Exploiting *k*-Constraints to Reduce Memory Overhead in Continuous Queries Over Data Streams

SHIVNATH BABU, UTKARSH SRIVASTAVA, and JENNIFER WIDOM
Stanford University

Continuous queries often require significant run-time state over arbitrary data streams. However, streams may exhibit certain data or arrival patterns, or *constraints*, that can be detected and exploited to reduce state considerably without compromising correctness. Rather than requiring constraints to be satisfied precisely, which can be unrealistic in a data streams environment, we introduce *k-constraints*, where k is an *adherence parameter* specifying how closely a stream adheres to the constraint. (Smaller k's are closer to strict adherence and offer better memory reduction.) We present a query processing architecture, called *k-Mon*, that detects useful k-constraints automatically and exploits the constraints to reduce run-time state for a wide range of continuous queries. Experimental results showed dramatic state reduction, while only modest computational overhead was incurred for our constraint monitoring and query execution algorithms.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Continuous queries, data streams, constraints

1. INTRODUCTION

There has been a surge of interest recently in query processing over continuous data streams [Gehrke 2003; Golab and Ozsu 2003a]. In many of the relevant applications—network monitoring, sensor processing, Web tracking, telecommunications, and others—queries are long-running, or *continuous*. One challenge faced by continuous-query processing engines is the fact that continuous queries involving joins or aggregation over streams may require significant amounts of memory to maintain the necessary run-time state. (Disk could also be used to maintain state; doing so does not change our basic algorithms or storage overhead results.) We begin by illustrating the overall problem and the solutions we propose in this article using a fairly detailed example drawn from the network monitoring domain.

This work was supported by the National Science Foundation under grants IIS-9817799 and IIS-0118173 and by a Sequoia Capital Stanford Graduate Fellowship.
Authors' address: Computer Science Department, Stanford University, 438 Gates Hall, Stanford, CA 94305-9040; email: {shivnath,usriv,widom}@cs.stanford.edu.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2004 ACM 0362-5915/04/0900-0545 $5.00

1.1 Motivating Example

One application of a data stream processing system is to support traffic monitoring for a large network such as the backbone of an Internet Service Provider (ISP) [Caceres et al. 2000; Cranor et al. 2003]. An example continuous query in this application is the following [Babcock et al. 2002; Duffield and Grossglauser 2000]:

> *Monitor the total traffic from a customer network that went through a specific set of links in the ISP's network within the last 10 minutes.*

A network analyst might pose this query to detect service-level agreement violations, to find opportunities for load balancing, to monitor network health, or for other reasons.

Let C denote the link carrying traffic from the customer network into the ISP's network. Let B be an important link in the ISP's network backbone and let O be an outgoing link carrying traffic out of the ISP's network. Data collection devices on these links collect packet headers (possibly sampled), do some processing on them (e.g., to compute packet identifiers), and then stream them to the system running the continuous query [Caceres et al. 2000; Cranor et al. 2003; Duffield and Grossglauser 2000; NETFLOW 2003]. Thus, we have three streams denoted C, B, and O, each with schema (`pid`, `size`): packet identifier and size of the packet in bytes. The above continuous query can be posed using a declarative language such as *CQL* [Arasu et al. 2002] or *GSQL* [Cranor et al. 2003]. In CQL:

```
Select   sum(C.size)
From     C [Range 10 minutes], B [Range 10 minutes],
         O [Range 10 minutes]
Where    C.pid = B.pid and B.pid = O.pid
```

This continuous query joins streams C, B, and O on `pid` with a 10-min *sliding window* of tuples on each stream and aggregates the join output to continuously compute the total common traffic [Arasu et al. 2002]. (A similar query could be used in sensor networks, e.g., to monitor moving objects and their paths [Hammad et al. 2003; Arasu et al. 2004].)

Based on recent stream query processing techniques suggested in the literature [Golab and Ozsu 2003b; Hammad et al. 2003; Madden et al. 2002; Raman et al. 2003; Tucker et al. 2003; Viglas et al. 2003], an efficient plan to execute this query over arbitrary streams is as follows: for each stream, maintain a hash table indexed on `pid` containing the last 10 min of data in the stream. When a tuple arrives in stream O, do a lookup in the hash table on B, and for each joining tuple do a further lookup on the hash table on C to compute all new tuples in the join result. (Of course, the join order could be reversed [Golab and Ozsu 2003b; Raman et al. 2003; Viglas et al. 2003].) For each new tuple in the join result, maintain the sum aggregate incrementally. Similar processing occurs when new tuples arrive in streams C and B. When a tuple expires (is more than 10 min old), join it with the two other hash tables to compute the tuples that drop out of the join result, and update the sum. The total memory required

is roughly the sum of the tuples in the three windows (plus some extra memory for the hash table structures). Assuming 10-byte tuples and tuple rates of 10^3, 10^4, and 10^3 per second in streams C, B, and O, respectively, the total memory requirement is at least 72 MB, which is relatively high for a single query.

The streams in this application exhibit some interesting properties. First, the packets we are monitoring flow through link C to link B to link O. Thus, a tuple corresponding to a specific `pid` appears in stream C first, then a joining tuple may appear in stream B, and last in stream O. Second, if the latency of the network between links C and B and between links B and O is bounded by d_{cb} and d_{bo}, respectively, then a packet that flows through links C, B, and O will appear in stream B no later than d_{cb} time units after it appears in stream C, and in stream O no later than d_{bo} time units after it appears in stream B. Both of these properties, if "known" to the continuous query processor, can be exploited to reduce the memory requirement significantly: when a tuple t arrives in stream B and no joining tuple exists in the window on C, t can be discarded immediately because a tuple in C joining with t should have arrived before t. Furthermore, assuming tuples arrive in timestamp order on stream B, the query processor can discard a tuple t with timestamp ts from the window on C when a tuple with timestamp $> ts + d_{cb}$ arrives on B and no tuple joining with t has arrived so far on B. Similar memory reductions can be applied to the windows on B and O. To appreciate the scale of the memory reduction, let us assume that approximately 10% of the tuples on link C go on to link B, and independently 10% of the tuples on B go on to O. Then, the total memory requirement is roughly 0.18 MB, a two-orders-of-magnitude reduction.

1.2 Challenges in Exploiting Stream Properties

The example in the previous section illustrates how the memory requirement can be reduced by orders of magnitude if stream properties are exploited during query processing. Three challenges need to be addressed:

(1) The stream properties used in our example seem application-specific. Is there a set of properties that are useful across a wide variety of applications and continuous queries?
(2) The query processor has little control over the data and arrival patterns of streams [Babcock et al. 2002; Golab and Ozsu 2003a]. We assumed that tuples in stream C would arrive before their joining tuples in B. However, delays and reordering in the network may cause minor violations of this assumption.
(3) Stream properties can change during the lifetime of a long-running continuous query [Gehrke 2003; Hellerstein et al. 2000]. For example, the latency bound d_{cb} may change based on congestion in the network.

To address the first challenge, we studied several data stream applications and identified a set of basic *constraints* that individually or in combination capture the majority of properties useful for memory reduction in continuous queries [SQR 2003]. The basic constraints we identified are *many-one joins*, *stream-based referential integrity*, *ordering*, and *clustering*.

To address the second challenge, we introduced the notion of *k-constraints*. $k \geq 0$ is an *adherence parameter* capturing the degree to which a stream or joining pair of streams adheres to the strict interpretation of the constraint. The constraint holds with its strict interpretation when $k = 0$. For example, *k-ordering* specifies that out-of-order stream elements are no more that k elements apart. The concept of k-constraints is very important in the stream context since it is unreasonable to expect streams to satisfy stringent constraints at all times, due to variability in data generation, network load, scheduling, and other factors. But streams may frequently come close to satisfying constraints and k-constraints enable us to capture and take advantage of these situations.

To address the third challenge, we developed an architecture where the query processor continuously monitors input streams to detect potentially useful k-constraints. This approach adapts to changes in stream constraints and enables the query processor to give the best memory reduction based on the current set of constraints. It frees users and system administrators from keeping track of stream constraints, thereby improving system manageability. As we will see, only modest computational overhead is incurred for constraint monitoring and for constraint-aware query processing.

1.3 Stream Constraints Overview

Next we informally describe in a bit more detail the constraint types and adherence parameters we consider. We continue considering the network monitoring application introduced in Section 1.1. Section 9 provides more examples from *Linear Road*, a sensor-based application being developed as a benchmark for data stream systems [Arasu et al. 2004; Arasu 2003]. Detailed specifications of these examples are provided in the *electronic appendix*.

1.3.1 *Join Constraints.* In the query in Section 1.1, the join between each pair of streams is a *one-one join*. One-one joins are a special case of *many-one joins*, which are very common in practice [SQR 2003]. As we will see in Section 9, most joins in the Linear Road queries are many-one joins. In this article we will assume that all joins in our queries are many-one joins. Our overall approach, theorems, and algorithms are fairly independent of this assumption, but the benefit of our algorithms is reduced in the presence of many-many joins.

An additional join constraint that we saw in Section 1.1 bounded the delay between the arrival of a tuple on one stream and the arrival of its joining tuple on the other stream. We define a *referential integrity constraint* on a many-one join from stream S_1 to stream S_2 with adherence parameter k as follows: for a tuple $s_1 \in S_1$ and its unique joining tuple $s_2 \in S_2$, s_2 will arrive within k tuple arrivals on S_2 after s_1 arrives. For the special case of $k = 0$ for this constraint, termed *strict referential integrity*, s_2 will always arrive before s_1. For example, in the join from stream C to stream B in Section 1.1, a referential integrity constraint holds with $k = d_{cb} \cdot r_B$, where r_B is the arrival rate of stream B. A strict referential integrity constraint holds on the join from stream O to stream B.

Note that we have chosen to use tuple-based constraints in this paper, but time-based constraints also can be used without affecting our basic approach.

1.3.2 *Ordered-Arrival Constraints.* Streams often arrive roughly in order according to one of their attributes, such as a timestamp or counter attribute. We define an *ordered-arrival constraint* on a stream attribute $S.A$ with adherence parameter k as follows: for any tuple s in stream S, all S tuples that arrive at least $k + 1$ tuples after s have an A value $\geq s.A$. That is, any two tuples that arrive out of order are within k tuples of each other. Note that $k = 0$ captures a strictly nondecreasing attribute.

In the network monitoring domain, network measurement streams often are transmitted via the UDP protocol instead of the more reliable but more expensive TCP protocol [NETFLOW 2003]. UDP may deliver packets out of order, but we can generally place a bound on the amount of reordering in the stream based on network delays. Similar scenarios arise in sensor networks [Hammad et al. 2003].

1.3.3 *Clustered-Arrival Constraints.* Even when stream tuples are not ordered, they may be roughly clustered on an attribute. We define a *clustered-arrival constraint* on stream attribute $S.A$ with adherence parameter k as follows: if two tuples in stream S have the same value v for A, then at most k tuples with non-v values for A occur on S between them.

For example, if we consider the union of streams C, B, and O in Section 1.1, then all tuples for a particular `pid` will be relatively close together in the stream. In the Linear Road application, the incoming sensor stream is approximately clustered on a combination of *car* and *segment* identifiers [Arasu et al. 2004]; see Section 9.

1.4 Queries and Execution Overview

The continuous queries considered in this paper are *Select-Project-Join* (*SPJ*) queries over data streams with optional sliding windows over the streams, like the example query in Section 1.1. We introduce another CQL example query to illustrate our execution strategy:

Select Istream(*)
From S_1 [Rows 50,000], S_2 [Rows 50,000]
Where $S_1.A = S_2.A$ and $S_2.B > 10$

Here we use 50,000-tuple sliding windows on each stream and the *Istream* operator outputs the query result as a stream [Arasu et al. 2002]. As in Section 1.1, the straightforward way to process this query is as follows: maintain two *synopses* (e.g., hash tables) containing the last 50,000 tuples in each stream. When a new tuple s arrives in S_1, join s with S_2's synopsis and output the joined tuples in the result stream. Add s to S_1's synopsis, discarding the earliest tuple once the window is full; similarly for S_2. (Tuples expired from windows can be discarded without any processing because the query result is a stream [Golab and Ozsu 2003b; Hammad et al. 2003; Kang et al. 2003; Viglas et al. 2003].)

Notice that the filter predicate cannot be applied independently before the join since S_2's window must be based on all tuples in S_2. However, we can discard S_2 tuples that fail the filter predicate provided we keep track of the arrival order of the discarded tuples so that S_2's window can be maintained

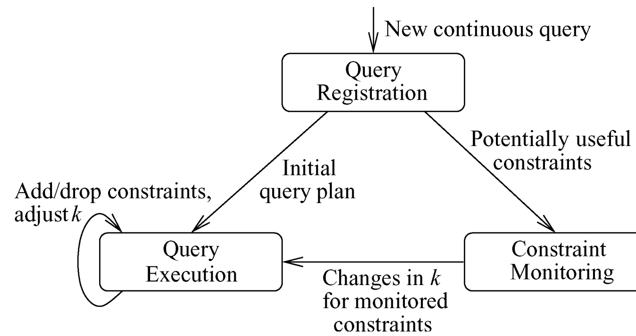

Fig. 1. The k-Mon architecture.

correctly. As an example, if the filter predicate's selectivity is 50%, then our S_2 synopsis would now contain 25,000 tuples on average (and 25,000 placeholders), instead of 50,000. In our experiments we refer to this overall algorithm as a *sliding-window join*, or *SWJ*.

Now suppose the join is many-one from S_1 to S_2. We can immediately eliminate any tuple in S_1's synopsis once it joins with a tuple in S_2, often reducing S_1's synopsis size considerably. For a tuple $s_2 \in S_2$ that cannot contribute to the result because it fails the filter predicate, we might prefer to store rather than discard s_2 (actually only $s_2.A$ needs to be stored) since it allows us to immediately discard any future joining tuples arriving on S_1 which would otherwise stay in S_1's synopsis until they drop out of S_1's window. If strict referential integrity holds over the join, then we need no synopsis at all for S_1, since for a tuple $s_1 \in S_1$ and its unique joining tuple $s_2 \in S_2$, when s_1 arrives, s_2 must either appear in S_2's synopsis or it has dropped out of S_2's window. If we have referential integrity with adherence parameter k, then a tuple $s_1 \in S_1$ must be saved for at most k arrivals on S_2 after the arrival of s_1. Furthermore, if S_2 satisfies k-ordered-arrival on $S_2.A$, then a tuple $s_1 \in S_1$ must be saved for at most k arrivals on S_2 following any $S_2.A$ value greater than $s_1.A$.

This example and the example in Section 1.1 illustrate how k-constraints can be used to reduce synopsis sizes considerably. However, obtaining the most memory reduction in the general case is quite complex since we must consider arbitrary queries and arbitrary combinations of stream constraints.

1.5 *k-Mon*: An Architecture for Exploiting *k*-Constraints

We now discuss our overall query processing architecture, called *k-Mon*, which detects and exploits different types of stream constraints automatically to reduce the memory requirement for continuous SPJ queries. *k-Mon* integrates algorithms for monitoring k-constraints and exploiting them during query execution. The basic structure of k-Mon is shown in Figure 1. Continuous queries are registered with the *Query Registration* component, which generates an initial query plan based on any currently known k-constraints on the input streams in the query. The *Query Execution* component begins executing this plan.

At the same time, the query registration component informs the *Constraint Monitoring* component about constraints that may be used to reduce the

memory requirement for this query. Identifying potentially useful constraints for SPJ queries is straightforward, as will be seen when our query execution algorithm is presented in Section 3.2. The monitoring component monitors input streams continuously and informs the query execution component whenever k values for potentially useful constraints change. (We actually combine constraint monitoring with query execution whenever possible to reduce the monitoring overhead; see Sections 4.3, 5.3, and 6.3.) The execution component adapts to these changes by adjusting its k values used for memory reduction. Obviously if a k value is very high (e.g., when a constraint does not hold at all, k grows without bound), the memory reduction obtained from using the constraint may not be large enough to justify the extra computational cost. The decision of when to exploit constraints and when not to is part of a larger cost-based query optimization framework we are developing, and is beyond the scope of this article. In this article we simply assume the query execution component ignores constraints with k values higher than some threshold.

Our query execution algorithm assumes adherence to k-constraints within the values for k given by the monitoring component. Specifically, during query execution some state is discarded that would otherwise be saved if the constraints did not hold or if k values were higher (indicating less adherence). If our monitoring algorithms underestimate k, particularly if k increases rapidly, then for the queries we consider, *false negatives* (missing tuples) may occur in query results. In many stream applications modest inaccuracy in query results is an acceptable tradeoff for more efficient execution [Dobra et al. 2002], especially if the inaccuracy persists for only short periods. The example query in Section 1.1 clearly has this property. If false negatives cannot be tolerated under any circumstance, then our approach can still be used, pushing "probably unnecessary" state to disk instead of discarding it entirely. Potential joins between tuples on disk and those in memory can be detected using one of two common approaches: join keys of tuples on disk can be retained in main-memory indexes or these join keys can be hashed into in-memory Bloom filters [Bloom 1970].

In this article we instantiate the k-Mon architecture for the referential-integrity, ordering, and clustering constraints outlined in Sections 1.3.1–1.3.3. We have implemented k-Mon as part of the *StreaMon* adaptive query processing engine in the *STREAM* prototype data stream management system at Stanford [Babu and Widom 2004]. In addition to k-constraints, StreaMon implements new adaptive algorithms for selection and join ordering [Babu et al. 2004a], and for subresult caching [Babu et al. 2004b].

1.6 Outline of Paper

We discuss related work in Section 2. Section 3 formalizes the queries we consider and describes our basic query execution algorithm. Sections 4–6 formalize the three constraint types we consider, incorporate them into our execution algorithm, present monitoring algorithms for them, and include experimental results for each constraint type. Section 7 measures the computational overhead of our architecture. Section 8 summarizes our complete approach. Section 9

provides examples and experiments from the Linear Road application, and Section 10 concludes the article.

2. RELATED WORK

A comprehensive description of work relating to data streams and continuous queries is provided in, for example, Golab and Ozsu [2003a]. Here we focus on work specifically related to query processing in the presence of constraints, run-time memory overhead reduction, and constraint monitoring.

Most current work on processing continuous queries over streams addresses the memory problem by requiring finite windows on all streams [Carney et al. 2002; Chandrasekharan and Franklin 2002; Das et al. 2003; Golab and Ozsu 2003b; Hammad et al. 2003; Kang et al. 2003; Madden et al. 2002; Viglas et al. 2003]. Our constraint-based approach serves two purposes in this setting. First, in many cases window sizes are dictated by semantic concerns like in Section 1.1, or window sizes are set conservatively in order to ensure with high probability that joining tuples do fall into concurrent windows, since properties of streams may not be known. In this case the SWJ algorithm (Section 1.4) may waste an excessive amount of memory, while our approach reduces synopses to contain only the data actually needed. Second, our approach permits users to omit window specifications entirely (with the default of an unbounded window), since we use k-constraints to effectively impose the appropriate windows based on properties of the data.

The work most closely approaching ours is *punctuated data streams* [Tucker et al. 2003]. Punctuations are assertions inserted into a stream to convey information on what can or cannot appear in the remainder of the stream. The query processor can use this information to reduce memory overhead for joins and aggregation and to know when results of blocking operators can be streamed. However, Tucker et al. [2003] did not address constraints over multiple streams, adherence parameters, or constraint monitoring. *W-join*, a multiway stream join operator supporting many types of sliding window specifications and algorithms to reduce stored data based on these specifications, was proposed in Hammad et al. [2003]. W-join does not address other types of constraints, adherence parameters, or constraint monitoring. Other techniques for controlling memory overhead in continuous query environments include using disk to buffer data for memory overflows [Carney et al. 2002; Urhan et al. 1998], grouping queries or operators to minimize memory usage [Chen et al. 2000; Madden et al. 2002], a wide variety of memory-efficient approximation techniques [Dobra et al. 2002; Ganguly et al. 2004; Srivastava and Widom 2004], and run-time *load shedding* [Tatbul et al. 2003]. None of these techniques are based on stream constraints.

Reference [Garcia-Molina et al. 1998] presents a language for expressing constraints over relations and views and develops algorithms to exploit the constraints for deleting data no longer needed for maintaining materialized views. However, the language and algorithms in Garcia-Molina et al. [1998] are inadequate to support constraints over streams (as opposed to relations)

because streams have arrival characteristics in addition to data characteristics. Helmer et al. [1998] exploited clustering based on the time of data creation to use SWJ-like techniques for joins over regular relations.

Algorithms to detect strict stream ordering or clustering with low space and time overhead are presented in Feigenbaum et al. [2000], and Ajtai et al. [2002] proposed algorithms to count the number of out-of-order pairs of stream elements. These works did not address constraints over multiple streams, adherence parameters, or query processing.

3. FOUNDATIONS

3.1 Data Streams and Continuous Queries

A *continuous data stream* (hereafter *stream*) is a potentially infinite stream of relational tuples. For exposition we will first consider continuous SPJ queries over streams with unbounded windows. Extending to streams with sliding windows is straightforward and is described in Section 3.4. The answer to a continuous query Q over a set of streams $S_1, S_2, \ldots, S_n$ at a point in time τ is the conventional relational answer to Q over the portion of the streams up to τ, treated as relations. We use $S(\tau)$ to denote the set of tuples that have arrived in stream S up to time τ. We assume that query results are themselves streams, so we do not account for the cost of storing query results.

For now we assume that all attributes in the streams are included in the query result. We will consider projection in Section 3.3.2. In this article the selection conditions we consider are conjunctions of any number of *filter predicates* over single streams along with any number of *equijoin predicates* over pairs of streams. For clarity of presentation, let us assume that the predicates in our queries are closed under implication.

As mentioned earlier, we assume that all joins in queries are many-one joins. That is, if Q contains one or more join predicates between streams S_1 and S_2, then we are guaranteed that each tuple on stream S_1 joins with at most one tuple on S_2 (e.g., if Q contains $S_1.A = S_2.B$ and $S_2.B$ is a key), or vice versa. We denote a many-one join from S_1 to S_2 as $S_1 \to S_2$, and we can thus construct a *directed join graph* $G(Q)$ for any continuous query Q we consider. Each stream $S \in Q$ along with any filter predicates over S produces a vertex in $G(Q)$, and each join $S_1 \to S_2$ produces an edge from S_1 to S_2. We assume that all join graphs are connected. A number of technical definitions related to join graphs are needed:

—Given $S_1 \to S_2$, S_1 is the *parent stream* and S_2 is the *child stream*. In a join graph $G(Q)$, *Children*(S) denotes the set of child streams of S and *Parents*(S) denotes the set of parent streams of S. A stream with no parents is called a *root stream*.

—Given $S_1 \to S_2$ with joining tuples $s_1 \in S_1$ and $s_2 \in S_2$, s_2 is the unique *child tuple of* s_1, and s_1 is a *parent tuple of* s_2.

—In a join graph $G(Q)$ containing a stream S, $G_S(Q)$ denotes the directed subgraph of $G(Q)$ containing S, all streams reachable from S by following

directed edges, the filter predicates over these streams, and all induced edges. We abuse notation and sometimes use $G_S(Q)$ to denote the result of the query corresponding to the join (sub)graph $G_S(Q)$.

—A set ρ of streams in $G(Q)$ is a *cover* of $G(Q)$ if every stream in $G(Q)$ is reachable from some stream in ρ by following directed edges. ρ is a *minimal cover* if no proper subset of ρ is a cover, and we use $MinCover(G(Q))$ to denote the set of minimal covers of $G(Q)$.

—$G(Q)$ is *directed-tree-shaped* (*DT-shaped*) if there are no cycles in the undirected version of the graph. (Recall that we assume join graphs are connected.) We cover only DT-shaped joins graphs in the main body of the paper. Algorithms for *DAG-shaped* and *cyclic* join graphs are presented in the *Electronic Appendix*.

For query execution and for synopsis reduction techniques, our synopsis for each stream S in a query Q is divided logically into three components formally defined as follows.

Definition 3.1 (*Synopsis*). Let S be a stream. $\mathcal{S}(S)$ denotes a *synopsis* for S and has three components defined as follows. Consider a time τ and a tuple $s \in S(\tau)$.

(1) $s \in \mathcal{S}(S).Yes$ at time τ if $s \ltimes G_S(Q)$ is nonempty at time τ. (Note that due to monotonicity of $G_S(Q)$, $s \ltimes G_S(Q)$ will remain nonempty for all times after τ if $s \ltimes G_S(Q)$ is nonempty at time τ.)
(2) $s \in \mathcal{S}(S).No$ at time τ if $s \ltimes G_S(Q)$ is empty at time τ and is guaranteed to remain empty at all future times.
(3) $s \in \mathcal{S}(S).Unknown$ at time τ if $s \notin \mathcal{S}(S).Yes$ and $s \notin \mathcal{S}(S).No$ at time τ.

Informally, *Yes* contains tuples that may contribute to a query result, *No* contains tuples that cannot contribute, and *Unknown* contains tuples we cannot (yet) distinguish.

3.2 Basic Query Execution Algorithm

In this section we define a query execution algorithm that we will use as a basis for our constraint-specific memory reduction techniques in subsequent sections. We separate two aspects of processing a continuous query using our synopsis approach:

(1) maintaining the synopses as new tuples arrive in the streams (*synopsis maintenance*), and
(2) generating new query result tuples as they become available (*result generation*).

Consider a join graph $G(Q)$. We maintain one synopsis for each stream in $G(Q)$. For now let us assume that all attributes (columns) are kept in all synopses; Section 3.3.2 shows how in many cases we can eliminate columns. The following theorems are based on Definition 3.1 of synopsis components, and they suggest a method for synopsis maintenance. (Proofs for all theorems are provided in the *electronic appendix*.)

```
/* Insert tuple s into the synopsis of stream S */
1.  Procedure 𝒮(S).InsertTuple(s) {
2.      if (s fails a filter predicate on S) {
3.          /* Add s to 𝒮(S).No */
4.          𝒮(S).No.InsertTuple(s); return; }
5.      For each stream R ∈ Children(S) {
6.          /* If the child tuple of s in R is present in 𝒮(R).No, then add s to 𝒮(S).No */
7.          if ((s → 𝒮(R).No) ≠ φ) {
8.              𝒮(S).No.InsertTuple(s); return; }}
9.      For each stream R ∈ Children(S) {
10.         /* If the child tuple of s in R is in 𝒮(R).Unknown, add s to 𝒮(S).Unknown */
11.         if ((s → 𝒮(R).Unknown) ≠ φ) {
12.             𝒮(S).Unknown.InsertTuple(s); return; }
13.         /* Further, if the child tuple of s in R is not present in 𝒮(R).Yes, then the
14.             child tuple has not yet arrived in R. Add s to 𝒮(S).Unknown */
15.         if ((s → 𝒮(R).Yes) = φ) {
16.             𝒮(S).Unknown.InsertTuple(s); return; }}
17.     /* Otherwise, S has no children, or all child tuples of s are present in the respective
18.         Yes components. Add s to 𝒮(S).Yes */
19.     𝒮(S).Yes.InsertTuple(s); }
```

Fig. 2. Procedure invoked when a tuple s arrives in stream S.

```
/* Insert tuple s into the Yes synopsis component of stream S */
1.  Procedure 𝒮(S).Yes.InsertTuple(s) {
2.      Insert s into 𝒮(S).Yes;
3.      /* Propagate the effects of the insertion */
4.      if (S is a root stream) {
5.          Join s with the Yes components of other streams to produce the new tuples in
6.              the query result; }
7.      else {
8.          For each stream R ∈ Parents(S) {
9.          /* Reevaluate the criteria for each tuple r in the Unknown component of parent
10.             stream R that joins with s */
11.             For each tuple r ∈ 𝒮(R).Unknown {
12.                 if ((r → s) ≠ φ) {
13.                     Delete tuple r from 𝒮(R).Unknown; 𝒮(R).InsertTuple(r); }}}}}
```

Fig. 3. Procedure to insert s into $\mathcal{S}(S)$.*Yes*.

THEOREM 3.2. *Consider any stream S, time τ, and tuple $s \in S(\tau)$ such that s satisfies all filter predicates on S. If Children$(S) = \phi$, or if for all streams $S' \in$ Children(S), $\mathcal{S}(S')$.Yes contains the child tuple of s in S', then $s \in \mathcal{S}(S)$.Yes at time τ.*

THEOREM 3.3. *Consider any stream S, time τ, and tuple $s \in S(\tau)$. If s fails a filter predicate on S, or if for some stream $S' \in$ Children(S), $\mathcal{S}(S')$.No contains the child tuple of s, then $s \in \mathcal{S}(S)$.No at time τ.*

A recursive algorithm for maintaining synopsis components (i.e., inserting and deleting synopsis tuples) as stream tuples arrive follows from Theorems 3.2 and 3.3. A procedural description of this algorithm is given in Figures 2–5. The

```
1.  /* Insert tuple s into the No synopsis component of stream S */
2.  Procedure S(S).No.InsertTuple(s) {
3.      Insert s into S(S).No;
4.      /* Propagate the effects of the insertion */
5.      For each stream R ∈ Parents(S) {
6.          /* Each tuple r in parent R joining with s goes to S(R).No */
7.          For each tuple r ∈ S(R).Unknown {
8.              if ((r → s) ≠ φ) {
9.                  Delete tuple r from S(R).Unknown;
10.                 S(R).No.InsertTuple(r); }}}}
```

Fig. 4. Procedure to insert s into $\mathcal{S}(S)$.*No*.

```
/* Insert tuple s into the Unknown synopsis component of stream S */
1.  Procedure S(S).Unknown.InsertTuple(s) {
2.      Insert s into S(S).Unknown; }
```

Fig. 5. Procedure to insert s into $\mathcal{S}(S)$.*Unknown*.

algorithm has been simplified somewhat for clarity of presentation and it is written in an object-oriented style, with the stream synopses and their components as the objects. Procedure $\mathcal{S}(S)$.*InsertTuple*(s) in Figure 2 is invoked when a new tuple s arrives in input stream S. Procedure $\mathcal{S}(S)$.*InsertTuple*(s) applies the criteria from Theorems 3.2 and 3.3 to determine whether s should be inserted in $\mathcal{S}(S)$.*Yes*, $\mathcal{S}(S)$.*No*, or $\mathcal{S}(S)$.*Unknown*, and invokes $\mathcal{S}(S)$.*Yes*.*InsertTuple*(s) (Figure 3), $\mathcal{S}(S)$.*No*.*InsertTuple*(s) (Figure 4), or $\mathcal{S}(S)$.*Unknown*.*InsertTuple*(s) (Figure 5) appropriately. (As we will describe momentarily, if S is a root stream, then Procedure $\mathcal{S}(S)$.*Yes*.*InsertTuple*(s) joins s with the *Yes* components of all other streams to produce the new tuples that are generated by the arrival of s in the query result.)

In Figures 2–5 we use the notation $s \to \mathcal{S}(R)$.*Yes* ($\mathcal{S}(R)$.*No*, $\mathcal{S}(R)$.*Unknown*) to denote the join of tuple $s \in S$ with the *Yes* (*No*, *Unknown*) synopsis component of stream $R \in Children(S)$. Note that s will join with at most one tuple in the synopsis maintained for R. The statement $(s \to \mathcal{S}(R).No) \neq \phi$ in Figure 2, where $R \in Children(S)$, therefore means that the child tuple in R of tuple $s \in S$ is present in $\mathcal{S}(R)$.*No*; likewise for $\mathcal{S}(R)$.*Yes* and $\mathcal{S}(R)$.*Unknown*.

Now consider result generation. By Definition 3.1, all tuples in the result of Q can be generated from the *Yes* synopsis components of the streams in $G(Q)$. We exploit the following two theorems.

THEOREM 3.4. *New tuples are generated in the result of Q only when a tuple is inserted into the Yes synopsis component of a stream $S \in G(Q)$ where $S \in \rho \in MinCover(G(Q))$.*

THEOREM 3.5. *The set of root streams is the only minimal cover in a (DT-shaped) join graph.*

Thus, new result tuples are generated only when a tuple s is inserted into the *Yes* synopsis component of a root stream in $G(Q)$. Our result generation algorithm joins s with the *Yes* synopsis components of all other streams to

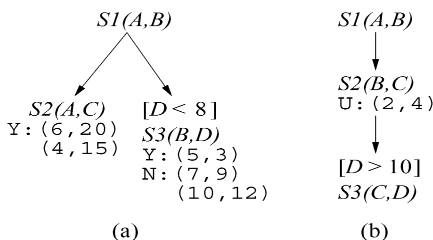

Fig. 6. Join graphs used in examples.

produce the new tuples in the result. Let us work through two examples to illustrate our algorithm so far. For presentation, all join graphs in our examples contain natural joins only.

Example 3.6. Consider a query Q having the join graph in Figure 6(a). Q contains two many-one joins, $S_1 \rightarrow S_2$ ($S_1.A = S_2.A$) and $S_1 \rightarrow S_3$ ($S_1.B = S_3.B$), and a filter predicate $D < 8$ on stream S_3. A state of the synopses also is shown in Figure 6(a): $\mathcal{S}(S_2).Yes = \{(6, 20), (4, 15)\}$, $\mathcal{S}(S_3).Yes = \{(5, 3)\}$, $\mathcal{S}(S_3).No = \{(7, 9), (10, 12)\}$, and all other synopsis components are empty. Suppose tuple $s = (6, 5)$ arrives next in S_1. Since the child tuples of s in both S_2 and S_3 are in *Yes*, s is added to $\mathcal{S}(S_1).Yes$ and result tuple $(6, 5, 20, 3)$ is emitted. Next suppose tuple $s' = (8, 10)$ arrives in S_1. The child tuple of s' in S_2 has not arrived yet. However, since the child tuple of s' in S_3 is in *No*, s' is added to $\mathcal{S}(S_1).No$.

Example 3.7. Consider the join graph and synopses in Figure 6(b). $s_2 = (2, 4)$ is in $\mathcal{S}(S_2).Unknown$ since its child tuple in S_3 has not arrived yet. Suppose tuple $s_1 = (1, 2)$ arrives in S_1. Since the child tuple of s_1 in S_2 belongs to $\mathcal{S}(S_2).Unknown$, s_1 is added to $\mathcal{S}(S_1).Unknown$. Next suppose $s_3 = (4, 12)$ arrives in S_3. Since s_3 satisfies the filter predicate on S_3, it is added to $\mathcal{S}(S_3).Yes$. As a result, s_2 is moved to $\mathcal{S}(S_2).Yes$, which further results in s_1 being moved to $\mathcal{S}(S_1).Yes$, and result tuple $(1, 2, 4, 12)$ is emitted.

3.3 Synopsis Reduction

In our basic query execution algorithm, the synopsis for a stream S simply contains each tuple of S in either *Yes*, *No*, or *Unknown*, thus the synopsis is no smaller than S itself. In this section we show how, even without k-constraints, we can reduce synopsis sizes under some circumstances. We present techniques to eliminate tuples from synopses as well as techniques to eliminate columns.

3.3.1 *Eliminating Tuples.* Our first technique is based on Theorem 3.8.

THEOREM 3.8. *Consider a join graph $G(Q)$. If a stream S forms a minimal cover for $G(Q)$, that is, $\{S\} \in MinCover(G(Q))$, then a tuple $s \in S$ inserted into $\mathcal{S}(S).Yes$ by our algorithm will not join with any future tuples to produce additional results.*

By this theorem, all result tuples using s can be generated when s is (logically) inserted into $\mathcal{S}(S).Yes$, so we need not create $\mathcal{S}(S).Yes$ at all. A common case is

when the join graph has a single root stream S, since for DT-shaped join graphs $\{S\}$ is a minimal cover.

Now consider *No* and *Unknown* components. Informally, *No* components contain tuples that will never contribute to a query result, while *Unknown* components contain tuples for which we do not yet know whether they may or may not contribute. As one simple reduction technique we can always eliminate the *No* component for root stream synopses. In fact we can always eliminate all *No* components without compromising query result accuracy, but it may not be beneficial to do so—eliminating any non-root-stream *No* tuple may have the effect of leaving some tuples in parent and ancestor *Unknown* components that may otherwise be moved to *No* components. If moved to *No* components these tuples might be discarded (if at a root) or might cause other root tuples to move to *No* and be discarded.

Formal modeling of the tradeoff between keeping non-root *No* components or eliminating them is beyond the scope of this work. The presence of k-constraints further complicates the tradeoff, although often k-constraints can be used to eliminate non-root *No* components without any detrimental effect, as we will see in Section 4.1. Hereafter we assume as a default that non-root *No* components are present except as eliminated by our k-constraint-based techniques.

3.3.2 *Eliminating Columns.* Handling queries with explicit projection does not change our basic query execution algorithm at all, and it helps us eliminate columns from synopses. Specifically, in the synopsis of a stream S we need only store those attributes of S that are involved in joins with other streams, or that are projected in the result of the query. A second column elimination technique specific to *No* synopsis components is that in $\mathcal{S}(S)$.*No* we need only store attributes involved in joins with *Parents*(S).

3.4 Sliding Windows

We explain how to extend our approach to handle tuple-based or time-based sliding windows over streams [Kang et al. 2003]. Two basic changes are required. First, a synopsis $\mathcal{S}(S)$ cannot consist simply of the three sets $\mathcal{S}(S)$.*Yes*, $\mathcal{S}(S)$.*No*, and $\mathcal{S}(S)$.*Unknown*. Now we must keep track of the order of tuples in a synopsis to maintain windows correctly, including the order of "missing" tuples that are eliminated by our algorithm. Second, when a tuple drops out of a window, we have the option of either discarding the tuple or moving it to $\mathcal{S}(S)$.*No*. The latter case may offer an opportunity to eliminate tuples in joining synopses, as with $\mathcal{S}(S)$.*No* in general (see Section 3.3.1). In our experiments we discarded tuples when they dropped out of windows, but as future work we plan to explore the alternative of placing dropped tuples in $\mathcal{S}(S)$.*No*.

3.5 Stream Characteristics and Experiments

For our experiments we developed a configurable synthetic stream generator which takes as input schema information, data characteristics, and arrival characteristics of multiple streams and generates an interleaved stream arrival order with the specified characteristics. Stream data characteristics relevant

to our experiments include the multiplicity of tuples in joins and the selectivity of filter predicates. Multiplicity of a tuple $s_2 \in S_2$ in a join $S_1 \rightarrow S_2$ is the number of S_1 tuples that join with s_2. The definition is analogous for a tuple $s_1 \in S_1$, although in this case the multiplicity has to be either 0 or 1. Except as noted otherwise, all many-one joins $S_1 \rightarrow S_2$ in our experiments have an average multiplicity of 5 for tuples in S_2, and a multiplicity of 1 for tuples in S_1. The selectivity of a filter predicate on a stream S is the percentage of tuples in S satisfying the predicate. Except as noted otherwise, all filter predicates in our experiments have an average selectivity of 50%.

We also consider stream *arrival characteristics*. For any set of streams $\rho = \{S_1, S_2, \ldots, S_n\}$ we assume a logical interleaving of the arrival of tuples in S_1, $S_2, \ldots, S_n$ and we denote this totally ordered sequence as Σ. Each tuple $s \in \Sigma$ is logically tagged with its *sequence number* in Σ, denoted $\Sigma(s)$. We define the following metrics for measuring the distance between two tuples in Σ:

—*Clustering distance*: For a pair of tuples $s_1, s_2 \in S$ with $s_1.A = s_2.A$, their clustering distance over attribute A is defined as the number of tuples $s \in S$ with $\Sigma(s_1) < \Sigma(s) < \Sigma(s_2)$ and $s.A \neq s_1.A$.

—*Scrambling distance*: For a pair of tuples $s_1, s_2 \in S$ with $s_1.A > s_2.A$ and $\Sigma(s_1) < \Sigma(s_2)$, their scrambling distance over attribute A is defined as the number of S tuples that arrive after s_1 and up to s_2 (including s_2).

—*Join distance*: For a join $S_1 \rightarrow S_2$, the join distance for a pair of joining tuples $s_1 \in S_1$ and $s_2 \in S_2$ is defined as follows: if $\Sigma(s_1) < \Sigma(s_2)$, it is the number of S_2 tuples arriving after s_1 and up to s_2 (including s_2), otherwise it is 0.

In the next three sections of the article we will consider our three constraint types in turn. For each constraint type, we provide its formal definition, identify memory-reduction techniques enabled by constraints of that type, present the monitoring algorithm, and show experimental results demonstrating memory reduction, monitoring accuracy, and the false-negative rate when the adherence parameter varies over time. Experimental results evaluating the computational overhead of each constraint type are presented in Section 7.

In our experiments we use sliding windows of size 50,000 tuples on all streams and we also compare our algorithm against SWJ (Section 1.4). Our SWJ implementation is optimized to reduce state as much as possible, but without any knowledge or exploitation of many-one joins or k-constraints. Comparing our constraint-based algorithm against SWJ identifies exactly the memory savings due to exploiting constraints. Furthermore, in Section 7 we compare the tuple-processing times of our algorithms with that of SWJ to quantify the extra computational overhead imposed by our algorithms. As we will see in Section 9, the scale of memory reduction enabled by our algorithms in real-life scenarios often improves the overall computational performance.

4. REFERENTIAL INTEGRITY CONSTRAINTS

We first consider the data stream equivalent of standard relational *referential integrity*. Referential integrity on a many-one join from relation R_1 to relation R_2 states that for each R_1 tuple there is a joining R_2 tuple. The definition

translates to streams S_1 and S_2 with a slight twist. In its strictest form, referential integrity over data streams (hereafter RIDS) on a many-one join $S_1 \rightarrow S_2$ states that the joining (child) tuple $s_2 \in S_2$ of any tuple $s_1 \in S_1$ must arrive before s_1. Unlike relational referential integrity, RIDS does not require that a joining tuple exist in S_2 for each tuple in S_1. RIDS only requires that if a joining tuple $s_2 \in S_2$ exists for a tuple $s_1 \in S_1$, then s_2 must arrive before s_1. The more relaxed *k-constraint* version states that when a tuple s_1 arrives on S_1, its joining tuple $s_2 \in S_2$ has already arrived or s_2 will arrive within k tuple arrivals on S_2. (When $k = 0$ we have the strictest form described above.)

Definition 4.1 (*RIDS*(k)). Constraint RIDS(k) holds on join $S_1 \rightarrow S_2$ if, for every tuple $s_1 \in S_1$, assuming S_2 produces a tuple s_2 joining with s_1, the join distance (Section 3.5) between s_1 and s_2 is $\leq k$.

4.1 Modified Algorithm to Exploit RIDS(k)

Consider any join graph $G(Q)$. In Section 3.3.1 we discussed that *No* synopsis components are not strictly necessary, but eliminating *No* components runs the risk of leaving tuples in parent and ancestor *Unknown* components until they drop out of their windows. RIDS constraints allow us to eliminate *No* components without this risk, using the following technique.

Consider a stream $S \in G(Q)$ and suppose for each stream $S' \in Parents(S)$ we have RIDS(k) on $S' \rightarrow S$, where the k values can differ across parents. We eliminate $\mathcal{S}(S).No$ entirely. Recall from Theorem 3.3 that our basic query execution algorithm uses $\mathcal{S}(S).No$ to determine whether a parent tuple $s' \in S'$ belongs in $\mathcal{S}(S').No$. If RIDS(k) holds with $k = 0$, then when s' arrives, its child tuple $s \in S$ must already have arrived, otherwise s' has no child tuple in S. If $s \notin \mathcal{S}(S).(Yes \cup Unknown)$ when s' arrives, then we can infer that either $s \notin S$, or s was discarded either because it belonged to $\mathcal{S}(S).No$ (which we do not keep), or because it dropped out of the window over S; s' will not contribute to any result tuple so we insert s' into $\mathcal{S}(S').No$ and proceed accordingly. If $k > 0$ and child tuple $s \notin \mathcal{S}(S).(Yes \cup Unknown)$ when s' arrives, then $s \notin S$, or s has not arrived yet, or s arrived and was discarded for the same reasons as before. We place s' in $\mathcal{S}(S').Unknown$. If k more tuples arrive on S without arrival of the child tuple s, we can infer that s will not arrive in future; we move s' to $\mathcal{S}(S').No$ and proceed accordingly.

Example 4.2. Consider the join graph and synopses shown in Figure 6(a). Suppose RIDS(1) holds on $S_1 \rightarrow S_3$ so we eliminate $\mathcal{S}(S_3).No$. Now suppose $s_1 = (4, 10)$ arrives on S_1. (s_1's child tuple $(10, 12) \in \mathcal{S}(S_3).No$ had arrived earlier and was discarded.) RIDS(1) specifies that the first S_3 tuple arriving after s_1 will be s_1's child tuple, or else either s_1 has no child tuple in S_3 or the child tuple must have arrived before s_1. Hence, s_1 can be moved to $\mathcal{S}(S_1).No$ and thus dropped (recall Section 3.3.1) as soon as the next tuple arrives in S_3.

4.2 Implementing RIDS(k) Usage

To exploit RIDS(k) for $k = k_u$ over $S' \rightarrow S$, we maintain a counter C_S of tuples that have arrived on S, and an extra sequence-number attribute

in $\mathcal{S}(S').Unknown$, denoted $C_{S' \rightarrow S}$, along with an index on this attribute that enables range scans. When a tuple $s' \in S'$ is inserted on arrival into $\mathcal{S}(S').Unknown$ because (possibly among other factors) its child tuple $s \in S$ is not present in $\mathcal{S}(S).Yes \cup \mathcal{S}(S).Unknown$, we set $s'.C_{S' \rightarrow S} = C_S$ and insert an entry for s' into the index on $C_{S' \rightarrow S}$. For each $s' \in \mathcal{S}(S').Unknown$ that joins with a newly arriving tuple $s \in S$, we delete the index entry corresponding to $s'.C_{S' \rightarrow S}$. (The join distance between s' and s is $C_S - s'.C_{S' \rightarrow S}$, which is used by the monitoring algorithm in Section 4.3.)

A periodic *garbage collection phase* uses the index on $C_{S' \rightarrow S}$ to retrieve tuples $s' \in \mathcal{S}(S').Unknown$ that have $s'.C_{S' \rightarrow S} + k_u \leq C_S$. Because of RIDS($k_u$) on $S' \rightarrow S$, $s'.C_{S' \rightarrow S} + k_u \leq C_S$ guarantees that the child tuple $s \in S$ of s' will not arrive in the future. Thus, we can infer that s' will not contribute to any result tuple, and we move s' to $\mathcal{S}(S').No$ and propagate the effects of this insertion in the usual manner. We also delete the index entry corresponding to $s'.C_{S' \rightarrow S}$.

4.3 Monitoring RIDS(*k*)

Our general goals for constraint monitoring are to inform the query execution component about changes in k for relevant constraints (recall Figure 1), not incurring too much memory or computational overhead in the monitoring process while still maintaining good estimates. If our estimate for k is higher than the actual value exhibited in the data, then our algorithm always produces correct answers but will not be as memory-efficient as possible. However, if we underestimate k then false negatives may be introduced, as discussed in Section 1.5. In addition to maintaining good estimates efficiently, we also do not want to react too quickly to changes observed in the data, since the changes may be transient and it may not be worthwhile changing query execution strategies for short-lived upward or downward "spikes."

We now describe how the monitoring component estimates k for a RIDS constraint on join $S' \rightarrow S$. As we will see, detecting decreases in k is easy, while detecting increases poses our real challenge. Let

—k_u denote the current value of k used by the query execution component; initially $k_u = \infty$;
—$k_e = c \cdot k_u$ for $c \geq 1$ denote the largest increase to k that the monitoring component is guaranteed to detect; c is a configuration parameter: a large c requires more memory but can provide more accurate results;
—p denote the probability that an additional tuple is kept to detect k values even higher than k_e; and
—W denote a window over which observed values are taken into account for adjustments to k; W is a configuration parameter that controls how quickly the monitoring component reacts to changes.

Our algorithm proceeds as follows. Logically the constraint monitor "mirrors" the RIDS-based join algorithm of Section 4.1 but using $k_e \geq k_u$ instead of k_u. In reality (and in our implementation), monitoring is integrated into query execution so we don't duplicate state or computation, but for presentation purposes let us assume they are separate. For each newly arriving tuple $s \in S$, we compute

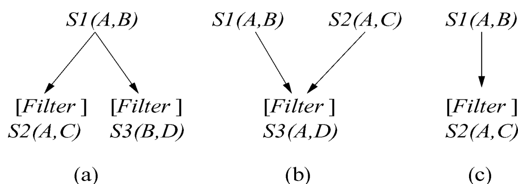

Fig. 7. Join graphs used in experiments.

the maximum join distance over all parent tuples of s in $\mathcal{S}(S').Unknown$ as described in Section 4.2. If the maximum observed join distance for tuples in S is $k' < k_u$ for the last W tuple arrivals in S, then we set $k_u = k'$ (and consequently $k_e = c \cdot k'$) and notify the query execution component accordingly.

Increases in k are more difficult for two reasons: (1) in order to detect increases, we need more data than would otherwise be kept for query execution; (2) unlike decreases, increases introduce false negatives. As part of (1), we ensure that any tuple in $\mathcal{S}(S').Unknown$ that is moved to $\mathcal{S}(S').No$ by the execution algorithm because of RIDS(k_u) is logically retained in $\mathcal{S}(S').Unknown$ until the tuple can be moved because of RIDS($k_e = c \cdot k_u$), $c \geq 1$. This step ensures that an increase in k up to k_e will be detected at the potential cost of lower memory reduction than permitted by k_u. In addition, each tuple $s' \in \mathcal{S}(S').Unknown$ is, with probability p, retained until s' drops out of S''s window specified in the query, if s' would otherwise be discarded because of RIDS(k_u). Effectively we are sampling in order to detect increases in k to values even higher than k_e. To address issue (2), as soon as an increase in k is detected, we conservatively set $k_u = \infty$ and notify the query execution component, so it stops using the constraint and possibly generating additional false negatives. (Recall from Section 1.5 that the query execution component ignores constraints with k_u values higher than some threshold.) The value of k_u will be reset by decrease detection after W more tuples have arrived on S. We set W conservatively to a large value in order to reduce oscillations in k_u, thereby reducing the chances of generating false negatives.

We have taken a conservative approach to detecting and handling increases in k, in order to ensure that we retain high query result accuracy. In the future we may extend our algorithms to exploit a potential memory-accuracy tradeoff here: we can be less conservative and lower the memory requirement if the application is willing to accept the possibility of some (temporary) query result inaccuracy.

4.4 Experimental Analysis for RIDS(*k*)

For the RIDS experiments we used the join graph in Figure 7(a). Figure 8 shows the memory reduction achieved by our query execution algorithm for different values of k. The x-axis shows the total number of tuples processed across all streams and the y-axis shows the total memory used, including synopsis size and monitoring overhead. We show plots for $k \in \{0, 5000, 10000, 20000\}$ and for SWJ. For each $k = k'$, we generated synthetic data for streams S_1, S_2, and S_3 with join distances distributed uniformly in $[0, \ldots, k']$ so that RIDS(k') always

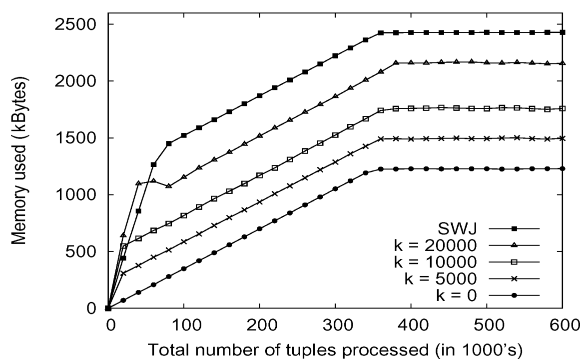

Fig. 8. Memory reduction using RIDS(k).

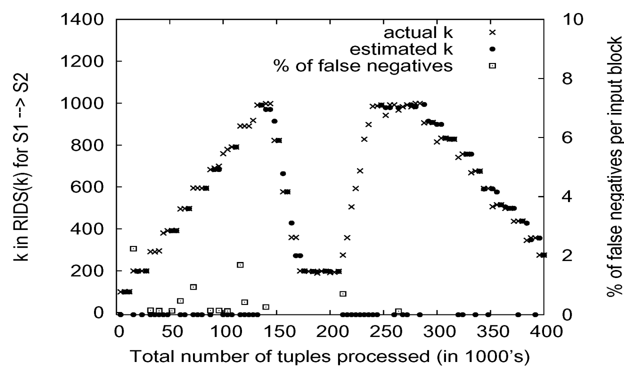

Fig. 9. Monitoring RIDS(k).

holds and RIDS(k'') does not hold for any $k'' < k'$. Note that the adherence is not varied over time in this experiment. All tuple sizes are 24 bytes each in this experiment and all subsequent experiments.

Recall that RIDS(k) on $S_1 \rightarrow S_2$ and $S_1 \rightarrow S_3$ eliminates $\mathcal{S}(S_2).No$ and $\mathcal{S}(S_3).No$, and prevents tuples from accumulating in $\mathcal{S}(S_1).Unknown$. $\mathcal{S}(S_1).Yes$ and $\mathcal{S}(S_1).No$ are eliminated by default (Section 3.3.1). On the other hand, SWJ stores a full window of tuples for S_1, and all tuples in the windows over S_2 and S_3 that pass the respective filter predicates. The total synopsis size stabilizes around 350,000 tuples once all windows get filled so that each newly arriving tuple will displace the oldest tuple in the respective window. (S_1's window fills up around 70,000 tuples.) Figure 8 shows the increase in memory overhead as the adherence to RIDS decreases, that is, as k increases.

Figure 9 shows the performance of the complete k-Mon framework using RIDS when k varies over time. The left y-axis shows the value of k in RIDS(k) and the right y-axis shows the percentage of false negatives per block of 4000 input stream tuples. Parameters c, p, and W for the monitoring algorithm were set to 1, 0.01, and 500, respectively. The two plots using the left y-axis show that the k estimated by our monitoring algorithm tracks the actual k in the data very closely. Five different types of variation in k are shown in Figure 9: no variation, gradual increase, gradual drop, quick increase, and quick drop.

Points of the "estimated k" plot on the x-axis itself indicate periods when $k_u = \infty$ and the constraint is not being used. Note that the percentage of false negatives remains close to zero except during periods of increase in k, and even then it remains reasonably low (<2%). (For clarity, only the nonzero false negative percentages are shown here and in subsequent experiments.)

5. CLUSTERED-ARRIVAL CONSTRAINTS

In its strictest form, a clustered-arrival constraint on attribute A of a stream S specifies that tuples having duplicate values for A arrive at successive positions in S. The relaxed k-constraint version (hereafter CA(k)) specifies that the number of S tuples with non-v values for attribute A between any two S tuples with A equal to v is no greater than k. As always, $k = 0$ yields the strictest form of the constraint. Note that CA(k) holds over a single stream, in contrast to RIDS(k) which holds over a join of two streams.

Definition 5.1 (*CA*(k)). Constraint CA(k) holds on attribute A in stream S if, for every pair of tuples $s_1, s_2 \in S$ with $s_1.A = s_2.A$, the clustering distance over A between s_1 and s_2 (Section 3.5) is no greater than k.

5.1 Modified Algorithm to Exploit CA(k)

The benefits of RIDS(k) constraints are focused on the reduction or elimination of *No* and *Unknown* synopsis components. CA(k) constraints help eliminate tuples from all three components. Elimination of tuples from *Yes* and *Unknown* components is based on the following theorem.

THEOREM 5.2. *Let S be a stream in a join graph $G(Q)$ with Parents$(S) = \{S_1, S_2, \ldots, S_n\}$. A tuple $s \in S$ will not join with any future tuples to produce result tuples if the following conditions are satisfied for some $\rho \subseteq \{S_1, S_2, \ldots, S_n\}$:*

C1: $\rho \in MinCover(G(Q))$.

C2: For all $S_i \in \rho$, no tuple in the current $\mathcal{S}(S_i)$.Unknown component joins with s.

C3: For all $S_i \in \rho$, no future tuple in S_i can join with s.

Each $\rho \subseteq \{S_1, \ldots, S_n\}$ that forms a minimal cover of $G(Q)$ can be identified at query compilation time. For each such ρ, condition C2 in Theorem 5.2 can be evaluated at a given time by joining s with the contents of $\mathcal{S}(S_i).Unknown$. A CA(k) constraint on any one of S_i's join attributes in $S_i \rightarrow S$ for each $S_i \in \rho$ is sufficient to evaluate condition C3, as follows. Let $S_i.A = S.B$ be a predicate in the $S_i \rightarrow S$ join, with CA(k) on $S_i.A$. Once tuple s_1 arrives on S_i with $s_1.A = v$, after $k + 1$ new tuples with $A \neq v$ arrive on S_i, no future S_i tuple can have $A = v$. That is, no future tuple will join with a tuple $s \in S$ with $s.B = v$.

When we determine that a tuple $s \in \mathcal{S}(S)$ satisfies conditions C1–C3 in Theorem 5.2, s can be eliminated. Also, any tuple in $\{S_1, \ldots, S_n\}$ that joins with s can be eliminated from whatever synopsis component it resides in. Recall from Section 3.3.1 that tuples in the *No* synopsis component of a stream S are used only by parents of S to move tuples from *Unknown* to *No*. Therefore, a tuple $s \in \mathcal{S}(S).No$ can be removed if no future tuple in any stream $S' \in Parents(S)$ can

join with s. CA(k) constraints can be used to identify such tuples as explained above.

Example 5.3. Consider again the join graph and synopses in Figure 6(a). Suppose CA(1) holds on attribute $S_1.B$ and consider the following sequence of tuple arrivals in S_1: $(6, 5)$, $(8, 8)$, $(4, 5)$, $(11, 10)$. After these arrivals, logically $\mathcal{S}(S_1).Yes = \{(6, 5), (4, 5)\}$, logically $\mathcal{S}(S_1).No = \{(11, 10)\}$, $\mathcal{S}(S_1).Unknown = \{(8, 8)\}$, and result tuples $(6, 5, 20, 3)$ and $(4, 5, 15, 3)$ are emitted (recall we do not store $\mathcal{S}(S_1).Yes$ or $\mathcal{S}(S_1).No$ in this case). On $S_1.B$ two non-5 values have appeared after the first 5, so by the CA(1) constraint no future tuple $s \in S_1$ will have $s.B = 5$. Furthermore, since no tuple in $\mathcal{S}(S_1).Unknown$ has $B = 5$, the tuple $(5, 3) \in \mathcal{S}(S_3).Yes$ cannot contribute to any future result tuples and can be eliminated.

5.2 Implementing CA(*k*) Usage

We use the criteria in Theorem 5.2 to delete tuples from the synopsis components of a stream S if some $\rho \subseteq Parents(S)$ is a minimal cover and, for each $S' \in \rho$, we have a CA(k) constraint on any one of S''s join attributes in $S' \rightarrow S$. For each $S' \in \rho$ we maintain an auxiliary data structure, denoted *CA-Aux*($S'.A$), where $S'.A$ is a join attribute in $S' \rightarrow S$ on which CA(k) holds with $k = k_u$. We also maintain a counter $C_{S'}$ of tuples that have arrived on S'. Furthermore, we maintain a bitmap of size $|\rho|$ per tuple $s \in \mathcal{S}(S)$, with one bit per $S' \in \rho$ indicating whether s satisfies Conditions C2 and C3 in Theorem 5.2 for S'.

CA-Aux($S'.A$) contains elements (v, C_v), where v is an A value that arrived in S' and C_v is $C_{S'}$ minus the number of tuples with non-v values of A that arrived in S' after the very first tuple in S' with $A = v$. A hash index is maintained on the A values in *CA-Aux*($S'.A$). Also, the elements in *CA-Aux*($S'.A$) are linked together in sorted order of C_v values using a doubly-linked list.

When a tuple $s' \in S'$ arrives, the value of $s'.A$ is looked up in the hash index on *CA-Aux*($S'.A$). If an element $(v = s'.A, C_v)$ is present in *CA-Aux*($S'.A$), then we increment the corresponding C_v value by 1. (The maximum clustering distance so far over $S'.A$ between any two tuples with $S'.A = v$ is $C_{S'} - C_v$, which is used by the CA(k) monitoring algorithm in Section 5.3.) Otherwise, we insert the element $(v = s'.A, C_{S'})$ into *CA-Aux*($S'.A$). Both steps require very limited maintenance of the doubly linked list linking the elements in sorted order of C_v values.

A periodic garbage collection phase uses the doubly linked list to retrieve the elements (v, C_v) with $C_v < C_{S'} - k_u$. For these elements CA(k_u) guarantees that no future tuple in S' will have $s'.A = v$. We look up $\mathcal{S}(S').Unknown$ to determine whether any tuple $s'' \in \mathcal{S}(S').Unknown$ has $s'' = v$. If so, we skip v as per Condition C2 in Theorem 5.2. Otherwise, we look up $\mathcal{S}(S)$ to find whether any tuple $s \in \mathcal{S}(S)$ has $s = v$. If not, we delete (v, C_v) from *CA-Aux*($S'.A$). Otherwise, we set the bit (initially false) corresponding to S' in s's bitmap to indicate that s satisfies Conditions C2 and C3 in Theorem 5.2 for S'. If the bits corresponding to all streams in ρ are set in s, we delete s and all tuples in parent and ancestor streams of S that join with s. Furthermore, we delete (v, C_v) from *CA-Aux*($S'.A$).

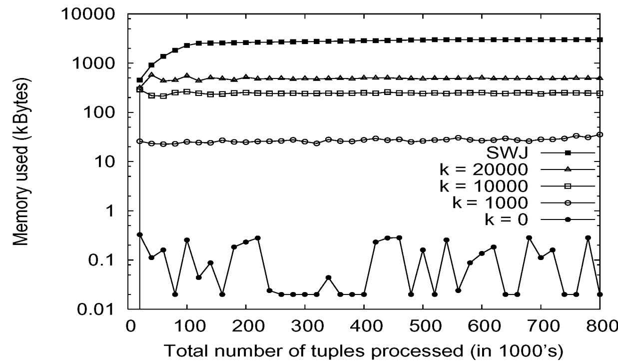

Fig. 10. Memory reduction using CA(k).

5.3 Monitoring CA(k)

Monitoring CA(k) can be done very similarly to monitoring RIDS(k) as described in Section 4.3, except now we track clustering distances between tuples in the same stream instead of join distances across streams as in RIDS(k). With reference to Theorem 5.2, suppose we are monitoring CA(k) on join attribute A in stream $S_i \in Parents(S)$. As with RIDS, our monitoring algorithm mirrors query execution using $k_e = c \cdot k_u$. In reality, the two are combined. Clustering distances can be tracked during query execution as described in Section 5.2. If the maximum clustering distance over $S_i.A$ is observed as $k' < k_u$ for the last W tuple arrivals in S_i, then we set $k_u = k'$. We ensure that the *CA-Aux*($S_i.A$) entry corresponding to a tuple $s \in S$ that would normally be discarded because of CA(k_u) on $S_i.A$ is retained until s can be discarded because of CA(k_e). As with RIDS, this step guarantees detection of increases in k within k_e. For detecting increases beyond k_e, with probability p we retain the *CA-Aux*($S_i.A$) entry corresponding to a tuple $s \in S$, which would normally be discarded because of CA(k_u), until s logically drops out of S's window specified in the query. As with RIDS, we conservatively set $k_u = \infty$ on increase detection and the value is reset by decrease detection after W more arrivals.

5.4 Experimental Analysis for CA(k)

For the CA experiments, we used the join graph shown in Figure 7(b). Figure 10 shows the memory reduction achieved by our query execution algorithm for different values of k. (Note the log scale on the y-axis in Figure 10.) We generated synthetic data for streams S_1, S_2, and S_3 with different arrival orders conforming to CA(k) on both $S_1.A$ and $S_2.A$. Maximum clustering distances for distinct values of $S_1.A$ and $S_2.A$ are distributed uniformly in $[0, \ldots, k]$. The adherence is not varied over time in this experiment. To isolate the effect of the CA(k) constraints, we generated the arrival order of tuples in S_3 to satisfy RIDS(0) on $S_1 \rightarrow S_3$ and $S_2 \rightarrow S_3$. However, the RIDS constraints are not used explicitly to reduce synopsis sizes. CA(k) on the join attributes in S_1 and S_2 enables the removal of tuples from $\mathcal{S}(S_3).Yes$, $\mathcal{S}(S_3).No$, $\mathcal{S}(S_1).Yes$, and $\mathcal{S}(S_2).Yes$. Although

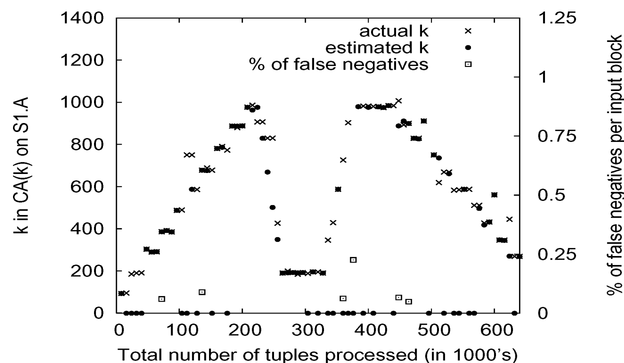

Fig. 11. Monitoring CA(k).

RIDS(0) is not used, its presence in the input streams keeps the *Unknown* components empty. Hence the total memory overhead for the CA(k) algorithm reaches its peak much before all windows fill up at around 550,000 tuples when the memory overhead of SWJ stabilizes. (Windows over S_1 and S_2 fill up around 110,000 tuples.)

Figure 11 shows the performance of k-Mon using CA when k varies over time. For this experiment, parameters c, p, and W for the monitoring algorithm were set to 1.2, 0.01, and 1000, respectively. Notice again that the k estimated by our monitoring algorithm tracks the actual k closely so the number of false negatives produced by our execution component remains close to zero. Recall from Section 4.4 that points of the "estimated k" plot on the x-axis indicate periods when $k_u = \infty$ and the constraint is not being used.

6. ORDERED-ARRIVAL CONSTRAINTS

In its strictest form, an ordered-arrival constraint on attribute A of a stream S specifies that the value of A in any tuple $s \in S$ will be no less than the value of A in any tuple that arrived before s, that is, the stream is sorted by A. (We assume ascending order; obviously descending order is symmetric.) The relaxed k-constraint version (hereafter OA(k)) specifies that for any tuple $s \in S$, S tuples that arrive at least $k+1$ tuples after s will have a value of A that is no less than $s.A$. As always, $k = 0$ is the strictest form, and like CA(k), an OA(k) constraint holds over a single stream.

Definition 6.1 (*OA*(k)). Constraint OA(k) holds on attribute A in stream S if for every pair of tuples $s_1, s_2 \in S$ with $\Sigma(s_1) < \Sigma(s_2)$ and $s_1.A > s_2.A$, the scrambling distance between s_1 and s_2 (Section 3.5) is no greater than k.

OA(k) is useful on join attributes, and we use it differently depending whether the constraint is on the parent stream or the child stream in a many-one join. Thus, we distinguish two classes of OA(k): *ordered-arrival of parent stream* (hereafter OAP(k)) and *ordered-arrival of child stream* (hereafter OAC(k)). The constraint monitoring algorithm is the same for both classes.

6.1 Modified Algorithm to Exploit OAP(k)

Like CA(k), OAP(k) constraints on the join attributes in streams $\{S_1, S_2, \ldots, S_n\}$ can be used to evaluate condition C3 in Theorem 5.2. Let $S_i.A = S.B$ be a predicate in the $S_i \rightarrow S$ join. If OAP(k) holds on $S_i.A$, once k S_i tuples have arrived after a tuple $s_i \in S_i$, no future S_i tuple can have $A < s_i.A$. That is, no future tuple will join with tuple $s \in S$ if $s.B < s_i.A$. Hence, an OAP(k) constraint on any one of S_i's join attributes in $S_i \rightarrow S$ for each $S_i \in \rho$ is sufficient to evaluate condition C3 in Theorem 5.2. Note an advantage of OAP(k) constraints over CA(k) constraints: in the absence of RIDS, OAP(k) constraints can always eliminate dangling tuples in S (tuples that never join), while CA(k) cannot. The algorithm can be extended in a straightforward manner to the case where a mix of CA(k) and OAP(k) constraints hold over streams in ρ in Theorem 5.2.

6.2 Implementing OAP(k) Usage

We use the criteria in Theorem 5.2 to delete tuples from the synopsis components of a stream S if some $\rho \subseteq \mathit{Parents}(S)$ is a minimal cover and, for each $S' \in \rho$, we have an OAP(k) constraint on one of S''s join attributes in $S' \rightarrow S$. Let $S' \in \rho$ and let $S'.A$ be a join attribute in $S' \rightarrow S$ on which OAP(k) holds with $k = k_u$. Also, let max denote the maximum value of A seen so far on S'. We maintain a sliding window $[max_1, \ldots, max_{k_u+1}]$ containing the values of max after each of the last $k_u + 1$ arrivals in S', with max_1 being the most recent value. OAP(k_u) guarantees that no future tuple $s' \in S'$ will have $s'.A < max_{k_u+1}$.

In addition, for each $S' \in \rho$ we maintain an equiwidth histogram, denoted $hist(S'.A)$, on the values of $S'.A$ in $\mathcal{S}(S').Unknown$. The histogram is implemented as a circular buffer that can grow and shrink dynamically. Whenever a tuple $s' \in S'$ is inserted into or deleted from $\mathcal{S}(S').Unknown$, the count of the bucket in $hist(S'.A)$ containing $s'.A$ is incremented or decremented, respectively. Whenever the count of the first bucket in $hist(S'.A)$, that is, the bucket corresponding to the smallest values, drops to 0, we delete the bucket if its upper bound is $< max_{k_u+1}$. Notice that any tuple $s' \in S'$ inserted into $\mathcal{S}(S').Unknown$ will have $s'.A \geq max_{k_u+1}$.

A periodic garbage collection phase retrieves the lower bound of the first bucket in $hist(S'.A)$, denoted A_{lo}. If $S.B$ is an attribute in S involved in a join with $S'.A$, then any tuple $s \in S$ with $s.B < A_{lo}$ will not join with any tuple $s' \in \mathcal{S}(S').Unknown$. Thus, s satisfies Condition C2 in Theorem 5.2. Also, if $s.B < max_{k_u+1}$, then s will not join with any future tuple in S', satisfying Condition C3 in Theorem 5.2. We use an index that enables range scans on $S.B$ in $\mathcal{S}(S)$ to retrieve tuples $s \in S$ that have $s.B$ less than the minimum of A_{lo} and max_{k_u+1}. For each retrieved tuple s, we set the bit corresponding to S' in a bitmap maintained with s (similar to CA(k) usage in Section 5.2) to indicate that s satisfies Conditions C2 and C3 in Theorem 5.2 for S'. (We use the index to scan $\mathcal{S}(S)$ in nonincreasing order of $S.B$ values so that we do not access tuples that were already marked in an earlier garbage collection step.) If the bits corresponding to all streams in ρ are set in s, we delete s and all tuples in parent and ancestor streams of S that join with s.

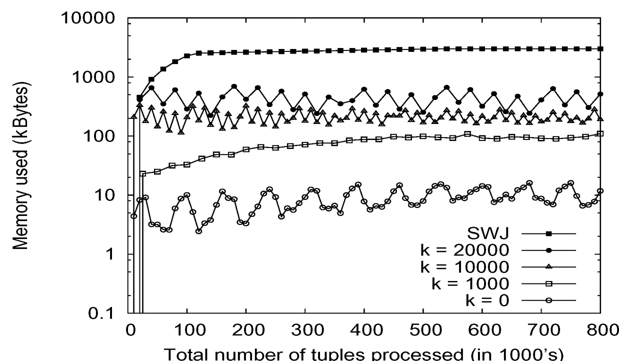

Fig. 12. Memory reduction using OAP(k).

We have also experimented with other ways of implementing OAP(k) usage. The technique described here gave us the best tradeoff between memory reduction and computation time.

6.3 Monitoring OA(k)

Consider monitoring k for OA on attribute A in stream S. We use a different technique than that used for RIDS and CA, although we still integrate monitoring with query execution to avoid duplicating state and computation. As mentioned in Section 6.2, we maintain a sliding window $[max_1, \ldots, max_{k_u+1}]$ containing the maximum value of A after each of the last $k_u + 1$ arrivals, with max_1 being the most recent value. When a tuple $s \in S$ arrives, we compute the current maximum scrambling distance d_s involving tuple s as follows. If $s.A \geq max_1$, then $d_s = 0$ since $s.A \geq$ all values seen so far. Otherwise, we perform a binary search on the window of max values to find i such that $max_{i+1} \leq s.A < max_i$. If such an i exists, then $d_s = i \leq k_u$, otherwise $d_s > k_u$.

Consider decreases to k first. If there is a $k' < k_u$ such that all d_s values are $\leq k'$ over the last W tuple arrivals in S, then we set $k_u = k'$ and notify the execution component. We have an increase when $d_s > k_u$. As with RIDS and CA, we set $k_u = \infty$, notify the query execution component, and allow k_u to be reset by decrease detection. Note that when $k_u = \infty$, the window of max values grows in size, but it can only grow indefinitely if k values increase indefinitely as well. (In practice we do not let the window grow beyond a threshold.) Finally, if we wish to speed up "convergence" of the new k value after an increase, we can maintain $k_e = c \cdot k_u$ elements in our window of max values for some $c > 1$.

6.4 Experimental Analysis for OAP(k)

For the OAP experiments we used the same join graph as for CA (Figure 7(b)). Figure 12 shows the memory reduction achieved by the query execution component for different values of k. The data generation was similar to that for CA except here we adhere to OAP(k) on $S_1.A$ and $S_2.A$. Maximum scrambling distances for distinct values of $S_1.A$ and $S_2.A$ are distributed uniformly in $[0, \ldots, k]$. In Figure 12, the total memory requirement for each value of k varies

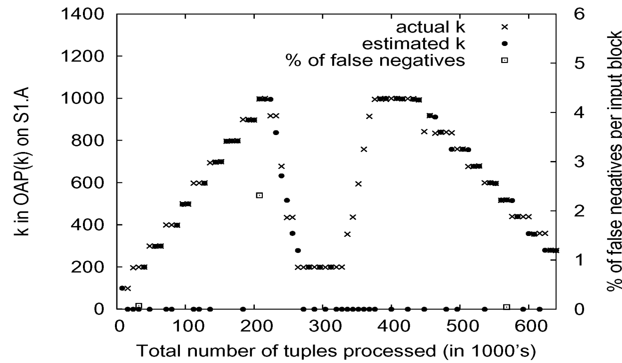

Fig. 13. Monitoring OAP(k).

around some fairly fixed value. The scale of variation is determined by the degree of out-of-order arrival in the streams, which in turn is proportional to k. Hence higher values of k cause larger variation. (Note the log scale on the y-axis in Figure 12.) Also, as adherence to OAP decreases, that is, as k increases, the peak memory overhead increases.

Figure 13 shows the performance of k-Mon using OAP when k varies over time. Parameters c and W for the monitoring algorithm were set to 1.2 and 1000, respectively. The number of false negatives produced remains close to zero except during one period of increasing k where the percentage of false negatives goes up to 2.3%.

6.5 Modified Algorithm to Exploit OAC(k)

Recall that an OAC(k) constraint is an OA(k) constraint holding on a join attribute in a child stream in a many-one join. The treatment of OAC(k) is similar to RIDS(k). OAC(k) constraints allow us to eliminate *No* components without running the risk of leaving tuples in parent or ancestor *Unknown* components until they drop out of their windows (Section 3.3.1). Recall from Section 4 that RIDS(k) constraints are used for the same purpose.

Consider a join graph $G(Q)$ and a stream $S \in G(Q)$. Suppose for each stream $S' \in Parents(S)$ we have OAC(k) on $S.A$, where $S'.B = S.A$ is a predicate in the $S' \rightarrow S$ join. Then we can eliminate $\mathcal{S}(S).No$ entirely. Recall that our basic query processing algorithm uses $\mathcal{S}(S).No$ to determine whether a parent tuple $s' \in S'$ belongs to $\mathcal{S}(S').No$. With an OA(k) constraint on $S.A$, we can continuously maintain a value $S.A_{lo}$ such that no future tuple $s \in S$ will have $s.A < S.A_{lo}$. For a tuple $s' \in S'$ with $s'.B < S.A_{lo}$, either s''s child tuple $s \in S$ has arrived, or it will never arrive. Hence, the absence of $\mathcal{S}(S).No$ will not leave tuples blocked in $\mathcal{S}(S').Unknown$ indefinitely.

Example 6.2. Consider the join graph and synopses in Figure 6(a). Suppose OAC(2) holds on $S_3.B$, so we eliminate $\mathcal{S}(S_3).No$, and suppose the S_3 tuples shown in the figure arrived in the order $(7, 9), (5, 3), (10, 12)$. By OAC(2), $S_3.B_{lo} = 7$. Suppose a tuple $s_1 = (6, 4)$ arrives in S_1. Since $s_1.B < S_3.B_{lo}$ and $\mathcal{S}(S_3).(Yes \cup Unknown)$ does not contain s_1's child tuple in S_3, either s_1's child

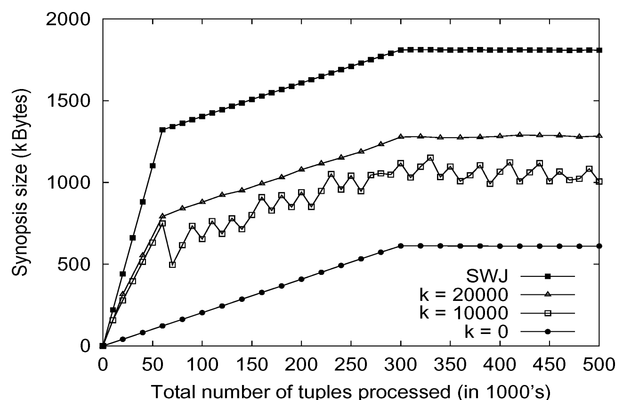

Fig. 14. Memory reduction using OAC().

tuple was eliminated as part of $\mathcal{S}(S_3).No$ or s_1 is a dangling tuple. In either case, logically $s_1 \in \mathcal{S}(S_1).No$ and it can be eliminated.

6.6 Implementing OAC(*k*) Usage

We exploit OAC(k) constraints to eliminate $\mathcal{S}(S).No$ if for each stream $S' \in Parents(S)$ we have OAC(k) on $S.A$, where $S.A$ is an attribute in the $S' \rightarrow S$ join. For simplicity, let us assume that all streams $S' \in Parents(S)$ are involved in a join with S on the same attribute $S.A$, and OAC(k) holds on $S.A$ for $k = k'$. It is easy to extend to the case when more than one attribute in S is involved in joins with the parent streams, and OAC(k) constraints hold on these attributes. We maintain a sliding window containing the values of $S.A$ in the last $k' + 1$ tuples in S. If we denote the values in the window as $W[0], W[1], \ldots, W[k']$, with $W[k']$ being the most recent value, OAC(k') guarantees that no future tuple $s \in S$ will have $s.A < W[0]$.

For each stream $S' \in Parents(S)$, we maintain an index enabling range scans on the attribute $S'.B$ involved in a join with $S.A$. During each garbage collection phase, we use this index to retrieve tuples $s' \in S'$ with $s'.B < W[0]$, which guarantees that the child tuple of s' in S will not arrive in the future. After retrieving s', we delete the entry corresponding to s' from this index. We then join s' with $\mathcal{S}(S).Yes \cup \mathcal{S}(S).Unknown$. (This join is a lookup on the hash index on $S.A$ that is used for regular join processing.) If the child tuple is not found, we move s' to $\mathcal{S}(S').No$ and propagate the effects of this insertion as listed in Procedure $\mathcal{S}(S).No.InsertTuple(s)$ (Figure 4). If the tuple is found, nothing needs to be done.

6.7 Experimental Analysis for OAC(*k*)

For the OAC experiments we used the join graph shown in Figure 7(c), and the results are shown in Figure 14. Streams S_1 and S_2 were generated with different arrival orders conforming to OAC(k) on $S_2.A$ for varying values of k. Maximum scrambling distances for distinct values of $S_2.A$ are distributed uniformly in $[0, \ldots, k]$. OAC(k) on $S_2.A$ eliminates $\mathcal{S}(S_2).No$ completely. The

Table I. Tuple-Processing Time (microseconds) for Different k

Algorithm	$k = 0$	$k = 1000$	$k = 5000$	$k = 10000$	$k = 20000$
RIDS	20/24	40/26	43/28	45/28	46/28
CA	22/20	24/21	25/23	27/23	28/24
OAP	21/21	23/21	24/22	25/23	27/25
OAC	20/18	20/18	20/18	21/18	22/18

sharp drop in synopsis size for $k = 10{,}000$ in Figure 14 around 60,000 tuples is because the total number of tuples in S_2 crosses 10,000 at this point and the system starts eliminating tuples from S_1 that arrived after their child tuple was dropped from $\mathcal{S}(S_2).No$. The corresponding drop for $k = 20{,}000$ is less dramatic because many of the tuples that could have been dropped have already dropped out of the window over S_1 (recall from 3.4 that we discard tuples that drop out of their respective windows). Figure 14 shows the increase in memory overhead as the adherence to OAC decreases, that is, as k increases.

7. COMPUTATIONAL OVERHEAD

The experiments in Sections 4.4, 5.4, and 6.4 demonstrate the effectiveness of our k-constraint approach in reducing the memory requirement compared to SWJ. In Table I we show the per-tuple processing time for each of our algorithms for different values of k, along with SWJ, which has no computational overhead apart from evaluating the join itself. Each entry in Table I is of the form X/Y, where X is the per-tuple processing time for k-Mon, which includes monitoring and all other overhead specific to k-Mon, and Y is the corresponding time for SWJ. These values were computed from the experiments in Figures 8, 10, and 12 from the total time to process a million tuples after the system had stabilized. Each value is the median of five independent runs. All times are in microseconds. The throughput achieved in our experiments was on the order of 20,000–50,000 tuples/s on a 700-MHz Linux machine with 1024-kB processor cache and 2-GB memory.

The computational overhead of our approach when compared to SWJ is low for the CA, OAP, and OAC algorithms, and it remains fairly stable as k increases. However, the overhead for RIDS increases with k, going to about 64% at k =20,000. Although 64% additional overhead per tuple may sound excessive, it can still be a viable approach if the data stream system has excess processor cycles but not enough memory to support its workload [Das et al. 2003; Kang et al. 2003].

8. CONSTRAINT COMBINATION

In Sections 4–6 we discussed constraint types RIDS, CA, OAP, and OAC, in each case exploiting constraints of that type without considering the simultaneous presence of constraints of another type. In this section we briefly explore the interaction of multiple simultaneous constraints of different types. To begin, we review the synopsis components that may be reduced or eliminated by the four constraint types independently, summarized in Table II.

Table II. Summary of Synopsis Reductions

k-constraint for $S_1 \to S_2$	Can reduce or eliminate
Default	$\mathcal{S}(S_1)$.*Yes* if $\{S_1\}$ is a cover, $\mathcal{S}(S_1)$.*No* if S_1 is a root stream
RIDS	$\mathcal{S}(S_2)$.*No*, $\mathcal{S}(S_1)$.*Unknown*
CA on $S_1.A$	$\mathcal{S}(S_1)$.*Yes*, nondangling tuples in $\mathcal{S}(S_2)$.(*Yes* $\cup$ *No* $\cup$ *Unknown*)
OAP on $S_1.A$	$\mathcal{S}(S_1)$.*Yes*, $\mathcal{S}(S_2)$.(*Yes* $\cup$ *No* $\cup$ *Unknown*)
OAC on $S_2.A$	$\mathcal{S}(S_2)$.*No*, $\mathcal{S}(S_1)$.*Unknown*

It is never the case that combining constraints of different types results in a situation where we can eliminate fewer synopsis tuples than the union of the tuples eliminated by considering the constraints independently. Furthermore, in some cases combining constraints allows us to eliminate more tuples, as seen in the following example.

Example 8.1. Consider the join graph and synopses in Figure 6(a). Suppose CA(0) holds on $S_1.A$ and OAC(0) holds on $S_3.B$. Consider the following sequence of tuple arrivals in S_1: (4, 5), (6, 8), (3, 13). Let us consider three different situations: (i) only the CA constraint is used; (ii) only the OAC constraint is used; (iii) both constraints are used simultaneously. All three situations infer (4, 5) to be in $\mathcal{S}(S_1)$.*Yes* and drop it after result tuple (4, 5, 15, 3) is emitted. When only CA(0) on $S_1.A$ is used, (6, 8) ends up in $\mathcal{S}(S_1)$.*Unknown* since its child tuple in S_3 has not arrived. CA(0) infers that $(4, 15) \in \mathcal{S}(S_2)$.*Yes* will not produce any future result tuples and eliminates it. But it is unable to eliminate $(6, 20) \in \mathcal{S}(S_2)$.*Yes* because parent tuple (6, 8) is in $\mathcal{S}(S_1)$.*Unknown*. OAC(0) (which eliminates $\mathcal{S}(S_3)$.*No*) infers (6, 8) to be in $\mathcal{S}(S_1)$.*No* since a value 10 has arrived in $S_3.B$ and no tuple in $\mathcal{S}(S_3)$.(*Yes* $\cup$ *Unknown*) has $B = 8$, and eliminates (6, 8). But OAC(0) on $S_3.B$ cannot eliminate any tuple in $\mathcal{S}(S_2)$.*Yes*. Now consider what happens when both constraints are used simultaneously. Independently, OAC(0) will eliminate $(6, 8) \in S_1$, and CA(0) will eliminate $(4, 15) \in S_2$, as explained above. Additionally, since no tuple in $\mathcal{S}(S_1)$.*Unknown* has $A = 6$, CA(0) eliminates $(6, 20) \in \mathcal{S}(S_2)$.*Yes*, which it was unable to eliminate earlier. Using both constraints simultaneously thus gives better synopsis reduction than the union of their independent reductions.

In Figure 15 we report an experimental result showing the effect of combining CA and OAC constraints for the join graph in Figure 7(a). We generated synthetic streams S_1, S_2, and S_3 with CA(0) on $S_1.A$ and OAC(5000) on $S_3.B$. On average, 25% of the tuples in S_1 have no joining (child) tuple in S_3. Using both constraints simultaneously gives the best memory reduction in Figure 15. In terms of computational overhead, the per-tuple processing time is 23 μs when OAC alone is used, 25 μs when CA alone is used, and 28 μs when both constraints are used simultaneously.

However, there is an interesting subtlety when we mix multiple constraint types. Although exploiting multiple constraints will never decrease the number of tuples that can be eliminated from synopses, in certain cases it can increase the length of time that tuples remain in synopses before they are eliminated, as seen in the following example.

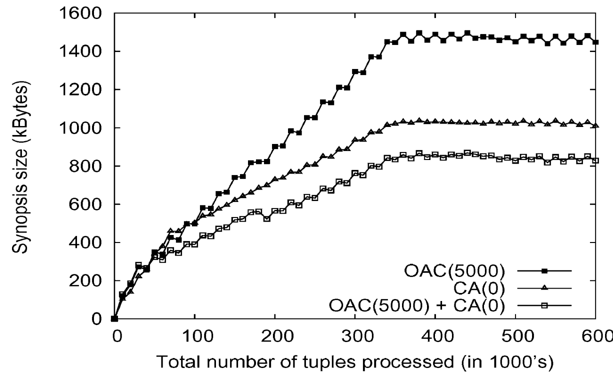

Fig. 15. Effect of combining CA and OAC.

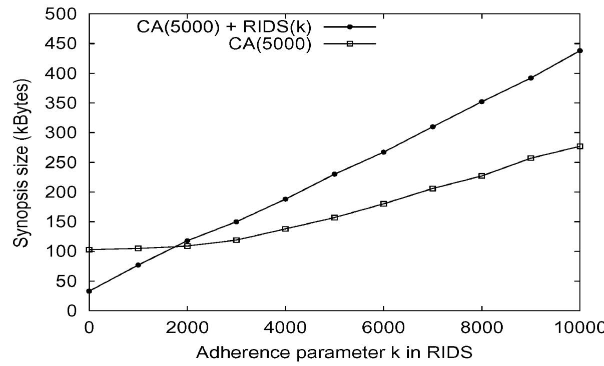

Fig. 16. Effect of combining CA and RIDS.

Example 8.2. Consider the join graph and synopses in Figure 6(a). Suppose CA(0) holds on $S_1.B$ and RIDS(3) holds on the $S_1 \rightarrow S_3$ join. Let us consider two different situations: (i) only the CA constraint is used; (ii) the CA and RIDS constraints are used simultaneously. Consider the following sequence of tuple arrivals in S_1: $(6, 10), (4, 10), (8, 8)$. When only CA(0) on $S_1.B$ is used, $(6, 10)$ and $(4, 10)$ join with their child tuple $(10, 12) \in \mathcal{S}(S_3).No$ and get dropped. Also, $(10, 12) \in \mathcal{S}(S_3).No$ is eliminated since CA(0) infers that no future tuple in S_1 will join with it. If RIDS(3) is also used, $(10, 12) \in S_3$ would have been dropped on arrival since $\mathcal{S}(S_3).No$ is not stored. Thus, $(6, 10)$ and $(4, 10)$ end up in $\mathcal{S}(S_1).Unknown$ on arrival. They are dropped only after three additional tuples arrive in S_3, and hence remain in $\mathcal{S}(S_1)$ longer than when CA(0) alone is used.

In Figure 16 we report an experimental result illustrating the effect. We used the join graph shown in Figure 7(c) for this experiment, with the filter predicate having 10% selectivity. We generated synthetic streams S_1 and S_2 with CA(5000) on $S_1.A$ and RIDS(k) on the join, varying the RIDS adherence parameter k in the experiment. $S_1 \rightarrow S_2$ has an average multiplicity of 2 for tuples in S_2 and a multiplicity of 1 for tuples in S_1. The y-axis in Figure 16 shows the total memory in use after 600,000 tuples have been processed. Once

k increases beyond 2000 (roughly), the simultaneous use of both constraints performs worse because of the extra time RIDS requires to eliminate tuples in $\mathcal{S}(S_1)$.*Unknown* that arrived after their child tuple was dropped from $\mathcal{S}(S_2)$.*No*.

Based on the observations in this section, if we are interested in minimizing the *time-averaged* total synopsis size, then we are faced with the problem of selecting which constraints to exploit and which to ignore. For a complex join graph with numerous interacting constraints of different types, this *constraint selection problem* may be quite difficult, and we plan to tackle it as future work. Our current implementation of k-Mon uses all k-constraints with k lower than a user-specified threshold and no others.

9. *K*-CONSTRAINTS IN THE LINEAR ROAD QUERIES

We have recently incorporated the entire architecture discussed in this article into the STREAM system at Stanford [Motwani et al. 2003]. Most of our applications [SQR 2003] include several k-constraints that are discovered and exploited by the system. We conclude the article in this section by briefly illustrating how some queries in the Linear Road application, a benchmark for data stream systems [Arasu et al. 2004], benefit from k-constraints.

Before discussing the Linear Road queries, we briefly explain the minor extensions to our k-constraints framework required for the specific semantics of the CQL language [Arasu et al. 2002] supported by the STREAM system. The only significant difference between the relation-based CQL semantics and the pure stream-based semantics used so far in this article is that CQL permits streams with both insertions and deletions, emulating relations. (See Arasu et al. [2002] for details and a discussion of the benefits of this model.)

Processing a deletion s^- arriving in an input stream S in a join is straightforward: s^- is joined with the synopses of all other streams to produce deletions in the join result stream. Without using constraints, the synopsis for S contains all insertions that have arrived in S so far for which matching deletions have not arrived. The notion of many-one joins is extended to accommodate deletions: a join is many-one from stream S_1 to S_2 if any tuple in S_1 joins with at most one insertion and the matching deletion in S_2. With these extensions, our definitions, theorems, and algorithms adapt directly to the relation-based semantics of CQL.

We consider one Linear Road query in detail, then summarize our results. For presentation we simplify the main input stream of the Linear Road application to

CarStr(cid, xpos, sid).

Each tuple in *CarStr* is a report from a sensor in a car identified by *cid*. The tuple indicates that the car was at position *xpos* in the expressway segment *sid* when the report was generated. For details see Arasu et al. [2004] and Arasu [2003].

One of the Linear Road queries, referred to as *AccSeg* in Arasu [2003], tracks segments where accidents may have occurred. A possible accident is identified when the last four reports from a car have the same *xpos*. (*xpos* is global, not relative to segments.) The query is specified in CQL in Figure 17. This query

```
Select distinct sid From
  (Select cid, sid From
     (CarStr [Partition By cid Rows 1]) as LastRep,
     (Select distinct cid From
      CarStr [Range 30 seconds]) as CurActiveCars
   Where LastRep.cid = CurActiveCars.cid)
   as CurCarSeg,
  (Select cid
   From CarStr [Partition By cid Rows 4]
   Group By cid
   Having count (distinct xpos) = 1 and count(*) = 4)
   as AccCars
WhereCurCarSeg.cid = AccCars.cid
```

Fig. 17. *AccSeg* query from Linear Road.

Table III. Results for Linear Road Queries

Query (from Arasu [2003])	Constraints	Memory used (ratio)	Tuple proc. time (ratio)
CurCarSeg	Many-one	0.09	0.65
AccSeg	RIDS	0.13	0.99
CarExitStr	RIDS	0.10	0.49
NegTollStr	RIDS	0.13	0.62

uses *partitioned windows* on *CarStr* which contain the last N ($N = 1, 4$) tuples in *CarStr* for each unique *cid*. Please refer to Arasu et al. [2002] for full syntactic and semantic specifics of CQL. Note that this query could have been written in a slightly simpler form by exploiting the fact that *sid* is functionally determined by *xpos*, but the more complex form is useful anyway for illustrative purposes.

All streams generated by the subqueries in this query have both insertions and deletions. *LastRep* tracks the most recent report from each car. *CurActiveCars* tracks cars that have reported within the last 30 s, which are the cars active currently. *CurCarSeg* is the join of *LastRep* and *CurActiveCars*, tracking the current segment for each active car. *AccCars* tracks cars involved in recent possible accidents, and its join with *CurCarSeg* locates the segments where these cars reported from.

Linear Road has around 1 million cars [Arasu et al. 2004]. Thus, joins in *AccSeg* require large synopses, for example, the synopsis for *LastRep* can occupy around 8 MB of memory. k-Mon identifies and exploits three constraints in *AccSeg*, reducing the memory requirement substantially as shown in Table III. The join from *LastRep* to *CurActiveCars* (producing *CurCarSeg*) and the join from *CurCarSeg* to *AccCars* (producing *AccSeg*) are both many-one. Furthermore, RIDS(k) holds on the join from *CurCarSeg* to *AccCars* for a small value of k that is data-dependent but easily tracked through monitoring.

Eighteen single-block queries are used to express the Linear Road continuous queries in CQL [Arasu 2003]. Twelve of them have joins, of which seven are many-one joins. (Four out of the remaining five are a special type of spatial join.) Six out of the seven single-block queries with many-one joins benefit substantially from our technique. The constraints that apply, the memory reduction achieved by k-Mon in steady state, and the tuple processing time are

given for four of these six single-block queries in Table III. The remaining two queries which benefit from our technique use the same joins as one of the four queries reported here, and thus the performance improvements are identical.

The memory used and tuple processing times in Table III are ratios of the form X/Y, where X and Y are the measurements with and without using constraints, respectively. For these experiments, we used a dataset provided by the authors of the Linear Road benchmark in June 2003. For the queries listed in Table III, k-Mon reduces the memory requirement by nearly an order of magnitude. The scale of memory reduction enables k-Mon to reduce tuple-processing times as well. (All joins used hash indexes on *cid*.) Furthermore, k-Mon produces accurate results for all of these queries.

The Linear Road application highlights the ability of our approach to achieve good memory reduction on complex queries. The user simply provides declarative query specifications and is freed from any concern over stream properties or special execution strategies. The system detects automatically those properties of the data and queries that can be exploited to reduce the ongoing memory requirement during continuous query processing.

10. CONCLUSIONS AND FUTURE WORK

In this article we introduced the concept of k-constraints: "relaxed" constraints that are more likely to hold in data stream environments than their strict counterparts. We showed empirically that exploiting k-constraints can be very effective at reducing the memory requirement for continuous SPJ queries over streams, and that k-constraints can be monitored and incorporated into query processing with low computational overhead. Finally, we presented a unified query-processing framework for exploiting k-constraints that incorporates our execution and monitoring algorithms.

Several avenues exist for further work:

— Modify our algorithm to use other interpretations for k in our k-constraints. We are particularly interested in the alternative of *time-based* k-constraints, where k denotes a time interval instead of number of tuples. We also plan to investigate alternative tuple-based definitions of k. (For example, the clustering distance for CA(k) could be specified in terms of the number of distinct values instead of the definition listed in Section 3.5.) In all cases, we suspect only minor changes to our approach would be needed and performance results would be similar, but thorough investigation is warranted.
— Develop a detailed cost-based optimization framework to decide when the memory reduction from a k-constraint is large enough to justify the computational cost of using the constraint. This framework should also incorporate the constraint selection problem identified in Section 8, where we saw that depending on the performance objective it may not always be beneficial to exploit all constraints on all streams simultaneously.
— Extend our algorithms to be less conservative and thereby give larger memory reduction for applications that are resilient to query result inaccuracy. In general, we would like the memory-accuracy tradeoff to be fully adjustable.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

The appendix contains detailed examples of k-constraints, extensions to the basic query processing algorithm for DAG-shaped and cyclic join graphs, and proofs of the theorems from the main body of the paper.

REFERENCES

AJTAI, M., JAYRAM, T., KUMAR, R., AND SIVAKUMAR, D. 2002. Counting inversions in a data stream. In *Proceedings of the 2002 Annual ACM Symposium on Theory of Computing*. ACM Press, New York, NY, 370–379.

ARASU, A. 2003. CQL Specification of the Linear Road Benchmark. Available online at http://www-db.stanford.edu/stream/cql-benchmark.html.

ARASU, A., BABU, S., AND WIDOM, J. 2002. An abstract semantics and concrete language for continuous queries over streams and relations. Tech. rep. Stanford University, Stanford, CA. Available online at http://dbpubs.stanford.edu/pub/2002-57.

ARASU, A., CHERNIACK, M. ET AL. 2004. Linear road: A stream data management benchmark. In *Proceedings of the 2004 International Conference on Very Large Data Bases*. Morgan Kaufmann, San Mateo, CA.

BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R., AND WIDOM, J. 2002. Models and issues in data stream systems. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM Press, New York, NY, 1–16.

BABU, S., MOTWANI, R., MUNAGALA, K., NISHIZAWA, I., AND WIDOM, J. 2004a. Adaptive ordering of pipelined stream filters. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY.

BABU, S., MUNAGALA, K., WIDOM, J., AND MOTWANI, R. 2004b. Adaptive caching for continuous queries. Tech. rep. Stanford University, Stanford, CA. Available online at http://dbpubs.stanford.edu/pub/2004-14.

BABU, S. AND WIDOM, J. 2004. StreaMon: An adaptive engine for stream query processing (demonstration). In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY.

BLOOM, B. 1970. Space/time tradeoffs in hash coding with allowable errors. *Commun. ACM 13*, 7, 422–426.

CACERES, R. ET AL. 2000. Measurement and analysis of IP network usage and behavior. *IEEE Commun. Mag. 38*, 5, 144–151.

CARNEY, D., CENTINTEMEL, U. ET AL. 2002. Monitoring streams—a new class of data management applications. In *Proceedings of the 28th International Conference on Very Large Data Bases*. Morgan Kaufmann, San Mateo, CA, 215–226.

CHANDRASEKHARAN, S. AND FRANKLIN, M. J. 2002. Streaming queries over streaming data. In *Proceedings of the 28th International Conference on Very Large Data Bases*. Morgan Kaufmann, San Mateo, CA, 203–214.

CHEN, J., DEWITT, D. J., TIAN, F., AND WANG, Y. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 379–390.

CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. 2003. Gigascope: A stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 647–651.

DAS, A., GEHRKE, J., AND RIEDEWALD, M. 2003. Approximate join processing over data streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 40–51.

DOBRA, A., GAROFALAKIS, M., GEHRKE, J., AND RASTOGI, R. 2002. Processing complex aggregate queries over data streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 61–72.

DUFFIELD, N. AND GROSSGLAUSER, M. 2000. Trajectory sampling for direct traffic observation. In *Proceedings of the 2000 ACM SIGCOMM*. ACM Press, New York, NY, 271–284.

FEIGENBAUM, J., KANNAN, S., STRAUSS, M., AND VISWANATHAN, M. 2000. Testing and spot checking of data streams. In *Proceedings of the 2000 Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM/SIAM, New York, NY, 165–174.

GANGULY, S., GAROFALAKIS, M., AND RASTOGI, R. 2004. Processing data-stream join aggregates using skimmed sketches. In *Proceedings of the 9th International Conference on Extending Database Technology*. Springer, Berlin, Heidelberg, Germany, 569–586.

GARCIA-MOLINA, H., LABIO, W., AND YANG, J. 1998. Expiring data in a warehouse. In *Proceedings of the 1998 International Conference on Very Large Data Bases*. Morgan Kaufmann, San Mateo, CA, 500–511.

GEHRKE, J. 2003. Special issue on data stream processing. *IEEE Comput. Soc. Bull. Tech. Comm. Data Eng. 26*, 1 (March).

GOLAB, L. AND OZSU, T. 2003a. Issues in data stream management. *SIGMOD Rec. 32*, 2 (June), 5–14.

GOLAB, L. AND OZSU, T. 2003b. Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings of the 2003 International Conference on Very Large Data Bases*. Morgan Kaufmann, San Mateo, CA, 500–511.

HAMMAD, M., AREF, W., AND ELMAGARMID, A. 2003. Stream window join: Tracking moving objects in sensor-network databases. In *Proceedings of the 2003 International Conference on Scientific and Statistical Database Management*. IEEE Computer Society Press, Los Alamitos, CA, 75–84.

HELLERSTEIN, J. M., FRANKLIN, M. J. ET AL. 2000. Adaptive query processing: Technology in evolution. *IEEE Comput. Soc. Bull. Tech. Comm. Data Eng. 23*, 2 (June), 7–18.

HELMER, S., WESTMANN, T., AND MOERKOTTE, G. 1998. Diag-join: An opportunistic join algorithm for 1:n relationships. In *Proceedings of the 1998 International Conference on Very Large Data Bases*. Morgan Kaufmann, San Mateo, CA, 98–109.

KANG, J., NAUGHTON, J. F., AND VIGLAS, S. 2003. Evaluating window joins over unbounded streams. In *Proceedings of the 2003 International Conference on Data Engineering*. IEEE Computer Society Press, Los Alamitos, CA.

MADDEN, S., SHAH, M. A., HELLERSTEIN, J. M., AND RAMAN, V. 2002. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 49–60.

MOTWANI, R., WIDOM, J. ET AL. 2003. Query processing, approximation, and resource management in a data stream management system. In *Proceedings of the 1st Conference on Innovative Data Systems Research*. 245–256.

NETFLOW. 2003. *Netflow Services and Applications*. Available online at www.cisco.com/warp/public/732/netflow.

RAMAN, V., DESHPANDE, A., AND HELLERSTEIN, J. 2003. Using state modules for adaptive query processing. In *Proceedings of the 2003 International Conference on Data Engineering*. IEEE Computer Society Press, Los Alamitos, CA.

SQR. 2003. SQR—a Stream Query Repository. Available online at http://www-db.stanford.edu/stream/sqr.

SRIVASTAVA, U. AND WIDOM, J. 2004. Memory-limited execution of windowed stream joins. In *Proceedings of the 2004 International Conference on Very Large Data Bases*. Morgan Kaufmann, San Mateo, CA.

TATBUL, N., CETINTEMEL, U., ZDONIK, S., CHERNIACK, M., AND STONEBRAKER, M. 2003. Load shedding in a data stream manager. In *Proceedings of the 2003 International Conference on Very Large Data Bases*. Morgan Kaufmann, San Mateo, CA, 309–320.

TUCKER, P. A., MAIER, D., SHEARD, T., AND FEGARAS, L. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowl. Data Eng. 15*, 3, 555–568.

URHAN, T., FRANKLIN, M. J., AND AMSALEG, L. 1998. Cost based query scrambling for initial delays. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 130–141.

VIGLAS, S., NAUGHTON, J. F., AND BURGER, J. 2003. Maximizing the output rate of multi-join queries over streaming information sources. In *Proceedings of the 2003 International Conference on Very Large Data Bases*. Morgan Kaufmann, San Mateo, CA, 285–296.

Received November 2003; revised June 2004; accepted May 2004