Frequent Pattern Mining from Dense Graph Streams

Juan J. Cameron
University of Manitoba, Winnipeg, MB, Canada
umcame33@cs.umanitoba.ca

Alfredo Cuzzocrea
ICAR-CNR & Uni. Calabria, Rende (CS), Italy
cuzzocrea@si.deis.unical.it

Fan Jiang
University of Manitoba, Winnipeg, MB, Canada
umjian29@cs.umanitoba.ca

Carson K. Leung
University of Manitoba, Winnipeg, MB, Canada
kleung@cs.umanitoba.ca

ABSTRACT

As technology advances, streams of data can be produced in many applications such as social networks, sensor networks, bioinformatics, and chemical informatics. These kinds of streaming data share a property in common—namely, they can be modeled in terms of graph-structured data. Here, the data streams generated by graph data sources in these applications are *graph streams*. To extract implicit, previously unknown, and potentially useful *frequent patterns* from these streams, efficient data mining algorithms are in demand. Many existing algorithms capture important streaming data and assume that the captured data can fit into main memory. However, problems arise when such an assumption does not hold (e.g., when the available memory is limited). In this paper, we propose a data structure called *DSMatrix* for capturing important data from the streams—especially, dense graph streams—onto the disk when the memory space is limited. In addition, we also propose two stream mining algorithms that use DSMatrix to mine frequent patterns. The tree-based *horizontal mining algorithm* applies an effective *frequency counting approach* to avoid recursive construction of sub-trees as in many tree-based mining. The *vertical mining algorithm* makes good use of the information captured in the DSMatrix for mining.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures—*graphs and networks*; H.2.8 [**Database Management**]: Database Applications—*data mining*

General Terms

Algorithms; Design; Experimentation; Management; Performance; Theory

Keywords

Data mining, frequent pattern discovery, graph patterns, graph-structured data, social networks, extending database technology, database theory

1. INTRODUCTION & RELATED WORK

Since the introduction of the research problem of frequent pattern mining from traditional static databases [3], numerous studies [9, 18, 22] have been proposed. Examples include the Apriori algorithm [3]. To improve efficiency, Han et al. [16] proposed the FP-growth algorithm, which uses an extended prefix-tree structure called *Frequent Pattern tree* (*FP-tree*) to capture the content of the transaction database. Unlike Apriori that scans the database k times (where k is the maximum cardinality of the mined frequent patterns), FP-growth scans the database twice. Although there are some works [5, 15] that use disk-based structure for mining, they mostly mine frequent patterns from *static* databases. As a preview, we mine frequent patterns from *dynamic* data streams. When dealing with these streaming data, we no longer have the luxury of scanning the data multiple times, for instance in support of complex knowledge discovery processes from data streams, like *OLAP analysis over data streams* (e.g., [10])).

Over the past decade, the automation of measurements and data collection has produced high volumes of valuable data at high velocity in many application areas. The increasing development and use of a large number of sensors has added to this situation. These advances in technology have led to streams of data such as sensor networks, social networks, road networks [19, 28]. These kinds of data share in common the property of being modeled in terms of *graph-structured data* [23] so that the streams they generate are, properly, *graph streams* (i.e., streams of graphs). In order to be able to make sense of streaming data, stream mining algorithms are needed [17, 24, 26]. When comparing with the mining from traditional *static* databases, mining from *dynamic* data streams is more challenging due to the following properties of data streams:

Property 1: Data streams are continuous and unbounded. To find frequent patterns from streams, we no longer have the luxury of performing multiple data scans. Once the streams flow through, we lose them. Hence, we need some data structures to capture the important contents of the streams (e.g., recent data—because users are usually more interested in recent data than older ones (e.g., [12, 11])).

Property 2: Data in the streams are not necessarily uniformly distributed; their distributions are usually changing with time. A currently infrequent pattern may become frequent in the future, and vice versa. So, we have to be careful not to prune infrequent patterns too early; otherwise, we may not be able to get complete information such as frequencies of certain patterns (as it is impossible to retract

those pruned patterns).

Several approximate and exact algorithms have been proposed to mine frequent patterns from data streams. *Approximate* algorithms (e.g., FP-streaming [14], TUF-streaming [20]) focus mostly on efficiency. However, due to approximate procedures, these algorithms may find some infrequent patterns or miss frequency information of some frequent patterns (i.e., some false positives or negatives). An *exact* algorithm mines only truly frequent patterns (i.e., no false positives and no false negatives) by (i) constructing a *Data Stream Tree* (*DSTree*) [21] to capture contents of the streaming data and then (ii) recursively building FP-trees for projected databases based on the information extracted from the DSTree.

While the above two properties play an important role in the mining of data streams in general, they play a more challenging role in the mining of a special class of data streams—namely, *graph streams*. Nowadays, various graph data sources can easily generate high volumes of streams of graphs (e.g., direct acyclic graphs representing human interactions in meetings [13], social networks representing connections or friendships among social individuals [7, 25]). However, when comparing with data streams in general, graph streams in particular are usually more difficult to handle [4]. Problems and state-of-the-art solutions are highlighted in recent studies. For instance, Aggarwal et al. [2] studied the research problem of mining *dense patterns* in graph streams, and they proposed probabilistic algorithms for determining such structural patterns effectively and efficiently. Bifet et al. [4] mined *frequent closed graphs* on evolving data streams. Their three innovative algorithms work on coresets of closed subgraphs, compressed representations of graph sets, and maintain such sets in a batch-incremental manner. Moreover, Aggarwal [1] explored a relevant problem of *classification* of graph streams. Along this direction, Chi et al. [8] proposed a fast graph stream classification algorithm that uses discriminative clique hashing (DICH), which can be applicable for OLAP analysis over evolving complex networks. Furthermore, Valari et al. [27] discovered top-k dense subgraphs in dynamic graph collections by means of both exact and approximate algorithms. As a preview, while these recent studies focus on graph mining, the mining algorithms we propose in the current paper work on both graph-structured data and other non-graph data.

Note that, although memory is not too expensive nowadays, the volume of data generated in data streams (including graph streams) also keeps growing at a rapid rate. Hence, algorithms for mining frequent patterns with limited memory are still in demand, so as to deal with the probing case of streams generated by graph data sources. For instance, Cameron et al. [6] studied this topic and proposed an algorithm that works well for *sparse* data streams in limited memory space. In contrast, the mining algorithms we propose in the current paper are designed to mine *dense* data streams in limited memory space. These algorithms can be viewed as complements to the sparse stream mining algorithm.

Here, *key contributions* of our current paper include a simple yet powerful on-disk data structure called *DSMatrix* for capturing and maintaining relevant data found in the streams, including dense graph streams. The DSMatrix is designed for *stream mining* of *frequent patterns* with window-sliding models. The corresponding tree-based *horizontal mining algorithm* builds a tree for data in the current window captured in the DSMatrix. Moreover, our *frequency counting technique* effectively avoids the recursive building of FP-trees for projected databases, and thus saving space. Furthermore, our *vertical mining algorithm* takes advantage of the data representation of the DSMatrix to mine frequent patterns efficiently. As the proposed DSMatrix can generally be applicable to different kinds of streaming data, it can be used in graph streams where memory requirements are very demanding.

This paper is organized as follows. Background is provided in Section 2. Section 3 presents our DSMatrix structure for capturing important information from dense streams and describes how our DSMatrix can efficiently mine frequent patterns from graph streams. Then, we discuss how we make use of the DSMatrix for horizontal (Section 4) and vertical (Section 5) mining of frequent patterns extracted from graph streams. Section 6 focuses on an analytical evaluation of the properties of the DSMatrix structure in comparison with other similar (stream) frequent pattern structures. Section 7 shows experimental results. Finally, conclusions are given in Section 8.

2. BACKGROUND

To mine frequent patterns, an exact stream mining algorithm [21] first constructs a **Data Stream Tree (DSTree)**, which is then used as a *global tree* for recursively generating smaller FP-trees (as *local trees*) for projected databases. Due to the dynamic nature of data streams (as seen in Properties 1 and 2), frequencies of items are continuously affected by the insertion of new batches (and the removal of old batches) of transactions. Arranging items in frequency-dependent order may lead to swapping—which, in turn, can cause merging and splitting—of tree nodes when frequencies change. Hence, in the DSTree, transaction items are arranged according to some canonical order (e.g., alphabetical order), which can be specified by the user prior to the tree construction or mining process. Consequently, the DSTree can be constructed using only a *single* scan of the graph streams. Note that the DSTree is designed for processing streams within a sliding window. So, for a window size of w batches, each tree node keeps (i) an item and (ii) a *list* of w frequency values (instead of a single frequency count in each node of the FP-tree for frequent pattern mining from *static* databases). Each entry in this list captures the frequency of the item in each batch of dynamic streams in the current window. By so doing, when the window slides (i.e., when new batches are inserted and old batches are deleted), frequency information can be updated easily. Consequently, the resulting DSTree preserves the usual tree properties that (i) the total frequency (i.e., sum of w frequency values) of any node is at least as high as the sum of total frequencies of its children and (ii) the ordering of items is unaffected by the continuous changes in item frequencies.

With the aforementioned DSTree, the mining is "delayed" until it is needed. Hence, once the DSTree is constructed, it is always kept up-to-date when the window slides. The exact mining algorithm mines frequent patterns from the updated DSTree by performing the following steps. It first traverses/extracts relevant tree paths upwards and sums the frequency values of each list in a node representing an item (or itemset)—to obtain its frequency in the current sliding window—for forming an appropriate projected database.

Afterwards, the algorithm constructs a FP-tree for the projected database of each of these frequent patterns of only 1 item (i.e., 1-itemset) such as an $\{x\}$-projected database (in a similar fashion as in the FP-growth algorithm for mining static data [16]). Thereafter, the algorithm recursively forms subsequent FP-trees for projected databases of frequent k-itemsets where $k \geq 2$ (e.g., $\{x, y\}$-projected database, $\{x, z\}$-projected database, etc.) by traversing paths in these FP-trees. As a result, the algorithm finds all frequent patterns. Note that, as items are consistently arranged according to some canonical order, the algorithm guarantees the inclusion of all *frequent* items using just upward traversals. Moreover, there is also no worry about possible omission or double-counting of items during the mining process. Furthermore, as the DSTree is always kept up-to-date, all frequent patterns—which are embedded in batches within the current sliding window—can be found effectively.

In the remainder of this paper, we call the aforementioned exact algorithm that uses the DSTree as the global tree, from which FP-trees for subsequent projected databases can be constructed recursively, the **⟨global DSTree, local FP-trees⟩ mining option**. It works well when memory space is not an issue. The success of this algorithm mainly relies on the assumption—usually made for many tree-based algorithms [16]—that all tree (i.e., the global tree together with subsequent FP-trees) fit into the memory. For example, when mining frequent patterns from the $\{x, y, z\}$-projected database, the global tree and two subsequent FP-trees (for the $\{x\}$-, $\{x, y\}$- and $\{x, y, z\}$-projected databases) are all assumed to be fit into memory.

However, there are situations where the memory is so limited that not all the trees can fit into memory, like the case of streaming generated from graph data sources. To handle these situations, the **Data Stream Table (DSTable)** [6] was proposed. The DSTable is a two-dimensional table that captures on the disk the contents of transactions in all batches within the current sliding window. Each row of the DSTable represents a domain item. Like the DSTree, items in the DSTable are arranged according to some canonical order (e.g., alphabetical order), which can be specified by the user prior to the construction of the DSTable. As such, table construction requires only a single scan of the graph stream. Each entry in the resulting DSTable is a "pointer" that points to the location of the table entry (i.e., which row and which column) for the "next" item in the same transaction. When dealing with graph streaming data, the DSTable also keeps w boundary values (to represent the boundary between w batches in the current sliding window) for each item. By doing so, when the window slides, transactions in the old batch can be removed and transactions in the new batch can be added easily.

Similar to mining with the DSTree, the mining with this DSTable is also "delayed" until it is needed. Once the DSTable is constructed and kept up-to-date when the window slides, the mining algorithm first traverses/extracts relevant transactions from the DSTable. Then, the algorithm (i) constructs a FP-tree for the projected database of each of these 1-itemsets and (ii) recursively forms subsequent FP-trees for projected databases of frequent k-itemsets (where $k \geq 2$) by traversing the paths of these FP-trees. As a result, the algorithm finds all frequent patterns. In the remainder of this paper, we call this the **⟨global DSTable, local FP-trees⟩ mining option**.

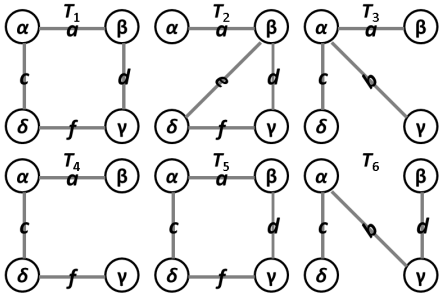


Figure 1: A graph stream (Example 1).

Example 1. For illustrative purpose, let us consider a sliding window of size $w = 2$ batches (i.e., only two batches are kept) and the following segments in a stream of graphs, where each graph $G = (V, E)$ consists of $|V| = 4$ vertices (Vertices α, β, γ and δ) and $|E| \leq 6$ edges:

- At time T_1, $E = \{(\alpha, \beta), (\alpha, \delta), (\beta, \gamma), (\gamma, \delta)\}$;
- At time T_2, $E = \{(\alpha, \beta), (\beta, \gamma), (\beta, \delta), (\gamma, \delta)\}$;
- At time T_3, $E = \{(\alpha, \beta), (\alpha, \gamma), (\alpha, \delta)\}$;
- At time T_4, $E = \{(\alpha, \beta), (\alpha, \delta), (\gamma, \delta)\}$;
- At time T_5, $E = \{(\alpha, \beta), (\alpha, \delta), (\beta, \gamma), (\gamma, \delta)\}$; and
- At time T_6, $E = \{(\alpha, \gamma), (\alpha, \delta), (\beta, \gamma)\}$.

See Figure 1. These graphs may represent some interactions in meetings or friendships among social individuals. For simplicity, we represent these edges by six symbols a, b, c, d, e and f. Consequently, we get (i) transactions $t_1 = \{a, c, d, f\}$, $t_2 = \{a, d, e, f\}$ and $t_3 = \{a, b, c\}$ in the first batch B_1; as well as (ii) transactions $t_4 = \{a, c, f\}, t_5 = \{a, c, d, f\}$ and $t_6 = \{b, c, d\}$ in the second batch B_2. Let the user-specified *minsup* threshold be 2. Then, the DSTable stores the following information:

DSTable:

Row	Boundaries	Contents
Edge a:	Cols 3 & 5	$(c, 1), (d, 2), (b, 1); (c, 3), (c, 4)$
Edge b:	Cols 1 & 2	$(c, 2); (c, 5)$
Edge c:	Cols 2 & 5	$(d, 1)$, end; $(f, 3), (d, 3), (d, 4)$
Edge d:	Cols 2 & 4	$(f, 1), (e, 1); (f, 4)$, end
Edge e:	Cols 1 & 1	$(f, 2)$;
Edge f:	Cols 2 & 4	end, end; end, end

In the DSTree, the first entry in Row a with value $(c, 1)$—which captures a transaction starting edge/item a and having c as the second edge—points to the 1st column of Row c. Its value $(d, 1)$ points to the 1st column of Row d, which captures the value $(f, 1)$. This indicates the third and fourth edges are d and f, respectively. Then, the 1st column of Row e with value "end" indicates the end of the transaction containing $\{a, c, d, f\}$. Based on contents of the entire DSTable, the mining algorithm first finds frequent singletons $\{a\}, \{b\}, \{c\}, \{d\}$ and $\{f\}$. The algorithm then constructs an FP-tree for the $\{a\}$-projected database (i.e., transactions containing a) to get frequent 2-itemsets $\{a, c\}$, $\{a, d\}$ and $\{a, f\}$. From this FP-tree, the algorithm recursively constructs subsequent FP-trees (e.g., for $\{a, c\}$-, $\{a, c, d\}$- and $\{a, d\}$-projected databases). Afterwards, the algorithm constructs an FP-tree for the $\{b\}$-projected database (i.e., transactions containing b), from which subsequent FP-trees are constructed. Similar steps apply to $\{c\}$- and $\{d\}$-projected databases.

The boundary information "Cols 3 & 5" for Row a indicates that (i) the boundary between batches B_1 and B_2 is at the end of column 3 and (ii) batch B_2 ends at column 5. Hence, when a new batch comes in, the old batch is removed. In this case, the first three columns of Row a (due to "Cols 3 & 5" in Row a), the first 1 column of Row b (due to "Cols 1 & 2" in Row b), the first 2 columns of Rows c and d, the first 1 column of Row e, as well as the first 2 columns of Row f can be removed. □

Observed from the above example, mining with the ⟨global DSTree, local FP-trees⟩ option may suffer from several problems when handling data streams (especially, dense graph streams) with limited memory. Some of these problems are listed as follows:

P1. To facilitate easy insertion and deletion of contents in the DSTable when the window (of size w batches) slides, the DSTable keeps w boundary values for each row (representing each of the m domain items). Hence, the DSTable needs to keep a total of $m \times w$ boundary values.

P2. Each table entry is a "pointer" that indicates the location in terms of row name (e.g., Row c) and column number (e.g., Column 1) of the table entry for the "next" item in the same transaction. When the data stream is sparse, only a few "pointers" need to be stored. However, when the graph stream is dense, many "pointers" need to be stored. Given a total of $|T|$ transactions in all batches within the current sliding window, there are potentially $m \times |T|$ "pointers" (where m is the number of domain items).

P3. During the mining process, multiple FP-trees need to be constructed and kept in memory (e.g., FP-trees for all $\{a\}$-, $\{a, c\}$- and $\{a, c, d\}$-projected databases are required to be kept in memory).

3. THE DSMatrix DATA STRUCTURE

In attempt to solve the above problems while mining frequent patterns from data streams (especially, dense graph streams) with limited memory, we propose a 2-dimensional structure called **Data Stream Matrix (DSMatrix)**. This matrix structure captures the contents of transactions in all batches within the current sliding window by storing them on the disk. Note that the DSMatrix is a binary matrix, which represents the presence of an item x in transaction t_i by a "1" in the matrix entry (t_i, x) and the absence of an item y from transaction t_j by a "0" in the matrix entry (t_j, y). With this binary representation of items in each transaction, each column in the DSMatrix captures a transaction. Each column in the DSMatrix can be considered as a bit vector.

Similar to the DSTable, our DSMatrix also keeps track of any boundary between two batches so that, when the window slides, transactions in the older batches can be easily removed and transactions in the newer batches can be easily added. Note that, in the DSTable, boundaries may vary from one row (representing an item) to another row (representing another item) due to the potentially different number of items present. Contrarily, in our DSMatrix, boundaries are the same from one row to another because we put a binary value (0 or 1) for each transaction.

Example 2. Let us revisit Example 1. The information captured by that DSTable can be effectively captured by our DSMatrix, but in less space:

Our DSMatrix:

Boundaries: Cols 3 & 6	
Row	Contents
Row a:	1 1 1; 1 1 0
Row b:	0 0 1; 0 0 1
Row c:	1 0 1; 1 1 1
Row d:	1 1 0; 0 1 1
Row e:	0 1 0; 0 0 0
Row f:	1 1 0; 1 1 0

When compared with the DSTable, we do not need to store the same boundary information multiple times (for the m domain items). We only need to store it once. □

Hence, with our DSMatrix, we solve previous Problems P1 and P2 of the DSTree, as follows:

S1. Recall that the DSTable needs to keep a total $m \times w$ boundary values. In contrast, our DSMatrix only keeps w boundary values (where $w \ll m \times w$) for the entire matrix, regardless how many domain items (m) are here.

S2. Recall that each table entry in the DSTable captures both the row name and column number to represent a "pointer" to the next item in a transaction. The computation of column number requires the DSTable to constantly keep track of the index of the last item in each row representing a domain item. Moreover, each "pointer" requires two integer (row name/number and column number). For P items in $|T|$ transactions, the DSTable requires $2 \times 32 \times P$ bits (for 32-bit integer representation). For dense data streams, the DSTable requires potentially $64m \times |T|$ bits. In contrast, our DSMatrix uses a bit vector to indicate the presence or absence of items in a transaction. The computation does not require us to keep track of the index of the last item in every row and thus incurring a lower computation cost. Moreover, given a total of $|T|$ transactions in all batches within the current sliding window, there are $|T|$ columns in our DSMatrix. Each column requires only m bits. In other words, our DSMatrix takes $m \times |T|$ bits (cf. potentially $64m \times |T|$ bits for dense data streams required by the DSTree).

4. TREE-BASED HORIZONTAL FREQUENT PATTERN MINING

Whenever a new batch of streaming data (e.g., streaming graph data) comes in, the window slides. Transactions in the oldest batch in the sliding window are then removed from our DSMatrix so that transactions in this new batch can be added. Following the aforementioned mining routines, the mining is "delayed" until it is needed. Once the DSMatrix is constructed, it is kept up-to-date on the disk.

To find frequent patterns, we propose *a tree-based horizontal mining algorithm*. When the user needs to find frequent patterns, we extract relevant transactions from the DSMatrix to form an FP-tree for each projected database of every frequent singleton. Key ideas of the algorithm are illustrated in Example 3.

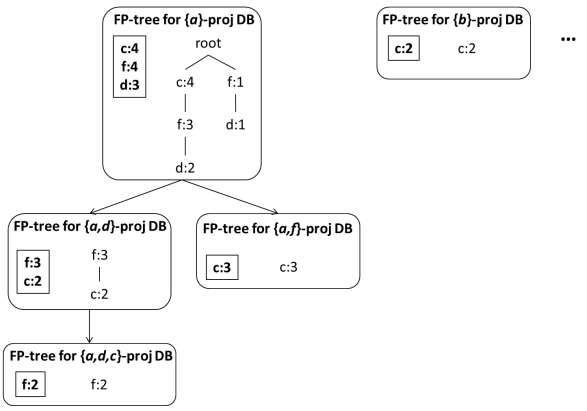


Figure 2: Multiple FP-trees built for $\{a\}$-, $\{b\}$-, ... projected DBs from the DSMatrix (Example 3).

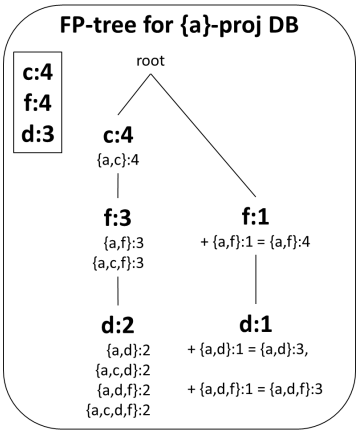


Figure 3: FP-tree for $\{a\}$-proj. DB (Example 4).

Example 3. Continue with Example 2. To form the $\{a\}$-projected database, we examine Row a. For every column with a value "1", we extract its column downwards (e.g., from edges/items b to e if they exist). Specifically, when examining Row a, we notice that columns 1, 2, 3, 4 and 6 contain values "1" (which means that a appears in those five transactions in the two batches of streaming graph data in the current sliding window). Then, from Column 1, we extract $\{c, d, f\}$. Similarly, we extract $\{d, e, f\}$ and $\{b, c\}$ from Columns 2 and 3. We also extract $\{c, f\}$ and $\{c, d, f\}$ from columns 4 and 5. All these form the $\{a\}$-projected database, from which an FP-tree can be built. From this FP-tree for the $\{a\}$-projected database, we find that 2-itemsets $\{a, c\}, \{a, d\}$ and $\{a, f\}$ are frequent. Hence, we then form $\{a, d\}$- and $\{a, f\}$-projected databases, from which FP-trees can be built. (Note that we do not need to form the $\{a, c\}$-projected database as it is empty after forming both $\{a, d\}$- and $\{a, e\}$-projected databases.) When applying this step recursively in a depth-first manner, we obtain frequent 3-itemsets $\{a, c, d\}, \{a, c, f\}$ and $\{a, d, f\}$, which leads to FP-trees for the $\{a, d, c\}$-projected database. (Again, we do not need to form the $\{a, f, c\}$- or $\{a, d, f\}$-projected databases as they are both empty.) At this moment, we keep FP-trees for the $\{a\}$-, $\{a, d\}$- and $\{a, d, c\}$-projected databases. Afterwards, we also find that 4-itemset $\{a, c, d, f\}$ is frequent. In the context of graph streams, this is a frequent collection of 4 edges—namely, Edges a, c, d and f. See Figure 2.

We backtrack and examine the next frequent singleton $\{b\}$. When examining Row b, we notice that Columns 3 and 6 contain values "1" (which means that b appears in those two transactions in the current sliding window). For these two columns, we extract downward to get $\{c\}$ and $\{c, d\}$ that appear together with b (i.e., to form the $\{b\}$-projected database. As shown in Figure 2, the corresponding FP-tree contains $\{c\}$:2 meaning that c occurs twice with b (i.e., 2-itemset $\{b, c\}$ is frequent with frequency 2). Similar steps are applied to other frequent singletons $\{c\}, \{d\}$ and $\{f\}$ in order to discover all frequent patterns. □

Note that, during the mining process, we require multiple FP-trees to be kept in the memory during the mining process (i.e., Problem P3 of the ⟨global DSTree, local FP-trees⟩ mining option). However, when the memory space is limited, *not* all of the multiple FP-trees can fit into the memory.

To solve this problem, which identifies the Problem P3 above, we propose the following effective frequency counting technique:

S3. Once an FP-tree for the projected database of a frequent singleton is built, we traverse every tree node in a depth-first manner (e.g., pre-order, in-order, or post-order traversal). For every first visit of a tree node, we generate the itemset represented by the node and its subsets. We also compute their frequencies.

Example 4. Based on the DSMatrix in Example 2, we first construct an FP-tree for the $\{a\}$-projected database. Then, we traverse every node in such an FP-tree. When traversing the leftmost branch $\langle c{:}4,\ b{:}1\rangle$, we visit nodes "c:4" (which represents itemset $\{a, c\}$ with frequency 4) and "b:1" (which gives $\{a, b\}$ with frequency 1 and $\{a, b, c\}$ with frequency 1). Next, we traverse the middle branch $\langle c{:}4,\ f{:}3,\ d{:}2\rangle$. By visiting nodes "f:3" and "d:2", we get $\{a, f\}$ and $\{a, c, f\}$ both with frequencies 3, as well as $\{a, d\}, \{a, c, d\}, \{a, d, f\}$ and $\{a, c, d, f\}$ all with frequencies 2. Finally, we visit nodes "f:1" and "d:1" in the rightmost branch $\langle f{:}1,\ d{:}1\rangle$, from which we get the frequency 1 for both $\{a, d\}, \{a, d, f\}$ and $\{a, f\}$. This frequency value is added to the existing frequency count of 2 (from the middle branch) to give the frequency of $\{a, d\}$ and $\{a, d, f\}$ equal to 3. Hence, with the *minsup* threshold set to 2, we obtain frequent patterns $\{a, c\}$:4, $\{a, c, d\}$:2, $\{a, c, d, f\}$:2, $\{a, c, f\}$:3, $\{a, d\}$:3, $\{a, d, f\}$:3 and $\{a, f\}$:4. Note that, during this mining process for the $\{a\}$-projected database, we count frequencies of itemsets without recursive construction of FP-trees. See Figure 3.

Afterwards, we build an FP-tree for the $\{b\}$-projected database and count frequencies of all frequent patterns containing item b. Similar steps are applied to the FP-trees for the $\{c\}$- and $\{d\}$-projected databases. □

Note that, at any moment during the mining process, only one FP-tree needs to be constructed and kept in the memory for this ⟨global DSMatrix, local FP-tree⟩ mining process (cf. multiple FP-trees required for the ⟨global DSTree, multiple local FP-trees⟩ mining option). This solves Problem P3.

5. VERTICAL FREQUENT PATTERN MINING

In Section 4, we described a tree-based horizontal frequent pattern mining algorithm that makes good use of the DSMatrix to form FP-trees for frequent singletons. From each of these FP-trees, the algorithm applies an effective frequency counting technique to find all frequent patterns with their frequency information in such a way that the algorithm avoids recursive construction of FP-trees for frequent k-itemsets (where $k > 2$).

In this section, we present *a vertical frequent pattern mining algorithm*. Given that the contents stored in our DSMatrix can be considered as a collection of bit vectors. It becomes logical to consider vertical mining. To mine frequent singletons, we examine each row (representing a domain item). The *row sum* (i.e., total number of 1s) gives the frequency of the item represented by that row. Once the frequent singletons are found, we *intersect the bit vectors* for two items. If the row sum of the resulting intersection $\geq$ the user-specified *minsup* threshold, then we find a frequent 2-itemset. We repeat these steps by intersecting two bit vectors of frequent patterns to find frequent patterns of higher cardinality.

Example 5. Based on the DSMatrix in Example 1, we first compute the row sum for each row (i.e., for each domain item). As a result, we find that edges/items a, b, c, d and f are all frequent with frequencies 5, 2, 5, 4 and 4, respectively. Afterwards, we intersect the bit vector of a (i.e., Row a) with any one of the remaining four bit vectors (i.e., any one of the four rows) to find frequent 2-itemsets $\{a, c\}, \{a, d\}$ and $\{a, f\}$ with frequencies 4, 3 and 4, respectively, because (i) the intersection of $\vec{a}$ and $\vec{c}$ gives a bit vector 101110, (ii) the intersection of $\vec{a}$ and $\vec{d}$ gives a bit vector 110010, and (iii) the intersection of $\vec{a}$ and $\vec{f}$ gives a bit vector 110110. Next, we intersect (i) $\overrightarrow{ac}$ with $\overrightarrow{ad}$, (ii) $\overrightarrow{ac}$ with $\overrightarrow{af}$ and (iii) $\overrightarrow{ad}$ with $\overrightarrow{af}$ to find frequent 3-itemsets $\{a, c, d\}, \{a, c, f\}$ and $\{a, d, f\}$. We also intersect $\overrightarrow{acd}$ with $\overrightarrow{acf}$ to find frequent 4-itemset $\{a, c, d, f\}$. These are all frequent patterns containing item a.

Afterwards, we repeat similar steps with the bit vectors for the other singletons. For instance, we intersect $\vec{b}$ with $\vec{c}, \vec{d}$ and $\vec{f}$. We find out that, among them, only $\{b, c\}$ is frequent with frequency 2. We also intersect $\vec{c}$ with $\vec{d}$ and $\vec{f}$ to find frequent 3-itemsets $\{c, d\}$ and $\{c, f\}$, each with frequencies of 3. We also find frequent 4-itemsets $\{c, d, f\}$ by intersecting $\overrightarrow{cd}$ and $\overrightarrow{cf}$. Finally, we intersect $\vec{d}$ and $\vec{f}$ to find frequent 2-itemset $\{d, f\}$ with frequency 3. □

6. ANALYTICAL EVALUATION

Recall from Section 1 that FP-streaming [14] is an approximate algorithm, which uses an "immediate" mode for mining. During the mining process for each batch of the streaming data, FP-streaming builds a global FP-tree and $O(f \times d)$ subsequent local FP-trees, where f is the number of frequent items in the domain and d is the height/depth of the global FP-tree. At any time during the mining process for each batch, the global FP-tree and $O(d)$ subsequent local FP-trees are stored in the memory. Hence, storage cost includes the memory space for the global FP-tree plus all $O(d)$ subsequent local FP-trees for each batch. In terms of efficiency, when the number of batches in the data stream increases, the CPU cost for the mining process increases. For example, let us consider a sliding window of w=5 batches. When handling a stream of S=100 batches, FP-streaming builds the global FP-tree plus all subsequent local FP-trees, and mines frequent patterns from these FP-trees for each of the 100 batches. In other words, FP-streaming builds 100 sets of the global and subsequent local FP-trees. Moreover, the computation effort for the first 95 batches is wasted (when users request frequent patterns at the end of the 100th batch).

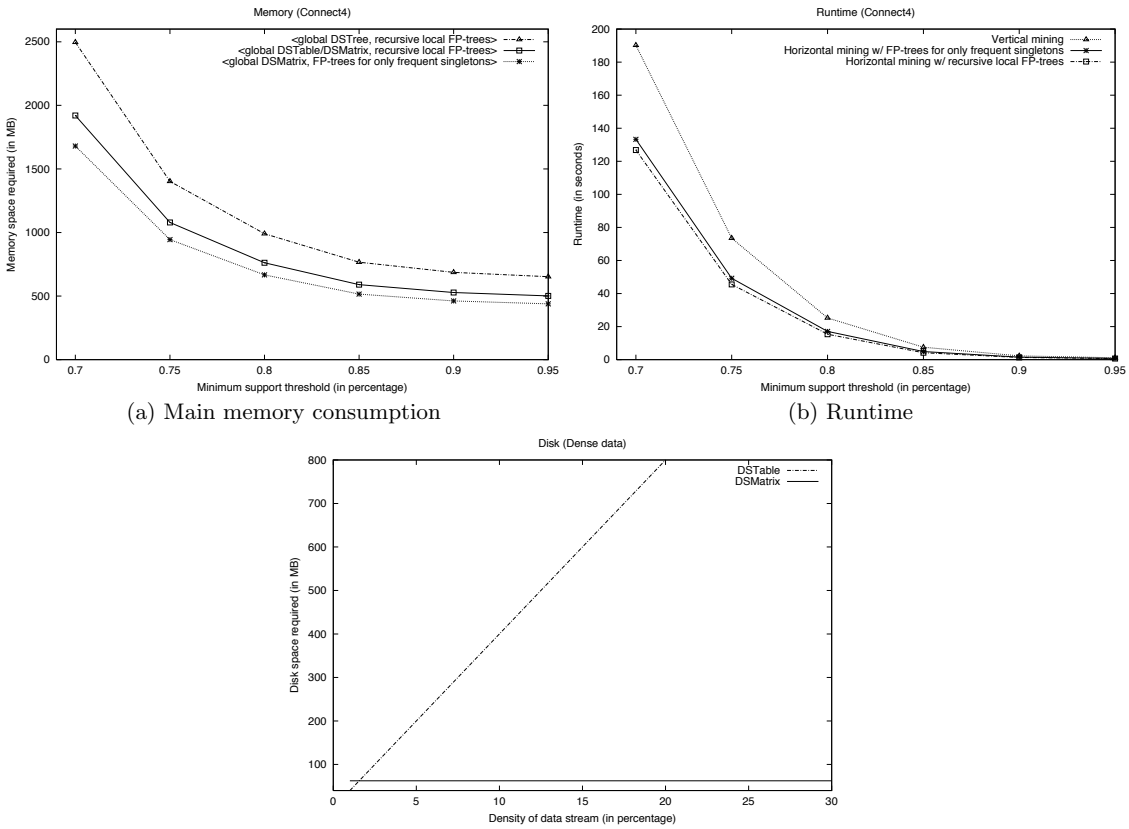
Recall from Section 2 that mining with the DSTree [21] or DSTable [6] uses a "delayed" mode for mining. So, the actual mining of frequent patterns is delayed until they are needed to be returned to the user. Hence, for S=100 batches, the mining algorithm needs to build a global DSTree or DSTable and updates it $S-w = 100-5 = 95$ times. Once an updated DSTree or DSTable has captured the 96th to the 100th batches, multiple FP-trees are constructed to find frequent patterns. Note that only one set of the updated global DSTree (or DSTable) and multiple FP-trees are required (cf. building 100 sets of a global tree and $O(f \times d)$ FP-trees by FP-streaming, one set for each of the 100 batches). Moreover, at any time during the mining process of the ⟨global DSTree, local FP-trees⟩ option, only the global DSTree and multiple FP-trees are needed to be present (cf. one global FP-tree and $O(d)$ subsequent FP-trees are needed to be present in FP-streaming). When using the ⟨global DSTable, local FP-trees⟩ option, the global DSTable is kept on disk. Thus, only multiple FP-trees are needed to be kept in the memory.

In contrast, the DSMatrix resides on disk. Being specialized to dense graph streams, it can serve as an alternative to the global FP-tree when memory is limited. Moreover, the size of the DSMatrix is independent of the user-specified minimum support threshold (*minsup*). Hence, it is useful for interactive mining, especially when users keep adjusting *minsup*, which is relevant for mining graph streams. It should be noted that the DSMatrix captures the transactions in the current sliding window. During the mining process, the algorithm skips infrequent items (i.e., items having support lower than *minsup*) and only includes frequent items when building subsequent FP-trees for projected databases. Furthermore, with our frequency counting technique, we do not even need to build too many FP-trees. Instead, we only need to build FP-trees for frequent singleton (i.e., for $\{x\}$-projected databases, where x is a frequent item). When users adjust *minsup* during the interactive mining process, we do not need to rebuild the DSMatrix. In contrast, when *minsup* changes, FP-streaming needs to rebuild the global FP-tree.

In terms of disk space, the DSTable [6] requires $64 \times P$ bits (for 32-bit integer representation), where P is the total number of items in $|T|$ transactions in the w batches of the data streams. In the worst case, the DSTable requires potentially $64m \times |T|$ bits for dense data streams. In contrast, our DSMatrix requires only $m \times |T|$ bits, which is desirable for applications that require dense graph stream mining.

7. EXPERIMENTAL EVALUATION

To acquire dense graph stream datasets, we first generated random graph models via a Java-based generator by varying model parameters (e.g., topology, average fan-out of nodes, edge centrality, etc.). We then generated graph streams as nodes and node-edge relationships derived from the above



(a) Main memory consumption (b) Runtime

(c) Disk consumption

Figure 4: Experimental results.

graph models, and obtained node values from popular data stream sets available in literature (stored in the projected database). In addition, we also used many different databases including IBM synthetic data, real-life databases (e.g., connect4) from the UC Irvine Machine Learning Depository as well as those from the Frequent Itemset Mining Implementation (FIMI) Dataset Repository. For example, connect4 is a dense data set containing 67,557 records with an average transaction length of 43 items, and a domain of 130 items. Each record represents a graph of legal 8-ply positions in the game of connect 4.

All experiments were run in a time-sharing environment in a 1 GHz machine. We set each batch to be 6K records and the window size w=5 batches. The reported figures are based on the average of multiple runs. Runtime includes CPU and I/Os; it includes the time for both tree construction and frequent pattern mining steps. In the experiments, we mainly evaluated the accuracy and efficiency of our DSMatrix by comparing with related works such as (i) DSTree [21] and (ii) DSTable [6].

In the first experiment, we measured the accuracy of the following four mining options: (i) ⟨global DSTree, recursive local FP-trees⟩; (ii) ⟨global DSTable, recursive local FP-trees⟩; (iii) ⟨global DSMatrix, recursive local FP-trees⟩; (iv) ⟨global DSMatrix, local FP-trees for only frequent singletons⟩ options. Experimental results show that mining with any of these four options give the same mining results.

While these four options gave the same results, their performance varied. In the second and third experiments, we measured the space and time efficiency of our proposed DSMatrix. Results show that the ⟨DSTree, recursive FP-trees⟩ option required the largest main memory space as it stores one global DSTree and multiple local FP-trees in main memory. The ⟨DSTable, recursive FP-trees⟩ and ⟨DSMatrix, recursive FP-trees⟩ options required less memory as their DSTable and DSMatrix were kept on disk. Among the four mining options, the ⟨DSMatrix, FP-trees for only frequent singletons⟩ option required the *smallest* main memory space because at most m FP-trees needed to be generated during the entire mining process, one for each frequent domain item. Note that not all m domain items were frequent. Moreover, at any mining moment, only one of these FP-trees needs to be presented. In other words, not all $\leq m$ FP-trees were generated at the same time. See Figure 4(a).

There are tradeoffs between memory consumption vs. runtime performance. Runtime performance of the three options also varied. For instance, vertical mining does not consume too much memory. While the bitwise operation (e.g., intersection) is quick, but vertical mining required many scans to read bit vectors from the DSMatrix. In contrast, horizontal mining with FP-trees built for only frequent singletons consumes more memory because it keeps these $\leq m$ FP-trees from $\leq m$ frequent singletons in memory. Along the same direction, horizontal mining with FP-

trees recursively built for subsequent frequent patterns consumes even more memory because it keeps all FP-trees (i.e., those for frequent singletons and their subsequent frequent k-itemsets, where $k \geq 2$) in memory. See Figure 4(a). It is important to note that reading from disk would be a *logical choice* in a limited main memory environment.

Moreover, we perform some additional experiments, by testing with the usual experiment (e.g., the effect of *minsup*). As shown in Figure 4(b), the runtime decreased when *minsup* increased. In another experiment, we tested scalability with the number of transactions. The results show that mining with our proposed DSMatrix was scalable (see Figure 4(c)). In particular, Figure 4(c) compares the disk consumption between the DSTable and our DSMatrix, and it clearly shows that our DSMatrix requires a constant amount of disk space, where the DSTable requires different amounts depending on the density of data streams. An interesting observation that, for dense data, our DSMatrix is beneficial due to its bit vector representation. As future work, we plan to conduct more extensive experiments on various datasets (including Big data) with different parameter settings (e.g., varying *minsup* and transaction lengths that represent the complexity of graphs).

8. CONCLUSIONS

As technology advances, streams of data (including graph streams) can be produced in many applications. Key contributions of this paper include (i) a simple yet powerful alternative disk-based structure—called *DSMatrix*—for efficient frequent pattern mining from streams (e.g., dense graph streams) with limited memory and (ii) two frequent pattern mining algorithms: a *tree-based horizontal mining algorithm* and a *vertical mining algorithm*. To avoid keeping too many FP-trees in memory when the space is limited, we also described an effective frequency counting technique, which requires only one FP-tree for a projected database to be kept in the limited memory. Analytical and experimental results show the benefits of our DSMatrix structure and its corresponding mining algorithms.

9. ACKNOWLEDGEMENTS

This project is partially supported by NSERC (Canada) and University of Manitoba.

10. REFERENCES

[1] C.C. Aggarwal. On classification of graph streams. In *Proc. SDM 2011*, pp. 652–663.

[2] C.C. Aggarwal, Y. Li, P.S. Yu, & R. Jin. On dense pattern mining in graph streams. *PVLDB*, **3**(1–2), pp. 975–984 (2010).

[3] R. Agrawal & R. Srikant. Fast algorithms for mining association rules. In *Proc. VLDB 1994*, pp 487–499.

[4] A. Bifet, G. Holmes, B. Pfahringer, & R. Gavaldà. Mining frequent closed graphs on evolving data streams. In *Proc. ACM KDD 2011*, pp. 591–599.

[5] G. Buehrer, S. Parthasarathy, & A. Ghoting. Out-of-core frequent pattern mining on a commodity. In *Proc. ACM KDD 2006*, pp. 86–95.

[6] J.J. Cameron, A. Cuzzocrea, & C.K. Leung. Stream mining of frequent sets with limited memory. In *Proc. ACM SAC 2013*, pp. 173–175.

[7] J.J. Cameron, C.K. Leung, & S.K. Tanbeer. Finding strong groups of friends among friends in social networks. In *Proc. SCA 2011*, pp. 824–831.

[8] L. Chi, B. Li, & X. Zhu. Fast graph stream classification using discriminative clique hashing. In *Proc. PAKDD 2013, Part I*, pp. 225–236.

[9] D.Y. Chiu, Y.H. Wu, & A. Chen. Efficient frequent sequence mining by a dynamic strategy switching algorithm. *VLDBJ*, **18**(1), pp. 303–327 (2009).

[10] A. Cuzzocrea. CAMS: OLAPing Multidimensional Data Streams Efficiently. In *Proc. DaWaK 2009*, pp. 48–62.

[11] A. Cuzzocrea & S. Chakravarthy. Event-based lossy compression for effective and efficient OLAP over data streams. *Data & Knowledge Engineering*, **69**(7), pp. 678–708 (2010).

[12] A. Cuzzocrea, F. Furfaro, G.M. Mazzeo & D. Saccà. A Grid Framework for Approximate Aggregate Query Answering on Summarized Sensor Network Readings. In *Proc. OTM Workshops 2004*, pp. 144–153.

[13] A. Fariha, C.F. Ahmed, C.K. Leung, S.M. Abdullah, & L. Cao. Mining frequent patterns from human interactions in meetings using directed acyclic graphs. In *Proc. PAKDD 2013, Part I*, pp. 38–49.

[14] C. Giannella, J. Han, J. Pei, X. Yan, & P.S. Yu. Mining frequent patterns in data streams at multiple time granularities. In *Data Mining: Next Generation Challenges and Future Directions*, ch. 6 (2004).

[15] G. Grahne & J. Zhu. Mining frequent itemsets from secondary memory. In *Proc. IEEE ICDM 2004*, pp. 91–98.

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

[17] R. Jin & G. Agrawal. An algorithm for in-core frequent itemset mining on streaming data. In *Proc. IEEE ICDM 2005*, pp. 210–217.

[18] C.K. Leung. Mining frequent itemsets from probabilistic datasets. In *Proc. EDB 2013*, pp. 137–148.

[19] C.K. Leung & C.L. Carmichael. Exploring social networks: a frequent pattern visualization approach. In *Proc. IEEE SocialCom 2010*, pp. 419–424.

[20] C.K. Leung, A. Cuzzocrea, & F. Jiang. Discovering frequent patterns from uncertain data streams with time-fading and landmark models. *LNCS TLDKS*, **8**, pp. 174–196 (2013).

[21] C.K. Leung & Q.I. Khan. DSTree: a tree structure for the mining of frequent sets from data streams. In *Proc. IEEE ICDM 2006*, pp. 928–932.

[22] C.K. Leung & S.K. Tanbeer. PUF-tree: a compact tree structure for frequent pattern mining of uncertain data. In *Proc. PAKDD 2013, Part I*, pp. 13–25.

[23] F. Mandreoli, R. Martoglia, G. Villani, & W. Penzo: Flexible query answering on graph-modeled data. In *Proc. EDBT 2009*, pp. 216–227.

[24] O. Papapetrou, M. Garofalakis, & A. Deligiannakis. Sketch-based querying of distributed sliding-window data streams. *PVLDB*, **5**(10), pp. 992–1003 (2012).

[25] S.K. Tanbeer, F. Jiang, C.K. Leung, R.K. MacKinnon, & I.J.M. Medina. Finding groups of friends who are significant across multiple domains in social networks. In *Proc. CASoN 2013*, pp. 21–26.

[26] S. Tirthapura & D.P. Woodruff. A general method for estimating correlated aggregates over a data stream. In *Proc. IEEE ICDE 2012*, pp. 162–173.

[27] E. Valari, M. Kontaki, & A.N. Papadopoulos. Discovery of top-k dense subgraphs in dynamic graph collections. In *Proc. SSDBM 2012*, pp. 213–230.

[28] F. Wei-Kleiner. Finding nearest neighbors in road networks: a tree decomposition method. In *Proc. EDBT/ICDT 2013 Workshops* (*GraphQ*), pp. 233–240.