Constraint-Based Frequent Itemset Mining from Data Streams

by

Quamrul Islam Khan

A thesis submitted to the Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada

June 2006

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES
*****
COPYRIGHT PERMISSION

Constraint-Based Frequent Itemset Mining from Data Streams

BY

Quamrul Islam Khan

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

OF

MASTER OF SCIENCE

Thesis advisor

Dr. Carson K. Leung

Author

Quamrul Islam Khan

Constraint-Based Frequent Itemset Mining from Data Streams

Abstract

In recent years, the problem of frequent itemset mining has evolved into a new form called *data stream mining*. In it, a continuous stream of data is mined or analyzed to find frequent itemsets. The continuous nature of streams brings new challenges to (traditional) frequent itemset mining. Although some algorithms have been proposed to mine frequent itemsets from data streams, they do not provide user control over the mining process through the use of constraints. Constraint-based mining can achieve the goal of user exploration. Moreover, various pruning techniques for constraint-based algorithms can be utilized in the stream mining process to make it more efficient.

In this thesis, we develop an algorithm for constraint-based frequent itemset mining from data streams. Specifically, we propose a novel tree structure, called Data Stream Storage Tree (*DSSTree*), to efficiently store and process incremental updates of streaming data. Experimental results show that our algorithm enables users to have a better control on the mining process and to extract interesting frequent itemsets from continuous streams of data efficiently.

Acknowledgements

I would like to express my deepest gratitude to Dr. Carson K. Leung, my research supervisor, for his support (both academically and financially). Dr. Leung gave me the opportunity to work as his research assistant, which helped me improve my research skills and gave me insight on this research project.

In addition, I would also like to thank Department of Computer Science for funding me in the form of a teaching assistantship and providing me with quality computer-lab facilities to conduct my thesis work. I would also like to thank TRLabs Winnipeg for supporting me in the form of a graduate student fellowship and for giving me the opportunity to work on one of their research projects.

My special thanks to my thesis examination committee members, Dr. Jeffrey E. Diamond and Dr. Ruppa K. Thulasiram, and the chair of my thesis defence, Dr. Dean Jin.

Moreover, I would also like to thank members of the Graduate Studies Committee in the Department of Computer Science for their useful comments and suggestions on my thesis proposal.

Last but not the least, all praises to Allah Subhana-wa-tala (God) for His blessing and bestowing me with strength and knowledge to complete my research work.

QUAMRUL ISLAM KHAN
B.S., North South University, Bangladesh, 2000

The University of Manitoba
June 2006

To my parents and my wife.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Data mining aims to search from existing data for information that is potentially useful, implicit, and previously unknown [FPM91]. Common data mining tasks include association rule mining, clustering, classification, outlier detection, and sequential pattern mining. Among these tasks, we focus on the task of association rule mining. The main goal of association rule mining [AIS93] is to process a collection of transactions in a transaction database TDB and generate a set of association rules of the form $X \Rightarrow Y$ that meets certain thresholds. To elaborate, each transaction $trans \subseteq Items$, where $Items = \{i_1, i_2, \cdots, i_k\}$ is the set of k distinct items in TDB. In a rule, X and Y are sets of items (also known as *itemsets*), where (i) $X \cap Y = \emptyset$ and (ii) both X and Y are subsets of one or more transactions in TDB [AS94]. Users can define two thresholds, namely the *support threshold* and the *confidence threshold*, to obtain the rules that are of interest to them. For a rule to satisfy these two thresholds, the following conditions must be met. First, itemsets X and Y must be *frequent*, which means that both X and Y have frequency counts (i.e., the number of

occurrence of itemsets in TDB) greater than or equal to the user-defined minimum support threshold *minsup*. Second, the confidence of the rule $X \Rightarrow Y$, which is defined as $confidence = (frequency(X \cup Y))/frequency(X)$, must be greater than or equal to the user-defined minimum confidence threshold *minconf*. Once the frequent itemsets X and Y are found (which means that the frequencies of itemsets X and Y are counted), it is not difficult to check the confidence and form the rules [Hid99]. This explains why in the problem of association rule mining, researchers usually focus on the sub-problem of finding frequent itemsets (instead of the sub-problem of forming rules).

Frequent itemset mining can be and has been extended in many directions, including the following: (i) *closed itemset mining* [PBTL99, PHM00, ZH02, GZ03b], (ii) *maximal itemset mining* [Bay98, GZ01, GZ03b, Yan04], and (iii) *constraint-based frequent itemset mining* [NLHP98, LLN02, LLN03, Leu04].

Closed itemset mining aims to find closed itemsets. A **closed itemset** is a frequent itemset satisfying an additional condition. Specifically, a frequent itemset CP is **closed** if there does not exist any proper superset CP' of CP such that $frequency(CP') = frequency(CP)$. With this additional condition, the number of closed itemsets generated is lower than that of frequent itemsets [WHP03].

Similarly, *maximal itemset mining* aims to find maximal itemsets. A **maximal itemset** is a frequent itemset satisfying a condition. Specifically, a frequent itemset MP is **maximal** if there does not exist any frequent proper superset of MP. Hence, with this condition, the set of MPs generated is even smaller than the set of CPs generated [GZ03b].

It can be easily observed that a closed or a maximal itemset is a compact representation of some (regular) frequent itemsets. For example, a maximal itemset $\{a, b\}$ can be considered as a compact representation of three frequent itemsets (namely, $\{a\}, \{b\}, \{a, b\}$).

In contrast, constraint-based frequent itemset mining does not find compact representations (e.g., closed or maximal itemsets). Instead, it finds itemsets that are of interest to the user (i.e., frequent itemsets satisfying the user-defined constraints). Hence, the extensions of mining mentioned above are complementary in the sense that one can perform the *constraint-based closed itemset mining* or the *constraint-based maximal itemset mining*, which finds itemsets that represent a collection of itemsets (e.g., itemsets and their subsets) satisfying the user constraints.

In the constraint-based itemset mining, users are allowed to specify not only the support (i.e., frequency) and the confidence thresholds but also the constraints that express their interests. A simple constraint is of the form $S.Price\ \theta\ constant$, where S is an itemset, *Price* is a property related to the itemset (i.e., price of each item in the itemset S), and $\theta \in \{=, <, \leq, >, \geq\}$. There are many different properties of constraints that can help in (i) the generation or enumeration of itemsets that satisfy the constraints and (ii) the pruning or removal of itemsets that violate the constraints.

In general, (traditional) frequent itemset mining can be described as finding frequent itemsets from a *static* database. Over the past decade, many studies have been conducted on analyzing and improving frequent itemset mining and its extensions. Researchers have continuously explored new directions. A new direction is to find frequent itemsets *from data streams.* The key challenge is that we are no longer find-

ing itemsets from a static database but a continuous stream of data. In other words, the incoming transaction data (i.e., streaming data) is continuous, dynamic, and of possibly infinite length [GHP+02]. Mining from streaming data is of great demand in many real-world applications, such as wireless sensor networks, distributed sensor networks, network traffic monitoring systems, Web click tracking, anomaly detection, and stock market data analysis. For example, in the area of wireless sensor networks a large amount of data from the sensors is transmitted to a base station, which is a local station that coordinates the sensors and gathers data from them. To extract potentially useful information from these data in the base station, data mining is needed. As the incoming data to a base station are continuous in nature, the application of data stream mining is appropriate.

1.1 Motivation

Mining data from streams can be quite challenging. Firstly, the mining algorithm requires a method for dealing with incremental updates of the data, because new data are arriving from streams continuously, which can be scanned only once. Secondly, with new data continuously adding onto the existing data collection, the size of the data collection can grow excessively large and become infeasible to handle. Therefore, it would be more useful to capture only the recent data (which is more interesting than older data in many applications) from the streams. Finally, even when working with recent data, which can still be huge (e.g., millions of transactions), the mining process may generate a large number (e.g., hundreds of thousands) of output frequent itemsets. Out of those large set of frequent itemsets, users may be interested in a

subset of it. Thus, it is necessary to incorporate some techniques to prune unwanted or uninteresting frequent itemsets. In the remainder of this section, we will briefly discuss the previous developments to handle these issues in frequent itemset mining from the data streams.

Incremental mining approaches are appropriate to process the streaming data (i.e., data of continuous nature). There are some existing incremental mining algorithms. Some of them are tree based [GHP+02, CZ03, KS04, LLS04], while some others are level-wise itemset generation based [CLK97, ATA99]. Typically, tree based algorithms store items of the incoming transactions in a tree, with the items arranged in descending frequency order along the tree paths. In our research on incremental mining, we designed a canonical ordering based mining data structure, called *CanTree* [LKH05], which can efficiently mine from incrementally updated data.

However, as the data are continuously growing, it is not feasible to store all incoming transactions in the tree. This explains why many existing data stream mining algorithms use some types of windowing techniques, such as the *sliding window technique* [ZS02]. With the sliding window technique, only a window of recent data (containing n recent transactions) in the data stream is analyzed. The window slides over time to include new data, and exclude old data at the same time. The rationale for using this technique is that users are mostly interested in the recent changes or the current state of frequent itemsets, rather than some older data (or the entire history of data) [GHP+02]. In order to implement the sliding window technique, some methods are needed to obtain the frequency counts of items in the window and to remove those infrequent items. Again, due to the dynamic nature of a data stream, it is difficult to

prune items based on frequency counts, because an itemset that is infrequent at the current time may become frequent in the near future. This has led to the proposal of some approximation methods (e.g., [GHP+02, MM02, CL03, LLS04]) for estimating frequency counts.

Chi et al. [CWYM04] proposed an algorithm for finding closed itemsets (instead of finding frequent itemsets) from streaming data with the aim to reduce the number of frequent itemsets, and thus improving the performance. Although the use of closed itemset mining may reduce the number of itemsets returned to the user, these itemsets are not necessarily all interested to the user. Hence, it would be more useful if we apply constraints on the mining algorithm. The use of constraints in data stream mining can facilitate selection of only desired itemsets that are of interest to the user. *Constraint-based frequent itemset mining* not only reduces the output frequent itemsets, but also *generates frequent itemsets that are of interest to users.* Previous work on constraint-based mining [BAG98, NLHP98, PHL01, LLN02, LLN03, Leu04] significantly improved performance and usability of the traditional mining process (i.e., when not dealing with a data stream). To the best of our knowledge, there is no work on constraint-based frequent itemset mining from data streams.

The issues and possible solutions for frequent itemset mining from data streams lead to three questions: (i) How can an efficient incremental mining technique that handles continuous incoming data from stream be developed? (ii) What type of windowing technique can be used to capture the most recent data from data streams: the sliding window techniques or others? and (iii) How can users mine for frequent itemsets that are of interest to them? These questions motivated us to pursue this

thesis in developing a constraint-based algorithm for mining frequent itemsets from data streams. The thesis statement is as follows:

> We develop a constraint-based frequent itemset mining algorithm for a data stream environment.

1.2 Contributions

In this section, we outline how mining of frequent itemsets from data streams can be performed to meet the three research questions (mentioned in the previous section) along with our research contributions.

To handle the incremental nature of the data stream, we design a data structure, called a Data Stream Storage Tree (*DSSTree*), which is an extension of the *CanTree* [LKH05]. The *DSSTree* uses the interesting idea of a canonical item ordering scheme in the tree. This scheme has been found efficient in our work [LKH05], which handles incremental updates of the tree structure. Nevertheless, incrementally updating the data is not sufficient, because data in streams are continuous and can be of infinite length. Therefore, the *DSSTree* has to capture only recent data. To resolve this, our algorithm uses a sliding window technique [ZS02] to capture the most recent data from data streams.

The *DSSTree* stores *all* the transaction information of the current window, thus it does not require any approximation method for frequency estimation. To clarify, why an approximation method is necessary? There are cases where some previously infrequent items become currently frequent. For example, if a previously infrequent item $\{X\}$ (which is currently frequent with the update of new data from streams) was

pruned, then we do not have the exact frequency count of $\{X\}$. This leads to the use of an approximation method to get an approximated frequency count of $\{X\}$. Most of the existing stream mining algorithms (e.g., [GHP+02, MM02, CL03, LLS04]) use an approximation method to handle this issue. In contrast, our *DSSTree* contains all the transactions in the current window, thus the algorithm does not prune any infrequent items. Hence, our algorithm does not use any approximation method; it has the actual frequency count of those items which changed from infrequent to frequent.

Furthermore, the data in the current window can be mined interactively allowing users to change the frequency threshold or any constraints. To elaborate, if we have the minimum support threshold (*minsup*) set to 5 (frequent count), then any items with frequency count below 5 are infrequent. Now, we mine with a new *minsup* set to 3, then some of those items that were infrequent in the previous run may become frequent. However, if we have pruned all the infrequent items from the previous run (with $minsup = 5$), then there is no way to get those items back during the stream mining process because we are allowed to scan the data once. As the *DSSTree* retains all frequent and infrequent itemsets, it can handle changes of any frequency threshold or any constraints.

So far, we have described the key components of the *DSSTree*, which handles the update of data from streams and can be incrementally mined. For mining the data, we still can have large number of output frequent itemsets, and not all of which is of interest to users. This issue is handled with the incorporation of constraint-based mining technique, which will reduce the output frequent itemsets and efficiently

generate frequent itemsets that are of interest to users. To be more specific, we explored single-variable constraints (e.g., *S.Price* θ *constant*) [NLHP98], because they are usually used to express typical user queries and capture typical user focus (e.g., the single-variable constraint "$S.Price \geq \$50$" expresses the user's intention to find sets of items with prices $\geq \$50$)[1].

To summarize, our key contributions are:

1. We introduce a novel stream mining data structure, called a *Data Stream Storage Tree* (*DSSTree*), which can efficiently update data from continuous data streams. The *DSSTree* also uses sliding window technique to capture only recent data from streams.

2. As the *DSSTree* maintains all the items (frequent or infrequent) in the current window, it *avoids the usage of approximation methods* to get frequency count of an item. This property also *enables interactive mining* on the streaming data.

To the best of our knowledge, our mining algorithm is the first one to incorporate constraint-based frequent itemset mining in data streams.

1.3 Thesis Outline

This thesis is organized as follows.

[1]Recall that the problem of association rule mining can be divided into two subproblems: the subproblem of finding frequent itemsets and the subproblem of forming rules. Single-variable constraints are usually used for the former, whereas multi-variable constraints are usually used for the latter. For example, the multi-variable constraint "$min(A.Price) > max(C.Price)$" can be used to express the user's intention of finding interesting association rules of the form $A \Rightarrow C$ such that prices of items in itemset A (in the antecedent of the rule) are higher than prices of items in itemset C (in the consequence of the rule). Given that we focus on the subproblem of finding frequent itemsets in this thesis, we use single-variable constraints.

Chapter 2 provides the background on some of the techniques applied in our algorithm and also discusses some of the relevant algorithms. We discuss some existing windowing techniques that can be used to capture recent data from data streams, and explain the usages and properties of some constraints that are incorporated with our data stream mining algorithm. Section 2.3 describes the frameworks of frequent itemset mining algorithms followed by algorithms on constraint-based mining, incremental mining, and stream mining. Our algorithm has components which involve all these forms of mining. We have also outlined the key differences in the existing streaming algorithms and ours.

After reviewing the previous work in Chapter 2, we start describing our current work (i.e., the contribution of this thesis) in Chapter 3. Specifically, Chapter 3 describes our proposed constraint-based stream mining algorithm. We describe the underlying architecture and design of the algorithm. We elaborate the features of the sliding window, the data stream mining data structure, and the application of constraints in the stream mining environment. We also describe our design of the following: (i) a novel data structure called Data Stream Storage Tree (*DSSTree*), which can efficiently handle incremental update of data from the continuous stream and mine frequent itemsets; (ii) a windowing technique to resolve the problem of capturing recent data from the incoming data streams; and (iii) the application of user-defined constraints in the data stream mining algorithm to mine output frequent itemsets that are of interest to users.

In Chapter 4, we articulate the implementation of the designed solution methodologies discussed in Chapter 3. We use pseudo-code to provide an in-depth expla-

nation of how the algorithms work, and also discuss some issues that we faced in the development process. These issues and their solutions are explained with examples. Furthermore, we describe the necessary algorithms for building and maintaining the stream mining data structure (*DSSTree*), followed by the mining algorithm with application of constraints.

In Chapter 5, we discuss the experimental evaluation of the implemented algorithm. We performed typical experiments for data mining algorithm evaluation, where we show the scalability, performance comparison with other existing algorithms, efficiency of stream mining (fast incremental update), and constraint-based mining (performance in terms of constraint selectivity).

Finally, we conclude in Chapter 6 with our research findings, and provide some ideas for extension and future work.

Chapter 2

Related Work

In this chapter, we discuss some background studies on three windowing techniques and a few properties of constraints. We describe the windowing techniques that can be applied to capture recent data from data streams. We also discuss different properties of constraints that we use to incorporate the features of constraint-based mining in data stream mining. Furthermore, we discuss several relevant data mining algorithms. In Section 2.3, we explore frameworks of frequent itemset mining algorithms, some incremental mining algorithms, constraint-based mining algorithms, and stream mining algorithms. We also outline some key advantages in terms of functionality that our algorithm provides when compared to existing streaming algorithms.

2.1 Background: Windowing Techniques

In the following, we describe three different types of *windowing techniques* [ZS02]: (i) *landmark window technique*, (ii) *sliding window technique*, and (iii) *tilted/damped*

window technique.

2.1.1 Landmark Window Technique

In the *landmark window technique*, the window starts from a given time-point (which is before than the current time) and ends at the current time. Alternatively, one can also set the size of the window as the entire database seen so far. Note that the starting point is fixed, but the end point (which is the current time) and the window size grow as time passes. In Figure 2.1, the landmark time-point has been set at t_{LM}. Let us consider the current time to be t_k. The current window consists of data received from the stream starting at t_{LM} till t_k. With the progress of time, the current time changes to t_{k+1}, which could be seconds, minutes, or hours after t_k. Therefore, with the change of time, the current window contains all data received from the stream starting from t_{LM} till t_{k+1}. The landmark time-point remains at t_{LM}, and thus increases the size of the window with the change in time. This makes the landmark window technique infeasible for stream mining as the incoming data

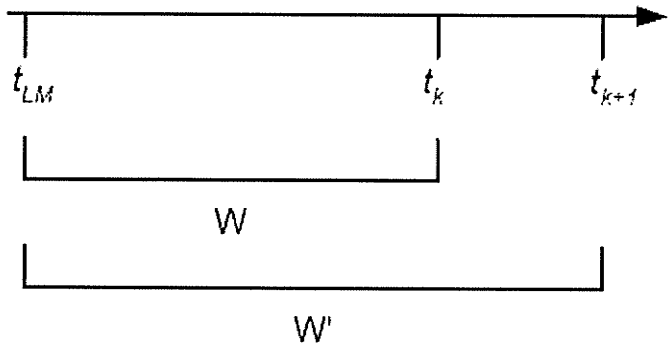


Figure 2.1: The landmark window technique

can be of infinite length. It may be infeasible to keep all the data from the stream in the available memory after some finite time period.

2.1.2 Sliding Window Technique

In the *sliding window technique*, the window size is fixed by users, and the window slides over periods of time. Hence, unlike the landmark window technique (which uses a variable-size window), the sliding window technique uses a *fixed-size* window that *starts* from time x (where, x = *current time* − *size of the window*) and ends at the current time. Figure 2.2 shows the shifting of window with time. In Figure 2.2, W is the fixed window size. For the first frame, the window starts at time-point t_{S1} and ends at time-point t_{E1}. After the shift, the current window starts at t_{S2} and ends at t_{E2}, where $t_{E1} - t_{S1} = t_{E2} - t_{S2} = W$. This windowing technique is appropriate for handling data streams for two reasons: (i) it captures data of a fixed window size (no memory overshoot) and (ii) the oldest block of data in the current window

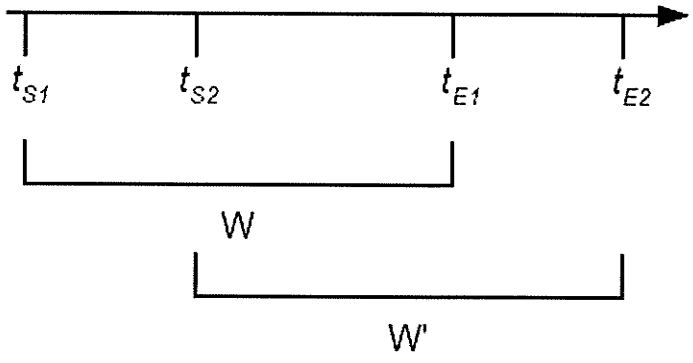


Figure 2.2: The sliding window technique

is discarded to accommodate new data (so the window size is fixed), which keeps consistent block of data in the current window.

2.1.3 Tilted/Damped Window Technique

The tilted/damped window technique considers that the recent block of data in the current window is more valuable than the old data blocks. So, there is a weight factor associated with each block of window, and the weight decreases over time. Thus, this technique reduces the contribution of older data, which becomes zero (and is discarded) after some period of time. In other words, the tilted/damped window technique is similar to that of the sliding window except that the weight factor determines whether the data will be included in the window or excluded. The shifting of this window is illustrated in Figure 2.3. The windows in Figure 2.3 are shown with smaller divisions, to define smaller units of data blocks within the windows. The empty blocks are newly arrived blocks; those with diagonal strips are the old blocks (which have been marked based on the weighing factor); and those with cross-diagonal

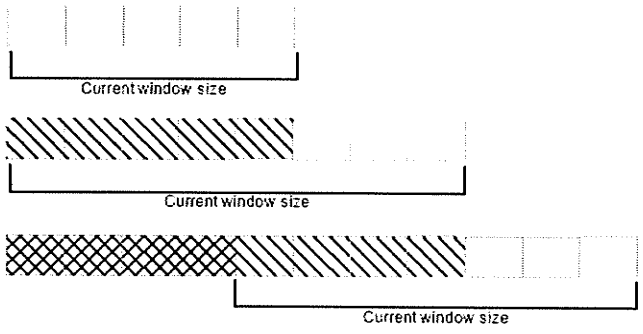


Figure 2.3: The tilted/damped window technique

strips are the pruned blocks (pruned as the weights on these blocks went below a threshold or became zero). This windowing technique has similar advantages to the sliding window technique in handling the streaming data, apart from the fact that a weighing factor has to be determined which may vary from application to application. Furthermore, the weighing factor affects the size of the window, as the window size determines the pruning of older data.

2.1.4 Brief Discussions

Among these three techniques which technique will be suitable for our stream mining algorithm? We learned that, the landmark window technique is not suitable, as its window size keeps increasing as time progresses. This leaves us with two options: the sliding window technique and the tilted/damped window technique. The advantage of the sliding window technique over the tilted/damped window technique is that the user can determine the size of the window depending on the average rate of incoming transactions from a stream. Therefore, based on the user's decision, the mining algorithm would capture necessary transactions from a stream and would have lower probability of overflowing.

Two types of *data units* can be used for the above windowing techniques:

1. One type is *count based* [GDD+03], where the size of the window is based on the number of transactions.

2. Another type is *time based* [GDD+03], where a time value is used to determine the period over which the window will capture data.

Among the two data units, we picked the count based data unit. Essentially, the data units define the size of the window and size of the block of transactions to be processed from data streams. For example, the time based unit will define the size of a window as data received in the last 1 hour or 24 hours, which does not specify any exact number of transactions. On the other hand, the count based data unit will define the size of the windows as recent 100,000 transactions. Similarly, for blocks of transactions to be processed from data streams, the rate can be defined as transactions received in last five minutes (for time based) or the last 10,000 transactions received (for count based). Though our algorithm can handle any of these units, we chose the count based unit, because it is more specific in terms of number of transactions to be processed by the window and the rate of incoming transactions, which makes examples and experimental results easier to understand. To summarize, our algorithm uses the *sliding window technique* with the *count based* data unit.

2.2 Background: Properties of Constraints

Constraint-based mining has several classes of "semantic" single-variable constraints. They can be of various forms including the following [NLHP98, LLN03]:

- *S.Price* θ *constant*, where *set variable* $S \subseteq Items$ (and $Items = \{i_1, i_2, \cdots, i_k\}$ is the set of k distinct items in the transaction database), *Price* is an attribute of an item, and $\theta \in \{=, <, \leq, >, \geq\}$ is a Boolean operator. It says the price of every element of S stands in relationship θ with the constant value.

- $agg(S.A)$ θ *constant*, where agg is one of the aggregate functions min, max,

sum, or avg.

Note that only one item variable is allowed in a single-variable constraint.

The constraint "$S.Price < 100$" says that all items in the set S are of price less than \$100, and the constraint "$max(S.Price) > 100$" says that the maximum price of items in S should be greater than \$100.

While constraints can be categorized based on their forms (as above), they can also be classified into overlapping classes depending on their properties. These classes include *succinct constraints*, *anti-monotone constraints*, *monotone constraints*, and *convertible constraints*. For this research, we use succinct constraints and anti-monotone constraints. These classes of constraints are interesting, because they have some useful properties that help optimize the mining process. For example, for every succinct constraint C_{SUCC}, there is a "formula" that can generate precisely all those itemsets satisfying C_{SUCC}.

Definition 2.1 (Succinct Constraints [NLHP98]) *Let Item denote the set of domain items. Succinctness is defined in several steps, as follows: Define $SAT_C(Item)$ to be the set of itemsets that satisfy the constraint C. With respect to the lattice space consisting of all itemsets, $SAT_C(Item)$ represents the pruned space (i.e., the solution space) consisting of those itemsets satisfying C.*

1. *An itemset $I \subseteq Item$ is a succinct set if it can be expressed as $\sigma_p(Item)$ for some selection predicates p, where σ is the selection operator (as in relational algebra).*

2. *$SP \subseteq 2^{Item}$ is a succinct powerset if there is a fixed number of succinct sets*

$Item_1, ..., Item_k \subseteq Item$, such that SP can be expressed in terms of the powersets of $Item_1, ..., Item_k$ using union and minus.

3. *A constraint C is* **succinct** *provided $SAT_C(Item)$ is a succinct powerset.*

Definition 2.2 (Anti-monotone Constraints [NLHP98]) *A constraint C_a is* **anti-monotone** *iff whenever an itemset S violates C_a, so does any superset of S.*

The following example helps to gain a better understanding.

Example 2.1 *Let us consider the constraint $C_{SUCC} \equiv max(S.Price) > \$10$ (the maximum price of an itemset must be greater than \$10) for the following transaction database and item information: $Items = \{a, b, c, d, e\}$*

TID	*Transaction*
1	*a, b, c, e*
2	*a, d, e*
3	*a, b, c, d*
4	*a, b, d, e*

Item	*a*	*b*	*c*	*d*	*e*
Price	*11*	*5*	*12*	*13*	*7*

With a minimum support set to 3, item c is pruned (i.e., discarded), as it is infrequent. Among the remaining items, a and d satisfy the constraint, but items b and e do not. However, itemset $\{a, b, e\}$ satisfies the constraint, as the item a satisfies the constraint. Therefore, items that do not satisfy the constraint (e.g., e) can still be in some valid itemsets (e.g., $\{a, b, e\}$). So, items can be divided into two groups, namely mandatory groups and optional groups. Items that individually satisfy the constraint (e.g., a and d) are called mandatory items, *whereas items that does not individually satisfy the constraint (e.g., b and e) are called* optional items. *The* member generating function *for constraint $C_{SUCC} \equiv max(S.Price) > \$10$ can be defined in the form*

$\{X \cup Y | X \subseteq \sigma_{Price>\$10}(Items)\ \&\ X \neq \emptyset\ \&\ Y \subseteq \sigma_{Price\leq\$10}(Items)\}$, *where* σ *is the selection operator,* X *contains the mandatory items and* Y *contains the optional items.*

Succinct constraints have the following properties [NLHP98, LLN03]:

- For every succinct constraint C_{SUCC}, there is a "formula" that can generate precisely all those itemsets satisfying C_{SUCC}. This "formula" is called a member generating function.
- A majority of constraints are *succinct.*

A property of anti-monotone constraints is that *all subsets of a valid itemset must be valid* [NLHP98, LLN03]. Hence, itemsets satisfying the succinct and anti-monotone constraint C_{SAM} consist of only items that individually satisfy C_{SAM} (i.e., only the mandatory part). Therefore, there is no optional part in the member generating function for a succinct constraint that is also anti-monotone [Leu04]. This constraint is called **succinct anti-monotone (SAM) constraint.**

We have considered all the three different properties of constraints in our implementation of the constraint-based stream mining algorithm.

2.3 Relevant Algorithms

In the previous two sections, we learned the background on windowing techniques and some constraints. We decided to use the *sliding window technique* to capture recent data from data streams. We *handled the succinct and the anti-monotone constraints.* In this section, we discuss algorithms that are relevant to our design and

development of constraint-based frequent itemset mining from data streams. Firstly, we discuss the basic framework of frequent itemset mining. Secondly, we discuss three incremental mining algorithms and identify a suitable approach for our algorithm. Thirdly, we describe some constraint-based frequent itemset mining algorithms. Finally, we talk about several stream mining algorithms and identify the benefits that our algorithm provides over the existing ones.

2.3.1 Frequent Itemset Mining Algorithms

In this section, we discuss two of the major frameworks for frequent itemset mining algorithms. The frameworks are mainly based on the Apriori algorithm [AS94] and the FP-tree structure [HPY00]. These algorithms underly most of the existing frequent itemset mining algorithms. After the introduction of the Apriori algorithm in 1994, most of the frequent itemset mining algorithms used it as a framework, until the FP-tree was proposed in 2000.

Agrawal et al. [AS94] proposed the **Apriori algorithm**. The Apriori algorithm follows the generate-and-test paradigm. To illustrate, for a given transaction database TDB, the Apriori algorithm performs a level-wise (breadth-first search) generation of itemsets (aka sets of items). The core idea behind this algorithm is that all subsets of a frequent itemset are also frequent. At level 1, the TDB is scanned to extract all the frequent 1-itemsets (where each itemset contains only 1 item). After the first pass, each of the remaining passes performs two tasks (i) generation of new candidate itemsets using the priori information (frequent itemset found in the previous pass) and (ii) re-scan the TDB to obtain the frequency count of the generated candidate

itemsets and prune the infrequent candidates. The algorithm stops when there is no more frequent itemset found in a pass.

The major drawbacks of Apriori are: (i) it generates all possible k-itemsets at a level k, given the frequent $(k-1)$-itemsets from the previous pass (where $k > 1$); and (ii) at every level, the algorithm re-scans the TDB, to get the frequency count of the candidate itemsets. The FP-tree based algorithm overcomes both of these drawbacks.

Han et al. [HPY00] proposed the **FP-tree based algorithm**, which uses an extended prefix-tree structure called Frequent Pattern Tree (FP-tree). The FP-tree is constructed by scanning the TDB twice, and it captures the content of the TDB. In the first scan all the frequent 1-itemsets are generated, and in the second scan the FP-tree is constructed, which selects only the frequent items from a transaction in the TDB. The FP-tree thus contains all the transactions containing the frequent itemsets. Once the FP-tree is built, the mining algorithm called *FP-growth* recursively mines the tree to extract all and only the frequent itemsets. The algorithm avoids generation of any candidate itemsets. It follows the test-only approach rather than the generate-and-test approach followed in the Apriori algorithm.

The advantage of the FP-tree based algorithm (FP-growth) over the Apriori-based algorithm is evident, as the former has proven to be faster by several order of magnitude than any Apriori-based algorithm. In our implementation of the constraint-based data stream mining algorithm, we choose to use the FP-tree based framework as the foundation of the stream mining data structure, and use the test-only mining approach of the FP-growth algorithm.

2.3.2 Incremental Mining Algorithms

We now discuss three of the existing algorithms which provide incremental mining feature: CATS tree [CZ03], AFPIM [KS04], and CanTree [LKH05]. The reason for choosing these set of algorithms is that these are recent improvements on incremental mining for a tree-based algorithm and we are also using a tree-based algorithm.

Cheung and Zaïane [CZ03] designed the **CATS tree (Compressed and Arranged Transaction Sequences tree)** mainly for interactive mining. The CATS tree extends the idea of the FP-tree to improve storage compression, and allows frequent-pattern mining without the generation of candidate itemsets. The aim is to build a CATS tree as compact as possible.

The idea of CATS tree construction is as follows. One database scan is required to build the tree. New transactions are added at the root level. At each level, items of the new transaction are compared with children (or descendant) nodes. If the same items exist in both the new transaction and the children (or descendant) nodes, the transaction is merged with the node at the highest frequency level. The remainder of the transaction is then added to the merged nodes, and this process is repeated recursively until all common items are found. Any remaining items of the transaction are added as a new branch to the last merged node. If the frequency of a node becomes higher than its ancestors, then it has to swap with the ancestors to ensure that its frequency is lower than or equal to the frequencies of its ancestors. Once the CATS tree is constructed, the algorithm called FELINE is used to mine the tree. FELINE, which stands for FrEquent/Large patterns mINing with CATS trEe, is similar to the FP-growth algorithm as it generates all the frequent itemsets with

some recursive pattern growth. There are some issues with the CATS tree: (i) it is not guaranteed to have maximal compression; (ii) the tree build requires splitting, swapping, and/or merging of tree nodes; and (iii) the mining algorithm for a CATS tree requires both upward and downward traversal of the tree paths, which can be computationally expensive.

Koh and Shieh [KS04] developed the **AFPIM algorithm (Adjusting FP-tree for Incremental Mining)**. The key idea of their algorithm can be described as follows. It uses the original notion of FP-trees, in which only frequent items are kept in the tree. Here, an item is "frequent" if its frequency is no less than a threshold, called *preMinsup*, which is lower than the usual user-support threshold minsup. As usual, all the "frequent" items are arranged in descending order of their global frequency. So, insertions, deletions, and/or modifications of transactions may affect the frequency of items. This, in turn, affects the ordering of items in the tree. The AFPIM algorithm adjusts the ordering by swapping items via the bubble sort, which recursively exchanges adjacent items. This can be computationally intensive, because the bubble sort needs to be applied to all the branches affected by the change in item frequency. In addition to changes in the item ordering, incremental updating may also lead to the introduction of new items in the tree. This occurs when a previously infrequent item becomes frequent in the updated database. When facing this situation, the AFPIM algorithm can no longer produce an updated FP-tree by simply adjusting items in the old tree. Instead, it needs to rescan the entire updated database to build a new FP-tree. This can be costly, especially for large databases. The AFPIM algorithm uses an FP-tree as the data structure; therefore, it provides reasonable compression.

However, it also has three major issues: (i) it requires two database scans to build the tree, (ii) the tree build requires splitting, swapping, and/or merging of tree node, and (iii) in cases (where a new frequent item is introduced to the tree) it has to rebuild the tree.

Leung et al. [LKH05] designed and proposed a **Canonical-order Tree (CanTree)** for incremental mining. The construction of the CanTree only requires one database scan. This is less than the construction of an FP-tree which requires two database scans: one scan for obtaining item frequencies, and another one for arranging items in descending frequency order. In the CanTree, items are arranged according to some canonical order, which can be determined by the user prior to the mining process or at runtime during the mining process.

The CanTree has some advantages over the FELINE and AFPIM algorithms which makes it more efficient. Firstly, transactions can be easily added to or deleted from the CanTree without any extensive searches for mergeable paths (like those in FELINE). Secondly, as canonical order is fixed, any changes in frequency caused by incremental updates (e.g., insertions, deletions, and/or modifications of transactions) do not affect the ordering of items in the CanTree at all. Consequently, swapping of tree nodes, which often leads to merging and splitting of tree nodes, is not needed. Once the CanTree is constructed, frequent patterns can be mined from the tree in a fashion similar to FP-growth. In other words, the algorithm employs a divide-and-conquer approach. Finally, the projected databases are formed by traversing the paths upwards only.

From the above discussions, it is clear that the CanTree data structure supports

an efficient incremental mining technique. Our implementation of the stream mining data structure uses the same idea of canonical ordering of items in the tree. This enables efficient incremental update of the stream mining data structure. In addition, our *DSSTree* data structure maintains the data in the tree with the consideration of the sliding window technique. This enables the algorithm to efficiently prune older data from a *DSSTree* during the sliding of the window. The CanTree data structure does not support this, as it is meant of incremental mining on static database (i.e., not streaming data).

2.3.3 Constraint-Based Mining Algorithms

In this section, we discuss some algorithms which use the feature of constraint-based mining. These algorithms are mainly based on Apriori or FP-tree.

Srikant et al. [SVA97] proposed an algorithm to apply **item constraint** in the mining process of association rules. Users are usually interested in a portion of the association rules that are mined. Therefore, it would be appropriate to apply some form of constraints which will define the subset of the output association rules in which the users are interested. Srikant et al. [SVA97] used taxonomy to define hierarchies among the distinct items in the transaction database. The users are allowed to define a subset of itemsets with the help of the taxonomies. Frequent itemsets that satisfy any of the user-defined constraints are returned. The algorithm uses the Apriori framework for the mining process. The constraints that can be defined by the user are confined to only Boolean expressions of itemsets. For example, (Milk $\wedge$ Honey) $\vee$ (Carrot $\wedge$ $\neg$Onion) means the frequent itemsets should have both milk and honey

or have carrot, but not onion.

Ng et al. [NLHP98] proposed techniques to handle three different issues in the process of association rule mining. The first techniques provides user exploration and control over the mining process such that users are allowed to manipulate threshold in between the mining process. Secondly, they proposed some properties and algorithms to enable mining with user focus. Finally, they proposed the idea of using alternate measurements (e.g., correlations) to obtain interesting rules other than typical support and confidence measurement of rules. Out of these proposed solutions, we are interested in the second technique which deals with the application of constraints in the mining process to achieve user focus in the mined output. Ng et al. [NLHP98] introduced the **succinct** and **anti-monotone properties** of constraints (discussed in Section 2.2). Their CAP algorithm uses these properties of constraints to efficiently prune the itemsets that do not satisfy the constraints, and output the frequent itemsets of interest to users. In terms of performance, CAP is several times faster than the other Apriori-based constraint-based mining algorithms.

Leung et al. [LLN02] proposed an FP-tree based constraint-based mining algorithm called FPS. The use of constraints allows users to specify the focus and to guide the mining process based on user input. The application of constraints can increase the performance of the mining algorithm by pruning the search space. In the FPS algorithm, the properties of succinct constraints have been exploited. Leung et al. [LLN02] proposed techniques to handle different sub-classes of succinct constraints namely, succinct anti-monotone and succinct non anti-monotone. With the advantage of the FP-tree, FPS outperforms the Apriori-based constrained mining

algorithms such as CAP.

These constraint-based mining algorithms provided some improvements in terms of performance and functionality when compared with the non-constraint-based mining algorithms. These factors have motivated us to explore the powerful features of constraint-based mining in the area of data stream mining, which none of these algorithms did.

2.3.4 Data Stream Mining Algorithms

In recent years, a few algorithms have been proposed on frequent itemset mining from data streams. Gaber et al. [GZK05] in their survey on mining data streams have categorized data stream mining solutions into two classes: (i) *data-based* solutions focusing on analysis of subset of data or on transformation of data to a compressed or summarized representation; and (ii) *task-based* solutions designed in support of computation theory to achieve better efficiency in terms of time and space. Our algorithm and the related algorithms discussed in this section fall under the class of *task-based* solutions. Gaber et al. [GZK05] discussed many techniques of mining, such as clustering, classification, frequency counting, and time series analysis. For our case, we only focus on frequency counting (algorithms for mining frequent itemset from data streams). Many of the proposed algorithms used various windowing techniques and itemset frequency count approximation methods. Here, we briefly discuss some of them and compare them with our proposed *DSSTree*. Table 2.1 shows the key differences between these algorithms and our proposed *DSSTree*.

Manku et al. [MM02] proposed two algorithms, **Sticky Sampling** and **Lossy**

Counting, to compute approximated frequency counts of itemsets with an efficient memory usage. They do not consider any windowing techniques to capture recent data (but they mention as a future work that they are exploring the usage of their algorithms for sliding windows). The Sticky Sampling algorithm only mines frequent items, whereas the Lossy Counting algorithm mines both items and itemsets. Both algorithms use an approximation factor to obtain an estimated frequency of items or itemsets. To elaborate, the Sticky Sampling algorithm is a sampling based algorithm that uses a probabilistic method to get an estimated frequency. The frequent items are maintained in a list of records, where each record contains an item and its estimated frequency. On the other hand, the Lossy Counting algorithm divides the input data stream into buckets of transactions. The algorithm uses a *trie* to maintain the current frequent items in a set of records, where each record contains an item, its estimated frequency, and the maximum error value.

To summarize, Manku et al. [MM02] used an estimation method to obtain the frequency count of itemsets with an error factor. The algorithm maintains all the current frequent itemsets from the data streams, but does not use any sliding window method to reduce the huge, continuously growing data from the stream.

Chang et al. [CL03] proposed an algorithm called **estDec Method** to find recent frequent itemsets for streaming data. Their algorithm uses a decaying technique, which resembles the *damped window technique* (Section 2.1.3), to prune old data from data streams. The decay rate specifies the current rate at which the weight assigned to a group of patterns will reduce over time. To store frequent itemsets, Chang et al. used a prefix-tree lattice structure, also known as *monitoring lattice*. As

the estDec Method only keeps frequent itemsets, some approximation methods are needed to obtain the estimated frequency count of the itemsets. In order to reduce the effect of early pruning of itemsets, the algorithm uses a secondary minimum support threshold (preMinsup) that is lower than the usual minimum support threshold value (minsup). In the initial itemset pruning stage, this secondary minimum support threshold (instead of minsup) is applied, because Chang et al. claimed that any item below this secondary minimum support threshold cannot become frequent soon.

Unlike Lossy Counting algorithm, the estDec Method uses a sliding window technique to capture the recent frequent itemset. However, both use approximation methods as they only keep the frequent itemsets. One key issue with the estDec Method is the use of a secondary minimum support threshold. This can trigger the re-building of the monitoring lattice as in the case of the AFPIM algorithm [KS04] which uses a similar concept for incremental mining. To avoid this type of issue, our algorithm maintains all the items captured from the stream in the sliding window.

Giannella et al. [GHP+02] proposed an algorithm called **FP-streaming** for finding frequent patterns from data streams with the use of a tree structure. They used the *tilted-time window technique* (which can also be used in regression analysis on data streams [CDH+02]). As users are more interested in changes in recent data, the granularity of data for analysis in the FP-streaming algorithm can be varied based on the age of the data: the older data are stored with a coarser granularity, whereas the more recent data have a finer granularity. The granularity is reduced from recent to older data in a logarithmic rate. Based on the typical sliding window concept, the oldest batch of data in the current window is pruned.

The data structure that Giannella et al. [GHP+02] introduced is *FP-stream*, which maintains a tilted-time window table for each node in the tree. In the table, the frequency count (or more precisely, the estimated frequency count returned by an approximation method) for each tilted-time frame is recorded. During the mining process, the FP-streaming algorithm first generates a *Frequent Pattern tree* (FP-tree) [HPY00] to store all the transactions of the current chunk of data in the streams, then finds the current frequent itemsets and updates the FP-stream structure, which contains all the current frequent itemsets.

To summarize, the *FP-stream* data structure uses a windowing technique to capture recent data from data streams. Unlike the estDec Method [CL03], which used some decay rate to change weights on older batch of frequent itemsets, Giannella et al. [GHP+02] changed the granularity of the older batch of frequent itemsets. The FP-streaming algorithm prunes all the infrequent items early on; it cannot avoid the use of estimation methods to determine approximate frequencies. Also, as the frequency counts of the items are changing over time (as new data are added to the tree), the FP-tree and the FP-stream structures may no longer maintain items with a global frequency order. This may result lesser sharing of tree paths, and thus larger tree size. To avert these issues, our algorithm takes the following measures: (i) maintains all items in the current sliding window so it can get the actual frequency counts of itemsets that change from infrequent to frequent, and (ii) our novel *DSSTree* maintains items in some canonical order which is independent of the local or global frequency counts of items.

Chi et al. [CWYM04] proposed an algorithm called **Moment** for finding closed

frequent itemsets from streaming data. They used the sliding window technique and a *Closed Enumeration Tree* (CET) to mine recent closed frequent itemsets data from the streams. In a CET, both frequent and infrequent closed itemsets are maintained, thus avoiding approximation methods. The nodes in a CET are arranged in such a way that each node represents an itemset X and its child node represent a superset of X. Each node in the tree also maintains status information that indicates whether the itemset is closed frequent or not. The mining algorithm updates the CET by updating the frequency count and the status of the nodes. As the window slides over time, the CET maintains the current closed frequent itemsets.

The use of the sliding window helps the CET data structure capture the recent block of data from stream. Furthermore, mining of closed itemsets reduces the number of itemsets to be returned. However, it does not ensure that all returned itemsets are of interest to users. In contrast, our algorithm uses constraint-based mining to both reduce the output frequent itemsets and provide itemsets of interest to users.

Table 2.1: Comparing the proposed DSSTree with four existing algorithms

Algorithms	Lossy Counting [MM02]	FP-streaming [GHP+02]	estDec Method [CL03]	Moment [CWYM04]	Proposed DSSTree
Itemset mining	√	√	√	√	√
Maintain a window to capture recent data	×	√	√	√	√
Avoid estimation calculation	×	×	×	√	√
Interactive Mining	×	×	×	×	√
Constraint-based mining	×	×	×	×	√

Table 2.1 illustrates, the additional functionalities that our algorithm provides

when compared with four related algorithms. From Table 2.1, we can see that algorithm Lossy Counting [MM02] does not use any windowing technique to capture recent data from data streams. This can be a critical issue on the long run as data streams can be of infinite length. The other three algorithms use some windowing techniques. Out of the three algorithms, FP-streaming [GHP+02] and estDec Method [CL03] require an approximation method to get estimated frequency counts of itemsets, whereas Moment [CWYM04] avoids approximation methods as it maintains all itemsets in the CET data structure. In comparison, our proposed *DSSTree* has all the necessary features of a stream mining algorithm and some more. To be more specific our proposed algorithm provides the following: (i) the *DSSTree* data structure for efficient incremental update of data from streams; (ii) a sliding window technique to capture the recent data; and (iii) it preserves all itemsets in a window, and thus avoids use of frequency approximation method. In addition, our algorithm *enables interactive mining*, and uses the powerful feature of the constraint-based frequent itemset mining from data streams. To this end, only the Moment algorithm generates closed frequent itemsets to reduce the output frequent itemsets. However, it does not support interactive mining, and unlike constraint-based frequent itemsets, closed frequent itemsets does not necessarily generate itemsets of interest to user.

2.4 Summary

To the best of our knowledge, the above algorithms do not fully address all three key research questions regarding the frequent itemset mining from data streams. If we recall, the questions were: (i) How can an efficient incremental mining technique

that handles continuous incoming data from stream be developed? (ii) What type of windowing technique can be used to capture the most recent data from data streams: the sliding window techniques or others? and (iii) How can users mine for frequent itemsets of interest to them?

With the insight on the current research on data stream mining, we have identified some potential approaches to resolve these existing issues. We learned that, both FP-tree based and Apriori based frameworks have been adapted in most of the frequent itemset mining algorithms. Between the two frameworks, the FP-tree based framework outperforms the Apriori based framework as the FP-tree based framework avoids the computationally expensive candidate-itemset generation process. Thus, we use the FP-tree based framework for our implementation of the stream mining algorithm. However, traditional mining approaches do not consider the dynamic and continuous nature of data, which can be dealt with by data stream mining techniques. For data streaming, it is important to find an efficient way to update the tree structure so as to keep it updated with the incoming data streams. To deal with this situation, we apply the *canonical ordering of tree items* to efficiently process incremental updates. This ordering has been found effective in previous research on incremental mining [LKH05]. This approach avoids the computationally expensive node splitting and node swapping operations during incremental update of tree structure, which are common operations used by other incremental mining algorithms. Such an *incremental mining* technique can be adapted to help keeping up to date the tree structure that captures all the stream data.

Since users are usually interested in mining results from recent data, we adapt

the technique of a *sliding window* to capture the recent data from data streams. The sliding window captures the most recent block of data, and slides forward to capture newer data and discard the oldest data. The use of such a windowing technique may focus on the recent data from data streams, but the data size can still be very large. Thus, mining will produce a large number of frequent itemsets. Only a subset of them will most likely be of interest to the user.

Some forms of *constraint-based mining* are necessary to extract itemsets that are of interest to the user. Moreover, the application of constraint-based mining provides better focus and control of the stream mining process for the user. The review of the literature in this chapter also revealed that existing stream mining algorithms did not use the powerful feature of constraint-based mining.

These potential improvements to stream mining algorithms motivated us to develop an algorithm on constraint-based frequent itemset mining from data streams, which we implemented as part of this thesis work. In the next chapter, we start discussing our work. Specifically, we discuss the architecture and design underlying the development of our algorithm on constraint-based frequent itemset mining from data streams.

Chapter 3

Architecture of Constraint-Based Data Stream Mining

In the previous chapter, we reviewed some existing work; in this chapter, we start describing our new work. We learned from Chapter 2 that mining frequent itemsets from data streams leads to the key problem of maintaining *large* and *continuous* data. Generally, we address the problem by: (i) adapting incremental mining techniques and windowing techniques to effectively capture, maintain and manage the streaming data; (ii) applying constraint-based mining to find itemsets that are of interest to the users; and (iii) providing interactive mining feature [CZ03, EZ03, Leu04] to allow users to modify their constraints.

Specifically, in this research, we develop a constraint-based algorithm for finding frequent itemsets from data streams. We use a sliding window technique to handle the issue of huge (potentially infinite) amount of data in data stream mining. We propose a tree-based data structure, called the *DSSTree*, to incrementally maintain the data

from streams. The *DSSTree* can also be used for interactive mining. The application of constraints in the mining process offers users more control over the mining process, which provides the ability to discover frequent itemsets of users' interest, and enables efficient pruning of the search space.

The remainder of this chapter is organized as follows. Firstly, we discuss the underlying architecture of the constraint-based stream mining algorithm. Secondly, we explain the windowing technique adapted in our algorithm. This will answer our research question on what technique can be adapted to capture the most recent data from data streams. Thirdly, we design and implement a stream mining data structure; we give details of this structure. The structure provides a suitable solutions for research question relating to the efficient incremental mining. Fourthly, we explain the methodologies in regards to the application of constraints in the stream mining process, answering the question on mining interesting frequent itemsets. Finally, the entire process of the constraint-based frequent itemset mining algorithm and interactive mining for data stream is discussed. In all these sections, we use examples and figures to illustrate the mining process and various parts of the algorithm.

3.1 An Overview of Our Constraint-Based Stream Mining Architecture

In this section, we explain the overall architecture of the constraint-based stream mining system, as illustrated in Figure 3.1. On the top left of the figure, we can see that the user initiates the stream mining configuration by setting the following:

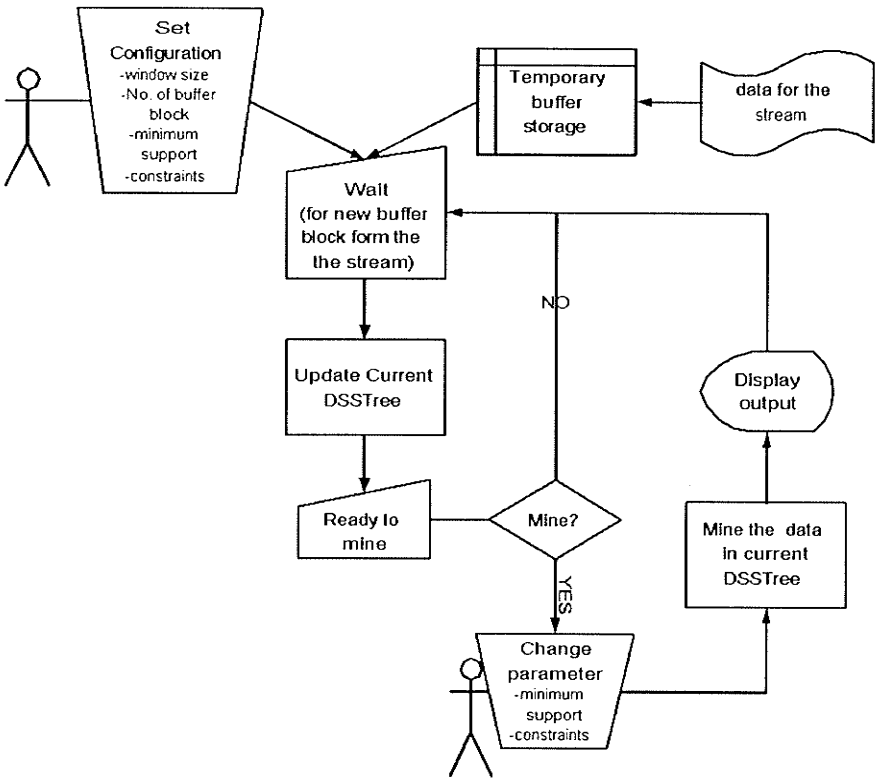


Figure 3.1: The architecture diagram of the constraint-based stream mining system

(i) the window size; (ii) the number of buffer blocks the window should maintain[1]; (iii) the minimum support threshold; and (iv) the user-defined constraints. Once the user completes the configuration process, the mining is ready to start. The top right of Figure 3.1 shows the data input from the stream which is stored in a temporary buffer storage. When the temporary buffer storage has enough data to form a buffer block, it passes the data to the waiting state of the mining system. Now, the configuration information and the data are passed to the *Update Current DSSTree* process,

[1]The sliding window that our algorithm uses is partitioned into smaller buffer blocks. On the shifting of the window, the oldest buffer block is discarded, and the new buffer block is added to the window.

which updates the data stream storage tree. (Details of the update can be found in Section 3.3.) After the completion of the tree update, the mining system is ready to mine the current DSSTree. If the user is not ready to mine the data (e.g., busy with other work) or a timeout occurs (for no response from the user), then the system goes to the "wait" state (to wait for more data from the stream). Otherwise, the system provides the user with the option to change the mining parameters: (i) minimum support threshold and/or (ii) user-defined constraints. The user can change the values for these parameters or proceed with the current parameters. Then the system runs the mining algorithm, and outputs constraint-based frequent itemsets. After finishing the mining process, it goes back to the "wait" state. This process continues unless terminated manually.

3.2 The Window Technique

Since data streams are continuous and of infinite length, it is not feasible to mine the entire history of data. As mentioned in most of the data stream mining literature (e.g., [GHP+02, GDD+03, CL04, CWYM04, LLS04]), users are mostly interested in changes and status of recent data in data streams. This leads to the use of windowing techniques, which captures only recent data. All transactions that fall within a window are the current set of input data to be mined.

We use the *sliding window technique* with *count based units.* As the rate of incoming transactions may vary from time to time, with a count based window, a given number of transactions will be processed as soon as they arrived, regardless of the timing. The algorithm can estimate the amount of memory required for processing,

because it knows the number of transactions to be processed. This in turn may allow the algorithm to notify the user beforehand, whether it is feasible to handle the amount of data with the available memory.

We use a sliding window technique; we call it *buffered sliding window*, where the window is partitioned into segments. A segment in the buffered sliding window is called a *buffer block*. Zhu et al. [ZS02] used this concept of sliding window for statistical monitoring (which is not frequent itemset mining) of data streams. Moreover, they used this concept only to partition the data in a sliding window to produce statistical reports on each segment. Our technique uses the *DSSTree* data structure (described in Section 3.3) to store and maintain the sliding window, whereas Zhu et al. have not used any data structure to efficiently store the data of a sliding window.

Let us elaborate our buffered sliding window technique. We BT to denote the number of transactions in a buffer block and BB to denote the number of buffer blocks in buffered sliding window. These parameters BT and BB are user-defined values. Each buffer block contains BT transactions, and there will be BB buffer blocks in the window. The buffer blocks are labelled from 1 to BB, where 1 is the oldest buffer block and BB is the most recent. During the sliding of the window, the buffer labelled 1 will be discarded; labels of all other buffers will be decremented; and the new buffer with label BB will enter into the window. When the number of buffer blocks received from the stream is less than or equal to BB, no sliding of the window is required. Once the window has BB buffer blocks, for every new buffer block there is a shift/slide of the window (we will use the words *shift* and *slide* interchangeably), which is required to discard the old data and insert the new data. In our algorithm,

the values of BB and BT can be set by the users based on their requirements. This allows the user the flexibility to specify their requirements based on the available memory (where larger BB or BT means more memory consumption).

Example 3.1 *Let us consider the following scenario. The sliding window has 3 buffer blocks, each block has 10 transactions, and the rate of incoming transactions from the data streams is 10 transactions per minute. There is no buffer block in the window at the start. After the first 3 minutes, all the incoming transactions are kept in the window to fill the 3 buffer blocks. However, starting from the fourth minute, the window shifts. As a result of the first shift (at the end the fourth minute), the first buffer block with 10 transactions will be discarded and 10 new transactions will be added.*

This technique of buffered sliding window is a conceptual process. In the actual implementation, no such window with transactions are stored. Instead, the *DSSTree* data structure keeps track of the contribution of items' frequency for each of the incoming buffer blocks. The algorithmic details of the process of frequency contribution is discussed in the next section.

3.3 The Stream Mining Data Structure

Our proposed *Data Stream Storage Tree* (*DSSTree*) is a data structure for stream mining. The *DSSTree* is designed based on the Canonical-order Tree (CanTree) [LKH05]. In both tree, a parent node always has a frequency (local frequency) value greater than or equal to the sum of the frequency values of its children.

In a CanTree, which is a prefix tree, the items in a transaction are ordered in any canonical order (not dependent on any frequency order). It requires only one scan of data to build the tree, which is necessary for data stream mining [CL03]. However, CanTree data structure is not sufficient to handle the continuous data from the stream, where the windowing technique is necessary to capture recent data. For this reason, our *DSSTree* only adapts the canonical ordering property of CanTree for incremental mining. Thus, the key difference between the CanTree and the *DSSTree* is the incorporation of the buffered sliding window in the latter. This enables the *DSSTree* to mine frequent itemsets from data streams. On the other hand, the CanTree cannot handle streaming data as it is designed for incremental mining from static database, not for continuous data streams.

In this thesis, we are considering the lexicographical ordering of the items. Hence, changes on the value of the item frequency have no effect on the ordering of the items in the tree. Equipped with this feature, a *DSSTree* can be easily updated when it receives transactions. Unlike many other incremental mining algorithms (e.g., [CZ03, KS04]), the *DSSTree* avoids node swapping, and merging or splitting; it does not need to rebuild the tree at any stage.

As we mentioned earlier, apart from canonical ordering of the items, the *DSSTree* also keeps track of the frequency count of each buffer blocks in a buffered sliding window. Separate frequency count information is required for each buffer block maintained in the window. In each node, a dynamic list of frequency count values is recorded, where each value corresponds to the respective buffer block. This list is very useful during the sliding of the window. The frequency of items of a deleted

buffer block can be easily removed from the *DSSTree* by deleting the first entry of the list (given that the first entry refers to the count of the oldest buffer). Similarly, the item frequency of the newly added buffer block can be added as the last entry of the frequency list of a node. With this process, the *DSSTree* can be easily built and updated with new data from the streams. It also captures only recent data with the shift of the buffered sliding window. As a tree structure, the *DSSTree* maintains a *header table.* In a header table, a dynamic list of all the distinct items present in the current tree is stored. With each of the item values in the header table, the frequency of the item and a link to a tree node containing that item are also recorded. The link from the header table items to the tree nodes can be used to traverse all the nodes containing the respective item, because all the nodes in the tree containing the same item is linked with each other.

Example 3.2 *Figure 3.2 shows a sample* DSSTree *and its associated header table. All the three buffer blocks in Table 3.1 have been used to build the sample* DSSTree. *The dotted lines represent the links from an item X in the header table to the leftmost*

Table 3.1: Sample 1 for data stream transactions

	Transaction Numbers	Contents of Transcation
buffer block 1	t_1	$\{a, d, b, e, c\}$
	t_2	$\{d, b, a, c\}$
buffer block 2	t_3	$\{d, a, b\}$
	t_4	$\{a, c, d\}$
buffer block 3	t_5	$\{c, b, a\}$
	t_6	$\{d, a, c\}$

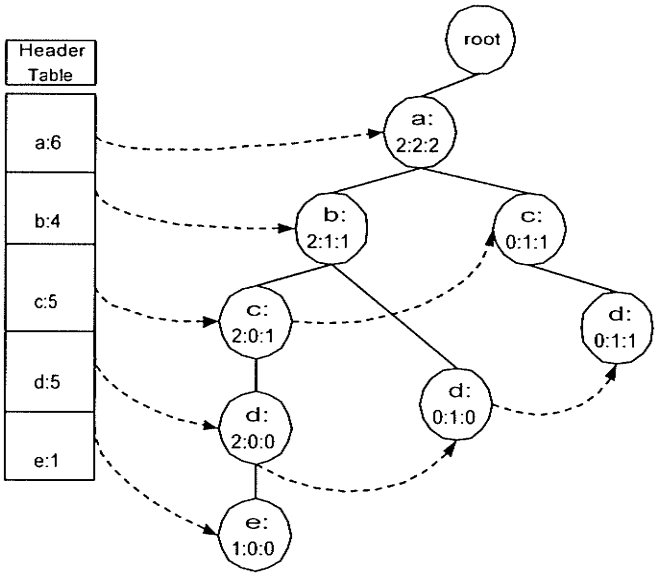


Figure 3.2: A sample DSSTree and its associated header table

X in the tree, and the links between the X's in the tree. In each node, several values are stored: (i) the first value is the value of the item in the node, and (ii) the remaining values separated by a colon are the frequency of each buffer block. For example, the node containing item b has a frequency count of 2 for the first buffer block and 1's for both second and third buffer blocks. The update of the DSSTree *due to the sliding of the window and the incremental update will be illustrated in Section 5.4.*

During the mining phase, the usual FP-growth algorithm can be applied to extract the complete set of frequent itemsets. In other words, for each valid frequent item X, a projected database is generated. An $\{X\}$-projected database is a subset of the input transaction database, where each transaction contains the item X. Valid frequent items are those, which satisfy both the frequency threshold and the user-defined constraints. Based on the $\{X\}$-projected database, an $\{X\}$-projected *Data*

Stream Mining Tree (*DSMTree*) is built. The difference between the *DSMTree* and the *DSSTree* is that, in the *DSMTree*, the list of frequency counts for each buffer block is not maintained, and only one cumulative frequency count (summing the counts of the entire sliding window) is used. The reason behind this is that the projected trees do not need to be updated in the mining process with new data from the stream. This also saves memory space in the mining phase. Each $\{X\}$-projected *DSMTree* is further mined and projected accordingly in a recursive fashion.

3.4 Exploitation of Constraints

Incorporating the constraint-based mining feature is an important task. Each of the constraints has its own properties, which can be exploited in the mining process. For this study of constraint-based data stream mining *succinct constraints* and *anti-monotone constraints* is considered.

3.4.1 Handling Succinct Constraints

When handling *succinct constraints*, the set of items in the transaction database is divided into two groups, namely *mandatory group* and *optional group.* This idea of grouping items in tree-based constraint-based mining was introduced by Leung [Leu04]. In the *DSSTree*, all items are maintained (regardless of whether they are frequent or infrequent, mandatory or optional). *We only form* projected DSMTrees *for mandatory frequent items.* Since the items are arranged in lexicographical order (e.g., some mandatory items may appear between some optional items on a path in the tree), both upward and downward traversal in the path are required to completely

gather optional items from the tree paths (i.e., not to miss any optional items when forming the projected DSMTrees) [Leu04]. Once these projected DSMTrees are built, the usual tree mining technique (e.g., FP-growth [HPY00]) can be applied recursively to find all frequent itemsets satisfying the succinct constraints.

Note that, when processing succinct constraints, we only need to form projected DSMTrees for mandatory frequent items because each valid itemset must contain a mandatory item (and may contain some optional items). This is an advantage of exploiting succinct constraints.

3.4.2 Handling Anti-monotone Constraints

For *anti-monotone constraints*, there is no notion of mandatory or optional items. When handling these constraints, every single item in the *DSSTree* must be checked. If the item X is valid, the algorithm forms a projected *DSMTree* for such an item X. In the $\{X\}$-projected *DSMTree*, the algorithm still needs to check for each item Y to see if $\{X\} \cup \{Y\}$ is valid. If valid, it forms the $(\{X\} \cup \{Y\})$-projected *DSMTree*. This process is applied recursively to find all frequent itemsets satisfying the anti-monotone constraints.

Note that, if the item X is invalid, the algorithm does not need to form any projected *DSMTree* for such an item X because any superset of an invalid itemset is also invalid. This is an advantage of exploiting anti-monotone constraints.

3.4.3 Handling Succinct Anti-monotone Constraints

Given that any itemsets satisfying *succinct anti-monotone (SAM) constraints* must comprise of only valid frequent items (i.e., items that individually satisfy the constraints and are frequent). It is necessary to find only all valid frequent items from the *DSSTree* and form a projected *DSMTree* for each of these items. Then, the usual tree mining technique can be applied recursively to find all frequent itemsets satisfying the SAM constraints. Note that the key difference between the SAM constraint and the above two constraints is that there is no longer a need to perform constraint check for projected DSMTrees (as for the anti-monotone constraints) and we have a smaller solution space to deal with (when comparing with the solution space for the succinct constraints, which contain both mandatory and optional items).

3.5 The Constraint-Based Data Stream Mining Algorithm

In this section, we discuss the complete process of constraint-based stream mining. We use an example to illustrate the data structure and the mining process of the *DSSTree*. The input data stream transactions shown in Table 3.2 are given in three separate blocks. The buffered sliding window configuration in the example has 2 buffer blocks in the window, and each block will contain 3 transactions. Like many typical tree based mining processes, there is a *tree building phase* and a *mining phase*. For this algorithm, the tree building phase is a continual process. The current *DSSTree* is updated with the new transaction block. Thereafter, if the user wants to mine the

Table 3.2: Sample 2 for data stream transactions

	Transaction Number	Contents
Incoming data block 1	t_1	$\{a, d, b, e, f, c\}$
	t_2	$\{d, f, b, a, e\}$
	t_3	$\{a\}$
Incoming data block 2	t_4	$\{d, a, b\}$
	t_5	$\{a, c, b\}$
	t_6	$\{c, b, a, d\}$
Incoming data block 3	t_7	$\{a, c, b\}$
	t_8	$\{c, b, a\}$
	t_9	$\{a, c, d\}$

current data, the mining process can be invoked; otherwise, the algorithm continues updating the *DSSTree* with the new transaction block from the data streams. After finishing the mining process, the proposed algorithm returns to the tree building phase. Then, the user is allowed to modify the defined frequency support threshold and the constraints at any time during the mining process.

Example 3.3 *Figure 3.3 shows the change in the* DSSTree *after processing each transaction block. During the processing of the first two blocks, no transaction is removed from the tree, because the sliding window size is 2 buffer blocks. For the third buffer block, all transactions from block 1 are removed, and the new block is read to update the tree. One can observe from Figure 3.3 that multiple frequency counts are shown in each node. These are the frequency counts of each buffer block received. The first count refers to the first block in the current window, and the second count is for the second block, and so on. To elaborate, if we consider the incremental update of*

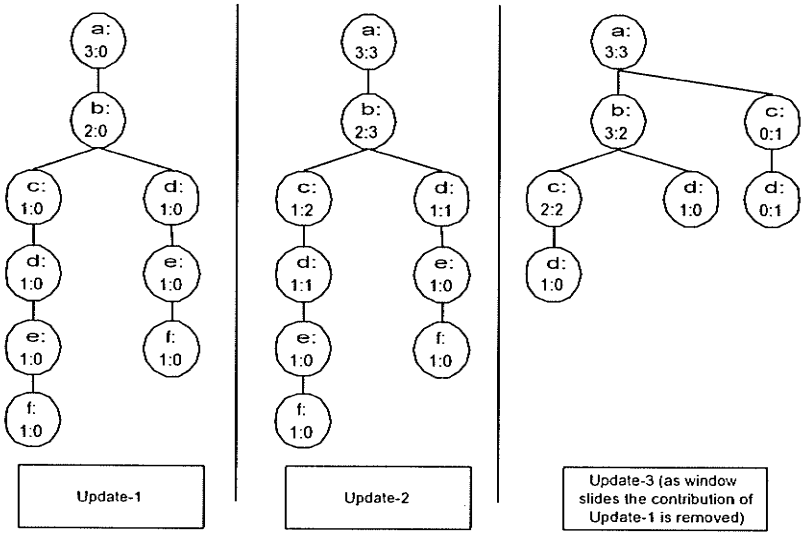


Figure 3.3: The DSSTree with each of the block updates during incremental mining.

the node containing item b, we can observe the following from Figure 3.3: (i) after update-1, the frequency count is 2 for buffer block 1; (ii) after update-2, the frequency count 3 is appended to the frequency list; and (iii) after update-3, the contribution of the update-1 is removed (as the window slides), therefore, the first frequency count is the contribution for the update-2 (with frequency count 3) followed by frequency count 2 for the update-3. Based on the minimum support threshold and constraint type (e.g., succinct, anti-monotone etc.), the mining is conducted as described in Sections 3.3 and 3.4.

3.6 Interactive Mining from Data Streams

Our proposed *DSSTree* data structure enables interactive mining on the data in the sliding window. The *DSSTree* data structure contains all transactions from the sliding window, and thus it is possible to mine the data again in the tree with changed

minimum support threshold (*minsup*) and/or constraints. Mining the *DSSTree* with the changed *minsup* can still generate a complete output frequent itemsets, because both frequent and infrequent items are maintained in the data structure. Existing stream mining algorithms do not support interactive mining, because of any of the following reasons:

1. The mining algorithm prunes infrequent items based on the *minsup*, and thus loses records of infrequent items' frequency count. Now, if the *minsup* is reduced, then previously infrequent items may become frequent. With the previously infrequent items already pruned, there is no alternative (because data from streams can be scanned only once) to get the exact frequency count of those pruned items.

2. The data structure does not support interactive mining.

The interactive mining feature can be potentially useful in data stream mining, as data from streams can be scanned only once, which seals the option of rescanning the data to build and mine with the new *minsup*. In addition, even if there is a method to rescan the data from the streams, it would reduce the performance because of extra rescan (causing rebuild of the tree). We illustrate this with the following example.

Example 3.4 *Let us consider two algorithms A and B. Algorithm A (like our proposed algorithm) maintains sufficient information on data (i.e., contains both frequent and infrequent itemsets) in its data structure, such that it does not require any rescan when the minsup is changed. On the other hand, algorithm B requires rescan when the minsup is changed. For simplicity, let us consider that for both algorithms A and*

B, the tree building time is TB and mining time is M. Table 3.3 shows the runtime for algorithms A and B with multiple changes in the minsup.

Table 3.3: Time comparison for interactive mining

	Runtime for Sample Algorithms	
	A	B
1st time building & mining	$TB + M$	$TB + M$
1st change in minsup	M	$TB + M$
2nd change in minsup	M	$TB + M$
Cumulative rumtime	$TB + 3M$	$3TB + 3M$

From the above table, we can see that each subsequent change in the minsup after the first build costs TB more time for algorithm B (than algorithm A).

From the above example, we observed that the interactive mining of the proposed *DSSTree* algorithm not only provides users with the feature of exploratory mining (e.g., mine with changing *minsup*), but also achieves better performance.

3.7 Summary

In this chapter, we identified the major tasks for this research. The flexible design of the proposed architecture and implementation methodologies for constraint-based stream mining meet all the identified tasks. In brief, the tasks and their corresponding solutions were:

Task 1 We used a windowing technique to capture recent data from the stream. We

implemented the buffered sliding window technique, which provides a procedure to capture recent data for the continuous data streams.

Task 2 We developed an efficient algorithm to incrementally update the data structure (storing the data), which includes the development of the data structure itself. The novel data structure proposed, called the Data Stream Storage Tree (*DSSTree*), can efficiently handle the features of the buffered sliding window technique, and the incremental update of data.

Task 3 We provided a way to mine interesting frequent itemsets from the data. The application of constraints not only produces output of interest to users, but also reduces the mining search space.

Task 4 We integrated the components in the above mentioned tasks to develop a constraint-based frequent itemset mining from data streams, with additional feature of interactive mining.

While we discussed the conceptual building blocks in this chapter, we will explain in the next chapter how we combine these conceptual building blocks, and how we implement a constraint-based frequent itemset mining from data streams. We will also describe the details of the algorithms with necessary pseudocodes.

Chapter 4

Implementation of Our Constraint-Based Stream Mining Algorithm

In Chapter 3, we described the details of the underlying architecture and design of the algorithm for the constraint-based frequent itemset mining from data streams. In this chapter, we focus on the implementation of this algorithm with reference to the corresponding pseudo-codes. We use the C programming language for implementation. We define C structures for the necessary data structures such as tree node, header table items, and transactions. This chapter is divided into three major parts: descriptions on the *DSSTree* construction algorithm (Section 4.1), the *DSSTree* (Section 4.2), and the *DSSTree* mining algorithm (Section 4.3). In Section 4.1, we give details of the *DSSTree* building algorithm. In Section 4.2, we describe the algorithms which implement the concept of the sliding window technique, and the pruning and

maintenance of the *DSSTree*. In Section 4.3, we focus on the recursive mining method, which generates all desired (based on user-defined constraints) frequent itemsets. We describe the implementation of algorithms for handling both succinct constraints and anti-monotone constraints.

4.1 The DSSTree Building Algorithm

The pseudocode in *BuildDSSTree* Algorithm lists the *DSSTree* building algorithm with the *InsertTransaction* procedure. In order to appreciate the details of the algorithm, it is important to have a clear understanding of the basic data structure involved in the *DSSTree* structure. Typically, a tree consists of nodes, which are linked together in a acyclic fashion. A *DSSTree* node consists of the following:

1. an item (the value of an item from the distinct set of items in the data)
2. *an array of frequency counts (containing the different frequency counts for all the buffer blocks in the current window)*
3. a pointer to the parent of the node
4. a pointer to the children list
5. a pointer to the sibling node
6. a pointer to the next node (*node_link*) in the tree containing the same item

For memory optimization, we use another node data structure. We call the tree built using this structure, a Data Stream Mining Tree (*DSMTree*). The only difference between *DSMTree* tree node and the *DSSTree* node is that, instead of an array to store the frequency, the former uses a single count value. The array of frequency

count keeps track of the different frequency contributions by each buffer block of the current sliding window. However, during mining, this information is not required as no updates are performed on the mined tree.

Algorithm *BuildDSSTree*(TDB)
(* The DSSTree building algorithm *)
Input: Block of Transactions TDB
Output: A DSSTree and its associated $HeaderTable$
1. Initialize $RootNode$
2. Initialize $HeaderTable$
3. Update $globalUpdateCount$
4. Update $globalOldestPtr$
5. **for all** $Transaction \in TDB$
6. **do for all** $item \in Transaction$
7. **do** add $item$ to $HeaderTable$
8. Sort $Transaction$ in defined canonical order
9. **call** $InsertTransaction(RootNode, Transaction, 0)$
10. **return** $RootNode$ *and* $HeaderTable$

Procedure $InsertTransaction(CurrentNode, Transaction, Index)$
(* The transaction insertion procedure *)
Input: A node where to insert, a transaction, and a index pointing to the current item in the transaction
1. $Item \leftarrow Transaction.Items[Index]$
2. **if** $CurrentNode$ has no child
3. **then** Create new node N with item value set to $Item$
4. Update $HeaderTable$ link to N
5. Update N with appropriate frequency contribution
6. **else if** $Item$ matched with any children N of $CurrentNode$
7. **then** Update frequency of N
8. **else** Create new node N with item value set to $Item$
9. Update N with appropriate frequency contribution
10. Update $HeaderTable$ link to N
11. **if** $Index + 1 < Transaction.Size$
12. **then call** $InsertTransaction(N, Transaction, Index + 1)$

Similar to many tree based algorithms, a header table is also associated with the tree. The records in the header table contain the following: (i) an item, (ii) a global frequency count of the item, and (iii) a pointer to the right most node of the

tree containing the item's current record. The pointer from the header table to the tree, and the *node_link*s among the tree nodes help the algorithm to extract all the itemsets containing an item X. Beside the node and header table structures, there is also an $ItemInfoList$ array that contains different properties (such as price, type) of the items. This data structure is necessary for constraint-based mining, in which the algorithm has to validate an item based on the user-defined constraints. For example, the algorithm may have to check whether the price of an item X is $\geq$ \$100.

As input, the *BuildDSSTree* algorithm takes the most recent block of transactions from the temporary buffer which stores data from the streams. It updates the global $RootNode$ (the $DSSTree$ root node) and $HeaderTable$ based on new block of transactions received. The initialization of the $RootNode$ and $HeaderTable$ (lines 1 and 2) is performed only for the first build of the $DSSTree$. During the tree update, the global $RootNode$ and $HeaderTable$ are updated accordingly (no need for initialization). In lines 3 and 4, the global counters are incremented to keep counter values current. The *globalUpdateCount* keeps track of the total number of tree updates performed. The *globalOldestPtr* is the index of the oldest frequency contribution in the frequency contribution list. The frequency contribution list is maintained in each tree node, which stores the frequency contribution of a buffer block in the buffered sliding window. In line 5, the algorithm iterates through each transaction in the most recent buffer-block. The most recent buffer-block represents the new block of transactions captured from the data streams. The items on a transaction are first inserted into the $HeaderTable$; if the item already exists, then only the frequency count of the item is incremented. Then, the transaction is sorted according to the user-defined canonical

order, which is set to lexicographical order in the current implementation. After sorting, the transactions are inserted to the *DSSTree* invoking the *InsertTransaction* procedure.

The *InsertTransaction* procedure takes a *CurrentNode*, a *Transaction*, and an *Index* as parameters. The *Index* refers to the current item in the transaction to be inserted in the tree. If the *CurrentNode* has no child, then a new node is created (line 3) with the current item, and added as the child of the *CurrentNode*. The link from the *HeaderTable* to the *DSSTree* is updated in line 4. For cases in which the *CurrentNode* has a child, the *Item* is searched in the children nodes. If the *Item* is found, then the frequency contribution list of that child node is updated (i.e., incremented). The details of the algorithm which updates the frequency contribution list are illustrated in Section 4.2.1. If the *Item* is not found, then a new node is created with the current item and added to the children list of the *CurrentNode* (line 8). The procedure is invoked (line 12) to recursively process the remaining items in the transaction. The recursion is terminated if there is no more item to process (the condition is checked in line 11).

Once the tree is built, the stream mining algorithm is ready to intake new transactions from the buffer block, as well as ready to mine. The mining algorithm for constraint-based mining is described in Section 4.3. In Section 4.2, we describe the algorithms required to maintain the *DSSTree*, which includes update of node frequency and tree pruning.

4.2 The DSSTree Maintenance Algorithms

In this section, we describe our tree maintenance algorithms. The first algorithm is on update of the frequency contribution list in a node, and the second one on periodical pruning of the *DSSTree*.

4.2.1 An Algorithm for Buffer Block Contribution in the DSSTree Node

The algorithm for buffer block contribution in the *DSSTree* node is useful, because it helps to keep consistency of the *DSSTree* during the shift of the window. Each node in the *DSSTree* maintains an array of frequency counts. The size of the array is equal to the number of buffer blocks in the buffered sliding window. A frequency count in the array corresponds to the respective buffer blocks. Some methods to maintain the frequencies in the array are described below.

Method 4.1 The 0-th index of the array refers to the contribution of the oldest buffer block that was inserted into the *DSSTree*. Similarly, the last index of the array represents the frequency contribution of the most recent buffer block. When the window shifts, the frequency of the i-th index will move to the $(i-1)$-th index, and the new frequency for the node is added to the last index. Thus, the oldest frequency is over-written when the new one is inserted. The process of copying the frequency count to the previous index can be computationally expensive—a complexity of $O(n)$ (n is the size of the array).

Note that, in the above method, each insertion causes the shifting of frequencies

for the entire array. This can be computationally expensive. For this reason we design the following method.

Method 4.2 A global oldest index (*globalOldestPtr*) keeps track of the index of the oldest buffer block frequency contribution in the array. During a shift of the window, a new frequency contribution needs to be inserted into the array. This process can be very efficient (with complexity of $O(1)$) as the *globalOldestPtr* stores the index of the oldest buffer block, so we can simply overwrite the frequency count at *globalOldestPtr*-th index with the new value. Thus, the computationally expensive shifting of frequency in the array is avoided.

Example 4.1 *In this example, we use a window with 3 buffer blocks (i.e., the frequency array in the node will be of size 3) to illustrate the frequency contribution update process. Five buffer blocks in the stream are to be inserted one after another. Each row in the figure represents an update. The globalOldestPtr is initially set to 0, and subsequently incremented such that the value loops through the index of the array in cyclic fashion. Therefore, if the globalOldestPtr goes out of bound on the array size, then it is reset to value 0. Figure 4.1 shows the change of the value of*

globalOldestPtr value	frequency count array in the node *X*	frequency count of *X* in the buffer blocks in stream
0	2 0 0	2 3 4 1 6
1	2 3 0	3 4 1 6
2	2 3 4	4 1 6
0	1 3 4	1 6
1	1 6 4	6

Figure 4.1: The update of frequency count array with the incoming buffer block

globalOldestPtr and the content of the frequency count array in an arbitrary node X. The index which has been updated with the new frequency count is shown in **bold**.

However, for both Methods 4.1 and 4.2, there can be inconsistency when there is no frequency contribution of a node X during a buffer block update. This scenario is illustrated in the example below.

Example 4.2 *The case of inconsistency is shown in Figure 4.2, where each row in the figure represents an update. Let us consider a window with 3 buffer blocks (i.e., the frequency array in the node will be of size 3). The globalOldestPtr will be updated automatically after a shift. On the first three updates, there were frequency count for the node X with values* $\{2, 3, 4\}$. *On the fourth update, the frequency count for the node X in the incoming stream data is 0, and the node is not updated because it is not traversed during tree update as there is no frequency contribution for node X during the update. Now, instead of having the frequency count set to 0, the index still contains the old frequency count (2), resulting in an incorrect frequency count (shown in* italics *in Figure 4.2). The updated counts should be* $\{0, 3, 4\}$ *but they are*

globalOldestPtr value	frequency count array in the node X	frequency count of X in the buffer blocks in stream
0	**2** 0 0	2 3 4 0 6
1	2 **3** 0	3 4 0 6
2	2 3 **4**	4 0 6
0	*2* 3 4	0 6
1	*2* **6** 4	6

Figure 4.2: The update of frequency count array with inconsistency

$\{2,3,4\}$. *The inconsistency stays there for the following updates, unless some new value overwrites the value at position* 0.

To resolve this inconsistency in frequency count update, we design Method 4.3, which is a upgrade of Method 4.2.

Method 4.3 In this method, we added some more indexing information. Now, along with the *globalOldestPtr*, we also have: (i) *localOldestPtr* for each node, which stores the local oldest index; (ii) *globalUpdateCount*, which stores the number of times the tree has been updated; and (iii) *localUpdateCount*, which stores the number of times a node has been updated. With the extra information, the algorithm can identify whether or not a node is current and can make necessary updates. The pseudocode of the node frequency update algorithm is given below (Algorithm *UpdateFrequencyList*).

Algorithm *UpdateFrequencyList*($frequency, firstUpdate$)
(* Update of frequencyList in a node *)
Input: the frequency count $frequency$ and the flag $firstUpdate$
1. **if** $localUpdateCount < globalUpdateCount - 1$
2. **then** Set zero to all those index in the frequencyList which are oldest
3. **if** ($localUpdateCount < globalUpdateCount - 1$) **or** ($firstUpdate =$ **true**)
4. **then** $frequencyList[globalOldestPtr] \leftarrow frequency$
5. **else** $frequencyList[globalOldestPtr] \leftarrow frequencyList[globalOldestPtr] + frequency$
6. $localOldestPtr \leftarrow (globalOldestPtr + 1)$ **mod** MAX_BUFFER_BLOCK
7. $localUpdateCount \leftarrow globalUpdateCount$

Line 1 indicates that the algorithm initially checks whether the current node is up-to-date with the number of global updates. If the node is not up-to-date, then (as an effect of the shift of window) the oldest buffer blocks' contributions are discarded by setting 0 to their frequency values. This will solve the inconsistency issue. In

line 3, the algorithm determines whether or not it is the first update of the node during the current incremental update (as a result of the window shift). If so, then the new *frequency* is assigned to the current oldest buffer block; otherwise, the frequency value is incremented with the new *frequency* (line 5). We use an example to illustrate how Method 4.3 works.

Example 4.3 *Let us consider a window with 3 buffer blocks (i.e., the frequency array in the node will be of size 3). The globalUpdateCount will be updated automatically after a shift, but the localUpdateCount is updated only when the node is visited. Similar to the other examples on the first three updates, there were frequency count for the node X with values* $\{2, 3, 4\}$. *On the fourth update, the frequency count for the node X in the incoming stream data is* 0, *and the node is not updated because it is not visited during tree update. Now, instead of having the frequency count set to 0, the index still contains the old frequency count (2), resulting an incorrect frequency count (shown in* italics *in Figure 4.3). However, the following updates (or any later visit to the node X), the Method 4.3 can detect the difference between the globalUpdateCount*

global Update Count value	local Update Count value	frequency count array in the node X	frequency count of X in the buffer blocks in stream
1	0	2 0 0	2 3 4 0 6
2	1	2 3 0	3 4 0 6
3	2	2 3 4	4 0 6
4	2	*2* 3 4	0 6
5	2	0 6 4	6

Figure 4.3: The update of frequency count array with fixed inconsistency

and localUpdateCount, and update the node accordingly. On the fifth update, the difference is detected, and the frequency count of the appropriate index is set to 0.

Using Method 4.3, the node frequency list update algorithm makes the nodes in the tree consistent with any shift of window. In the next section, we discuss our *DSSTree* pruning algorithm, which periodically updates the tree to remove redundant branches.

4.2.2 An Algorithm for DSSTree Pruning

In this section, we focus on the pruning of the *DSSTree* to remove redundant paths from the tree. The application of this algorithm reduces memory usage and increases efficiency in long run. During the shift of the window, a new block of data is added to the tree, and the frequency contribution from the oldest buffer block is removed. This frequency information is stored in the frequency list of each node. If a node X is not updated for BB number of times (where BB is the total number of buffer blocks stored in the window), then the frequency of X becomes 0. Note that, all the nodes in the subtree with X as root will also have a frequency count of 0.[1] Hence, the subtree is redundant in the *DSSTree*, and should be removed to reduce memory occupation, which may increase the efficiency of the *DSSTree*. The procedure for the *DSSTree* pruning is applied periodically to trim the tree.

The algorithm implemented to prune the *DSSTree* is listed in Algorithm *Prune-DSSTree*. The algorithm traverses the *DSSTree* starting at the root node, and identifies a node with frequency of 0. Once the algorithm finds a node X with

[1] As the tree maintains the basic property of a FP-tree, where the frequency count of all the children of a node X is less than or equal to the frequency count of X.

frequency count 0, then the algorithm removes the subtree starting at node X. The algorithm is recursive in nature, and it checks all the children of the current node. If a child node has non-zero frequency count, then the procedure is recursively called, with the child node as the first argument (line 3). Otherwise, the child node and its subtree are removed from the *DSSTree* and the header table is updated accordingly (line 4). In the remove subtree operation, the links to and from all the nodes in the subtree are removed, and the node themselves are freed from memory. The constraint-based stream mining system periodically calls this algorithm to prune the *DSSTree.*

Algorithm *PruneDSSTree*(*currentNode*, *header*)
(* DSSTree Pruning *)
Input: the ROOT node of the DSSTree *currentNode* and associated header table *header*
1. **for** all *child* $\in$ *currentNode.children*
2. **do if** *child.frequency* $\neq 0$
3. **then call** *pruneDSSTree*(*child*, *header*)
4. **else** Remove subtree with *child* as root of the subtree and update *header*
5. *child* = *child.sibling*

4.3 The Mining Algorithm

Our implementation of algorithm for mining frequent itemsets from data streams also supports constraint-based mining. It requires user-defined constraints to mine the frequent itemsets of interest to them. The mining algorithm follows the typical divide-and-conquer approach used in the FP-growth algorithm.

4.3.1 Constraint-Based Mining Algorithm

The *DSSTree* mining algorithm follows a divide-and-conquer approach. The basic idea is to extract valid frequent itemsets from the constructed *DSSTree*. The mining algorithm is performing constraint-based mining. Therefore, it checks for (i) itemset frequency (i.e., checking whether the frequency of an itemset is greater than or equal to the defined minimum support threshold *minsup*), and (ii) constraint validation (i.e., itemset satisfies the user-defined constraints like price of an item X is $\geq$ \$100). Itemsets satisfying both of these checks are called valid frequent itemsets. We consider to explore succinct constraints and anti-monotone constraints in our implementation.

Algorithm *ConstraintMineDSSTree*($DSSTree, HeaderTable, XProjectList$)
($*$ Constraint-based DSSTree Mining $*$)
Input: A data stream storage tree $DSSTree$, a header table $HeaderTable$ associated with the tree, and a list of items $XProjectList$ for which the projection of database is computed
Output: Valid Frequent Itemsets VFI

1. **for** all *item* $X \in HeaderTable$
2. **do** Consistency check of the current *item* X in the tree
3. **if** X is frequent **and** X is valid based on current constraint
4. **then** $VFI = VFI \cup X$
5. **if** Mining Succinct Constraints
6. **then** Extract X projected conditional database $xTDB$ from the $DSSTree$ with both upward and downward traversal
7. **else** Extract *item* projected conditional database $xTDB$ from the $DSSTree$ with upward traversal only and select only frequent itemsets which satisfies the current anti-monotone constraint.
8. $[xTree, xHeaderTable] \leftarrow BuildDSMTree(xTDB)$
9. **if** $xTree$ is **not** empty
10. **then call** $MineDSMTree(xTree, xHeaderTable, XProjectList \cup item)$

Below we have the MineDSMTree procedure, which is used to mine the $\{X\}$-projected tree.

Procedure *MineDSMTree*$(DSMTree, HeaderTable, XProjectList)$
$(*$ To mine the $\{X\}$-projected tree $*)$
Input: A data stream mining tree $DSSTree$, a header table $HeaderTable$ associated with the tree, and a list of items $XProjectList$ for which the projection of database in computed
Output: Valid Frequent Itemsets VFI

```
1.  for all itemX ∈ HeaderTable
2.      do if X is frequent
3.          then FI = FI ∪ {XProjectList ∪ X}
4.               Extract X projected conditional database xTDB from
                 the DSSTree with only upward traversal
5.               (* During upward tree traversal selects all frequent items. *)
6.               (* Build a xTree project DSMTree and xHeaderTable from the
                 xTDB *)
7.               [xTree, xHeaderTable] ←BuildDSMTree(xTDB)
8.               if xTree is not empty
9.                 then call MineDSMTree(xTree, xHeaderTable,
                                          XProjectList ∪ item)
```

In the mining process, a significant computation is required on generating projected trees. A projected tree is a smaller tree extracted from the *DSSTree*. A projection is done on a valid frequent itemsets X such that all valid frequent itemsets X' (where $X \subseteq X'$) are extracted from the *DSSTree*. The algorithm constructs an X-projected *DSMTree* with the extracted set of itemsets also called X-conditional database. Line 1 of Algorithm *ConstraintMineDSSTree* shows each item in the header table is processed. With the generation of projected trees for all valid frequent itemset X, the mining algorithm can extract the complete set of valid frequent itemsets from the *DSSTree*.

In line 2, the *itemX* is checked for consistency of its frequency count. This is necessary because after an incremental update with a newly arrived buffer block from

the stream, not all the nodes in the tree are updated. As a result, inconsistent values may be found in the frequency count list in a node of the tree. During consistency checking, if any node is found inconsistent (in other words, not up to date with the recent shift of the window), then the node is updated according to Algorithm *UpdateFrequencyList*. Thereafter, X is checked for frequency and validity based on the *minsup* and the constraints, respectively (line 3). Thus, only valid frequent itemsets are selected for projection.

Now, X is added at the global set of output valid frequent itemset VFI. The next step is the extraction of the X-projected conditional database. In the itemset extraction process, the algorithm starts from the header table record containing X; then, it uses the *node_link* to traverse all the tree paths containing X. All the items in a path are not extracted because the path may contain frequent, infrequent, valid, and invalid items[2]. The extraction method is different for the two types of constraints (succinct and anti-monotone) that the algorithm handles.

For succinct constraints, both upward and downward traversals of a path are required (line 6). An upward traversal means that the algorithm checks all the ancestor nodes of the current node (i.e., the node containing X). On the other hand, a downward traversal means that all the descendants of the current node are checked. As mentioned earlier, in succinct constraints, a valid itemset can have *optional* items (which do not satisfy the constraint) as long as there is at least one item from each *mandatory* group (which satisfies the succinct constraint). During the upward traversal of the path, the algorithm selects all frequent valid, and frequent invalid items. In

[2]An item is determined frequent or infrequent based on the current frequency support threshold set by the user. Similarly, an item is judged valid or invalid based on the current user-defined constraints.

the downward traversal, only the invalid frequent items are selected. No valid item is selected in the downward traversal. This is because, for any valid frequent item Y, which is descendant of X, the combination of that item with X will be considered during the extraction process of Y (i.e., during upward traversal of Y).

In contrast, for anti-monotone constraints, only upward traversal is necessary (line 7). Furthermore, only valid frequent items are selected during the extraction process of anti-monotone constraints, because there is no optional portion of an itemset which satisfies an anti-monotone constraint.

Once the itemsets are extracted, an X-projected *DSMTree* is built (Algorithm *ConstraintMineDSSTree*, line 8). If the X-projected tree is not empty, then in line 10, the *MineDSMTree* is invoked. The major differences of the *MineDSMTree* procedure when compared to *ConstraintMineDSSTree* are: (i) it uses a different node structure; (ii) it appends X with the *XProjectList* when adding to the *VFI* (line 3); and (iii) in the extraction process, only upward traversal is required and the items are checked for frequency only (line 4).

4.4 Summary

In this chapter, we described the algorithms that are necessary to *build*, *maintain*, and *mine* the *DSSTree*. In addition to efficient incremental updates, the *DSSTree* building algorithm showed how the *DSSTree* node data structure efficiently captures items and item frequency. The item frequency is stored in the frequency contribution list of each node, which represents the contribution of each buffer block in the buffered sliding window. The maintenance algorithm performs the node frequency update dur-

ing the shift of the window. Moreover, the pruning algorithm is applied periodically on the *DSSTree* to remove all the redundant tree paths, and thus reducing memory consumption and increasing efficiency in the long run.

The *DSSTree* mining algorithm uses a recursive mining procedure and extracts frequent itemsets based on user-defined constraints (e.g., succinct and anti-monotone). The *DSMTree* structure, which is a variant of the *DSSTree* used in $\{X\}$-projected tree mining, increases the efficiency and reduced the memory usage of the mining process. Our algorithm descriptions showed the effective design and implementation of the major code segments. Moreover, the implementation covers the necessary features of a constraint-based frequent itemset mining from data streams.

In the next chapter, we will empirically evaluate our algorithm to measure its effectiveness. To be more specific, we will observe the effects on execution time of the algorithm with the change of various mining parameters like changes in minimum support threshold, changes in data size, changes in number of buffer blocks of the buffered sliding window, and changes in the selectivity of the applied constraints.

Chapter 5

Experimental Results

In this chapter, we experimentally evaluate our *DSSTree* in order to better understand the advantages and disadvantages of the algorithm, in terms of its performance and applicability.

The *DSSTree* exploits the simple yet powerful canonical ordering technique and implemented the concept of the buffered sliding window. These enable the *DSSTree* to efficiently process incremental data from continuous data streams. The *DSSTree* maintains all the data of the current window, and thus avoids the use of approximation methods for estimation of the frequency of the itemsets. The *DSSTree* mining algorithm achieves fast, interactive, and constraint-based mining from data streams. The constraint-based mining technique effectively prune unwanted itemsets based on applied frequency threshold and constraints. This will provide users with itemsets of interest to them. The interactive mining feature enabled user to make change of mining parameters (e.g., minimum support threshold, constraints) during the mining process.

To justify our claims, we designed six sets of experiments. The algorithm was run using both synthetic data and real-life data. The environment setup of the experiments are described in the next section. The remaining sections discuss details of each of the experiments and their outcomes.

5.1 Experimental Setup

We implemented the constraint-based data stream mining algorithm in the C programming language. Experiments were conducted on a Pentium-III machine with a 1GHz processor, 512MB memory and 20GB hard drive. As usual to most data mining experiments, we have used the synthetic database generated by the program developed at IBM Almaden Research Center [AS94], real-life databases from the UC Irvine Machine Learning Repository [BM98], and Frequent Itemset Mining Dataset Repository [GZ03a] as the testing databases, because they are considered to be *benchmark datasets* in the field of data mining. The IBM dataset was generated with 1000 distinct items and average length of a transaction is 10. From the UC Irvine Machine Learning Repository, we used the mushroom dataset which has 137 distinct items, and fixed length transactions of 22 items each.

Various forms of tests can be conducted on these datasets to determine the execution time, scalability, and memory occupancy of the *DSSTree* building and mining phases. In particular, we have conducted the following sets of experiments:

1. The runtime with various minimum support thresholds ranging from 0.01% to 0.5% for the IBM Almaden datasets, and from 3% to 10% for the mushroom datasets were examined.

2. Tested scalability with the change of window size ranging from 100,000 to 1,000,000 transactions for the IBM Almaden datasets, and from 800 to 8,000 transactions for the mushroom datasets.

3. The effect in the runtime with the change of the number of buffer blocks in the buffered sliding window with a fixed window size was tested. The number of buffer blocks was changed from 2 to 10 blocks. We also observed the effect of tree pruning in the mining time.

4. Observed the runtime with the change in constraints with various selectivity. A constraint with $p\%$ selectivity means, $p\%$ of items are selected (i.e., satisfying the constraint). Therefore, the higher the selectivity, the higher the expected number of itemsets to be returned.

5. The number of nodes generated by the *DSSTree* was monitored to estimate the memory usage.

6. Tested interactive mining with increasing and decreasing minimum support threshold.

5.2 Experiment Set 1: Testing the Effect of Minimum Support Thresholds on Runtime

In this experiment set, we focused on the effect on runtime of the *DSSTree* building and mining with respect to the change of minimum support threshold (*minsup*). We used both synthetic data (IBM dataset) and real life data (mushroom dataset) for

the experiments. To measure the effect of minimum support thresholds on runtime, we kept the streaming parameters (e.g., number of buffer blocks, sliding of window) of the *DSSTree* fixed.

Experiment 5.1 In this experiment, we used the IBM dataset. We considered the size of the window to be 1,000,000 transactions, with 1 buffer block in the window and no update from the stream was considered (as this might change the items frequency on the dataset). We ran the experiment with the *minsup* ranging from 0.01% to 0.5%. The y-axis of Figure 5.1 shows the runtime, and the x-axis shows *minsup*.

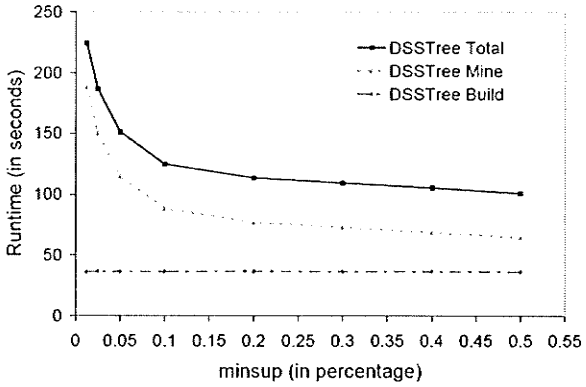


Figure 5.1: Runtime with respect to minsup for the IBM dataset (Experiment 5.1)

As expected, when the *minsup* was increased, fewer items satisfied the *minsup*, thus fewer items were selected for mining. An increase in *minsup* reduced the total mining time. Specifically, from Figure 5.1, we observed a high runtime with a low minsup of 0.01% and a quick dive in the runtime with the minsup 0.1% and higher. The reason for the steep slope is that the number of frequent itemsets increased exponentially with the decrease of minsup. We also noticed that there was no effect on the *DSSTree* building time. This is because the tree building process stores all

items in transactions in the current window irrespective of their frequencies (i.e., frequent or infrequent).

Experiment 5.2 For the mushroom dataset, we considered the size of the window to be 8,000 transactions, and same window setup as in Experiment 5.1. We ran the experiment with the *minsup* ranging from 3.0% to 10.0%. The y-axis of Figure 5.2 shows the runtime, and the x-axis shows *minsup*. Similar to Figure 5.1, Figure 5.2 also shows a reduction in the runtime with an increase in *minsup*. However, the graph is less smooth than that of Figure 5.1. The bumps occurred because of the distribution of items in the mushroom dataset, which is unknown, whereas in case of synthetic IBM dataset we know that it follows the Poisson distribution.

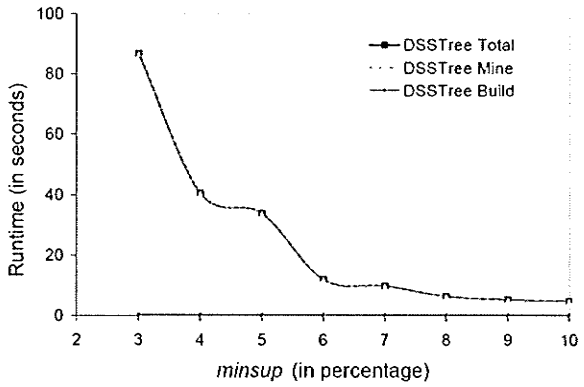


Figure 5.2: Runtime with respect to minsup for the mushroom dataset (Experiment 5.2)

5.3 Experiment Set 2: Testing the Effect of Window Sizes on Runtime

In this section, we consider the scalability of the *DSSTree* algorithm in terms of the change in window size. Similar to experiments in previous section, we kept the other stream mining factor constant in this set of experiment.

Experiment 5.3 For this experiment, we have changed the size of the dataset and kept the *minsup* constant at 0.0125% for the IBM dataset. The window size was changed from 100,000 to 1,000,000 transactions. The y-axis of Figure 5.3 shows the runtime, and the x-axis shows the window size in number of transactions. As expected, runtime increased with the increase of window size. Specifically, Figure 5.3 depicts a gradual increase of the runtime with the increase of the dataset size, and maintained a linear scale-up. The size of data has the same effects on both the *DSSTree* building and mining time.

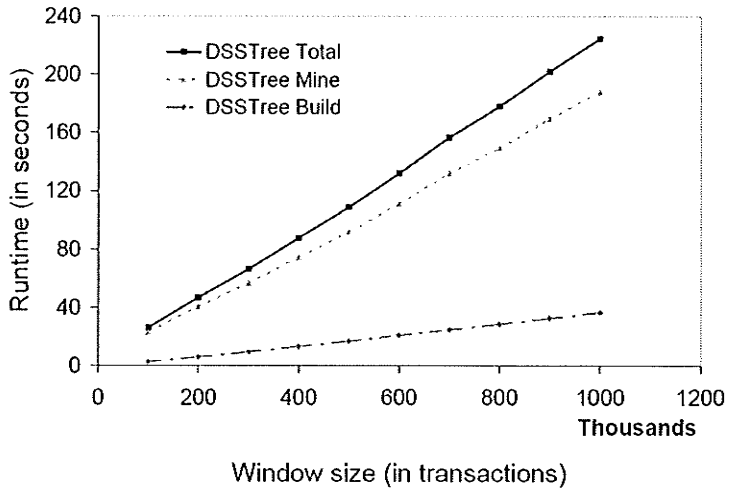


Figure 5.3: Runtime with respect to window size for the IBM dataset (Experiment 5.3)

Experiment 5.4 For the mushroom dataset, we kept the *minsup* constant at 10% and changed the window size from 800 to 8,000 transactions. Figure 5.4 shows the change in the runtime for the mushroom dataset. This experiment has a very interesting outcome. The key reason behind this is the unknown distribution of items in the dataset. Several bumps on the graphs are clearly seen with the increase in window size. For this experiment, we have used two variants of the *DSSTree* algorithm: one with the "single-path" optimization (labelled DSSTree) and other one with no optimization (labelled DSSTree Non-opt).

Figure 5.4: Runtime with respect to window size for the mushroom dataset (Experiment 5.4)

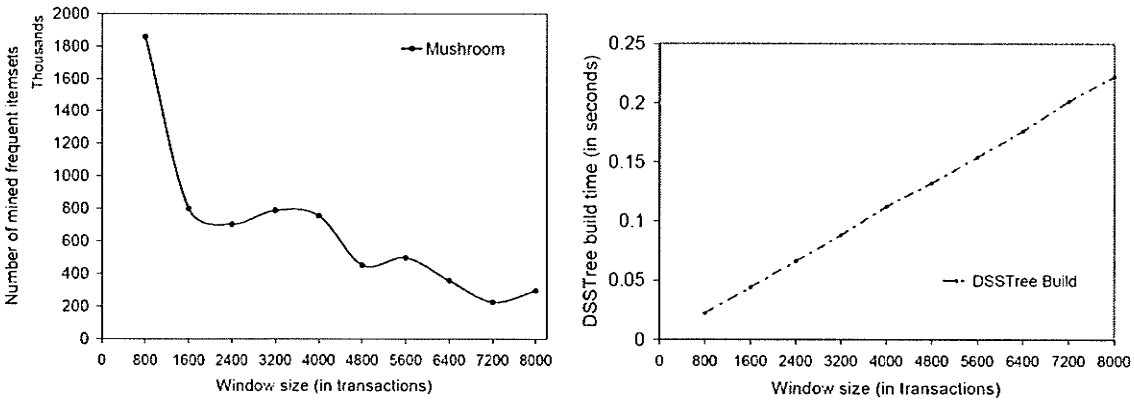
In Figure 5.4, we have plotted graphs for both the regular *DSSTree* and the Non-opt *DSSTree* (which is the non-optimized *DSSTree*). During the analysis of this experiment, we spotted that there can be a significant effect of mining optimization for single path projected trees. This "single-path" optimization was proposed by Han et al. [HPY00] for the FP-growth algorithm. The idea behind this optimization is that if the tree to be mined has a single path, then instead of generating a projected tree to compute all the frequent itemsets, the algorithm can enumerate all the frequent

itemsets from the path. Figure 5.4 shows the drastic drop in the runtime between the *DSSTree* and the Non-opt *DSSTree*. However, the optimized algorithm has uneven bumps resulting from the unknown distribution of frequent itemsets in the dataset. We further investigated this in the next experiment to be sure that the distribution of itemsets in the mushroom dataset is causing this trend.

Experiment 5.5 We used the same run configuration as in Experiment 5.4. We plotted the graph (Figure 5.5(a)) with the number of frequent itemsets generated in y-axis and number of transactions in x-axis. If the distribution of items is uniform in a dataset, then the increase in window size should also increase the number of frequent itemsets. It may behave otherwise for unknown distribution.

The pattern in the graph (Figure 5.5(a)) shows a similar trend to that of the Non-opt *DSSTree* in Figure 5.4. It is due to the uneven distribution[1] of frequent itemsets

[1]In this particular trend (Figure 5.4), the distribution of percentage frequency count of itemsets



(a) Change in number of output frequent itemsets with respect to the change in data

(b) Change of runtime with respect to the change in window size

Figure 5.5: Effects with the change in window size for the mushroom dataset (Experiment 5.5)

in the mushroom dataset which is affecting the tree mining time.

We also noticed that the tree building time was very small compared to the mining time. This is understandable as the window size was varied from 800 to 8,000 transactions (affects the building time), which generated frequent itemsets varied from 200,000 to 1,800,000 (affects the mine time). The total runtime reflected the trend of the mining time. To clarify this, we plotted a graph for the *DSSTree* building time only (Figure 5.5(b)). The figure shows that the *DSSTree* building time linearly scales with the increasing window size. The reason is that tree size is depended on the window size (than on the number of output frequent itemsets).

5.4 Experiment Set 3: Testing the Effect of Stream Parameters on Runtime

In this set of experiments, we performed experiments on the stream mining parameter: the *change of the number of buffer blocks* in the window. This is the key factor affecting the runtime as it determines the update time, tree pruning time, update intervals, and degree of window shift/slide. The effect of periodic tree pruning was also observed. All the cases evaluated in this section uses a *fixed window size* of 500,000 transactions. We only used the IBM dataset for this set of experiments as we needed huge data to observe the effect of continuous data stream in the *DSSTree* building time, updating time, and pruning time.

in a data can suddenly increase or decrease with the increase in window size (i.e., data size). This results in fluctuation in the output frequent itemsets with increase in window size.

Experiment 5.6 First, we measured the runtime to build a *DSSTree* with changes in the number of buffer blocks in the window. The number of buffer blocks with respect to the size of the window determined the new block of data that was to be updated. For a fixed sized window, the increase in the number of buffer blocks would reduce the size of each buffer block. Thus, an increase in the number of buffer blocks would lead to faster tree update with new incoming buffer blocks from the stream. However, maintaining a larger number of buffer blocks could also become an overhead, because each node in the tree needs to maintain the frequency contribution list and will occupy more memory space. This is a *trade-off* that the user needs to decide between: (i) shorter update intervals and smaller update time, but more memory consumption; and (ii) longer update intervals and less memory consumption, but higher update time.

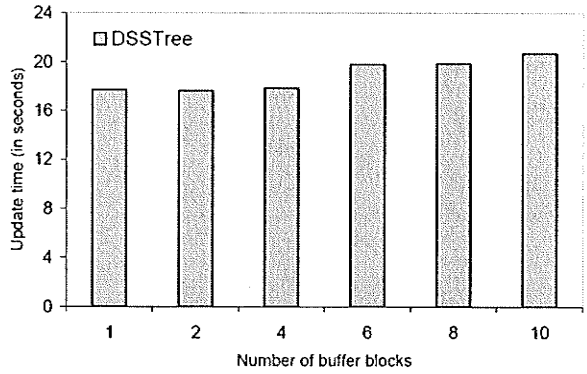


Figure 5.6: Effects on window fill time with the change in the number of buffer blocks (Experiment 5.6)

For this experiment, we had a fixed window size of 500,000 (number of transactions). We changed the number of buffer blocks in the window. The runtime shown in Figure 5.6 is the time our algorithm took to fill a window size of 500,000 (i.e., pro-

cess 500,000 transactions to build the *DSSTree*), where BB number of buffer blocks ranged from 1 to 10. Figure 5.6 shows that there was a slight increase in the runtime with a higher number of buffer blocks. The increase in the runtime was reasonable as it only increased by 1.169 times if we compare $BB = 1$ with $BB = 10$. This indicates that there are some overheads in the building process with a larger number of buffer blocks. However, this slight increase of time was compensated by higher efficiency in the long run of the algorithm when continuous streaming data was processed (shown in Experiment 5.7).

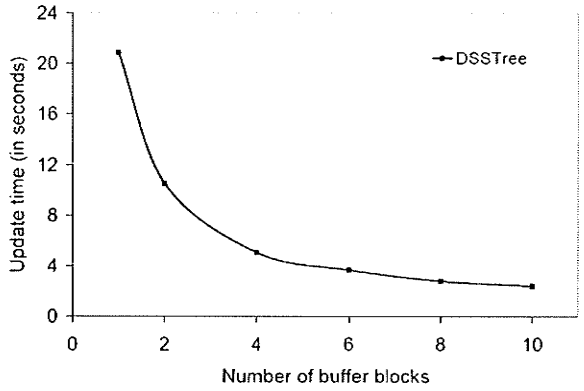


Figure 5.7: Effects on DSSTree update time with the change in size of buffer blocks (Experiment 5.6)

Subsequently, we also measured the average update time required for the sliding windows with different numbers of buffer blocks. The y-axis of Figure 5.7 shows the runtime of tree update, and the x-axis shows the number of buffer blocks in the window. As mentioned earlier, update time was expected to decrease with the increase in number of buffer blocks. Figure 5.7 shows exactly the expected trend in the runtime. There was a sharp decrease in update time with the increase in number of buffer blocks.

Experiment 5.7 In this experiments, we checked for the long term effect on the tree building time. We found in Experiment 5.6 that, up to a certain point, increasing the number of buffer blocks helps achieve faster buffer block updates. In the continuous stream mining process a periodic tree pruning is necessary. Tree pruning is required to remove redundant tree paths from the tree, which are effects of shifts of the window.

The data processing includes the tree building/update time and the tree pruning time. If the tree is not pruned, then it may reduce mining efficiency in the long run. The question arises whether it would be effective in the long run, where a periodic tree pruning algorithm is invoked to prune the redundant tree paths. The tree pruning algorithm can take considerable runtime, because it has to recursively traverse paths in the tree to perform the pruning process (The algorithm, called *PruneDSTree*, was described in Section 4.2.2).

For this experiment, we had a fixed window size of 500,000 (the number of transactions). We changed the number of buffer blocks in the window. We recorded time to process 1,000,000 transactions from the continuous data stream. Hence, we process the same amount of data for all the windows with different numbers of buffer blocks.

As the window size was set to 500,000, the first 500,000 transactions from the stream would fill the window. During that time there would be no shift of window and no pruning is required. After that, the remaining 500,000 transactions in the stream would be processed in blocks of transaction where

$$BlockSize = Window\_Size/Number\_of\_BufferBlocks.$$

Changing the number of buffer block will also affect the streaming processing.

Figure 5.8 shows there can be a significant increase in efficiency with the larger

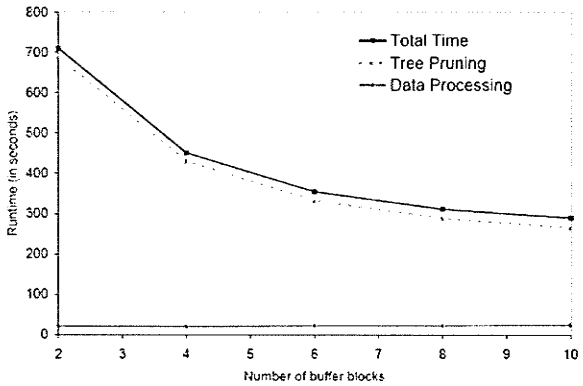
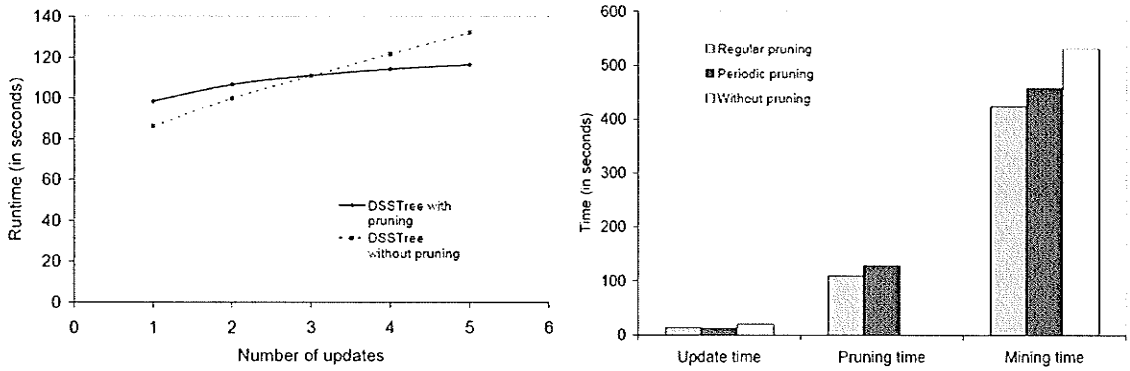


Figure 5.8: Effects on DSSTree stream processing time with the change in size of buffer block (Experiment 5.7)

number of buffer blocks. The key reason behind this increase in performance is the effect of the new buffer block on the data in the current window. A large number of buffer blocks BB reduces the size of each buffer block as we have a fixed window size. Smaller buffer block size has less effect on the trend of the items in the current window. This prevents drastic change of the data in the current window. During the shifting of the window with a new buffer block, the oldest buffer block is removed from the window. With smaller buffer block size, a fewer itemsets are likely to be removed from the tree. This results in less traversal of the tree paths by the pruning algorithm. Thus, even after cumulating, the pruning time of a window with larger number of buffer blocks, performs significantly better than the window with lesser buffer blocks (Figure 5.8).

From our observation on this experiment, we found that a window with larger number of buffer blocks performs better in the long run and also provides faster tree updates with the new buffer block. It provides a data processing rate (which is the update of tree—*no mining*) of about 41,494 transactions per second (without prun-



(a) Change of runtime with/without pruning on updates

(b) Change of runtime with/without pruning

Figure 5.9: Effects on runtime for pruning (Experiment 5.8)

ing), and about 3,451 transactions per second (with pruning). As pruning somewhat reduces the data processing, which does not include the mining time, we examined the effect of pruning to the mining time in the next experiment.

Experiment 5.8 In this experiment, we observed the effect of pruning of the *DSSTree* on the mining time. We ran with the window size set to 500,000 transactions and number of buffer blocks set to 10, and with the *minsup* of 0.0125%. We ran our *DSSTree* in two modes: (i) with pruning after every update (*DSSTree with pruning*) and (ii) without any pruning (*DSSTree without pruning*).

In Figure 5.9(a), we observed that the runtime for the *DSSTree without pruning* is better than *DSSTree with pruning* in the first few updates. However, in the long run (after 3 updates), *DSSTree with pruning* performs better because the *DSSTree* gets bigger (if not pruned), which affects the update time and the mining time.

Figure 5.9(a) shows the effect of pruning on the tree update time and mining time in the long run (in this case, after processing 750,000 transactions from the stream).

For this run, we added another variation of the *DSSTree*, which performs periodic pruning. We labelled the different runs as follows: (i) *Regular pruning* for the run with pruning after every update; (ii) *Periodic pruning* for the run with application of pruning after every even number of updates; and (iii) *Without pruning* for the run with no pruning. Firstly, we found the update time for *Regular pruning* and *Periodic pruning* are equal, but higher value for *Without pruning.* Secondly, the pruning time for the *Periodic pruning* is higher than *Regular pruning*, whereas there is no pruning time for *Without pruning.* Lastly, the mining time for both *Periodic pruning* and *Without pruning* is much higher than *Regular pruning. Periodic pruning* did not help because of high pruning and mining time. Out of the other two, the *Regular pruning* is better than the *Without pruning* one with average streaming data processing and mining rate of 1,319 transactions per second for the former and 1,309 transactions per second for the latter. The rate of data processing and mining showed only slight improvement for the *Regular pruning* one, however, it is much better in overall terms as it reduces memory consumption.

5.5 Experiment Set 4: Testing the Effect of Constraints on Runtime

This set of experiments involve the evaluations regarding the constraint-based stream mining. In this case, we compare CanTree [LKH05] with our *DSSTree.* We added the constraint-based mining feature with CanTree as a post processing, and we call it *CanTree++.* Choosing CanTree for comparison was logical as it uses the same

idea of canonical ordering of tree items as the *DSSTree*. For this set of experiments, we kept the stream mining factors of the algorithm constant (as CanTree++ does not support data stream mining).

For this set of experiments, we used the IBM dataset with the window size fixed at 1,000,000 transactions and 1 buffer block. We have also fixed the minimum support threshold to 0.0125% to better understand the effect of constraints. The changing factors for this set of experiments were: (i) the type of constraints, succinct constraint and anti-monotone constraint; (ii) the selectivity of the constraints. A constraint with p% selectivity means, p% of distinct items in the current window are selected (i.e., satisfying the constraint) for mining. Therefore, the higher the selectivity, the higher is the expected number of itemsets to be returned (and the longer would be the expected runtime).

Experiment 5.9 For this experiment, we selected the succinct constraint with selectivity ranging from 10% to 100%. The y-axis of Figure 5.10 shows the runtime, and the x-axis shows the p% of selectivity. Figure 5.10 shows that at lower selectivity there is a huge gain in performance of the constraint-based *DSSTree* mining and *CanTree++*; there is about 2.87 times reduction in the runtime by the *DSSTree* algorithm at selectivity 10%. The gap between the two algorithms closes with the increase in the selectivity. The result of this experiment clearly outlines the power of constraint-based mining in terms of the performance gain.

Experiment 5.10 With a similar experimental setup as in Experiment 5.9, we ran with anti-monotone constraint. The results for this case showed even better gain in

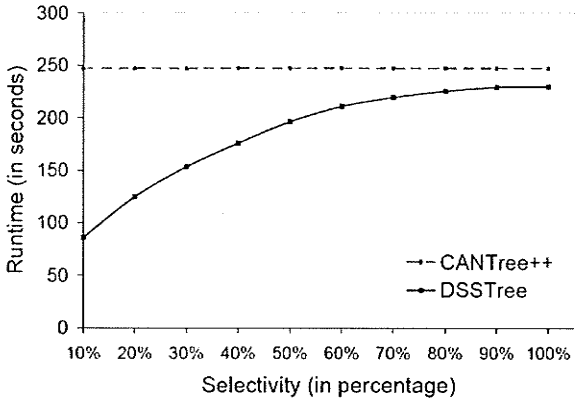


Figure 5.10: Changing selectivity for succinct constraint (Experiment 5.9)

performance compared with the *CanTree++* (Figure 5.11). The pruning power for the anti-monotone constraints are more strict than the succinct constraints, resulting in a greater degree of pruning and faster execution. If we recall the algorithm description, the processing of the succinct constraints involved some upward and downward traversal of the *DSSTree*. On the other hand, there is only upward traversal required by the anti-monotone constraints. This is reflected in the results that with the same $p\%$ of selectivity the anti-monotone constraints running was 2.15 times faster than the succinct constraint mining with selectivity set to 10%. Thus, the application of both the constraints successfully enhanced the performance of the *DSSTree* mining. Not only that, it also outputs frequent itemsets of interest to users.

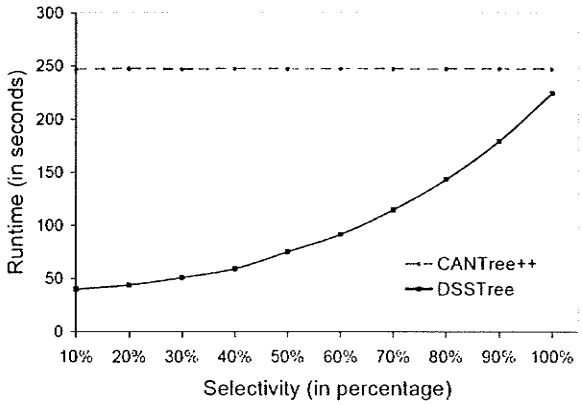


Figure 5.11: Changing selectivity for anti-monotone constraint (Experiment 5.10)

5.6 Experiment Set 5: Testing the Memory Usage of the DSSTree

For this experiment, we measured the memory usage of the *DSSTree* and compared it with the other frequent itemset mining data structures such as CanTree and FP-tree. As the *DSSTree* captures all the itemsets (both frequent and infrequent) in the current window of the streaming data, it tends to be quite memory hungry. Furthermore, there is a extra storage required for each *DSSTree* node to store the frequency contribution list (necessary to store the frequency of each buffer block). Therefore, a node in the *DSSTree* occupies more memory than the other data structures in comparison. Hence, we expected that the *DSSTree* would occupy more memory.

Experiment 5.11 We used the IBM and the mushroom datasets as experimental datasets. We used 8000 transactions (fixed window size) for both datasets in the experiment and set the minimum support to 1 so that all items would be selected

by the other algorithms. For *DSSTree* the number of buffer blocks in the windows was set to BB=10 buffer blocks. The amount of memory consumption may vary for different datasets because the overlapping of itemsets determines the compression of the tree. Also to note that the factor of overlapping for the *DSSTree* and CanTree depends on their canonical ordering (e.g., lexicographical order), whereas for FP-tree it depends on the frequency order.

As seen in Table 5.1, in both types of datasets, the *DSSTree* consumed more memory. In addition, the degree of memory usage between the *DSSTree* and the others changed for the different datasets. In the IBM dataset, the *DSSTree* memory consumption was 1.428 and 1.476 times when compared to CanTree and FP-tree, respectively. On the other hand, for the mushroom data, the *DSSTree* memory consumption changed to 1.428 and 1.047 times of CanTree and FP-tree, respectively.

Table 5.1: Bytes of memory occupied by different trees

	Memory usage in number of bytes		
Dataset	*DSSTree*	CanTree	FP-tree
IBM	2,757,080	1,929,956	1,867,712
Mushroom	794,520	556,164	758,380

To further investigate the drop in memory usage of the *DSSTree* when compared to the FP-tree for the mushroom data, we extracted the number of nodes occupied by each tree. Referring to Table 5.2, we can see the number of nodes in a *DSSTree* and FP-Tree are fairly close for the IBM dataset. In fact, for the mushroom dataset the number of nodes in a *DSSTree* is smaller than FP-Tree. This indicates that the *DSSTree* path overlapping and compression is good enough to even produce tree

smaller than FP-Tree. Thus, we conclude that the *DSSTree* may consume more memory, but the amount is reasonable and not exceedingly high compared to the other frequent itemset mining data structures.

Table 5.2: The number of nodes in different trees

	Number of nodes in tree	
Dataset	*DSSTree*	FP-tree
IBM	68,927	66,704
Mushroom	19,863	27,085

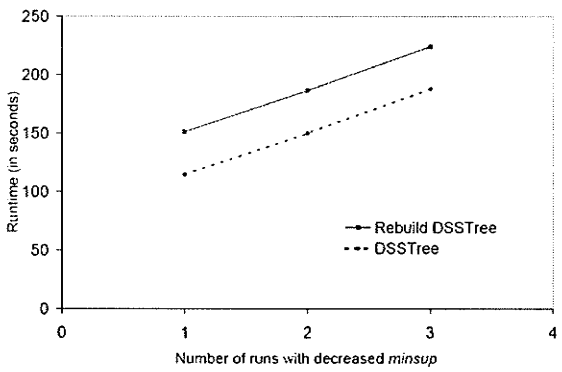
5.7 Experiment Set 6: Testing the Effect of Interactive Mining

In this experiment, we tested the interactive mining feature of our *DSSTree*. In interactive mining, the user is allowed to change the mining parameter like minimum support threshold at anytime in the mining process. An algorithm which supports interactive mining (e.g., DSSTree) works in the following steps: (i) construction of a data structure to store transactions, also known as *tree building*; (ii) mining of data structure based on some parameters (e.g., minimum support threshold *minsup*) to generate output; and (iii) re-mining of the data structure with updated parameters. An alternative to this process is to rebuild the tree then mine in Step (iii).

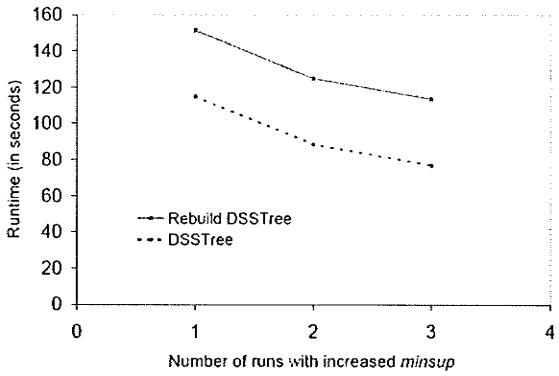
Experiment 5.12 In this experiment, we used a variant of the *DSSTree*, called Rebuild DSSTree, to simulate the alternative process for interactive mining. For both

figures in Figure 5.12, the y-axis shows the runtime, and x-axis shows the number of runs with changed *minsup* values, where the *minsup* decreases for Figure 5.12(a) and the *minsup* increases for Figure 5.12(b). We did not consider Step (i) of the mining process, we assumed that the tree has been built for the first time. Both figures in Figure 5.12 clearly show that the interactive mining of the *DSSTree* is faster than Rebuild DSSTree. Thus, whether the *minsup* is decreased or increased, the *DSSTree* provides fast interactive mining.

Moreover, in case of streaming data mining there is no option (unless the entire streaming data is archived in some secondary data structure) to rebuild the tree as the data can only be scanned once.



(a) Change of runtime with decreasing threshold



(b) Change of runtime with increasing threshold

Figure 5.12: Effects on runtime for interactive mining (Experiment 5.12)

5.8 Summary

The experimental results presented in this chapter showed that the *DSSTree* can efficiently mine constraint-based frequent itemsets from data streams. Our algorithm linearly scales with the change of minimum support threshold and window size. Though memory consumption by the *DSSTree* may be a bit high, the efficient mining functionality and fast constraint-based mining from data streams outweighs the issue of memory usage. In addition, with the advancement of technology, memory is not the key issue. Likewise, other existing algorithms [PHM00, CZ03, LKH05] considered the same assumption that the algorithm have enough main memory space to fit the applied data structure.

The constraint-based mining feature enabled users to extract itemsets of interest to them. Not only that, the application of constraints in the mining process have enhanced the mining performance by several times. On comparison with one of the existing algorithms, with constraints applied as post-processing, we found our algorithm outperforms others. Our algorithm achieved a high streaming data processing and mining rate of 1,319 transactions per second, with the number of buffer blocks set to high values and with regular tree pruning. The experimental results showed the *DSSTree* data structure and the algorithm are effective and efficient for constraint-based frequent itemset mining form data streams.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The problem of frequent itemset mining from data streams has come to the attention of many researchers in the recent years, and has brought some new challenges. One major issue focuses on how to manage and mine the continuous bulk of data from the data stream. To address this, it is necessary to incrementally update the existing data with the incoming data from streams, and to maintain the most recent portion of data as users are more interested in recent data. The data stream mining algorithms maintain recent data not only because of user's interest, but also for the reason that it becomes infeasible to maintain the entire collection of streaming data as it can be of infinite length.

Besides these two issues, we found that the streaming data can still be huge and may generate an extremely large number of frequent itemsets. In this regard, Chi et al. [CWYM04] proposed an algorithm to mine closed frequent itemsets from the

data stream to reduce the output frequent itemsets. However, this does not necessarily produce output itemsets that are of interest to the user. From all these issues, we identified three key questions in the area of mining frequent itemsets from data streams. The questions are: (i) How can an efficient incremental mining technique that handles continuous incoming data from streams be developed? (ii) What type of windowing technique can be used to capture the most recent data from data streams? The sliding window techniques or others? and (iii) How can users mine for frequent itemsets that are of interest to them? To address all these questions in our research we developed a constraint-based algorithm for mining frequent itemsets from data streams.

Our developed algorithm uses our novel data structure called the Data Stream Storage Tree (*DSSTree*). In this thesis, we also designed a buffered sliding window technique to maintain recent data from the data stream, which is a key feature for data stream mining as the incoming data from the stream can be of infinite length. The *DSSTree* keeps the items from the transactions (from the stream) in a canonical order, and uses the concept of the buffered sliding window technique. This enabled efficient processing and incremental update of data from data streams. The usability of interactive mining can also be explored in the *DSSTree*, because it maintains all items (frequent and infrequent) in the tree. In interactive mining the user is allowed to change the values of mining parameter, like minimum support threshold and constraints, any time in the mining process.

Moreover, existing data stream mining algorithms have not applied the powerful features of constraint-based mining as our algorithm does. Constraint-based mining

not only reduces the search space but also provides user control over the mining process to extract the frequent itemsets of interest to them. These developed features have been experimentally evaluated and justified.

From the experimental results, we observed that our *DSSTree* building and mining algorithms linearly scaled with the change of minimum support threshold and data size. Although our algorithm has higher memory consumption, our algorithm can efficiently mine constraint-based frequent itemsets from data streams. Results showed that our algorithm provides fast data stream processing and mining. The application of constraints in the mining process has enhanced the mining performance by several times when compared with an existing algorithm (with constraint applied as post processing).

To summarize, our key contributions of this thesis are:

1. We introduced a novel stream mining data structure, called a Data Stream Storage Tree (*DSSTree*), which can efficiently update data from continuous data streams. The *DSSTree* used the buffered sliding window technique to capture only recent data from streams.

2. The *DSSTree* maintains all the items (frequent or infrequent) in the current window. As a result, it avoids the usage of approximation methods to get frequency count of itemsets. This property also enabled interactive mining on the streaming data.

To the best of our knowledge, our mining algorithm is the first one to incorporate constraint-based frequent itemset mining in data stream mining.

We concluded that our *DSSTree* algorithm can efficiently and effectively achieve constraint-based frequent itemset mining from data streams and can provide better performance and usability than the existing algorithms on frequent itemset mining from data streams.

6.2 Future Work

For the future work in this research, we plan to further improve the efficiency of our algorithm. To be specific, we plan to work on the following: (i) compression of the *DSSTree* to reduce memory consumption and further improve mining time; (ii) incorporation of incremental update of discovered frequent itemsets (as currently, we only focus on incremental updates of the data) by trying to make use of the already mined frequent itemsets as a priori knowledge to the mining procedure; and (iii) exploration of some real-life applications of the developed system such as analysis of data for wireless sensor networks, and disease outbreak detection using medical data from emergency departments.

Here, we discuss some of our thoughts about the above mentioned future work. First, we plan to work on the compression of the *DSSTree*. Why compression is required? For sparse dataset the transactions tend to overlap less resulting large tree structure. It is true that we assume that the system memory is large enough to fit the data structure we are using, but too large structure may reduce efficiency. Therefore, we have to device a way to compress the current *DSSTree* data structure. There are some existing ways to compress prefix tree structure where node with same property can be collapsed together into one special node structure. Hence, we can further think

in that line and derive a more compressed data structure.

Second, we plan to work on incremental update of discovered frequent itemsets. The main idea behind this is that, after every mining process we get some output frequent itemsets; is there any way to make use of these frequent itemsets for future mining on the data? This can be a very interesting research to work because it may seem quite easy but there are challenges to overcome. One of the challenge is if the minimum support is reduced (assuming the data is not changed), then all the frequent itemset found must still be frequent and in addition there will be some new frequent itemsets. How to efficiently extract this new set of frequent itemset?

Finally, we also plan to explore the application area of data stream mining. For example, disease outbreak detection systems. Where data are continually recorded at the emergency department, which is a form of incoming data. These data can be used to identify possible changes on the trend of admitting patients. To elaborate, the emergency department record data can be divided into two distinct set of records labelled *baseline (historic) data* and *current data.* The current data can be records of the past 24 hours or past 7 days. On the other hand, any data prior to the current data is considered as baseline data. The period of the baseline data can be records of one year or one season.

To this end, our *DSSTree* can be used to maintain these data sets in two separate trees. There can be a current data tree and a baseline data tree. The current implementation of the sliding window can be modified to capture data based on time frame. Furthermore, the data removed from the current data tree will be added to the baseline data tree. Once, we have both the data trees ready, it can be analyzed

to detect possible disease outbreak in the current data when compared with baseline data. The detection method can be achieved with the aid of some data mining algorithms and statistical tests.

For example, we can use association rule mining algorithms for the anomaly detection process. Association rule mining algorithm can be applied on the baseline data tree to generate set of rules which can be labelled "normal". A set of rules can also be generated from the current data tree, and the rules can be labelled "doubtful". Out of these "doubtful" rules we can identify some rules which do not comply with the "normal" rules. These incompliant rules can be further tested with some statistical significance tests to verify their degree of abnormality or anomalousness. If a rule is found statistically significant (i.e., deviated from the "normal" rules in this case), then it can be reported as an "anomalous" rule.

Bibliography

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD*, pages 207–216, 1993.

[AS94] R. Agrawal and R. Srikant. Fast algorithm for mining association rules. In *Proc. VLDB*, pages 487–499, 1994.

[ATA99] N. F. Ayan, A. U. Tansel, and E. Arkun. An efficient algorithm to update large itemsets with early pruning. In *Proc. ACM SIGKDD*, pages 287–291, 1999.

[BAG98] R. J. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining in large, dense databases. In *Proc. IEEE ICDE*, pages 188–197, 1998.

[Bay98] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proc. ACM SIGMOD*, pages 85–93, 1998.

[BM98] C. L. Blake and C. J. Merz. UCI repository of machine learning databases, 1998. Department of Information and Com-

puter Science, University of California, Irvine, CA, USA, `www.ics.uci.edu/~mlearn/MLRepository.html`.

[CDH+02] Y. Chen, G. Dong, J. Han, B. W. Wah, and J. Wang. Multi-dimensional regression analysis of time-series data streams. In *Proc. VLDB*, pages 323–334, 2002.

[CL03] J. H. Chang and W. S. Lee. Finding recent frequent itemsets adaptively over online data streams. In *Proc. ACM SIGKDD*, pages 487–492, 2003.

[CL04] J. H. Chang and W. S. Lee. A sliding window method for finding recently frequent itemsets over online data stream. *Journal of Information Science and Engineering*, 20:753–762, 2004.

[CLK97] D. W. Cheung, S. D. Lee, and B. Kao. A general incremental technique for maintaining discovered association rules. In *Proc. DASFAA*, pages 185–194, 1997.

[CWYM04] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz. Moment: maintaining closed frequent itemsets over a stream sliding window. In *Proc. IEEE ICDM*, pages 59–66, 2004.

[CZ03] W. Cheung and O. R. Zaïane. Incremental mining of frequent patterns without candidate generation or support constraint. In *Proc. IDEAS*, pages 111–116, 2003.

[EZ03] M. El-Hajj and O. R. Zaïane. Inverted matrix: efficient discovery of

frequent items in large datasets in the context of interactive mining. In *Proc. ACM SIGKDD*, pages 109–118, 2003.

[FPM91] W. J. Frawley, G. Piatetsky-Shapiro, and C. J. Matheus. Knowledge discovery in databases: an overview. In *Knowledge Discovery in Databases*, pages 1–30. AAAI/MIT Press, 1991.

[GDD+03] L. Golab, D. DeHann, E. D. Demaine, A. Lopez-Ortiz, and J. I. Munro. Identifying frequent items in sliding windows over on-line packet streams. In *Proc. ACM Internet Measurement Conference*, pages 173–178, 2003.

[GHP+02] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu. Mining frequent patterns in data streams at multiple time granularities. In *Proc. NSF Workshop on Next Generation Data Mining*, pages 191–212, 2002.

[GZ01] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *Proc. IEEE ICDM*, pages 163–170, 2001.

[GZ03a] B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: introduction to FIMI'03. In *Proc. FIMI Workshop*, 2003.

[GZ03b] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proc. FIMI Workshop*, 2003.

[GZK05] M. M. Gaber, A. Zaslavsky, and S. Khrishnaswamy. Mining data streams: A review. *SIGMOD Records*, 34(2):18–26, 2005.

[Hid99] C. Hidber. Online association rule mining. In *Proc. ACM SIGMOD*, pages 145–156, 1999.

[HPY00] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. ACM SIGMOD*, pages 1–12, 2000.

[KS04] J.-L. Koh and S.-F. Shieh. An efficient approach for maintaining association rules based on adjusting FP-tree structures. In *Proc. DASFAA*, pages 417–424, 2004.

[Leu04] C. K.-S. Leung. Interactive constrained frequent-pattern mining system. In *Proc. IDEAS*, pages 49–58, 2004.

[LKH05] C. K.-S. Leung, Q. I. Khan, and T. Hoque. CanTree: a tree structure for efficient incremental mining of frequent patterns. In *Proc. IEEE ICDM*, pages 274–281, 2005.

[LLN02] C. K.-S. Leung, L. V. S. Lakshmanan, and R. T. Ng. Exploiting succinct constraints in FP-tree. *SIGKDD Explorations*, 4(1):40–49, 2002.

[LLN03] L. V. S. Lakshmanan, C. K.-S. Leung, and R. T. Ng. Efficient dynamic mining of constrained frequent sets. *ACM TODS*, 28(4):337–389, 2003.

[LLS04] H.-F. Li, S.-Y. Lee, and M.-K. Shan. An efficient algorithm for mining frequent itemsets over the entire history of data streams. In *Proc. First Int'l. Workshop on Knowledge Discovery in Data Streams (in conjunction with ECML and PKDD)*, 2004.

[MM02] G. Manku and R. Motwani. Approximate frequent counts over data stream. In *Proc. VLDB*, pages 346–357, 2002.

[NLHP98] R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. ACM SIGMOD*, pages 13–24, 1998.

[PBTL99] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. ICDT*, pages 398–416, 1999.

[PHL01] J. Pei, J. Han, and L. V. S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Proc. IEEE ICDE*, pages 433–442, 2001.

[PHM00] J. Pei, J. Han, and R. Mao. CLOSET: an efficient algorithm for mining frequent closed itemsets. In *Proc. ACM SIGMOD*, pages 21–30, 2000.

[SVA97] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *Proc. KDD*, pages 67–73, 1997.

[WHP03] J. Wang, J. Han, and J. Pei. Closet+: searching for the beat strategies for mining frequent closed itemsets. In *Proc. ACM SIGKDD*, pages 236–245, 2003.

[Yan04] G. Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *Proc. ACM SIGKDD*, pages 344–352, 2004.

[ZH02] M. J. Zaki and C.-J. Hsiao. An efficient algorithm for closed association rule mining. In *Proc. SIAM Data Mining*, 2002.

[ZS02] Y. Zhu and D. Shasha. StatStream: statistical monitoring of thousands of data streams in real time. In *Proc. VLDB*, pages 358–369, 2002.