Exploiting k -Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams^{*}

Shivnath Babu and Jennifer Widom
Stanford University
{shivnath, widom}@cs.stanford.edu

Abstract

We consider the problem of efficiently processing continuous queries over multiple continuous data streams in the presence of constraints on the data streams. We specify several types of constraints, and for each constraint type we identify an “adherence parameter” k that captures how closely a given stream or joining pair of streams adheres to a constraint of that type. We then present a query execution algorithm that takes k -constraints over streams into account in order to reduce memory overhead. In general, the tighter the adherence parameters are in the k -constraints, the less memory required. Furthermore, if input streams do not adhere to constraints within the specified adherence parameters, our algorithm automatically degrades gracefully to provide continuous approximate answers. We have implemented our approach in a testbed continuous query processor and preliminary experimental results are reported.

1 Introduction

There has been a surge of interest recently in the area of query processing over continuous data streams, e.g., [CCC⁺02, CF02, DGGR02, MSHR02, MW⁺03, VN02]. In many of the relevant applications—network monitoring, sensor processing, Web tracking, telecommunications, and others—queries are long-running, or *continuous* [BW01, CDTW00, MSHR02]. It has been shown that continuous queries over unbounded data streams often require very large or unbounded amounts of memory¹ during execution in order to produce correct answers over all possible streams [ABB⁺02]. In this paper, we show that when data streams satisfy certain *constraints*, memory overhead may be reduced significantly if the query processor takes the constraints into account.

We first provide an overview of our approach. We informally describe the data stream constraints and continuous queries we consider, then briefly discuss our query execution algorithm that exploits stream constraints to reduce memory overhead.

1.1 Stream Constraints Overview

One contribution of this paper is to identify several types of constraints that are relevant in the continuous data streams environment, and to specify the notion of *k-constraints*. Informally, the constraint types we introduce are:

^{*} This work was supported by the National Science Foundation under grants IIS-0118173 and IIS-9817799.

¹We use “memory” to refer to the intermediate storage required during processing of continuous queries over continuous data streams. Disk could be used for this storage as well, and does not change our basic algorithms or storage overhead results.

- *Many-one join constraints* between two streams. We will assume that all joins in our queries are many-one equijoins. Our overall approach does not rely on this assumption, but the benefit of our approach may be reduced in the presence of many-many joins. Many-one joins (of which one-one joins are a special case) are extremely common in practice.
- *Referential-integrity constraints* between the streams in a many-one join, specifying that when a stream tuple arrives on the many side of the join, the joining tuple will already have arrived on the other stream.
- *Clustered-arrival constraints* on an attribute of a stream, specifying that duplicate values for the attribute arrive together.
- *Ordered-arrival constraints* on an attribute of a stream, specifying that the attribute values are both clustered and ordered.

For each constraint type except many-one joins, we also define an *adherence parameter* k , which captures the degree to which a stream or pair of streams adheres to the constraint. When $k = 0$ the constraint holds with its strict interpretation. We refer to a constraint with an adherence parameter k as a *k-constraint*. Stream constraints and their adherence parameters are defined formally in Sections 4–6. Informally:

1. A referential-integrity constraint on a many-one join from stream S_1 to stream S_2 with adherence parameter k says that when a tuple s_1 arrives on S_1 , its joining tuple s_2 will arrive within k tuples on S_2 .
2. A clustered-arrival constraint on stream attribute $S.A$ with adherence parameter k says that if two tuples in S have the same value v for A , then at most k tuples with non- v values for A occur on S between them.
3. An ordered-arrival constraint on stream attribute $S.A$ with adherence parameter k says that for any tuple s in S , S tuples that arrive at least $k + 1$ tuples after s have an A value greater than or equal to $s.A$.

The concept of k -constraints is very important in the stream context. Often it may be unreasonable for data 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 we want to capture and take advantage of these situations.

1.2 Continuous Queries Overview

We consider relational *Select-Project-Join (SPJ)* queries over continuous relational data streams. As mentioned in the previous subsection, we require all joins to be many-one (or one-one as a special case). The semantics we use for continuous SPJ queries over continuous streams is a relatively standard one [ABB⁺02]: The answer to a continuous query Q over a set of unbounded streams $S_1, S_2, \dots, S_n$ at any point in time is the conventional relational answer to Q over the portion of the streams seen so far, treated as relations.

Note that our query language is monotonic in the sense that once a tuple is in a query result, it remains in the result forever.

1.3 Query Execution Algorithm Overview

Consider a continuous query Q . Our query execution algorithm is based on keeping a *synopsis* for each data stream involved in Q . In most cases it is not feasible for each synopsis to record the entire stream history in order to produce continuous query results, due to unbounded memory overhead, query response time requirements, or other performance concerns.

For query processing and for synopsis reduction techniques, our synopsis for each stream S is divided logically into three components:

- The *Yes* component logically contains all tuples $s \in S$ seen so far such that s has the potential to contribute to one or more query result tuples.
- The *No* component logically contains all tuples $s \in S$ seen so far such that s cannot contribute to a query result tuple.
- The *Unknown* component logically contains all tuples $s \in S$ seen so far such that we cannot yet place s into *Yes* or *No*.

Of course the union of these three components comprises the entire stream history (assuming all attributes are kept). Our query execution algorithm uses k -constraints on the streams (as well as a few constraint-independent techniques) to component-wise reduce the size of—or completely eliminate—the synopsis data that is actually stored.

While it may appear that *Yes* components always must be saved and *No* components always may be discarded, this heuristic is not necessarily a good one, as can be seen by the following very simple example. Suppose $Q = (S_1) \bowtie_{S_1.A=S_2.A} (\sigma_{B>10}(S_2))$, a many-one equijoin from stream S_1 (many side) to stream S_2 (one side) with a selection predicate on stream S_2 . If a tuple s_2 arrives on S_2 that cannot contribute to the result because it fails the selection predicate, then we want to store s_2 (actually only $s_2.A$ needs to be stored) in S_2 's *No* component. We can remove all tuples from stream S_1 's *Unknown* component that join with s_2 , and we can discard any future joining tuples arriving on S_1 . If in addition strict referential integrity holds over the join, then we need no synopsis at all for S_1 , since when a tuple arrives on S_1 the matching join tuple must already appear somewhere in the synopsis for S_2 and we know how to proceed. If we have referential integrity with adherence parameter k , then a tuple s on S_1 must be saved for at most k arrivals on S_2 after the arrival of s . As a final scenario, if S_2 satisfies k -ordered-arrival on $S_2.A$, then a tuple s on S_1 must be saved for at most k arrivals on S_2 following any $S_2.A$ value greater than $s.A$.

Although these simple examples illustrate some of the basic ideas behind our approach, solving the general case is quite complex, since we must consider arbitrary queries and arbitrary combinations of stream constraints.

1.4 Discussion

1.4.1 Time-Based Adherence Parameters

In this paper we consider several different constraint types, and for each one we specify an adherence parameter k that relaxes the strictest interpretation of the constraint. We provide a specific interpretation for k for each constraint type, but in all cases k is *tuple-based*. For example, in a k -clustered-arrival constraint on attribute A in stream S , k bounds the number of S tuples with non- v values for A between any two tuples with the same value v for A . An equally useful—indeed perhaps more useful in some environments—interpretation for k would be *time-based*. For example, a clustered-arrival constraint with adherence parameter k might state that at most k units of time pass between any two tuples with the same attribute value.

The fundamentals of our approach do not change if we use time-based instead of tuple-based adherence parameters: We still obtain a reduction of synopsis sizes based on k -constraints, with exactly the same types of reductions for the same types of constraints (summarized at the end of the paper in Table 1). However, there are two important differences. First, the implementation mechanism obviously must be based on timeouts instead of tuple counting. Second, tuple-based k -constraints often allow us to state an exact bound on the maximum synopsis sizes needed for a given query and set of constraints (although we do not state those bounds in this paper due to space limitations), while with time-based k -constraints we cannot state such a bound since any number of tuples may arrive within a given time period. We plan to further investigate and evaluate time-based adherence parameters as future work.

1.4.2 Specifying Constraints

For clarity we will assume that all constraints on streams, including their associated adherence parameters, are specified statically, i.e., when streams are “registered” and their schema is specified (although our algorithms do not rely on this assumption). Although we do not provide details in this paper, it will be clear that our query execution algorithm can adjust its behavior if k -constraints are added and deleted over time. Furthermore, if a constraint specification underestimates an adherence parameter k , our algorithm produces an approximate result whose imprecision usually corresponds to the error in k .

An alternate approach is not to declare constraints explicitly at all, or to provide initial, adjustable specifications. The system then monitors streams to detect periods when constraints hold with acceptably low adherence parameters. (Note that every finite stream prefix satisfies any k -constraint as long as parameter k is large enough.) Determining how to monitor constraints efficiently is an interesting area of future work not addressed in this paper.

1.4.3 Sliding-Window Joins

In some approaches to query processing over unbounded streams, joins are by default *sliding-window* (or *band*) joins: tuples in one stream by definition can join only with tuples in the other stream that occur within a prespecified window (based on time, number of tuples, sequence number, etc.). One primary reason for redefining joins in this way is to bound the required memory [CCC⁺02, CF02, KNV03, MSHR02],

although sliding-window joins also might be used for semantic reasons. In this paper, we do not redefine the join operator. Instead, our approach uses k -constraints on streams to automatically reduce memory overhead for regular joins, effectively producing execution strategies similar to sliding-window joins in many cases. Furthermore, even when windows are specified, our techniques can further reduce memory overhead, sometimes considerably.

1.5 Outline of Paper

Section 2 reviews related work. Section 3 provides the basics for our work: It formalizes our stream data model and continuous queries, specifies a basic query processing algorithm which in subsequent sections is extended to exploit knowledge of stream constraints, and describes our experimental evaluation setup. Sections 4–6 consider our three constraint types independently, while Section 7 covers the simultaneous use of different constraints. Section 8 discusses what happens when input streams do not adhere to constraints within the specified parameters, and finally in Section 9 we outline future work.

Due to space limitations, some details of our algorithms and proofs of theorems appear in a longer technical report version of this paper.

2 Related Work

A comprehensive description of work relating to data streams and continuous queries is provided in, e.g., [BBD⁺02]. Here we focus on work specifically related to query processing in the presence of constraints and memory overhead reduction.

In Section 1.4 we mentioned sliding-window or *band* joins, which effectively limit the result of joins on data streams to reduce memory overhead [CCC⁺02, CF02, DNS91, HWM98, JC00, KNV03, MSHR02], and we explained how band joins relate to our approach. Previous work in [ABB⁺02] exactly characterized those SPJ queries that can be answered in bounded memory over unbounded streams, but assumed arbitrary unconstrained streams (i.e., considered the worst-case scenario). The work most closely approaching ours is *punctuated data streams* [TMSF02]. 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. [TMSF02] does not address constraints over multiple streams or adherence parameters. Broadly, we can place related work in these areas into three categories:

1. Punctuations are *instance-level stream constraints*, specified in the data itself, with a worst-case scenario of no constraints holding at all.
2. Our work considers *schema-level stream constraints*, which are expected to hold (within their adherence parameters) over all possible streams.
3. Sliding window joins can be thought of as *query-level constraints*, which constrain the result of a query to a portion that can be computed efficiently over any streams.

The three categories all yield different techniques and results, and each is valid in certain environments. In some sense the three approaches are complimentary and therefore could be combined, although particular environments are likely to favor one approach over another.

There are other techniques for controlling memory overhead in continuous query environments, including using disk to buffer data for memory overflows [CCC⁺02, IFF⁺99, UFA98], grouping queries or operators to minimize memory usage [CDTW00, MSHR02, RDH03], a wide variety of memory-efficient approximation techniques [DGGR02, GKMS01, GK02], and run-time *load shedding* [CCC⁺02]. None of these approaches consider constraints on streams as a method of reducing the data that must be stored during query processing.

In data warehousing there has been work on *data expiration*—identifying and deleting data no longer needed for maintaining materialized views. Specifically, [GMLY98] presents a language for expressing constraints over relations and views and develops algorithms to exploit the constraints for expiration. The language and algorithms in [GMLY98] are inadequate to support constraints over streams (as opposed to relations) because streams have arrival characteristics in addition to data characteristics, and as a consequence [GMLY98] cannot support adherence parameters. Work on *self-maintenance* [BCL89, GJM96, QGMW96] considered the related but distinct problem of identifying the minimum amount of data required for maintaining a materialized view, and the *Chronicle data model* [JMS95] specified a constrained model and language that guarantees bounded memory overhead for view maintenance.

Constraints are used in *semantic query optimization* for query rewriting and plan cost prediction [CGM90, GGZ01]. In contrast, we use constraints to reduce memory overhead during query execution.

3 Basics

3.1 Data Streams and Continuous Queries

A *continuous data stream* (hereafter *stream*) is a potentially infinite stream of relational tuples. Each stream has a fixed schema, i.e., a known finite set of attributes. We consider continuous SPJ queries over streams only, although adding relations to our model would not be difficult. The result of a continuous query Q at time τ is the result of evaluating Q over all tuples that have arrived on all relevant streams up to time τ . Since the queries we consider are monotonic, 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 paper 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 $S_1 \rightarrow S_2$, 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 $S_1 \rightarrow S_2$ produces an edge from S_1 to S_2 . For exposition we assume that all join graphs are connected, although this assumption is not intrinsic to our approach.

A number of technical definitions related to join graphs are needed:

- Given $S_1 \rightarrow S_2$, 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 $S_1 \rightarrow S_2$ 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 , 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 . 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 $G(Q)$.
- We consider three classes of join graphs, based on their shape. (Recall that we assume join graphs are connected.) $G(Q)$ is *directed-tree-shaped (DT-shaped)* if there are no cycles in the undirected version of the graph. $G(Q)$ is *DAG-shaped* if $G(Q)$ does not contain directed cycles. (Note that all DT-shaped graphs are DAG-shaped.) Otherwise, $G(Q)$ is *cyclic*.

In Section 1.3 we informally described the three-part *synopses* maintained by our query processing algorithm for each stream S in a query Q . We can now formalize our most general synopses.

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

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

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 Processing Algorithm

In this paper we are not focusing on query plan space exploration or efficient execution strategies, but rather on how stream constraints can be used to reduce memory overhead during query processing. While the use of constraints certainly may affect query optimization techniques and vice-versa, we believe that studying the constraint exploitation problem independently is the right first step. To that end, in this section we define a “naive” (but still far from simple) query processing algorithm that we use as a basis for our constraint-specific memory reduction techniques in subsequent sections. Throughout the paper 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*)
2. Generating new query result tuples as they become available (*result generation*)

We begin by considering DT-shaped join graphs in Section 3.2.1, then extend to DAG-shaped and cyclic join graphs in Section 3.2.2. In Section 3.3 we discuss some basic constraint-independent techniques for eliminating certain tuples and columns from synopses.

3.2.1 DT-Shaped Join Graphs

Consider a DT-shaped 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 for DT-shaped join graphs are based on Definition 3.1 of synopsis components, and they suggest a method for synopsis maintenance. Recall that proofs for all theorems are omitted due to space limitations but are provided in the technical report version of this paper.

Theorem 3.1 Consider any stream S and tuple $s \in S$ 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*. □

Theorem 3.2 Consider any stream S and tuple $s \in S$. 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*. □

Theorem 3.3 Consider any stream S and tuple $s \in S$ such that s satisfies all filter predicates on S . If $Children(S) \neq \phi$, there is some stream $S' \in Children(S)$ for which $\mathcal{S}(S')$.*Yes* does not contain the child tuple of s in S' , and there is no stream $S'' \in Children(S)$ for which $\mathcal{S}(S'')$.*No* contains the child tuple of s in S'' , then $s \in \mathcal{S}(S)$.*Unknown*. □

A relatively straightforward recursive algorithm for maintaining synopsis components (i.e., inserting and deleting synopsis tuples) as stream tuples arrive follows from Theorems 3.1–3.3. Constraint-independent techniques for reducing synopsis sizes without violating Theorems 3.1–3.3 are discussed in Section 3.3.

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.

Figure 1: Join graphs and synopses used in examples

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 $S \in \rho \in \text{MinCover}(G(Q))$. $\square$

Theorem 3.5 The set of root streams is the only minimal cover in a DT-shaped join graph. $\square$

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 produce the new tuples in the result. Note that if $G(Q)$ contains a single root stream S , then a new tuple $s \in \mathcal{S}(S)$. *Yes* can produce at most one tuple in the result. If $G(Q)$ contains multiple root streams, then an arbitrary number of new result tuples may be produced. Detailed algorithms for synopsis maintenance and result generation are specified in the technical report version of this paper.

Before moving on to DAG-shaped join graphs, 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 3.1 Consider a query Q having the DT-shaped join graph in Figure 1(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 1(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 $\mathcal{S}(S_1).No$. $\square$

Example 3.2 Consider the DT-shaped join graph and synopses shown in Figure 1(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. $\square$

3.2.2 DAG-Shaped and Cyclic Join Graphs

Theorem 3.1 does not hold when a query's join graph is DAG-shaped instead of DT-shaped, as illustrated by the following simple example.

Example 3.3 Consider a query Q with the DAG-shaped join graph and synopses shown in Figure 1(c). Suppose tuple $s = (3, 4)$ arrives in stream S_1 . Although both child tuples of s are in the respective *Yes* components, clearly $s \bowtie G_{S_1}(Q)$ is empty. $\square$

A small modification to our algorithm for synopsis maintenance in DT-shaped join graphs is needed to accommodate DAG-shaped join graphs. Instead of inserting a tuple s into $\mathcal{S}(S).Yes$ when all of s 's child tuples are in the corresponding child *Yes* components, we must check explicitly if $s \bowtie G_S(Q)$ is nonempty. Evaluating this semijoin involves joining the *Yes* synopsis components in $G_S(Q)$, substituting s for stream S . If the semijoin is nonempty, s belongs in $\mathcal{S}(S).Yes$, otherwise it belongs in $\mathcal{S}(S).No$. Result generation proceeds exactly as in the DT-shaped case since Theorems 3.4 and 3.5 also hold for DAG-shaped join graphs.

The modifications to handle cyclic join graphs are somewhat involved and details are omitted due to space constraints. The basic idea is that before we insert a tuple s into $\mathcal{S}(S).Unknown$ because its child tuple s' is in $\mathcal{S}(S').Unknown$, we need to check if a cyclic relationship exists that permits us to move all tuples in the cycle to their respective *Yes* or *No* synopsis components. For result generation, there may be more than one minimal cover in a cyclic join graph, so we need to compute the minimal covers and then apply Theorem 3.4 which holds for cyclic join graphs as well. Detailed algorithms for DAG-shaped and cyclic joins graphs are given in the technical report version of this paper.

3.3 Synopsis Reduction

In our basic query processing 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 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 the following theorem.

Theorem 3.6 Consider a join graph $G(Q)$. If a stream S forms a minimal cover for $G(Q)$, i.e., $\{S\} \in \text{MinCover}(G(Q))$, 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. $\square$

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 a DT- or DAG-shaped join graph has a single root stream S , since in these classes of 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.

Space limitations preclude detailed examples or formal modeling of the tradeoff between keeping non-root *No* components or eliminating them. The presence of constraints further complicates the tradeoff, although often 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 constraint-based techniques.

3.3.2 Eliminating Columns

Handling queries with explicit projection does not change our basic query processing 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 $Parents(S)$. Together, these techniques can significantly reduce overall synopsis sizes.

3.4 Stream Characteristics and Experiments

In the next three sections of the paper we will consider our three constraint types in turn. For each constraint type we:

1. Define the constraint type formally, including the meaning of adherence parameter k .
2. Present the memory-reduction techniques enabled by constraints of that type, specified as extensions to the basic query processing algorithm from Section 3.2.
3. Show experimental results demonstrating the memory overhead reduction enabled by constraints of that type.

Since in this paper we are interested primarily in memory usage, the graphs for our experimental results all have total synopsis size on the y axis. On the x axis we vary either the number of tuples arrived so far on the input streams (to study how synopses grow over time), or we vary the adherence parameter k (to study how constraint adherence affects memory overhead).

Stream data characteristics relevant to our experiments are:

- **Join tuple multiplicity:** Consider a join $S_1 \rightarrow S_2$. Join tuple multiplicity for a tuple $s_2 \in S_2$ 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 the latter case the multiplicity is either 0 or 1 since the join is many-one. 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 .

- **Filter predicate selectivity:** The percentage of tuples on a stream S satisfying a particular filter predicate on S . Except as noted otherwise, all filter predicates in our experiments have an average selectivity of 50%.

We also must consider stream *arrival characteristics*. For any set of streams $\rho = \{S_1, S_2, \dots, S_n\}$ we assume a logical interleaving of the arrival of tuples in $S_1, S_2, \dots, S_n$ 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 $s.A \neq s_1.A$. That is, the clustering distance between tuples s_1 and s_2 having the same value of A is the number of tuples that arrive between them with a different value of 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 between s_1 and s_2 , including s_2 . Informally, this distance is the number of tuples by which s_1 needs to be moved later in the stream in order to “unscramble” s_1 and s_2 so they arrive ordered by A .
- **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. That is, the join distance is 0 if the tuple on the many side ($s_1 \in S_1$) arrives after the tuple on the one side ($s_2 \in S_2$), otherwise it specifies how many S_2 tuples, including s_2 , arrive between them.

For our experiments we developed a configurable synthetic stream generator, and we developed a prototype continuous query processor that implements our algorithms for exploiting k -constraints. The stream generator takes as input schema information, data characteristics, and arrival characteristics of multiple streams and generates an interleaved stream arrival order with the specified characteristics. (Details are given for each experiment.) Our query processor provides a command interface to register data streams, k -constraints on streams, and continuous SPJ queries, all of which are stored in a system catalog. A query plan consisting of push-based and pull-based operators (similar to [MF02]) and synopsis storage structures based on k -constraints is generated and runs continuously for each continuous query. Our initial continuous query processor implements the algorithms for DT-shaped join graphs only, so all of our experimental results are over this class of queries.

4 Referential Integrity Constraints

We first consider the data stream equivalent of standard relational *referential integrity* [UW97]. 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 when a tuple s_1 arrives

on S_1 , its joining (child) tuple has already arrived on S_2 . The more relaxed k -constraint version states that when a tuple s_1 arrives on S_1 , its joining tuple s_2 will arrive within k tuple arrivals on S_2 . (When $k = 0$ we have the strictest form described above.) In terms of our arrival characteristics model (Section 3.4), the join distance between s_1 and s_2 is no greater than k .

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 continues to produce tuples there is a joining tuple $s_2 \in S_2$ and the join distance between s_1 and s_2 is no greater than k . $\square$

Note that for each of our constraint types we have assigned a specific interpretation to adherence parameter k . While we believe that our k -constraint interpretations are intuitive, there are other interpretations that may be intuitive as well. For example, for RIDS(k) we could say that k bounds the number of additional tuples on S_1 before the joining tuple arrives, instead of on S_2 , or the adherence parameter could be based on time instead of number of tuples, as discussed in Section 1.4.1. Algorithmic details and exact memory overhead reduction can differ for different interpretations of k , but for the alternate interpretations we have explored the overall approach remains the same.

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 indefinitely. RIDS constraints allow us to eliminate *No* components without this risk, using the following technique. (Complete algorithmic details appear in the technical report version of this paper.)

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 that our basic query processing algorithm uses $\mathcal{S}(S).No$ to determine whether a parent tuple s' belongs in $\mathcal{S}(S').No$. If RIDS(k) holds with $k = 0$, then when s' arrives its child tuple s must already have arrived. If $s \notin \mathcal{S}(S).(Yes \cup Unknown)$ then we can infer that logically $s \in \mathcal{S}(S).No$. We insert s' into $\mathcal{S}(S').No$ and proceed accordingly. If $k > 0$ and child tuple $s \notin \mathcal{S}(S).(Yes \cup Unknown)$, then either the child tuple has not arrived yet, or the child tuple arrived, belonged in $\mathcal{S}(S).No$ (which we do not keep), and was discarded. We place s' in $\mathcal{S}(S').Unknown$. If k more tuples arrive on S without arrival of the child tuple, we can infer that $s \in \mathcal{S}(S).No$ and s arrived before s' . We move s' to $\mathcal{S}(S').No$ and proceed accordingly.

Example 4.1 Consider the join graph and synopses shown in Figure 1(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 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 . $\square$

4.2 Experimental Analysis for RIDS(k)

We now present our first experimental results. For the RIDS experiments we used the join graph shown in Figure 2(a). We generated synthetic data for streams S_1 , S_2 , and S_3 with different arrival orders conforming

Figure 2: Join graphs used in experiments

Figure 3: Synopsis size using RIDS(k) as tuples arrive

Figure 4: Synopsis size using RIDS(k) as k varies

to RIDS(k) (for both joins, varying values of k). Individual tuple join distances were chosen uniformly at random from $[0, \dots, k]$. All tuple sizes are 8 bytes each in this and all subsequent experiments.

We show two graphs, both with total synopsis size on the y -axis. In Figure 3 we have the number of arrived tuples on the x -axis, and show plots for $k \in \{0, 1000, 5000, 10000, 20000\}$. In Figure 4 we fix the number of arrived tuples and vary the adherence parameter k on the x -axis. Recall that RIDS(k) on $S_1 \rightarrow S_2$ and $S_1 \rightarrow S_3$ eliminates $\mathcal{S}(S_2).No$ and $\mathcal{S}(S_3).No$ completely, 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 (recall Section 3.3.1). However, total memory overhead still increases over time in Figure 3 because of $\mathcal{S}(S_2).Yes$ and $\mathcal{S}(S_3).Yes$, which are not affected by the RIDS constraints. When we combine RIDS with other constraints (Section 5.2), we will see convergence to a constant total synopsis size. Figure 4 shows the increase in memory overhead as the adherence to RIDS decreases.

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 ($\text{CA}(k)$) Constraint $\text{CA}(k)$ 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 between s_1 and s_2 (Section 3.4) is no greater than k . $\square$

5.1 Modified Algorithm to Exploit $\text{CA}(k)$

The benefits of $\text{RIDS}(k)$ constraints were focused on the reduction or elimination of *No* and *Unknown* components. $\text{CA}(k)$ constraints help eliminate tuples from *Yes*, *No*, and *Unknown* components. Elimination of tuples from *Yes* and *Unknown* components is based on the following theorem.

Theorem 5.1 Consider a join graph $G(Q)$ and a stream $S \in G(Q)$ with $\text{Parents}(S) = \{S_1, S_2, \dots, S_n\}$. A tuple $s \in S$ will not join with any future tuples to produce result tuples if the following conditions are satisfied for some $\rho \subseteq \{S_1, S_2, \dots, S_n\}$:

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

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 . $\square$

Each $\rho \subseteq \{S_1, S_2, \dots, S_n\}$ that forms a minimal cover of $G(Q)$ can be identified at query compilation time. For each such ρ , condition C2 in Theorem 5.1 can be evaluated at a given time by joining s with the contents of $\mathcal{S}(S_i).Unknown$. A $\text{CA}(k)$ 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 $\text{CA}(k)$ on $S_i.A$. If a tuple s_1 arrives on S_i with $s_1.A = v$, then once $k + 1$ tuples with a non- v value for A have arrived in 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 5.1, s can be eliminated. Furthermore, any tuple in $\{S_1, S_2, \dots, S_n\}$ that joins with s also 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 $S' \in \text{Parents}(S)$ can join with s . $\text{CA}(k)$ constraints can be used to identify such tuples as explained above. Implementation details of incorporating tuple eliminations based on $\text{CA}(k)$ constraints into our query processing algorithm are given in the technical report version of this paper.

Example 5.1 Consider again the join graph and synopses in Figure 1(a). Suppose $\text{CA}(1)$ 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 $\text{CA}(1)$ 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. $\square$

Figure 5: Synopsis size using $CA(k)$ as tuples arrive

Figure 6: Synopsis size using $CA(k)$ as k varies

5.2 Experimental Analysis for $CA(k)$

For the CA experiments we used the join graph shown in Figure 2(b). 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 $S_2.A$, varying values of k). Maximum clustering distances for distinct values of $S_1.A$ and $S_2.A$ are distributed uniformly in $[0, \dots, k]$. 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 RIDS(0) is not used, its presence in the input streams keeps the *Unknown* components empty. Hence the total memory overhead remains constant over time for each value of k , as observed in Figure 5. Figure 6 shows the increase in memory overhead as the adherence to CA decreases.

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 , i.e., 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 is no greater than k . $\square$

$OA(k)$ 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*

Figure 7: Synopsis size using OAP(k) as tuples arrive

Figure 8: Synopsis size using OAP(k) as k varies

of parent stream (hereafter OAP(k)) and ordered-arrival of child stream (hereafter OAC(k)). We consider the two classes in turn.

6.1 Modified Algorithm to Exploit OAP(k)

Like CA(k), OAP(k) constraints on the join attributes in streams $\{S_1, S_2, \dots, S_n\}$ can be used to evaluate condition C3 in Theorem 5.1. Let $S_i.A = S_j.B$ be a predicate in the $S_i \rightarrow S_j$ 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 $s \in S_j$ if $s.B < s_i.A$. Hence, an OAP(k) constraint on any one of S_i 's join attributes in $S_i \rightarrow S_j$ for each $S_i \in \rho$ is sufficient to evaluate condition C3 in Theorem 5.1. 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_j (tuples that never join), while CA(k) cannot. The algorithm can be extended in a straightforward manner to the case where a mixture of CA(k) and OAP(k) constraints hold over streams in ρ in Theorem 5.1.

Example 6.1 Consider again the join graph and synopses in Figure 1(a). Suppose OAP(3) holds on attribute $S_1.B$, and consider the following sequence of tuple arrivals in S_1 : (11, 10), (6, 5), (6, 8), (9, 10). After these arrivals, logically $\mathcal{S}(S_1).Yes = \{(6, 5)\}$, logically $\mathcal{S}(S_1).No = \{(11, 10), (9, 10)\}$, $\mathcal{S}(S_1).Unknown = \{(6, 8)\}$, and result tuple (6, 5, 20, 3) is emitted. On $S_1.B$ three tuples have appeared after the first tuple with $S_1.B = 10$, so OAP(3) guarantees that no future tuple $s \in S_1$ can have $s.B < 10$. Since no tuple in $\mathcal{S}(S_1).Unknown$ has $B = 5$ or $B = 7$, tuples $(5, 3) \in \mathcal{S}(S_3).Yes$ and $(7, 9) \in \mathcal{S}(S_3).No$ can be discarded. $\square$

6.2 Experimental Analysis for OAP(k)

For the OAP experiments we used the same join graph as for CA (Figure 2(b)). The synthetic 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 $[0, \dots, k]$. In Figure 7, the total

synopsis size for each value of k varies around some fairly fixed value. The “sawtooth” pattern is due to tuples entering synopses one at a time, but being deleted in batches. 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 result in larger variation. Figure 8 shows the increase in memory overhead as the adherence to OAP decreases.

6.3 Modified Algorithm to Exploit OAC(k)

OAC(k) constraints allow us to eliminate *No* components without running the risk of leaving tuples in parent or ancestor *Unknown* components indefinitely (recall 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 $S' \in \text{Parents}(S)$ 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 OAC(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 again the join graph and synopses in Figure 1(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 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.

6.4 Experimental Analysis for OAC(k)

For the OAC experiments we used the join graph shown in Figure 2(c). 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 $[0, \dots, k]$. OAC(k) on $S_2.A$ eliminates $\mathcal{S}(S_2).No$ completely. But like with RIDS, total synopsis size increases over time because of $\mathcal{S}(S_2).Yes$. The sharp drop in synopsis size for $k = 20000$ in Figure 9 between 120000 and 130000 tuples is because the total number of tuples in S_2 crosses 20000 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 $k = 10000$ is less dramatic. Figure 10 shows the increase in memory overhead as the adherence to OAC decreases.

7 Simultaneously Exploiting Different k -Constraints

In Sections 3–5 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

Figure 9: Synopsis size using $OAC(k)$ as tuples arrive

Figure 10: Synopsis size using $OAC(k)$ as k varies

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

Table 1: Summary of synopsis reductions

the synopsis components that may be reduced or eliminated by the four constraint types independently, summarized in Table 1.

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 7.1 Consider the join graph and synopses in Figure 1(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

Figure 11: Effect of combining CA + OAC constraints

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 $S_3.B$ will not 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. $\square$

In Figure 11 we report an experimental result showing the scope of the effect. We used the join graph shown in Figure 2(a) for this experiment, with both filter predicates having 100% selectivity. We generated synthetic streams S_1 , S_2 , and S_3 with CA(5000) 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 . Figure 11 clearly demonstrates the benefits of using both constraints 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.

Example 7.2 Consider the join graph and synopses in Figure 1(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. $\square$

Figure 12: Effect of combining CA + RIDS constraints

In Figure 12 we report an experimental result illustrating the effect. We used the join graph shown in Figure 2(c) for this experiment, with the filter predicate having 10% selectivity. We generated synthetic streams S_1 and S_2 with $CA(0)$ 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 12 shows the total synopsis size after 500000 tuples have been processed. Once k increases beyond 11000 (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.

8 Violation of k -Constraints

So far we have assumed that all input streams satisfy all specified k -constraints at all points in time. In this section we consider how our query processing algorithm behaves when specified k -constraints are not adhered to. More precisely, we consider what happens when our algorithm assumes a k -constraint holds with $k = k_u$ (for k -used), while in reality the constraint holds with $k = k_r$ (for k -reality). If $k_u \geq k_r$, then our algorithm produces correct query results, but may store larger synopses than if it were using k_r . If $k_u < k_r$, then our algorithm may not produce correct query results at all points in time. However, our algorithm has two desirable properties in this case:

1. Because the SPJ queries we consider are monotonic, the algorithm can never produce “false positives,” i.e., the result produced is a subset of the actual result.

Figure 13: Approximation when $k_u < k_r$

2. The number of “false negatives” (missing result tuples) is roughly proportional to the deviation of k_r from k_u , although the actual value depends on the data and arrival characteristics of the streams.

In the remainder of this section we report on some experimental analysis of the false negative rate.

We used the join graph shown in Figure 2(b) for this experiment. We generated synthetic data for streams S_1 , S_2 , and S_3 so that both joins $S_1 \rightarrow S_3$ and $S_2 \rightarrow S_3$ satisfy RIDS(0). Streams S_1 and S_2 satisfy CA(k_r) on $S_1.A$ and $S_2.A$ respectively. Both the RIDS and CA constraints are used for synopsis reduction. In Figure 13 we fix k_u for the CA constraints at 1000 and vary k_r from 1000 to 2000. The y -axis shows the percentage of missed tuples in the query result after 80000 tuples have been processed. (After startup this percentage remains fairly constant as tuples are processed.) Plots are shown for two different distributions of the maximum clustering distance over distinct values of attribute A in the streams: the *uniform CA* distribution where the distances are distributed uniformly in $[0, \dots, k_r]$, and the *skewed CA* distribution where 90% of the distances are distributed uniformly in $[0, \dots, k_r/10]$ and the remaining 10% are distributed uniformly in $[0, \dots, k_r]$. That is, in the skewed distribution most tuples are clustered much more closely than the maximum k_r . Figure 13 shows that the correctness of the query result produced by the algorithm degrades gracefully with the deviation of k_r from k_u . In fact, for the skewed CA distribution, the percentage of false negatives remains around 4% even when k_r is twice the size of k_u .

9 Conclusions and Future Work

In this paper we have demonstrated the effectiveness of incorporating constraints into a data stream query processor in order to reduce the memory overhead required for continuous queries. We introduced the important concept of k -constraints, which are likely to hold in data stream environments even when strict constraints do not hold. We showed empirically that k -constraints are very effective at reducing the memory requirement in a wide variety of SPJ queries. Furthermore, we showed that precision of query results degrades gracefully in cases when constraints may be compromised. There are many avenues for continued work:

- Modify our algorithm to use other interpretations for k in our k -constraints. We are particularly interested the alternative of time-based (as opposed to tuple-based) constraints, but we also plan to investigate alternative tuple-based definitions of k . In all cases we suspect only minor changes to our approach are needed and performance results will be similar, but thorough investigation is warranted.
- Extend the class of queries for which our algorithm can exploit stream constraints, including queries with aggregation, difference, and sliding windows, and queries where not all joins are many-one. Doing so is a necessary prerequisite to the next step of fully integrating constraint specification and exploitation into the prototype stream processing system we are developing, since our prototype supports an expressive declarative query language.
- Consider the *constraint selection* problem identified in Section 7, where we saw that depending on the performance objective it may not always be beneficial to exploit all constraints on all streams simultaneously. We also plan to consider the computational overhead of exploiting constraints, in which case the constraint selection problem becomes even more central.
- Use our algorithm as a basis for providing approximate continuous query results when insufficient memory is available, even when we have no knowledge of constraints. Here we run into the challenging problem of allocating a limited amount of memory to different synopses in order to maximize result precision.
- Consider whether k -constraints may have other uses in a stream processing system, perhaps similar to those identified in the *punctuations* work of [TMSF02].

References

- [ABB⁺02] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 221–232, May 2002.
- [BBD⁺02] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of the 2002 ACM Symp. on Principles of Database Systems*, pages 1–16, June 2002.
- [BCL89] J. A. Blakeley, N. Coburn, and P. A. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. on Database Systems*, 14(3):369–400, 1989.
- [BW01] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, September 2001.
- [CCC⁺02] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams—a new class of data management applications. In *Proc. 28th Intl. Conf. on Very Large Data Bases*, August 2002.

- [CDTW00] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagraCQ: A scalable continuous query system for internet databases. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390, May 2000.
- [CF02] S. Chandrasekaran and M. Franklin. Streaming queries over streaming data. In *Proc. 28th Intl. Conf. on Very Large Data Bases*, August 2002.
- [CGM90] U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Trans. on Database Systems*, 15(2):162–207, 1990.
- [DGGR02] A. Dobra, J. Gehrke, M. Garofalakis, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 61–72, 2002.
- [DNS91] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equi-join algorithms. In *Proc. of the 1991 Intl. Conf. on Very Large Data Bases*, pages 443–452, September 1991.
- [GGZ01] P. Godfrey, J. Gryz, and C. Zuzarte. Exploiting constraint-like data characteristics in query optimization. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, May 2001.
- [GJM96] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *Proc. of the 1996 Intl. Conf. on Extending Database Technology*, pages 140–144, March 1996.
- [GK02] S. Guha and N. Koudas. Approximating a data stream for querying and estimation: Algorithms and performance evaluation. In *Proc. of the 2002 Intl. Conf. on Data Engineering*, pages 567–576, 2002.
- [GKMS01] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, pages 79–88, 2001.
- [GMLY98] H. Garcia-Molina, W.J. Labio, and J. Yang. Expiring data in a warehouse. In *Proc. of the 1998 Intl. Conf. on Very Large Data Bases*, pages 500–511, August 1998.
- [HWM98] S. Helmer, T. Westmann, and G. Moerkotte. Diag-join: An opportunistic join algorithm for 1:n relationships. In *Proc. of the 1998 Intl. Conf. on Very Large Data Bases*, pages 98–109, August 1998.
- [IFF⁺99] Z.G. Ives, D. Florescu, M. Friedman, A.Y. Levy, and D.S. Weld. An adaptive query execution system for data integration. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 299–310, June 1999.

- [JC00] T. Johnson and D. Chatziantoniou. Joining very large data sets. In Willem Jonker, editor, *Databases in Telecommunications*, volume 1819 of *Lecture Notes in Computer Science*, pages 118–132. Springer, 2000.
- [JMS95] H.V. Jagadish, I.S. Mumick, and A. Silberschatz. View maintenance issues for the Chronicle data model. In *Proc. of the 1995 ACM Symp. on Principles of Database Systems*, pages 113–124, May 1995.
- [KNV03] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Proc. of the 2003 Intl. Conf. on Data Engineering*, March 2003. (To appear).
- [MF02] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. of the 2002 Intl. Conf. on Data Engineering*, pages 555–566, February 2002.
- [MSHR02] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–60, June 2002.
- [MW⁺03] R. Motwani, J. Widom, et al. Query processing, approximation, and resource management in a data stream management system. In *Proc. First Biennial Conf. on Innovative Data Systems Research (CIDR)*, January 2003.
- [QGMW96] D. Quass, A. Gupta, I.S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc. of the 1996 Intl. Conf. on Parallel and Distributed Information Systems*, pages 158–169, December 1996.
- [RDH03] V. Raman, A. Deshpande, and J. Hellerstein. Using state modules for adaptive query processing. In *Proc. of the 2003 Intl. Conf. on Data Engineering*, March 2003. (To appear).
- [TMSF02] P. Tucker, D. Maier, T. Sheard, and L. Fegar. Enhancing relational operators for querying over punctuated data streams. *manuscript*, 2002. Available at <http://www.cse.ogi.edu/dot/niagara/pstream/punctuating.pdf>.
- [UFA98] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 130–141, June 1998.
- [UW97] J.D. Ullman and J. Widom. *A First Course in Database Systems*. Prentice Hall, Upper Saddle River, New Jersey, 1997.
- [VN02] S. Viglas and J. Naughton. Rate-based query optimization for streaming information sources. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 37–48, June 2002.