Frequent Subgraph Mining from Streams of Linked Graph Structured Data

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

Nowadays, high volumes of high-value data (e.g., semantic web data) can be generated and published at a high velocity. A collection of these data can be viewed as a big, interlinked, dynamic graph structure of linked resources. Embedded in them are implicit, previously unknown, and potentially useful knowledge. Hence, efficient knowledge discovery algorithms for mining frequent subgraphs from these dynamic, streaming graph structured data are in demand. Some existing algorithms require very large memory space to discover frequent subgraphs; some others discover collections of frequently co-occurring edges (which may be disjoint). In contrast, we propose—in this paper—algorithms that use *limited memory* space for discovering collections of frequently co-occurring *connected* edges. Evaluation results show the effectiveness of our algorithms in frequent subgraph mining from streams of linked graph structured data.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures—*graphs and networks*; E.2 [**Data**]: Data Storage Representations—*linked representations*; H.2.8 [**Database Management**]: Database Applications—*data mining*

General Terms

Algorithms; Design; Experimentation; Management; Performance; Theory

Keywords

Data mining, frequent patterns, graph structured data, linked data, extending database technology, database theory

1. INTRODUCTION

Nowadays, high volumes of valuable semantic web, life science, social network, or bibliographical network data can be generated from diverse real-life applications [3, 14, 23]. For example, semantic web data—such as blogs, forums, wikis, and users' reviewers—can be published and connected at a high velocity as the web enables users to link related resources (e.g., related documents and related data). These *linked data* [19] are commonly published by using technologies like (i) uniform resource identifiers (URIs) which identify resources, (ii) hypertext transfer protocol (HTTP) which retrieves or describes resources, and (iii) resource description framework (RDF) which graphically models linkage among resources. A collection of these data can be viewed as a big, interlinked, and dynamic graph structure of linked resources. Embedded in these data are implicit, previously unknown, and potentially useful knowledge. Having techniques for modelling, querying, and reasoning these linked data [13, 15] is desirable. In this paper, we focus on mining frequent subgraphs from these dynamic streaming graph structured data. Note that some existing algorithms require very large memory space to mine frequent subgraphs; some others discover collections of frequently co-occurring edges (which may be disjoint). In contrast, we propose—in this paper—algorithms that use *limited memory* space for discovering collections of frequently co-occurring *connected* edges.

1.1 Related Works

Since the introduction of the frequent pattern mining problem [2], numerous algorithms have been proposed [25, 27, 28]. For example, FP-growth [18] uses an in-memory extended prefix-tree structure called Frequent Pattern tree (FP-tree)—which captures the content of the transaction database—for mining sets of frequently co-occurring items (e.g., shopper market baskets of frequently purchased merchandise items) from traditional static databases (e.g., containing shopper market transactions). Some works [6, 17] use disk-based structure for mining. However, they mine from *static* databases.

As technology advances, *dynamic* streams of graph structured data (e.g., streams of semantic web, sensor network, social network, and road network data [10]) can be easily generated at high velocity. When comparing with mining from traditional *static* databases, mining from *dynamic* data streams [20, 29, 30] is more challenging due to the following properties of data streams: (i) *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 [11, 12]). (ii) *Streaming data are not necessarily uniformly distributed; their distributions are usually changing with time.*

©2015, Copyright is with the authors. Published in the Workshop Proceedings of the EDBT/ICDT 2015 Joint Conference (March 27, 2015, Brussels, Belgium) on CEUR-WS.org (ISSN 1613-0073). Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

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). To mine frequent patterns from data streams, both approximate and exact algorithms have been proposed. For instance, *approximate* algorithms (e.g., FP-streaming [16], TUF-streaming [24]) 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) [26] 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.

The aforementioned properties play an important role in the mining of data streams in general; they play a more challenging role in the mining of a specific class of streaming data—namely, *streams of graph structured data*. State-of-the-art solutions to these challenges include the following: Aggarwal et al. [1] 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, Valari et al. [31] discovered top-k dense subgraphs in dynamic graph collections by means of both exact and approximate algorithms. Furthermore, Chi et al. [9] proposed a fast graph stream classification algorithm that uses discriminative clique hashing (DICH), which can be applicable for OLAP analysis over evolving complex networks. We [5, 7] previously mined frequent patterns—in the form of collections of frequently co-occurring edges—from dense graph streams.

1.2 Our Contributions

Our previous solution [7] finds collections of frequently co-occurring edges, which include *connected* as well as *disjoint* edges. In many real-life situations (e.g., social or business applications [21, 22]), it is desirable to obtain collections of frequent disjoint edges so as to help the discovery of the missing links (e.g., connect two or more disjoint groups of social entities sharing common research or business interests). However, in some other situations, it is more efficient to find only the collections of frequent *connected* edges. Hence, in this paper, we propose algorithms that find collections of frequently co-occurring *connected* edges from streaming graph structured data. The algorithms either prune irrelevant (disjoint) edges at a post-processing step or push such a prune step early in the mining process. Consequently, only relevant patterns (i.e., frequent *connected* subgraphs) are returned to users. Moreover, as high volumes of streaming graph structured data can be generated at a high velocity, data may be too big to fit into memory. Our algorithms were designed in such a way that they use *limited memory*.

This paper is organized as follows. Background is provided in Section 2. Section 3 presents our algorithms that first build an on-disk data structure to capture and maintain relevant streaming graph structured data, recursively discover collections of frequent edges, and then prune those disjoint edges at a post-processing step. Section 4 presents an improved algorithm that pushes the prune step early in the mining process. Section 5 shows experimental results. Finally, conclusions are given in Section 6.

2. BACKGROUND

In this section, we provide background information on three different structures for capturing streaming data.

2.1 DSTree

When mining frequent patterns from streaming data, an exact algorithm [26] first constructs a *Data Stream Tree* (*DSTree*), which is then used as a *global tree* for recursive generation of smaller FP-trees (as *local trees*) for projected databases. Due to the dynamic nature of data streams, 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 streaming data. Note that the DSTree is designed for processing streams within a sliding window. 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 as in the FP-tree for frequent pattern mining from *static* databases). Each entry in this list captures the frequency of an 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.

After the construction of the (global) DSTree, it is always kept up-to-date when the window slides. The actual mining process is "delayed" until it is needed. To start mining, the algorithm first traverses relevant tree paths upwards and sums the frequency values of each list in a node representing an item (or a set of items)—to obtain its frequency in the current sliding window—for forming an appropriate projected database. Afterwards, the algorithm constructs a (local) 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 [18]). 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.

2.2 DSTable

The success of mining with the DSTree mainly relies on the assumption—usually made for many tree-based algorithms [18]—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 three subsequent local 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 (e.g., for streaming graph structured data) where the memory is so limited that not all these trees can fit into memory. To handle these situations, the *Data Stream Table* (*DSTable*) [8] 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 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 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.

Once the DSTable is constructed and updated, the algorithm first extracts relevant transactions from the DSTable. Then, the algorithm (i) constructs an 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. On the positive side, the algorithm finds all frequent patterns. On the negative side, 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. Moreover, each table entry is a "pointer" that indicates the location in terms of row name and column number 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). Furthermore, 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).

2.3 DSMatrix

The use of a two-dimensional structure called *Data Stream Matrix* (*DSMatrix*) [7] solves the aforementioned problems while mining frequent patterns from data streams with limited memory because this matrix structure captures the contents of transactions in all batches within the current sliding window by storing them on the disk. 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. The DSMatrix 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. Unlike the DSTable (in which boundaries may vary from one row representing an item to another row representing another item due to the potentially different number of items present), boundaries in DSMatrix are the same from one row to another because we put a binary value (0 or 1) for each transaction. Hence, the DSMatrix only keeps w boundary values (where $w \ll m \times w$) for the entire matrix, regardless how many domain items (m) are here. Moreover, as 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. 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, the DSMatrix takes $m \times |T|$ bits (cf. potentially $64m \times |T|$ bits for dense data streams required by the DSTree).

3. FREQUENT CONNECTED SUBGRAPH MINING WITH A POST-PROCESSING STEP

To find collections of frequent edges in streams of graph structured data, our proposed algorithms first construct a DSMatrix to capture and maintain within the current window those relevant streaming data. When a new batch of streaming graph structured data comes in, the window slides. Transactions in the oldest batch in the sliding window are then removed from the DSMatrix so that transactions in this new batch can be added. In other words, the mining is "delayed" until it is needed. Once the DSMatrix is constructed, it is kept up-to-date on the disk. See 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 stream of graphs, where each graph $G = (V, E)$ consists of $|V| = 4$ vertices (Vertices v_1, v_2, v_3 and v_4) and $|E| \leq 6$ edges:

- At time T_1, $E_1 = \{(v_1, v_4), (v_2, v_3), (v_3, v_4)\}$;
- At time T_2, $E_2 = \{(v_1, v_2), (v_2, v_4), (v_3, v_4)\}$;
- At time T_3, $E_3 = \{(v_1, v_2), (v_1, v_4), (v_3, v_4)\}$;
- At time T_4, $E_4 = \{(v_1, v_2), (v_1, v_4), (v_2, v_3), (v_3, v_4)\}$;
- At time T_5, $E_5 = \{(v_1, v_2), (v_2, v_3), (v_2, v_4), (v_3, v_4)\}$;
- At time T_6, $E_6 = \{(v_1, v_2), (v_1, v_3), (v_1, v_4)\}$;
- At time T_7, $E_7 = \{(v_1, v_2), (v_1, v_4), (v_3, v_4)\}$;
- At time T_8, $E_8 = \{(v_1, v_2), (v_1, v_4), (v_2, v_3), (v_3, v_4)\}$;
- At time T_9, $E_9 = \{(v_1, v_3), (v_1, v_4), (v_2, v_3)\}$.

See Figure 1. These graphs may represent some insertions, deletions, and/or updates on the linkages among linked data or documents in a semantic web. For simplicity, we represent these edges by six symbols a, b, c, d, e and f. Consequently, we get (i) edges $E_1 = \{c, d, f\}$, $E_2 = \{a, e, f\}$ and $E_3 = \{a, c, f\}$ in the first batch B_1; as well as (ii) edges $E_4 = \{a, c, d, f\}$, $E_5 = \{a, d, e, f\}$ and $E_6 = \{a, b, c\}$ in the second batch B_2. Then, the DSMatrix stores the following information at the end of time T_6:

DSMatrix (capturing E_1–E_6):

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

DSMatrix keeps track of the global boundary information (which is applicable for all rows).

When the third batch B_3 of streaming graph structured data flows in, the window slides. DSMatrix uses the boundary information to remove data in all columns up to Col 3 while keeping data in Col (3+1) to Col 6 (or more precisely, shifting all columns from Cols 4–6 to Cols 1–3. After the removal of the first three columns, DSMatrix appends three columns representing (iii) edges $E_7 = \{a, c, f\}, E_8 = \{a, c, d, f\}$ and $E_9 = \{b, c, d\}$ in the third batch B_3. In other words, DSMatrix stores the following information for E_4–E_9 in batches B_2 & B_3 at the end of time T_9.

DSMatrix (capturing E_4–E_9):

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

Again, DSMatrix keeps track of the global boundary information (which is applicable for all rows). □

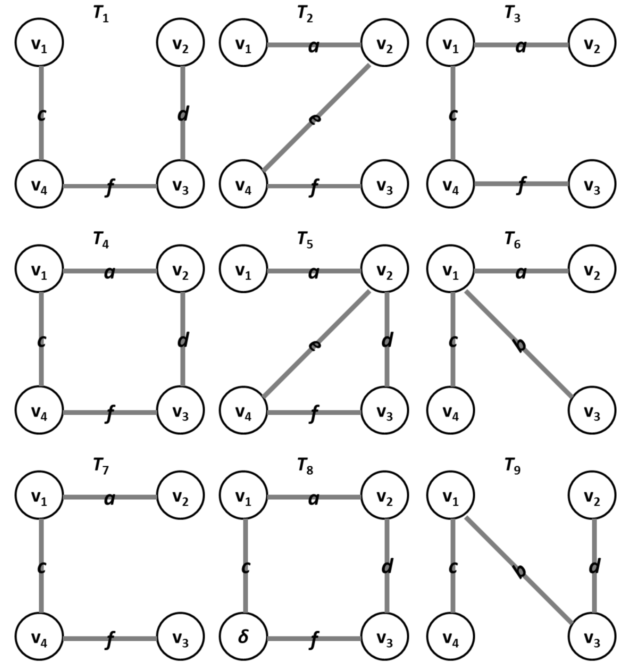

Figure 1: A stream of graph structured data (Example 1).

3.1 Mining with Multiple FP-trees

After constructing a DSMatrix, our first algorithm extracts columns from the DSMatrix to build a tree in memory for each $\{x\}$-projected database (which is a collection of all the edges containing x). Afterwards, the algorithm recursively finds collections of frequent edges from the tree for this projected database. See Example 2.

Example 2. At the end of time T_9, our first algorithm mines frequent patterns with the DSMatrix capturing E_4–E_9 in Example 1 by first forming 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 edges a appears in those five graphs in the two batches of streaming graph structured 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 edge-pairs $\{a, c\}, \{a, d\}$ 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 edge-triplets $\{a, c, d\}, \{a, c, f\}$ 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 edge-quadruplet $\{a, c, d, f\}$ is frequent. In the context of graph streams, this is a frequent collection of 4 edges—namely, Edges a, c, d and f. To recap, in addition to the five frequent singletons (i.e., edges a, b, c, d and f), a total of seven collections of frequent edges were found from the $\{a\}$-projected database: $\{a, c\}, \{a, c, d\}, \{a, c, d, f\}, \{a, c, f\}, \{a, d\}, \{a, d, f\}$ & $\{a, f\}$.

Afterward, we backtrack and examine the next frequent singleton $\{b\}$. For Row b, we notice that Columns 3 and 6 contain values "1" (which means that b appears in those two graphs in the current sliding window). For these two columns, we extract downward to get $\{c\}$ and $\{c, d\}$ that appear together with b (to form the $\{b\}$-projected database). The corresponding FP-tree contains $\{c\}$:2 meaning that c occurs twice with b (i.e., edge-pair $\{b, c\}$ is frequent with frequency 2). To recap, a total of 1 collection of frequent edges was found from the $\{b\}$-projected database: $\{b, c\}$.

Similar steps are applied to other frequent singletons $\{c\}$, $\{d\}$ and $\{f\}$ in order to discover all collections of frequent edges. For instance, a total of 3 collections of frequent edges

were found from the $\{c\}$-projected database: $\{c,d\}$, $\{c,d,f\}$ and $\{c,f\}$. Similarly, a total of 1 collection of frequent edges was found from the $\{d\}$-projected database: $\{d,f\}$. Consequently, our first algorithm found a total of 5+7+1+3+1 = 17 collections of frequent edges, which include some connected edges like $\{a,d\} \equiv \{(v_1,v_2),(v_2,v_3)\}$ as well as some disjoint edges such as $\{a,f\} \equiv \{(v_1,v_2),(v_3,v_4)\}$. □

3.2 Frequency Counting on a Single FP-tree

In Example 2, the mining process requires multiple FP-trees to be kept in the memory during the mining process. However, when the memory space is limited, *not* all of the multiple FP-trees can fit into the memory. One way to solve this problem is to apply an effective *frequency counting technique*: Once an FP-tree for the projected database of a frequent singleton is built, the algorithm traverses 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, the algorithm generates the collection of edges represented by the node and its subsets. We also compute their frequencies.

Example 3. Revisit Example 2 but with the frequency counting techniques applied to a single FP-tree. Specifically, our second algorithm first constructs an FP-tree for the $\{a\}$-projected database. It then traverses 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 edge-pair $\{a,c\}$ 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 subgraphs without recursive construction of FP-trees.

Afterwards, we build an FP-tree for the $\{b\}$-projected database and count frequencies of all frequent subgraphs containing item b. Similar steps are applied to the FP-trees for the $\{c\}$- and $\{d\}$-projected databases. Consequently, our second algorithm found the same 17 collections of frequent edges as those in Example 2. However, at any moment during the mining process, only one FP-tree needs to be constructed and kept in the memory (cf. multiple FP-trees required by our first algorithm described in Section 3.1). □

3.3 Mining a Single FP-tree in a Top-Down Fashion

An alternative way to avoid the construction of multiple FP-trees is to apply top-down tree mining (similar to that of the TD-FP-growth algorithm [32]). Specifically, we (i) form only a projected database for each frequent singleton (cf. Section 3.1, in which projected databases for singletons and non-singletons are recursively formed) and (ii) in reverse order—i.e., the top-down order (cf. bottom-up fashion as in the FP-growth algorithm or that described in Section 3.1).

Example 4. When applying this top-down tree-based mining, our third algorithm found the same 17 collections of frequent edges as those in Examples 2 and 3. □

3.4 Vertical Mining

With the representation of relevant graph structured data in the DSMatrix, it is logical to mine frequent subgraphs *vertically*. Specifically, our fourth algorithm examines each row (representing an edge). The *row sum* (i.e., total number of 1s) gives the frequency of the edge represented by that row. Once the frequent singleton edges are found, we *intersect the bit vectors* for two edges. If the row sum of the resulting intersection ≥ the user-specified *minsup* threshold, then we find a frequent edge-pair. We repeat these steps by intersecting two bit vectors of frequent patterns to find frequent subgraphs consisting of multiple edges.

Example 5. Revisit Examples 2, 3 and 4. Our fourth algorithm first computes the row sum for each row (i.e., for each domain item). As a result, we find that edges a,b,c,d and f are all frequent with frequencies 5, 2, 5, 4 and 4, respectively. Afterwards, the algorithm intersects 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 edge-pairs $\{a,c\},\{a,d\}$ 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 edge-triplets $\{a,c,d\},\{a,c,f\}$ and $\{a,d,f\}$. We also intersect $\overrightarrow{acd}$ with $\overrightarrow{acf}$ to find frequent edge-quadruplet $\{a,c,d,f\}$. These are all collections of frequent edges containing a.

Afterwards, we repeat similar steps with the bit vectors for other edges. 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 edge-triplets $\{c,d\}$ and $\{c,f\}$, each with frequencies of 3. We also find frequent edge-quadruplet $\{c,d,f\}$ by intersecting $\overrightarrow{cd}$ and $\overrightarrow{cf}$. Finally, we intersect $\vec{d}$ and $\vec{f}$ to find frequent edge-pair $\{d,f\}$ with frequency 3. Consequently, our fourth algorithm found the same 17 collections of frequent edges as those in Examples 2, 3 and 4. □

3.5 Post-Processing Step

So far, we have described how our four algorithms find collections of all frequent edges, which include *connected* edges such as $\{a,d\} \equiv \{(v_1,v_2),(v_2,v_3)\}$ as well as *disjoint* edges such as $\{a,f\} \equiv \{(v_1,v_2),(v_3,v_4)\}$. To filter out disjoint edges, we apply the following post-processing step to check every frequent edges. We look up the vertex information of each edge such as (v_1,v_2) for edge a. See Table 1. Let X represent a collection of multiple frequent edges. Then, for each edge $e \equiv (v_i,v_j) \in X$ (where $|X| \geq 2$), count the frequency (or occurrence) of v_i and v_j in X. If frequency of v_i (or v_j) is at least 2 in X, then v_i (or v_j) is a vertex connecting at least 2 edges (i.e., these 2 edges are connected):

- $\forall e \equiv (v_i,v_j) \in X$, [*frequency*$(v_i) \geq 2$ or *frequency*$(v_j) \geq 2$] $\Rightarrow X$ is a *connected* subgraph.

Otherwise—i.e., there exists an edge $e' \equiv (v'_i,v'_j)$—such that frequency of v'_i and that of v'_j are both less than 2 in X, such an edge e' is disjoint (i.e., an isolated edge):

Table 1: Table capturing vertices of each edge

EDGE	VERTICES
a	(v_1, v_2)
b	(v_1, v_3)
c	(v_1, v_4)
d	(v_2, v_3)
e	(v_2, v_4)
f	(v_3, v_4)

- $\exists e' \equiv (v'_i, v'_j) \in X'$, [*frequency*$(v'_i) < 2$ and *frequency*(v'_j) < 2] $\Rightarrow X'$ is *not* a connected subgraph.

For instance, we check and keep $\{a, d\}$ because it is a pair of *connected* edges; we check and prune away $\{a, f\}$ because it is a pair of *disjoint* edges.

Example 6. Continue with Examples 2, 3, 4, or 5. Before the post-processing step, each algorithm finds a total of 17 collections of frequent edges from the streaming graph structured data. Among them, let us consider $\{a, c\} \equiv \{(v_1, v_2), (v_1, v_4)\} = X$. (i) For (v_1, v_2), *frequency*$(v_1) = 2$ (and *frequency*$(v_2) = 1$); (ii) for (v_1, v_4), *frequency*$(v_1) = 2$ (and *frequency*$(v_4) = 1$). So, for each edge in X, it satisfies the condition that [*frequency*$(v_i) \geq 2$ or *frequency*$(v_j) \geq 2$]. Hence, X is a connected subgraph.

In contrast, consider $\{a, f\} \equiv \{(v_1, v_2), (v_3, v_4)\} = X'$. For (v_1, v_2), *frequency*$(v_1) = 1$ and *frequency*$(v_2) = 1$. Hence, there exists an edge $(v_1, v_2) \in X'$ such that [*frequency*(v_1) < 2 and *frequency*$(v_2) < 2$]. Hence, X' is *not* a connected subgraph.

Similarly, consider $\{c, d\} \equiv \{(v_1, v_4), (v_2, v_3)\} = X''$. For (v_1, v_4), *frequency*$(v_1) = 1$ and *frequency*$(v_4) = 1$. Hence, there exists an edge $(v_1, v_4) \in X''$ such that [*frequency*(v_1) < 2 and *frequency*$(v_4) < 2$]. Hence, X'' is *not* a connected subgraph.

Applying a similar post-processing step to check all 17 collections of frequent edges, we find that $\{a, f\} \equiv \{(v_1, v_2), (v_3, v_4)\}$ (consisting of two disjoint edges $a \equiv (v_1, v_2)$ and $f \equiv (v_3, v_4)$) and $\{c, d\} \equiv \{(v_1, v_4), (v_2, v_3)\}$ (consisting of two disjoint edges $c \equiv (v_1, v_4)$ and $d \equiv (v_2, v_3)$) are both not connected subgraphs, and thus can be pruned. Consequently, only 15 frequent *connected* subgraphs are then returned to the user. □

4. DIRECT FREQUENT CONNECTED SUBGRAPH MINING

So far, we have described how to mine frequent connected subgraphs by finding all collections of frequent edges and then pruning collections of disjoint edges in a post-processing step. When the number of vertices increases, chances of having disjoint edges also increase. Consequently, a lot of time and effort may have been spent on mining all collections of frequent edges including many disjoint edges, which are then pruned. To deal with this issue, we propose an alternative algorithm that mines frequent connected subgraphs directly.

Specifically, our fifth algorithm *directly* mines frequent connected subgraphs vertically. First, to mine frequent singletons, we examine each row (representing an edge). The *row sum* (i.e., total number of 1s) gives the frequency of the edge represented by that row. If the row sum $\geq$ the user-specified *minsup* threshold, then we find a frequent edge.

Table 2: Table capturing neighbors of each edge

EDGE	NEIGHBORING EDGES
a	b, c, d, e
b	a, c, d, f
c	a, b, e, f
d	a, b, e, f
e	a, c, d, f
f	b, c, e, d

Once the frequent singleton edges are found, we *intersect the bit vectors* for two *connected* edges based on the neighborhood information. See Table 2. If the row sum of the resulting intersection $\geq$ the user-specified *minsup* threshold, then we find a frequent *connected* subgraph consisting of 2 edges. We repeat these steps by intersecting two bit vectors of frequent connected subgraphs to find frequent connected subgraph of multiple edges.

During the mining process, the neighborhood information for frequent edge can be looked up from Table 2. The neighborhood information for a frequent connected pair $\{x, y\}$ can be computed by the following:

$$\begin{aligned} & \textit{neighbor}(\{x, y\}) \\ = \quad & \textit{neighbor}(\{x\}) \cup \textit{neighbor}(\{y\}) - \{x, y\}, \end{aligned} \qquad (1)$$

where $y \in \textit{neighbor}(\{x\})$. Similarly, the neighborhood information for a frequent connected subgraph $X \cup \{y\}$ consisting of k edges can be computed by the following:

$$\begin{aligned} & \textit{neighbor}(X \cup \{y\}) \\ = \quad & \textit{neighbor}(X) \cup \textit{neighbor}(\{y\}) - X - \{y\}, \end{aligned} \qquad (2)$$

where (i) $y \in \textit{neighbor}(X)$ and (ii) $|X| = k - 1$.

Example 7. Revisit Example 6. Our direct algorithm first computes the row sum for each row (i.e., for each edge). As a result, we find that edges 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 bit vectors of any of its neighbor *neighbor*$(\{\text{a}\}) = \{b, c, d, e\}$ to find the following:

- connected subgraph $\{a, b\}$ consisting of 2 edges a & b and with frequency 1 and thus infrequent;
- connected subgraph $\{a, c\}$ consisting of 2 edges a & c and with frequency 4 and thus frequent; as well as
- connected subgraph $\{a, d\}$ consisting of 2 edges a & d and with frequency 3 and thus frequent.

Note that, as the algorithm only intersects vectors of frequent edges, it does not intersect with infrequent edge e even though $e \in \textit{neighbor}(\{a\})$. Moreover, when compared with Example 5, our direct algorithm does not produce $\{a, f\}$. Although single edge f is frequent, it is not in the neighborhood of $\{a\}$ and thus not connected with a.

Next, we intersect (i) $\overrightarrow{ac}$ with $\overrightarrow{d}$ to find frequent connected edge-triplet $\{a, c, d\}$ because $d \in \textit{neighbor}(\{a, c\})$, which can be computed as $\textit{neighbor}(\{a\}) \cup \textit{neighbor}(\{c\}) - \{a, c\} = \{b, d, e, f\}$. Then, we intersect (i) $\overrightarrow{acd}$ with $\overrightarrow{f}$ to get connected edge-quadruplet $\{a, c, d, f\}$ because $f \in \textit{neighbor}(\{a, c, d\})$, which is computed as $\textit{neighbor}(\{a, c\})$ $\cup \textit{neighbor}(\{d\}) - \{a, c, d\} = \{b, e, f\}$. Similarly, we intersect (i) $\overrightarrow{ac}$ with $\overrightarrow{f}$ to find frequent connected edge-triplet

$\{a, c, f\}$ as $f \in neighbor(\{a, c\})$. We also intersect (i) $\overrightarrow{ad}$ with $\vec{f}$ to get frequent connected edge-triplet $\{a, d, f\}$ because $neighbor(\{a, d\}) = neighbor(\{a\}) \cup neighbor(\{d\}) - \{a, d\} = \{b, c, e, f\}$ contains f. These are all collections of frequent *connected* edges containing a. In the above procedure, we only extend on connected subgraphs.

Afterwards, we repeat similar steps with the bit vectors for other edges. 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 $\vec{f}$ to find frequent connected edge-pair $\{c, f\}$ with frequency 3. Note that we do not intersect $\vec{c}$ with $\vec{d}$ because $d \notin neighbor(\{c\}) = \{a, b, e, f\}$. However, we find frequent edge-triplet $\{c, d, f\}$ by intersecting $\overrightarrow{cf}$ and $\vec{d}$ because $d \in neighbor(\{c, f\}) = neighbor(\{c\}) \cup neighbor(\{f\}) - \{c, f\} = \{a, b, d, e\}$. Finally, we intersect $\vec{d}$ and $\vec{f}$ to find frequent edge-pair $\{d, f\}$ having frequency 3. □

5. EXPERIMENTAL EVALUATION

To acquire streams of linked graph structured data, 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 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 first experiment, we measured the accuracy of mining with the following structures: (i) DSTree [26], (ii) DSTable [8], and (iii) DSMatrix. Experimental results show that the four mining algorithms that use the DSMatrix with the post-processing steps (Section 3) gave the same mining results as the direct algorithm (Section 4) that uses the DSMatrix without the post-processing step. Experimental results also show that these five algorithms (which all use the DSMatrix) gave the same mining results as any algorithms that conduct mining with the DSTree or DSTable.

In the second experiment, we measured the space efficiency. Experimental results show that mining with the DSTree stored one global DSTree and multiple local FP-trees in main memory, and thus took the largest main memory space. Mining with the DSTable and DSMatrix required less memory because the DSTable and DSMatrix were kept on disk. Among those algorithms that mine with the DSMatrix, the first algorithm (i.e., the one mines with multiple FP-trees and described in Section 3.1) required the largest amount of memory space because it keeps at most k FP-trees in the memory during the entire mining process, where k is

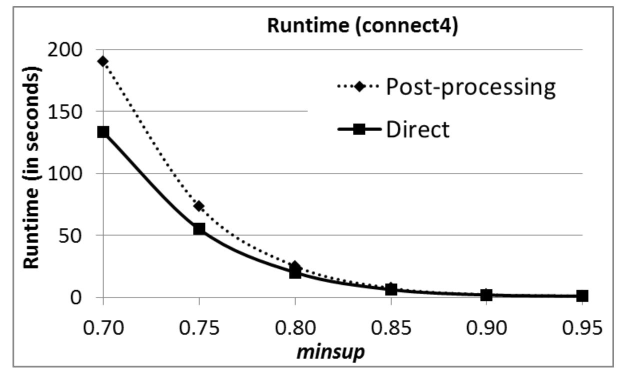

Figure 2: Experimental results on vertical mining.

the maximum number of edges in any collection of frequent edges. The algorithms that mine with a single FP-tree (Sections 3.2 and 3.3) required less space because they keep at most one FP-tree in the memory during the entire mining process. The two vertical mining algorithms (Sections 3.4 and 4) required the least amount of memory space because they both work with bit vectors.

In the third experiment, we measured the time efficiency. Among those algorithms that mine with the DSMatrix, the first algorithm (i.e., the one mines with multiple FP-trees and described in Section 3.1) required the longest runtime because it recursively constructs FP-trees during the entire mining process. The algorithms that mine with a single FP-tree (Sections 3.2 and 3.3) required shorter runtime because they construct at most one FP-tree for each frequent edge (i.e., for a total of at most m FP-trees, one for each of the $|E|$ edges) during the entire mining process. The two vertical mining algorithms (Sections 3.4 and 4) required the shortest runtime because they both work with bitwise and set intersection operators. Between these two vertical mining algorithms, as expected, the one with the post-processing step required longer runtime than the direct algorithm because the latter mines frequent connected subgraphs directly. Figure 2 shows the runtimes of our fourth algorithm (i.e., vertical mining with *post-processing* step) and our fifth algorithm (i.e., *direct* vertical mining).

We also performed some additional experiments (e.g., evaluating the effect of *minsup*). Results show that the runtime decreased when *minsup* increased. In another experiment, we tested scalability with the number of batches in the stream of graph structured data. The results show that the scalability of our (five) algorithms, especially the two vertical mining algorithms.

As future work, we plan to conduct more extensive experiments on various datasets (including Big data) with different parameter settings (e.g., varying *minsup*, the number of vertices and edges in graph structured data and/or linked data).

6. CONCLUSIONS

Motivated by the demand of having algorithms that use *limited memory* space for discovering collections of frequently co-occurring *connected* edges from big, interlinked, dynamic graph structures of linked data, we proposed five algorithms for frequent subgraph mining. All our algorithms use a DSMatrix to capture important contents of streams of graph structured linked data. The DSMatrix is updated when the window slides. The discovery of frequent connected sub-

graphs is "delayed" until the mining is needed. Three of our algorithms use horizontal tree-based mining approaches: (i) The first algorithm builds multiple FP-trees recursively in a bottom-up fashion; (ii) the second algorithm builds FP-trees in a bottom-up fashion, but builds only a single FP-tree for each singleton; and (iii) the third algorithm also builds only a single FP-tree for each singleton, but builds in a top-down fashion. The fourth algorithm uses a vertical bitwise mining approach. Note that all these four algorithms mine collections of all frequent (connected or disjoint) edges, and prune those disjoint edges at a *post-processing* step. In contrast, our fifth algorithm also uses a vertical bitwise mining approach, but *directly* mines collections of all *connected* edges. Experimental results show the space and time efficiency of vertical frequent subgraph mining from streams of linked graph structured data.

7. ACKNOWLEDGEMENTS

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

8. REFERENCES

[1] 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)

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

[3] D. Bianchini, S. Castano, V. de Antonellis, A. Ferrara, E. Quintarelli, & L. Tanca. RUBIK: proactive, entity-centric and personalized situational web application design. *TLDKS*, **13**, pp. 123–157 (2014)

[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] P. Braun, J.J. Cameron, A. Cuzzocrea, F. Jiang, & C.K. Leung. Effectively and efficiently mining frequent patterns from dense graph streams on disk. *Procedia Computer Science*, **35**, pp. 338–347 (2014)

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

[7] J.J. Cameron, A. Cuzzocrea, F. Jiang, & C.K. Leung. Frequent pattern mining from dense graph streams. In *Proc. EDBT/ICDT 2014 Workshops*, pp. 240–247.

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

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

[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. *DKE*, **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. Cuzzocrea, C.K. Leung, & S.K. Tanbeer. Mining of diverse social entities from linked data. In *Proc. EDBT/ICDT 2014 Workshops*, pp. 269-274.

[14] R. de Virgilio & D. Bianchini. SeeVa: a model based framework for semantic web service discovery. *TLDKS*, **14**, pp. 51–82 (2014)

[15] A. Ferrara, L. Genta, & S. Montanelli. Linked data classification: a feature-based approach. In *Proc. EDBT/ICDT 2013 Workshops*, pp. 75–82.

[16] 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)

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

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

[19] T. Heath & C. Bizer *Linked data: evolving the web into a global data space*. Synthesis lectures on the semantic web: theory and technology, Morgan & Claypool, 2011.

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

[21] F. Jiang & C.K. Leung. A business intelligence solution for frequent pattern mining on social networks. In *Proc. IIEEE ICDM Workshops 2014*, pp. 789–796.

[22] F. Jiang, C.K. Leung, D. Liu, & A.M. Peddle. Discovery of really popular friends from social networks. In *Proc. IEEE BDCloud 2014*, pp. 342–349.

[23] W. Lee, C.K. Leung, & J.J. Song. Reducing noises for recall-oriented patent retrieval. In *Proc. IEEE BDCloud 2014*, pp. 579–586.

[24] 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)

[25] C.K. Leung & F. Jiang. A data science solution for mining interesting patterns from uncertain big data. In *Proc. IEEE BDCloud 2014*, pp. 235–242.

[26] 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.

[27] C.K. Leung, R.K. MacKinnon, & S.K. Tanbeer. Fast algorithms for frequent itemset mining from uncertain data. In *Proc. IEEE ICDM 2014*, pp. 893–898.

[28] R.K. MacKinnon, T.D. Strauss, & C.K. Leung. DISC: efficient uncertain frequent pattern mining with tightened upper bounds. In *Proc. IIEEE ICDM Workshops 2014*, pp. 1038–1045.

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

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

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

[32] K. Wang, L. Tang, J. Han, & J. Liu. Top down FP-growth for association rule mining. In *Proc. PKDD 2002*, pp. 334–340.