

DASPA: A Disk-Aware Stream Processing Architecture

Amol Ghoting and Srinivasan Parthasarathy
The Ohio State University
{ghoting, srini}@cis.ohio-state.edu

Abstract

The past few years have seen the emergence of application domains that need to process data elements arriving as a continuous stream. One key aspect that has been ignored by existing stream processing research is the fact that often there is a direct need to reference past data. In this paper, we present a novel query processing architecture called DASPA, that allows for online data stream processing with low memory requirements. The architecture is well suited to a new breed of data mining applications that need to reference precise representations of past data produced by the data stream. We validate the effectiveness of our architecture on a key data mining approach: frequent itemset mining on data streams.

1 Introduction

Traditionally, database and data mining researchers have primarily concentrated on querying and analyzing *static datasets* that are stored on stable storage systems [26, 30, 31, 32]. However, the past few years have seen the emergence of application domains wherein the need of the hour is the ability to process dynamic datasets [3]. Many such applications are often characterized by data elements arriving in the form of a continuous stream. Examples of such streaming datasets include stock tickers, click streams, data streams generated by scientific simulations and network data. Applications based on such datasets, often place stringent requirements on response times, which in turn requires the ability to process queries or in the case of mining, to discover actionable knowledge, efficiently. Often these data streams can be characterized as *infinite streams* that have no pre-defined size. This requirement has motivated the *online* processing of data streams as and when they arrive and by developing algorithms that bound memory usage. Existing algorithms have been re-designed to process the stream in one pass using a summary structure, which stores an approximate representation of the data stream in memory [10, 12, 33, 34]. A *sliding window* based query

processing scheme evaluates queries based on a recent time window of the data stream coupled with an approximate representation of the past data stream, sacrificing precise representations of past data for processing efficiency.

One key aspect that has been ignored by most existing research in stream processing is the fact that there is often a direct need to reference past data. This element is neither accounted for in recently proposed stream-based architectures nor in recently proposed algorithms for processing and querying stream data. We believe that this feature is particularly essential if one wants to support ad-hoc mining queries. For example, consider the problem of network intrusion detection. In such an application, determining if a transaction is anomalous or not (a likely intrusion), does require access to past network transactions as a benchmark for comparison. Moreover, it may require access to comparable transactions that occurred days, months or weeks ago. Existing stream processing architectures fail to address the needs of the above applications and it is impossible to construct a single summary for all scenarios.

In this paper, we present preliminary work addressing this need. Specifically, a novel query processing architecture built on top of existing architectures used over streaming datasets is proposed. Our proposed architecture enables us to seamlessly integrate past data references, through the use of a specific API targeted at *data streams*. We test the effectiveness of our architecture on a key data mining approach: frequent itemset mining. Frequent itemset mining on streams is non-trivial. A truly online algorithm on streams would require processing each transaction on a stream as it arrives. It also requires processing the stream in bounded amount of memory. As a result, we only track frequencies for the minimal number of itemsets needed and issue a stream from the disk in the event of a newly discovered frequent itemset. This will let us discover new itemsets on the fly. The specific disk-stream we use is a progressive sample of the data stream from the disk that lets us efficiently decide on an appropriate sample size. Finally, our

preliminary design is naturally extensible to work in a distributed setting. More specifically, our contributions are:

- (a) A novel query processing architecture that allows for efficient access to previously processed data. This feature allows us to support *ad-hoc mining queries* requiring access to the history of the data stream, a key advantage of our approach.
- (b) A truly online algorithm for mining frequent itemsets on distributed data streams.

The rest of this paper is organized as follows: Section 2 describes our new query processing architecture. We deal with the problem of mining frequent itemsets on streams in section 3. We empirically evaluate the effectiveness of our architecture and algorithms in section 4. In section 5, we present related data mining work on data streams. Finally, we conclude with directions for future work in section 6.

2 DASPA

Henzinger et al [1] were the first to propose a formal model for data streams. A data stream consists of a potentially unbounded sequence of data elements: $x_1, x_2, \dots, x_i, \dots, x_n$ arriving in increasing order of their indices, independent of their values. They also arrive online and need to be processed in real-time. A more restricted view of data streams has been studied by Tucker et al [2]. In the *punctuated stream* model, each stream is considered to consist of constrained sub-streams, specified through the use of punctuations. Our work is based on a more generic model that encompasses the punctuated stream model.

Manku and Motwani [12] make a distinction between *offline streams* that arrive in the form of regular bulk updates and *online streams* that require processing each element at a time. Offline streams are typical in a data warehouse scenario, where updates need to be applied to a database through the incremental management of a summary structure. Algorithms developed for streams need to operate on online streams to be truly suited to the stream domain. Henceforth, any reference made to a stream will refer to an online data stream. The storage space available when processing data streams is relatively small compared to the size of the data stream. This enforces a one pass processing requirement when working with data streams. Once the data element is processed, it is sent to an archival storage system unless it is explicitly stored in temporary storage. Multiple passes on the content stored in temporary storage are allowed.

Stream Processing Architectures: The past few years have seen the development of several stream

management systems. We define *scalability* for a data stream management system as the ability of the system to manage increasingly larger query loads and stream rates. On the other hand, *adaptivity* for a data stream management system is defined as the ability of system to efficiently manage resources in the face of changing query work loads. Queries running on streaming datasets typically run through the entire life of the stream. Continuous queries [6] are representative of queries running on streams. Continuous queries require concurrent query processing and hence, scalability and adaptivity are considered to be valuable metrics when evaluating stream processing systems. Figure 1 represents the overall architecture of most stream processing engines. The user registers the continuous queries prior to the arrival of the stream. Once the stream begins to arrive, it is processed and some representation of the stream (such as samples, sketches, summaries) [3, 36] is stored in temporary storage. Once the data element is processed, it is forwarded to permanent storage and cannot be referenced again.

The Stanford Stream Data Manager [3, 7] is a general purpose stream manager that will support a declarative SQL style query language. The system scales through the use of queues shared between operators, load shedding, sampling and synopsis compression. The system does not support adaptivity as of yet. Niagara CQ [8] is a stream processing system that has been developed to support queries on XML files streaming at variable rates. The system uses dynamic re-grouping [13], a queue sharing technique, to achieve scalability. Under the condition of changing continuous query workloads, the system adapts through the optimal placement of the selection operator in the query tree. The system also motivated the need for obtaining partial results from a continuous query through the use of synchronization packets inserted into the stream. The Telegraph System [9] is another stream processing system that has been developed based on the Eddy concept [14]. Each tuple carries its lineage and the query is processed by routing the tuples through operators. Only those tuples that satisfy the required lineage make their way through to the output of a query. Shared operators and queues achieve scalability, while optimal routing strategies make the system adaptive. These techniques are equivalent to the group optimization and selection placement techniques developed in the context of Niagara CQ.

Motivation for DASPA: Many streaming applications [4, 5] can process data streams based on algorithms operating on a recent time window of the

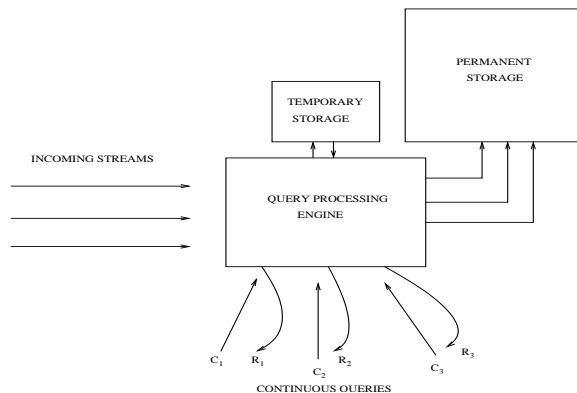


Figure 1: Overall Architecture

stream. This allows processing the data stream in one pass with low memory requirements. These algorithms are categorized under the *sliding window* based query processing scheme.

As motivated in section 1, there exists a class of applications operating on streaming datasets that need to reference past data produced by the same data stream. Bounded memory requirements do not permit effective representation of the stream history within the synopsis/summary structure. The following real world examples expose the limitations of present day query processing architectures for data streams.

Network intrusion detection systems [22] would ideally like to operate on an incoming stream of network transactions and discover anomalies in real-time as opposed to depending solely on pre-defined patterns. Detecting an anomaly on the fly needs access to the history of the transaction stream. For example, we may need to reference the ten previous time windows of network transactions, which can no longer be accurately represented in main memory. One approach would be to maintain an approximate summary for the previous time windows within main memory. However, the fact that the representation is approximate may prevent us from verifying the anomaly. The ideal scenario would involve maintaining an accurate representation of the data stream on disk and fetching it when needed. Scientific simulations have been developed that simulate complex natural processes. One example would be molecular dynamics simulations [23]. These simulations allow discovery and tracking of distinct molecular patterns over time. They produce data at very low time scales and the data generated can reach the Giga-byte range within minutes. Interactive data mining in this setting would require mining this data stream

as it arrives. One possible data mining query could involve detecting similarities and differences between patterns produced by this stream, on the fly. The sheer size of the stream will not allow storing the stream in temporary storage, making past data references a must.

All the above examples need to run in real-time and need to access the past of a data stream. *The crux of the problem lies in the fact that the distant past is no longer accurately represented in temporary storage.* Retrieving the stream from archival storage is a slow and time consuming process. In several cases, however, one can predict, which part of the data stream is needed and when it is needed. Moreover, in many cases, it is possible to characterize an application's data access pattern. As a result, we can do the following to alleviate poor access times associated with archival storage systems:

- (a) Pre-fetch parts of the data on disk that are likely to be used in the future. This will reduce the effective access time for the application.
- (b) Strategically allocate data streams to the disk for efficient retrieval. This can be achieved using several indexing [39] techniques.
- (c) Filter the data stream close to the disk before retrieving it. This will minimize traffic to and from the disk. These issues are specifically addressed in DASPA, described next.

Disk-Aware Stream Processing: Figure 2 represents the query processing architecture of a generic stream processor coupled with our new architecture. A query is constructed using a tree of query operators and the incoming stream is routed through various operators using queues, which can be shared between operators. Tuples that make their way to the root of the query tree constitute the output for a query. When the operator needs to reference a part of the

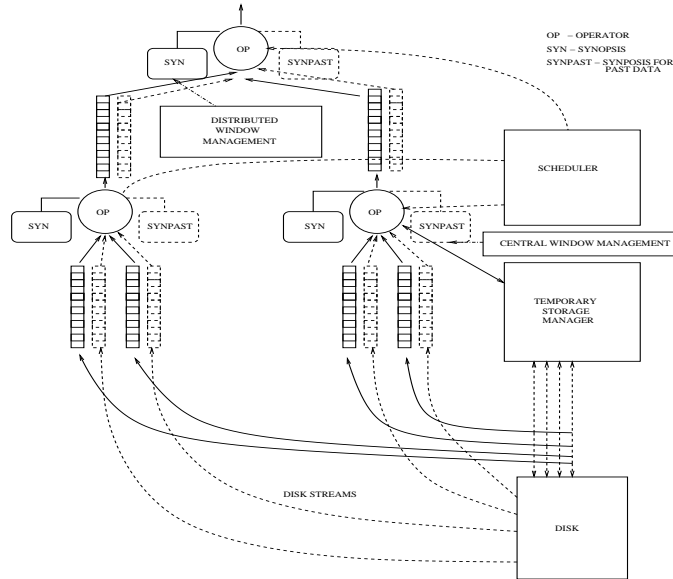


Figure 2: DASPA Query Processing Architecture

stream that has already passed through, it needs to have explicitly stored that portion of the stream in its synopsis or it can reference that part of the stream through a central temporary storage manager. Due to the potential infinite size of streams, the temporary storage manager only buffers a recent portion of the stream. In the *Distributed Window Management* scheme, each operator stores some representation of the stream in its own space through the use of synopses for future reference. On the other hand, *Centralized Window Management* requires that each operator accesses the required data elements through a central temporary storage manager. The Stanford DSMS and Niagara CQ architectures are examples of the distributed window management scheme, while the Telegraph System is an example of the centralized window management scheme.

The Scheduler schedules each operator and results in tuples being routed up the query tree. It has been widely recognized that stream processing systems need to resort to approximation when processing in bounded memory. Most systems achieve approximation through the use of input load shedding, output load shedding and synopsis compression [7], which can be readily incorporated into our model.

The query operators that constitute the query tree for a continuous query now need access to the past or some representation of the stream that can no longer be handled within the synopsis allocated to the query operator. Hence, each operator now needs two more

inputs. Let ip_1 and ip_2 represent the operator inputs associated with the online stream and ip_3 and ip_4 represent the operator inputs associated with the disk-stream. ip_3 and ip_4 manage access to the past of the data stream. Each operator also maintains separate queues and separate synopses for online streams and disk-streams. Let S_{12} and S_{34} represent the synopses for online streams and disk-streams respectively.

Traditionally, once processed, data streams are sent to an archival storage system, which can only be accessed with unpredictable response times. Prior to going to an archival storage system, some representation of the data stream is written to the local disk. This representation can be used to satisfy past data references when requested by the query operators. For example, we store variable sized samples of the data stream on the local disk that we use to satisfy past data references, when mining for frequent itemsets. Segment support maps [38] can be stored on local disks and retrieved when predicting whether an itemset is frequent or not. In the case when we would need access to a certain time span or a range based on other metrics (such as a spatial range), we can use multi-dimensional indexing techniques [39] to manage the history of the data stream. Essentially, we can index/summarize the data generated by the stream for application specific disk-stream references. This disk-stream is not a part of temporary storage allocated to the query operator and is stored on secondary storage. This lets the architecture run

in bounded memory. Each operator has an event e associated with it that can trigger the need to access past data. For example, network intrusion detection systems may detect a possible anomaly and need past data accesses to confirm the anomaly. The event e causes the engine to issue a disk-stream to a subset of the data stream seen so far and possibly even to a partition of the query tree. The lowest level operators in the tree start processing the disk-streams passed on ip_3 and ip_4 and forward their output to ip_3 and ip_4 of their parents in the tree. Over time, the disk-stream completes and the operator converges to a value that is a function of the values maintained for the online streams and the disk-streams. Thus $S_{12} = f(S_{12}, S_{34})$. This architecture seamlessly integrates past data accesses into the query tree and can be incorporated into all the stream processing architectures presented in [7, 8, 9]. It can also be used in a distributed setting [21].

Application Programming Interface: Data mining applications can use our query processing architecture by using our API. An implementation that uses our API needs to specify the following:

- (a) *Data placement for disk-streams:* data streams need to be represented on a local disk for future disk stream references. A simple representation would be a data stream in its raw format. For example, when mining for frequent itemsets, we place $\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \dots$ samples of the data stream on local disks. On the other hand, some applications may need to place a pre-processed representation of the data stream on the local disk. Data on a local disk can also be maintained as a view of the data stored on archival storage.
- (b) *Data placement on an archival storage system:* archival storage systems are characterized by multiple storage servers that may be widely distributed. We can use a middle-ware like DataCutter [40] that can manage data distribution through multi-dimensional range queries. For example, when mining for frequent itemsets, we can distribute data using DataCutter through the specification of range queries on the time dimension.
- (c) *Data extraction for disk-streams:* an application specific event triggers the transmission of the disk-stream from the local disk. Implementations need to specify the event and how they want to process the event. When processing the event, they can choose to issue a disk-stream. Efficient implementations can also choose to stream a subset of the disk-stream through the use of a data filter.
- (d) *Data filters for the disk-stream:* although disk-streams stored on the local disk are processed streams, we may need the option to stream a sub-

set of the disk-stream from local disks. An application specific data filter can handle this. For example, when mining for frequent itemsets, we use a data filter that uses progressive sampling [18] to converge to a sample size. Figure 3 represents our interface.

In the next section we detail how we use our architecture to mine frequent itemsets on data streams.

3 Mining Frequent Itemsets

Mining frequent itemsets serves as an initial step for several data mining tasks such as association rule mining [15], pattern mining [24] and mining for correlations [25]. Making these applications possible on data streams requires efficient generation of frequent itemsets on the same. Sometimes, applications need to discover frequent itemsets across distributed data streams and derive contrasting knowledge from these streams. For example, anomalies in network transactions can be attributed to high contrast frequent itemsets across multiple data streams [37]. These anomalies can lead to a set of network transactions that are prime candidates to be network intrusions.

We define the problem as follows: Let $I = I_1, I_2, \dots, I_m$ denote the universe of items and let T denote the set of transactions. Each transaction $t \in T$ is a binary vector, with $t[k] = 1$ if t includes item I_k and $t[k] = 0$ otherwise. An itemset $X \subseteq I$ is said to have support s if X occurs as a subset in at least fraction s of T . The goal is to find all itemsets X over T for a specified value of s . This problem is further complicated in the stream domain as T is not known in advance and hence, frequent itemsets need to be updated on the arrival of each t . This section describes our technique for mining frequent itemsets on distributed data streams based on the architecture presented in section 2.

Data Layout: Our technique is inspired by the *Eclat* algorithm [19] for frequent itemset mining. In their paper, they propose four techniques based on combinations of equivalence class clustering and maximal hyper-graph cliques and lattice traversals based on bottom-up and hybrid traversals. Eclat is based on equivalence class based clustering and bottom-up traversals. The incoming transactions in a data stream can have two layouts. The horizontal layout [26] views incoming transactions as a unique transaction identifier (*tid*) followed by the items in that transaction. The vertical layout consists of each item in the data stream with transaction identifiers (*tidlist*) of all the transactions that contain the item. We can construct the vertical layout for a data

```

data_placement_disk_stream(Stream S, int Num_buckets, int Distribution, int Epoch_length, void *Custom_handler);
//S - Incoming stream
//Num_buckets - Number of buckets used to sample the data stream
//Distribution - Sampling distribution such as uniform, normal, uniform variable binary
//Custom_handler - Custom handler for data placement
//Epoch_length - Maximum number of disk blocks available for storage

data_placement_archival_storage(Stream S, void *Archival_storage_handler);
//S - Incoming stream
//Archival_storage_handler - Handler to manage storage on an archival storage system.

data_extraction_disk_stream(Stream S, void *Data_filter_disk_stream);
//S - Incoming stream
//Data_filter_disk_stream - Filter function that will filter the disk stream.
//
//      In our case, this will return the correct sample for the data stream
//      using progressive sampling. (Progressive sampling will be addressed in
//      Section 3).

```

Figure 3: API

Step 0: Pick the first available transaction from the distributed data streams.

Step 1: Update the set of frequent two itemsets.

Step 2: Update the set of potential maximal frequent itemsets based on frequent two itemsets.

Step 3: Update the query tree corresponding to the lattice for the potential maximal frequent itemsets.

Step 4: Stream the transaction in the vertical layout (*tidlists*) through the query tree.

Step 4.1: For every operator in the query tree, continue to stream *tids* towards the root of the tree if the itemset corresponding to the operator is frequent.

Step 4.2: If the itemset just turned frequent, *stream an appropriate sample of past transactions as a disk-stream to the operators that are one level higher in the query tree*. Operators that have never received any inputs can now converge to a frequency based on a sample of the past data stream.

Figure 4: Algorithm for frequent itemset mining

stream on the fly. *The vertical layout is particularly amenable to online stream processing, as frequency counts for itemsets can be computed simply by joining the streams of tids for each item that is natural to the stream domain.*

Algorithms:

Algorithm for frequent itemset mining: The algorithm for frequent itemset mining on distributed data streams is presented in Figure 4. Step 1 involves maintaining all the frequent two itemsets. In the Equivalence class based clustering approach, for all $k \geq 2$, we generate the set of potential maximal itemsets from the set of frequent itemsets L_k . We partition L_k into equivalence classes, based on their common $k - 1$ length prefix, given as $[a] = \{b[k]|a[1 : k - 1] = b[1 : k - 1]\}$. In our example presented in Figure 5, we maintain L_2 and the equivalence classes are shown. Each equivalence class, thus, produces a possible maximal frequent itemset. Hence, for [1], 1234 becomes a possible maximal frequent itemset. For $k = 1$ however, we end up with the entire item universe as the maximal frequent itemset. For $k \geq 2$ we begin to extract more precise information as k increases. For higher values of k , we get more precise maximal frequent itemsets at the cost of increased processing when maintaining larger k itemsets. Step 2 generates a set of possible maximal frequent item-

sets using the list of frequent two itemsets. Step 3 uses the set of possible maximal frequent itemsets produced in step 2 to generate a lattice as depicted in Figure 5. The lattice uses *tidlist* intersections to generate frequency counts for itemsets. Corresponding to each lattice, we can generate a query tree, in which every query operator corresponds to an itemset in the lattice. Each operator performs a join operation on incoming *tids* and forwards *tids* that satisfy the join condition. At the end of step 3, the query tree is updated so as to reflect the changes made to the lattice. In step 4, the incoming transaction is streamed through the query tree as a *tid*. However, we do not stream the *tids* through the entire query tree for performance reasons. Step 4 is divided into 2 sub-steps. We stream the *tids* towards the root of the query tree only if the itemset corresponding to the operator is frequent. If the itemset is not frequent, all the itemsets higher up in the query tree that receive an input from the operator will not be frequent and thus, we need not stream the *tid* any further. However, if the itemset just turned frequent, it means that itemsets a level higher in the query tree might have also turned frequent. This corresponds to the event e discussed in section 2. Since, query operators corresponding to itemsets higher up in the tree did not receive *tid* streams previously, we issue a disk-

stream. This allows the itemsets one level higher in the query tree to converge to their frequency counts.

Algorithms for efficiently managing and extracting information from past data: When mining for frequent itemsets, we only maintain frequency counts for the first level of infrequent itemsets. When an itemset turns frequent, there is a possibility that some of the itemsets higher up in the lattice might have also turned frequent. As a result, we now need to stream past transactions so as to get the frequency counts for the $k + 1$ itemsets corresponding to the k itemset that just turned frequent. Lakshmanan et al [38] have proposed an approach based on segment support maps that can be used to get an upper bound on the frequency counts for higher cardinality itemsets, based on exact frequency counts for smaller itemsets, taken over several time windows. This approach can be used as an initial step to prune away higher cardinality itemsets that do not satisfy the minimum required frequency counts. Itemsets that are retained after this initial step will need to receive the stream of past transactions. Unfortunately, streaming the entire transaction history is infeasible due to its sheer size. Also, some past of the history may not agree with the current concept of the process producing the transactions. As outlined previously, we use stream samples as a representation for past data. Storage space on the disk is divided into buckets B_i of size s_i . Each bucket represents one sample and its size s_i is fixed. For each incoming transaction, we place the transaction in a bucket B_i with probability equal to $\frac{s_i}{\sum s_i}$, proportional to the size of the bucket. Active disk based approaches [28, 29] can be used to move the computational overhead associated with sampling closer to the disk.

When the query processing engine requests a disk-stream, one of the samples B_i needs to be streamed as a disk-stream. In order to quickly estimate the sample size, researchers have recently turned to *progressive sampling*. In progressive sampling, we take increasingly larger samples of the dataset until we decide that the current sample serves as an effective representation for the complete dataset. We have proposed a technique [18] to converge to a sample size in the context of frequent itemset mining. We adapt this technique to work on data streams as shown in Figure 6. For step 0, computing the representative set first requires picking the set. In the stream domain, the entire dataset is not known in advance, hence we choose the set based on the history of the data stream. We define equivalence superclass for an item as the set of all frequent itemsets that can be recursively enumerated from a given partition. Thus, for

$[B] = \{BC, BD, BE\}$, we have the equivalence superclass as $ES_B = \{BC, BD, BE, BCD, BDE\}$. We investigate representative sets based on equivalence superclasses for:

- (a) Most frequent items
- (b) Items most frequently co-occurring in frequent itemsets.

Real-time processing requirements require fast representative set computation. Hence, we investigate how using the top few frequent items or frequently co-occurring items can reduce computational overhead associated with the complete representative set. For step 1, convergence is measured using self-similarity between samples. Let A be the set of frequent itemsets chosen using the representative set. For $x \in A$, let $sup_{B_i}(x)$ and $sup_{B_j}(x)$ denote the frequencies of x in samples B_i and B_j . Our metric is:

$$Sim(B_i, B_j) = \frac{\sum_{x \in A} \max\{0, 1 - \alpha |sup_{B_i}(x) - sup_{B_j}(x)|\}}{\|A\|}$$

where α is a scaling parameter. The parameter α has a default value of 1 and can be modified to reflect the significance the user attaches to variations in supports. For $\alpha = 0$, support variance carries no significance. Sim values lie between $[0, 1]$.

4 Empirical Evaluation

In this section, we empirically evaluate the effectiveness of our algorithms and query processing architecture on several datasets. We first describe the experimental setup. We then evaluate the precision and memory utilization for our frequent itemset mining algorithms. We then demonstrate the ability of our architecture to support updates to application parameters at runtime. In the end, we compare the accuracy achieved by using representative set of equivalence superclasses for the most frequently co-occurring items when measuring self-similarity for consecutive samples.

Experimental Setup: We simulated a streaming environment using two real datasets (Gazelle and VBook) and one synthetic dataset (T5.I100.D500K). The Gazelle dataset (59602 transactions) formed a part of the KDD Cup 1999 competition and the VBook dataset (65536 transactions) was obtained from a prominent online book-store retailer in South America. The T5.I100.D500K synthetic dataset was generated using a data generator made by the IBM Quest group. All experiments were performed on a dual Pentium node machine, 1GHz Pentium III, having 512 MB RAM and running Linux 2.4.

Mining Frequent Itemsets on Data Streams: We varied support s from 0.005 to 0.01. We compare our approach to the approach proposed by Manku and Motwani [12] for offline streams. We imple-

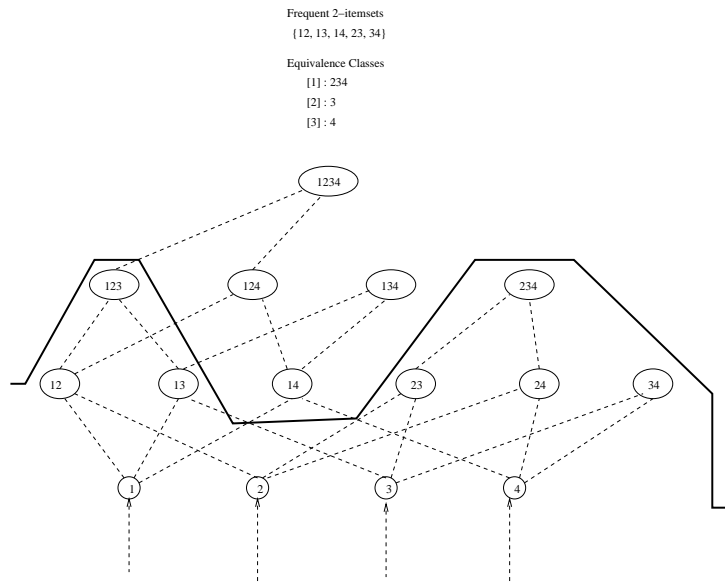


Figure 5: Mining frequent itemsets

For two consecutive samples B_i and B_j on disk
 Step 0: Compute representative set for B_i and B_j
 Step 1: Is Convergence criterion met?
 Step 2: If yes, set effective sample size and break.
 Step 3: If no continue.

Figure 6: Algorithm for Progressive Sampling

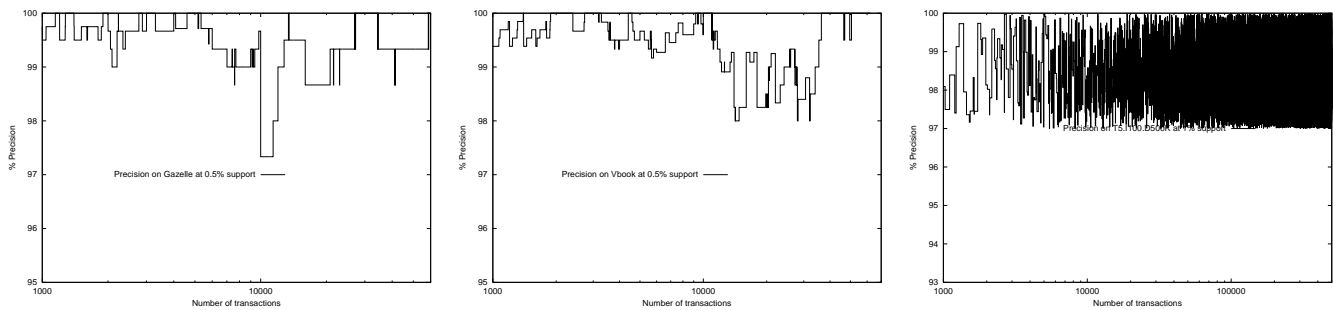


Figure 7: Precision at 0.5 percent support (Gazelle and VBook) and 1 percent support (T5.I100.D500K)

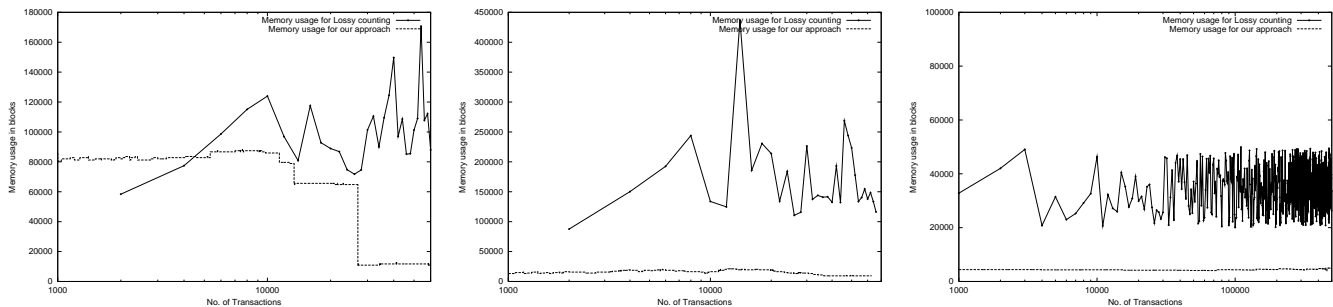


Figure 8: Memory usage at 0.5 percent support (Gazelle and VBook) and 1 percent support (T5.I100.D500K)

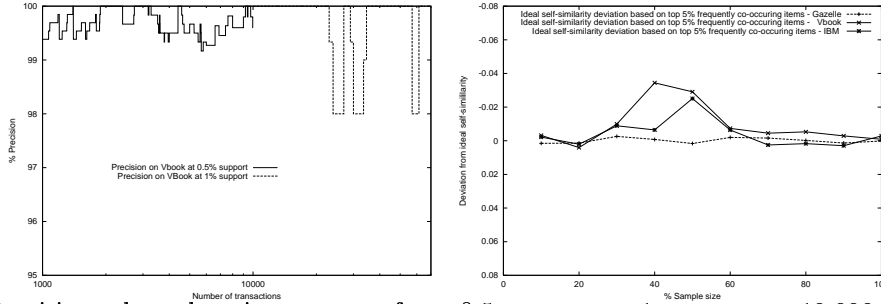


Figure 9: (a) Precision when changing support from 0.5 percent to 1 percent at 10,000 transactions (b) Deviation from ideal self-similarity on Gazelle, Vbook and T5.I100.D500K

mented a version of their algorithm to make this comparison. Manku and Motwani need to buffer a batch of transactions in main memory before they can mine the stream for frequent itemsets. We fix this buffer size to $\frac{1}{0.1s}$. Figure 7 depicts the precision achieved when mining for frequent itemsets on the Gazelle, VBook and T5.I100.D500K datasets at different support values. Precision was determined by comparing our results with the Apriori algorithm [15]. The architecture allows for efficient discovery of new itemsets as shown by converging precision values on all the datasets over different support values. Figure 8 illustrates how our architecture runs in bounded amount of memory. Memory utilization is bounded by the number of first level infrequent and frequent itemsets. We do not need to buffer transactions in memory allowing execution with smaller memory stamps. The buffer size for the Lossy Counting approach increases with decreasing values of support. Memory utilization in our approach is independent of support as majority of the processed data is stored on disk and streamed when needed.

Changing support values at runtime: Our architecture supports changing application parameters at runtime. This is a requirement when processing streams online. Figure 9(a) illustrates how the architecture adapts to a change in the support value when we reach 10000 transactions on the VBook Dataset. The architecture provides for past data accesses in the event of changing support values allowing precise discovery at runtime. This feature lets us support ad-hoc data mining queries on data streams.

Impact of Representative set: We empirically evaluated two different representative sets based on the equivalence superclass for most frequent items and most frequently co-occurring items. Please refer to our paper [18] for self-similarity using representative sets based on most frequent items. The ideal self-similarity curve is the path followed by the representative set for all frequent items in the dataset. Figure 9(b) represents the deviation from the ideal self-

similarity curve. As illustrated, representative sets based on the top few frequently co-occurring items are very effective representative sets for self-similarity estimation. This is evident on all three datasets in which, the top 5 percent of frequently co-occurring items can nearly capture the self-similarity curve for the complete set.

5 Related Work

The first solution to the problem of mining frequent itemsets was proposed by Agrawal and Srikant [15]. The fastest algorithms till date work in two passes. Savasere et al [16] proposed a technique to mine frequent itemsets by partitioning the input into chunks that fit in memory. Toivonen [17] proposed a technique based on sampling. The algorithm computes the frequent itemsets together with the negative border using sampling in the first pass and verifies the validity of the negative border in the second pass. Incremental versions of these algorithms similar to those proposed by Manku and Motwani [12] for offline streams have been proposed. Manku and Motwani proposed a one pass algorithm for mining frequent itemsets on data streams. The approach uses *lossy counting* over streams. Frequent itemsets are generated over transactions in memory and are stored in a summary structure. The stream is then incrementally loaded in memory and the summary structure is updated. The problems associated with this approach are that it suffers from possibly high number of false positives and that frequency counts for itemsets have errors. The algorithm is designed to operate on offline streams and cannot be directly adapted to work on online streams. To the best of our knowledge, our technique is the first technique that addresses the problem of mining frequent itemsets on online streams. In the stream domain, the underlying process producing the stream may change over time, resulting in a concept drift. In the context of data streams, Hulten et al [10] were the first to address the problem of concept drift when mining for decision trees over online streams. Distributed streams

need to be combined in some way so as to derive actionable knowledge from the same. Fourier transforms [11] have been used to reduce the complexity of constructing a decision tree from an ensemble of decision trees, which can be distributed in nature.

6 Conclusion and ongoing work

A key contribution of this paper is a new query processing architecture for data streams. This was motivated in part by new applications with new requirements. The architecture coupled with effective algorithms allows online stream processing with low memory utilization. We evaluated our architecture for a key data mining approach: frequent itemset mining. Experimental results have shown the benefits of our architecture by providing approximate itemset frequency counts over data streams. The architecture is not restricted to providing approximate results and is also capable of producing precise results when needed. Consequently, the architecture can be applied to several other streaming applications and algorithms need to be devised for the same. For instance, several classification and clustering techniques can use our architecture to support past data accesses at decisive points in the algorithm, allowing better precision over data streams.

The current implementation is restricted to run on a single processor. We are currently implementing a distributed version of our architecture. Algorithms using our architecture need to be optimized to efficiently use resources in a distributed setting. However, this extension is not trivial. In the case of frequent itemset mining, the lattice needs to be partitioned across multiple processors. Partitioning should allow good load balancing, while individual query operators need to be scheduled so as to maximize CPU utilization and output rate, while minimizing traffic between multiple processors.

We plan to use this architecture for an online network intrusion detection system. Our algorithms can be extended to mine high contrast frequent itemsets across distributed data streams. A good candidate set of anomalies together with a distributed architecture should permit real-time detection of network intrusions.

References

- [1] M. Henzinger, P. Raghavan and S. Rajagopalan. Computing on datastreams, 1998.
- [2] P. Tucker, D. Maier, T. Sheard, and L. Fegaras. Punctuating Continuous Data Streams, manuscript, 2002. Available at <http://www.cse.ogi.edu/dot/niagara/pstream/punctuating.pdf>
- [3] B. Babcock, S. Babu, M. Datar, R. Motwani and J. Widom. Models and Issues in Data Stream Systems. In Proc. of the 21st ACM Symposium on Principles of Database Systems, 2002.
- [4] Trader Bot <http://www.traderbot.com>
- [5] Yahoo.com <http://www.yahoo.com>
- [6] D. Terry, D. Golderg, D. Nichols and B. Oki. Continuous queries over append-only databases. In Proceedings of the 1992 ACM SIGMOD Intl. Conf. on Management of Data, June 1992.

- [7] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In Proc. of the 2003 Conference on Innovative Data Systems Research (CIDR), January 2003.
- [8] J. Chen, D. DeWitt, F. Tian, Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In Proceedings of the 2000 ACM SIGMOD Conf. on Management of Data.
- [9] S. Madden, M. Shah, J. Hellerstein, V. Raman. Continuously Adaptive Continuous Queries over Streams. In Proceeding of the 2002 ACM SIGMOD Conf. on Management of Data.
- [10] G. Hulten, L. Spencer, P. Domingos. Mining Time-Changing Data Streams. In Proc. of the seventh International Conference on Knowledge Discovery and Data Mining, 2001.
- [11] B. Park, H. Kargupta. Constructing Simpler Decision Trees from the Fourier Spectrum of Ensemble Models: Theoretical Issues and Application in Mining Data Streams.
- [12] G. Manku, R. Motwani. Approximate Frequency Counts over Data Streams. In Proc. of the 28th VLDB Conf., Hong Kong, China, 2002.
- [13] J. Chen, D. DeWitt. Dynamic Re-grouping of continuous queries. In Proc. of the 28th VLDB Conf., Hongkong, China, 2002.
- [14] R. Avnur, J. Hellerstein. Eddies: Continuously Adaptive Query Processing. In Proc. of the 2000 ACM SIGMOD Conf. on Management of Data.
- [15] R. Agrawal, R. Srikant. Fast Algorithms for mining association rules. In Proc. of 20th Int. Conf. on Very Large Data Bases, 1994.
- [16] A. Savasere, E. Omiecinski, S. Navathe. An efficient algorithm for mining association rules in large databases. In Proc. of 21st Intl. Conf. on Very Large Data Bases, 1995.
- [17] H. Toivonen. Sampling large databases for association rules. In Proc. of 22nd Intl. Conf. on Very Large Data Bases, 1996.
- [18] S. Parthasarathy. Efficient Progressive Sampling for Association Mining. In Proc. of the 2002 IEEE Intl. Conf. on Data Mining.
- [19] M. Zaki, S. Parthasarathy, M. Ogihara and W. Li. New Algorithms for Fast Discovery of Association Rules. In Proc. of the 3rd Intl. Conf. on Knowledge Discovery and Data Mining, 1997.
- [20] V. Ganti, J. Gehrke, R. Ramakrishnan. DEMON: Mining and monitoring evolving data. In Proc. of the 16th Intl. Conf. on Data Engineering, 2000.
- [21] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing and S. Zdonik. Scalable Distributed Stream Processing. In Proc. of the First Biennial Conference on Innovative Database Systems (CIDR'03), Asilomar, CA, January 2003.
- [22] D. Barbara and S. Jajodia, editors. Applications of Data Mining in Computer Security. Kluwer, 2002.
- [23] R. Machiraju, S. Parthasarathy, J. Wilkins, D. Thompson, B. Gatlin, D. Richie, T. Choy, M. Jiang, S. Mehta, M. Coatney, and S. Barr. Mining of Complex Evolutionary Phenomena, Next Generation Data Mining, H. Kargupta and A. Joshi, eds, MIT Press (to appear).
- [24] R. Agrawal and R. Srikant. Mining sequential patterns. In ICDE'95.
- [25] S. Brin, R. Motwani and C. Silverstein. Beyond market basket: Generalizing association rules to correlations.
- [26] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen and A. Verkamo. Fast Discovery of Association rules. In Advances in KDD, 1996.
- [27] M. Holsheimer, M. Kersten, H. Mannila and H. Toivonen, A perspective on databases and data mining. In first KDD Conf., 1995.
- [28] A. Acharya et al. Active Disk: Programming model, algorithms and evaluation. In ASPLOS, 1998.
- [29] E. Riedel, G. Gibson and C. Faloutsos, Active storage for large scale data mining and multimedia, in VLDB' 98
- [30] J. A. Hartigan, Clustering Algorithms. John Wiley and Sons, New York, 1975.
- [31] L. Breiman, J. H. Friedman, R. A. Olshen and C. J. Stone. Classification and Regression Trees. Wadsworth, Belmont, CA, 1984.
- [32] M. R. Paterson, Progress in Selection, Dept. of Computer Science, University of Warwick, Coventry, UK, 1997.
- [33] S. Guha, N. Mishra, R. Motwani and L. O'Callaghan. Clustering Data Streams. In Proc. of the Annual Symp. on Foundations of Computer Science (FOCS 2000), November 2000
- [34] G. S. Manku, S. Rajagopalan and B. G. Lindsay. Approximate Medians and other Quantiles in One Pass and with Limited Memory. In Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD 1998), June 1998
- [35] M. J. Zaki, S. Parthasarathy, W. Li and M. Ogihara, Evaluation of Sampling for Data Mining of Association Rules. In Proceedings of the Seventh International Workshop on Research Issues in Data Engineering, Birmingham, UK, April 7-8, 1997.
- [36] P. Flajolet and G. Martin. Probabilistic counting. In Proc. of the 1983 Annual IEEE Symp. on Foundations of Computer Science, 1983.
- [37] S. Bay and M. Pazzani. Detecting change in categorical data: mining contrast sets, in Proceedings of the Fifth ACM International Conference on Knowledge Discovery and Data Mining, New York, NY: ACM Press, 302-305, 1999.
- [38] L. Lakshmanan, C. Leung, and R. Ng. The Segment Support Map: Scalable Mining of Frequent Itemsets. SIGKDD Explorations, Volume 2, Issue 2, Paul Bradley, Usama Fayyad, Sunita Sarawagi, and Kyuseok Shim (Editors), Special Issue on Scalable Data Mining, pages 21-27. December 2000.
- [39] H. Samet, Spatial data structures in Modern Database Systems: The Object Model, Interoperability, and Beyond, W. Kim, Ed., Addison-Wesley/ACM Press, 1995, 361-385.
- [40] M. Beynon, R. Ferreira, T. Kure, A. Sussman and J. Saltz. DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage Systems, Proceedings of the 2000 Mass Storage Conference, College Park, MD, March 27-30, 2000.