Mining Unusual Patterns by Multi-Dimensional Analysis of Data Streams *

Jiawei Han
Department of Computer Science
University of Illinois at Urbana-Champaign
Email: hanj@cs.uiuc.edu

Abstract

It has been popularly recognized that stream data represents an important form of data, with broad applications. There have been a lot of studies on effective stream data management and query processing, as well as some recent studies on stream data mining. Although this is a promising direction, most existing studies have not paid enough attention to one critical fact: most data streams reside at a rather low level of abstraction and are multi-dimensional in nature, whereas most analysts are interested in finding characteristic features, unusual patterns, and dynamic changes (such as trends and outliers) at relatively high levels of abstraction and in certain multi-dimensional space. To accomplish such tasks, one may need to develop effective mechanisms for on-line, multi-dimensional analysis and mining of stream data. This poses great challenges on system architecture, implementation methodology, algorithm development, and performance tuning.

In this paper, we discuss the issues related to effective, on-line, multi-dimensional analysis and mining of unusual events and patterns in data streams, including research challenges, potential architectures, and implementation methodologies.

1 Introduction

Recent emerging applications call for study of a new kind of data, called stream data, where data takes the form of continuous, potentially infinite data streams, as opposed to finite, statically stored data sets. Stream data has broad applications because huge amounts of dynamically changing data are in this form, including time-series data, scientific and engineering data, and data produced in other dynamic environments, such as power supply, network traffic, stock exchange, telecommunication, Web clicking, weather or environment monitoring, and so on. Stream data management systems and continuous stream query processors are under popular investigation and development [3, 4, 11, 13, 12, 8]. Moreover, mining data streams, including classification of stream data [9, 18], clustering data streams [13, 24], and finding frequent patterns in data streams [23] are also under popular investigation.

Although this is exciting, most existing studies have not paid enough attention to one critical fact: most data streams reside at rather low level of abstraction and are multi-dimensional in nature, whereas most analysts are interested in finding characteristic features, unusual patterns, and dynamic changes (such as trends and outliers) at relatively high levels of abstraction and in certain multi-dimensional space. To accomplish such tasks, one may need to develop effective mechanisms for on-line, multi-dimensional analysis and mining of stream data. This poses great challenges on system architecture, implementation methodology, algorithm development, and performance tuning.

Let’s examine an example.

Example 1. Suppose that a Web server, such as Yahoo.com, receives a huge volume of Web click streams, requesting various kinds of services and information. Usually, such stream data resides at rather low level, consisting of time (down to subseconds), Web page address (down to concrete URL), user ip address (down to detailed machine IP address), etc. However, an analyst may often be interested in changes, trends, and unusual patterns, happening in the data streams, at certain high levels of abstraction. For example, it is interesting to find that the Web clicking traffic in North
America on sports in the last 15 minutes is 40% higher than the last 24 hours' average.

From the point of view of a Web analysis provider, given a large volume of fast changing Web click streams, and with limited resource and computational power, it is only realistic to analyze the changes of Web usage at certain high levels, discover unusual events and patterns, and drill down to some more detailed levels for in-depth analysis, when needed. Moreover, such analysis should be done in almost real time in order to make timely responses.

Interestingly, both the analyst and analysis provider share a similar view on such stream data analysis: instead of bogging down to every detail of data stream, a demanding request is to provide on-line analysis of changes, trends and unusual patterns at high levels of abstraction, with low cost and fast response time.

In this paper, we examine the research challenges and potential methods for mining unusual events and patterns by multidimensional analysis of data streams.

The remaining of the paper is organized as follows. In Section 2, we discuss the major challenges for multidimensional analysis and mining of data streams. In Section 3, we examine the issues on construction of stream data cubes and present an implementation methodology. In Section 4, we examine the issues on mining multi-dimensional frequent patterns and another implementation methodology. A summary of the general methodologies for multidimensional mining of data streams is presented in Section 5, which also concludes our study.

2 Major Challenges for Multidimensional Analysis and Mining of Data Streams

Stream data is generated continuously in a dynamic environment, with huge volume, potentially infinite flow, and fast changing behavior. Mining this new kinds of data, due to its huge volume, limited resources, and the often required fast response time, poses great challenges to research. Here we examine a few important issues.

1. There have been many applications that require multidimensional analysis and mining of data streams. Besides the example presented above which requires multi-dimensional analysis of stream characteristics, other examples include (1) comparison of multiple data streams to find unusual ones, e.g., which sector of stocks and in which region may have rather different trend from others; (2) multi-dimensional classification of data streams, e.g., in the analysis of power consumption streams, how to build classification scheme to distinguish power-user from budget-user based on the associated properties, including user group, period of time, history of changes, etc.; and (3) clustering high dimensional data, e.g., clustering streams of e-mails which contain many keywords and other high dimensional features.

2. For mining stream data, an important question is at what levels of abstraction one would like to mine. As collected, stream data is almost always at rather low level, consisting of various kinds of detailed temporal and other pieces of information. To find interesting or unusual patterns, it is essential to perform analysis and mining at certain meaningful abstraction level, discover critical changes of data, and, when needed, drill down to more detailed levels for in-depth analysis. However, people rarely want to drill down to the lowest level, which is not only uninteresting but also expensive. Thus it is beneficial for both implementors and users to have a minimal level of abstraction in stream data analysis.

3. Incremental computation and partial materialization seem to be the necessity in stream data analysis. To meet the requirement of fast response time with multi-dimensional data streaming in, it is unavoidable to incrementally compute and save some partial or intermediate results because it is too slow to compute everything from scratch on line. On the other hand, it is also too costly in both space and time to pre-compute and materialize the entire multi-dimensional data cube or the complete set of mining results for data streams. Thus in-depth study is needed to examine how to incrementally compute and partially materialize intermediate results for stream data.

4. Mining queries: continuous vs. one-time vs. continuous plus limited ad-hoc drilling. It is too expensive to mine all the patterns in a multi-dimensional space in real time. Thus it is realistic to ask users to pose queries with user-specified constraints. The question becomes whether the mining query should be a continuous one (posing the same query while new data streaming in), a one-time one (one-time, ad-hoc mining query), or continuous plus limited ad-hoc drilling (continuous in general, but when certain condition holds, providing limited flexibility for drilling around surrounding areas).
5. Tilt-time frame for mining patterns with fading factor and mining evolutionary patterns. In stream data analysis, people are usually interested in recent changes at a fine scale, but long term changes at a coarse scale. Since one cannot register all the patterns or data along the entire time spectrum, one should use a tilt-time frame, i.e., time at different levels of granularity: The most recent time is registered at the finest granularity; the more distant time is registered at coarser granularity; and the level of coarseness depends on the application requirements and on how old the time point is (from the current time). With this time frame, one can register data correspondingly and mine patterns between two registered time frames and the evolution of patterns along with time.

In the following discussion, we examine two typical scenarios: one is multidimensional analysis of stream data using stream cubes; and the other is mining multidimensional frequent patterns with stream data. We will see how the above methodologies can be applied to effective stream data analysis.

3 Construction of Stream Data Cubes for Multi-Dimensional Analysis of Stream Data

The design and construction of data cubes greatly facilitate fast, on-line, multidimensional analysis of data warehouse data. Therefore, we may predict that the design and development of stream data cube may play an important role at multi-dimensional analysis of stream data. In this section, we examine a stream data cube architecture and the methodologies for development and use of such a stream cube for multi-dimensional stream data analysis.

3.1 A stream data cube architecture

The major challenges for construction of a stream data cube is the huge volume of data, and the potentially even huger size of stream data cube, and the incremental update of such a stream cube. In [7], we propose a stream data cube architecture with the following features: (1) tilt time frame, (2) two critical layers: a minimal interesting layer and an observation layer; and (3) an efficient data structure for partial computation of data cubes. The stream data cubes so constructed are much smaller than those constructed from the raw stream data but will still be effective for many multi-dimensional stream data analysis tasks.

3.1.1 Tilt time frame

A tilt time frame can adopt two kinds of windows: natural time partition and logarithmic scale time partition, as shown in the following example.

Example 2. A tilt time frame can be constructed in natural time partition as shown in Fig. 1(a), where the time frame is structured in multiple granularity: the most recent 4 quarters (15 minutes), then the last 24 hours, 31 days, and 12 months. Based on this model, one can compute frequent itemsets in the last hour with the precision of quarter of an hour, the last day with the precision of hour, and so on, until the whole year, with the precision of month1. This model registers only 4 + 24 + 31 + 12 = 71 units of time instead of 366 × 24 × 4 = 35,136 units, a saving of about 495 times, with an acceptable trade-off of the grain of granularity at a distant time.

As an alternative, the tilt time frame can also be constructed in logarithmic scale time partition as shown in Fig. 1(b), where the time frame is structured in multiple granularity according to a logarithmic scale. Suppose the current window holds the transactions in the current quarter. Then the remaining slots are for the last quarter, the next two quarters, 4 quarters, 8 quarters, 16 quarters, etc., growing at an exponential rate. According to this model, with one year of data and the finest precision at quarter, we will need log₂(365 × 24 × 4) + 1 = 16.1 units of time instead of 366 × 24 × 4 = 35,136 units. That is, we will just need 17 time frames to store the compressed information.

![Figure 1](image-url)
3.1.2 Critical layers

Even with the tilt time frame model, it could still be too costly to dynamically compute and store a full cube since such a cube may have quite a few number of dimensions, each containing multiple levels with many distinct values. Since stream data analysis has only limited memory space but requires fast response time, a realistic arrangement is to compute and store only some mission-critical cuboids in the cube.

In our design, two critical cuboids are identified due to their conceptual and computational importance in stream data analysis. We call these cuboids layers and suggest to compute and store them dynamically. The first layer, called \( m \)-layer, is the minimally interesting layer that an analyst would like to study. It is necessary to have such a layer since it is often neither cost-effective nor practically interesting to examine the minute detail of stream data. The second layer, called \( o \)-layer, is the observation layer at which an analyst (or an automated system) would like to check and make decisions of either signaling the exceptions, or drilling on the exception cells down to lower layers to find their lower-level exceptional descendants.

![Figure 2](image)

**Figure 2.** Two layers (\( m \)-layer and \( o \)-layer) in a lattice representing a virtual stream cube

**Example 3.** Assume that \( \text{(individual\_user, URL, minute)} \) forms the primitive layer of the input stream data in Ex. 1. With the tilt time frame as shown in Figure 1, the two critical layers for power supply analysis are: (1) the \( m \)-layer: \( \text{(user\_group, URL\_group, quarter)} \), and (2) the \( o \)-layer: \( \text{(*, theme, hour)} \), as shown in Figure 2, where the solid lines indicate the drillable space from \( o \)-layer down whereas the dashed and solid lines indicate the drillable space from \( m \)-layer up, with the dashed ones indicating those not covered by the drill-down paths from the \( o \)-layer.

Based on this design, the cuboids lower than the \( m \)-layer will not need to be computed since they are beyond the minimal interest of users. Thus the minimal aggregate cells that our base cuboid needs to be computed and stored will be those computed by grouping by \text{user\_group, URL\_group, and quarter}. This can be done by aggregations (1) on two dimensions, \text{user} and \text{URL}, by rolling up from \text{individual\_user} to \text{user\_group} and from \text{URL} to \text{URL\_group}, respectively, and (2) on time dimension by rolling up from \text{minute} to \text{quarter}.

Similarly, the cuboids at the \( o \)-layer should be computed dynamically according to the tilt time frame model as well. This is the layer that an analyst takes as an observation deck, watching the changes of the current stream data by examining the slope of changes at this layer to make decisions. The layer can be obtained by rolling up the cube (1) along two dimensions to * (which means all \text{user\_category} and \text{theme}, respectively, and (2) along time dimension to \text{hour}. If something unusual is observed, the analyst can drill down to examine the details and the exceptional cells at low levels.

3.2 Partial materialization of stream cube

Materializing a cube at only two critical layers leaves much room for how to compute the cuboids in between. These cuboids can be precomputed fully, partially, not at all (i.e., leave everything computed on-the-fly), or precomputing exception cells only.

Let us first examine the feasibility of each possible choice in the environment of stream data. Since there may be a large number of cuboids between these two layers and each may contain many cells, it is often too costly in both space and time to fully materialize these cuboids, especially for stream data. On the other hand, materializing nothing forces all the aggregate cells to be computed on-the-fly, which may slow down the response time substantially. Moreover, for the choice of computing exception cells only, the problem becomes how to set up an exception threshold. A too low threshold may lead to computing almost the whole cube, whereas a too high threshold may leave a lot of cells uncomputed and thus not being able to answer many interesting queries. Thus, it seems that the only viable choice is to perform partial materialization of a stream cube.

Partial materialization of data cubes has been studied in previous work [17, 6]. With the concern of both space and on-line computation time, we propose a “popular path” approach, which computes and maintains a single popular aggregation path from \( m \)-layer to \( o \)-layer so that queries directly on those (layers) along the popular path can be answered without further computation, whereas those deviating from the path can be answered with minimal computation from those reach-
able from the computed layers.

We present these ideas using an example.

**Example 4.** Suppose the stream data to be analyzed contains 3 dimensions, A, B, and C, each with 3 levels of abstraction (excluding the highest level of abstraction “*”), as \( (A_1, A_2, A_3), (B_1, B_2, B_3), (C_1, C_2, C_3) \), where the ordering of “* > A_1 > A_2 > A_3” forms a high-to-low hierarchy, and so on. The minimal interesting layer (the \( m \)-layer) is \( (A_2, B_2, C_2) \), and the \( o \)-layer is \( (A_1, *, C_1) \). From the \( m \)-layer (the bottom cuboid) to the \( o \)-layer (the top-cuboid to be computed), there are in total \( 2 \times 3 \times 2 = 12 \) cuboids, as shown in Figure 3.

Suppose that the popular drilling path is given (which can usually be derived based on domain expert knowledge, statistical analysis or experiments). Assume that the given popular path is \( ((A_1, C_1) \rightarrow B_1 \rightarrow B_2 \rightarrow A_2 \rightarrow C_2) \), shown as the dark-line path in Figure 3, where \( (A_1, C_1) \rightarrow B_1 \) means that the drilling is performed from the cuboid \( (A_1, *, C_1) \) to the cuboid \( (A_1, B_1, C_1) \).

![Figure 3. Cube structure from the \( m \)- to the \( o \)-layer](image)

**3.3 An H-tree structure for efficient computation of partial stream data cubes**

Although efficient cube computation methods have been studied with several efficient methods developed, including multi-way array aggregation [30], BUC [5], and H-cubing [15], for computing partial stream data cube by aggregating regression cells from the \( m \)-layer up to the \( o \)-layer, an extended H-cubing method provides space-saving, efficient computation as reasoned below.

First, based on the notion of \( m \)-layer, i.e., the minimal interesting layer, and the tilt time frame, stream data can be directly aggregated to this layer according to the tilt time scale. Then the data can be further aggregated following the popular path to reach the observation layer.

To facilitate efficient computation and storage of such a partial stream cube, a compact data structure, called H-tree, a hyper-linked tree structure introduced in [15], is revised and adopted here to ensure that a compact structure is maintained in memory for efficient computation of multi-dimensional and multi-level aggregations.

**Example 5.** An example H-tree is shown in Fig 4. The H-tree from root to leaf is ordered the same as the popular path. This ordering generates a compact tree because the set of low level nodes that share the same set of high level ancestors will share the same prefix path using the tree structure. Each tuple, which represents the currently in-flow stream data, after being generalized to the \( m \)-layer, is inserted into the corresponding path of the H-tree. In the leaf node of each path, we store relevant measure information of the cells of the \( m \)-layer. The upper level measures are computed using the H-tree and its associated links.

An obvious advantage of the popular path approach is that the nonleaf nodes represents the cells of those layers (cuboids) along the popular path. Thus these nonleaf nodes naturally serves as the cells of the cuboids along the path. That is, it serves as a data structure for intermediate computation as well as the storage area for the computed measures of the layers (i.e., cuboids) along the path.

Furthermore, the H-tree structure facilitates the computation of other cuboids or cells in those cuboids. When a query or drill-down clicking requests to compute cells outside the popular path, one can find the closest lower level computed cells and use such intermediate computation results to compute the measures requested, because the corresponding cells can be found via a linked list of all the corresponding nodes contributing to the cells.

![Figure 4. H-tree structure for cube computation](image)
Algorithm 1 (Popular-path) Computing cuboids along the popular-path between the m-layer and the o-layer.

Input. (1) multi-dimensional multi-level stream data, (2) the m and o-layer specifications, and (3) a given popular drilling path.

Output. All the aggregated cells of the cuboids along the popular path (and between the m- and o-layers).

Method.

1. Each tuple, which represents a minimal addressing unit of multi-dimensional multi-level stream data, is scanned once and generalized to the m-layer. The generalized tuple is then inserted into the corresponding path of the H-tree, increasing the count and aggregating the measure values of the corresponding leaf node.

2. Since each branch of the H-tree is organized in the same order as the specified popular path, aggregation is performed from the m-layer all the way up to the o-layer by aggregating along the popular path. The step-by-step aggregation is performed while inserting every new generalized tuple.

3. The aggregated cells are stored in the nonleaf nodes in the H-tree, forming the computed cuboids along the popular path.

Analysis. The H-tree ordering is based on the popular drilling path given by users or experts. This ordering facilitates the computation and storage of the cuboids along the path. The aggregations along the drilling path from the m-layer to the o-layer are performed during the generalizing of the stream data to the m-layer, which takes only one scan of stream data. Since all the cells to be computed are the cuboids along the popular path, and the cuboids to be computed are the nonleaf nodes associated with the H-tree, both space and computation overheads are minimized.

3.4 Incremental update based on the stream cubing model

Although our algorithms have been developed for computing stream data for all the data accumulated so far from the start of the stream, the process discussed in the algorithm is essentially an incremental computation method, using the tilt time frame of Figure 1. Assuming that the memory contains the previously computed m and o-layers, plus the cuboids along the popular path, and stream data arrive every minute. The new stream data are accumulated (by generalization) in the corresponding H-tree leaf nodes. Since the time granularity of the m-layer is quarter, the aggregated data will trigger the cube computation once every quarter, which rolls up from leaf to the higher level cuboids. When reaching a cuboid whose time granularity is hour, the rolled measure information remains in the corresponding quarter slot until it reaches the full hour (i.e., 4 quarters) and then it rolls up to even higher levels, and so on.

Notice in this process, the measure in the time interval of each cuboid will be accumulated and promoted to the corresponding coarser time granularity, when the accumulated data reaches the corresponding time boundary. For example, the measure information of every 24 hours will be aggregated to one day and be promoted to the day slot, and in the mean time, the hour slots will still retain sufficient information for hour-based analysis. This design ensures that although the stream data flows in-and-out, measure always keeps up to the most recent granularity time unit at each layer.

The feasibility and performance study in [7] have demonstrated that this model is efficient and effective, with minimal space-overhead for multi-dimensional analysis of stream data.

4 Mining Multi-Dimensional Frequent Patterns in Data Streams

Frequent-pattern mining has been studied extensively in data mining, with many algorithms proposed and implemented (e.g., Apriori [1], FP-growth [16], CLOSET [25], and CHARM [29]). Frequent pattern mining and its associated methods have been popularly used in association rule mining [1], sequential pattern mining [2], structured pattern mining [20], iceberg cube computation [5], cube gradient analysis [19], associative classification [21], frequent pattern-based clustering [27], and so on. However, it is challenging to mine frequent patterns in data streams because mining frequent itemsets is essentially a set of join operations as illustrated in [1], whereas join is a typical blocking operator, i.e., computation for any itemset cannot complete before seeing the past and future data sets. Since one can only maintain a limited size window due to the huge amount of stream data, it is difficult to mine and update frequent patterns in a dynamic, data stream environment.

Recently, [23] studied mining frequent counts in streams by exploring an active mining approach, i.e., one keeps reporting or updating the frequent patterns regularly when new data items stream in. However, since the set of frequent patterns easily goes larger than the set of transactions, and relationships among
frequent items change dynamically with incoming data streams, it is difficult to maintain mined patterns in a fast updating, streaming environment. Especially, without good knowledge of application, it is difficult to predetermine the \( \minsup \) threshold or maintain a reasonable size frequent patterns with data streams.

In our model, which is called \( \text{FP-stream} \) \cite{10}, instead of actively mining frequent patterns, we adopt a lazy mining approach: actively maintain the compressed frequent portion of the transaction database and perform mining only in response to a user’s particular mining request, i.e., “keep preparation ready but mine it only upon request”. We believe this could be a more realistic methodology for mining data streams than actively updating the mined patterns.

With our proposed \( \text{FP-stream} \) model, it is easy to mine time-sensitive patterns since with a tilt-time framework plus some acceptable storage cost, one can maintain a compressed, time-sensitive \( \text{FP-stream} \) structure which facilitates the flexible fading and weighting of old transactions and the discovery of various kinds of time-related patterns. However, this is difficult to be done with the model in \cite{23} since the model is constructed according to a fixed starting point.

The next two subsections outline our method for (1) lazy mining of approximate frequent patterns in data streams, and (2) mining of evolution and changes of frequent patterns in the data stream environment.

4.1 Lazy mining of approximate frequent patterns in data streams

Since there have been substantial studies on efficient mining on the frequent portion of transaction database, our task becomes how to efficiently maintain and incrementally update compressed frequent portion of a transaction database.

In our design, the transactional data streams are compressed and stored using an \( \text{FP-tree} \)-like structure \cite{16} and is being updated incrementally with incoming data items. In \cite{16}, the \( \text{FP-tree} \) is aimed at providing a base structure to facilitate mining in a static batch environment. It is built according to the item frequency which is obtained through a prior scan of the transaction database. As the data continues to stream in, the item frequency is not available a priori and can change drastically over time as we do not assume the stream data to be stationary as in \cite{23}. We need to be able to adjust the compressed data structure dynamically, including deletion and re-insertion of items, based on the incoming data. Although incremental update is performed on the \( \text{FP-tree} \)-like data structures, the mining algorithms to derive frequent patterns from the \( \text{FP-trees} \) are basically the same as the batched data. Thus by building an \( \text{FP-tree} \)-like structure for stream data, we can take advantage of the efficient mining algorithms developed for the \( \text{FP-tree} \) for the batch data \cite{16} to perform lazy mining.

\textbf{Definition 1} The support of an itemset \( p \) in a data stream \( T \) is the the number of transactions in which \( p \) occurs divided by the total number of transactions in \( T \) observed so far.\(^2\) Let the \( \minsup \) be \( \sigma \) and the relaxation ratio be \( p \). An item \( i \) is frequent if its support is no less than \( \sigma \); it is sub-frequent if its support is less than \( \sigma \) but no less than \((1-p)\sigma\); otherwise, it is infrequent. An \( \text{FP-tree} \) consisting of only frequent items is a frequent \( \text{FP-tree} \), and that consisting of only frequent and subfrequent items is a subfrequent \( \text{FP-tree} \).

An itemset \( p \) in a data stream \( T \) is a frequent pattern if \( p \) appears no less frequently than a \( \minsup \) threshold in the flow of data stream observed so far.

First, similar to \cite{16}, the set of subfrequent items of each transaction is inserted as a path, which leads to construction of a subfrequent \( \text{FP-tree} \). With data streaming in, some subfrequent items may become infrequent, or vice versa. Notice that if an item is infrequent, it is dropped from the subfrequent \( \text{FP-tree} \) and its count and its connections with other items will be lost. If it becomes (sub)frequency later, the count not accumulated or already dropped out cannot be collected nor correctly computed. Thus one cannot find precise frequent patterns for such itemsets: only certain approximation is possible. Finally, to find the set of frequent patterns, mining can be performed on the subfrequent \( \text{FP-tree} \) to derive frequent itemsets on the fly.

For mining multi-level, multi-dimensional frequent patterns, the \( \text{FP-tree} \) can be constructed in a manner similar to single-level, single-dimensional \( \text{FP-tree} \) by taking concepts at different levels of abstraction as well as values associated with different dimensions as independent items and being inserted into the corresponding \( \text{FP-tree} \) path. For example, if Wonder bread was sold in a Champaign store on Sat. morning, the item (Wonder bread), the general concept (bread), the time (Sat. morning), and the other dimension values such as location (Champaign), can all be inserted as independent nodes of the \( \text{FP-tree} \) path. Then the concepts associated with multi-level, multi-dimensional information can be discovered automatically in a similar way as that in \cite{26,14}.

\(^2\)The support used here means relative support of an itemset \( p \). One may also use absolute support of \( p \), the number of transactions in which \( p \) occurs in a data stream \( T \).
The methods for construction of FP-tree, the algorithms for mining FP-trees, as well as its various kinds of improvements have been discussed in many papers, such as [16, 22, 28]. Moreover, the method for incremental update of FP-tree can be worked out easily. Therefore, these algorithms will not be presented here.

The critical issues that should be discussed further are (1) what are the rules that an infrequent item should be dropped from the subfrequent FP-tree? and (2) since the subfrequent FP-tree does not register the complete information of potentially frequent items along the time, what could be the approximation that this model may provide?

To simplify our discussion, we assume the rules to drop an item in the FP-tree if it is infrequent with the respect of the total number of unfaded transactions occurring so far.

Suppose a natural tilt time frame model is adopted as shown in Figure 1(a).

Let the total number of transactions accumulated in the previous data stream but not yet faded out be |ODB|, associated with a fading factor \( \phi \) (i.e., each transaction is counted only as a fraction of \( \phi \) transaction). The number of FP-tree-buffered transactions in the current data stream be |CDB|. Also, let \( \text{min sup} = \sigma \) and relaxation ratio be \( \rho \). The maximum occurrence frequency that an item \( \text{p} \) was dropped from ODB is less than \((1 - \rho)\sigma|\text{ODB}|\). Suppose the probability that \( \text{p} \) appears in current dataset CDB is \( \omega \).

To make this item frequent in the overall dataset, we have

\[
\omega|\text{CDB}| + (1 - \rho)\sigma|\text{ODB}| \times \phi \geq \sigma(\phi|\text{ODB}| + |\text{CDB}|)
\]

that is,

\[
(\omega - \sigma)|\text{CDB}| \geq \rho \sigma\phi|\text{ODB}|
\]

or,

\[
\omega - \sigma \geq \rho \phi \left| \frac{\text{ODB}}{|\text{CDB}|} \right|
\]

Let \( \rho = 0.2 \), \( \frac{|\text{ODB}|}{|\text{CDB}|} = 100 \), \( \phi = 0.5 \), and \( \sigma = 0.01 \). We have \( \omega \geq 0.11 \). That is, when the relaxation ratio is 20%, the minimum support is 1%, if an item \( \text{p} \) which is almost subfrequent in ODB may become frequent in overall weighted transaction data set if it appears in the incoming transactions with the relative frequency of 11%.

This implies that if the item \( \text{p} \)'s relative occurrence frequency is not as high as 11%, it is safe to drop this item from the ODB since it will not be frequent in the global transaction database as long as its relative frequency in the current stream database is not as high as 11 times of the minimum support threshold. This calculation shows it is safe (i.e., still mining the related patterns with precise counts) as long as the previous infrequent items not become too frequent in the new stream.

When item \( \text{p} \) in incoming transactions really go beyond this threshold, since we did not register \( \text{p} \)'s frequency in ODB, we cannot correctly predict its overall occurrence frequency. Then how much information about \( \text{p} \) did we really lose in the worst case? Currently, \( \text{p} \)'s frequency is registered as \( \omega|\text{CDB}| \) and the maximal frequency information it has lost is \((1 - \rho)\sigma|\text{ODB}|\). Thus the proportion of the frequency it has lost is at most,

\[
\frac{(1 - \rho)\sigma|\text{ODB}|}{\omega|\text{CDB}| + (1 - \rho)\sigma|\text{ODB}|} = \frac{(1 - \rho)\sigma\phi}{\omega|\text{CDB}| + (1 - \rho)\sigma\phi}
\]

With the given parameters, it is about 78%. That is, if we allow items to be dropped once they become infrequent, there could be as high as 78% of frequency go unregistered if the ratio of the size of the current database vs. that of the old database is only 1%. This could be serious loss of frequent pattern information. Therefore, to reduce such information loss, one should keep current database bigger, or adopt higher relaxation ratio (i.e., register more subfrequent items) as long as the resource is available (unless the old data set fades out quickly).

4.2 Mining time-sensitive frequent patterns in data streams

Since our model has a tilt time window, transaction flows coming at different time window can be registered separately so that time-related frequent patterns can be mined based on an extended FP-tree structure. This is highly desirable in practice. For example, a shopping transaction stream could start long time ago (e.g., a few years ago), and the model constructed by treating all the transactions, old or new, equally cannot be very useful at guiding the current business since some old items may have lost their attraction, and fashion and seasonal products may change from time to time. With time-window information registered in an augmented FP-tree, not only can one fade (e.g., reduce the weight of) old transactions but also find changes or evolution of frequent patterns with time.

Let's examine how to extend our model for mining time-sensitive frequent patterns in data streams.

The tilt time-window augmented FP-tree structure is designed as follows. With the tilt time window, one
can register window-based counts in each FP-tree node. This forms a time-slot-augmented FP-tree structure, with a small overhead on memory. Then one has flexibility to mine a variety of frequent patterns associated with time. For example, one can (1) mine frequent patterns in the current window, (2) mine frequent patterns with the time ranging from any point to any point, with the granularity confined by the specification of window size and boundary, (3) put different weights on different windows to mine various kinds of weighted frequent patterns, and (4) mine evolution of frequent patterns based on the changes of their occurrences in a sequence of windows. Detailed algorithms are left to readers as exercises.

5 Discussion and Conclusions

We have discussed a few important issues and some methodologies for multi-dimensional analysis and mining of data streams. In particular, we have studied two frequently encountered stream data analysis tasks: (1) construction of stream data cubes to find unusual patterns, and (2) mining multi-dimensional frequent patterns in data streams.

From our study and discussion, one can see that it is challenging to mine data streams with limited main memory and computer processing speed. However, with carefully designed methods, it is possible to successfully perform multidimensional analysis and mining of data streams.

Based on our study, we believe that the following design considerations are particularly important,

1. Adopt a tilt time frame when appropriate. Since there is too much data in data stream, it is difficult to keep track of detailed events at every point of time with limited memory and the fast response requirement. A good tradeoff is to adopt a tilt time frame, with more recent events registered more frequently, and remote events registered sparsely.

2. Make good use of the concepts of minimal interesting layer and observation layer. Stream data is often collected in detailed form but users are usually only interested in some higher level, abstracted information. Using the concept of observation layer, most interesting information will be presented at user-desired, high-level abstraction so that patterns can be comprehend easily. Moreover, using the concept of minimal interesting layer, one can generalize the detailed stream data into this layer to provide limited drilling power for on-line multi-dimensional analysis.

3. Perform only partial materialization or preprocessing, based on the available time and space. It is impossible to materialize the whole cube or frequent patterns in the data stream environment. Thus, lazy computation, partial materialization, and memory-conservative preprocessing become interesting methods for preprocessing of data streams to facilitate fast on-line computation. For example, the computation of popular path follows the idea of limited pre-computation and compact storage space for stream cubing. The incremental computation and maintenance of subfrequent FP-tree for frequent pattern mining presents another good example of memory-conserving, limited pre-computation.

This study of multi-dimensional analysis and mining of stream data examines only two tasks. There are a lot of interesting patterns that can be mined from data streams, including classification, clustering, outlier analysis, gradient analysis, evolutionary patterns, periodic patterns, and so on. Systematic studies on multi-dimensional and mining of such patterns and their applications are interesting topics for future research.

References


