Characterizing Memory Requirements for Queries Over Continuous Data Streams

ARVIND ARASU, BRIAN BABCOCK, SHIVNATH BABU, JON McALISTER, and JENNIFER WIDOM
Stanford University, Stanford, California

This article deals with continuous conjunctive queries with arithmetic comparisons and optional aggregation over multiple data streams. An algorithm is presented for determining whether or not any given query can be evaluated using a bounded amount of memory for all possible instances of the data streams. For queries that can be evaluated using bounded memory, an execution strategy based on constant-sized synopses of the data streams is proposed. For queries that cannot be evaluated using bounded memory, data stream scenarios are identified in which evaluating the queries requires memory linear in the size of the unbounded streams.

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

General Terms: Algorithms, Performance, Theory

Additional Key Words and Phrases: Continuous queries, memory requirement, streams

1. INTRODUCTION

In many recent applications, data takes the form of continuous, unbounded *data streams*, rather than finite stored data sets [Gehrke 2003; Golab and Ozsu 2003; Babcock et al. 2002]. Examples of data streams include stock ticks in financial applications, performance measurements in network monitoring and traffic management, log records or click-streams in Web tracking and personalization, data feeds from sensor applications, network packets and messages in firewall-based security, call detail records in telecommunications, and so on.

Due to their continuous nature data streams are typically queried using long-running *continuous* queries rather than the traditional *one-time* queries. A continuous query is a query that is logically issued once but run forever. At any point of time, the answer to a continuous query reflects the elements of the input data streams seen so far, and the answer is updated as new stream elements

This work was supported by the National Science Foundation under grants IIS-9817799 and IIS-0118173 and by an Okawa Foundation Research Grant.
Authors' address: Stanford University, Room 412, Gates Building, 353 Serra Mall, Stanford, CA 94305; email: arvinda@cs.stanford.edu.

arrive. For example, continuous queries over network packet streams can be used to monitor network behavior to detect anomalies (e.g., link congestion) and their cause (e.g., hardware failure, intrusion, denial-of-service attack) [Cranor et al. 2003]. In financial applications, continuous queries over stock tick data, news feeds, and historical company data can be used to monitor trends and detect fleeting opportunities [Traderbot 2003].

Continuous-query processing introduces several interesting new challenges not seen in traditional one-time query processing. This article addresses one of the most fundamental challenges that stems from the unbounded nature of input streams—characterizing (worst-case) memory requirements of continuous queries. A query (since this article deals only with continuous queries, we omit the term "continuous" in the rest of the article) typically has to store some state based on the input stream elements seen so far. For example, a query that computes the join of two streams has to remember all the elements of both the streams seen so far, since a new stream element could join with any of the previous elements of the other stream (window-based joins are discussed in Section 1.3). On the other hand, a simple filter query does not need to maintain any historical state. Note that we use the term "memory" to mean any sort of storage; our results apply equally to systems resident in main memory and to those that use secondary storage.

This article proves two interesting results on memory requirements of queries. First, it proves that all queries we consider (conjunctive queries with optional aggregation) fall into just two classes based on their asymptotic memory requirements—queries that can be evaluated with bounded memory (i.e., a finite amount of memory), and those that require memory that grows linearly with the input data. Therefore, most of the article is focused on distinguishing the class of queries that can be evaluated with bounded memory from those that cannot. Note that we are concerned with exact evaluation of queries in this article: there exist many queries with aggregates, those involving quantiles for example, that can be approximately evaluated with memory that grows logarithmically with input data [Greenwald and Khanna 2001].

On first thought, it might seem that only simple filter queries can be evaluated with bounded memory. The second interesting result shown in this article is that there is a relatively large class of queries, including some queries that join arbitrary number of streams, that can be computed with bounded memory. However, the class of queries computable in bounded memory does not admit a trivial characterization as we illustrate in the following examples.

1.1 Examples

Our set of example queries is shown in Table I. Two data streams, $S(A, B, C)$ and $T(D, E)$, are used in the example queries. The domain of attributes A–E is the set of integers. We use standard relational algebra with a few minor modifications to represent the queries. Specifically, π denotes the duplicate-eliminating projection operator, $\dot{\pi}$ the duplicate-preserving projection operator, σ the selection operator, and $\times$ the cross-product operator. We use Π as a place-holder for one of π and $\dot{\pi}$. The answer to a continuous query at any point

Table I. Example Queries Over Data Streams $S(A, B, C)$ and $T(D, E)$

		Bounded-memory computable?	
		$\Pi = \dot{\pi}$	$\Pi = \pi$
Q_1	$\Pi_A(\sigma_{(A>10)}\ (S))$	Yes	No
Q_2	$\Pi_A(\sigma_{(A=D)}\ (S \times T))$	No	No
Q_3	$\Pi_A(\sigma_{(A=D)\wedge(A>10)\wedge(D<20)}\ (S \times T))$	Yes	Yes
Q_4	$\Pi_A(\sigma_{(B<D)\wedge(A=10)}\ (S \times T))$	No	Yes
Q_5	$\Pi_A(\sigma_{(B<D)\wedge(C<E)\wedge(A=10)}\ (S \times T))$	No	No
Q_6	$\Pi_A(\sigma_{(B<D)\wedge(C<E)\wedge(B<E)\wedge(C<D)\wedge(A=10)}\ (S \times T))$	No	Yes
Q_7	$\Pi_A(\sigma_{(B<D)\wedge(D>10)\wedge(B<20)\wedge(A=10)}\ (S \times T))$	Yes	Yes

of time is the answer using relational semantics over the bag of input stream tuples seen so far. We assume that the query evaluation system does not need to store the answer to a query but can "stream" the tuples in the answer as they are generated. This approach does not cause any ambiguity for the monotonic queries that we are considering. (Monotonic queries are queries for which a new input tuple never causes the deletion of an existing output tuple, thus the set of output tuples grows continuously as more input tuples arrive [Ullman 1988].)

Consider Query Q_1, $\Pi_A(\sigma_{(A>10)}\ (S))$, a selection and projection over one data stream. When the projection is duplicate-preserving ($\Pi = \dot{\pi}$), Q_1 is a simple filter on S and can be evaluated by tuple-at-a-time processing of the stream. Thus, it can always be evaluated without using any extra memory for storage of stream tuples or intermediate state. If the projection in Q_1 is duplicate-eliminating ($\Pi = \pi$), we need to keep track of each distinct value of A greater than 10 in S so far, in order to eliminate duplicates in the answer. In this case, there is no finite bound on the amount of memory required for evaluating this query over all possible instances of Stream S. Query Q_2, $\Pi_A(\sigma_{(A=D)}\ (S \times T))$, is an equi-join over streams S and T. In order to correctly evaluate Query Q_2, it is necessary to store every distinct value of A seen so far in Stream S and every distinct value of D seen so far in Stream T, which requires unbounded space.

Query Q_3, $\Pi_A(\sigma_{(A=D)\wedge(A>10)\wedge(D<20)}\ (S \times T))$, is similar to Query Q_2 but has two additional selection predicates on attributes A and D. Observe that a tuple of stream S can join with a tuple of stream T only if their corresponding A and D values lie within the interval [11, 19]; this observation can be used to evaluate Query Q_3 in bounded memory. We briefly describe an evaluation strategy for computing Q_3 in bounded memory. The evaluation strategies that we describe for bounded-memory evaluation of queries involve keeping constant-sized *synopses* for each stream S and T. A synopsis for a stream is a summary

of the tuples of the stream seen so far that contains sufficient state information to compute future answers correctly. For the duplicate-preserving case of Q_3, the synopsis for S contains for each value v in the interval [11, 19] the count of tuples seen so far with $A = v$. Similarly, the synopsis for T contains for each value $v \in [11, 19]$ the count of tuples with $D = v$. It is easy to see that the above synopses are sufficient to compute Q_3 correctly. For example, consider a new tuple arriving on stream S with $A = v$. If v does not lie in the interval [11, 19], then the new tuple cannot join with any tuple of T and it can be ignored. If v lies in [11, 19], the exact number of past T tuples that the new tuple joins with is stored in the synopsis for T, and that many copies of the $\langle v \rangle$ tuple are generated in the output. Note that it was crucial that the output project list of Q_3 only has attribute A and not, for instance, A and C which would have made the query not computable in bounded memory. A similar evaluation strategy can be designed to compute Q_3 in the duplicate-eliminating case.

Consider the series of queries Q_4, Q_5, Q_6 for duplicate-eliminating projection. Note that Q_5 is derived from Q_4 by adding an additional predicate $(C < E)$ and Q_6 is derived from Q_5 by adding two additional predicates $(B < E)$ and $(C < D)$. While Q_4 and Q_6 are computable in bounded memory, Q_5 is not. Query Q_4 can be evaluated by maintaining as synopsis of S the minimum value of attribute B among all tuples of S (so far) which have $A = 10$, and maintaining as synopsis of T, the maximum value of attribute D among all tuples of T so far. In order to see why the above synopses are sufficient, consider the arrival of a new tuple t on Stream T. Assume t joins with some past tuple s of S with $A = 10$. From the join condition it follows that $s[B] < t[D]$. Clearly, t also joins with that tuple of S that has the minimum value of attribute B among all tuples with $A = 10$. Therefore, we can determine if t joins with some tuple of S by just checking if $t[D]$ is strictly larger than the value stored in the synopsis for S. Similarly, Q_6 can be evaluated by maintaining as synopsis of S the value $\min\{\max\{s[B], s[C]\}\}$ over all tuples s of S so far, and maintaining the value $\max\{\min\{t[D], t[E]\}\}$ over all tuples t of T so far. We leave it to the reader to verify that the above synopses are sufficient to correctly evaluate Q_6.

None of the queries Q_4, Q_5, Q_6 can be computed in bounded memory for duplicate-preserving projection. Note that for duplicate-preserving projection it is not sufficient to just know whether or not a new stream tuple joins with any past tuples—we also need to determine the exact number of past tuples with which it joins. One can easily verify that the synopses described earlier for queries Q_4 and Q_6 for duplicate-eliminating projection cannot be used to determine the exact number of past tuples with which a new stream tuple joins.

1.2 Contributions

As the examples of the previous section suggest, the problem of determining the bounded-memory computability of continuous queries is nontrivial. We make the following contributions on bounded-memory computability of queries:

—We consider conjunctive queries with arithmetic comparisons and optional aggregation over multiple data streams, and we specify an algorithm that

determines whether or not any given query can be evaluated using a bounded amount of memory for all possible instances of the data streams.

—When a query can be evaluated using bounded memory, we produce an execution strategy based on constant-sized synopses of the data streams, characterizing the memory requirements of the query for all possible instances of the streams.

—When a query cannot be evaluated using bounded memory, for any query execution strategy, we identify specific instances of input streams for which the strategy requires memory at least linear in the sum of lengths of the input streams.

1.3 Sliding Windows

Continuous queries over streams may apply sliding windows, either to reflect the semantics of the application or in some cases simply to solve the unbounded memory problem [Babcock et al. 2002]. Continuous query languages may support *row-based windows*, specifying that a fixed number of the most recent stream tuples are considered instead of the entire stream history, and *time-based windows*, specifying that only stream tuples that have arrived within a specified time are considered, instead of the entire stream history.

A row-based window on a stream S by definition implies that we need only a bounded synopsis for S. Thus, if in query Q every stream has a row-based window, clearly Q is computable with bounded memory. We cannot necessarily bound the state required to store a time-based window, since arbitrary numbers of stream tuples may arrive within a finite time interval. Furthermore, we are interested in studying the case where streams are not required to be bounded by applying windows. In addition to handling queries without windows, our techniques can determine whether queries with time-based windows are bounded-memory computable and suggest appropriate bounded synopses for this case.

1.4 Overview and Organization

Section 2 briefly surveys related work. Section 3 formally defines data streams, continuous queries over data streams, the query execution model, and the problem statement. Section 4 introduces notation and terminology used in the rest of the article.

Sections 5, 6 and 7 deal with bounded-memory computability of a well-known class of queries, namely, Select-Project-Join (SPJ) queries without self-join. All the example queries used in Section 1.1 belong to this class. Since it seems hard to determine bounded-memory computability of arbitrary SPJ queries directly (as the examples suggest), we take an indirect approach. We first rewrite a given SPJ query Q as a union of queries each of which belongs to a special class that we call *Locally Totally Ordered* queries, or *LTO* queries for short. Next, we check if each LTO query in the union is bounded-memory computable. An LTO query has a special structure that makes it easier to determine if it is bounded-memory computable. The original query Q is bounded-memory computable iff all the LTO queries in the union are bounded-memory computable. This

LTO-rewriting approach to determining bounded-memory computability of SPJ queries is presented in Section 5.

The basic LTO-rewriting approach to determining bounded-memory computability requires time exponential in the size of the query. Section 6 refines this approach and presents an efficient polynomial-time algorithm to determine if an SPJ query is computable in bounded-memory. Section 7 presents an execution strategy for bounded-memory computable queries.

Finally, Section 8 extends our results to a larger class of queries, namely, SPJ queries that may include self-joins and an optional aggregation.

Most of the proofs in the article are relegated to the *electronic appendix*.

2. RELATED WORK

This article extends an original short article of the same title [Arasu et al. 2002a]. Material in this article that is not in Arasu et al. [2002a] includes a memory-efficient execution strategy for queries that can be executed in a bounded amount of memory, and extensions to the class of queries under consideration to support grouping and aggregation. This work was done as part of the STREAM [2003] (STanford stREam datA Manager) project, whose goal is to develop a general purpose Data Stream Management System (DSMS) that supports continuous queries.

Continuous queries were first introduced by Terry et al. [1992]. Babcock et al. [2002] discuss the new research directions in continuous-query processing. There are several ongoing research projects aimed at developing continuous query processing systems: Aurora [Carney et al. 2002], NiagaraCQ [Chen et al. 2000], STREAM [2003], and TelegraphCQ [Chandrasekharan et al. 2003]. More specific related work from the above projects is described below.

To the best of our knowledge, no past work on continuous query processing has considered the problem of characterizing memory requirements of queries statically. Some recent work has explored using constraints over streams to minimize memory requirements. Babu and Widom [2002] identify several static constraints and present techniques that exploit the constraints to reduce memory requirements of continuous queries. Tucker et al. [2003] suggest a mechanism for embedding dynamic constraints within streams using *punctuations*. This article differs from the above work in that it studies the worst-case memory requirement without assuming any constraints over the input streams.

Some of the evaluation strategies that we propose for bounded-memory computable queries are similar to those used in Chomicki [1995]. However, Chomicki [1995] does not characterize the memory requirements of its evaluation strategies.

There has been lot of recent work on non-memory-related aspects of continuous-query processing such as adaptivity, approximation and sharing state among related queries. Madden et al. [2002] study adaptivity and sharing in continuous queries. Chandrasekharan and Franklin [2002] consider a variant of continuous queries where the answer to the query need not be computed continuously but only intermittently on demand. Shah et al. [2003] propose a new adaptive operator for parallelizing query evaluation.

A large body of work on data streams has focused on specific tasks over streams, such as computing quantiles or aggregates. Usually, these tasks require unbounded memory, and most of the work in this category provides approximate answers, often with static guarantees on the accuracy of the answer. Manku et al. [1999] consider maintaining approximate quantiles over streams; Vitter [1985] studies the problem of maintaining fixed-size random samples over streams; Alon et al. [1999] consider the problem of computing frequency moments; Feigenbaum et al. [1999] compute l_1-differences between streams; Gilbert et al. [2002] and Guha et al. [2001] consider the problem of maintaining histograms over data streams; Datar et al. [2002] maintain statistics for sliding windows over data streams; Babcock et al. [2002] consider sampling for the same scenario; Gehrke et al. [2001] develop histogram-based techniques to provide approximate answers for a specific class of queries called correlated aggregate queries; and Dobra et al. [2002] use synopses based on *sketches* [Alon et al. 1999] to provide approximate answers to aggregates over joins.

3. QUERY LANGUAGE, EXECUTION MODEL, AND PROBLEM STATEMENT

We formally define our model for data streams, our initial query class, and the semantics used for queries over streams. We then formalize our query execution model and state the problem of determining whether or not a query can be evaluated with bounded memory.

3.1 Continuous Data Streams

A *continuous data stream* (hereafter simply a *stream*) is a potentially infinite stream of relational tuples. Each stream has a fixed schema, that is, a known finite set of attributes. We assume that the domain of each attribute is the set of integers. All the results in the article easily extend to any discrete and totally ordered domain for attributes. We assume streams are *bags*, that is, the same tuple can appear any number of times in a stream. The *instance* of a stream at any point in time is the bag of stream tuples seen until that point.

We assume that streams are generated by an independent source, meaning that a query evaluation algorithm has no control over the streams. This assumption has two important implications. First, a stream can be read only once from its source, and it is read in the order generated by the source. A query evaluation algorithm can, of course, store a part of the stream in its local memory and access it subsequently. Second, if a query involves multiple streams, an evaluation plan cannot make any assumptions about the relative order in which the tuples of different streams are read. We use the term *interleaving* to denote the exact interleaved sequence in which tuples of different input streams are read. Thus, a particular input instance consists of both the input streams and a particular interleaving.

3.2 Queries over Continuous Data Streams

A query in a traditional database is specified over finite data sets and the query answer is a function of the entire input data. Since data streams are potentially infinite and continuously arriving, streams are typically queried

using *continuous queries*, where the answer to a query at any point in time is a function of the input streams seen so far. The answer changes continuously as new input stream tuples arrive.

Languages and semantics for continuous queries is an ongoing topic of research [Arasu et al. 2002b; Chandrasekharan et al. 2003]. For the purposes of this article, we consider a simple class of continuous queries whose semantics are relatively easy to define. The class of queries that we consider is Select-Project-Join (SPJ) queries with an optional aggregation.

We use standard relational symbols with a few modifications to represent continuous SPJ queries. We use the relational symbols σ and $\times$ to denote selection and Cartesian product operators, respectively. We use π and $\dot{\pi}$ to denote duplicate-eliminating and duplicate-preserving projection operators, respectively. We use the "variable" symbol Π to represent one of the two projection operators. Thus, an SPJ query is of the form $\Pi_L(\sigma_P(S_1 \times S_2 \times \cdots \times S_n))$, where L is the list of projected attributes, P is the selection predicate, and $S_1, S_2, \ldots, S_n$ denote the input data streams. We restrict ourselves to SPJ queries where the selection predicate P is a conjunction of atomic predicates. An atomic predicate is of the form $(S_i.A\ Op\ S_j.B)$ (where i and j can be the same or different) or of the form $(S_i.A\ Op\ k)$, where Op is one of the comparison operators in $\{<, =, >\}$ and k is some constant integer. Our results can be extended in a straightforward way to include the operators $\{\leq, \geq\}$. An atomic predicate of the form $(S_i.A\ Op\ S_i.B)$ or $(S_i.A\ Op\ k)$ involving attributes of just one stream is called a *filter*; otherwise, it is called a *join*. To simplify the presentation, we start off assuming that there are no self-joins in the query, that is, $S_i \neq S_j$ for $i \neq j$. We extend our results to include self-joins in Section 8.1.

The semantics of continuous SPJ queries without aggregation is a simple extension of the semantics for the traditional relational case. The output of a continuous SPJ query is also a stream, and at any point of time the bag of tuples in the output stream corresponds to the result of applying the SPJ query using traditional relational semantics on the bag of tuples of the input streams that have arrived so far. Note that this semantics is unambiguous since SPJ queries without aggregation are all *monotonic* [Ullman 1988].

The representation and semantics of continuous SPJ queries with aggregation is presented in Section 8.2. In all other sections of the article, an SPJ query implicitly refers to an SPJ query without aggregation.

3.3 Execution Model and Problem Specification

We assume that the query evaluation environment has access to some local memory that can be used, for example, to store some information about the input streams seen so far. We say that a *unit* of the memory can store one attribute value or a count.[1] We are not concerned with memory for storage of

[1]We assume that a count only takes up one unit of memory although the number of bits necessary to represent a count grows logarithmically with the number of items being counted. In practice, no count is likely to require more than one or two words of memory on any modern computer architecture.

the answers to SPJ queries since the answer is also a data stream (except for aggregate queries, covered in Section 8.2).

The goal of this article is to characterize the worst-case memory requirements of queries over all possible input stream instances and their interleaving (Section 3.1). It turns out that the worst-case memory requirement of all the queries that we consider in this article falls into two asymptotic classes: linear in the size of the input, and bounded by a constant. Therefore, it suffices to characterize the class of queries that can be evaluated in a bounded amount of memory.

Definition 3.1. A query is *computable in bounded memory* if there exists a constant M and an algorithm that evaluates the query using fewer than M units of memory for all possible instances and interleavings (Section 3.1) of the input streams of the query.

We focus primarily on the problem of identifying exactly the above class of queries. The proof that any query that is not bounded-memory computable requires space that is linear in the size of the input is presented in the *electronic appendix*.

4. PRELIMINARIES AND DEFINITIONS

This section introduces notation and terminology and reviews some basic concepts from discrete mathematics that are used in our results.

As described in Section 3, we initially consider SPJ queries of the form $\Pi_L(\sigma_P(S_1 \times S_2 \times \cdots \times S_n)$, where $\Pi = \pi$ for duplicate-eliminating projection and $\Pi = \dot{\pi}$ for duplicate-preserving projection. When the streams and the list of projected attributes are not important to the discussion, we may write a query Q as $Q(P)$, where P is the selection predicate. This notation is also used to represent two queries that are identical except for their selection predicate. For example, query $Q(P_2)$ is obtained from query $Q(P_1)$ by just replacing the selection predicate P_1 with the predicate P_2. For convenience, we represent the selection predicate as a set instead of a conjunction of atomic predicates. Recall from Section 3 that we only consider queries whose selection predicate is a conjunction of atomic predicates.

A set of atomic predicates P is *satisfiable* if there exists some assignment of integer values to the attributes in P that makes every predicate in the set P evaluate to true. For example, the set of atomic predicates $\{(A < B), (B < C), (C < A)\}$ is not satisfiable. Note that although the general problem of boolean expression satisfiability is intractable, there exist efficient algorithms to check satisfiability for a conjunction (and by our convention, a set) of atomic predicates [Ullman 1989]. Observe that any query $Q(P)$ with an unsatisfiable selection predicate P has an empty output stream and therefore is trivially computable in bounded memory. In the rest of the article, we assume that the selection predicates of the queries considered are satisfiable unless mentioned otherwise.

Table II. Notation for Some Query Q and Stream S

$\mathcal{S}(Q)$	set of streams that appear in Q
$\mathcal{C}(Q)$	set of constants that appear in Q
$\mathcal{A}(S)$	set of attributes in Stream S
$\mathcal{A}(Q)$	set of attributes in all streams in Q, i.e., $\bigcup_{S \in \mathcal{S}(Q)} \mathcal{A}(S)$
$\mathcal{E}(Q)$	set of elements in Q, i.e., $\mathcal{A}(Q) \cup \mathcal{C}(Q)$

We use the term, *element*, to refer to either a constant integer value or an attribute of a stream. Table II lists various notation related to constants, attributes and elements of a query that we use in the rest of the article. For example, for Query Q_3 in Table I, $\mathcal{S}(Q_3) = \{S, T\}$, $\mathcal{C}(Q_3) = \{10, 20\}$, $\mathcal{E}(Q_3) = \{10, 20, A, B, C, D, E\}$.

The *transitive closure* of a set of atomic predicates P, denoted P^+, is the set of atomic predicates logically implied by the predicates in P. The linear ordering of integers is implicitly used in determining transitive closure. For example, the transitive closure of the set of atomic predicates $\{(A < 5), (8 < B)\}$ is the set $\{(A < 5), (A < 8), (5 < B), (8 < B), (A < B)\}$. For simplicity, we assume that the transitive closure P^+ contains atomic predicates only involving elements occurring in P. Continuing the previous example, the predicate $(A < 6)$ does not occur in the transitive closure although it is logically implied by the set of predicates $\{(A < 5), (8 < B)\}$. This assumption ensures that the transitive closure of a finite set of atomic predicates is also finite. Note that two queries $Q(P)$ and $Q(P')$ are equivalent whenever $P^+ = (P')^+$.

Definition 4.1. An inequality predicate $(e_1 < e_2) \in P$ is said to be *redundant* in P if one of the following three conditions hold: (1) there exists an element e such that $(e_1 < e) \in P^+$ and $(e < e_2) \in P^+$; (2) there exists a constant integer k such that $(e_1 = k) \in P^+$ and $(k < e_2) \in P^+$; (3) there exists a constant integer k such that $(e_1 < k) \in P^+$ and $(e_2 = k) \in P^+$.

For example, the predicate $(A < C)$ is redundant for both the sets $P = \{(A < B), (B < C), (A < C)\}$ and $P = \{(A < C), (A = 5), (C > 5)\}$. Removing all the redundant predicates from P leaves its transitive closure unchanged, and therefore it is sufficient to consider only nonredundant predicates of a query in determining its bounded-memory computability (Section 5). The converse is not true however: any atomic predicate whose removal leaves the transitive closure unchanged is not necessarily redundant in accordance with our definition. For example, none of the predicates in the set $\{(A = B), (A < C), (B < C)\}$ are redundant, while removing the predicate $(A < C)$ still leaves the transitive closure unchanged.

Our definition of redundancy depends on constant values in a subtle way: for example, the predicate $(A < C)$ is redundant for the set $P = \{(A = 5), (5 < C), (A < C)\}$, while it is not so for the set $P = \{(A = B), (B < C), (A < C)\}$. This subtlety arises because evaluating filter conditions requires no additional memory (see Theorem 5.3); exact details should become clearer from our characterization of bounded-memory computable queries in Section 5.

Definition 4.2. For a given set of atomic predicates P, the set of atomic predicates *induced* by a set of elements E, denoted $\mathrm{IND}(P, E)$, is the set of predicates in P^+ that involve only elements in E.

Definition 4.3. A set of elements E is *totally ordered* by a set of predicates P if for any two elements e_1 and e_2 in E, exactly one of the three atomic predicates $(e_1 < e_2)$ or $(e_1 = e_2)$ or $(e_1 > e_2)$ is in P^+.

Definition 4.4. For a given set of predicates P, the equality predicates in the set partition the elements in P into *equivalence classes*: two elements e_1 and e_2 belong to the same equivalence class if $(e_1 = e_2) \in P^+$.

Definition 4.5. Given a set of predicates P, an attribute A is *lower-bounded* if there exists an atomic predicate $(A > k) \in P^+$, or an atomic predicate $(A = k) \in P^+$ for some constant k. Similarly, an attribute A is *upper-bounded* by P if there exists an atomic predicate $(A < k) \in P^+$, or an atomic predicate $(A = k)$. An attribute is *bounded* if it is both upper-bounded and lower-bounded and *unbounded* otherwise.

Note that an attribute that is upper-bounded (respectively, lower-bounded) but not lower-bounded (respectively, upper-bounded) is unbounded by our definition.

Definitions 4.1–4.5, when used in the context of a query $Q(P)$, implicitly refer to the selection condition P of the query. For example, the term *bounded attributes of a query* refers to the attributes of the query that are bounded given the selection predicate of the query.

5. BOUNDED-MEMORY COMPUTABILITY OF SPJ QUERIES

This section provides a characterization for bounded-memory computable SPJ queries.

In order to determine the bounded-memory computability of SPJ queries, we use a special class of queries that we call *Locally Totally Ordered* queries, or *LTO* queries for short. Informally, an LTO query imposes, for each stream in the query, a total ordering of the attributes of the stream and the constants in the query. Any SPJ query can be converted into an LTO query by repeatedly adding filter conditions to its selection predicate; adding different sets of filter conditions results in different LTO queries. We show that an SPJ query is bounded-memory computable iff all the LTO queries derived from it, by adding filter conditions, are also bounded-memory computable. The special structure of LTO queries makes it easier to determine if they are bounded-memory computable.

Definition 5.1. An SPJ query $Q(P)$ is *Locally Totally Ordered* if, for every $S \in \mathcal{S}(Q)$, the set of elements $(\mathcal{A}(S) \cup \mathcal{C}(Q))$ is totally ordered (Definition 4.3) by P.

LTO queries are "maximal filter queries" in a certain sense: adding additional filter conditions involving only $\mathcal{E}(Q)$ to the selection predicate of an LTO query Q either results in an equivalent query, or makes the selection predicate of the

resulting query unsatisfiable. Consider an SPJ query $Q(P)$ over two streams, $S(A, B, C)$ and $T(D, E)$, where $P = \{(A = 10), (B < A), (A < C), (D < E)\}$. The set of elements $(\mathcal{A}(S) \cup \mathcal{C}(Q)) = \{A, B, C, 10\}$ is totally ordered since $B < A = 10 < C$; however, the set of elements $(\mathcal{A}(T) \cup \mathcal{C}(Q)) = \{D, E, 10\}$ is not totally ordered, since D and E are not comparable to 10. Therefore, $Q(P)$ is not an LTO query. However, the query $Q(P \cup \{(E < A)\})$ formed by adding an additional predicate to P is an LTO query, since now the set of elements $\{D, E, 10\}$ is totally ordered: $D < E < A = 10$.

We can form an LTO query from an SPJ query $Q(P)$ by adding filter conditions to the selection predicate P to ensure that for each stream S appearing in Q, the set $\mathcal{A}(S) \cup \mathcal{C}(Q)$ is totally ordered. The LTO query formed in this way is said to be derived from Q. The formal definition of derived LTO queries takes into account the equivalence of queries with the same transitive closure of their selection predicates.

Definition 5.2. An LTO Query $Q(P_L)$ is said to be *derived* from an SPJ Query $Q(P)$ if $(P_L)^+ = (P \cup F)^+$, where F is some arbitrary set of filter predicates.

For instance, from the example query $Q_3 = \Pi_A(\sigma_{(A=D)\wedge(A>10)\wedge(D<20)}\ (S \times T))$ in Table I, we can derive an LTO query by adding the filter predicates $\{(B > 20), (C < 10), (E = D)\}$ to the selection predicate.

The bounded-memory computability of an SPJ query is directly related to the bounded-memory computability of the set of LTO queries derived from it, as formalized by the following theorem.

THEOREM 5.3. *An SPJ query $Q(P)$ is bounded-memory computable iff all LTO queries derived from it are bounded-memory computable.*

PROOF. Assume Q is bounded-memory computable. Then any query $Q(P \cup F)$, where F is a set of filter conditions, is also bounded-memory computable. A straightforward bounded-memory evaluation strategy for $Q(P \cup F)$ is as follows: use a bounded-memory evaluation strategy for Q, and check the additional filter conditions F on the output of Q, which can be done without any additional memory. Since any LTO query derived from Q is also formed by adding just filter conditions to P, it is bounded-memory computable as well. This completes the "only-if" part of the proof.

Since all LTO queries derived from Q involve just the elements of Q, there is only a finite number of such LTO queries (since we do not differentiate queries with the same transitive closure of their selection predicates). Let $Q_1, \ldots, Q_m$ be an enumeration of all LTO queries derived from Q. We claim that Q is equivalent to the union of the LTO queries derived from it, that is, $Q \equiv \bigcup_{1 \le i \le m} Q_i$. The union operator $\bigcup$ is duplicate-preserving if the projection operator of Q is duplicate-preserving, and duplicate-eliminating if the projection of Q is duplicate-eliminating. The following example illustrates this equivalence. Its formal proof is generalized easily from the example and is not presented in the article.

Consider the SPJ query $Q(P) = \Pi_A(\sigma_{(B=C)\wedge(A=10)}\ (S \times T))$, involving two streams $S(A, B)$ and $T(C)$. Three LTO queries Q_1, Q_2, and Q_3 are derived from

$Q(P)$; these are formed by adding predicates $(B < 10)$, $(B = 10)$, and $(B > 10)$, respectively, to the selection predicate P. Consider a tuple s of stream S and a tuple t of stream T that successfully join in Query Q, that is, satisfy all the predicates in P. Then, depending on the whether the value of $s[B]$ $(= t[C])$ is less than, equal to, or greater than 10, tuples s and t join in exactly one of Q_1–Q_3, and fail to do so in the other two. Conversely, any two tuples s of stream S and t of stream T join in at most one of the queries Q_1–Q_3, and if they join in some query they also join in Q (since the predicates of Q are a subset of the predicates of any of Q_1–Q_3). Therefore, Q is equivalent to the duplicate-preserving (when $\Pi = \dot{\pi}$) or duplicate-eliminating (when $\Pi = \pi$) union of Q_1–Q_3. Note that when $\Pi = \pi$, although each derived LTO query is duplicate-eliminating, the same output tuple may be produced by different LTO queries, and hence a duplicate-eliminating union is necessary to remove the duplicates from the outputs of these queries.

The equivalence $Q \equiv \bigcup_{1 \le i \le m} Q_i$ can be used to derive a bounded-memory evaluation strategy for Q, if all the LTO queries Q_i $(1 \le i \le m)$ are bounded-memory computable. The evaluation strategy is simpler when Q has a duplicate-preserving projection: evaluate in bounded-memory each LTO query $Q_1, \ldots, Q_m$, and transfer any output of these queries into the output of Q. When Q has a duplicate-eliminating projection, the evaluation strategy, as before, evaluates each LTO query independently. However, in addition, it remembers the set of all output tuples produced so far by any LTO query, and uses this set to remove duplicates. Remembering this set of output tuples does not require unbounded memory, since, as we will prove in Theorem 5.10, the number of output tuples of a bounded-memory computable LTO query with duplicate-eliminating projection is bounded. □

Theorem 5.3 reduces the problem of determining if an SPJ query is bounded-memory computable to that of determining if an LTO query is bounded-memory computable. The bounded-memory computability of an LTO query depends on two special sets of attributes identified by the selection predicate of the query.

Definition 5.4. Consider a query $Q(P)$ and a stream $S_i \in \mathcal{S}(Q)$. *MaxRef*(S_i) is the set of all unbounded attributes (Definition 4.5) A of S_i that participate in a nonredundant (Definition 4.1) inequality join $(S_j.B < S_i.A)$, $i \neq j$, in P^+. *MinRef*(S_i) is similarly defined as the set of all unbounded attributes A of S_i that participate in a nonredundant inequality join of the form $(S_i.A < S_j.B)$, $i \neq j$, in P^+.

Note that *MaxRef* and *MinRef* of a stream depend only on the selection predicate of the query. In particular, they do not depend on the query being an LTO query.

Example 5.5. Figure 1 illustrates *MaxRef* and *MinRef* for an example set of predicates involving attributes from three streams S_1, S_2, and S_3. An inequality join predicate between two elements is represented in Figure 1 by a directed edge between the elements; for example, the edge from 10 to E

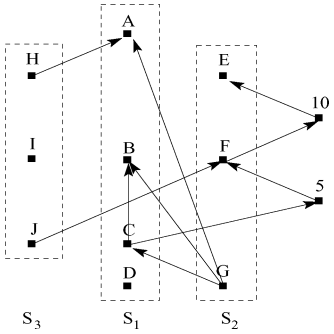

Fig. 1. *MaxRef* and *MinRef*.

represents the atomic predicate $(10 < E)$. Attributes belonging to the same stream are indicated using an enclosing rectangle; for example, the attributes of S_1 are A, B, C, and D.

For stream S_1, $MaxRef(S_1) = \{C, A\}$, due to the predicates $(G < C)$ and $(H < A)$, and $MinRef(S_1) = \phi$. Attribute B does not appear in $MaxRef(S_1)$ since the predicate $(G < B)$ is redundant. For stream S_2, $MaxRef(S_2) = \phi$ and $MinRef(S_2) = \{G\}$. Attribute F does not appear in $MaxRef(S_2)$ since F is bounded. For stream S_3, $MaxRef(S_3) = \phi$, and $MinRef(S_3) = \{H, J\}$.

We consider the cases of LTO queries with duplicate-preserving projection and duplicate-eliminating projection separately.

5.1 LTO Queries with Duplicate-Preserving Projection

Duplicate-preserving LTO queries that involve only one stream can always be computed in bounded memory, since without joins every predicate is a filter and can be computed one tuple at a time. The following theorem characterizes bounded-memory computability for queries involving more than one stream.

THEOREM 5.6. *Let $Q = \dot{\pi}_L(\sigma_P(S_1 \times \cdots \times S_n))$ be an LTO query where $n > 1$. Q is bounded-memory computable (Definition 3.1) iff:*

C1: Every attribute in the project list L is bounded.

C2: For every equality join predicate $(S_i.A = S_j.B)$, where $i \neq j$, $S_i.A$ and $S_j.B$ are both bounded.

C3: $|MaxRef(S_i)| = |MinRef(S_i)| = 0$ for $i = 1, \ldots, n$.

This section informally discusses the ideas behind Theorem 5.6. A formal proof for the "if" part can be derived in a straightforward way from the discussion here, and is not presented; a formal proof for the "only-if" part is given in the *electronic appendix*.

Informally, conditions C1–C3 state that, if we ignore the attributes that are involved only in filter conditions (since we can handle filter conditions with no memory), all the attributes that influence the output of Q are bounded. Conditions C1 and C2 directly state that any attribute in the project list or any

attribute involved in an equality join condition is bounded. For LTO queries, Condition C3 is equivalent to the statement that any nonredundant inequality join predicate involves only bounded attributes (see Lemma A.1 in the *electronic appendix*).

For the "if" part of Theorem 5.6, we describe a bounded-memory evaluation strategy for Q, when conditions C1–C3 hold. The evaluation strategy uses the observation above that only bounded attributes influence the output of Q, once the filter conditions are accounted for. The evaluation strategy maintains *synopses* for the n streams. The synopsis for S_i is, conceptually, the bag of tuples formed by projecting on the bounded attributes of S_i all the tuples of S_i seen so far that satisfy the filter conditions of S_i. To keep memory bounded, S_i stores only distinct tuples and maintains the number of times each distinct tuple appears in the bag. The number of distinct tuples in the synopsis is bounded, since each attribute is bounded. When a new stream tuple s_i arrives on stream S_i, the synopsis for S_i is updated; also, the synopses for other streams are used to determine any new output tuples resulting from the join of s_i with the earlier tuples of these streams. To update the synopsis for S_i, tuple s_i is checked against all the filter conditions of S_i that appear in P^+; if s_i satisfies all the filter conditions, its projection on the bounded attributes of S_i is computed and stored in the synopsis. It is straightforward to see that the synopses for n streams, maintained similarly, contain sufficient information to compute new output tuples resulting from the join of s_i with all earlier tuples of other streams, since all the nonredundant join conditions and projections involve only bounded attributes. The following example illustrates our evaluation strategy.

Example 5.7. Consider $Q(P) = \dot{\pi}_A(\sigma_{(A<20)\wedge(A=C)\wedge(C>10)\wedge(B>20)}\ (S \times T))$, an LTO query over streams $S(A, B)$ and $T(C)$. The synopsis for stream S is formed by projecting onto attribute A the bag of tuples of S seen so far that satisfy all the filter conditions of S, that is, the tuples having a value strictly between 10 and 20 on their A attribute and a value greater than 20 on their B attribute. Note that we enforce the filter condition $(A > 10)$ although it occurs only in P^+ and not in P. The synopsis for S is therefore a bag of tuples over A with values between 10 and 20. This synopsis is compactly represented by storing, for each value $v \in [11, 19]$, the number of tuples in the synopsis with $A = v$. Similarly, the synopsis for T contains for each $v \in [11, 19]$, the count of tuples of T seen so far with $C = v$. Consider the arrival of a new T tuple, $t = \langle v \rangle$. If v lies outside the range [11, 19], it does not join with any S tuple, and can be ignored; if v lies within the range, it joins with all earlier S tuples having $A = v$ and $B > 20$: this is just the number of tuples in synopsis for S with $A = v$. Therefore, the new output tuples resulting from the arrival of t can be determined using the synopsis for S. Similarly, the new output tuples resulting from the arrival of a new S tuple can be determined from the synopsis for T.

Now consider the "only-if" part of Theorem 5.6. A formal proof appears in the *electronic appendix*. Here we give intuition and examples. If some condition C1–C3 is not satisfied, some unbounded attribute influences the output of Q, either as a projection attribute or as an attribute in a nonredundant join predicate.

Thus, any evaluation algorithm for Q might be forced, in the worst case, to remember an unbounded number of values of this attribute, thus requiring unbounded memory. Example 5.8 illustrates how the existence of an unbounded attribute in the project list can force any evaluation algorithm to use unbounded space, and Example 5.9 illustrates the same for an unbounded attribute in a nonredundant join predicate.

Example 5.8. Consider the LTO query $Q = \dot{\pi}_A(\sigma_{(A>10)\wedge(B=C)\wedge(B=10)}(S \times T))$ over streams $S(A, B)$ and $T(C)$. The projection attribute A is clearly not bounded. Consider an instant in the evaluation of Q, where input tuples, $\langle v_1, 10\rangle, \ldots, \langle v_N, 10\rangle$, such that each $v_i > 10\,(1 \leq i \leq N)$, have arrived on stream S, and no tuple has arrived on stream T. Any evaluation algorithm for Q has to remember all N values of A, $\{v_1, \ldots, v_N\}$ at this instant, since a future arrival of a $\langle 10\rangle$ tuple on stream T would require the algorithm to produce all the N values in the output. Since N can be made arbitrarily large, any evaluation algorithm requires unbounded memory in the worst-case.

Example 5.9. Consider the LTO query $Q = \dot{\pi}_A(\sigma_{(A=10)\wedge(B<C)\wedge(B>10)\wedge(C>10)}(S \times T))$ over streams $S(A, B)$ and $T(C)$. The projection attribute A of this query is bounded; however, there exists a nonredundant inequality join predicate $(B < C)$ involving unbounded attributes. Therefore, $MinRef(S) = \{B\}$ and $MaxRef(T) = \{C\}$ are both nonempty, violating Condition C3. We assert that Q cannot be computed in bounded memory. For the sake of contradiction, suppose there exists an algorithm A that can evaluate Q with fewer than a constant M units of memory. We will construct a class of input scenarios and show that Algorithm A will fail to produce the correct output of Q for at least one of the input scenarios.

Define two sets of N tuples, $\mathcal{S} = \{\langle 10, 11\rangle, \langle 10, 12\rangle, \ldots, \langle 10, 10 + N\rangle\}$ and $\mathcal{T} = \{\langle 12\rangle, \langle 13\rangle, \ldots, \langle 11 + N\rangle\}$. Consider a class of input scenarios where some subset of tuples chosen from $\mathcal{S}$ arrives on stream S (the order in which the tuples arrive does not matter), followed by one tuple chosen from $\mathcal{T}$ that arrives on stream T. For each input scenario, consider the instant when all the tuples of stream S have arrived, but the single tuple of stream T has not. Algorithm A will be in some state at this instant; since Algorithm A has fewer than M units of memory, the number of distinct states that it can be in is finite. However, since there are 2^N subsets of $\mathcal{S}$, for some sufficiently large N, there exist two distinct subsets of $\mathcal{S}$ such Algorithm A ends up in the same state after seeing the tuples of either subset. Let $\mathcal{S}'$ and $\mathcal{S}''$ be two such subsets, and let $\langle 10, k\rangle$ be the tuple with the smallest value of k that is present in one of $\mathcal{S}'$ or $\mathcal{S}''$ but not in the other. Assume, without loss of generality, that $\langle 10, k\rangle \in \mathcal{S}'$. Now consider two input scenarios from the class above: $\mathcal{S}'$ followed by tuple $\langle k + 1\rangle$ on stream T, and $\mathcal{S}''$ followed by the same tuple $\langle k + 1\rangle$. The output of Q after the arrival of the $\langle k + 1\rangle$ tuple on T differs for the two scenarios; the count of $\langle 10\rangle$ tuples in the output for the former case, constructed from $\mathcal{S}'$, is always one greater than the output for the latter case, constructed from $\mathcal{S}''$. Since Algorithm A is unable to distinguish between $\mathcal{S}'$ and $\mathcal{S}''$, it will give the same answer in both cases and one answer will be incorrect.

5.2 LTO Queries with Duplicate-Eliminating Projection

This section presents a characterization of LTO queries with duplicate-eliminating projection.

THEOREM 5.10. *Let* $Q = \pi_L(\sigma_P(S_1 \times \cdots \times S_n))$ *be an LTO query.* Q *is bounded-memory computable* (*Definition* 3.1) *iff:*

$C1$: *Every attribute in the project list* L *is bounded.*

$C2$: *For every equality join predicate* $(S_i.A = S_j.B)$, $i \neq j$, $S_i.A$ *and* $S_j.B$ *are both bounded.*

$C3$: $|MaxRef(S_i)|_{eq} + |MinRef(S_i)|_{eq} \leq 1$ *for* $i = 1, \ldots, n$.

In condition $C3$, $|E|_{eq}$ *denotes the number of equivalence classes into which the element set* E *is partitioned by the set of predicates* P.

Theorem 5.10 differs from Theorem 5.6 in two respects. First, unlike Condition C3 of Theorem 5.6, Condition C3 of Theorem 5.10 allows each stream to have one non-bounded attribute (or a set of attributes belonging to the same equivalence class) participating in a nonredundant inequality join predicate. Second, Theorem 5.10 is applicable to all duplicate-eliminating LTO queries, including queries with just one stream, while Theorem 5.6 holds only for duplicate-preserving LTO queries involving more than one stream. Duplicate-preserving LTO queries involving just one stream are always bounded-memory computable.

As we did for the duplicate-preserving case, we only discuss the ideas behind Theorem 5.10 in this section; the formal proof for the "if" part of the theorem can be derived from the discussion, while the formal proof for the "only-if" case is presented in the *electronic appendix*.

For the "if" part of Theorem 5.10, we describe a bounded-memory evaluation strategy for Q, when conditions C1–C3 are satisfied. As in the duplicate-preserving case, the evaluation algorithm maintains a synopsis for each stream summarizing the set of tuples seen so far in the stream; in addition, it also remembers the output tuples produced so far, to ensure duplicate elimination. Note that Condition C1 guarantees that the set of possible output tuples is bounded in size.

The synopsis for Stream S_i depends on the values of $MaxRef(S_i)$ and $MinRef(S_i)$. The simplest case occurs when both $MaxRef(S_i)$ and $MinRef(S_i)$ are empty. To maintain the synopsis of S_i for this case, each new input tuple s_i of S_i is checked against all the filter conditions of S_i in P^+; if s_i satisfies all the filter conditions, its projection on the bounded attributes of S_i is inserted into the synopsis. However, duplicate insertions into the synopsis are ignored. In other words, the synopsis for S_i is the set (not bag) of tuples resulting from first projecting on the bounded attributes of S_i all tuples seen so far on S_i that satisfy its filter conditions, and then removing duplicates among the resulting projected tuples. The size of this set is clearly bounded since the projected tuples involve only bounded attributes. It is easy to verify, using the fact that all nonredundant join predicates of S_i involve only bounded attributes, that any output tuple produced by the original bag of tuples of S_i, by joining with tuples

of other streams, is also produced by the projected set of tuples in the synopsis for S_i, by joining with same set of tuples of other streams.

Next, consider the case where $MaxRef(S_i)$ is not empty. From Condition C3, it follows that $MinRef(S_i)$ is empty. Now, some unbounded attribute A of S_i is involved in a nonredundant, inequality predicate $(S_i.A > S_j.B)$, $(i \neq j)$. For this case, the synopsis of S_i contains attribute A in its schema in addition to the usual bounded attributes of S_i, and is maintained as follows. For each input tuple s_i on stream S_i that satisfies all the filter conditions of S_i in P^+, the tuple s_p formed by projecting s_i onto A and the bounded attributes of S_i is computed. If there is no tuple $s_p{}'$ in the current synopsis that agrees with s_p on all the bounded attributes, s_p is inserted into the synopsis; however, if there exists some such tuple $s_p{}'$, the tuple among s_p and $s_p{}'$ having the larger value on attribute A is retained in the synopsis, and the other is discarded. Since the two tuples differ only on their value on attribute A, and this value is used only to check the predicate $(S_i.A > S_j.B)$, any output tuple produced using the discarded tuple can be produced using the tuple retained in the synopsis. Therefore, among all tuples of S_i that agree on all the bounded attributes and satisfy the filter conditions of S_i, the tuple with the maximum value of A is chosen, and its projection is stored in the synopsis for S_i. Again, it is relatively straightforward to verify that at any instant, the set of tuples in the synopsis behave exactly like the entire bag of tuples of S_i, as far as the output of Q is concerned.

Finally, the case where $MinRef(S_i)$ is not empty and $MaxRef(S_i)$ is empty is handled analogously. The following example illustrates the evaluation strategy described above.

Example 5.11. Consider the query $Q = \pi_A(\sigma_{(A=10)\wedge(B<C)\wedge(B>10)\wedge(C>10)}(S \times T))$ over streams $S(A, B)$ and $T(C)$, obtained from the query in Example 5.9 by replacing the duplicate-preserving projection by a duplicate-eliminating one. The duplicate-eliminating version of the query is bounded-memory computable unlike the duplicate-preserving version. The schema of the synopsis for S contains both attribute A, since it is bounded, and attribute B, since it occurs in $MinRef(S)$. As there is only a single possible value of A that satisfies the filter conditions of S, the synopsis for S contains at most one tuple: the tuple, if it exists, with the minimum value of B among all tuples with $A = 10$ and $B > 10$. The synopsis for T also contains at most one tuple: the tuple with the maximum value of C so far with $C > 10$. Any new tuple of T is joined with the synopsis of S, and analogously, a new tuple of S is joined with the synopsis of T. For this query, the output of any successful join is always the tuple $\langle 10 \rangle$. Therefore, the first successful join results in the output $\langle 10 \rangle$, and all subsequent joins are ignored.

Conditions C1–C3 are less restrictive for duplicate-eliminating queries than for duplicate-preserving queries because for the duplicate-eliminating case, when a new stream tuple arrives, it is sufficient to know whether or not the new tuple produces a given output tuple by joining with the earlier tuples of the other streams, while for the duplicate-preserving case, it is necessary to know the number of different ways the new tuple joins with tuples of other streams

to produce the same output tuple. For Query Q of this example, to check if a new S tuple joins with some earlier T tuple producing output tuple $\langle 10 \rangle$, the T tuple t_{max} with the maximum value of attribute C greater than 10 that we store in our synopsis suffices, since if the new S tuple joins with any other T tuple, it joins with t_{max} as well. However, to determine the exact number of T tuples that the new S tuple joins with, we need to remember the entire distribution of C values seen so far on Stream T, as we illustrated in Example 5.9.

For the "only-if" part of Theorem 5.10, the following example illustrates how violation of Condition C3 can force any evaluation strategy for Q to use unbounded memory. A formal proof, covering conditions C1 and C2 as well, is presented in the *electronic appendix*.

Example 5.12. Consider query

$$Q = \pi_A \left(\sigma_{\substack{(A=10) \wedge (B>D) \wedge (C>E) \wedge (B>10) \\ \wedge (C<10) \wedge (D>10) \wedge (E<10)}} (S \times T) \right)$$

over streams $S(A, B, C)$ and $T(D, E)$. This query has two nonredundant inequality join predicates between S and T, which violates Condition C3. Consider an instant in the evaluation of Q where N tuples, $\langle 10, 11, -11 \rangle, \ldots, \langle 10, 10 + N, -(10 + N) \rangle$, have arrived on stream S, and no tuple has arrived on stream T. For each S tuple $\langle 10, c, -c \rangle$, there exists a potential T tuple $\langle c - 1, -c - 1 \rangle$ that joins only with $\langle 10, c, -c \rangle$ and not with any of the other $N - 1$ tuples. Therefore, any evaluation strategy for Q has to remember all the N tuples of S, and, since N can be made arbitrarily large, requires unbounded memory.

5.3 Summary

To summarize this section, we first reduced the problem of characterizing bounded-memory computability of SPJ queries to the problem of characterizing bounded-memory computability of a special class of queries called LTO queries. Then, in Sections 5.1 and 5.2, we presented a characterization of bounded-memory computable duplicate-preserving and duplicate-eliminating LTO queries, respectively. The results of this section can not only be used to determine if an SPJ query Q is computable in bounded memory—by checking using either Theorem 5.6 or Theorem 5.10 if each LTO query derived (Definition 5.2) from Q is computable in bounded memory—but also suggest an evaluation strategy if Q is computable in bounded memory: rewrite Q as a duplicate preserving union of all LTO queries derived from it, evaluate each LTO query independently, and accumulate the output of all LTO queries (Theorem 5.3).

This naive technique for checking bounded-memory computability of SPJ queries and evaluating them is, however, very inefficient. The number of LTO queries derived from an SPJ query is usually exponential in the number of attributes of the SPJ query. In the next two sections, we use the basic ideas from this section to derive more efficient techniques for checking and evaluation that avoid explicitly rewriting the query into LTO queries.

Input: SPJ query $Q = \prod_L(\sigma_P(S_1 \times \cdots \times S_n))$, $\prod \in \{\dot{\pi}, \pi\}$
Output: *Yes*, if Q is computable in bounded memory. *No*, otherwise.

(1) If P is not satisfiable, or if $n = 1$ and $\prod = \dot{\pi}$, return *Yes*.
(2) If some attribute $A \in L$ is unbounded, return *No*.
(3) If there exists a predicate $S_i.A = S_j.B \in P$ $(i \neq j)$, and at least one attribute A or B is unbounded, return *No*.
(4) For each $X \subseteq \mathcal{A}(Q)$ with $|X| \leq 4$, form a query Q' with an empty projection list, IND($X \cup \{k_{max}, k_{min}\}$, P) as the selection predicate, and joining the (at most 4) streams that have at least one of their attributes in X. If any such Q' is not computable in bounded memory (using Theorems 5.10 and 5.6), return *No*. k_{min} and k_{max} denote the minimum and the maximum constant values, respectively, that appear in Q.
(5) Return *Yes*. ◇

Fig. 2. Algorithm to check bounded memory computability of SPJ Queries.

6. CHECKING ALGORITHM

This section presents a simple polynomial algorithm that determines if an SPJ query $Q = \prod_L(\sigma_P(S_1 \times \cdots \times S_n))$ is bounded-memory computable, without explicitly checking each LTO query derived from Q.

The algorithm is shown in Figure 2. It handles both duplicate-preserving and duplicate-eliminating queries. The terms k_{max} and k_{min} in the algorithm denote the maximum and minimum constant values, respectively, appearing in Q.

The algorithm proceeds in five steps. Step (1) checks if Q is trivially computable in bounded memory. Steps (2), (3), and (4), respectively, check if one of the conditions C1, C2, or C3 of Theorem 5.6 (if $\prod = \dot{\pi}$) or Theorem 5.10 (if $\prod = \pi$) is violated by some LTO query derived from Q. If one of the three conditions is violated, then Q is not bounded-memory computable, and a *No* is returned in the corresponding step; otherwise, a *Yes* is returned in Step (5).

We briefly describe how Steps (2)–(4) check the conditions C1–C3 without explicitly enumerating each derived LTO query. Steps (2) and (3) are based on the observation that an unbounded attribute of $Q(P)$ remains unbounded in some LTO query derived from Q, and, conversely, any unbounded attribute of an LTO query derived from Q is unbounded in Q as well. For Step (2), this observation implies that an unbounded attribute belonging to L remains unbounded in some LTO query derived from Q violating Condition C1. Similarly, for Step (3), two unbounded attributes A and B involved in an equality join $(A = B)$ remain unbounded in some LTO query derived from Q, thereby violating Condition C2.

Step (4) of the algorithm uses the fact that just two nonredundant predicates—one in *MaxRef* of a stream and the other in *MinRef*—suffice as a witness to the violation of Condition C3. Two predicates involve at most four attributes, so any violation of Condition C3 can be detected by checking all queries with four attributes or less constructed from Q in an appropriate way. For concreteness, let Q have a duplicate-eliminating projection (the case of

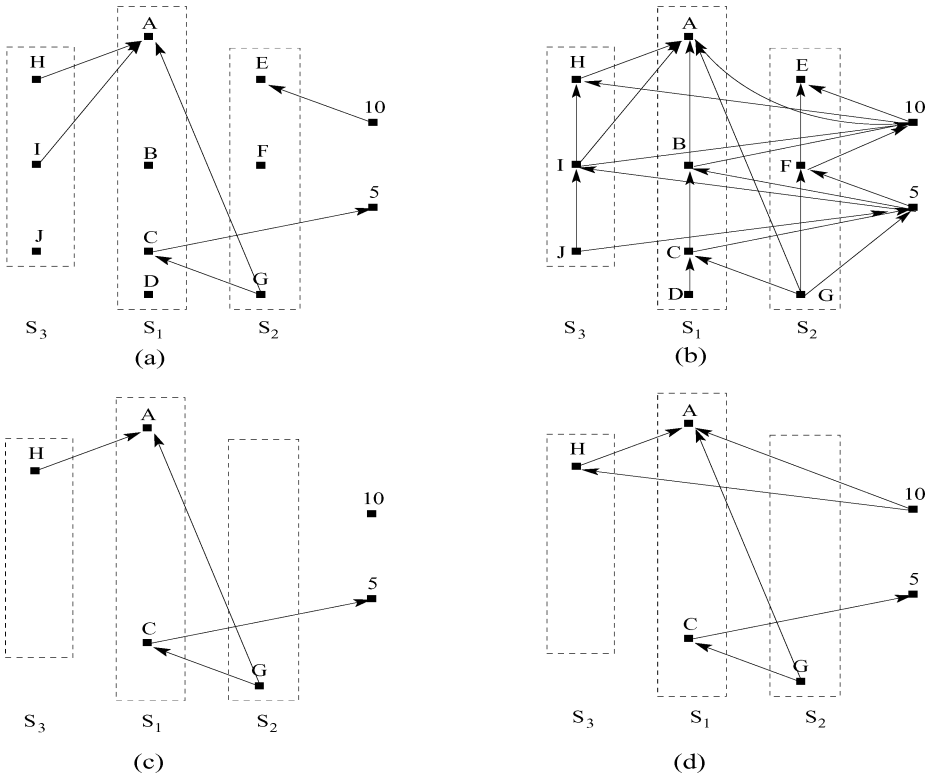

Fig. 3. Example illustrating checking algorithm of Figure 2.

duplicate-preserving projection can be argued similarly). Assume that some LTO query, Q_L, derived from Q violates Condition C3 (of Theorem 5.10). Then, there exist two nonredundant inequality join predicates p_1 and p_2, involving a common stream S_i, that cause $|MaxRef(S_i)|_{eq} + |MinRef(S_i)|_{eq} > 1$. Let X denote the set of (at most 4) attributes participating in p_1 and p_2. The query Q' constructed from X in Step (4) is not computable in bounded memory, and therefore our algorithm correctly returns *No*. Conversely, if some query Q' constructed in Step (4) is not bounded-memory computable due to a violation of Condition C3 by some LTO query Q'_L derived from Q', then Condition C3 is also violated by some LTO query Q_L derived from Q. The following example illustrates Step (4) of our algorithm for an example input query.

Example 6.1. Figure 3 illustrates Step (4) of our algorithm for an example input query. A query is represented in Figure 3 using its selection predicate alone (since the projected attributes are not relevant to Step (4)). An inequality join predicate between two elements is represented by a directed edge between the elements, and attributes belonging to the same stream are indicated using an enclosing rectangle.

Figures 3(a) and 3(b) show an input query Q, with duplicate-eliminating projection, over three streams $S_1(A, B, C, D)$, $S_2(E, F, G)$ and $S_3(H, I, J)$, and an LTO query Q_L derived from Q. Query Q is not bounded-memory computable,

since Q_L violates Condition C3 of Theorem 5.10: $MaxRef(S_1) = \{A, C\}$ (due to $(H < A)$ and $(G < C)$). Figure 3(c) shows the query Q' constructed from $X = \{H, A, C, G\}$, the set of attributes involved in violation of Condition C3 in Q_L, in Step 4 of our algorithm. Query Q' is not bounded-memory computable since an LTO query, Q'_L (Figure 3(d)), derived from Q' violates Condition C3. Observe that Q'_L is related to Q_L the same way Q' is related to Q.

Clearly, Steps (1)–(3) can be executed in polynomial time in the size of the input query. Each query Q' in Step (4) is of $O(1)$ size, so we can check its LTO queries in constant time. Since there are $\Theta(|\mathcal{A}(Q)|^4)$ subsets of $\mathcal{A}(Q)$ having cardinality at most 4, Step (4) takes polynomial time. Thus, our algorithm is polynomial in the size of the input query.

7. QUERY EVALUATION STRATEGY

As described in Section 5.3, a naive evaluation strategy for SPJ queries is to rewrite the query as a union of the LTO queries, evaluate each LTO query independently, and accumulate their output. In this section, we present an evaluation strategy that is more efficient computationally and uses less memory.

Intuitively, the efficient evaluation strategy logically merges all the individual synopses maintained by each LTO query of the naive evaluation strategy into a single synopsis. Since there is substantial redundancy among the different synopses of the naive strategy, the merged synopsis is not much larger than any of the individual synopses, and therefore the efficient evaluation strategy uses significantly less memory than the naive strategy, by a factor roughly equal to the number of LTO queries.

We first introduce some notation used to describe our evaluation strategy. We use uppercase letters to denote streams and the corresponding lowercase letters to denote instances of streams. For example, s_i denotes an instance of Stream S_i. Recall from Section 3.1 that the instance of a stream at any point in time is the bag of tuples seen so far on that stream. For a query Q over streams $S_1, \ldots, S_n$, $Q(s_1, \ldots, s_n)$ denotes the output of Q when the instances s_i $(1 \leq i \leq n)$ of streams have been seen.

We present our evaluation strategy for duplicate-preserving queries and duplicate-eliminating queries separately.

7.1 Evaluation Strategy for Duplicate-Preserving Queries

Consider a bounded-memory computable, duplicate-preserving query $Q = \dot{\pi}_L(\sigma_P\ (S_1 \times \cdots \times S_n))$. We ignore the trivial case of $n = 1$. As for the case of LTO queries, our evaluation strategy for Q maintains a synopsis, $Syn(s_i)$, summarizing the current instance s_i of each stream. The synopsis $Syn(s_i)$ is a bag of tuples having the same schema as S_i, and it has the property that it is equivalent to s_i for the purposes of evaluating Q. Formally, for any s_i $(1 \leq i \leq n)$, $Q(s_1, \ldots, s_n) = Q(Syn(s_1), \ldots, Syn(s_n))$. When a new tuple arrives on a stream, our evaluation strategy joins the new tuple with the synopses for all other streams, and outputs any resulting tuples. The property of the synopses described above guarantees the correctness of this evaluation strategy.

In order to maintain the synopsis for S_i, the tuples of S_i are logically partitioned into a bounded number of *buckets*. Let k_{max} and k_{min} denote the maximum and minimum constant value appearing in Q, respectively. Let $\mathcal{K}$ denote the set of integers between k_{max} and k_{min}, that is, $\mathcal{K} = \{k_{min}, k_{min} + 1, \ldots, k_{max}\}$. Partition the domain of integers into $|\mathcal{K}| + 2$ nonoverlapping ranges: $(-\infty, k_{min} - 1], [k_{min}, k_{min}], \ldots, [k_{max}, k_{max}], [k_{max} + 1, \infty)$; the first and the last ranges are unbounded, and the intermediate ones span exactly one integer. A bucket b of S_i assigns, for each attribute A of S_i, one of the $|\mathcal{K}| + 2$ ranges. Thus, there are $(|\mathcal{K}| + 2)^{|\mathcal{A}(S_i)|}$ distinct buckets of S_i corresponding to all the possible assignments of ranges to attributes. We use the notation $b[A]$ to denote the range identified by bucket b for attribute A, writing $b[A] = -\infty$ if the range is $(-\infty, k_{min} - 1]$, $b[A] = k$ if the range is $[k, k]$, $k \in \mathcal{K}$, and $b[A] = \infty$ if the range is $[k_{max} + 1, \infty)$. A tuple t *belongs* to a bucket if its value for each attribute falls within the range assigned by the bucket. Clearly, each tuple belongs to a single bucket.

Example 7.1. Consider Query $Q_7 = \dot{\pi}_A(\sigma_{(B<D)\wedge(D>10)\wedge(B<20)\wedge(A=10)}(S \times T))$ from Table I. For this query $k_{max} = 20$ and $k_{min} = 10$. Therefore, there are 13^3 different buckets for stream $S(A, B, C)$. An example of a bucket of S is the bucket b with ranges $b[A] = -\infty$, $b[B] = 14$, and $b[C] = \infty$. All tuples of S with a value less than 10 for attribute A, the value 14 for attribute B, and a value greater than 20 for attribute C belong to this bucket.

Any two tuples t_1 and t_2 belonging to the same bucket are equivalent for the purposes of evaluating Q. In other words, whenever t_1 joins with a set of tuples from other streams to produce an output tuple, t_2 also joins with the same set of tuples to produce the same output tuple. This equivalence property holds only for bounded-memory computable, duplicate-preserving queries, and is formally proven in the *electronic appendix*. The following example illustrates this equivalence property.

Example 7.2. Consider Query Q_7 shown in Example 7.1, again. Consider bucket b_1 with ranges $b_1[A] = 10$, $b_1[B] = -\infty$ and $b_1[C] = \infty$. All tuples of bucket b_1 have a value less than 10 on attribute B. Consequently, they all join with any tuple of stream T of the form $\langle d, e\rangle, d > 10$ producing the same output $\langle 10\rangle$, and fail to join with any other T tuple.

This equivalence property is used to maintain the synopsis for S_i. For a bucket b of stream S_i, let $count(b)$ denote the number of tuples of S_i seen so far that belong to b, and let $t_{first}(b)$ denote the tuple belonging to b that first appears on S_i. The synopsis for S_i consists of $count(b)$ copies of $t_{first}(b)$ corresponding to each bucket b of S_i, discounting the tuples that fail to satisfy the filter conditions of S_i. From the equivalence property above, $count(b)$ copies of the tuple $t_{first}(b)$ is equivalent to the bag of $count(b)$ tuples that belong to b; hence, our synopsis for S_i is equivalent to the bag of tuples of S_i seen so far. Physically, the synopsis for S_i is represented by storing, for each bucket b, $t_{first}(b)$ and $count(b)$, which requires bounded memory since the number of buckets is bounded.

7.2 Evaluation Strategy for Duplicate-Eliminating Queries

Consider a bounded-memory computable, duplicate-eliminating query $Q = \pi_L(\sigma_P(S_1 \times \cdots \times S_n))$. As for the case of duplicate-preserving queries, our evaluation strategy for Q summarizes the current stream instances s_i using a bounded memory synopsis $Syn(s_i)$ that is equivalent to s_i for evaluating Q, that is, $Q(s_1, \ldots, s_n) = Q(Syn(s_1), \ldots, Syn(s_n))$.

To maintain the synopsis for S_i, the tuples of S_i are partitioned into buckets exactly as in the case of duplicate-preserving queries. However, unlike the case of duplicate-preserving queries, the property that any two tuples belonging to the same bucket are equivalent with respect to Q is not valid for duplicate-eliminating queries. But a weaker property still holds: for any finite set (bag) of tuples belonging to the same bucket, it is always possible to pick a subset of one or two representative tuples that is equivalent to the entire set for the purpose of evaluating Q. Using this property, our synopsis contains for each bucket b of S_i, at most two tuples representing the bag of tuples of S_i seen so far that belong to b and satisfy all the filter conditions of S_i.

Informally, we pick the tuple with the maximum value in each *MaxRef* attribute (Definition 5.4) and the tuple with the minimum value in each *MinRef* attribute as representative tuples. For each bucket b, associate a set of filter predicates P_b such that only tuples belonging to the bucket satisfy P_b. The set P_b contains one atomic predicate corresponding to each attribute A of S_i: if $b[A] = -\infty$, the atomic predicate corresponding to A is $(A < k_{min})$; if $b[A] = k \in \mathcal{K}$, then the atomic predicate is $(A = k)$; if $b[A] = \infty$, then the atomic predicate is $(A > k_{max})$. The definition of *MaxRef* and *MinRef* for a bucket b differs slightly from Definition 5.4, and takes into account the additional filter predicates P_b that the tuples belonging to b satisfy.

Definition 7.3. Consider a query $Q(P)$ and a stream $S_i \in \mathcal{S}(Q)$. For a bucket b of S_i, $MaxRef(S_i, b)$ is defined as the set of all unbounded attributes A of S_i in $(P \cup P_b)$ that participate in a nonredundant inequality join $(S_j.B < S_i.A)$, $i \neq j$, in $(P \cup P_b)^+$. $MinRef(S_i, b)$ is similarly defined as the set of all unbounded attributes A of S_i that participate in a nonredundant inequality join of the form $(S_i.A < S_j.B)$, $i \neq j$, in $(P \cup P_b)^+$.

The representative tuples for a bucket b of S_i are picked as follows:

(1) If $MaxRef(S_i, b)$ is nonempty, pick the tuple t that has the maximum value of $\min\{t[A] : A \in MaxRef(S_i, b)\}$.
(2) If $MinRef(S_i, b)$ is nonempty, pick the tuple t that has the minimum value of $\max\{t[A] : A \in MinRef(S_i, b)\}$.
(3) If both $MaxRef(S_i, b)$ and $MinRef(S_i, b)$ are empty, pick the first tuple that appears on S_i.

Example 7.4. Consider the query

$$Q(P) = \pi_A \left(\sigma_{\substack{(B>F)\wedge(C>F)\wedge(D>F) \\ \wedge(A=10)\wedge(E=A)}} (S \times T) \right)$$

over streams $S(A, B, C, D)$ and $T(E, F)$. The reader can verify that Q is bounded-memory computable. Consider a bucket with ranges $b[A] = 10$, $b[B] = 10$, $b[C] = -\infty$ and $b[D] = -\infty$. The predicate P_b for this bucket is $\{(A = 10), (B = 10), (C < 10), (D < 10)\}$. Attributes C and D are unbounded for predicate $(P \cup P_b)$; attribute B is bounded for $(P \cup P_b)$ although it is unbounded for predicate P. Both C and D occur in $MaxRef(S_i, b)$ due to nonredundant predicates $(C > F)$ and $(D > F)$. $MinRef(S_i, b)$ is empty.

One representative tuple is sufficient for bucket b: it is the tuple s with the maximum value of $\min\{s[C], s[D]\}$ among all tuples of S_i seen so far that belong to b.

8. EXTENSIONS

In this section, we extend our results to a larger class of queries. Section 8.1 considers queries with self-joins, and Section 8.2 considers queries with grouping and aggregation.

8.1 Queries with Self-Joins

In a self-join query, at least one stream appears more than once in the join list. We use the notation $S^1, S^2, \ldots$ to denote different occurrences of the same stream S in a query. For instance, query $\pi_{S^1.A}(\sigma_{(S^1.A=S^2.A)}(S^1 \times S^2))$ is a join of stream S with itself on attribute A.

The characterization of bounded-memory computable self-join queries differs slightly from that of non-self-join queries due to an implicit constraint on self-join streams: at any point of time, the instances of any two self-join streams, S^j and S^k, are the same. This additional constraint affects our characterization for duplicate-eliminating queries only. For duplicate-preserving queries, our characterization of bounded-memory computability using Theorems 5.3, 5.6, and 5.10 continues to hold. However, the "only-if" proofs of Theorems 5.6 and 5.10 have to be modified slightly, since currently they assume that instances of streams could be arbitrary. Since our characterization remains unchanged, the checking algorithm of Section 6 and the efficient evaluation strategy of Section 7 can be used for duplicate-preserving queries with self-joins as well.

For duplicate-eliminating queries, our reduction of bounded-memory computability of SPJ queries to that of LTO queries still holds (Theorem 5.3); however, our characterization of bounded-memory computable LTO queries (Theorem 5.10) does not, as the following example illustrates.

Example 8.1. Consider the self-join LTO query $Q = \pi_{S^1.A}(\sigma_P(S^1 \times S^2))$, where $P = \{(S^1.A = 10), (S^1.A = S^2.A), (S^1.B = S^2.B), (S^1.B > 10)\}$. The equality join of unbounded attributes $S^1.B$ and $S^2.B$ violates condition C2 of Theorem 5.10, but Q is equivalent to the query $Q' = \pi_A(\sigma_{(A = 10) \wedge (B > 10)}(S))$, which is clearly bounded-memory computable.

Conditions C1–C3 of Theorem 5.10 are still sufficient to ensure that a self-join LTO query is computable in bounded memory, but they are not necessary, that is, there exist queries (e.g., query Q of Example 8.1) that violate one or more of conditions C1–C3, but are computable in bounded memory.

Informally, one of the two occurrences of stream S in query Q of Example 8.1 was "redundant," which allowed Q to be rewritten as Q' using just one occurrence of S. Theorem 5.10 fails only for queries with redundant streams. In other words, a duplicate-eliminating LTO query without any redundant streams is bounded-memory computable iff conditions C1–C3 hold. Our strategy for handling duplicate-eliminating LTO queries is, therefore, to first rewrite these queries without redundant streams and then use Theorem 5.10 to check bounded-memory computability of the rewritten queries. We formalize redundant streams, and show how a query can be rewritten without redundant streams.

Definition 8.2. A stream S_i^j is said to be *redundant* in an SPJ query $Q(P)$ if there exists a stream $S_i^k (j \neq k)$ such that:

(1) If $(S_i^j.A\ Op\ k) \in P^+$, then $(S_i^k.A\ Op\ k) \in P^+$, and if $(S_i^j.A\ Op\ S_i^j.B) \in P^+$, then $(S_i^k.A\ Op\ S_i^k.B) \in P^+$ (i.e., any filter condition of S_i^j is also a filter condition of S_i^k).

(2) If $(S_i^j.A\ Op\ S_l.B) \in P^+$, then $(S_i^k.A\ Op\ S_l.B) \in P^+$.

(3) If $S_i^j.A \in L$, where L is the list of projected attributes, then $(S_i^j.A = S_i^k.A) \in P^+$.

In such a case, we say that S_i^k *covers* S_i^j in Q.

In Example 8.1, S^1 covers S^2, and vice-versa. If S_i^k covers S_i^j in an SPJ query Q, the query Q' obtained from Q by eliminating S_i^j, and replacing all occurrences of $S_i^j.A$ by $S_i^k.A$ is equivalent to Q. By repeatedly using this rewriting step, we can eliminate all redundant streams from a query.

To summarize, a duplicate-eliminating query with self-joins is bounded-memory computable iff all the LTO queries derived from it are bounded-memory computable. In order to determine if a duplicate-eliminating LTO query is bounded-memory computable, we remove all redundant streams from the query and check if the resulting rewritten query is bounded-memory computable using Theorem 5.10. Currently, we do not know how to extend the more efficient checking (Section 6) and evaluation algorithms (Section 7) to handle duplicate-eliminating SPJ queries with self-joins; extending these algorithms is future work.

8.2 Queries with Grouping and Aggregation

We now extend our main results to queries involving aggregation and (optionally) grouping. For queries involving only the standard aggregate functions (`SUM`, `COUNT`, `MIN`, `MAX`, `AVG`, `COUNT-DISTINCT`, and `MEDIAN`), we provide necessary and sufficient conditions for bounded-memory computability, and argue that any query that is not bounded-memory computable requires a worst-case space that is linear in the size of its input (Theorems 8.5 and 8.6). For queries that involve user-defined aggregate functions, we only provide a set of sufficient conditions for bounded-memory computability (Theorems 8.3 and 8.4). Stating necessary conditions for bounded-memory computability that apply to all

aggregate functions is difficult because one can easily design nonstandard aggregate functions that admit special optimization tricks.

We use the relational symbol $\mathcal{G}$ to denote the grouping and aggregation operator. A query involving grouping and aggregation (hereafter simply an "aggregate query") is of the general form ${}_G\mathcal{G}_F(\sigma_P(S_1 \times \cdots \times S_n))$, where G is a (possibly empty) set of grouping attributes and F is a set of aggregate expressions. An aggregate expression is of the form $f(S_i.A)$, where f is an aggregate function like SUM or COUNT. The aggregate query ${}_G\mathcal{G}_F(\sigma_P(S_1 \times \cdots \times S_n))$ is equivalent to the SQL query:

SELECT G, F FROM $S_1, \ldots, S_n$ WHERE P GROUP BY G

The answer to an aggregate query at any point of time is the answer using standard relational algebra semantics over the bag of input stream tuples seen so far. Most aggregate queries are nonmonotonic, meaning we cannot stream the output tuples as we did for an SPJ query. We therefore use a different technique for producing the answer to an aggregate query. At any point of time, we require the evaluation strategy to store the current answer to the aggregate query in its memory. We count the memory used to store the query answer when determining whether an aggregate query can be evaluated in bounded memory. Trivially, this approach implies that any aggregate query with an unbounded result size (i.e., an unbounded number of groups) is not bounded-memory computable.

We use the classification from Gray et al. [1997] that divides aggregate functions into three categories: *distributive*, *algebraic*, and *holistic*. Consider a bag of values X and a single value x drawn from some domain. An aggregate function f over the domain is *distributive* if $f(X \cup \{x\})$ can be computed from $f(X)$ and x. An aggregate function f is *algebraic* if it is not distributive, but there exists a "synopsis function" g such that for all X: (1) $f(X)$ can be computed from $g(X)$; (2) $g(X)$ can be stored in bounded memory; and (3) $g(X \cup \{x\})$ can be computed from $g(X)$ and x. An aggregate function f is *holistic* if it is neither distributive or algebraic. A distributive or algebraic aggregate function can be computed in an online fashion using a bounded amount of memory, while a holistic aggregate function cannot. Among the standard aggregate functions, SUM, COUNT, MIN, and MAX are distributive, AVG is algebraic since it can be computed from a synopsis containing SUM and COUNT, and COUNT-DISTINCT and MEDIAN are holistic.

Further, we classify aggregate functions as *duplicate-sensitive* or *duplicate-insensitive*. Let X be a bag of values, and X' the set of distinct elements in X. An aggregate function f is duplicate-insensitive if $f(X) = f(X')$ for all bags X, and duplicate-sensitive otherwise. Functions MIN, MAX, and COUNT-DISTINCT are duplicate-insensitive, while SUM, COUNT, AVG, and MEDIAN are duplicate-sensitive.

THEOREM 8.3. *A single-stream aggregate query $Q = {}_G\mathcal{G}_F(\sigma_P(S))$ can be computed in bounded memory if:* (1) *every grouping attribute in G is bounded; and* (2) *there is no aggregate expression $f(A) \in F$ such that f is holistic and A is unbounded.*

PROOF. Partition the input tuples that satisfy P into the groups defined by the grouping attributes. There are a bounded number of groups because the

grouping attributes are all bounded. Within each group, the values of distributive and algebraic aggregates can be maintained using bounded memory, by definition. For each attribute A that is aggregated using a holistic aggregate function, maintain counts of the number of times each value of A occurs within each group; by condition (2) attribute A must be bounded, so these counts can be maintained using bounded memory. The counts completely capture the distribution of A within the group, allowing the holistic aggregates to be computed from them. □

Now consider a multistream aggregate query $Q = {}_G\mathcal{G}_F(\sigma_P(S_1 \times \cdots \times S_n))$. Let $\mathcal{A}(F) = \{S_i.A \| f(S_i.A) \in F\}$ denote the set of attributes used in aggregate expressions in F. We define the *characteristic query* Q' for Q as $\Pi_{G \cup \mathcal{A}(F)}(\sigma_P(S_1 \times \cdots \times S_n))$, where $\Pi = \pi$ if all aggregate functions in F are duplicate-insensitive and $\Pi = \dot{\pi}$ otherwise.

THEOREM 8.4. *Consider an aggregate query $Q = {}_G\mathcal{G}_F(\sigma_P(S_1 \times \cdots \times S_n))$. If the characteristic query Q' for Q can be computed in bounded memory, then so can Q.*

PROOF. We provide a bounded-memory evaluation strategy for evaluating Q. Evaluate as a subroutine the characteristic query Q' using the appropriate algorithm from Section 7. As output tuples are generated by the subroutine, partition them into groups defined by the grouping attributes, and within each group, maintain counts of the number of occurrences of each value for each of the attributes being aggregated. If Q' is duplicate-preserving, then the counts are a complete description of the distribution of the aggregated attributes, so they are sufficient to compute the aggregate functions for each group. If Q' is duplicate-eliminating, then the number of times each attribute value occurred is lost, but that does not matter since Q' is only duplicate-eliminating when all aggregates in Q are duplicate-insensitive. The fact that Q' is computable in bounded memory implies that all grouping attributes and all aggregated attributes in Q are bounded, so the total memory required for "post-processing" the output of Q' also is bounded. □

Next we present the necessary and sufficient conditions for determining whether an aggregate query, involving only the standard aggregates, is bounded-memory computable. Again, we consider single- and multistream queries separately.

THEOREM 8.5. *Let $Q = {}_G\mathcal{G}_F(\sigma_P(S))$ be an aggregate query over a single stream, where the aggregate functions in F are drawn from* `SUM`*,* `COUNT`*,* `MIN`*,* `MAX`*,* `AVG`*,* `COUNT-DISTINCT`*, and* `MEDIAN`*. Q is bounded-memory computable iff:* (1) *every grouping attribute in G is bounded; and* (2) *there is no aggregate expression $f(A) \in F$ such that f is holistic* (*i.e.,* `COUNT-DISTINCT` *or* `MEDIAN`) *and A is unbounded.*

PROOF. This theorem states that Conditions (1) and (2) from Theorem 8.3 are necessary as well as sufficient for the standard aggregates. The "if" direction is a special case of Theorem 8.3. The "only if" direction is straightforward. As we observed earlier, every grouping attribute has to be bounded to keep the size

of the output bounded. For Condition (2), it is well known that computing the number of distinct values or the median of a bag requires memory proportional to the number of distinct values in the bag, which implies that all attributes aggregated by COUNT-DISTINCT and MEDIAN must be bounded. □

The *reduced characteristic query* of an aggregate query $Q = {}_G\mathcal{G}_F(\sigma_P(S_1 \times \cdots \times S_n))$ is defined in the same way as the characteristic query for Q defined earlier, except now the attributes in the project list are only the grouping attributes G rather than $G \cup \mathcal{A}(F)$. Formally, the reduced characteristic query Q'_R for Q is $\Pi_G(\sigma_P(S_1 \times \cdots \times S_n))$, where $\Pi = \pi$ if all aggregate functions in F are duplicate-insensitive and $\Pi = \dot{\pi}$ otherwise.

THEOREM 8.6. *Let $Q = {}_G\mathcal{G}_F(\sigma_P(S_1 \times \cdots \times S_n))$ be an aggregate query over multiple streams, involving only the standard aggregates SUM, COUNT, AVG, MAX, MIN, COUNT-DISTINCT, and MEDIAN. Let Q'_R be the reduced characteristic query for Q. Then, Q is bounded-memory computable iff:*

*C*1: *Q'_R is computable in bounded memory.*

*C*2: *For every aggregate expression COUNT-DISTINCT$(S_i.A)$ or MEDIAN$(S_i.A)$ in F, $S_i.A$ is bounded.*

*C*3: *For every aggregate expression MAX$(S_i.A)$, if $S_i.A$ is unbounded, either $MaxRef(S_i)$ is empty or $|MaxRef(S_i)|_{eq} = 1$ and $S_i.A \in MaxRef(S_i)$; similarly, for every aggregate expression MIN$(S_i.A)$, if $S_i.A$ is unbounded, either $MinRef(S_i)$ is empty or $|MinRef(S_i)|_{eq} = 1$ and $S_i.A \in MinRef(S_i)$.*

We discuss the ideas behind Theorem 8.6. A formal proof can be derived from the discussion here.

Without loss of generality, we can assume that Q consists of just one aggregate expression. An aggregate query having $n > 1$ aggregate expressions can be equivalently rewritten as a natural join (on the grouping attributes) of the output of n aggregate queries, each having one aggregate expression of the original query, but otherwise identical. For example, the query ${}_A\mathcal{G}_{SUM(B),MAX(C)}(\sigma_{A=10}(S(A, B, C)))$ is equivalent to ${}_A\mathcal{G}_{SUM(B)}(\sigma_{A=10}(S)) \bowtie_{AA} \mathcal{G}_{MAX(C)}(\sigma_{A=10}(S))$. Since the set of possible groups in a bounded-memory computable aggregate query is bounded, a query with more than one aggregate expression is bounded-memory computable iff all the queries with one aggregate expression "derived" from it are bounded-memory computable. Therefore, for the rest of this discussion we assume that F contains only one expression of the form $f(S_i.A)$.

We break the discussion of Theorem 8.6 into three cases depending on the type of the aggregate function used in the single aggregate expression—holistic (COUNT-DISTINCT and MEDIAN), nonholistic and duplicate-sensitive (SUM, COUNT, AVG), nonholistic and duplicate-insensitive (MAX, MIN). For each of these cases, we discuss both the "if" and "only-if" arguments of the proof.

For the holistic aggregates, COUNT-DISTINCT and MEDIAN, the relevant conditions C1 and C2 of Theorem 8.6 reduce to the sufficiency conditions of Theorem 8.4. Therefore, the "if" part of the proof for holistic aggregates is a special case of the proof of Theorem 8.4. For the "only-if" part of the

proof, first assume that Q'_R is not bounded-memory computable violating Condition C1. The non-bounded-memory computability of Q'_R results either from some unbounded attribute in its project list G (which causes the violation of Condition C1 of Theorem 5.6 for some LTO query), or from some join predicates in P (which cause the violation of conditions C2 or C3 of Theorem 5.6). In the former case, Q is clearly not bounded-memory computable, since one of its grouping attributes is not bounded; in the latter case, informally, since even checking all the join predicates of P requires unbounded memory, evaluating an aggregation on top of the join is not feasible in bounded memory either. Finally, by the definition of holistic aggregates, the aggregated attribute $S_i.A$ has to be bounded for bounded-memory computability of Q. Therefore, Condition C2 is necessary as well.

Next consider the case of duplicate-sensitive aggregates `SUM`, `COUNT`, and `AVG`. For these aggregates, only Condition C1 is relevant, that is, Q is bounded-memory computable iff Q'_R is bounded-memory computable. First, consider the "if" part of the proof. Note that Condition C1 relaxes the sufficiency conditions of Theorem 8.4: for these aggregates, it is possible to evaluate the query even if the aggregated attribute is unbounded. A query involving these aggregates need not be evaluated, as suggested by the proof of Theorem 8.4, by first computing the join of the streams, projecting the grouping and the aggregated attributes from the result of the join, and then computing the aggregate on the projections. Instead, we can partially "push" the aggregation below the join, which makes it possible to compute these aggregates even for unbounded aggregated attributes. The following examples illustrates this evaluation strategy.

Example 8.7. Consider the aggregate query $Q = {}_{A,C}\mathcal{G}_{SUM(B)}(\sigma_P(S \times T))$ over two streams $S(A, B)$ and $T(C)$, where $P = \{(A < 20), (C < 20), (A > 10), (C > 10)\}$. Although the aggregation is over an unbounded attribute B, we assert that Q is computable in bounded-memory. As always, our evaluation strategy maintains a synopsis for S and T; in addition, it also maintains the current answer to the query as required by the semantics of aggregate queries. The synopsis for S contains, for each value v in the interval [11, 19], the sum of attribute B of all S tuples seen so far with $A = v$. The synopsis for T contains, for each value v in [11, 19], the count of T tuples seen so far with $C = v$. The arrival of a new T tuple $\langle c \rangle$ $(10 < c < 20)$ changes the output as follows: for each value $v \in [11, 19]$ add the sum corresponding to v in the current synopsis for S to each group $A = v, C = c$ in the output. Similarly, the arrival of a new S tuple $\langle a, b \rangle$ $(10 < a < 20)$ changes the output as follows: for each $v \in [11, 19]$, let n_v denote the current count of tuples in the synopsis for T with $C = v$; add a value $b * n_v$ to each group $A = a, C = v$ in the output. The reader can verify that this evaluation strategy correctly computes the output of Q.

If the aggregated attribute $S_i.A$ is bounded, we can use the evaluation strategy of Theorem 8.4. If it is not, the evaluation strategy of Section 7.1 can be modified as follows. As before, maintain a synopsis for each stream. The synopses for all streams other than S_i (recall that $S_i.A$ is the aggregated attribute) remain unchanged. In order to maintain the synopsis of S_i the tuples of S_i are partitioned into buckets exactly as described in Section 7.1. For each bucket,

in addition to remembering one representative tuple and the count of tuples, remember the aggregate over attribute A. For example, if the aggregate function is SUM, remember the sum of attribute A over all tuples belonging to each bucket. The reader can verify that these synopses are sufficient to answer Q using a technique similar to the one shown in Example 8.7. For the "only-if" proof, if Q'_R is not bounded-memory computable, we can argue that Q is not bounded-memory computable as well, exactly as we did for holistic aggregates.

Finally, consider the case of MIN and MAX. We only discuss MAX since the discussion for MIN is analogous. Again consider the "if" part. If the aggregated attribute $S_i.A$ is bounded, then Q satisfies the conditions of Theorem 8.4, and, therefore, we can use the evaluation strategy presented in the proof of Theorem 8.4. If $S_i.A$ is not bounded, then the bounded-memory computability of Q depends on $MaxRef(S_i)$. If $MaxRef(S_i)$ is empty, we can partially push the MAX aggregate below the join exactly as we did for the duplicate-sensitive queries above: for each bucket in the synopsis for S_i, maintain the maximum value of attribute A over all the tuples that belong to the bucket. If $MaxRef(S_i)$ is nonempty, Condition C3 states that $S_i.A$ must be the only attribute in $MaxRef(S_i)$ (ignoring attributes belonging to the same equivalence class) for Q to be bounded memory computable. Informally, the maximum value of the attributes in $MaxRef$ is needed (in some buckets) for the evaluation of the predicate P. If A is the only attribute in $MaxRef(S_i)$ (ignoring attributes belonging the same equivalence class as A), then there is no conflict, and the maximum value of A can be stored for each bucket exactly as before; this maximum value is now used both for checking P and for computing MAX(A). For the "only-if" part, if Q'_R is not bounded-memory computable, we can argue that Q is not bounded-memory computable exactly as we did for the case of holistic aggregates. If A is not the only attribute in $MaxRef(S_i)$, the proof is similar to the only-if proof of Theorem 5.10 for the case $|MaxRef(S_i)|_{eq} > 1$.

9. FURTHER DISCUSSION

Our results have made the assumption that all attributes have discrete, ordered domains. We can relax this assumption as follows. Define $sat(A, P)$ for an attribute A and a set of predicates P as the set of all possible values that can be assigned to A that make P true for some assignment of values to the rest of the attributes in P. Boundedness of an attribute A (Definition 4.5) can now be generalized: A is bounded by the set of predicates P iff $|sat(A, P)|$ is a constant. This definition of boundedness extends Theorems 5.6, 5.10, and 8.6 to attributes with arbitrary domains. (We assume that an atomic predicate of the form $A > B$ or $A < B$ is used only if the domain of attributes A and B is ordered.) In addition to allowing attributes from arbitrary domains, it would be useful to handle a richer set of predicates (e.g., atomic predicates using domain-specific operators, disjunctions of atomic predicates, etc.). Expanding the class of predicates is an important avenue of future work.

Our results in Sections 5.1, 5.2, and 8 assume that the data inputs to a query consist solely of continuous data streams. In the case of queries over streams, the query evaluation algorithm has no control over the instances and

interleavings of the input streams. For queries over relations stored in conventional databases, the instances of the relations are finite and may be partially known to the query processor (e.g., in the form of statistics on the attributes). Also, in a conventional database system, the query evaluation algorithm usually has some control over the ordering (interleaving) of the relations. Nevertheless, there are cases in "traditional" settings where it is desirable to perform query processing using only one pass over each relation. In such cases, our results can be used to generate evaluation plans that use a constant amount of additional memory regardless of relation sizes.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library. The appendix contains the proofs of many theorems from the main body of the article.

ACKNOWLEDGMENTS

Thanks to Rajeev Motwani, Jeff Ullman, and the entire STREAM group at Stanford for many useful discussions.

REFERENCES

ALON, N., MATIAS, Y., AND SZEGEDY, M. 1999. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci. 58*, 1 (Feb.), 137–147.

ARASU, A., BABCOCK, B., BABU, S., MCALISTER, J., AND WIDOM, J. 2002a. Characterizing memory requirements for queries over continuous data streams. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, New York, 221–232.

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

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, New York, 1–16.

BABCOCK, B., DATAR, M., AND MOTWANI, R. 2002. Sampling from a moving window over streaming data. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM, New York, 633–634.

BABU, S. AND WIDOM, J. 2002. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. Tech. Rep. http://dbpubs.stanford.edu/pub/2002-52, Stanford University. Nov.

CARNEY, D., CENTINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., SEIDMAN, G., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. B. 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, Calif., 215–226.

CHANDRASEKHARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S., RAMAN, V., RESIS, F., AND SHAH, M. A. 2003. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the 1st Conference on Innovative Data Systems Research*. 269–280.

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, Calif., 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, New York, 379–390.

CHOMICKI, J. 1995. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Datab. Syst. 20*, 2, 149–186.

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, New York, 647–651.

DATAR, M., GIONIS, A., INDYK, P., AND MOTWANI, R. 2002. Maintaining stream statistics over sliding windows. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM, New York, 635–644.

DOBRA, A., GAROFALAKIS, M. N., 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, New York, 61–72.

FEIGENBAUM, J., KANNAN, S., STRAUSS, M., AND VISWANATHAN, M. 1999. An approximate l1-difference algorithm for massive data streams. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., 501–511.

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

GEHRKE, J., KORN, F., AND SRIVASTAVA, D. 2001. On computing correlated aggregates over continual data streams. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 13–24.

GILBERT, A. C., GUHA, S., INDYK, P., MUTHUKRISHNAN, S., AND STRAUSS, M. 2002. Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*. ACM, New York, 389–398.

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

GRAY, J., CHAUDHURI, S., BOSWORTH, A., LAYMAN, A., REICHART, A., AND VENKATRAO, M. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Disc. 1*, 1 (Mar.), 29–53.

GREENWALD, M. AND KHANNA, S. 2001. Space-efficient online computation of quantile summaries. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 58–66.

GUHA, S., KOUDAS, N., AND SHIM, K. 2001. Data-streams and histograms. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*. ACM, New York, 471–475.

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, New York, 49–60.

MANKU, G. S., RAJAGOPALAN, S., AND LINDSAY, B. G. 1999. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 251–262.

SHAH, M., HELLERSTEIN, J. M., CHANDRASEKHARAN, S., AND FRANKLIN, M. J. 2003. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings of the 19th International Conference on Data Engineering*. IEEE Computer Society Press, Los Alamitos, Calif.

STREAM. 2003. Stanford stream data management project. http://www-db.stanford.edu/stream.

TERRY, D. B., GOLDBERG, D., NICHOLS, D., AND OKI, B. M. 1992. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. ACM, New York, 321–330.

TRADERBOT. 2003. Traderbot home page. http://www.traderbot.com.

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.

ULLMAN, J. D. 1988. *Principles of Database and Knowledge-Base Systems*. Vol. I. Computer Science Press, Rockville, Md., Chapter 3, 121–122.

ULLMAN, J. D. 1989. *Principles of Database and Knowledge-Base Systems*. Vol. II. Computer Science Press, Rockville, Md., Chapter 14, 892–893.

VITTER, J. 1985. Random sampling with a reservoir. *ACM Trans. Math. Softw. 11*, 1 (Mar.), 37–57.

Received December 2002; revised May 2003; accepted September 2003