and error ε to determine frequent items. If user specified minimum support s = .1 (10%), then the method suggests to keep ε=.01 (1%) that is one tenth of s. Thus, size of the bucket or window will be 1/.01=100 single item transactions. After counting the frequency of first window or bucket, the ε-frequent items (items that have frequency more than ε) are stored into a data structure *D*. The items are stored in *D* in the form of (e,f,Δ) where e is an element, f is the count of e and Δ is the maximum possible error count of the items.

Algorithm: Initially, *D* is empty. Whenever a new element e arrives, it checks *D* whether e already in *D*. If e found, it increments e's frequency f by 1. Otherwise, it creates a new entry of the form $(e, 1, b_{current} - 1)$.

At the boundary of the bucket or window, it performs two types of pruning.

1) It decrements all elements by 1 % (ε's value) in *D*.

2) It deletes items from the structure *D* if $f + \Delta \leq b_{current}$. Here $b_{current}$ is current bucket.

Whenever, a user requests a list of items that have support s, the algorithm generates output for items that have frequency $f > (s - \varepsilon) N$. Here, N is current length of a stream. We will run their algorithm with our example for better understanding.

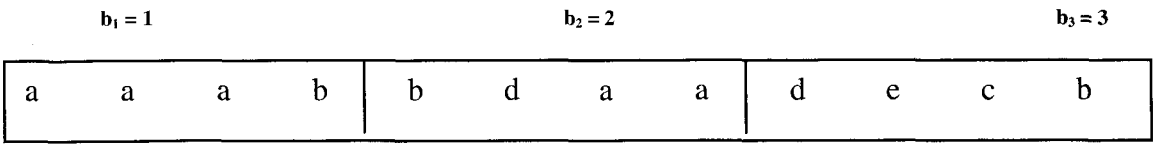

Figure 2.9- 1: Data Stream in a buffer

Assume Figure 2.9-1 is a buffer that is filled by data stream. Data stream is nothing but incoming items or elements from a source (web, stream generator program). As per algorithm, we divide the data of buffer into several windows or buckets. We have three buckets (b_1, b_2, b_3, Figure 2.9-1). Minimum support is s = .50 (50%) and ε =.25 (25%, we took higher percentage of ε than recommended to reduce calculation). We determine the size of the bucket w = [1/.25 = 4] single transactions. We labeled the bucket as $b_1 = 1$, $b_2 = 2$ and $b_3 = 3$ (Figure 2.9-1).

e	f	Δ

Figure 2.9- 2: Empty structure *D*

Initially, we have an empty data structure *D* (Figure 2.9-2). As per algorithm, we start with first bucket or b_1. We take first element "a" from b_1 (left to right) and insert into *D as* (e f $b_{current}$-1) form = (e f b_1-1) = (a 1 0). As we know, $b_{current} = b_1 = 1$. Then, we insert next element "a", as "a" already in *D,*

e	f	Δ
a	3	0
b	1	0

Figure 2.9- 3: Data structure *D*

we increment its frequency in *D*. Similarly, we increment third element "a" in *D*. Now, when we insert "b" from b_1 into *D*, we insert in the form (b 1 0) for the first time. After insertion of b_1, *D* looks like Figure 2.9-3.

After finishing counting one bucket, the algorithm marks this point as boundary. At the boundary, the algorithm performs two types of pruning before start processing next batch (in our case b_2).

Pruning 1: it decrements all entries in *D* by the threshold of ε (25%) (or 1 out of 4 single transactions).

Pruning 2: the algorithm performs pruning if any items have frequency $f + \Delta \leq b_{current}$. Here, $b_{current}$ is current bucket that is b_1= 1. In other words, an item (items) will be deleted in *D* if its frequency $f + \Delta$ is less than or equal to $b_{current} = 1$. We do not find any elements to delete them in *D* based on pruning 2 after

performing pruning 1 on *D* (Figure 2.9-3).

After processing b_1 and performing two types of pruning methods at the boundary, it starts processing next bucket b_2. Now, b_2 becomes current bucket or $b_{current}$ and it is already labeled 2. We start with the first element "b" (reading elements from left to right in Figure 2.9-1 in b_2). Before we insert "b" into *D*, we lookup *D* and check whether element "b" already exists in *D*. We find nothing. We insert "b" into *D* in the data structure form (e f Δ). Thus, we insert (e f $b_{current}$ -1) = (e f 2-1) = (b 1 1). Similarly, we insert other elements (d e) into *D* but algorithm treats last element "a" differently because "a" already exist in *D*. In that case, the frequency of "a" gets incremented by 1 but the value of Δ remain same for "a" in *D*. After inserting all elements of b_2 in *D*, it looks like Figure 2.9-5.

e	f	Δ
a	2	0

Figure 2.9- 4: *D* after b_1 boundary

e	f	Δ
a	4	0
b	1	1
d	1	1
e	1	1

Figure 2.9- 5: Updated *D*

Before it processing b_3, it performs two types of pruning methods that occurs at the boundary. It decrements all entries in *D* by 25% (or 1 out of 4 single transactions) because ε =.25. The algorithm performs second pruning that is if any items' frequency $f + \Delta \leq b_{current}$. $b_{current}$ is current bucket that is b_2 and we labeled it 2. So, an item (items) will be deleted in *D* if its frequency $f + \Delta$ is less than or equal to $b_{current} = 2$. After performing both pruning, D looks like Figure 2.9-6.

e	f	Δ
a	3	0

Figure 2.9- 6: *D after processing* b_2

Similarly b3,...,bn can be processed.

Now, if a user requests a list of items that have support s, the algorithm generates output those items have frequency $f > (s-\varepsilon) N$ or (.50-.25) * 8 = 2 = {a}. At this point, the current length of a stream N = 8 because we have completely processed b_1 and b_2 and each of them has 4 single transactions.

Similarly, lossy counting algorithm can be used to count when the buffer will be loaded by frequent patterns instead of single items.

Discussion: lossy counting algorithm uses decrement mechanism from the beginning. At the boundary, it decrements all entries by the percentage of maximum error tolerance threshold or value of ε. Small items with counts less than or equal to ε get eliminated by this decrement method from the beginning. As a matter of fact, a huge number of small items required to be processed in the process of finding large items. It is efficient to keep decrementing items in the data structure rather than incrementing items.

The items with higher frequencies would remain in D after all of the decrements at each boundary. However, all the items would not have true counts at the end. For example, we have processed up to b_2 bucket and the true count of element "a" is 5. After two times decrements, its current count or frequency is 3. The formula lowers the support threshold as much as it decremented during the process. In other words, the formula is lowered from .50 to (.50-.25=.25). By using the formula, we can identify large items. As a matter of fact, at the end, we can find large elements or items using lossy counting algorithm. In many cases, we might do not need true frequency of items or elements.

The algorithm also prunes individual items if they go below a threshold. However, if those trimmed items come back later, it would compensate an approximate loss frequency with the items that were pruned earlier.

2.10 FP-Stream

FP-Stream algorithm [Giannella et al.2003] is introduced to mine frequent patterns in data stream. Previously, landmark model [Manku & Motwani2002] was introduced that mines frequent patterns in data stream by assuming that patterns are measured from the start of the stream up to current moment. The authors of FP-Stream [Giannella et al.2003] extended their framework to answer time-sensitive queries over data stream. For example, if itemset "ab" becomes large at the end after several incremental processes, as per theory, it is possible to say what time period "ab" was large and what time period "ab" was small even though its end result is large. In order to facilitate time-sensitive queries, FP-Stream uses a pattern-tree, FP-tree and a fixed-time window frame with FP-Stream structure.

FP-tree: This tree was introduced in [Han et al.2004] [Han, Pie & Yin2000] earlier to mine frequent patterns using FP-Growth [Han et al.2004] [Han, Pie & Yin2000] algorithm.

Pattern-tree: It stores frequent patterns instead of data stream and each node of this tree maintains a time-window table. For example, we can tell what were the frequencies of the pattern "ac" in different time periods (t_o, t_1, …, t_3) from the Figure 2.10-4.

Tilted Time Window: The design of the tilted-time window [Chen et al.2002] is based on the short term period facts. There are two types of tilted-time window that can be used to provide time-sensitive queries: 1) Natural Tilted-Time Window and 2) Logarithmic Tilted-Time Window.

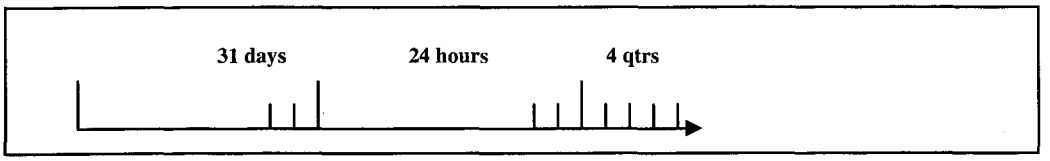

Figure 2.10- 1: Natural Tilted-Time Window Frame

1) Natural tilted-time window frame (Figure 2.10-1) for holding one month's data which has 4+24+31=59 units. First four units are 15 minutes each, next 24 units are one hour each and the last 31 units are 1 day each. This window can hold up to 31 days data. When first four units are filled with data, they merge together to form an hour unit, similarly, 24 hours units form a day unit. Based on this model, it is possible to compute in the last hour with the precision of quarter of an hour, the last day with the precision of hour, …., until the whole month.

2) Logarithmic tilted-time frame can be constructed based on logarithmic time scale (Figure 2.10-2, with ratio 2). Here, we have $[Log_2(n)]+1$ frequencies. As per this model, with one year of data and the smallest precision at quarter, we need $\log_2(365 \times 24 \times 4)+1 \approx 17$ units instead of $366 \times 24 \times 4 = 35{,}136$ units.

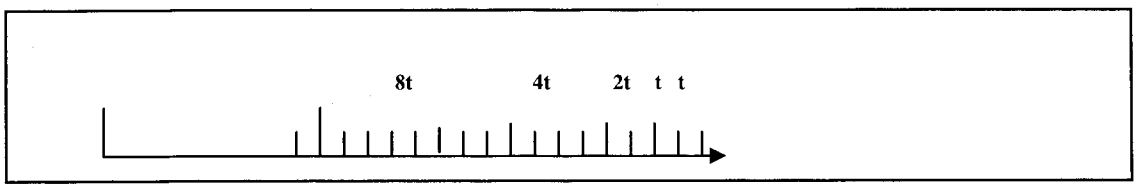

Figure 2.10- 2: Tilted-Time Window Frame with Logarithmic Partition

Algorithm:

We will go through a detailed discussion of the algorithm in this section. Let the minimum support be σ, an error rate ε (or maximum support error), an FP-Stream structure, incoming batch B_i are arriving one after the other, and transactions t. The process waits about 15 minutes to form a batch. Once the time is expired, (say the first batch B_1), it computes all of the items frequencies and stores them in the main memory in a descending order in a data structure named f_list. The ordering remains fixed for all remaining batches. Then, it constructs an FP-tree, pruning all items with frequency less than $\varepsilon|B_1|$. The algorithm finds ε-frequent itemsets (itemsets with frequency more than ε) from the FP-tree. The ε-frequent itemsets are stored into pattern-tree. All remaining batches B_i where $i \geq 2$ are processed according to algorithm below.

Algorithm with an example and discussion:

1) Algorithm forms a batch B_1 of data stream after waiting 15 minutes.

2) The elements and their frequency are listed in a data structure in descending order named f_list.

3) Construct a FP-tree [Han et al.2004] [Han, Pie & Yin2000] by extracting each incoming transaction t from B_1 according to f_list and inserting into the FP-tree without pruning any items for the first time.

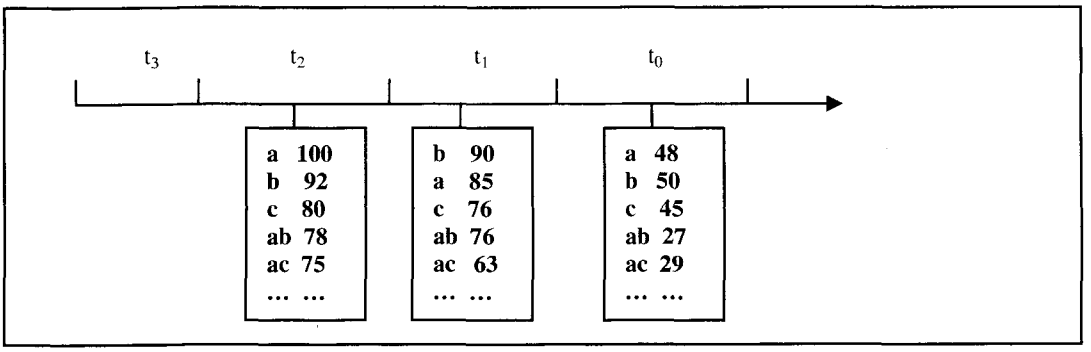

Figure 2.10- 3: Frequent Patterns for Tilted-Time Windows

4) Run FP-Growth algorithm [Han et al.2004] [Han, Pie & Yin2000], and mine ε-frequent itemsets (itemsets that have frequency more than ε).

5) Store the pattern into tilted time-window, Figure 2.10-3 shows that it found ε-frequent itemsets from batch B_3 that are stored under t_0 time slot (right to left). As we know that B_3 was holding data streams that were collected after 15 minutes of waiting period. Thus, we can say that t_0 has last 15 minutes ε-frequent patterns.

Update Method

1) Initialize the FP-tree to empty.

2) When all of the transactions in B_2 are accumulated, update as follows:

3) Update elements and their frequency in f_list in descending order with batch B_2's elements.

4) Construct a FP-tree [Han et al.2004] [Han, Pie & Yin2000] by extracting each incoming transaction t that are ε-frequent or frequency has more than ε from B_2 according to f_list and insert into the FP-tree.

5) Mine the FP-tree using FP-Growth algorithm [Han et al.2004][Han, Pie & Yin2000] and find all frequent patterns.

6) Update tilted time-window with frequent patterns that are found after mining B_2 as follows:

Check each itemsets I in tilted time window.

If yes,

i) Add each $f_I(B_2)$ to the tilted time window for I. Keep repeating this process. If I is not in the window and if $f_I(B_2) \geq \varepsilon |B_2|$, then insert I into the time window.

Once, we are done with update process in the tilted time-window, then updated frame slides to the next time slot and the new frame sits in the current time slot. Here, in Figure 2.10-3, time slot t_1 is holding two batches mined patterns (batch B_1 and B_2) and we know each batch was holding 15 minutes long data. Therefore, we know time slot t_1 is storing last 30 minutes mined pattern. Similarly, t_2 is holding last 45 minutes mined pattern (Figure 2.10-3). From Figure 2.10-3, we can say that pattern "ac" had frequency 75 at last 45 minutes, we can also say that pattern "ac" had frequency 63 in last 30 minutes and 29 in last 15 minutes. Now, we can easily calculate against predefined minimum user support and find out whether "ac" was frequent or infrequent at last 45, 30 or 15 minutes. The authors referred this kind of query as time sensitive queries. This tilted time window keeps short term period data because it maintains several time slots and the data of slots slides from right to left when new data arrive. Therefore, it requires a big storage.

To speed up searching and updating the tilted time window, the authors embedded this tilted time window in a tree called pattern tree. Figure 2.10-4 is a Pattern-tree. The construction of pattern-tree is similar to FP-tree [Han et al.2004] [Han, Pie & Yin2000] but pattern-tree stores frequent patterns instead of frequent elements. Each node has a label, a tilted time window table. This tilted time window table maintains frequency of frequent patterns and their time slots as well.

Similarly, batch B3,....Bn can be stored into Pattern-tree. The compact structure of the pattern-tree and embedded tilted time window are referred to as FP-Stream structure.

When a user wants to see the result, the pattern-tree is mined against the user defined minimum support threshold.

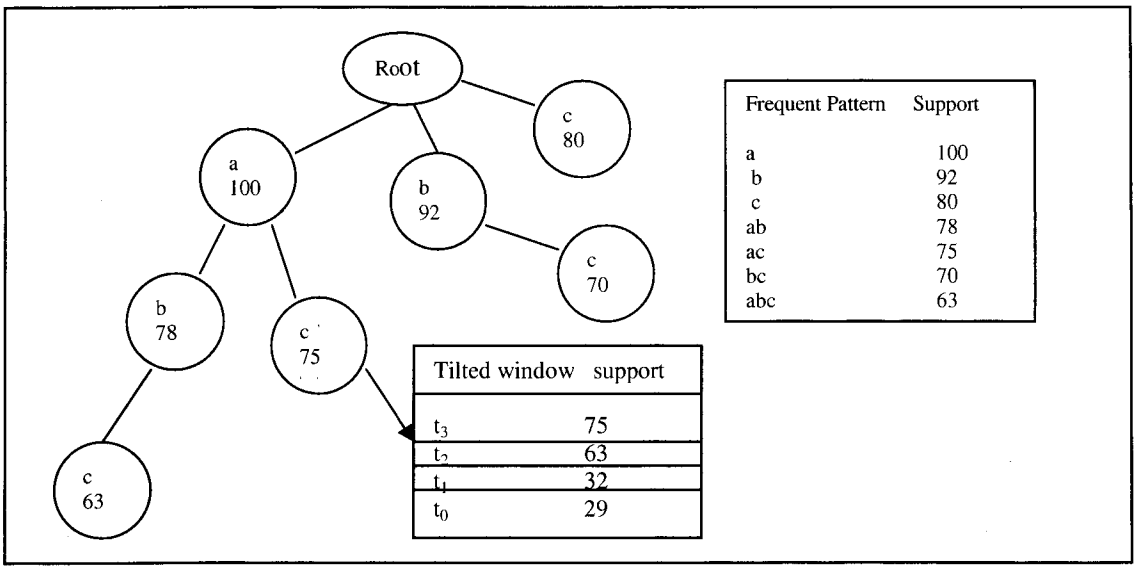

Figure 2.10- 4: Pattern-tree

Mine Pattern-tree: We read pattern tree from top to down. For example, from the root node, we visit left child first and that is "a". We find the frequency of 'a" is 100. We visit its left child "b" and find frequency 78. This 78 is the count or frequency of "ab" pattern (not the count of "b"). Similarly, we find the frequency of "abc" that is 63 from the tree. Once it reaches to the leaf node, it goes back to its parent and checks for its siblings to mine. As a matter of fact for our example, it finds nothing. It moves back to its parent and that is "a" and find sibling "c" is not mined yet. It finds frequency 75 for "ac' pattern. Similarly, all other nodes of the tree can be mined recursively.

2.11 FTP-DS Algorithm

Teng et al. proposed FTP-Algorithm [Teng, Chen & Yu2003] to mine frequent temporal patterns of data streams. FTP-DS scans online transaction flows and generate frequent patterns in real time. Sliding window model is used in this paper. Data expires after N time units after its arrival. Here, N is the user specified window size. As per FTP-DS rules, a temporal pattern is frequent if its frequency in the current window is no less than user defined support threshold.

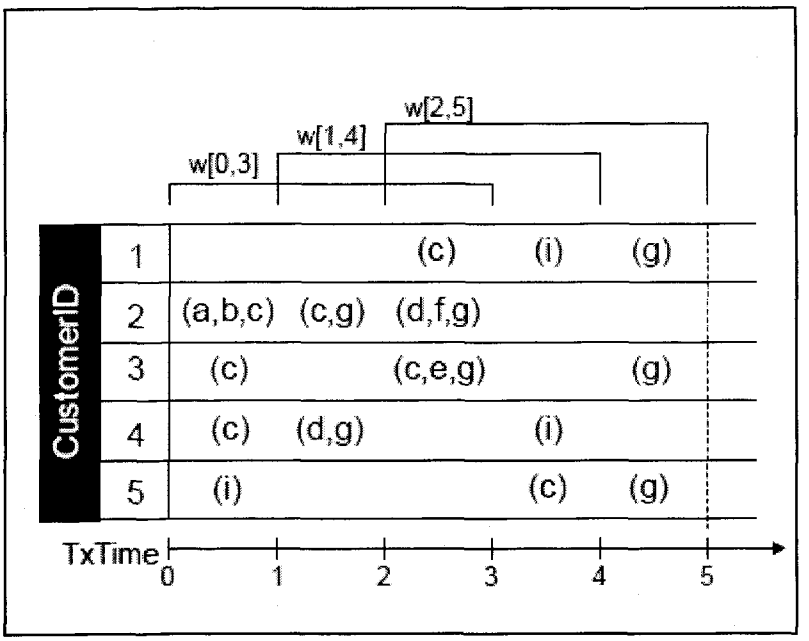

Figure 2.11- 1: An example of online transaction flow

The transaction flow of an application is shown in Figure 2.11-1 where items (a to g) are purchased by five customers. Third customer (Figure 2.11-1) bought item c during time t = [0,1], item c, e, and g during t = [2,3] and item g during t = [4,5]. In such a data stream environment it is very difficult to conduct frequent pattern identification due to limited time and space. In order to handle this kind of computation in data streams Teng et al. designed a sliding window model. Say, the window size is given and that is N =3, three sliding windows w[0,3], w[1,4] and w[2,5] are shown in Figure 2.11-1. Table 2.11-1 shows that the temporal frequent pattern result found against Figure 2.11-1 based on support 0.4. Table 2.11-1 (c) item d found frequent from Figure 2.11-1 because d

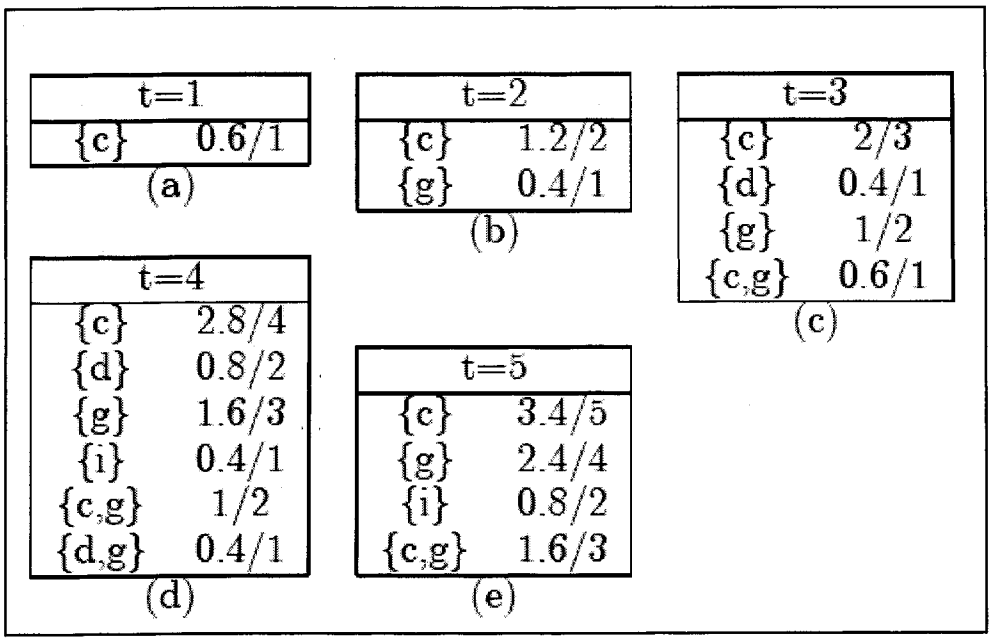

Table 2.11- 1: Generation of frequent temporal itemsets (MinSup=0.4)

occurred two times up to time point t = 3, (see TxTime in Figure 2.11-1, TxTime shows time point). As per rule accumulated support / number of transcations = 2 / 5 = .4. Thus, d became frequent from the time point t = 3. However, it could not maintain its minimum support .4 at the end. So, it discarded in Table 2.11-1(e). As time advances patterns those support fall below the minimum support threshold are removed from the records. At the end, only frequent patterns are recorded. In practice, if a pattern does not maintain a steady frequency above the threshold for a certain time can become frequent later. It would be appropriate to make the remove process delay for a pattern once it was recorded as frequent in the record. A linear estimate function $f_n = \alpha + \beta t$ is used to correspond with frequency variation of the temporal pattern. α and β are computed as $\infty = f - \beta t$ where $\beta = S_{tf} / S_{tt}$. Here, f stands for frequency and t for time. $S_{tf} = \sum t^2 - \{(\sum t)^2\} / n$ and $S_{tt} = \sum tf - \{(\sum t)(\sum f)\} / n$.

3 Mining Frequent Sequential Patterns in Data Stream

3.1 Introduction

A data stream is a continuous, unbounded, and high-speed flow of data items. To mine a data stream, we must switch from "one-time" traditional approach to a new approach that is able to mine continuous, high volume and open-ended data as they arrive. The previous techniques ([Agrawal & Srikanth1995][Srikanth & Agrawal1996][Hin2002][Ezeife & Lu2005][Lu & Ezeife2003][Ezeife & Chen2004a][Ezeife & Chen2004b]) could be modified with new requirements such as unbounded data with bounded memory, scan original data only once, online process, and deliver result on demand in order to use them in the stream environment. By considering stream environment requirements, we have developed a method that uses SSM-Algorithm (sequential stream mining algorithm) to collect data stream into a buffer from web applications, forms dynamic sized batches by taking data stream from the buffer and mines each batch to deliver results. A batch is a group or a number of customer access sequences. SSM-Algorithm maintains three data structures (d_list, PLWAP-tree, FSP-tree) in order to handle and generate result for click stream data. Click stream data can be generated from the clicks of users on the net. Each data structure has its own algorithm to update and retrieve data from the structures.

D_list is a hash chain based data structure that stores all incoming items' ID and their frequency if they are above a threshold. D_list is very efficient when there are huge numbers of items that are used at the e-commerce site. Brand new items get posted to the e-commerce site very often. It is very realistic that each e-commerce site introduces new items as soon as they get items from the vendors. We assume that we do not have any information on number of items and their IDs for our algorithm. In other words, our d_list is totally dynamic, it grows with incoming data stream.

PLWAP-tree [Ezeife & Lu2005] [Lu & Ezeife2003] gets constructed by selecting frequent sequences from batches. PLWAP-mining algorithm uses preordered linkage and position coding method to avoid costly reconstruction of intermediate trees to generate frequent sequential patterns unlike WAP-tree [Pei et al.2000].

The process of mining frequent sequences continues batch by batch. Once we find frequent patterns from first batch, we keep incrementing frequent patterns onto frequent sequential pattern-tree (FSP-tree) batch by batch. The construction process of FSP-tree is

similar to Pattern-tree that was introduced in [Giannella et al.2003] but the structure of FSP-tree is simpler that pattern-tree. We made FSP-tree simpler to do our tasks efficiently and quickly. FSP-tree stores frequent patterns instead of customer access sequences. Whenever we need to view the results, we are able to generate updated frequent patterns from the FSP-tree.

The proposed method is very memory efficient and ideal for click stream environment. As the mining process moves only forward and there is no way to go backward and rescan previous data stream, therefore we do not keep any items that have support less than maximum support error threshold in d_list structure where we store updated candidate 1-sequences. The items less than maximum support error threshold are very small and chances are very slim for them to become large items later. Therefore, we get rid of those small items from the beginning. However, there are some small items that are potentially large items (very close to large items) and have chances to be large later, we do not ignore them as we keep all of the items in the d_list that have support equal to or more than maximum support error.

Maximum support error is a predefined threshold that gives a tolerance of error in the result. It is standard now in stream mining to have support and error. It is obvious that if maximum support error is very small compared to user defined support, then the error in the result will be very nominal. By keeping all items that have support more than or equal to maximum support error in the d_list, we assure that our result does not cross error boundary.

3.2 Problem Domain

Assume that we are mining frequent sequential patterns for a busy e-commerce site XYZ. Our intension is to mine online weekly store flyer. Big corporations spend a lot of money on their flyers if they are in retail business. They want to make their flyers as attractive as possible in order to get attention of the customers and to increase their revenue. Our intention is to find frequent sequential patterns in click stream data from XYZ's website. The click stream data can be captured of the customers when they click on a particular product or products on the website. By analyzing click stream data, XYZ can predict the products that are being considered by the customers as future buying

products or the customers are interested in knowing more about. By identifying the frequent sequential products based on customers' interest, XYZ can increase their revenue by taking some of the following steps. For example:

1) XYZ can reorganize their site based on the result of the analysis such that frequent sequential products are placed on each other's site to make finding them easy.

2) XYZ can always remind customers about the future products when they buy products through their site based on the analysis. For instance, if a customer buys a printer, the site can give a pop up message to the customer that you will be needing ink cartridge soon because all of the printers come with a starter cartridge. If you buy an additional cartridge now, you can save shipment cost. This kind of effective message always helps to have more products in the shopping baskets of the customers.

If we consider to mine a website like XYZ, we need to consider the following issues:

1) Every click stream data is important because it reflects a customer's buying interest. On the other hand, it is impossible to store all of the click stream data when thousands of customers generate click stream data every day.

2) If we wait long enough to perform mining tasks, mining algorithm will run out of memory when it will try to generate interesting patterns on extremely large dataset.

3) As data come continuously from XYZ's website, we are required to mine with the arrival of the data. In other words, we need to perform mining tasks while the customers are generating click stream data because this is an ongoing online process.

4) The management of XYZ may want to see the result from time to time. In other words, we need to keep updated results and be able to deliver while the mining process is on going.

5) The result has to be considerably accurate even though it is stream mining procedure.

These are very crucial issues we have considered before building our model.

3.3 Main Components

The main components of our model are given below (Figure 3.3-1).

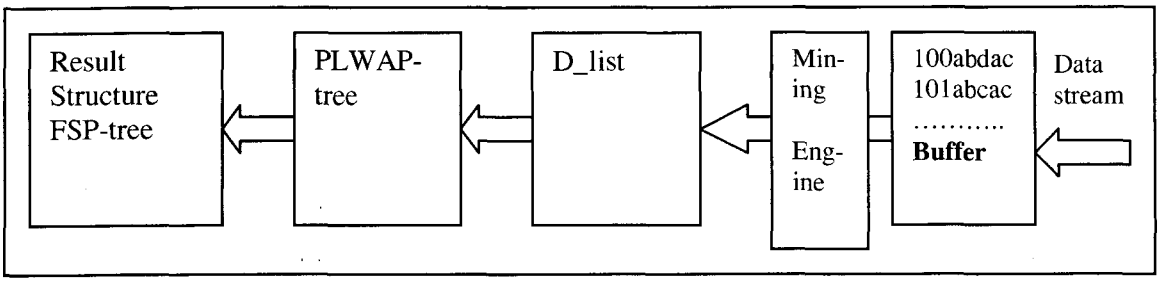

Figure 3.3- 1: Sequential Stream Mining Process

Buffer is basically a staging area where preprocessed transaction IDs and customer access sequences (CAS) arrive. We can envision a buffer as a long empty string initially. Once the stream started coming, we add stream into buffer. For example, <100, a b d a c>, <101, a b c a c>, <.....>. Here, 100 is the transaction ID and the letters (a b d a c) followed by transaction 100 are item IDs. Similarly, 101 is transaction ID and letters followed by 101 are item IDs (Figure 3.3-1). Lossy Counting Algorithm [Manku & Motwani2002] has used buffer mechanism to deal with incoming data stream. On the other hand, FP-Stream [Giannella et al.2003] used main memory to handle data stream. It is important to mention here that our mining work start from the buffer. In other words, if we receive sessionID and the sequential web pages visited by a user for that particular session instead of transaction ID followed by the item IDs, we would treat same. Our model can be used to find frequent sequential patterns of visiting pages for a site as well.

Mining Engine forms batches of the CAS (customer access sequences) data from the buffer. A batch is a group of customer access sequences. The size of the batch depends on the incoming stream. We do not keep our batch size fixed unlike Lossy Counting Algorithm [Manku & Motwani2002] or FP-Stream [Giannella et al.2003]. The size of the batches in our method changes dynamically. The size of the batch can grow bigger and smaller with incoming streams. If the environment is a heavy stream environment, then it would be efficient to have bigger batches. On the other hand, if the environment is a low stream environment, then it would be efficient to have smaller batches to avoid long time waiting period to form a batch. Before we form a batch, our mining engine waits for a

certain time. We can set this wait period time to 5 minutes, more or less. Once the predefined time elapses, it checks two thresholds (or variables) that are min_CAS and max_CAS. Min_CAS is a minimum number of customer access sequences and max_CAS is a maximum number of customer access sequences to form a batch. For example, we can set min_CAS = 4 and max_CAS = 1000 to form a batch. The engine follows two rules to form a batch.

Batch creating rules

Rule 1: After the elapsed time, if the buffer does not contain a minimum number of CAS (min_CAS), the batch creating process waits for a certain time. Here, time is a variable and it can be set to 20 minutes or 1 minute based on the requirements of the stream environment. For this example, say time = 5 (minutes). Here, the batch creating process retries to form a batch after 5 minutes and it continues this process as long as there are not a minimum numbers of transactions in the buffer.

For example, after elapsing five minutes, if there are two transactions or CAS waiting in the buffer but we set min_CAS to 4, it would wait another five minutes and try to form a batch at the second round. If there are still two transactions after a second round, it would go for third wait and continues this process until buffer does not contain a minimum number of transactions. Rule 1 prevents a continuous search to form a on the buffer if there are not enough transactions waiting in the buffer. This rule is ideal for a low stream environment.

Rule 2: After the elapsed time (say five minutes) if buffer has minimum number of batches, the process forms a batch and checks immediately whether the remaining transactions are more than maximum number of CAS or max_CAS threshold.

Otherwise, the process goes for a wait period after forming a batch.

For example, suppose there are 1800 transactions waiting in the buffer after elapsed time, the process takes 1000 transactions to form the first batch because we set maximum number of CAS or max_CAS to 1000 records for a batch. It checks immediately how many is remaining in the buffer and finds 800 is waiting to be batched. The process goes for a second wait and tries to form the next batch after second wait.

If yes, then it forms a next batch immediately and there is no wait period for this case. An example of case 2: suppose there are 5000 transactions waiting in the buffer after elapsed time, the process takes 1000 transactions to form the first batch. It checks immediately how many is remaining in the buffer, it finds 4000 left. It forms next batch right after taking care of first batch. There is no waiting period and it continues this process as long as the transactions in the buffer are not below the max_CAS threshold.

Batching and processing immediately based on Rule 2 minimizes system time and prevents waiting when there are enough transactions waiting in the buffer. If the process takes 5 minutes break regardless after forming a batch, and if the stream rate is higher than mining rate, then the buffer has chances to be over flooded at some point. At the same time, we do not want to keep the system always busy on the off pick hours when the traffic is slow.

Our intention is to utilize the systems efficiently compared to other proposed mechanism. For example, Lossy Counting Algorithm [Manku & Motwani2002] waits until the fixed sized buffer gets filled with data stream, then incoming stream is conceptually divided into batches to be processed. Now, if the buffer size is big and incoming traffic is slow, then there is a long waiting period. On the other hand, FP-Stream [Giannella et al.2003] waits 15 minutes regardless to form a batch. Now, if the environment is a heavy stream environment (say Google, Amazone.com), then every 15 minutes wait will accumulate millions of data and the mining algorithm has a risk to fall behind with the speed of incoming data stream.

Mining Algorithm or SSM-algorithm uses three data structures: D_list, PLWAP-tree and FSP-tree. It scans each batch to find frequent items. The frequent items are stored in d_list structure, PLWAP-tree is constructed by taking frequent subsequences from the batch, PLWAP-algorithm is used to mine frequent sequential patterns and the sequential patterns are stored into FSP-tree incrementally. This is an ongoing process. When the next batch arrives, we extract frequent items from the batch and update frequencies in d_list if they are already in d_list. Otherwise, we insert them as new elements. Perform PLWAP-mining by extracting frequent subsequences from the current batch. If the

obtained frequent patterns are already in FSP-tree, we update the frequencies in the tree. Otherwise, we insert them into tree as new patterns.

D_list is a data structure that keeps each items' ID and their frequency. We store items' ID and their frequency into this structure. A hash chained based indexing is implemented to update this structure. A detailed explanation is given about this indexing below in construction of d_list paragraph in section 3.4. Hash based indexing is used in d_list to speed up the search process.

PLWAP-tree or Pre-ordered Linked WAP-tree was introduced by Ezeife and Lu in [Ezeife & Lu2005][Lu & Ezeife2003]. The basic idea behind of PLWAP tree algorithm is to add pre-ordered linkage on a WAP-tree [Pei et al.2000]. In that way, the several intermediate WAP-tree reconstruction can be avoided that reduces storage as well as I/O computation time.

FSP-tree: Frequent Sequential Pattern-tree or FSP-tree is a simple form of Pattern-tree [Giannella et al.2003]. A detailed explanation is given about FSP-tree in section 3.4. Pattern-tree [Giannella et al.2003] is a part of FP-Stream [Giannella et al.2003] structure and we discussed earlier about this tree in section 2.10.

3.4 Discussion of SSM-Algorithm

In our example, we use the following symbols:

I = items

f = frequency (occurrences of a items)

Is = itemsets (a set of items)

C_1 = candidate 1-sequence (length of the sequence is 1 of items before pruning).

L_1 = large 1-sequences (the length of the sequence is 1 of items after pruning).

Required input:

(1) Minimum support threshold (s) where (0<s<1).

(2) Maximum support error threshold (e) where (0<e<s). The value of "e" gives us flexibility to have an error tolerance in our result up to that level. As stream mining algorithm moves forward with incoming stream and does not have flexibility to go

backward and scan previous data stream because of that it does not guarantee 100% accurate result. Therefore, a certain level of error is tolerable in the result. However, this tolerance varies from application to application. When the support error (e) is very small compared to minimum support (s) threshold, then the result will have very minimal errors. There is a rule of thumb from [Manku & Motwani2002] for "e" of one tenth of the value of "s". For our example, we have set s =.75 (75%) and e = .25 (25%) in order to show less calculations and make it easy for the readers to follow our example. For our experimental analysis, we followed the rule of thumb.

Required variables

(1) Maximum number of CAS or max_CAS, (2) Minimum number of CAS or min_CAS, (3) An incoming batch B_i. The batches arrive one after another. Our method, groups transactions and form a batch and send the batch to be mined to the next level.

Output: (1) FSP-tree. This structure stores all of the sequential frequent patterns.

Method

i) Form the first batch B_1 (Figure 3.4-1) from the buffer where $|B_1| = 4$.

ii) Scan B_1 and generate C_1^{B1}={a:4, b:4, c:3, d:1, e:1, f:2, g:1}.

iii) Construct data structure d_list and insert C_1^{B1} into d_list without pruning any items (elements) for the first batch (Figure 3.4-2). The construction and insertion method of d_list is described below.

Construction of d_list: Build a hash table (hash array) with a number of buckets. In our case, we decided to have 100 buckets so that hash array size would be 100. They are labeled 0 to 99 by default. Initialize each bucket that points to null value. We already found C_1^{B1}={a:4, b:4, c:3, d:1, e:1, f:2, g:1}from batch B_1. We take first element (a:4) from C_1^{B1} where "a" is the item ID and "4" is the frequency and pass item's ID to the hash function.

Create hash function: We create a function that takes an integer value and returns an integer. For instance, in this case, we pass item's SKU to a hash function that returns SKU mod hash array size. SKU number is a unique number for an item. SKU numbering is used in retail or e-commerce for

ItemID	SKU
a	2
b	100
c	0
d	202
e	10
f	110
g	99

Table 3.4- 1: Items SKU value

items to identify them. As a matter of fact, item "a" has a unique SKU number. Item "a" is the name of an item but we find its SKU from Table 3.4-1. We pass integer 2 (SKU of "a") to the hash function. It returns SKU % hash table size = 2 % 100 = 2.

Insertion into d_list: returned value 2 from hash function is matching with third array element of the hash table (Figure 3.4-2). We insert a node that is labeled "a" and its frequency "4" into the third array's chain. We create a linkage that points to the new node from the third array of hash table and the new node points to the null value (Figure 3.4-2). We take next element (b:4) and pass the item's SKU that is 100 (from Table 3.4-1) to the hash function. It calculates as 100 % 100 = 0 and returns 0 that matches with the first bucket of hash table. We place a node that has label "b" and frequency '4" into that

CAS	FS
a b d a c	a b a c
a b c a c	a b c a c
b a b f a e	b a b f a
a f b a c f c g	a f b a c f c

Figure 3.4- 1: B_1 (first batch)

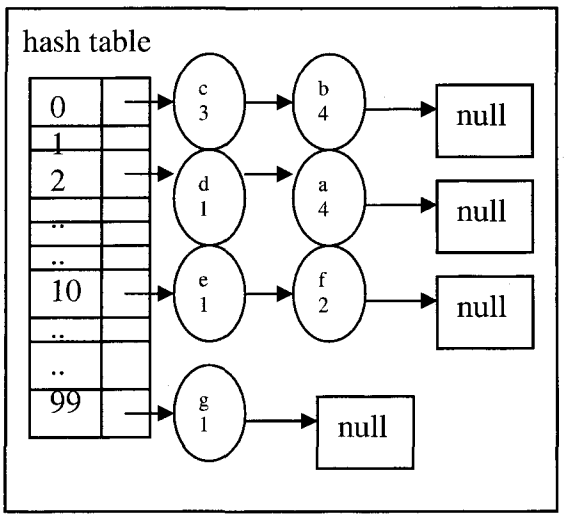

Figure 3.4- 2: d_list

bucket chain (the chain grows longer when more nodes arrive in the same bucket). We create a linkage for bucket 0 or first bucket that points to this new node and new node points to null value. Now, we take next element (c:3) from C_1^{B1} and pass its SKU to the hash function. Hash returned value (0 % 100 = 0) matched with bucket 0 because its SKU is 0 (from Table 3.4-1). Now, the new node that has label c and frequency 3 get placed into bucket 0. We modify linkage, bucket 0 points to this new node and new node points to node that was pointed to by bucket 0 before. Similarly, we pass the SKU of item "d" to the hash function. Hash function returns value 2 by calculating its SKU value. There is another node "a" and its frequency 4 is already in this chain. We place new node "d" with

frequency 1 before node "a"(Figure 3.4-2). In other words, we modify the linkage just the way we have done before for element "b", bucket 2 points to the new node and new node points to node that was pointed to by bucket 2 before. Similarly, next elements "e" and "f" goes to bucket 10 and element "g" goes to bucket 99 from C_1^{B1}.

Once d_list is constructed, the performance of insertion, update and delete of nodes becomes faster through this hash chain structure. D_list guarantees to limit the search nodes by number of items divided by number of buckets when there are update requires. For example, if we have 100 buckets and 1000 distinctive items, d_list restricts the search within 10 nodes (number of items / size = 1000 / 100). For the worst case, it would search 10 nodes to find the appropriate node and for the best case, it search only one node.

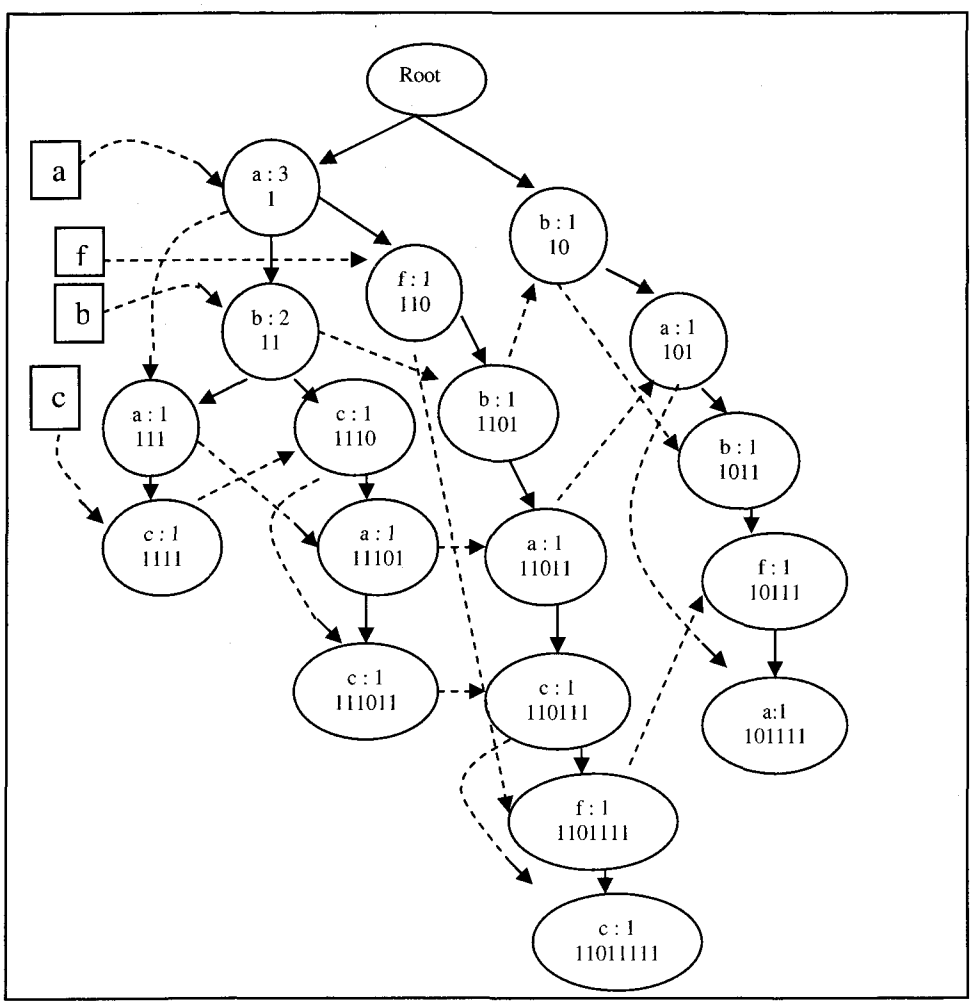

Figure 3.4- 3: PLWAP-Tree for B_1

We have constructed d_list and inserted all elements into d_list from C_1^{B1} into d_list that was found from first batch B_1. Now, we need to find large 1-sequences from d_list in order to find frequent sequential patterns. Note, we do not need C_1^{B1} anymore because we will be updating d_list from now on. So, we drop C_1^{B1}.

iv) We scan d_list select items as Large 1-sequences or L_1^{B1} from d_list if f $\geq$ num_CAS * (s-e) = 4 * (.75 - .25) = 4 * .50 = 2. Here, num_CAS is number of customer access sequence and that is 4 because we had 4 transactions in batch B_1. We find L_1^{B1} = {a:4, b:4, c:3, f:2}.
Note: this large 1-sequence can be selected during the insertion process of elements into d_list rather than selecting after inserting all elements into d_list. We recommend to perform L_1 selection process during the insertion process of d_list because it saves a scan.

v) Scan each row of B_1 (CAS) and delete the items that are not in L_1^{B1} to generate frequent subsequence or FS in Figure 3.4-1, FS column.

vi) Select each FS from Figure 3.4-1 and construct a PLWAP-tree [Ezeife & Lu2005][Lu & Ezeife2003] (Figure 3.4-3). The construction of PLWAP-tree and mining algorithm of PLWAP-tree is described in [Ezeife & Lu2005][Lu & Ezeife2003] and in section 2.7 of this thesis.

Construction of PLWAP-tree for B_1: We create root of the tree with null value. Insert fiirst frequent subsequence (a b a c) into the tree. We take first element "a" from the subsequences and label it "a" node and place it under the root node. It becomes child node of Root node. We enter value 1 and acquire position 1. Similarly, next element "b" from the subsequences becomes "b" node that has frequency 1 and position code 11 goes right under node "a" as its child. We insert rest of the sequences (a c) in a similar fashion. We move on to the next sub sequences (a b c a c) and insert them into the tree. If a new node matches with an existing node in the tree, we increment its frequency, otherwise we create a new branch from that point and enter rest of the elements under that branch. By entering all of the frequent subsequences from batch B_1, we construct PLWAP-tree

(Figure 3.4-3). Once the construction of PLWAP is completed, a pre-order traversal mechanism is used to add a pre-order linkage on the tree.

vii) Mine PLWAP-tree to generate frequent sequences using PLWAP-mining algorithm [Ezeife & Lu2005] [Lu & Ezeife2003] and also discussed in section 2.7 of thesis.

Mining PLWAP-tree: The mining method recursively mines frequent sequential patterns from the tree. We recursively generate frequent sequential patterns for batch B1 (or FP^{B1}). ALL of our frequent sequential pattern or FP^{B1} have frequency $f \geq (s\text{-}e) * |B_1| = .50 * 4 = 2$. We find FP^{B1} = {a:4, aa:4, aac:3, ab:4, aba:4, abac: 3, abc: 3, ac: 3, acc:2, af: 2, afa: 2, b: 4, ba: 4, bac: 3, bc: 3, c: 3, cc: 2, f: 2, fa: 2}.

FSP-tree: Once the frequent patterns are found for a particular batch say B_1, then the next task would be to store these patterns in a structure that could be retrieved easily future use because this is an ongoing process and more batches are coming one after another. Next batch B_2 will be mined and obtained patterns will contain some new patterns and old patterns as well. We meant new patterns that are completely new, we did not obtain this kind of patterns from previous batch or batches. Old patterns means, these patterns are found already. In that situation, we need to update the frequency of old patterns and insert new patterns into the structure. Similarly, batch $B_3 \ldots B_n$ will be handled. At one point, some patterns will become small because the frequency of incoming patterns varies from batch to batch. Therefore, we need to delete small patterns from the structure in order to keep the structure compact, updated and able to deliver updated result.

We proposed to construct a tree named FSP-tree that will store only frequent sequential patterns to handle above issue. Please note here, we are using PLWAP-Algorithm to mine batch one after another and every time we are initialing PLWAP-tree empty once we find the patterns from a particular batch. Our FSP-tree stores obtained patterns batch after batch and update its structure accordingly. FSP-tree is basically result structure and we use FSP-tree to deliver result on demand. (Please note here in case reader is interested,

PLWAP-tree can be updated after every batch rather than reconstructing like FSP-tree to save construction time. But this is not definite that it would work in stream environment because this PLWAP-tree update process would require to keep all small items' information and their possible position in the tree and we usually do not follow this mechanism in stream environment. However, the solution of updating PLWAP-tree in data stream could be a future work in this direction.)

As was already mentioned, frequent sequential pattern-tree or FSP-tree is a simple form of Pattern-tree [Giannella et al.2003]. Pattern-tree [Giannella et al.2003] is a part of FP-Stream [Giannella et al.2003] structure. [Giannella et al.2003] uses Pattern-tree to store frequent Patterns of incoming batches (a complete discussion is done of Pattern-tree in section 2.10). However, pattern-tree and FSP-tree differ from each other in many ways. The major differences in between Pattern-tree and FSP-tree are:

1) A tilted time window table is embedded into Pattern-tree. It is possible to perform time sensitive queries on Pattern-tree because of this tilted time window. As we are not interested in time sensitive queries, therefore, FSP-tree does not maintain any tilted time window tables.

2) Pattern-tree stores and maintains frequent patterns. FSP-tree stores and maintains frequent sequential patterns.

3) FSP-tree maintains a footer list unlike Pattern-tree. This footer list has linkage to each leaf node of the tree. We use this list for the maintenance purposes because our algorithm starts searching the tree from the leaf node to the root in order to save searches. However, for inserting new nodes or updating frequencies of existing nodes in the tree, our algorithm reads from the root of the tree to the leaf. This gives flexibilities to access the tree from the root or from the leaf. We find it is very efficient. It saves additional scans on the tree for maintenance purposes. As per tree construction, the lower frequency nodes reside at the bottom of the tree. We can get rid of the unwanted nodes easily from the lower part of the tree if we read the tree from the bottom to top. At the same time, it is

quicker if we insert new nodes or read the tree from top to bottom to find supported nodes or patterns for result.

Now, let's continue our example. We obtained frequent patterns from batch B_1 using PLWAP-Algorithm [Ezeife & Lu2005] and the patterns are FP^{B1} = {a:4, aa:4, aac:3, ab:4, aba:4, abac: 3, abc: 3, ac: 3, acc:2, af: 2, afa: 2, b: 4, ba: 4, bac: 3, bc: 3, c: 3, cc: 2, f: 2, fa: 2}. Now, we need to construct FSP-tree and insert the patterns into the tree. We have explained above why do we need FSP-tree earlier and what are the major differences in between Pattern-tree and FSP-tree.

viii) Construct FSP-tree and insert all FP^{B1} into FSP-tree without pruning any items for the first batch B_1. Please note, we are still handling batch B_1.

Construction of FSP-tree: The construction of FSP-tree is similar to Pattern-tree that was introduced in [Giannella et al.2003]. First create the root node of this tree and make

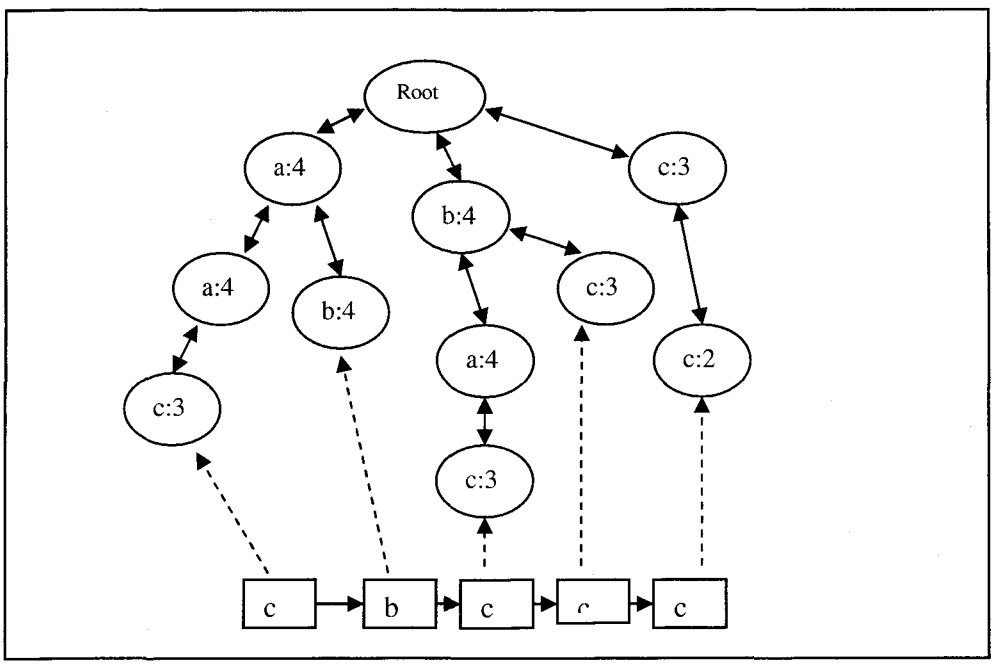

Figure 3.4- 4: Frequent Sequential Pattern- tree

all values to NULL value. Then insert frequent patterns FP^{B1} = {a:4, aa:4, aac:3, ab:4, aba:4, abac: 3, abc: 3, ac: 3, acc:2, af: 2, afa: 2, b: 4, ba: 4, bac: 3, bc: 3, c: 3, cc: 2, f: 2, fa: 2}into the FSP-tree one by one. Insertion starts with the first pattern of FP^{B1} (a:4). Create a node, label it "a" and its frequency is 4. We insert node "a" with frequency 4 under root as the left child of it. Next pattern (aa:4), we insert into the tree as flows: pattern (aa:4) has two elements or sequences. First element "a" of pattern (aa:4) is matching with existing left child of root node "a". Therefore, we stay within the same branch. Now, second element "a" of pattern (aa:4) does not match with any existing node. Therefore, we create a new node labeled as "a" with frequency 4 and insert it as a left child of node "a" with frequency 4 (root's left most child). In other words, we extend the same branch of the tree (Figure 3.4-4). Please note here, pattern (aa:4) has two elements. If a pattern has more than one element, in that case, we enter frequency of that particular pattern into the last element's node.

Similarly, the first element "a" of pattern (aac:3) is matching with the left child of root node "a" with frequency 4, we follow the same branch. Next element also "a" is matching with existing node "a" with frequency 4 as well into the tree, so, we follow the same branch. Now, the last element "c" of pattern (aac:3) does not match with any nodes. We create a new node, label it "c" and enter frequency 3 into it because it is the last element of the pattern (aac:3). Insert this new node as a child of node "a" with frequency 4 in the tree (Figure 3.4-4).

When we try to enter next pattern (ab:4) into the tree, the first element "a" of ("ab") is matching with left child of root node "a" with frequency 4, therefore, we follow the same branch. But the second or last element "b" of (ab:4) does not match with any nodes. We create a new branch from that point, create a new node, label it "b" and enter frequency 4. Insert this new node into the tree as right child of (a:4) node.

After entering pattern (aba:4, abac: 3, abc: 3, ac: 3, acc:2, af: 2, afa) into the tree in similar fashion, when we try to enter pattern (b:4) into the tree, the only first element of this pattern does not match with any existing children of the root. In that case, we create a new branch under root and make it as next right child of the root and keep extending that branch as long as we receive compatible elements. We insert rest of the patterns from FP^{B1} into the tree and complete construction process.

FSP-tree maintains a footer list (Figure 3.4- 4). We named it footer list because it has linkage to each leaf node of the tree. Footer list is a linked list that grows with the leaves of the tree. Our FSP-tree maintains this list in order to read the tree from the leaf instead of root for maintenance purpose of the tree. We discuss more about this maintenance process in section 3.5.

Once the construction, insertion, and linkage are done on FSP-tree for a particular batch, in our case B_1, then we move on to update procedure. In update procedure, we constantly update d_list and FSP-tree with incoming batches.

Update method: Assume next batch B_2 arrived. We treat first batch differently than other batches because we do not perform any delete operations for the first batch.

CAS	FS
a b a c g	a b a c g
a b a g	a b a g
b a b a g	b a b a g
a b a g h	a b a g

Figure 3.4- 5: Next batch B_2 formed

i) We form the second batch B_2 (Figure 3.4-5) by taking x number of records from buffer where min_CAS $\leq x \leq$ max_CAS. Say $|B_2| = 4$. //say min_CAS=4 and max_CAS=1000

ii) Scan B_2 and generate C_1^{B2} = {a:4, b:4, c:1, g:4} from B_2. It is obvious that d_list will grow bigger with incoming batches when we will have thousands of items. We take the advantages of hash based indexing to update d_list.

Update d_list: The update method is simple. When a new node arrives, the hash function places it into appropriate bucket. The method checks the linkage for that bucket chain. If it finds another node has same label, it increments the frequency of that node in d_list. If there are no identical nodes existing in that chain, the new node gets placed right at the beginning position of the chain.

Read d_list: When we need to know a particular items' frequency in the d_list, we pass the item ID to the hash function, it finds appropriate bucket for that item. Once the bucket

is found, the process checks the linkage from that bucket and finds right node and returns its frequency.

Delete nodes in d_list: The delete process of nodes in d_list is similar to read method. The delete process deletes the node when it is found. On the other hand, read process reads the node when it is found.

Now, we have C_1^{B2} = {a:4, b:4, c:1, g:4} and we want to update d_list. We check each element in C_1^{B2} sequentially and update corresponding elements in d_list. For example, first element a:4 in C_1^{B2} gets incremented in d_list to a:8 because we found an element a:4 is existing in d_list. After incrementing its frequency or count, if the current frequency of the updated item is f $\geq$ num_CAS * (s-e) = 8 * (.75 - .25) = 8 * .50 = 4, we select that item as L_1^{B2}. Thus, we select (a:8) as L_1^{B2}. Similarly, we check next element in C_1^{B2} (b:4), we increment its frequency to (b:8) and select it as L_1^{B2}. We find (c:4) and (g:5) are in L_1^{B2} as well from C_1^{B2} and L_1^{B2} = {a:8, b:8, c:4, g:5}. However, if frequency of updated element in d_list is f < num_CAS * e = 8 * .25 = 2, we delete that item from d_list. We did not have to delete any items because all of our updated elements have frequency more than num_CAS * e = 8 * .25 = 2. Here, num_CAS is the total number of transactions up to current batch. B_1 had 4 transactions and B_2 has 4 transactions. So, num_CAS = 4 + 4 = 8.

If any items in C_1^{B2} do not match with the elements in d_list, we check frequency of that item or items. If the frequency of that items is f $\geq$ num_CAS * (s-e) = 8 * (.75 - .25) = 8 * .50 = 4 (say, for batch B_2), we select that item as L_1^{B2} and insert that element in the d_list. If the frequency of this item is f < num_CAS * (s-e) = 8 * (.75 - .25) = 8 * .50 = 4 and f $\geq$ num_CAS * e $\geq$ 8 * .25 = 2, we insert that element in the d_list but do not include as L_1^{B2}. Otherwise, delete that item from C_1^{B2}.

iii) It is also need to check the status of previous large items that did not appear in C_1^{B2}. We use our formula $\{\neg(L_1^{B(i-1)} \cap C_1^{Bi})\} \cap L_1^{B(i-1)}$ = {f:2} to find out previous large (L_1^{Pi}) items that did not appear in current candidate 1-sequences.

Example of the Correctness of the Formula:

We previously found L_1^{B1} = {a:4, b:4, c:3, f:2} for B_1

We found candidate 1-sequences for current batch B_2 or C_1^{B2}= {a:4, b:4, c:1, g:4}

For i = 2 to n (Current batch = B_i where i = 2 now, or current batch = B_2.

Thus, previous batch = $B_{i-1} = B_1$).

First part of the formula: NOT INTERSECTS elements in between large 1-sequences for previous batch and candidate 1-seqeunces for current batch = { NOT $(L_1^{B(i-1)} \cap C_1^{Bi})$}= $\{\neg(L_1^{B1} \cap C_1^{B2})\}$= $\{\neg(\{a:4, b:4, c:3, f:2\} \cap \{a:4, b:4, c:1, g:4\})\}$={f:2,g:4}.

Second part of the formula: Result of first part that intersects with large 1-sequences for previous batch = $\{f:2,g:4\} \cap L_1^{B(i-1)}$ = $\{f:2,g:4\} \cap L_1^{B1}$ = $\{f:2,g:4\} \cap \{a:4, b:4, c:3, f:2\}$= f:2 = L_1^{Pi}

Justification of formula:

Element "f" (L_1^P) did not appear in current candidate 1-sequences (C_1^{B2}). However, element "f" was large previously. Even though, "f" is not in C_1^{B2} but we still need to check its frequency to find whether it is still large. What would happen suppose the frequency of "f" is 4 for B_1. In this case, we would not find "f" in L_1^{B2} without using our formula. Thus, mining result would not be accurate.

We found L_1^P using our formula, now we need to find where L_1^P belongs. If frequency of L_1^P f $\geq$ num_CAS * (s-e) = 8 * (.75 - .25) = 8 * .50 = 4, then, it is large items for current batch. If frequency of L_1^P f $\geq$ num_CAS * e= 8 * .25 = 2, it remain in d_list but we do not include it as L_1^{B2}. If frequency of L_1^P f < num_CAS * e = 8 * .25 = 2, we delete it. In our case, element "f" remain in d_list but do not get included in L_1^{B2}. At the end, we find L_1^{B2} = {a:8, b:8, c:4, g:5}. Once the update process is completed of d_list for the second batch, we drop L_1^{B1}.

iv) We scan each row of B_2 (CAS) and delete the items that are not in L_1^{B2} to generate frequent subsequence or FS in Figure 3.4-5, FS column.

v) We construct PLWAP-tree and apply PLWAP-algorithm [Ezeife & Lu2005] to generate itemsets that have frequency f $\geq$ (s-e) * size of a batch = (s-e) * $|B_2|$=.50 * 4 = 2.

We find frequent sequential pattern or FP^{B2} = {a:4, aa:4, aag:4, ab:4, aba:4, abag:4, abg:4, ag:4, b:4, ba:4, bag:4, g:4}. We delete B_2, initialize PLWAP-tree to empty.

vi) Insert frequent FP^{B2} into FSP-tree, if the itemsets are already in FSP-tree, we increment their frequency. If the items are not in FSP-tree but in FP^{B2} and frequency f >= num_CAS * e = 8 * .25 = 2, we insert as new elements in the tree. Otherwise, we delete them from FP^{B2}.

CAS	FS
a b c g	a b g
a e g d	a g
a b f a g	a b a g
a f e g	a g

Figure 3.4- 6: Batch B_3

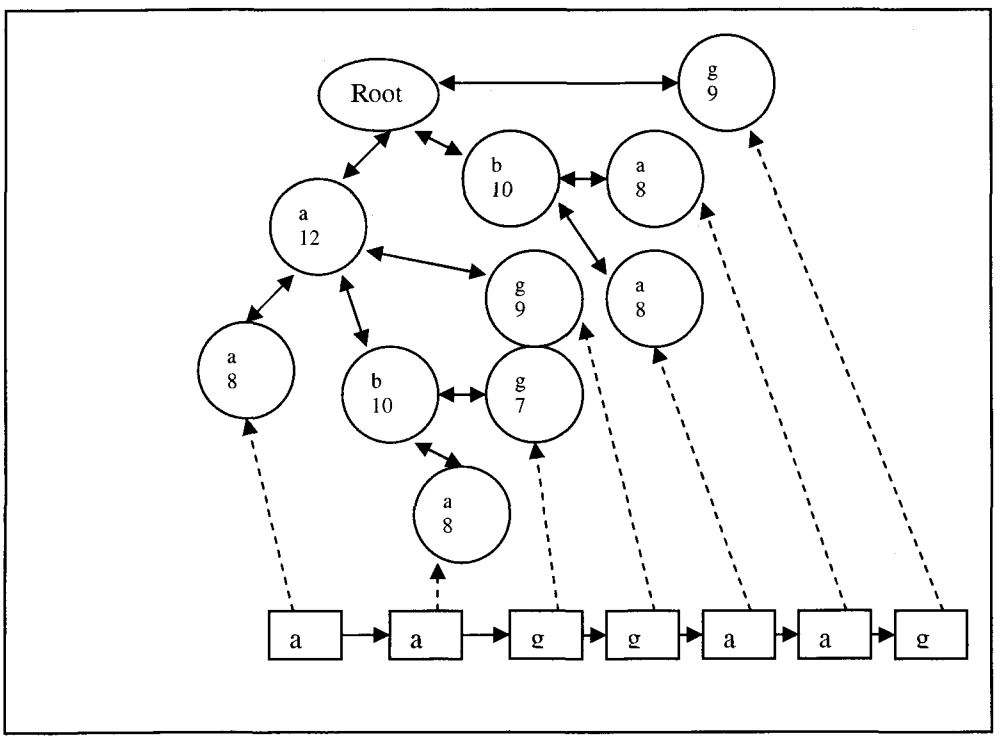

Figure 3.4- 7: updated partial frequent sequential pattern –tree (FSP-tree)

Similarly, we form next batch B_3 (Figure 3.4-6) and follow all of the steps as we have performed for batch B_2. We find FP^{B3} = { a:4, ag:4, ab:2, abg:2, b:2, bg:2, g:4} from batch B_3 using PLWAP-Algorithm. Now, we need to insert them in FSP-tree. We insert

FP^{B3} into FSP-tree, increment the frequencies of itemsets in FSP-tree if they are in FP^{B3}. If the items are in FP^{B3} but not in FSP-tree, we insert as new elements in FSP-tree if the frequency of items f ≥ num_CAS* e = 12*.25 = 3. Otherwise, we delete the items in FP^{B3}. This result structure or FSP-tree basically maintains updated result (Figure 3.4-7).

vii) So far, we have taken care of batch B_1, B_2 and B_3 and updated d_list and FSP-tree accordingly. Suppose our user wants to view the result. In that case, we find itemsets from FSP-tree that have frequency f ≥ (s –e) * num_CAS = .5 * 12 = 6. We recursively find FP = {a:12, aa:8, ab:10, **aba:8, abg:7**, ag:9, b:10, ba:8, **bg:7**, g:9}from FSP-tree using FSP-Mining algorithm.

FSP-Mining algorithm: Suppose we are mining FSP-tree in Figure 3.4-7. The mining algorithm starts mining from root node of the tree recursively. Move to the left most child of the root first. It finds node "a" and its frequency is 12 that is more than 6, so, the algorithm selects it as FP. Next it goes down to its left child. The left child is "a" with frequency 8. Select (aa:8) as FP because its frequency is more than 6. Now, node (a:8) is a leaf node. So, the process goes back to its parent to check its sibling. Parent (a:12) has next right child to be mined. The process visit next right child of node (a:12). Find (b:10), so, it selects (ab:10) as FP. The process goes one level down to its left child (a:8) and select (aba:8) as FP. Now, node (a:8) is a leaf node. Therefore, the mining process goes back to its parent node (b:10) and find its right child to be mine. The process visit its right child and find pattern (abg:7). Similarly, it finds rest of the pattern (ag:9, b:10, ba:8, bg:7, g:9) recursively.

Property 1: The parent nodes always have frequency more than or equal to its children in FSP-tree. Therefore, if parent node does not have minimum support, its children are ignored during mining process. While the mining algorithm searching frequent pattern or FP in FSP-tree from root to down to leaf node for a particular branch during the journey, if it finds any node that does not have minimum support, it does not go further down. It cuts the suffix sequence of the branch from that point. We name it Suffix Pruning I.

For example, we are mining a branch of a FSP-tree (Figure 3.4-8) where minimum support is set to 6. We start with root and visit its left child node "a" with frequency 8. We select (a:8) as FP. Now, the process moves one level down, it find node "d" and its frequency 4. Node "d" does not have minimum support. Therefore, it does not select (ad:4) pattern as FP rather it drops the mining process for that particular branch based on property 1.

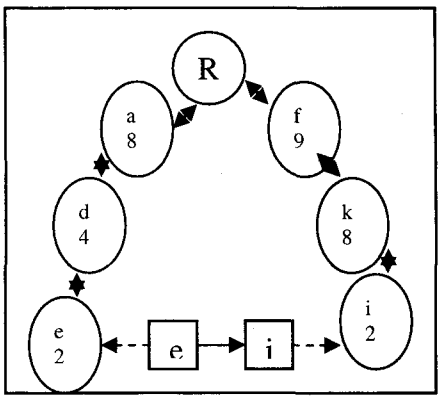

Figure 3.4- 8: A branch of FSP-tree

3.5 Maintenance of FSP-tree

We use the footer list of FSP-tree for deleting nodes or maintaining the structure of the tree. In order to delete a node or nodes that do not have minimum support or a defined threshold, we read from the leaf of the tree (footer list has linkage to the leaf nodes) and keep going up toward the root. The reason we want to delete from the bottom to the top because as per our design construction of FSP-tree, the lower nodes of the tree usually have lower supports. For this reason, our delete operation will be quicker and we do not need to visit the entire structure. Once it deletes a node, it moves to the next level up and checks its frequency. After several search operations or an operation when the algorithm finds a node that has minimum support (or required support), it stops searching that branch, comes back to the next element of the footer list and continues deleting the nodes that was pointed from the next element of the list.

Property 2: The children nodes always have lower or equal frequency than its parent in FSP-tree. Therefore, if children have minimum support, there is no need to check its parent's frequency. Property 2 is the vice versa of property 1.

While the algorithm is moving from the leaf to the root of the tree, it checks the frequency of each node during the journey. If it finds any node has minimum support, it would not move forward to check its parent's support. As per our tree construction, the children node always have lower or equal frequency to its parent and parent nodes always have frequency higher or equal to its children. We named it Suffix Pruning II.

For example, we are performing delete operation in order to keep the size of the tree small during the maintenance process. Here, predefined minimum support is set to 6. We use footer list for this purpose (Figure 3.4-8). We start with the leaf node of the tree that was pointed from the first element of the list. Suppose, we node "l" from the list and it has frequency 2 which is less than minimum support. So, we delete this node and move up. In Figure 3.4-8, we find node "k" has frequency 8. As we know from property 2 that its parent, parent's parent up to root will have equal frequency or higher (not lower) than the frequency of "k". So, the algorithm does not check the branch above node "k". Instead, it comes back to the footer list and start deleting next node if it is pointed from the next element of the list.

Maintenance Method:

i) Perform suffix pruning II in FSP-tree after a boundary. Boundary is a variable and we can set this to 10000 transactions for our experiment. In other words, we perform suffix prune II in FSP-tree after taking care of 10000 transactions in order to reduce the size of the structure. The footer list has linkage to all leaf nodes to the FSP-tree. We prune those elements that do not have support f < num_CAS* e. Above we discussed with an example about Suffix Prunning II in Property 2 (section 3.5 of thesis).

ii) Perform a Sequential Pruning on d_list once we are done with suffix pruning II at the boundary. Sequential pruning is the process that checks the frequency of each element in d_list sequentially. If the frequency of elements f < num_CAS* e, we delete them from d_list.

Now, if we take B_1, B_2, and B_3 as one piece and mine to generate frequent sequential pattern with support s (.75), using PLWAP-algorithm, we find FP' = { a:12, aa:9, ab:10, ag:9, b:10, ba:9, g:9}. We assume that FP' is 100% accurate because this mining process is already proven its accuracy by the authors.

Claim1: If we compare itemsets in between FP (frequent sequentially patterns generated by SSM-Algorithm incrementally from FSP-tree) and FP' (sequential pattern generated

by PLWAP-Algorithm), we find that FP has all itemsets that FP' have, in addition FP has some additional itemsets (aba:8, abg:7, bg;7) that FP' does not have. We can refer them as false positive itemsets. As per result, we can claim that FP' is a subset of FP. Thus, we claim that our result (FP) contains all itemsets that have support s.

Claim 2: It is true that we have some false positive itemsets with our result because we had to lower our support threshold from s (.75) to (s-e=.75-.25=.50) during the mining process as we do not keep any records for small items from the beginning. Now if we deduct the maximum support error from user defined support, the acceptable result is 50%. If we calculate required support (or acceptable support) after error tolerance in 12 transactions is (s-e=.75-.25=50% or (.5 * 12 = 6). That means an item has to occur 6 times in 12 transactions in order to be accepted as large itemset if we utilize error tolerance threshold. We do not have any itemsets in our result that occurred less than 6 times. Therefore, we can claim that our result does not cross the maximum error support threshold or error tolerance boundary.

Algorithm 0: Main Algorithm
Input: (1) Minimum support threshold (s) where 0<s<1, (2) Maximum support error threshold (e) where 0<e<s
Output: 1) Frequent sequential patterns
Begin
Temp variables boolean exit = true, boundary;
1. Create_Batch(CAS) //Batch create method
1.1 Scan B_i and generate candidate 1-sequences or C_1^{Bi} // C_1 generation
2. Create D_list[Size] // Hash index based structure creation
3. Generate_Frequent_Pattern(FS) // FP generation
4. Construct_FSP_tree (FP) // construct method for FSP-tree
5. Update_FSP_tree(P) // Update method for FSP-tree
While (exit){ // exit when user wants
6. Continuous_Update(CAS) // continuous update of FSP-tree
7. If user wants to view result, then Result(Size) //result on demand
8 If i ==boundary, then Maintain_Structures() // run maintain method at boundary
9 If user wants to exit, exit = false;
}
End

Algorithm 1: Batch Creation Program
Create_Batch (CAS)
Input: (1) Customer Access Sequences (CAS), Incoming data stream into Buffer
Output: A Batch
// Static Variables: min_CAS, max_CAS;
Begin
For 1 to m
Create a Batch B_i by taking number of records from Buffer
where $|B_i| >$ min_CAS && $|Bi| \leq$ max_CAS)
End

Algorithm 2: Create d_list, a hash chain based data structure
d_list[size]
Input: (1) Size
Output: 1) L_1^{Bi}
//Temp Variable $L_1^{Bi} = 0$;
Begin
1 Create d_list [size] (a hash array) with buckets initialize to Null.
For i to q
Insert (M)
2 For each item (M) in C_1^{B1}
2.1 Find hash bucket in d_list [size] using hash function (fn) r = M % size, then insert into appropriate bucket
2.2 If element of bucket d_list [r] has count $\geq$ num_CAS * (s-e), then $L_1^{B1} = L_1^{B1} \cup M$
End

Algorithm 3: Generate Frequent Sequential Pattern
Generate_Frequent_Pattern(FS)
Input: (1) Frequent Subsequences (FS), incoming batch Bi
Output: 1) Frequent Sequential Pattern
Begin
1. Scan each row of B_i, delete items that are not in L_1^{Bi} to generate FS
2 Select each FS and construct a PLWAP-tree [Ezeife & Lu2005] (in section 2.7 of thesis).
2.1 Apply PLWAP-algorithm [Ezeife & Lu2005] (section 2.7of thesis) to generate (FP^{Bi}) for B_1 where each $FP^{Bi} \geq (s\text{-}e) * |B_i|$
3 Initialize PLWAP-tree to empty and delete B_i
End

Algorithm 4: Construct FSP-tree (FP^{B1})
Construct_FSP_tree (FP^{B1})
Input: 1) Frequent patterns or FP^{B1}
Output: 1) Frequent Sequential Pattern –tree (FSP-tree) T
// Construct method of FSP-tree
Begin
1 Create a root node of tree T and set all value to NULL, make root as current_node
2 For each pattern p in FP^{B1} do
2.1 Create a node by taking 1st element of p as a child of root, then
make new node as current_node
2.2 If length of p > 1
keep extending current branch until comes to the last element of p.
2.3 Enter count of the pattern at the last element of p, then
make it current_node.
2.4 Else
2.5 Enter count and make it current_node
End

Algorithm 5: Update FSP-tree (P)
Update_FSP_tree(P)
Input: 1) Frequent patterns or FP^{Bi}
Output: 1) Frequent Sequential Pattern –tree (FSP-tree) T
Begin
//Update method of FSP-tree
3 If child of current_node == first element of p, then node Found ()
3.1 If node Found()
3.2 Keep following and matching current branch with pattern p
3.3 Add count at the last element of p, make it current_node
3.3.1 If pattern does not match with current branch, then
create a new branch from that point, extend current branch until last element of p
3.3.2 Enter count at the last element of p, make it current_node
3.4 Else if node NOT Found()
3.4.1 Create a new branch under root,
// continue algorithm 5

3.4.1.1 Keep extending current branch until comes to the last element of p
3.4.1.2 Enter count at the last element of p, make it current_node.
4 Maintain a footer list that points to each leaf_node of T
End

Algorithm 6: Update FSP-tree for incoming batches
Continuous_Update(CAS)
Input: Customer Access Sequences (CAS)
Output: (1) Update FSP-tree structure
Begin
For 2 to r
1 Create_Batch (CAS)
2 Compute candidate list C_1^{Bi} from batch B_i
3 For each item M in C_1^{Bi} do
3.1 Find hash bucket in d_list [size] using hash fn r = M % size,
increment appropriate element in d_list[r] if M found
3.1.1 If the appropriate element of bucket d_list[r] containing M has count ≥ num_CAS * (s-e), then $L_1^{Bi} = L_1^{Bi} \cup M$ //initially $L_1^{Bi} = 0$
3.1.2 If M has count < num_CAS * e, delete that element in d_list[r]
3.2 Else If M not found in appropriate bucket in d_list[r]
3.2.1 If count of M ≥ num_CAS * (s-e), then $L_1^{Bi} = L_1^{Bi} \cup M$, insert M in d_list [r]
3.2.2 Else if count of M < num_CAS * (s-e) && f ≥ num_CAS * e, insert M in d_list [r]
4 Find previous large items L_1^{pi} using formula $\{\neg(L_1^{Bi-1} \cap C_1^{Bi}) \cap L_1^{Bi-1}\}$
4.1 If count of $L_1^p \geq$ num_CAS * (s-e), then $L_1^{Bi} = L_1^{Bi} \cup L_1^p$
4.2 Else if count of L_1^p < num_CAS * e, delete L_1^p in d_list[r]
5 drop L_1^{Bi-1}.
6. Scan each row of B_i (CAS) and generate FS.
7. Select each FS, construct a PLWAP-tree [Ezeife & Lu2005] with FS, then generate (FP^{Bi}) with count ≥ (s-e) * $|B_i|$ for B_i // $|B_i|$ = size of current batch
8. Initialize PLWAP-tree to empty, delete B_i.
//Insert frequent sequential patterns or FP^{Bi} into FSP-tree
9. Update_FSP_tree(P) (Algorithm 5 of this thesis)

Algorithm 7: Result
Result()
Input: Use previously defined input (1) support s (0<s<1) (2) error e (0<e<s)
Output: 1) Result from FSP-tree
//Mine FSP-tree using count ≥ (s –e) * num_CAS
// Temp variable num_CAS = total number of customer access sequences
Begin
While (Parent NOT Found()) {
1 Visit left child of root node, make current_node = child_node
1.1 If count of current_node ≥ (s –e) * num_CAS, then
// continue algorithm 7

1.2 save pattern from root to current_node
1.3 Visit left child of current_node, make current_node = child_node,
Repeat process from 1.1 until 2.1 is true
2.1 Else if current_node == leaf_node or count of current_node < (s –e) * num_CAS
2.2 Move up to its parent
If Parent Found(), make current_node = parent_node
2.3 Check current_node has any siblings
2.3.1 If sibling Found ()
2.3.2 Visit the sibling, make current_node = sibling_node
// continued algorithm 5
2.3.3 Repeat from 1.1 until 2.1 is true
2.3.5 Else if sibling NOT Found ()
2.3.5 Repeat from 2.2 to until end of the loop.}
End

Algorithm 8: Maintain FSP-tree and d_list structure
Maintain_Structures (Size)
Input: Size
Output: updated d_list and FSP-tree T
Begin
// Perform Suffix Pruning II on FSP-tree at boundary
//Footer_list() is a list that maintains linkage to each leaf_node of T
//Variable h points to the head of Footer_List()
While (h != Null) {
1 Find leaf_node using h of T, then current_node = leaf_node
1.1 If count of current_node < num_CAS* e, then
delete current_node, move up to its parent, make current_node = parent_node,
continue until 1.2 is true
1.2 If count of current_node ≥ num_CAS* e or current_node == root of T
Then h = h.next
1.3 Find next leaf_node using h of T, make current_node = leaf_node}

//Sequential Pruning
//Temp variable y = 0;
While (!d_list[size]){
1 Find start_node from d_list [y], make current_node = start_node
While (current_node != Null) {
1.1 if count of current_node < num_CAS* e, then
1.2 delete current_node, make current_node = current_node.next
1.3 Else
1.5 make current_node = current_node.next} y = y +1;}
End

4 Experimental Evaluation and Performance Analysis

4.1 FP-Stream VS SSM-Algorithm

The concept of SSM-Algorithm came from FP-Stream [Giannella et al.2003]. The differences of FP-Stream and SSM-Algorithm are as follows:

1) FP-Stream mines frequent patterns. Time sensitive queries can be performed using FP-Stream. On the other hand, SSM-Algorithm is used for finding frequent sequential patterns. FP-Stream uses FP-tree [Han et al.2004] to mine frequent patterns. SSM-Algorithm uses PLWAP-tree [Ezeife & Lu2005] to mine frequent sequential patterns.

2) FP-Stream maintains an ordering list f_list is created in which items are by decreasing frequencies (as done in [Han et al.2004] [Han, Pie & Yin2000] this list). SSM maintains a dynamic structure named d_list that stores items and frequencies without any orders. A hash chain based indexing is used in d_list to boost performance. When incoming elements arrive, d_list gets updated using hash function and at the same time the eligibility of updated elements gets checked to see whether it can be considered as large 1-sequences.

3) FP-Stream forms a batch after 15 minutes elapsed time. The batch size changes dynamically in SSM-Algorithm. We gave a range for a batch, in other words, the algorithm constantly forms bigger batches at the pick time and at the slow time, it forms smaller batches by staying within the range to utilize resources efficiently.

4.2 Algorithm Analysis

We compare FP-Stream with our proposed algorithm. Say, batch B_1, (Table 4.2-1) as an input data for both algorithms and Table 4.2-2 shows the comparisons in between two algorithms from various

100	a b c d
200	a b d c
300	b c e f
400	b a g c

Table 4.2- 1: A Batch with 4 transactions

aspects. Based on the comparisons, the conclusion can drawn that computation time for mining of FP-Stream will constantly go higher than the computation time of SSM-

	FP-Stream	SSM-Algorithm
Index Mechanism for stream items	No	Yes
Pattern	Frequent Pattern	Frequent Sequential Pattern
Batch Size	Fixed (15 minutes wait)	Dynamic
Time Sensitive Queries	Yes	No
Batch Scan	2	2
Construct tree	1	1
Construct Intermediate Trees	7	0
Save and Calculate Intermediate DB	7	0
Calculate Frequent Event in Conditional patterns	7	/
Position Coding	No	Yes

Table 4.2- 2: Comparisons in between FP-Stream and SSM-Algorithm

Algorithm when both FP-Stream and SSM-Algorithm will be running simultaneously for mining more batches (B_2,B_3,B_4,.....B_n) using similar data.

4.3 Experimental Setup

In this chapter, we report the performance of our algorithm. SSM is written in Java. The experiment is performed on a 2.8 GHz (Celeron D processor) machine with 512 MB main memory. The operating system is Windows XP professional.

4.3.1 Dataset

The synthetic datasets are generated using the publicly available synthetic data generation program of the IBM Quest data mining project at: http://www.almaden.ibm.com/software/quest/. A data loader program is incorporated

with SSM to loads datasets into the Buffer from the source. The loader loads a transaction, waits 1 ms and then load the next transaction. Below is the description of the dataset.

The parameters shown below are used to show datasets.

[T] = Number of transactions

[S] = Average Sequence length for all transactions

[I] = Number of distinctive items

For example, T20K,S5,I1K represents 20000 transactions, 5 is average sequence length of all transactions, 1000 distinctive items.

4.3.2 Experimental Results

The test was performed by running a series of experiments using five different datasets (T10K,S3,I2K, T20K,S3,I2K, T40K,S3,I2K, T60K,S3,I2K, T80K,S3,I2K}. The first

	SSM-Algorithm		FP-Stream Alg.	
Dataset	**Average CPU Time per batch (sec)**	**Total CPU Time (sec)**	**Average CPU Time per batch (sec)**	**Total CPU Time (sec)**
T10K,S3,I2K	4.85	9.7	7.25	14.5
T20K,S3,I2K	4.5	18.03	6.5	26
T40K,S3,I2K	4.37	35.01	5.75	46
T60K,S3,I2K	6.25	75	11.66	139.92
T80K,S3,I2K	5.69	91.04	10.56	168.99

Table 4.3- 1: Result against support s= .0045 (.45%) and error e = .0004(.04%)

dataset has 10K transactions, average transaction sequence length is 3 and 2000 distinctive items. Similar pattern follows rest of the datasets with higher number of transactions. User defined support is set to .0045 (.45%), .004 (.4%), and .0035 (.35%) and minimum support error is 1/10 of each support. For testing, the support had to lower

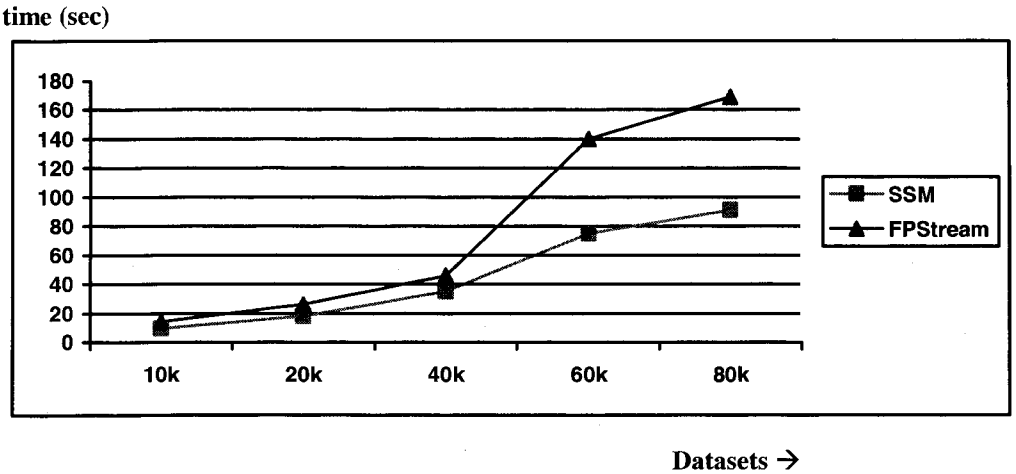

Figure 4.3- 1: Result against support s= .0045 (.45%) and error e = .0004(.04%)

to 1% because there are no items in the datasets that have support 1%. Table 4.3-1, Figure 4.3-1 and Figure 4.3-2 show the computation time in between two algorithms using support s = .0045 (.45%) and error e = .0004(.04%). SSM requires less time than FP-Stream. The reason behind is, SSM-Algorithm uses PLWAP-tree structure and

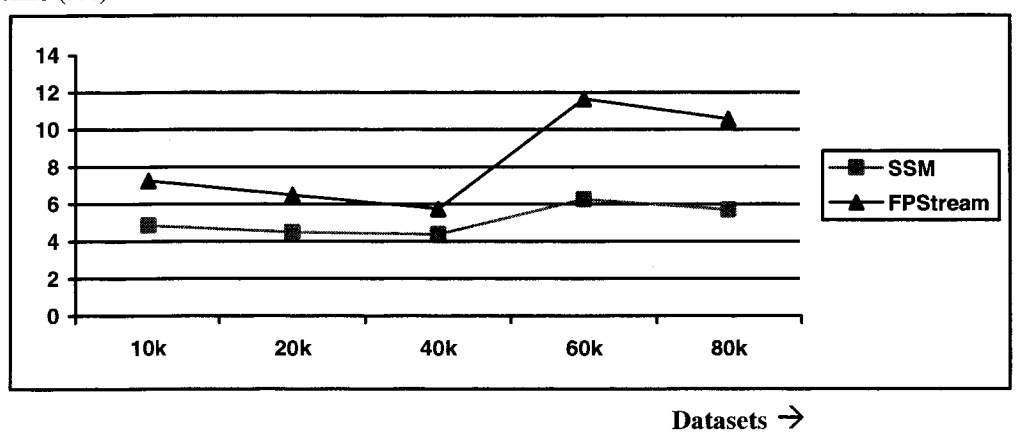

Figure 4.3- 2: Result for average batch with support s= .0045 (.45%) and error e = .0004(.04%)

PLWAP-Algorithm to generate patterns. The main advantage of using PLWAP-Algorithm over FP-Growth is PLWAP does not require to construct intermediate trees to mine frequent sequential patterns. It needs to compare position code nodes through the linkage queue with that of nodes in the roots set. On the other hand, FP-Growth algorithm

saves conditional patterns and then read these patterns to construct intermediate trees. For this reason, FP-Growth requires more storage, more computation time than PLWAP.

	SSM-Algorithm		FP-Stream Alg.	
Dataset	**Average CPU Time per batch (sec)**	**Total CPU Time (sec)**	**Average CPU Time per batch (sec)**	**Total CPU Time (sec)**
T10K,S3,I2K	5.61	11.23	8.75	17.5
T20K,S3,I2K	5.75	23.02	7.5	30
T40K,S3,I2K	5.43	43.44	7	56
T60K,S3,I2K	6.66	80	11.83	142
T80K,S3,I2K	6	96.04	11.54	184.64

Table 4.3- 2: Result against support s= .004 (.4%) and error e = .0004(.04%)

For both algorithms, the average time of batches varies batch to batch. It does not go higher constantly. From Figure 4.3-2, we can say that average time of a batch is dependant on the data of the datasets. It does not relate to the size of the datasets. In this

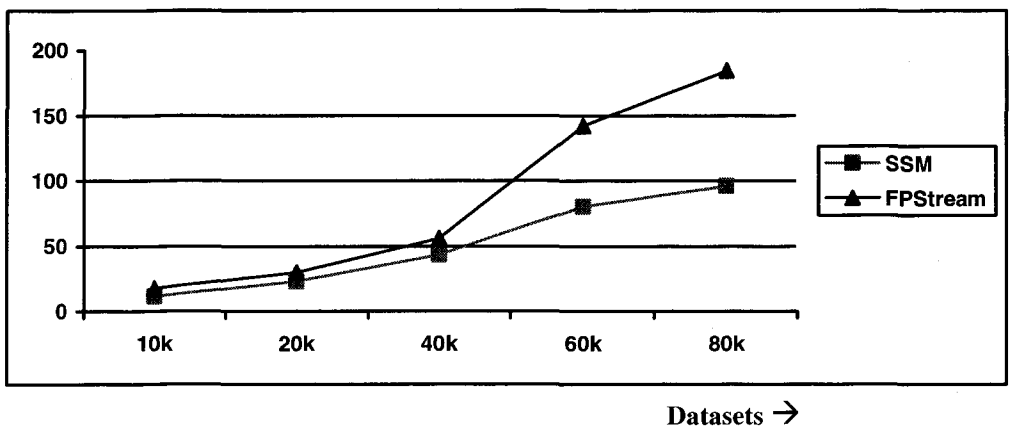

Figure 4.3- 3: Result against support s= .004 (.4%) and error e = .0004(.04%)

experiment a batch is holding approximate 5000 transactions.

Dataset	SSM-Algorithm		FP-Stream Alg.	
	Average CPU Time per batch (sec)	Total CPU Time (sec)	Average CPU Time per batch (sec)	Total CPU Time (sec)
T10K,S3,I2K	6.06	12.12	10.56	21.12
T20K,S3,I2K	6.81	27.27	11.77	47.08
T40K,S3,I2K	6.9	55.27	12.55	100.4
T60K,S3,I2K	7.0	84.02	12.44	149.28
T80K,S3,I2K	6.93	111.01	12.23	195.68

Table 4.3- 3: Result against support s= .0035 (.35%) and error e = .0003(.03%)

By looking at Figure 4.3-1, 4.3-3, and 4.3-4, it is clear that if support increase, the computation time reduces. It happens because of the datasets. If datasets have plenty of distinctive items and similar items are less, in that case, if support increases, the generation of candidate 1-sequnecs reduces, for the same reason the process finds less large 1-sequences. Thus tree construction and computation is quicker. This applies for both of the algorithms.

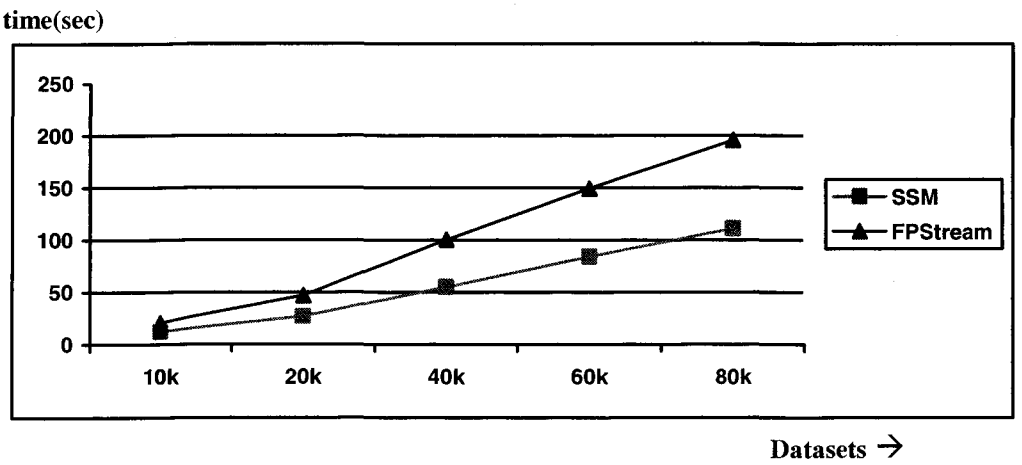

Figure 4.3- 4: Result against support s= .0035 (.35%) and error e = .0003(.03%)

4.3.2 Correctness of Algorithm Implementations

To show that the implementations of three algorithms are correct, we used a small database (Table 4.3-4) is used to test the correctness of implementations.

Trans ID	Customer Access Sequences	Frequent Sequences (FS)	Frequent Ordered Items
100	10 20 30 40	10 20 30	20 30 10
200	10 20 40 30	10 20 30	20 30 10
300	20 30 50 60	20 30	20 30
400	20 10 70 30	20 10 30	20 30 10

Table 4.3- 4: Database

As shows in Table 4.3-4, database includes 4 transaction IDs (100 through 400, first column of the Table 4.3-4), the second column of the database shows the events (total 16 events) in it. The third column of the Table 3.4.-4 is showing frequent subsequences (FS) that are generated after finding large 1-sequences from Table 3.4-4. We use FS for sequential mining. The fourth column is showing frequent ordered items in the above table that is used for frequent pattern mining. The minimum support of the database is set to .75 or 75%.

After running three algorithms against the database (in Table 4.3-4), the results are shown in Table 4.3-5. The results of PLWAP and SSM are same because they use similar

Algorithm	Result or Patterns
SSM	{{10}, {20}, {20, 30}, {30}
PLWAP	{{10}, {20}, {20, 30}, {30}
FP-Stream	{{20}, {20,30}, {20,30,10}, {20,10}, {30}, {30,10}, {10}}

Table 4.3- 5: Results from algorithms

algorithm. As PLWAP [Ezeife and Lu2005] is already an established algorithm and SSM generates same result, therefore, we claim that the implementations are correct.

On the other hand, result from FP-Stream is different. Because this algorithm constructs FP-tree using frequent ordered items instead of frequent sequences. Sequential mining method keep customer access sequences intact, therefore, the pattern (20,30,10) and (30, 10) were not generated by SSM and PLWAP but included in FP-Stream result.

5 CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

This thesis proposes a new algorithm (SSM-Algorithm) to mine frequent sequential patterns in data streams. SSM-Algorithm uses three types of data structures (d_list, PLWAP and FSP-tree) in order to handle and mine frequent sequential patterns in data streams efficiently and dynamically.

D_list is a complete dynamic data structure that stores incoming candidate 1-sequences. It grows bigger with incoming data streams. Hash based indexing is implemented to speed search and update process of this list. D_list is very useful for finding current large 1-sequences (L_1) and we need L_1 to generate current frequent patterns.

A continuous automated batch mining process is implemented to discover sequential patterns so that a small amount of memory will be used constantly to handle unbounded data in data streams.

An existing mining algorithm and structure PLWAP [Ezeife & Lu2005] is used in SSM to generate sequential patterns for incoming batches. PLWAP fits well in SSM and it avoids several intermediate tree reconstructions in order to save I/O computation time and storage usage. SSM takes the advantage of PLWAP-tree [Ezeife & Lu2005].

Sequential pattern tree (FSP-tree) gets updated incrementally with incoming sequential patterns from batches. FSP-tree is simple form of Pattern-tree [Giannella et al.2003] that is able to deliver current result on demand. This is a compact tree and stores only elements that are above the threshold to deliver quick result.

Suffix Pruning I, Suffix Pruning II and Sequential Pruning techniques are introduced in this thesis. The proposed pruning techniques are used in the algorithm very efficiently to maintain the size of the structures in the memory.

SSM-Algorithm is introduced to support mining tasks constantly for new applications where the primary source of the data is click stream data. It is a complete system that fulfills all of the requirements for mining frequent sequential patterns in data streams. SSM-Algorithm can be deployed for mining ecommerce's click stream data.

5.2 Future Work

1). There are some important issues can be incorporated with sequential mining in data streams like GSP algorithm [Srikanth & Agrawal1996]. Sliding window techniques can be added to SSM to create batches and candidate 1-sequences. Sliding window method will save batch creation and candidate generation time.

SSM-Algorithm generates frequent sequential patterns based on frequency of items. In other words, it uses only one dimension (frequency) to do its task. It is possible to add multiple dimensions (e.g. time dimension) or constraints along with frequency to discover interesting patterns in data streams.

2). In chapter 3 (section 3.2) of this thesis, we envisioned XYZ e-commerce's site and intended to mine XYZ's click stream data. Now, it would be very challenging to mine a several company's click stream data at the same time to find correlations. In other words, a multiple streams mining techniques at the same time to discover interesting patterns.

3). Rate (R) of incoming data stream is a constraint of stream mining. Mining rate has to be greater than or equal to incoming data stream rate. Otherwise, stream mining will fall behind. An advanced algorithm and method that overcomes Rate constraint and guarantees to support highest bandwidth could be another future work.

4). Proposed SSM method generates sequential patterns through batch mining process. In other words, it groups customer access sequences and applies PLWAP-Algorithm [Ezeife &Lu2005] to generate frequent sequential patterns. Then, it initializes existing PLWAP-tree to empty set. When next batch arrives, it reconstructs another PLWAP-tree and applies PLWAP-Algorithm on new batch to generate frequent sequential patterns. It is an ongoing process. Now, it might be possible to keep first PLWAP-tree and update incrementally with incoming batches rather than constructing a new PLWAP-tree for each batch. However, this is not definite that it will work in stream environment because this PLWAP-tree update process would be required to keep the information of all small items and their possible position in the tree. Stream mining does not follow this kind of

approach. Thus, the solution of updating main PLWAP-tree for all of the batches would be very challenging and could be a possible future work in this direction.

6 References:

[Agrawal, Imielinski, & Swami1993] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining Association rules between sets of items in large databases. *In Proc. Of the ACM SIGMOD conference on management of data*, Pages 207 – 216, Washington, DC, 1993

[Agrawal & Srikant1994] Rakesh Agrawal, Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules, *Proceedings of the 20th VLDB conference*, Pages 487-499, Santiago, Chile, 1994

[Agrawal & Srikanth1995] Rakesh Agrawal, Ramakrishnan Srikanth. Mining Sequential Patterns, *Research Report, IBM Almaden ResearchCenter 650 Harry Road, San Jose, CA 95120, IEEE Publication*, Pages 1-22, 1995

[Ansari et al.2001] Suhail Ansari, Ron Kohavi, Llew Mason, Zijian Zheng. Integrating e-commerce and data mining: architecture and challenges, *Proceedings IEEE International Conference on Data Mining*, Pages: 27-34, San Jose, CA , USA, 2001

[Berkhin, Becher, & Randall2001] Pavel Berkhin, Jonathan D. Becher, Dee Jay Randall. Interactive Path Analysis of web Site Traffic, *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining table of contents*, Pages 414 – 419, San Francisco, California, USA, 2001

[Buchner & Mulvenna1998] Alex G. Buchner, Maurice D. Mulvenna. Discovering Internet Marketing Intelligence through Online Analytical Web Usage Mining, SIGMOD Record, Vol.27, No.4. Pages 54 – 61, New York, NY, USA, 98

[Chandrasekaran & Franklin2002] Sirish Chandrasekaran, Michael J. Franklin. Streaming Queries over Streaming Data, *In Proc. Of the 2002 VLDB Conference, 28th International Conference on Very Large Databases (VLDB)*, Honk Kong, China 2002

[Chen et al.2002] Y. Chen, G. Dong, J. Han, W. B. Wah, J. Wang. Multidimensional regression analysis of time-series data streams, *proceedings of the 28th VLDB conference,* pages 323-334, Hong Kong, China, 2002

[Cooley, Mobasher & Srivastava1997] R. Cooley, B. Mobasher, J. Srivastava. Web Mining: Information and Pattern Discovery on the World Wide Web, *Proceedings of the ninth IEEE International Conference on Tools with Artificial Intelligence*, Pages 558 – 567, Newport Beach, CA, USA, 1997

[Cooley, Mobasher & Srivastava1999] R. Cooley, B. Mobasher, J. Srivastava. Data preparation for mining world wide web browsing patterns, *Knowledge and Information Systems*, 1(1), Pages 1-26, 1999

[Domingos & Hulten2000] P. Domingos and G. Hulten. Mining high-speed data streams. *proceedings of the 2000 ACM SIGKDD Int. Conf. knowledge Discovery in Database (KDD'00),* Pages 71-80, 2000

[Dutta et al.2001] Kaushik Dutta, Debra VanderMeer, Anindya Datta, Krithi Ramamritham. *Proceedings of the 3rd ACM conference on Electronic Commerce*, pages 65–74, Tampa, FL, USA, 2001

[Ezeife & Chen2004a] C.I.. Ezeife and Min Chen, Mining Web Sequential Patterns Incrementally with Revised PLWAP Tree, *proceedings of the fifth International Conference on Web-Age Information Management (WAIM 2004) Dalian*, published in LNCS by Springer Verlag. UK, June 2004

[Ezeife & Chen2004b] C.I. Ezeife and Min Chen, Incremental Mining of Web Sequential Patterns Using PLWAP Tree on Tolerance MinSupport, *proceedings of the IEEE 8th International Database Engineering and Applications Symposium*, Coimbra, Portugal, July 7th to 9th, 2004.

[Ezeife & Lu2005] C.I. Ezeife and Yi Lu, Mining Web Log sequential Patterns with Position Coded Pre-Order Linked WAP-tree, *the International Journal of Data Mining and Knowledge Discovery (DMKD),* Vol. 10, No. -, pp. 5-38, Kluwer Academic Publishers, June 2005

[Giannella et al.2003] C. Giannella, J. Han, J. Pei, X. Yan and P.S. Yu. Mining Frequent Patterns in Data Streams at Multiple Time Granularities, H. Kargupta, A. Joshi, K. Sivakumar, and Y. Yesha (eds.), *Next Generation Data Mining*, 2003.

[Guna et al.2000] S. Guna, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. *proceedings of IEEE Symposium on Foundations of Computer Science (FOCS'00),* Pages 359-366, 2000

[Guna et al.2003] S. Guna, A. Meyerson, N. Mishra, and R. Motwani. Clustering data streams: *Theory and Practice TKDE special issue on clustering*, vol. 15, 2003

[Gunduz & Ozsu2003] S. Gunduz, M. T. Ozsu. A web page prediction model based on click-stream tree representation of user behavior, SIGKDD 2003, Page 535-540

[Han & Kamber2000] J. Han and M. Kamber. Data Mining: Concepts and Techniques, Morgan Kaufmann, 2000

[Han, Pie & Yin2000] Jiawei Han, Jian Pei, Yiwen Yin. Mining frequent patterns without candidate generation, *Proceedings of the 2000 ACM SIGMOD international conference on Management of data and Symposium on Principles of Database Systems,* Pages 1 –12, Dallas, Texas, USA, 2000

[Han et al.2004] J. Han, J. Pei, Y. Yiwen, R. Mao. Mining frequent patterns without candidate generation: a frequent pattern tree approach, *Data Mining and Knowledge Discovery, 8, 1*, Page 53-87, 2004

[Hin2002] Phillip Hingston. Using Finite State Automation for Sequence Mining, *Proceeding of the twenty-fifth Australasian conference on computer science – volume 4*, Pages 105 – 110, Melbourne, Victoria, Australia, 2002

[Hu & Cercone2002] Xiaohua Hu, Nick Cercone. An OLAM framework for Web Usage Mining and Business Intelligence reporting, *Proceedings of the 2002 IEEE International Conference on Fuzzy Systems FUZZ-IEEE'02, Volume: 2*, Pages 950 –955, 2002

[Karp, Shenker, & Papadimitriou2003] R. Karp, S. Shenker, and C. Papadimitriou. A Simple Algorithm for Finding Frequent Elements in Streams and Bags. *ACM Transactions on Database Systems*, 28(1):51--55, 2003

[Keahey & Eick2002] T. A. Keahey, S. G. Eick. Visual Path Analysis, *Proceedings of the IEEE symposium on Information Visualization*, Pages 165-168, 2002

[Kosala & Blockeel2000] R. Kosala and H. Blockeel. Web mining research: a survey, *ACM SIGKDD Explorations, 2. 1, ACM SIGKDD*, 2000

[Kurose & Ross2003] James F. Kurose, Keith W. Ross. *Computer Networking, 2nd Edition, Addison Wesley Longman, Inc, 2003*

[Las2002] M. Last, Online classification of nonstationary data streams, *Intelligent Data Analysis, Vol. 6, No. 2,* Page 129-147, 2002

[Lee & Podlaseck2000] Juhnyoung Lee, Mark Podlaseck. Using a Starfield Visualization for Analyzing Product Performance of Online Stores, *Proceedings of the 2nd ACM conference on Electronic commerce table of contents*, Pages 168 – 175, Minneapolis, Minnesota, United States, 2000

[Lin et al.2003] J. Lin, E. Keogh, S. Lonardi, and B. Chiu. A symbolic representation of time series, with implication for streaming algorithms. In Proc. of the 8th ACM SIGMOD workshop on research issues in data mining and knowledge discovery, San Diego, USA, 2003

[Liu & Chang2004] Bing liu, Kevin Chang. Special issue on web content mining, *ACM SIGKDD Explorations, Newsletter, Volume 6 , Issue 2,* pages 1-4, 2004

[Liu, Chen & Song2002] Lizhen Liu, Junjie Chen, Hantoa Song. The Research of Web Mining, *Proceedings of the 4th World Congress on Intelligent Control and Automation, vol.3*, Pages 2333- 2337, Beijing, China, 2002

[Lu & Ezeife2003] Y. Lu and C. I. Ezeife. Position Code Pre-Ordered Linked WAP-Tree for Web log sequential pattern mining, *In Proceedings of the 7th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2003),* Seoul, Korea, 2003

[Mah, Hoek & Li2001] Teresa Mah, Hank Hoek, Ying Li. Funnel Report Mining fro the MSN Network, *Proceedings of the seventh ACM SIGKDD international conference on Knowledge Discovery on Data,* Pages 450 – 455, San Francisco, California, USA, 2001

[Manku & Motwani2002] Gurmeet Singh Manku and Rajeev Motwani, Approximate frequency counts over data streams, *proceedings of the 28th VLDB conference,* Hong Kong, China, 2002

[Masseglia, Teisseire & Poncelet2001] Florent Masseglia, Maguelonne Teisseire, Pascal Poncelet, Real Time Web usage Mining: a Heuristic based Distributed Miner, *Proceedings of the second International conference on Web Information Systems Engineering, Volume: 1,* Pages 288-297, 2001

[Masseglia, Teisseire & Poncelet2002] Florent Masseglia, Maguelonne Teisseire, Pascal Poncelet, Real Time Web usage Mining with a Distributed Navigation Analysis, *Proceedings of the twelfth International Workshop on Engineering E-Commerce/E-Business Systems,* Pages 169-174, San Jose, CA, USA, 2002

[Nahm, Bilenko & Mooney2002] Un Yong Nahm, Mikhail Bilenko, Raymond J. Mooney. Two approaches to handling Noisy Variation in Text Mining. *In proceedings of the ICML-2002 Workshop on Text Learning (TextML '2002),* pages 18-27, Sydney, Australia, July 2002

[Pei et al.2000] Jian Pei, Jiawei Han, Behzad Mortazavi-asi, Hua Zhu. Mining Access Patterns Efficiently from web logs, *Proceedings 2000 Pacific-Asia conference on Knowledge Discovery and data Mining,* Pages 396-407, Kyoto, Japan, 2000

[Srikanth & Agrawal1996] Ramakrishnan Srikanth and Rakesh Agrawal. Mining Sequential Patterns: generalizations and performance improvements, *Research Report, IBM Almaden ResearchCenter 650 Harry Road, San Jose, CA 95120,* Pages 1-15, 1996

[Srivastava et al.2000] J. Srivastava, R. Cooley, M. Deshpande, P. N. Tan. Web usage mining: Discovery and applications of usage patterns from web data, *SIGKDD Explorations, Volume 1, Issue 2,* Pages 12 –23, 2000

[Teng, Chen & Yu2003] W. Teng, M. Chen, P. Yu. A regression-based temporal pattern mining scheme for data streams, *In proceedings of the 29th VLDB conference,* Berlin, Germany, 2003

[Theusinger & Huber2000] Christiane Theusinger, Klaus-Peter Huber. Analyzing the foot steps of your customers, A case study by ASK-net and SAS Institute GmbH, *Proceedings of the WebKDD-2000 conference*, 2000

[Tuzhilin & Adomavicius2002] Alexander Tuzhilin, Gediminas Adomavicius. Handling very large numbers of association rules in the analysis of microarray data, *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data minin,* pages 396-404, Edmonton, Alberta, Canada, 2002

[Woon, Ng, & Lim2002] Yew-Kwong Woon, Wee-Keong Ng, Ee-Peng Lim. Online and Incremental mining of Separately – Grouped Web Access Logs, *Proceedings of the Third International Conference on Web Information Systems Engineering*, Pages 53- 62, Singapore, Singapore, 2002

[Xiao & Zhang2001] Jitian Xiao, Yanchun Zhang. Clustering of Web Users Using Session-based Similarity Measures, *Proceedings of the 2001 International Conference on Computer Networks and Mobile Computing*, Pages 223 – 228, Los Alamitos, CA , USA, 2001

[Xie & Phoha2001] Yunjuan Xie, Vir V. Phoha. Web User Clustering from Access Log Using Belief Function, *Proceedings of the international conference on Knowledge Capture*, Pages 202 – 208, Victoria, BC, Canada, 2001

[Yang, Haining, & Li2001] Qiang Yang, Haining Henry Zhang, Tianyi Li. Mining web logs for prediction models in WWW caching and prefetching, *Proceedings of the ACM SIGKDD International conference, ACM Press*, 2001

[Zhang & Chang2002] Feng Zhang, Hui-You Chang. Research and Development in Web Usage Mining System-key Issues and Proposed Solutions: A survey, *Proceedings 2002 International Conference on Machine Learning and Cybernetics*, Volume 2, Pages 986-990, 2002

VITA AUCTORIS

Mostafa Monwar was born in 1972 in Sylhet, Bangladesh. He completed his bachelor of science in honors in computer information systems at Strayer University, Washington, DC, USA in 1998. He also worked as a systems support specialist at Intel Corporation from 1999 to 2001. He is currently a candidate for the master's degree in computer science at the University of Windsor and expects to complete the degree in September 2005.