Interactive Disk-Aware Data Stream Processing and Mining

Amol Ghoting and Srinivasan Parthasarathy
Department of Computer and Information Science, The Ohio State University
E-mail: ghoting@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. In this work we consider two enhancements to existing stream architectures partially motivated by data mining applications. The first enhancement is based on the observation that in many situations there is often a direct need to reference past data while simultaneously processing incoming streams. To address this need we propose a disk-aware stream processing solution and examine the impact of the proposed solution in the context of mining frequent itemsets on data streams. The second enhancement is based on the observation that data mining is an interactive process and that in such an environment, response time is crucial because lengthy time delay between responses can disturb the flow of human perception and formation of insight. We propose a novel scheduling scheme that explores the trade-offs between response time guarantees, and maintaining optimal memory utilization. The proposed algorithm is kept cognizant of applications characterized by data streams arriving at distributed end points. The architecture allows for distributed data stream processing with response time guarantees. We test the effectiveness of the proposed enhancements on several data stream processing queries and key data mining approaches: frequent itemset mining and clustering, with promising results.

1 Introduction

Traditionally, database and data mining researchers have primarily concentrated on querying and analyzing *static datasets* that are stored on stable storage systems. 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 [1]. Many such applications are often characterized by data elements arriving in the form of a continuous stream (e.g. stock tickers, simulation data, network data). This requirement has motivated *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 [2, 3, 4, 5]. Query processing semantics on data streams have been adapted to follow *sliding window* based schemes that evaluate 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.

Recently several architectures to support such streaming applications have been proposed in the literature [10, 11, 12]. Our claim is that while these architectures may be suitable for query processing, they have limitations when it comes to supporting data mining queries.

First, data mining is an interactive process. Guaranteeing reasonable response times to ad-hoc or continuous queries is essential. This issue of guaranteeing reasonable response times is further complicated over data streams arriving at unpredictable rates. In some instances, not providing a reasonable response time over streams might not allow the user to update an application parameter, thus hindering the knowledge discovery process. Second, there is often a direct need to reference past data produced by the data stream. This element is neither accounted for in recently proposed stream-motivated architectures nor in recently proposed algorithms for processing and querying streaming data. We believe that this feature is particularly essential if one wants to support ad-hoc data mining queries. For example, consider the problem of network intrusion detection. In such an application, determining if a network transaction is anomalous or not (a likely intrusion), requires 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. It is impossible to construct a single summary for all scenarios. Third, many stream-based applications are characterized by streams arriving at distributed end-points. We need to investigate approaches that efficiently utilize distributed computational resources for data stream processing.

To address these limitations, in this article we:

- Propose 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. Our proposed architecture enables us to seamlessly integrate past data references, through the use of a specific API targeted at data streams. This gives the end users the ability to explore precision and response time trade-offs over data streams with a reasonable guarantee on response time.

- Propose an adaptive scheduling technique over data streams. The user can tune the application to the desired level of interactivity, thus facilitating the data mining process. We achieve this through a step-wise degradation in response time beginning from a schedule that will be optimal in terms of response time and using this sacrifice in response time towards optimal memory utilization. We also extend our scheduling technique for a distributed setting.

- Describe online algorithms for mining frequent itemsets and clusters over distributed data streams using our architectural enhancements. 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. Clustering data streams has been studied previously [4], and we extend the algorithm to operate on streams arriving at distributed endpoints at variable rates, while providing the user the desired level of response time.

- Evaluate the proposed architecture and algorithms extensively on real and synthetic datasets.

The rest of this paper is organized as follows: Section 2 briefly presents the stream domain together with a decomposition of stream processing applications. We present our scheduling scheme in section

3. Section 4 describes disk-aware stream processing. Algorithms for mining streams for clusters and frequent itemsets are covered in section 5. We empirically evaluate the effectiveness of our architecture and algorithms in section 6. Finally, we conclude with directions for future work in section 7.

2 Stream Processing Model and Assumptions

Background: Henzinger *et al* [6] 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,..., x_i,..., 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* [7]. 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 [3] 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. Algorithms developed for streams need to operate on online streams to be truly suited to the stream domain. Hence-forth, 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. The available storage space is used towards a stream summary and multiple passes on the summary are allowed.

Stream Processing Model: Our study of popular data stream processing and mining algorithms has shown that their operations share the following canonical form. Let F be primitive operation on data streams. Let $I = (i_1, i_2, ..., i_n)$ represent n incoming streams. Let T represent a global summary maintained for F. Let $O = (o_1, o_2, ..., o_m)$ represent m output streams produced by the operation. The state prior to the the execution of F can be represented by a $(k + m + 1)$ tuple (I, T, O). Let $S = (I, T, O)$ be the state prior to the application of F and $S' = (I', T', O')$ be the state after the application of F. F can be represented by the following three sub-operations, F_1, F_2, F_3 applied in order as follows: $T' = F_1\left(I, T\right), O' = F_2\left(I, T', O\right)$ and $I' = F_3\left(I, T'\right)$. F_1 updates the summary structure to create T' based on the incoming streams, F_2 may append a value to one or more of the output streams to create O' and F_3 removes an element from at least one of the incoming streams to create I'. A primitive operator $OP = (F, I, T, O)$ over data streams executes the primitive operation F in a loop.
$OP \equiv loop\left\{S' = F\left(S\right), S = S'\right\}$
Stream processing and mining algorithms can be represented as a *directed acyclic graph (DAG)* $G = (V, E)$, where $V = (V_1, V_2, ..., V_i)$ are graph nodes of type OP and $E = (E_1, E_2, ..., E_j)$ are the edges in the graph such that if $E_k = \left(V_a\left(F, I, T, O\right), V_b\left(F', I', T', O'\right)\right)$ then $\exists i_c \in I', o_d \in O : i_c = o_d$. An application such as frequent itemset mining can be decomposed into a DAG in which, each individual node is an operator corresponding to an itemset in the frequent itemset lattice that repeatedly performs a join operation on transactions identifiers being streamed in from operators corresponding to its subset itemsets. Each operator in-turn streams out transaction identifiers to operators corresponding to its superset itemsets. In a similar fashion, a distributed clustering implementation can be decomposed into a hierarchy of operators in which, the leaf operators cluster a subset of the stream and pass on the discovered cluster to an operator higher up in the hierarchy that re-clusters the clusters generated by the leaf operators. The operator at the root of the hierarchy would report the final clusters. Similarly, it is easy to see how stream processing systems such as the Stanford DSMS [10], Niagara CQ [11] and

Telegraph System [12] would directly fall under the above decomposition.

3 Scheduling

In this section we present our scheduling technique that schedules data stream processing and mining applications at the abstraction of a DAG of primitive operators. Every tuple that enters a system needs to pass through a unique set of operators. Let us call this the *operator path* for the tuple. To illustrate the need for adaptive scheduling, consider a simple operator path that consists of two operators: O_1 followed by O_2. Assume that O_1 takes unit time to process a tuple and its selectivity is 0.2. Further, assume that O_2 takes 5 units of time to process a tuple and its selectivity is 1. Thus, on the average it takes 2 time units to process one tuple. If we assume that the average arrival rate for tuples is no more than 1 tuple per 2 units of time then we do not have an unbounded build-up of tuples over time. However, the arrival of tuples could be bursty. Consider the following arrival pattern: A tuple arrives at every time instant from t = 0 to t = 6, then no tuples arrive from time t = 7 through t = 13. Now consider the following two scheduling strategies:

- FIFO scheduling: Tuples are processed in the order that they arrive. A tuple is passed through both operators in two consecutive units of time, during which time no other tuple is processed
- Greedy scheduling: At any time instant, if there is a tuple that is buffered before O_1, then it is operated on using 1 time unit; otherwise, if tuples are buffered before O_2, they are operated on using 5 time unit each.

The following table shows the total size of input queues and resulting output tuple response time for the two strategies:

Time	Greedy scheduling	FIFO scheduling	Output tuple
0	1	1	-
1	1.2	1.2	-
2	1.4	2.0	t_0
3	1.6	2.2	-
4	1.8	3.0	t_1
5	2.0	3.2	-
6	2.2	4.0	t_2

After time t = 6, input queue sizes for both strategies decline until they reach 0 after time t = 13. Observe that Greedy scheduling has smaller maximum memory requirement than FIFO scheduling. In fact, if the memory threshold is set to 3, then FIFO scheduling becomes infeasible, while Greedy scheduling does not. On the other hand, FIFO scheduling produces output tuples in the presence of the burst, which may be a requirement, if the application needs to support a high degree of interactivity. Greedy scheduling on the other hand does not produce any output tuples through the burst. If we need to support interactivity, what we need is a robust scheduling strategy that can handle variations in stream rates and at the same time, provide a guarantee on the degree of interactivity, while providing optimal memory utilization for that particular degree of interactivity. This example serves to motivate the rest of this section.

3.1 The State of Primitive Operators

In this section we describe the state of a stream processing application represented as a DAG, $G = (V, E)$. Let $V = (V_1, V_2, V_3, ..., V_n)$ be n primitive operators that need to be scheduled. Let $\sigma\,(V_i)$ be the

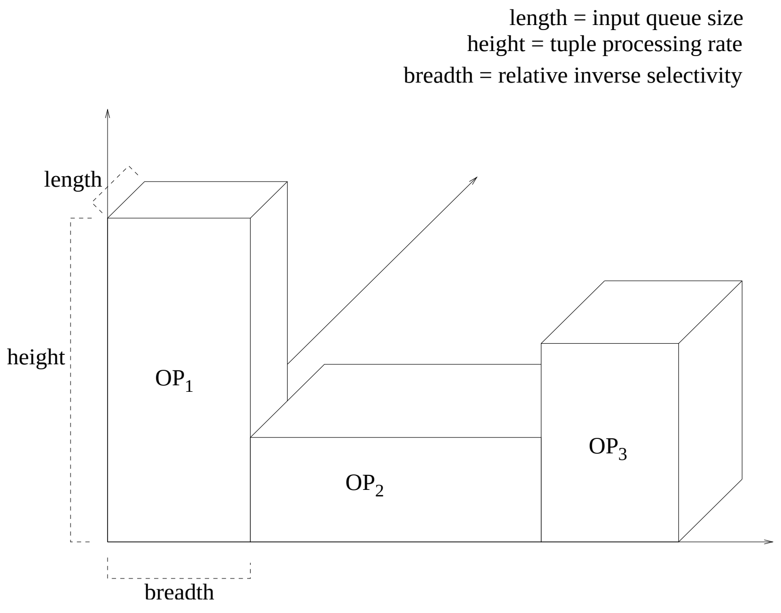

length = input queue size
height = tuple processing rate
breadth = relative inverse selectivity
length
height
breadth
OP$_1$
OP$_2$
OP$_3$

Figure 1. Representation for operator and engine state

selectivity for the operator V_i. $\sigma\left(V_i\right) = \frac{\text{total tuples placed on } O_i}{\text{total tuples received on } I_i}$. Let $\tau\left(V_i\right)$ be the tuple processing rate for the operator V_i. $\tau\left(V_i\right) = \frac{\text{total number of tuples processed on } I_i}{\text{total time spent processing the tuples}}$. Let $\iota\left(V_i\right)$ represent the number of tuples waiting to be processed on I_i. Let $TOT\left(G\right)$ be the total amount of memory used by the stream processing engine. Then $TOT\left(G\right) = \sum_{i=1}^{n} \iota\left(V_i\right)$. We assume that the size of the operator summaries is a constant and hence for simplicity, we only take the input queue sizes into account. We can capture the state of each individual operator and the entire stream processing engine through a three dimensional bar chart. The breadth of an operator in the chart is given by $breadth\left(OP_i\right) = \frac{\frac{1}{\sigma(V_i)}}{\sum_{i=1}^{n}\left(\frac{1}{\sigma(V_i)}\right)}$, the height is given by $height\left(OP_i\right) = \tau\left(V_i\right)$ and the length is given by $length\left(OP_i\right) = \iota\left(Vi\right)$.

Definition 1 *The state of an operator V_i changes if $\frac{\tau(V_i)}{\sigma(V_i)}$ changes.*

3.2 Response Time Optimal and Memory Optimal Schedules

Definition 2 *A response time optimal schedule is an operator schedule that for every incoming tuple, routes it to the root of the operator path in the minimum time possible.*

Lemma 1 *A schedule that schedules incoming tuples in a FIFO order is a response time optimal schedule*

In the case of a FIFO schedule, each incoming tuple is routed through its operator path till it reaches its farthest point and then the next incoming tuple is considered. If FIFO was not a response time optimal schedule, then some response time optimal schedule would have to process a tuple in the operator path, while another tuple higher up in the path would also be to ready to be processed. This would delay the latter tuple by atleast one time step. This contradicts the definition of a FIFO schedule.

Definition 3 *A memory optimal schedule is an operator schedule that for every time step keeps $TOT\,(G)$, the memory required by the stream processing engine to the minimum.*

Lemma 2 *A greedy schedule that schedules the operator that will process and eliminate the maximum amount of memory per unit time is a memory optimal schedule.*

As it is not possible to predict future memory utilization for the engine, a schedule that would process and eliminate the maximum amount of memory per unit time based on the previous behavior of the operators will be considered to be memory optimal. A greedy scheduling technique schedules the operator that has the largest value for $\frac{\tau(V_i)\iota(V_i)}{\sigma(V_i)}$ at any given instant of time, which will be memory optimal. This operator corresponds to the bar with the largest volume in the bar chart presented in Figure 1. The volume represents the rate at which the operator processes and eliminates the memory used by its queue and hence scheduling the operator with the largest volume will me memory optimal for that time step. Let $VOLMAX$ and $VOLMAX_2$ be the largest and the second largest volumes in the bar chart. A greedy schedule will continue to be memory optimal for $\frac{(VOLMAX-VOLMAX_2)\sigma(V_{max})}{\iota(V_{max})\tau(V_{max})}$ time steps as long as the state of any of the operators in the engine does not change. At any instant of time however, scheduling the active operator with the largest value for $\frac{\tau(V_i)}{\sigma(V_i)}$ will be memory optimal for that time step.
Related work: In the context of data streams, Babcock *et al* [14] present Chain operator scheduling for near optimal continuous query memory minimization in the presence of a burst in the stream. Our work bears complementary goals with respect to Chain operator scheduling. Our scheduling techniques work toward providing response time guarantees, while a technique such as Chain operator scheduling can be used to deal with scheduling overhead in the general case.

3.3 A Ticket-based Scheduling Scheme

Having defined response time optimal and memory optimal schedules, we propose to schedule the operators so as to give a guarantee on response times with respect to a response time optimal schedule, while taking memory optimal decisions whenever possible.

- Let $t_i = t\,(V_i)$ be a new attribute for the number of tickets assigned to each node V_i in the graph.
- Let $D = (t_1, t_2, ..., t_n)$ be a non zero number of tickets assigned to V.
- The scheduler for the engine will schedule the operator with the most number of tickets at any given instant of time. Tickets establish an inherent priority between the operators. In case of a tie in the number of tickets, we can use the index of the operator to break the tie. The index is proportional to the distance from the root of the graph.
- The ticketing problem is to make the assignment D such that the scheduler will schedule the operators following the desired schedule.

Definition 4 *We define a $RTOS - k$ schedule for a state of G as a schedule for operators that produce output tuples, none of which are delayed by more than k time steps, when compared to the response time optimal schedule, where a time step $= max\left(\frac{1}{\tau(V_i)}\right)$, and at the same time provides the best memory utilization possible.*

```
Algorithm K-ASSIGN to find a ticket distribution for $RTOS - k$ schedule defined on a state
$D_{RTOS}(V)$ = response time optimal ticket distribution $D_{MOS}(V)$ = memory optimal ticket distribution
$D_{RTOS-h}(V) = RTOS - h$ schedule ticket distribution

Algorithm $K - ASSIGN(h)$ :
while ($h$ tickets have not been assigned)
operator index for the $a^{th}$ assignment
$= i : SSD_i(D_{RTOS-a-1}, D_{MOS})$
$= min\left(^{n}_{j=1} SSD_j(D_{RTOS-a-1}, D_{MOS})\right)$
assign a ticket to the operator
a = a + 1
end while
$SSD_j(D_{RTOS-a-1}, D_{MOS}) = \sum_{i=1}^{n} f(i)$ and
$f(i) = \left( \begin{array}{ll} (D_{RTOS-a-1}(i) - D_{MOS}(i))^2 & \text{if } i \neq j \\ (D_{RTOS-a-1}(i) + 1 - D_{MOS}(i))^2 & \text{if } i = j \end{array} \right)$
```

Figure 2. Algorithm K-ASSIGN

Theorem 1 *A ticket assignment that assigns each operator equal number of tickets is $RTOS - 0$ schedule or a FIFO schedule*

Proof sketch: A ticket assignment D and an index i establishes a total order for V. At any point in time, only a single active operator will be scheduled for a single time-step, followed by the next active operator with the maximum number of tickets, closest to the root, based on its index, which is a FIFO pattern. If an equal ticket distribution was not response time optimal, there must exist an operator that was scheduled out of order, when compared to the schedule followed by the assignment D, for the RTOS schedule. If something was scheduled out of order compared to the FIFO schedule, that would imply that there was an active operator in the tree that was closer to the root of the tree and was not scheduled, which cannot be a response time optimal schedule.

Theorem 2 *A ticket assignment with tickets proportional to $\frac{\tau(V_i)\iota(V_i)}{\sigma(V_i)}$ is a memory optimal schedule*

When tickets are assigned proportional to $\frac{\tau(V_i)\iota(V_i)}{\sigma(V_i)}$, at every instant of time, we schedule the operator with optimal memory usage making it a memory optimal for time steps $= \frac{(VOLMAX - VOLMAX_2)\sigma(V_{max})}{\iota(V_{max})\tau(V_{max})}$, as long as the state of the operators does not change. When tickets are assigned proportional to $\frac{\tau(V_i)}{\sigma(V_i)}$, scheduling the active operator is a memory optimal schedule for that time step.

Algorithm K-ASSIGN: Figure 2 presents an algorithm to distribute k tickets for an $RTOS - k$ schedule. K-ASSIGN assigns k tickets so that the resulting response times are not delayed by more than k time steps. In the i^{th} iteration, the algorithm assigns a ticket to that operator that minimizes the squared distance between the $RTOS - i - 1$ ticket assignment and the memory optimal assignment. Each ticket assignment is used towards scheduling an operator that would be memory optimal, resulting in a possible delay in the response time by a maximum of one time step.

Theorem 3 *Algorithm K-ASSIGN (Figure 2) gives us a $RTOS - k$ schedule which is memory optimal*

Let us consider the $RTOS - 1$ ticket assignment. We minimize the squared distances between the response time optimal ticket assignment and the memory optimal ticket distribution. This guarantees to schedule an operator for at most one time step $= max\left(\frac{1}{\tau(V_i)}\right)$ and at the same time provide optimal memory utilization. Algorithm K-ASSIGN presents optimal sub-structure[1]. The i^{th} assignment is obtained from the $(i-1)^{th}$ assignment. The i^{th} assignment will not be a memory optimal assignment if and only if the $(i-1)^{th}$ assignment was not memory optimal. At each stage we take a memory optimal decision while increasing response time by at most one time step. Thus, an $RTOS - k$ schedule will be the memory optimal ticket distribution such that the output tuples are not delayed by more than k time steps from the $RTOS$ schedule. Algorithm K-ASSIGN takes $O\left(kn^2\right)$ time to assign k tickets to n operators.

Updating the $RTOS - k$ schedule on a change in state: As described previously, an $RTOS - k$ schedule is defined for a particular state of the engine. Over time however, the ticket assignment is likely to change because of a change in engine state. We can encounter changes in selectivities and tuple processing rates making previous ticket assignments invalid. Let T_{MOS} be the MOS ticket assignment sorted by ticket values in descending order, totally ordered by indices. Let T'_{MOS} be the new MOS ticket assignment due to change in state. We consider how to determine if the change in state affects the $RTOS - k$ ticket distribution.

Theorem 4 *The largest value of k for which, the $RTOS - k$ schedule can be affected by the c^{th} ticket assignment in T_{MOS} is given by $k = \sum_{i=1}^{c-1} i\left(T_{MOS}\left(i\right) - T_{MOS}\left(i+1\right)\right)$*

The proof by induction is trivial. Let k' correspond to the largest $RTOS - k'$ schedule to be affected by a change in the c^{th} ticket assignment change in T_{MOS}. If $k < k'$ then there will be no change to the ticket assignment. If $k \geq k'$ there should be a change in the ticket assignment, which will contribute towards an error in the response time. The maximum error δ introduced in response time is given by $\delta = |k - k'|max\left(\frac{1}{\tau(V_i)}\right)$. If the new MOS ticket assignment for V_i increases and affects the $RTOS - k$ schedule as well, it implies in an increase in the $RTOS - k$ ticket assignment for V_i and possible decrease in the ticket assignment for operators $(V_1, \ldots, V_{i-1})$ in T_{MOS}. On the other hand, if the new MOS ticket assignment for operator V_i decreases and affects the $RTOS - k$ as well, it implies in a decrease in the $RTOS - k$ ticket assignment for V_i and possible increase in the ticket assignment for operators $(V_1, \ldots, V_{i-1})$ in T'_{MOS}.

Scheduling operators in a distributed setting: Scheduling primitive operators on distributed memory nodes requires a cost model for operator placement on nodes. We incorporate this cost into the tuple processing time for a primitive operator. The time it takes a primitive operator to process a tuple is now the the time it takes to get the input tuple from the output queue of its child operator (possibly over a network) and the time it would normally take to process the tuple.

Definition 5 *We define a distributed allocation for $RTOS - k$ schedule of operators as a response time optimal placement of the operators with maximal number of concurrent operators in the system.*

Theorem 5 *Algorithm DISTRIBUTED - K - ASSIGN - ALLOCATE (Figure 3) is an optimal distributed allocation for the $RTOS - k$ schedule.*

[1]An optimal solution contains within it optimal solutions to sub-problems

```
Algorithm: DISTRIBUTED - K - ASSIGN - ALLOCATE
while operators are left in the graph
    Pick the leaf operator with the least number of tickets
    Place it on the node that minimizes the response time for output tuples
    In case of a tie, assign it to the node with the least number of tickets
end while
```

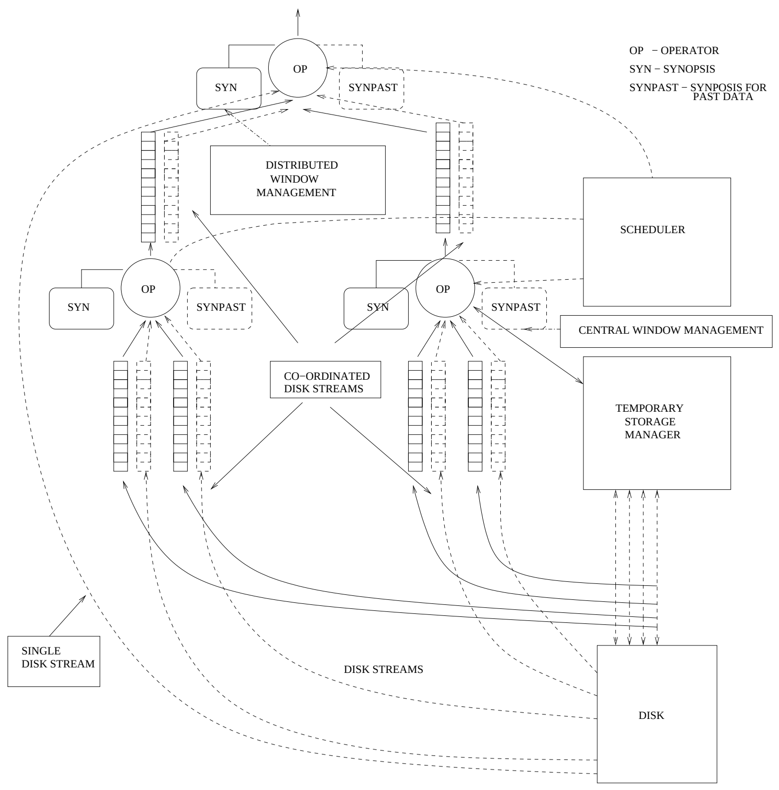
Figure 3. Algorithm to allocate operators in a distributed setting

We first show that the allocation provides a response time optimal schedule. The problem of scheduling a DAG on multiple nodes presents optimal sub-structure. The response time for a tuple at an operator will be optimal (minimal) only if it receives the tuple from its child in the operator graph in an optimal manner. This is similar to affinity-based scheduling [24]. Thus by placing each leaf operator on a node that minimizes response time for the operator, we get a response time optimal schedule. When we take the number of tickets into account, we establish a total order between all the leaf operators. Picking the leaf operator with the least number of tickets lets us establish an order between all leaf operators, while placing the operator on the node with the least number of tickets guarantees it the best chance of being executed. For contradiction purposes, let us assume there exists another response time optimal operator assignment to a node with a larger number of tickets. A larger number of tickets implies that you will have to share the node with other operators, while another node with a fewer number of tickets will remain vacant. This would not maximize concurrency.

Updating the distributed ticket assignment locally and its implications on the global optimal ticket assignment: We assume that each node in the distributed setting possesses the memory optimal ticket assignment T_{MOS}. In this setting, the bounds on response time in the centralized case continue to hold in the distributed case.

4 Disk-Aware Data Stream Processing

Motivation: Many streaming applications can process data streams based on algorithms operating on a recent time window of the stream. This allows processing the data stream in one pass with low memory requirements. The following real world examples serve to expose the limitations of present day stream processing architectures. Network intrusion detection systems [15] 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. 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 [16]. These simulations allow discovery and tracking of distinct molecular patterns over time and produce data at very low time scales. Interactive data mining in this setting would require mining this data stream as it arrives. The sheer size of the stream will not allow storing the stream in the summary, making past data references a must. All the above examples need to run in *soft* 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 the summary*.



OP – OPERATOR
SYN – SYNOPSIS
SYNPAST – SYNPOSIS FOR PAST DATA
OP
SYN
SYNPAST
DISTRIBUTED WINDOW MANAGEMENT
SCHEDULER
CENTRAL WINDOW MANAGEMENT
CO–ORDINATED DISK STREAMS
TEMPORARY STORAGE MANAGER
SINGLE DISK STREAM
DISK STREAMS
DISK

Figure 4. Disk-Aware Stream Processing

4.1 Disk-Aware Stream Processing

Figure 4 represents the query processing architecture of a generic stream processor coupled with our new architecture. A query is constructed using a DAG of 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 graph constitute the output for a query. When the operator needs to reference a part of the 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 that buffers a recent portion of the stream. The Scheduler schedules each operator, one at a time, 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 [10], which can be readily incorporated into our model.

The operators that constitute the application 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 additional inputs. Let $I = (i_1, i_2, ..., i_n, id_1, id_2, ..., id_n)$. We add n additional input streams to n streams present previously. This lets each operator accept an additional stream per input stream that will be streamed in from the disk, termed as *disk-streams*. Each operator also maintains a separate synopses T_d for disk-streams. 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 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. 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 operator and is stored on secondary storage. This lets the application 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 operator graph. Disk-streams have two primary variants.

- A single disk-stream is directly accessed from disk by the operator that needs access. The primitive function F is given by $T' = F_1\left(I_{1..n}, T\right), T'_d = F_1\left(I_{n+1..2n}, T_d, e\right), O' = F_2\left(I, T', T'_d, O\right)$ and $I' = F_3\left(I, T', T'_d\right)$. If the event e is triggered, the operator sets the values in O' as a function of its input streams and disk-streams, else it proceeds without any disk accesses.

- A co-ordinated disk-stream requires that a group of operators process the stream before it can be utilized. Let $O = (o_1, o_2, ..., o_m, od_1, od_2, ..., od_m)$ represent m output streams and m additional output disk-streams. Co-ordination between operators can be represented using a co-ordination graph specifying the order in which the operators need to process disk-streams. Here F is given by $T' = F_1\left(I_{1..n}, T\right), T'_d = F_1\left(I_{n+1..2n}, T_d, e\right), O'_{1..m} = F_2\left(I_{1..n}, T', O\right), O'_{m+1..2m} = F_2\left(I_{n+1..2n}, T'_d, O\right)$ and $I' = F_3\left(I, T', T'_d\right)$. The leaf operators in the co-ordination graph start processing the disk-streams passed on $id_1, id_2, ..., id_n$ and forward their outputs to $od_1, od_2, ..., od_m$. Over time, the disk-stream completes and the operator that fired e converges to a value that is a function of the values maintained for the online streams and the disk-streams.

This architecture seamlessly integrates past data accesses into the query tree and can be incorporated into all the stream processing architectures presented in [10, 11, 12].

Application Programming Interface: 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. Indexing [18] techniques or smart placement [25] may be used here. (c) Filter the data stream close to the disk before retrieving it. This will minimize traffic to and from the disk. Our API (Figure 5) specifically supports these issues.

5 Data mining operators

5.1 Frequent Itemset Mining using Disk-aware techniques

Mining frequent itemsets serves as an initial step for several data mining tasks such as association rule mining [13], pattern mining [19] and mining for correlations [20]. 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

```
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
  //the next section).
```

Figure 5. API

itemsets across multiple data streams [21]. We define the problem are follows: Let $I = I_1, I_2, ..., I_m$ 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: The incoming transactions in a data stream can have two layouts. The horizontal layout [22] 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 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 (Figure 7) that is natural to the stream domain.*

Algorithm for frequent itemset mining: The algorithm for frequent itemset mining on distributed data streams is presented in Figure 6 and is inspired by Eclat [22]. 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 7, 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 (Note: For $k = 1$ however, we end up with the entire item universe as the maximal frequent itemset). 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 itemsets 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 7. The lattice uses $tidlist$

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 ($tidlist$s) 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 6. Algorithm for frequent itemset mining

intersections to generate frequency counts for itemsets. Corresponding to each lattice, we can generate a query tree, in which each query operator corresponds to an itemset in the lattice and repeatedly performs a join. 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. 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 the previous section. 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. Segment support maps [17] can be used to get an upper bound on the frequency counts for these higher cardinality itemsets in an initial pruning step. Unfortunately, streaming the entire transaction history is infeasible. Also, some past of the history may not agree with the current concept of the process producing the transactions. Generating a random sample from disk is expensive. For a stream of size N and disk block size b, generating a random sample of size n takes $O\left(\frac{N}{b}\right)$. We adopt a smart placement approach to store streaming transactions. We divide the N entries into a fixed number of bins (say 6). Let us say each bin contains $N/2, N/4, N/8, N/16, N/32$ and $N/32$ entries respectively. Note, that after each set of N transactions we can repeat this process. As each record is processed it is placed in one of these seven bins at random. Then to compute a sample of size n one could use the appropriate bin(s), thereby avoiding a complete disk scan and taking only $O\left(\frac{n}{b}\right)$.

When the query processing engine requests a disk-stream, a sample 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

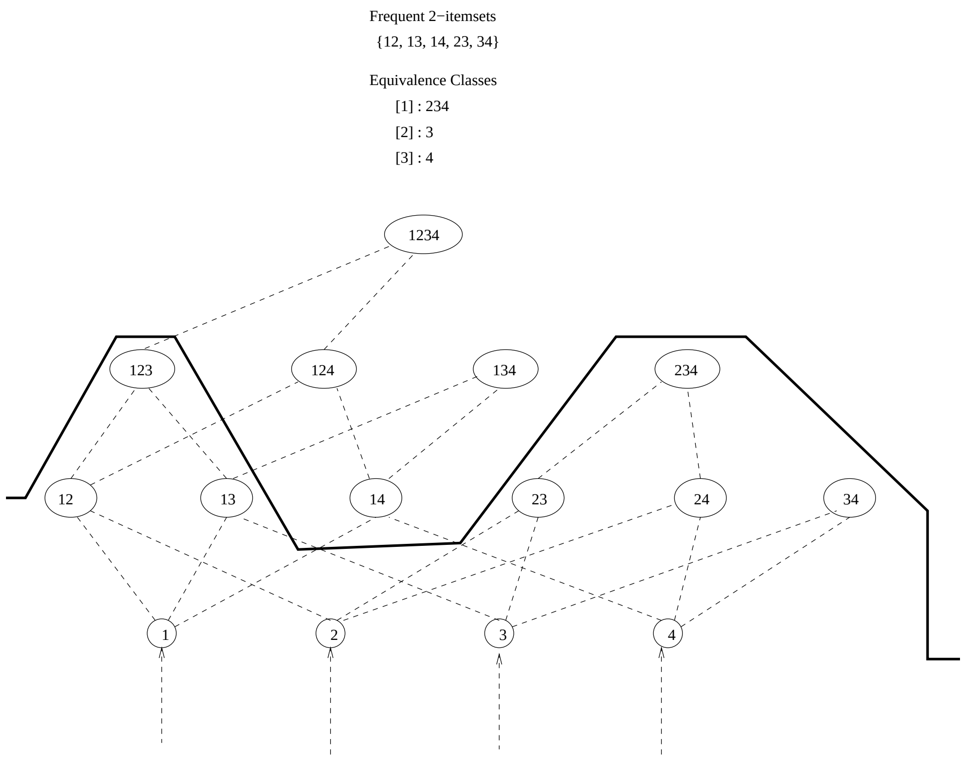


Figure 7. Mining frequent itemsets

that the current sample serves as an effective representation for the complete dataset. We have proposed a technique [23] 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 8. For step 0, computing the representative set first

```
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 8. Algorithm for Progressive Sampling

requires picking the set. 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 $ES_B = \{BC, BD, BE, BCD, BDE\}$. Real-time processing requirements require fast representative set computation. Hence, we use equivalence superclasses corresponding to the top few frequently co-occurring items as representative sets. 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 $[0, 1]$.

Related work: Manku and Motwani [3] present a one pass algorithm for mining frequent itemsets on data streams with strong error bounds. Our approach for frequent itemset mining allows us to change support parameters at runtime and operate with smaller memory stamps.

5.2 Clustering Distributed Data Streams

Clustering has been studied across several disciplines. A common formulation of the clustering problem is the k-Median problem: find k centers in a set of n points so as to minimize the sum of distances from data points to their closest centers. Guha *et al* [4] present a constant factor approximation algorithm for k-Median clustering of data streams. We extend their work to operate on clustering distributed data streams arriving at variable rates. The goal is to report k-medians at regular intervals of time. Consider the following decomposition to the clustering problem over data streams. Let OP_{mn} be a clustering operator with a single input queue and output queue. The operator performs the following functions: (a) Waits for n input tuples on the input queue (b) Clusters the n tuples to create m weighted k-Medians. (Note: This gives the operator a selectivity of $\frac{m}{n}$). Let us assume for now that the streams arrive at a constant rate. We can construct a tree of OP_{mn} such that each leaf operator waits for n tuples and clusters them to create $O\left(k\right)$ (say $2k$) weighted clusters, which are sent to the clustering operator at the next level. Once we have n weighted clusters at the next level in the tree, we can cluster these n clusters to get the final k clusters. The goal is to report k clusters at the arrival of every n tuples at the input queue of the root of the tree. In this construction, each leaf operator will have constant selectivity. However, if the streams are arriving at variable rates, for a fixed value of m, we can set the value of n for OP_{mn} proportional to the rate of the incoming stream. Higher the influx rate of the incoming stream, larger will be the value for n. In this scenario, we have an approximation factor of $2c\left(1+2b\right)$ [4], where the leaf operator with the highest selectivity provides us with a constant factor approximation of b.

6 Experiments

In this section we will empirically evaluate our algorithms for operator scheduling and mining over data streams. First, we will briefly describe our experimental setup and stream processing implementation. Next, we will evaluate our scheduling scheme for its ability to adapt to heavy bursts in data streams, with streams arriving in a centralized and distributed setting, while providing the desired response time. Finally, we will evaluate precision, memory utilization and run-time adaptability for disk-aware stream processing schemes, with frequent itemset mining as our application.

6.1 Experimental Setup

All the experiments were conducted on a Dual-Pentium III (1GHz and 512MB RAM) four node cluster running Linux kernel 2.4. We used C++ with MPI [2] for message passing for the implementation. The system was implemented using a client-server architecture, the client nodes would process the data streams and locally schedule the operators, while the server node would provide the client with updates to their operator schedules as needed. In order to take accurate measurements for response times and memory usage, it was necessary for the clients not to out-run the server by a large margin. As a result, we used a MPI Barrier for the purpose of synchronization. Stream-based applications are still emerging

[2]http://www-unix.mcs.anl.gov/mpi

with no well accepted benchmarks. Babcock *et al* [14] used the Internet Traffic Archive [3] for burst simulations. Bursts being application specific, the stream rate in their dataset was not sufficient to generate a significant burst in our case and hence we decided to generate a synthetic stream using synthetic and real datasets. We generated a burst in the data-stream by modeling the tuple inter-arrival time as a Gaussian distribution with mean arrival time equal to half a time step, where a time step reflects the time required to process a single tuple, on average. The synthetic datasets for scheduling scheme evaluations were generated with random tuple attributes and those for frequent itemset mining were generated using the IBM Quest group synthetic dataset generator [4]. As for real datasets, we used the network intrusion detection dataset that formed a part of KDD Cup 1999 for scheduling scheme evaluations and the VBook dataset for frequent itemset mining. In order to achieve optimal scheduling, we would ideally like to re-assign tickets for every operator state change that would affect the schedule. This can be a significant overhead especially when states change rapidly initially. We handle this by re-issuing a ticket distribution every 1000 tuples, if needed.

6.2 Scheduling scheme evaluation

We considered the following different cases to verify response time adaptability for our scheduling scheme.

- Queries over a single stream without joins: The continuous query was constructed using four select operators, with different selectivities that could change at runtime as the stream is processed. The leaf node was set to be the most selective. As shown in Figure 9, for both the datasets, with the arrival of the burst in the data stream, the most selective operator is scheduled for an increasing number of time steps with increasing values of k, thus exhibiting distinct trends in response times and allowing the user to tune the application to the desired level of interactivity (Note: each k contributed towards 40 clock ticks approximately). In the most selective case $k = 500$, there is a near vertical rise in response time initially, during the burst. This is due to the repeated scheduling of the leaf operator in the query tree. We have consistent results on both the synthetic and real dataset.

- Queries over multiple distributed streams with and without joins: For this experiment, the continuous query was constructed using sixteen select operators with different selectivities, to be scheduled over four nodes, with four different streams arriving at each of the four nodes, streaming in at different rates. A burst was applied to the data streams arriving at three of the four nodes. Next, four join operators (based on sliding windows) were added to the query tree at non leaf positions at varying heights in the tree. We see distinct trends (Figures 10 and 11) over multiple nodes with and without joins, as was the case on a single node. Joins can contribute to a *selectivity* > 1 and are considered for completeness. Our distributed scheduling scheme is able to adapt to a distributed setting giving the user a level of control similar to the centralized case (Note: each k contributed towards 40 clock ticks approximately).

- k-Median clustering over data streams: As explained in the previous section, we decomposed the problem into a hierarchy of clustering operators with varying selectivities at the leaf nodes of the

[3]http://www.acm.org/sigcomm/ITA
[4]http://www.almaden.ibm.com/software/quest

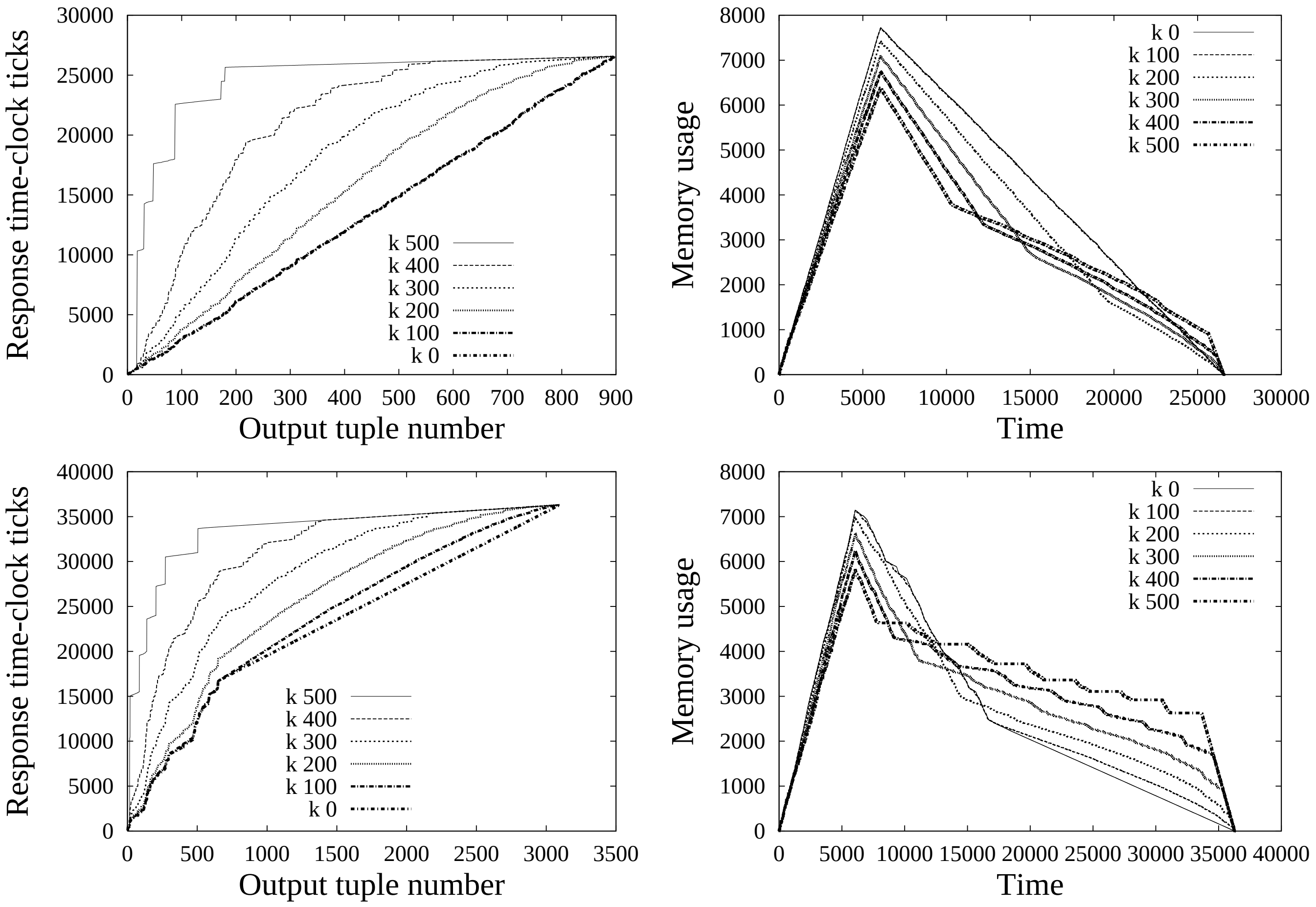


Figure 9. Single stream without join (a) Response times (b) Memory usage on Synthetic dataset (c) Response times (d) and Memory usage on Real dataset

tree due to streams arriving at different rates. A burst was introduced to a subset of leaf nodes varying the selectivity. As evident from Figure 12, the very same scheduling scheme extends to data mining operations as well (Note: each k contributed towards 20 clock ticks approximately).

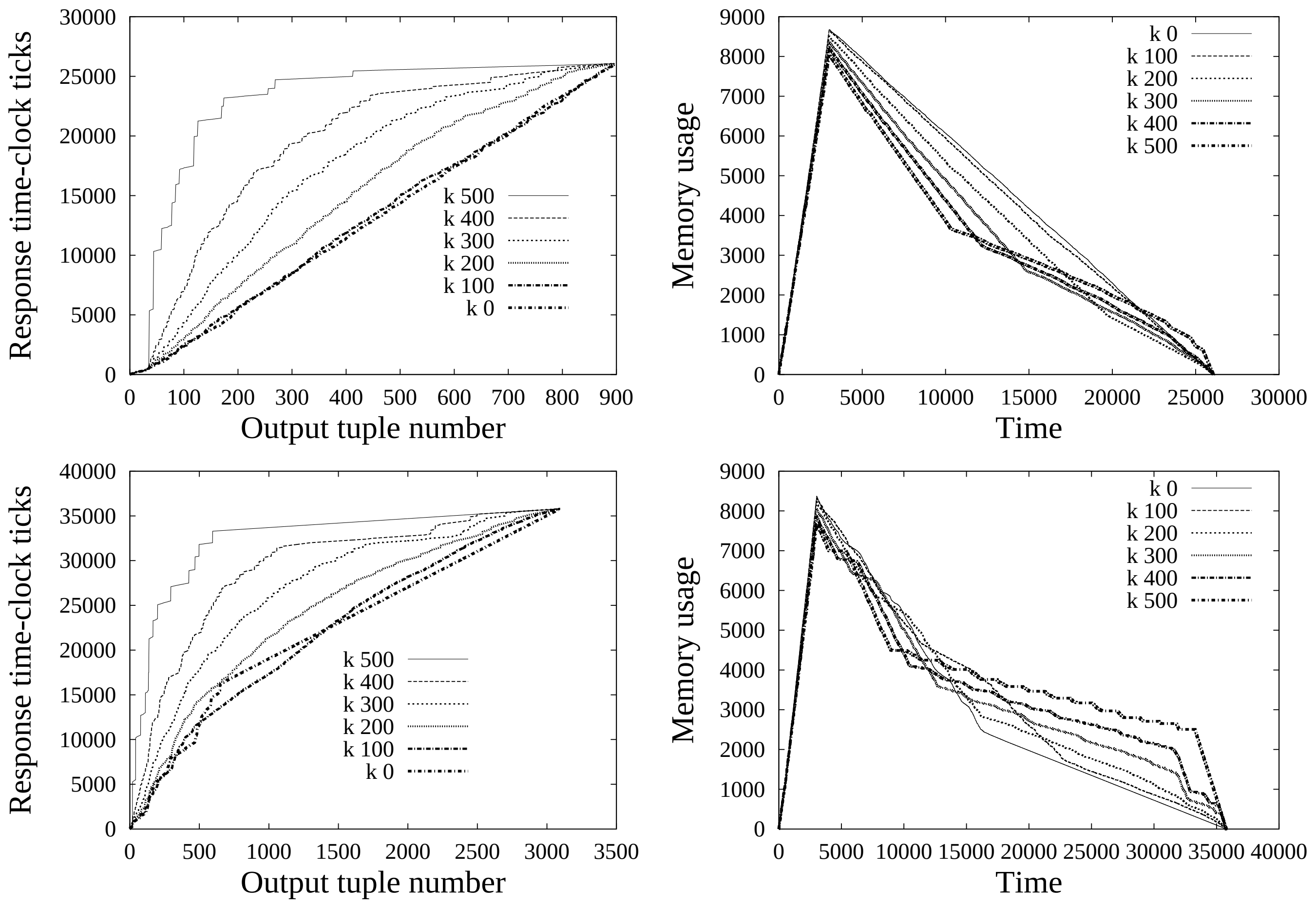


Figure 10. Multiple streams without join (a) Response times (b) Memory usage on Synthetic dataset (c) Response times (d) and Memory usage on Real dataset

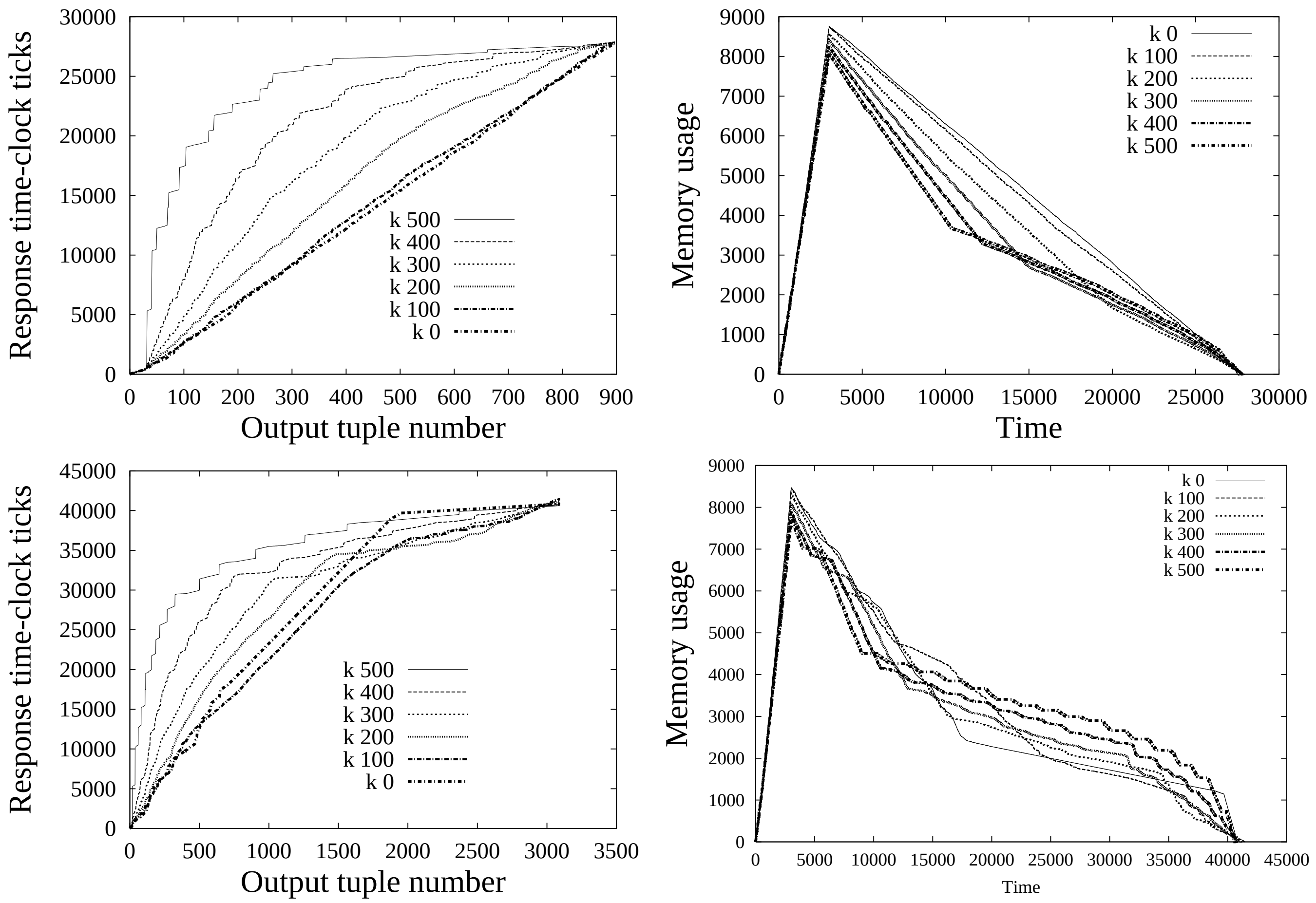


Figure 11. Multiple streams with join (a) Response times (b) Memory usage on Synthetic dataset (c) Response times (d) and Memory usage on Real dataset

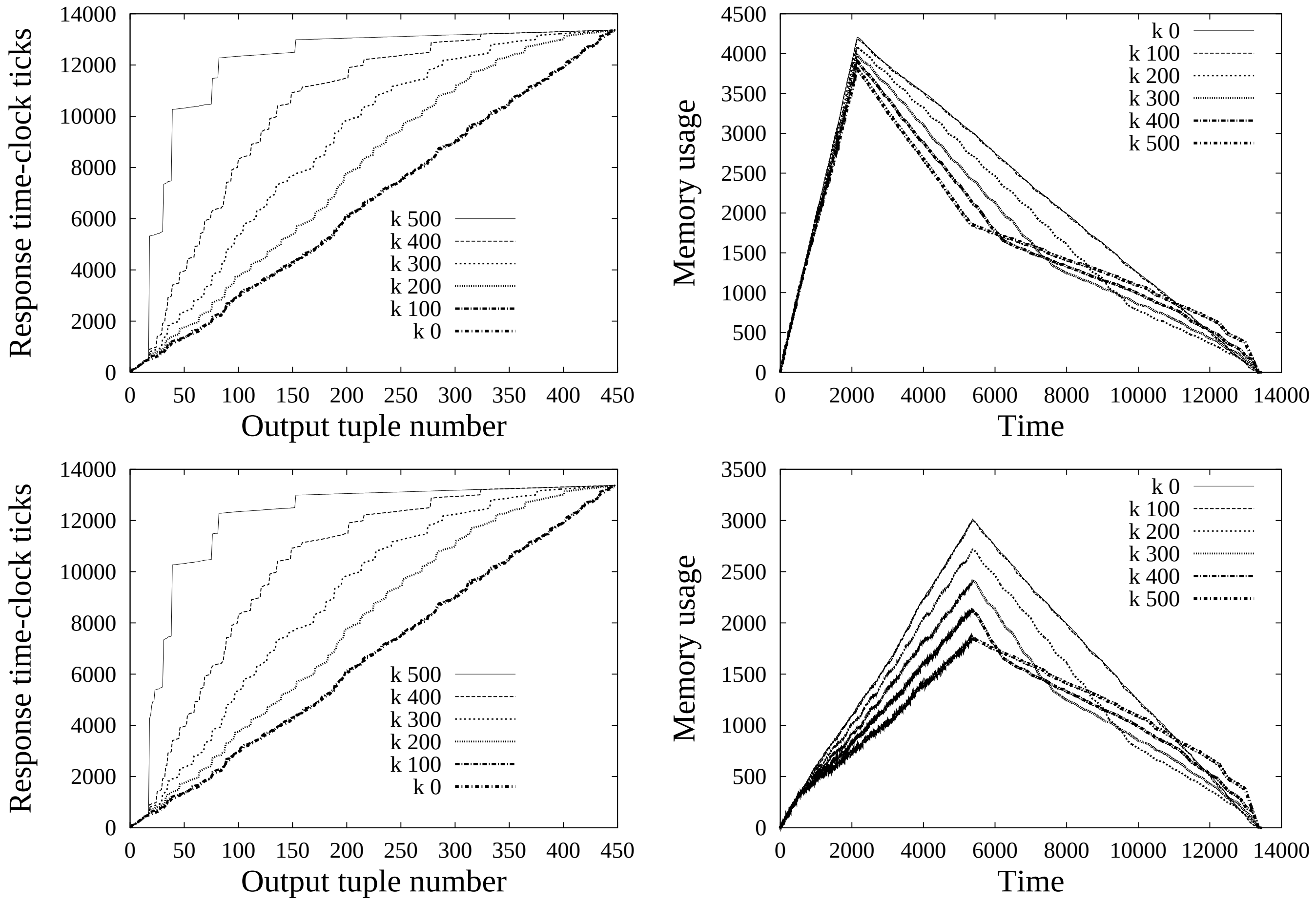


Figure 12. Clustering multiple streams (a) Response times (b) Memory usage on Synthetic dataset (c) Response times (d) and Memory usage on Real dataset

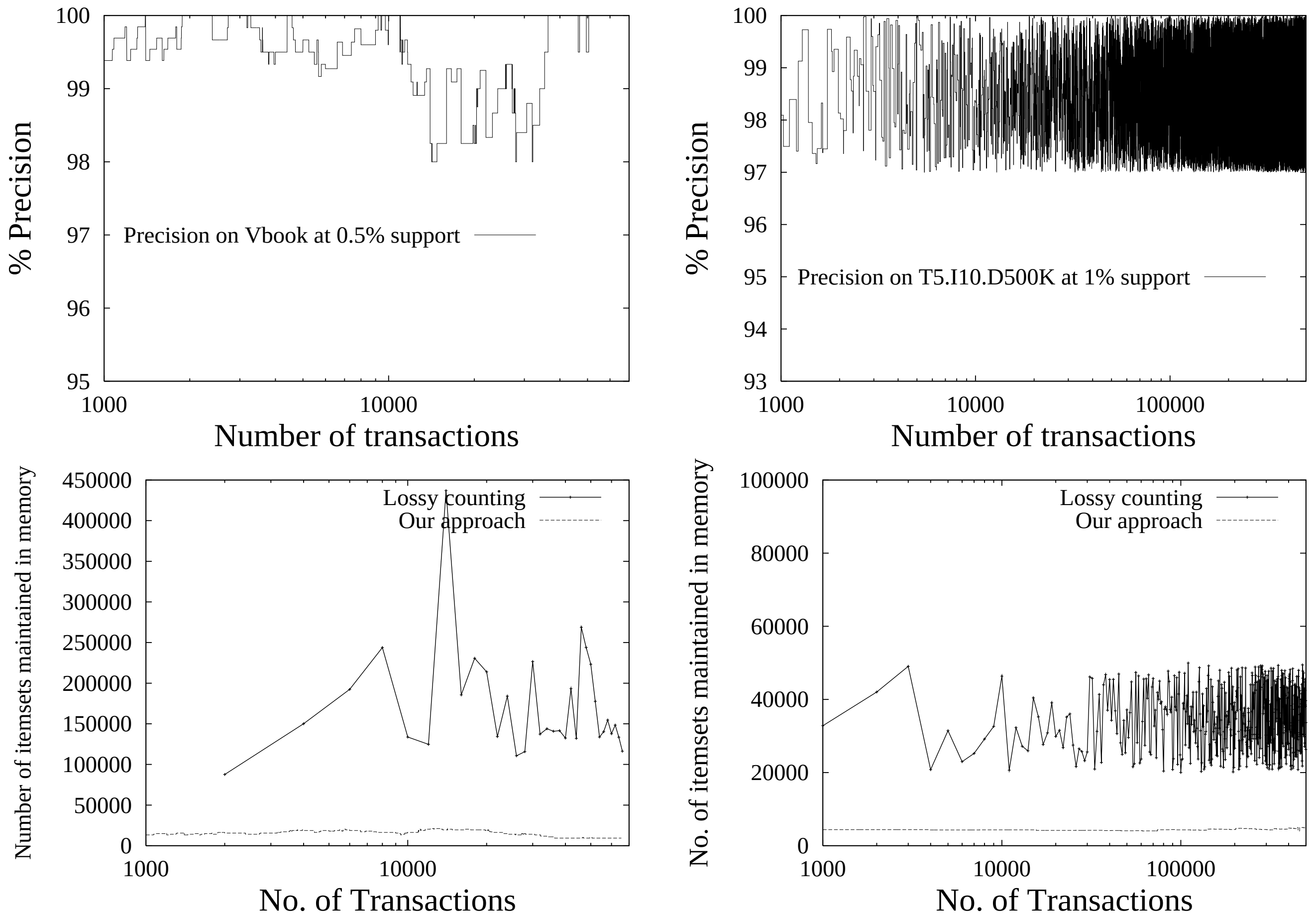


Figure 13. (a) Precision at 0.5 percent support (VBook) and (b) 1 percent support (T5.I10.D500K) (c) Itemsets in Memory at 0.5 percent support (VBook) and (d) 1 percent support (T5.I10.D500K)

6.3 Disk-aware frequent itemset mining

We varied support s from 0.005 to 0.01. We compared our approach to an implementation of the lossy counting approach described by Manku and Motwani [3]. We fixed the buffer size to $\frac{1}{0.1s}$ to achieve comparable precision. Figures 13 (a) (b) depict the precision achieved when mining for frequent itemsets at different support values. Precision was determined by comparing our results with the Apriori algorithm [13]. Disk-aware stream processing allows for efficient discovery of new itemsets as shown by converging precision values on all the datasets over different support values. Figures 13 (c) (d) illustrate how disk-aware processing allows the application to run in bounded amount of memory. The number of itemsets maintained in memory are bounded by the number of frequent itemsets, allowing us to run with smaller memory stamps. The Lossy counting approach needs to maintain a far larger number of itemsets in memory, which is a direct measure of the algorithm's memory usage. Lossy counting needs to buffer transactions in memory with a buffer size inversely proportional to support, which is not required with our approach. Disk-aware stream processing supports changing application parameters at runtime. This is a requirement when processing streams online, specially for ad-hoc queries, and is not supported by

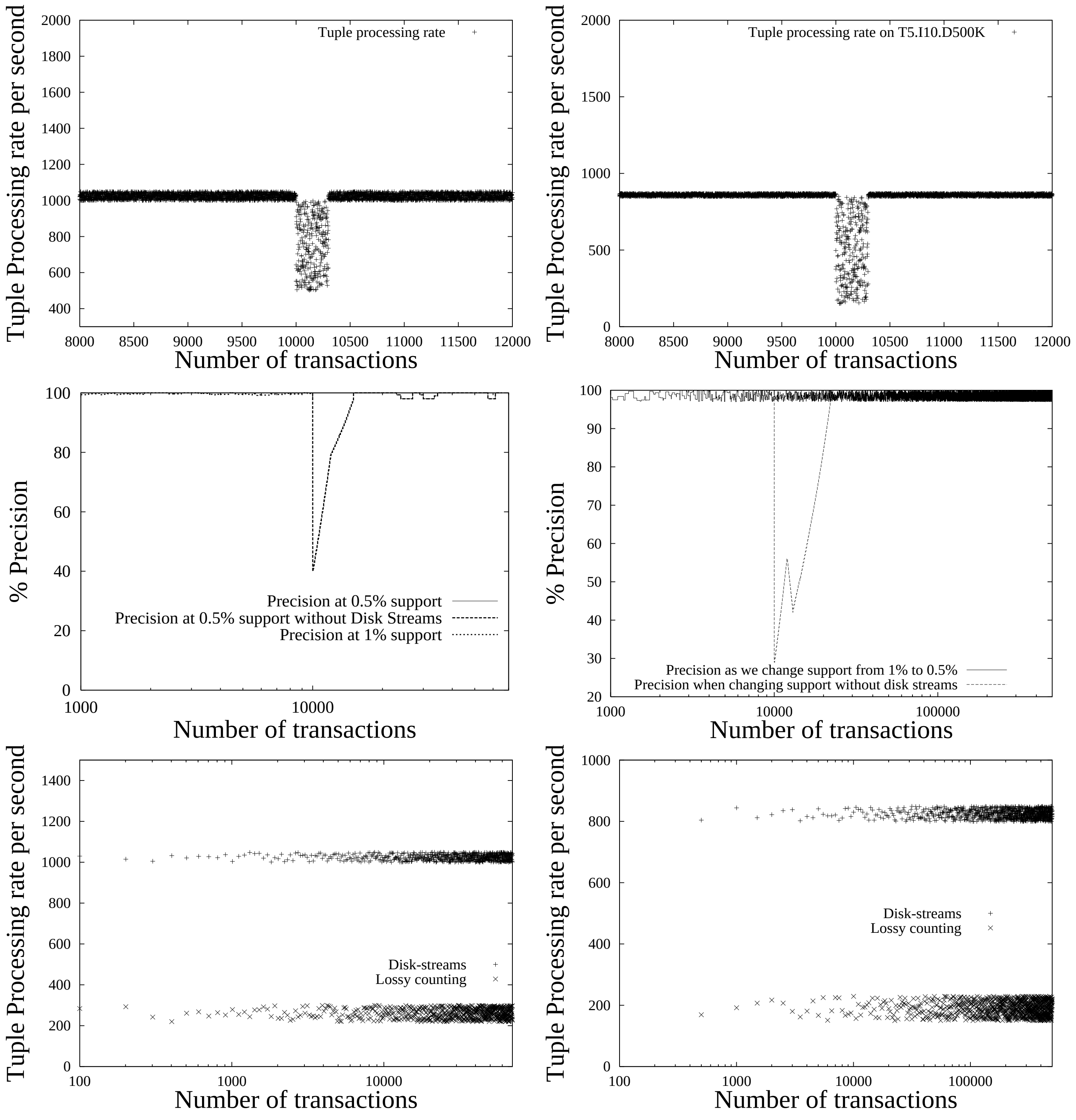


Figure 14. (a) Change in tuple processing rate when updating support values from 1 percent to 0.5 percent on VBook and (b) T5.I10.D500K (c) Change in precision when updating support values from 1 percent to 0.5 percent on VBook and (d) T5.I10.D500K (e) Tuple processing rate comparison on Vbook at 1 percent support and (f) T5.I10.D500K at 0.5 percent support

Lossy counting. The trade-off with our approach is a slight decrease in transaction processing rate due to disk-streams. Figure 14 illustrates how the architecture adapts to a change in the support value when we reach 10000 transactions on different datasets. A change in the support value results in an increase in the size of the lattice. Disk-streams allow us achieve the desired precision. The presence of disk-streams reduces the resulting tuple processing rate by a factor little more than 2 for a short period of time and is not very expensive because of the initial random binning. The decrease in the rate can be attributed to the fact that each operator needs to do some extra work for disk-streams. The disk-stream based approach is capable of processing each transaction as it arrives, while the lossy counting approach needs to buffer transactions with a buffer size inversely proportional to support values. In order to make a direct comparison between tuple processing rates for disk-stream and lossy counting based approaches, we buffered the same number of transaction for each of the approaches. A disk-stream based approach will operate on a smaller lattice as compared to the lossy counting approach, as is evident by larger tuple processing rates in Figures 13 (e) (f).

7 Conclusion

This paper makes the following key contributions. First, we looked at the problem of giving end users response-time guarantees over streams, specifically during a burst in the stream. We presented $RTOS-k$ schedules to solve the above problem and gave a ticketing algorithm to solve for ticket distributions for $RTOS-k$ schedules. We also proved the correctness of our ticket distributions. Second, we extended the scheduling problem to a distributed setting and defined and solved for optimal solutions in this setting. Third, we proposed a disk-aware stream processing solution motivated in part by new applications with new requirements. A disk-aware architecture coupled with effective algorithms allows online stream processing with low memory utilization. We also proposed an online algorithm for frequent itemset mining using the disk-aware scheme. Fourth, we evaluated our scheduling and disk-aware processing schemes extensively on both synthetic as well as real datasets. As for future work, we plan to use our schemes for an online network intrusion detection system and look at near optimal schemes to deal with scheduling overheads.

References

[1] B. Babcock *et al*, Models and Issues in Data Stream Systems. In Proceedings of PODS, 2002.

[2] G. Hulten, L. Spencer and P. Domingos, Mining Time-Changing Data Streams. In Proceedings of SIGKDD, 2001.

[3] G. Manku and R. Motwani, Approximate Frequency Counts over Data Streams. In Proceedings of VLDB, 2002.

[4] S. Guha, N. Mishra, R. Motwani and L. O'Callaghan, Clustering Data Streams. In Proceedings of FOCS, 2000.

[5] G. S. Manku, S. Rajagopalan and B. G. Lindsay, Approximate Medians and other Quantiles in One Pass and with Limited Memory. In Proceedings of SIGMOD, 1998.

[6] M. Henzinger, P. Raghavan and S. Rajagopalan, Computing on datastreams, 1998.

[7] P. Tucker, D. Maier, T. Sheard, and L. Fegaras, Punctuating Continuous Data Streams, manuscript, 2002.

[8] D. Terry, D. Golderg, D. Nichols and B. Oki, Continuous queries over append-only databases. In Proceedings of SIGMOD, 1992.

[9] P. Flajolet and G. Martin, Probabilistic counting. In Proceedings of FOCS, 1983.

[10] R. Motwani *et al*, Query Processing, Resource Management and Approximation in a Data Stream Management System. In Proceedings of CIDR, 2003.

[11] J. Chen *et al,* NiagaraCQ: A Scalable Continuous query System for Internet Databases. In Proceedings of SIGMOD, 2000.

[12] S. Madden, M. Shah, J. Hellerstein, V. Raman, Continuously Adaptive Continuous Queries over Streams. In Proceedings of SIGMOD, 2002.

[13] R. Agrawal, R. Srikant, Fast Algorithms for Mining Association Rules. In Proceedings of VLDB, 1994.

[14] B. Babcock, S. Babu, M. Datar and R. Motwani, Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. In Proceedings of SIGMOD, 2003.

[15] D. Barbara and S. Jajodia, editor, Applications of Data Mining in Computer Security. Kluwer, 2002.

[16] R. Machiraju *et al*, Mining of Complex Evolutionary Phenomena, Next Generation Data Mining, H. Kargupta and A. Joshi, eds, MIT Press.

[17] L. Lakshmanan, C. Leung, and R. Ng, The Segment Support Map: Scalable Mining of Frequent Itemsets. SIGKDD Explorations, Special Issue on Scalable Data Mining, 2000.

[18] 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.

[19] R. Agrawal and R. Srikant, Mining sequential patterns. In Proceedings of ICDE, 1995.

[20] S. Brin, R. Motwani and C. Silverstein, Beyond market basket: Generalizing association rules to coorelations. In Proceedings of SIGMOD, 1997.

[21] S. Bay and M. Pazzani, Detecting change in categorical data: mining contrast sets. In Proceedings of SIGKDD, 1999.

[22] M. Zaki, S. Parthasarathy, M. Ogihara and W. Li, New Algorithms for Fast Discovery of Association Rules. In Proceedings of SIGKDD, 1997.

[23] S. Parthasarathy, Efficient Progressive Sampling for Association Mining. In Proceedings of IEEE ICDM, 2002.

[24] E. Markatos and T. Leblanc, Using processor affinity in loop scheduling on shared-memory multiprocessors. In Proceedings of IEEE TPDS, 1994.

[25] S. Parthasarathy, M. Zaki and W. Li, Memory placement techniques for association mining. In Proceedings of SIGKDD, 1998.