Characterizing Memory Requirements for Queries over Continuous Data Streams*

Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister, Jennifer Widom

Stanford University
{arvinda,babcock,shivnath,jonmac,widom}@cs.stanford.edu

Abstract

We consider conjunctive queries with arithmetic comparisons over multiple continuous data streams. We specify an algorithm for determining whether or not a 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, our algorithm produces an evaluation plan based on constant-sized synopses of the data streams.

1 Introduction

In many recent applications, data takes the form of continuous *data streams*, rather than finite stored data sets [BW01]. Examples 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. These applications have a need for queries over data streams that go well beyond simple element-at-a-time processing, they have multiple rapid and time-varying streams, and they require timely online answers. In network traffic management, for example, queries over streams of network packets and/or performance measurements can be used to monitor network behavior online in order to detect anomalies (e.g., link congestion) and their cause (e.g., hardware failure, intrusion, denial-of-service attack) [DG00]. In financial applications, queries over stock tick data, news feeds, and historical company data can be used to monitor trends and detect fleeting opportunities [Tra].

In the *STREAM* (*STanford stREam datA Management*) project [STR], we are developing a *Data Stream Management System* (*DSMS*) that allows some or all of the data being managed to come in the form of continuous, possibly very rapid and time-varying, data streams [BW01]. Developing the query processing component of a DSMS involves many novel and challenging problems, since queries tend to be *continuous* (long-running) rather than *one-time*, stream data distributions and arrival characteristics may be unpredictable, and system conditions (e.g., available memory) will fluctuate over time. As one initial step, our work in this paper addresses the problem of characterizing memory requirements for queries over continuous streams of relational tuples. We motivate the problem through a series of examples.

1.1 Examples

Our set of example queries is shown in Table 1. We use standard relational algebra, introducing $\dot{\pi}$ for duplicate-preserving projection. Two unbounded relational data streams, $S(A, B, C)$ and $T(D, E)$, are used in the example queries. The answer to a query at any point in time is the answer using standard algebra

*This work was supported by the National Science Foundation under grants IIS-9817799 and IIS-0118173 and by an Okawa Foundation Research Grant.

		Computable with bounded memory?	
		For $\dot{\pi}$	For π
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	$\pi_A\,(\sigma_{B<D\,\wedge\,A>10\,\wedge\,A<20}\,(S\times T))$	No	Yes
Q_5	$\pi_A\,(\sigma_{B<D\,\wedge\,B<120\,\wedge\,D>20\,\wedge\,A>10\,\wedge\,A<20}\,(S\times T))$	Yes	Yes
Q_6	$\pi_A\,(\sigma_{B>D\,\wedge\,B>E\,\wedge\,A=10}\,(S\times T))$	No	Yes
Q_7	$\pi_A\,(\sigma_{A<D\,\wedge\,B<E\,\wedge\,A>10\,\wedge\,A<20}\,(S\times T))$	No	Yes
Q_8	$\pi_A\,(\sigma_{B<D\,\wedge\,C<E\,\wedge\,A>10\,\wedge\,A<20}\,(S\times T))$	No	No
Q_9	$\pi_A\,(\sigma_{B<D\,\wedge\,C<E\,\wedge\,A>10\,\wedge\,A<20\,\wedge\,B<E\,\wedge\,C<100\,\wedge\,D>50}\,(S\times T))$	No	Yes

Table 1: Example queries over data streams $S(A, B, C)$ and $T(D, E)$.

semantics over the bag of data stream tuples seen so far. The examples are crafted to each illustrate specific points.

Consider query Q_1, a selection and projection over one data stream: $\pi_A(\sigma_{A>10}(S))$. When the projection is duplicate-preserving, 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.[1] If the projection in Q_1 is duplicate-eliminating, 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.

On first inspection, most queries over data streams, particularly join queries, seem to require unbounded memory. For example, consider query Q_2, an equijoin of streams S and T: $\pi_A(\sigma_{A=D}(S\times T))$. Q_2 requires each tuple t in S (respectively T) to be saved, since t could potentially join with tuples in T (respectively S) that arrive in the future. However, as soon as we consider attributes with discrete ordered domains and queries with inequalities, many more queries over data streams become computable with bounded memory. For the remaining example queries, suppose all attributes are of type integer. Like query Q_2, query Q_3 is an equijoin of streams S and T, but Q_3 adds selection predicates on A and D: $\pi_A(\sigma_{A=D\,\wedge\,A>10\,\wedge\,D<20}(S\times T))$. Q_3 can be evaluated with bounded memory by maintaining, for each integer v in the interval $[11, 19]$, the current number of occurrences of tuples with $A = v$ in S and $D = v$ in T. (We assume that counts can be saved in bounded memory.) Query Q_4 is an inequality join of S and T: $\pi_A(\sigma_{B<D\,\wedge\,A>10\,\wedge\,A<20}(S\times T))$. Q_4 can be evaluated with bounded memory for duplicate-eliminating projection by maintaining, for each integer v in the interval $[11, 19]$, the current minimum value of B among all tuples in S with $A = v$, and maintaining the current maximum value of D among all tuples in T. With this information, each time we see a new S or T tuple, we can decide whether to generate a new A value in the answer. Q_4 cannot be evaluated with bounded memory for duplicate-preserving projection because, to preserve duplicates correctly in the answer, we must save all (A,B) combinations in S where the value of A lies in the interval $[11, 19]$.

Our goal was to develop an algorithm that accurately determines whether or not a given query over data streams can be evaluated with bounded memory over all possible instances of the streams. As the more complex example queries Q_5–Q_9 in Table 1 indicate, this problem turned out to be nontrivial. We will revisit some of these example queries in later sections.

[1]Note two assumptions. First, data stream tuples can be processed at least as fast as the rate at which they arrive. Second, we are not concerned with storage of the query answer. For the monotonic queries we consider, answers can also be data streams.

1.2 Contributions

We make the following contributions in this paper:

- We consider conjunctive queries with arithmetic comparisons over multiple data streams, and we specify an algorithm that determines whether or not a 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, our algorithm produces an evaluation plan 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 evaluation plan, our algorithm identifies specific instances of input streams for which the plan requires memory at least linear in the length of the input streams.

1.3 Related Work

Past and ongoing work on processing *continuous queries* [CDTW00, NACP01, SPAM91, TGNO92], and on querying remote data sets across networks [HF+00, IFF+99, UFA98], strongly relates to the problem of processing queries over data streams, which forms the context for our work. To the best of our knowledge, no past work in this area has considered the problem of characterizing memory requirements for queries statically, leaving memory management entirely to the query execution phase. Memory management strategies used during query execution in these environments include the use of disk to buffer data for memory overflows [IFF+99, UFA98] and grouping queries with common subexpressions to minimize memory usage [CDTW00].

Clearly there is a relationship between queries over data streams and the well-known area of *materialized views* [GM95], since materialized views are effectively queries that need to be reevaluated or updated incrementally whenever the base data changes. Particularly relevant is the *Chronicle data model* [JMS95], which defined a restricted view definition language and algebra that operates over append-only ordered sequences of tuples (*chronicles*). The view definition restrictions, along with restrictions on the sequence order within and across chronicles, guarantees that the views can be maintained with bounded memory. Work on *self-maintenance* [BCL89, GJM96, QGMW96] and *data expiration* [GMLY98] considered the related but distinct problem of identifying the minimum amount of base and/or auxiliary data required for maintaining a materialized view.

Query processing over data streams using *approximate* synopses (summaries) of the streams has been the subject of some recent work. Reference [GKS01a] develops histogram-based techniques to provide approximate answers for *correlated aggregate queries* over data streams. Reference [GKMS01] presents a wavelet-based approach for building small-space summaries over data streams to provide approximate answers for many classes of aggregate queries. Our work differs from work in this area in that we are considering memory requirements for producing exact, not approximate, answers.

In more theoretical work on data streams, [HRR98] studies basic tradeoffs in processing finite data streams, specifically among storage requirements, number of passes required, and result approximations. References [AMS96, FKSV99, Ind00] consider the problem of approximating frequency moments and computing L_p differences over data streams. Reference [GKS01b] considers the problem of maintaining optimal time-based histograms over data streams, [DGIM02] considers maintaining statistics for *sliding windows* over data streams, and [BDM02] considers sampling for the same scenario.

2 Query Language, Execution Model, and Problem Statement

We formally define our model for data streams, our query language, 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.

2.1 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, i.e., a known finite set of attributes. For most of this paper we assume that the domain of each attribute is discrete and totally ordered (e.g., the domain of integers). In Section 6 we extend our results to attributes with more general domains. We assume streams are *bags*, i.e., the same tuple can appear any number of times in a stream.

We assume that streams are generated by an independent source, meaning that a query evaluation plan has no control over the streams. This assumption has two important implications. First, a stream can be read only once from its source, and 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. The *instance* of a stream at any point in time is the bag of tuples of the stream seen until that point. We use the term *presentation* to denote the exact interleaved sequence in which tuples are generated in the input data streams.

2.2 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, we consider *continuous queries*, where the answer to a query over data streams at any point in time is a function of the input streams seen so far. The semantics of queries over continuous relational data streams is therefore a simple extension of the semantics for the traditional relational case.

In this paper, we consider a class of Select-Project-Join (SPJ) queries, which also could be termed as conjunctive queries with arithmetic comparisons. In addition to the conventional duplicate-eliminating semantics of SPJ queries, we also consider duplicate-preserving semantics as in SQL. We use the standard 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. Thus, a duplicate-preserving SPJ query is of the general form $\dot{\pi}_L(\sigma_P(S_1 \times S_2 \times \ldots \times S_n))$, 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 $S_i.A\ Op\ S_j.B$ ($i = j$ or $i \neq j$) or of the form $S_i.A\ Op\ k$, where Op is one of the comparison operators in $\{<, =, >\}$ and k is some constant. Our results can be extended in a straightforward way to include the operators $\{\leq, \geq\}$. We assume that there are no self-joins in the query, i.e., $S_i \neq S_j$ for $i \neq j$. Appendix C explains how to extend our results to include self-joins.

Note that the SPJ queries we consider are *monotonic* [Ull89]. Monotonicity implies that any tuple that appears in the answer at any point continues to do so forever. The query answer can, therefore, also be treated as a data stream, although we do not use this closure property explicitly.

2.3 Execution Model and Problem Specification

We assume that the query evaluation environment has access to some local memory which 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.[2] We are not concerned with memory for storage of the query answer since the answer also can be a data stream. (It turns out that in many, but not all, cases, when a query can be evaluated in bounded memory its answer can be stored in bounded memory.) The goal of this paper is to characterize the worst case memory requirements of SPJ queries over all possible input stream instances and their presentation (Section 2.1). For many SPJ queries, the worst case memory requirement is linear in the length of the input streams and therefore unbounded. However, we identify an interesting class of queries that can always be evaluated with a bounded amount of memory.

Definition 2.1 (**Bounded-Memory Computability of a Query**) An SPJ 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 presentations of the input streams of the query. □

We focus primarily on the problem of identifying exactly the above class of queries. It follows from the proof in Appendix A that any query that does not fall into this class requires memory linear in the length of the input streams to evaluate.

3 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 2, we consider two kinds of SPJ queries, both of the form $\pi_L(\sigma_P(S_1 \times S_2 \times \cdots \times S_n))$ but using either duplicate-eliminating (π) or duplicate-preserving ($\dot{\pi}$) 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. For convenience, we represent the selection predicate as a set instead of a conjunction of atomic predicates. The term *element* is used to refer to either a constant or an attribute of a stream.

For a given SPJ query Q:

$\mathcal{C}(Q)$ denotes the set of constants that appear in Q.

$\mathcal{S}(Q)$ denotes the set of streams that appear in Q.

$\mathcal{A}(Q)$ denotes all the attributes of all the streams in Q, i.e., $\mathcal{A}(Q) = \bigcup_{S \in \mathcal{S}(Q)} \mathcal{A}(S)$, where $\mathcal{A}(S)$ denotes the set of attributes in stream S.

$\mathcal{E}(Q) = \mathcal{A}(Q) \cup \mathcal{C}(Q)$ is the set of all elements relevant to Q.

$\mathcal{E}(S) = \mathcal{A}(S) \cup \mathcal{C}(Q)$ is the set of all elements in Q potentially relevant to stream S.

A set of atomic predicates P is *satisfiable* if there exists some assignment of values to the attributes in P that makes every predicate in the set P evaluate to true. 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 paper we assume that the selection predicates of the queries considered are satisfiable unless mentioned otherwise.

Let P be a set of predicates. The *transitive closure* of P, denoted P^+, is the set of atomic predicates logically implied by the predicates in P. Note that any query $Q(P)$ can be rewritten as an equivalent query $Q(P')$ if $P^+ = (P')^+$. For a given set of predicates P, the set of 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.

[2] 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.

Definition 3.1 (**Redundant Predicate**) An inequality predicate $(e_1 < e_2) \in P$ is said to be *redundant* in P if there exists an element e such that $(e_1 < e) \in P^+$ and $(e < e_2) \in P^+$. Note that removing all the redundant predicates from any P leaves its transitive closure unchanged. □

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 $e_1 < e_2$ or $e_1 = e_2$ or $e_1 > e_2$ is in P^+. Consider a set of predicates P involving only elements in a set E. P is *order-inducing* if E is totally ordered by P. There are exponentially many different order-inducing sets of predicates for a given set of elements. Define $\mathrm{TO}(E)$ as the set of all order-inducing sets of predicates for E. For example, consider the set of elements $E = \{A, B, 5\}$. Two of the order-inducing sets of predicates for E are $\{A < B, 5 < A\}$ and $\{A = B, B < 5\}$. Note that if a set of elements E is totally ordered by a set of predicates P, then $\mathrm{IND}(P, E) \in \mathrm{TO}(E)$.

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 iff $e_1 = e_2 \in P^+$.

Definition 3.2 (**Boundedness of Attributes**) One of the most important properties we use is that of *boundedness* of attributes. An attribute A is *lower-bounded* by a given set of predicates P if there exists 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 $A < k \in P^+$. An attribute is *bounded* if it is both upper-bounded and lower-bounded. An attribute is *unbounded* if it is not bounded. □

Finally, a *filter* is an atomic predicate whose operands are either an attribute and a constant or two attributes of the same stream, i.e., $S_i.A\ Op\ k$ or $S_i.A\ Op\ S_i.B$. Filters form an important class of atomic predicates since they can be evaluated using no extra memory. An atomic predicate that is not a filter requires "joining" two streams and thus potentially requires unbounded memory.

4 Queries with Duplicate-Preserving Projection

In this section we consider SPJ queries with a duplicate-preserving projection operator. Duplicate-eliminating queries are covered in the next section. To determine whether a query Q can be evaluated using a bounded amount of memory, we first rewrite 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. LTO queries have a special structure that makes it easier to determine the maximum amount of memory required to evaluate them, and every SPJ query can be decomposed into a union of LTO queries. 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.

Definition 4.1 (**Locally Totally Ordered (LTO)**) A query $Q(P)$ is *Locally Totally Ordered* if for every $S \in \mathcal{S}(Q)$, $\mathcal{E}(S)$ is totally ordered by P. □

Theorem 4.1 Let $Q = \dot{\pi}_L(\sigma_P(S_1 \times S_2 \times \cdots \times S_n))$. Q can be rewritten as $Q_1 \cup Q_2 \cup \cdots \cup Q_m$, where each Q_i is an LTO query and the unions are duplicate-preserving.

Proof: For each S_i, let $\mathrm{TO}(\mathcal{E}(S_i)) = \{T_i^1, T_i^2, \ldots, T_i^{m_i}\}$. That is, each T_i^j is a *local total ordering*—one possible total ordering of the attributes of S_i together with the query constants. We consider an exhaustive union of $m_1 \times m_2 \times \cdots \times m_n$ queries that combines local total orderings of all streams in the query in all possible ways:

$$
\begin{array}{c}
Q(P \cup T_1^1 \cup T_2^1 \cup \cdots \cup T_n^1) \cup Q(P \cup T_1^2 \cup T_2^1 \cup \cdots \cup T_n^1) \cup \cdots \cup \\
Q(P \cup T_1^1 \cup T_2^2 \cup \cdots \cup T_n^1) \cup Q(P \cup T_1^2 \cup T_2^2 \cup \cdots \cup T_n^1) \cup \cdots \cup \\
\cdots \\
\cdots \cup Q(P \cup T_1^{m_1} \cup T_2^{m_2} \cup \cdots \cup T_n^{m_n})
\end{array} \tag{1}
$$

Each branch of the union is an LTO query. Furthermore, it can be shown that there is a one-to-one correspondence between the tuples in the answer of Q and the tuples in the answer of the union of LTO queries in Query (1). □

Definition 4.2 (**MaxRef and MinRef**) Consider a query $Q(P)$ and a stream $S_i \in \mathcal{S}(Q)$. $MaxRef(S_i)$ is the set of all unbounded attributes A of S_i (Definition 3.2) that participate in an inequality join of the form $S_j.B < S_i.A$, $i \neq j$, where $S_j.B < S_i.A$ is not redundant (Definition 3.1). $MinRef(S_i)$ is similarly defined as the set of all unbounded attributes A of S_i that participate in an inequality join of the form $S_i.A < S_j.B$, $i \neq j$, that is not redundant. □

Theorem 4.2 Let $Q = \dot{\pi}_L(\sigma_P(S_1 \times S_2 \times \cdots \times S_n))$ be an LTO query where P is satisfiable and $n > 1$. Q is bounded memory computable (Definition 2.1) 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 $i = 1, \ldots, n$.

Proof: First consider the "if" portion of the proof: If conditions C1, C2, and C3 are satisfied then Q can be evaluated with bounded memory. We create *synopses* of the n data streams as follows. For each stream S_i, partition the tuples of S_i that satisfy the total order condition on S_i into distinct buckets based on the values of the bounded attributes. Tuples that agree on the values of all bounded attributes are placed in the same bucket; tuples that differ on at least one bounded attribute are placed in different buckets. For each bucket, store the values for the bounded attributes (which are common across all tuples in the bucket) and a count of the total number of tuples falling into that bucket. By the definition of bounded attributes and our discrete domain assumption, the total size of these synopses is bounded by a constant.

These synopses are sufficient to evaluate Q for all input streams and presentations. Attributes that do not occur in the project list or join conditions can be ignored. (All local selection conditions must be implied by the total order, since P is satisfiable, so any tuple satisfying the total order necessarily satisfies all filters that are part of the selection predicate.) Conditions C1 and C2 guarantee that all attributes involved in the projection or equijoins are bounded, and each synopsis maintains full information about the values of all bounded attributes for every tuple, so projections and equijoins can be handled properly. By Definition 4.2 of *MaxRef* and *MinRef*, condition C3 amounts to an assertion that for each atomic inequality join predicate $S_i.A < S_j.B$, $i \neq j$, either both attributes appearing in the predicate are bounded, or else the total orders on S_i and S_j imply $S_i.A < c < S_j.B$ where c is some constant appearing in the query. (If no such constant exists, then it can be shown that $MinRef(S_i)$ cannot be zero.) In the former case, the synopses maintain full information about the attribute values, and in the latter case, the actual attribute values are not needed, since all tuples from S_i that satisfy the total order on S_i join with all tuples from S_j that satisfy the total order on S_j (at least as far as this particular join predicate is concerned). Thus no relevant information is lost by consolidating tuples into buckets.

Now consider the "only if" portion of the proof: If one of the conditions C1, C2, or C3 does not hold, then Q cannot always be evaluated in bounded memory. For each condition we show that if the condition is violated, then for any query evaluation algorithm A and any memory bound M, one can construct instances and a presentation of the input streams for which A requires more than M memory to correctly evaluate Q. The following example illustrates the technique. The complete "only if" proof is provided in Appendix A.

Consider query Q: $\dot{\pi}_A(\sigma_{B<D \wedge A>10 \wedge A<20 \wedge B>20 \wedge C<10 \wedge D>20 \wedge E<10}(S \times T))$. Q is one of the LTO queries for example query Q_4 with duplicate-preserving projection: $\dot{\pi}_A(\sigma_{B<D \wedge A>10 \wedge A<20}(S \times T))$. $S.B \in MinRef(S)$, so condition C3 is violated in Q and we assert that Q cannot be computed in bounded memory. For the sake of a contradiction suppose there exists an algorithm A that can always evaluate Q with fewer than a constant M units of memory. Define two sets of N tuples, $s = \{(15, 21, 5), (15, 22, 5), \ldots, (15, 20 + N, 5)\}$ and $t = \{(22, 5), (23, 5), \ldots, (21 + N, 5)\}$. Consider a class of inputs where the instance of stream S consists of a set of (A, B, C) tuples chosen from s and the instance of stream T consists of a single (D, E) tuple chosen from t, and suppose that all the S tuples are presented to algorithm A before the T tuple is presented. (The order in which the S tuples are presented does not matter.) After receiving the S tuples but before receiving the T tuple, algorithm A will be in some state. Since A has fewer than M units of memory, the number of distinct states that A can be in is limited. However, since there are 2^N subsets of s, for some sufficiently large N there must be two distinct subsets of s that leave A in the same state. Let s' and s'' be two such subsets, and let $(15, k, 5)$ be the tuple with the smallest value of k that is present in one of s' or s'' but not in the other. Assume without loss of generality that $(15, k, 5) \in s'$. When T consists of the tuple $(k + 1, 5)$, the correct answer to Q depends on whether S consists of s' or s'': the count of tuple (15) in the answer for s' is one more than that for s''. Since algorithm A is unable to distinguish between s' and s'', it will give the same answer in both cases and one answer will be incorrect. □

Lemma 4.1 If a query $Q(P)$ is computable in bounded memory and F is a filter predicate, then $Q(P \cup \{F\})$ also is computable in bounded memory.

Proof: Straightforward. □

Theorem 4.3 Let $Q = \dot{\pi}_L(\sigma_P(S_1 \times S_2 \times \ldots \times S_n))$. From Theorem 4.1, Q is equivalent to the union of LTO queries in (1). Q is computable in bounded memory iff every LTO query in (1) is computable in bounded memory.

Proof: Let Q be computable in bounded memory. Observe that each $T_i^{k_i}$ in Equation (1) is a filter. Using Lemma 4.1, each LTO query $Q(P \cup T_1^{k_1} \cup \cdots \cup T_n^{k_n})$ in (1) is computable in bounded memory. Now let $n > 1$ and suppose each LTO query $Q(P \cup T_1^{k_1} \cup \cdots \cup T_n^{k_n})$ in (1) is computable in bounded memory. Since the size of the answer to each LTO query is bounded (by condition C1 of Theorem 4.2), we can compute their union, and hence Q, in bounded memory. Finally, if $n = 1$ then P is a filter and both Q and the LTO queries in (1) are computable in bounded memory. □

As an illustration of Theorem 4.3, consider example query Q_5 for duplicate-preserving projection: $\dot{\pi}_A(\sigma_{B<D \wedge B<120 \wedge D>20 \wedge A>10 \wedge A<20}(S \times T))$. Like Q_4, query Q_5 is an inequality join of streams S and T, but Q_5 adds selection predicates on B and D making B upper-bounded and D lower-bounded. The presence of these additional predicates reduces $|MinRef(S)|$, $|MaxRef(S)|$, $|MinRef(T)|$, and $|MaxRef(T)|$ to zero for each LTO query for Q_5. All LTO queries for Q_5 satisfy conditions C1–C3 in Theorem 4.2, therefore Q_5 is computable in bounded memory for duplicate-preserving projection. We leave it to the reader to verify that applying Theorem 4.3 to all of the example queries in Table 1 produces the results in the column for $\dot{\pi}$.

5 Queries with Duplicate-Eliminating Projection

Our results for SPJ queries with duplicate-eliminating projection follow a similar path to the results for duplicate-preserving projection.

Theorem 5.1 Let $Q = \pi_L(\sigma_P(S_1 \times S_2 \times \cdots \times S_n))$. Q can be rewritten as $Q_1 \cup Q_2 \cup \cdots \cup Q_m$, where each Q_i is an LTO query and the unions are duplicate-eliminating.

Proof: Same as the proof of Theorem 4.1. □

Theorem 5.2 Let $Q = \pi_L(\sigma_P(S_1 \times S_2 \times \cdots \times S_n))$ be an LTO query where P is satisfiable. Q is bounded memory computable 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 P-induced equivalence classes in the element set E.

Proof: The proof is similar to the duplicate-preserving case. Here we explain how the "if" argument differs from the one for the duplicate-preserving case. We also give an example to provide some intuition for the "only if" part of the proof; the complete "only if" proof is provided in Appendix A.

As in Theorem 4.2, we keep a synopsis for each stream with tuples assigned to buckets based on the values of their bounded attributes. For duplicate-preserving projection, the synopsis stored the values of the bounded attributes for each bucket and also the count of tuples in the bucket. For queries with duplicate-eliminating projection, it is not necessary to remember the count of tuples in the bucket—it suffices to know only whether the bucket is empty or whether there has been at least one tuple assigned to the bucket. There is, however, one additional piece of information that we must store for each bucket in stream S_i's synopsis when $MinRef(S_i)$ or $MaxRef(S_i)$ is nonempty. If $MinRef(S_i)$ is nonempty, we store the the minimum value for any attribute in $MinRef(S_i)$ among tuples that have been assigned to that bucket. Similarly, if $MaxRef(S_i)$ is nonempty, we store the maximum value for any attribute in $MaxRef(S_i)$ among tuples that have been assigned to that bucket. Note that by condition C3 at most one of $MaxRef(S_i)$ and $MinRef(S_i)$ can be nonempty, and all attributes in the nonempty set must be from the same equivalence class.

With duplicate-eliminating projection, the number of tuples from other streams that join to a particular tuple of a stream S_i is not important; only the existence of at least one combination of joining tuples from the other streams is important. Therefore, if there are two tuples $t, t' \in S_i$ that agree on all attributes in the projection list L and the set of tuples from other streams that will join with t' is a subset of those that will join with t, then t' can be ignored assuming that t is remembered. This property makes it possible to determine whether predicates such as $S_i.A < S_j.B$, $i \neq j$, ever hold for any pair of tuples $t_1 \in S_i$ and $t_2 \in S_j$ by remembering the current minimum value for $S_i.A$ and the current maximum value for $S_j.B$ at any point in time.

We use an example to illustrate the technique of the "only if" proof. Consider query Q: $\pi_A(\sigma_{B<D \wedge C<E \wedge A>10 \wedge A<20 \wedge B>20 \wedge C<10 \wedge D>20 \wedge E<10}(S \times T))$. Q is one of the LTO queries for example query Q_8 with duplicate-eliminating projection: $\pi_A(\sigma_{B<D \wedge C<E \wedge A>10 \wedge A<20}(S \times T))$. $S.B \in MinRef(S)$ and $S.C \in MinRef(S)$, so condition C3 is violated in Q and we assert that Q cannot be computed in bounded memory. Consider an input presentation that begins with a large number of tuples from S that have (B, C) values of $(c, -c)$ where $c \in \{21, 23, 25, \ldots\}$. For each such S tuple with (B, C) equal to $(c', -c')$, there exists a T tuple with (D, E) equal to $(c'+1, 1-c')$ that joins only with $(c', -c')$, and not with any other S tuple. Therefore any algorithm that fails to remember all of the S tuples will produce incorrect results for some inputs. □

Theorem 5.3 Let $Q = \pi_L(\sigma_P(S_1 \times S_2 \times \ldots \times S_n))$. From Theorem 5.1, Q is equivalent to the union of LTO queries in (1). Q is computable in bounded memory iff every LTO query in (1) is computable in bounded memory.

Proof: Similar to the proof of Theorem 4.3. □

We leave it to the reader to verify that applying Theorem 5.3 to all of the example queries in Table 1 produces the results in the column for π.

6 Discussion and Future Work

To determine whether a query Q is computable in bounded memory, we can rewrite Q into the union of LTO queries in (1) and then use Theorem 4.2 or 5.2 to determine whether each of the LTO queries is computable in bounded memory. This technique would obviously be very inefficient since the number of LTO queries for Q is $\prod_{i=1}^{n} m_i$, where $m_i = |\mathrm{TO}(\mathcal{E}(S_i))|$, $S_i \in \mathcal{S}(Q)$, and $n = |\mathcal{S}(Q)|$. In Appendix B we give an efficient algorithm for determining bounded memory computability of Q that is polynomial in $|\mathcal{E}(Q)|$, where recall $\mathcal{E}(Q)$ is the set of all elements relevant to Q.

Now let us consider the sizes of our synopses. Recall that our approach is to identify, for each LTO query for query Q, constant-sized synopses of the data streams that are sufficient to evaluate the LTO query. This approach might turn out to be overly conservative in estimating memory requirements, as illustrated by example query Q_6 with duplicate-eliminating projection: $\pi_{A}(\sigma_{B>D \wedge B>E \wedge A=10}(S \times T))$. In terms of the notation defined in Section 3, $|\mathrm{TO}(\mathcal{E}(S))| = 3$ and $|\mathrm{TO}(\mathcal{E}(T))| = 13$. The query evaluation algorithm that follows from Theorem 5.2 uses one memory unit for each element of $\mathrm{TO}(\mathcal{E}(S))$ and $\mathrm{TO}(\mathcal{E}(T))$, for a total of 16 memory units. However, Q_6 also can be evaluated by maintaining the maximum value of B over all tuples in stream S with $A = 10$, and maintaining the minimum value of $max(t.D, t.E)$ over all tuples t in stream T. This scheme needs only 2 memory units.

The above example shows that our algorithm for building synopses does not build the smallest possible synopses for each query. In fact, our memory characterization is quite conservative, in part because it dramatically simplified the proofs of Theorems 4.3 and 5.3. We have designed an alternate algorithm that builds smaller synopses, including the 2-memory-unit synopses for query Q_6. Furthermore, a practical implementation can build synopses dynamically during query execution that may not approach the size of the worst-case bound. However, an algorithm for statically determining the minimum-size synopses for a query Q in all cases is a topic of future work.

Our results in Sections 4 and 5 were based on 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 3.2) 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 4.2 and 5.2 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 is useful to handle a richer set of predicates (e.g., atomic predicates using domain-specific operators, disjunctions of atomic predicates). Expanding the class of predicates is an important avenue of future work. We also plan to extend the expressiveness of the query language, e.g., by including grouping, aggregation, and subqueries.

Our results in Sections 4 and 5 also 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 presentation (including interleaving) 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 presentation 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.

7 Acknowledgements

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

References

[AMS96] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proc. of the 1996 Annual ACM Symp. on Theory of Computing*, pages 20–29, May 1996.

[BCL89] J. A. Blakeley, N. Coburn, and P. A. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. on Database Systems*, 14(3):369–400, 1989.

[BDM02] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *Proc. of the 2002 Annual ACM-SIAM Symp. on Discrete Algorithms*, January 2002.

[BW01] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, September 2001.

[CDTW00] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390, May 2000.

[DG00] N. G. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. In *Proc. of the 2000 ACM SIGCOMM*, pages 271–284, September 2000.

[DGIM02] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *Proc. of the 2002 Annual ACM-SIAM Symp. on Discrete Algorithms*, January 2002.

[FKSV99] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An approximate L1-difference algorithm for massive data streams. In *Proc. of the 1999 Annual Symp. on Foundations of Computer Science*, pages 501–511, October 1999.

[GJM96] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *Proc. of the 1996 Intl. Conf. on Extending Database Technology*, pages 140–144, March 1996.

[GKMS01] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: one-pass summaries for approximate aggregate queries. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, pages 79–88, September 2001.

[GKS01a] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 13–24, May 2001.

[GKS01b] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proc. of the 2001 Annual ACM Symp. on Theory of Computing*, pages 471–475, July 2001.

[GM95] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, June 1995.

[GMLY98] H. Garcia-Molina, W. J. Labio, and J. Yang. Expiring data in a warehouse. In *Proc. of the 1998 Intl. Conf. on Very Large Data Bases*, pages 500–511, August 1998.

[HF+00] J. M. Hellerstein, M. J. Franklin, et al. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, June 2000.

[HRR98] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. Technical Report TR-1998-011, Compaq Systems Research Center, Palo Alto, California, May 1998.

[IFF+99] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 299–310, June 1999.

[Ind00] P. Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. In *Proc. of the 2000 Annual Symp. on Foundations of Computer Science*, pages 189–197, November 2000.

[JMS95] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues for the Chronicle data model. In *Proc. of the 1995 ACM Symp. on Principles of Database Systems*, pages 113–124, May 1995.

[NACP01] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 437–448, May 2001.

[QGMW96] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc. of the 1996 Intl. Conf. on Parallel and Distributed Information Systems*, pages 158–169, December 1996.

[SPAM91] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An architecture for transforming a passive DBMS into an active DBMS. In *Proc. of the 1991 Intl. Conf. on Very Large Data Bases*, pages 469–478, September 1991.

[STR] Stanford Stream Data Management (STREAM) Project. http://www-db.stanford.edu/stream.

[TGNO92] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous queries over append-only databases. In *Proc. of the 1992 ACM SIGMOD Intl. Conf. on Management of Data*, pages 321–330, June 1992.

[Tra] Traderbot home page. http://www.traderbot.com.

[UFA98] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 130–141, June 1998.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I and II*. Computer Science Press, Rockville, Maryland, 1989.

A Proof of Theorems 4.2 and 5.2

In this section we prove Theorems 4.2 and 5.2, which identify the duplicate-preserving and duplicate-eliminating LTO queries, respectively, that can be computed in bounded memory. The "if" proofs of the theorems are given in Sections 4 and 5. We formalize the only-if proofs here. The general technique is as follows: For any LTO query that does not satisfy one (or more) of the conditions C1–C3, produce a class of input instances such that any evaluation plan using bounded memory provably fails to produce the correct answer on at least one instance of the class. In the rest of this section, a query refers to an LTO query.

Note that only condition C3 and the special case of $n = 1$ (queries involving only a single stream) differ between Theorems 4.2 and 5.2. Thus, we combine the only-if proofs for the two theorems, distinguishing the different conditions C3, the $n > 1$ in Theorem 4.2, and the different projection operators, only as necessary. Lemma A.2 proves that any query that violates condition C1, whether duplicate-preserving or duplicate-eliminating, is not computable in bounded memory. Lemma A.4 proves the same for condition C2. Lemma A.5 proves that any duplicate-preserving query that violates condition C3 in Theorem 4.2 is not computable in bounded memory. Lemma A.6 proves the same for duplicate-eliminating queries and Theorem 5.2. Lemmas A.1 and A.3 are used in the proofs of other lemmas.

We use t^X, where X is a set of attributes, to denote a tuple t with schema X. Thus $t^{\mathcal{A}(S)}$ denotes a tuple of stream S. We use $t[Y]$ to denote the projection of tuple t onto the attributes of Y. For a stream S, we use t^S and $t[S]$ as shorthand for $t^{\mathcal{A}(S)}$ and $t[\mathcal{A}(S)]$ respectively. If X and Y are disjoint sets of attributes and t_1^X and t_2^Y are tuples, $\langle t_1^X, t_2^Y \rangle$ denotes the tuple over $X \cup Y$ that uses the mapping of t_1 for attributes in X and that of t_2 for attributes in Y. In order to construct input instances for a query $Q(P)$ in our proof, we often need tuples of the streams that occur in Q that join with each other, i.e., that satisfy all the conditions of the selection predicate P. For convenience, we do this by constructing a tuple $t^{\mathcal{A}(Q)}$ that assigns a value to every attribute that is relevant to the query. We call a tuple $t^{\mathcal{A}(Q)}$ *valid* if it satisfies the predicate P.

For a query $Q(P)$, let $\text{EQ}(Q) = \{\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_m\}$ denote the equivalence classes of the elements of Q induced by P (recall the definition of element equivalence classes from Section 3). The inequality predicates induce a partial ordering on the equivalence classes—$\mathcal{E}_i < \mathcal{E}_j$ if there exist elements $e_i \in \mathcal{E}_i$ and $e_j \in \mathcal{E}_j$ such that $e_i < e_j$ is in P^+. Without loss of generality assume that the equivalence classes are numbered in *topological sort* order, i.e., if $\mathcal{E}_i < \mathcal{E}_j$ then $i < j$. A valid tuple $t^{\mathcal{A}(Q)}$ assigns the same value to all the attributes of an equivalence class. Thus, when discussing valid tuples, we may use "attribute" and "equivalence class" interchangeably.

We assume $|\mathcal{C}(Q)| > 0$ for any query Q considered in this section. The proofs can be extended in a straightforward way for $|\mathcal{C}(Q)| = 0$.

Lemma A.1 Let $m = |\text{EQ}(Q)|$ be the number of equivalence classes of the elements of an LTO query Q. Let $X \subseteq \text{EQ}(Q)$ be some set of unbounded equivalence classes of Q. Let t^X be a tuple that assigns values to the classes in X. Then we can form a valid tuple that assigns a value to all the equivalence classes $\text{EQ}(Q)$ of the form $\langle t^X, t'^{(\text{EQ}(Q)-X)} \rangle$ if:

1. For any $\mathcal{E}_i, \mathcal{E}_j$ $(i > j) \in X$, $(t[\mathcal{E}_i] - t[\mathcal{E}_j]) \geq m$.
2. For any lower-bounded $\mathcal{E}_i \in X$ (Definition 3.2) and $k \in \mathcal{C}(Q)$, $(t[\mathcal{E}_i] - k) \geq m$.
3. For any upper-bounded $\mathcal{E}_i \in X$ and $k \in \mathcal{C}(Q)$, $(k - t[\mathcal{E}_i]) \geq m$.

(Note that $t[\mathcal{E}_i]$ and $t[\mathcal{E}_j]$ are individual numeric values.)

Proof Intuitively the lemma asserts that if values of some unbounded equivalence classes, those in X, are fixed by tuple t, then we can always generate values for the other equivalence classes in Q so as to satisfy the selection predicate P, provided the values assigned to classes in X are sufficiently "far apart" from each other and from the constants in the query. The lemma is proved formally by constructing a mapping t' for all the equivalence classes in $\text{EQ}(Q) - X$.

1. Since the set of predicates P is satisfiable, there exists some valid tuple μ. Let the tuple t' use the mapping of μ for all the bounded equivalence classes, i.e., if $\mathcal{E}_i$ is bounded, $t'[\mathcal{E}_i] = \mu[\mathcal{E}_i]$.
2. For any lower-bounded equivalence class $\mathcal{E}_i \in (\text{EQ}(Q) - X)$, let $j_{mi} = \max\{j \mid j < i \text{ and } \mathcal{E}_j \in (X \cup \mathcal{C}(Q))\}$. If a constant k is in $\mathcal{E}_{j_{mi}}$, then define $t'[\mathcal{E}_i] = k + (i - j_{m_i})$. Otherwise, define $t'[\mathcal{E}_i] = t[\mathcal{E}_{j_{m_i}}] + (i - j_{m_i})$.
3. For any upper-bounded equivalence class $\mathcal{E}_i \in (\text{EQ}(Q) - X)$, let $j_{mi} = \min\{j \mid j > i \text{ and } \mathcal{E}_j \in (X \cup \mathcal{C}(Q))\}$. If a constant k is in $\mathcal{E}_{j_{mi}}$, then define $t'[\mathcal{E}_i] = k - (i - j_{m_i})$. Otherwise, define $t'[\mathcal{E}_i] = t[\mathcal{E}_{j_{m_i}}] - (i - j_{m_i})$.

It is straightforward to prove that the tuple $\langle t, t' \rangle$ is valid. □

Lemma A.2 Let Q be an LTO query and L its list of projected attributes. Let attribute $A \in L$ be unbounded. Then Q is not computable in bounded memory if:

1. $|\mathcal{S}(Q)| > 1$, or

2. Q has a duplicate-eliminating projection.

Proof Let A belong to stream S_a. Let $\mathcal{E}_a$ be the equivalence class of A. Since A is unbounded, for any integer N we can construct valid tuples $t_1, \ldots, t_N$ defined on $\mathcal{A}(Q)$ such that each t_i has a different value for attribute A, i.e., $t_i[A] \neq t_j[A]$ if $i \neq j$. We construct 2^N presentations, $\mathcal{P}_1, \ldots, \mathcal{P}_{2^N}$, where each presentation consists (in any order) of a different subset of the set of tuples $\{t_1[S_a], \ldots, t_N[S_a]\}$ of stream S_a. We claim that any correct evaluation algorithm for query Q must be in a different memory state after seeing each of the above presentations. Consider our two cases.

Case 1 ($|\mathcal{S}(Q)| > 1$). In this case the query involves more than one stream. Since none of the presentations $\mathcal{P}_i$ contain tuples from streams other than S_a, the answer at the end of each presentation is empty. Consider any two presentations $\mathcal{P}_i$ and $\mathcal{P}_j$ $(i \neq j)$. Without loss of generality, assume there exists a tuple $t_k[S_a]$ that occurs in $\mathcal{P}_i$ but not in $\mathcal{P}_j$. Consider the set of tuples $\Delta\mathcal{P} = \{t_k[S_x] : (x \neq a, S_x \in \mathcal{S}(Q))\}$. If the set of tuples $\Delta\mathcal{P}$ appears in the input after $\mathcal{P}_i$, the tuple $t_k[L]$ is in the answer. The tuple $t_k[L]$ is not in the answer if $\Delta\mathcal{P}$ occurs after $\mathcal{P}_j$. Thus any evaluation algorithm for query Q that has the same memory state after seeing either $\mathcal{P}_i$ or $\mathcal{P}_j$ will produce the wrong answer for at least one of two inputs: $\mathcal{P}_i$ followed by $\Delta\mathcal{P}$, or $\mathcal{P}_j$ followed by $\Delta\mathcal{P}$.

Case 2 *(Q has duplicate-eliminating projection).* Assume $|\mathcal{S}(Q)| = 1$; otherwise Case 1 holds. The answer after seeing $\mathcal{P}_i$ is the projection of the tuples of $\mathcal{P}_i$ onto the attributes in L. Consider any two distinct presentations $\mathcal{P}_i$ and $\mathcal{P}_j$ and (without loss of generality) a tuple t_k that occurs in $\mathcal{P}_i$ but not $\mathcal{P}_j$. If tuple t_k appears (again) after $\mathcal{P}_i$, there are no new tuples produced in the answer because t_k is a duplicate. If, however, t_k appears after $\mathcal{P}_j$, the tuple $t_k[L]$ must be produced in the answer when t_k appears. Thus any evaluation algorithm for Q that has the same memory state after seeing either $\mathcal{P}_i$ or $\mathcal{P}_j$ will produce the wrong answer for at least one of the two input scenarios above.

Any correct evaluation algorithm for Q needs at least the $log\ 2^N = \Theta(N)$ bits of memory that are required to encode different states for each of $\mathcal{P}_1, \ldots, \mathcal{P}_{2^N}$. Therefore Q is not computable in bounded memory. □

Lemma A.3 Let Q be an LTO query with $|\mathcal{S}(Q)| > 1$. Q is not computable in bounded memory if for any value of N there exists a stream $S \in \mathcal{S}(Q)$ and valid tuples $t_1, t_2, \ldots, t_N$ defined on $\mathcal{A}(Q)$ such that:

1. The projections of the tuples on S are all distinct, i.e., $t_i[S] \neq t_j[S]$ if $i \neq j$, and

2. $\langle t_i[S], t_j[\mathcal{A}(Q) - \mathcal{A}(S)]\rangle$ is not valid if $i \neq j$.

Proof Consider any value of N. We construct 2^N presentations, $\mathcal{P}_1, \ldots, \mathcal{P}_{2^N}$, where each presentation consists (in any order) of a different subset of the set of tuples $\{t_1[S], \ldots, t_N[S]\}$ of stream S. We claim that any correct evaluation algorithm for Q must be in a different memory state after seeing each of the above presentations. Since none of the presentations $\mathcal{P}_i$ contains tuples from streams other than S, the answer at the end of each presentation is empty. Consider any two presentations $\mathcal{P}_i$ and $\mathcal{P}_j$ $(i \neq j)$. Without loss of generality, assume there exists a tuple $t_k[S]$ that occurs in $\mathcal{P}_i$ but not in $\mathcal{P}_j$. Consider the set of tuples $\Delta\mathcal{P} = \{t_k[S'] \mid (S' \neq S, S' \in \mathcal{S}(Q))\}$. If the set of tuples $\Delta\mathcal{P}$ appears in the input after $\mathcal{P}_i$, a new answer tuple $t_k[L]$ is generated. If $\Delta\mathcal{P}$ appears in the input after $\mathcal{P}_j$, no new answer tuples are generated because of the second condition in the lemma. Thus any evaluation algorithm for Q that has the same memory state after seeing $\mathcal{P}_i$ and $\mathcal{P}_j$ will produce the wrong answer for at least one of two inputs: $\mathcal{P}_i$ followed by $\Delta\mathcal{P}$, or $\mathcal{P}_j$ followed by $\Delta\mathcal{P}$. Thus any correct evaluation algorithm for Q needs at least $log\ 2^N = \Theta(N)$ bits of memory, so Q is not computable in bounded memory. □

Lemma A.4 Let $Q(P)$ be an LTO query. Let there be an equality join condition of the form $S_i.A = S_j.B$ in P such that both attributes A and B are unbounded. Then $Q(P)$ is not computable in bounded memory.

Proof Let $\mathcal{E}$ denote the (common) unbounded equivalence class of A and B. For any N we can construct valid tuples $t_1, \ldots, t_N$ defined on $\mathcal{A}(Q)$ such that each tuple assigns a different value to the equivalence class $\mathcal{E}$, i.e., $t_k[\mathcal{E}] \neq t_l[\mathcal{E}]$ if $k \neq l$. Observe that these tuples satisfy the conditions of Lemma A.3 by using S_i of this lemma as the S in Lemma A.3. The projections $t_k[S_i]$ of the tuples onto S_i are all distinct since they differ at least on the attribute A (equivalence class $\mathcal{E}$). The tuple $\langle t_k[S_i], t_l[\mathcal{A}(Q) - \mathcal{A}(S_i)]\rangle\ (k \neq l)$ is not valid since $S_i.A = S_j.B$ is not satisfied. Thus, by Lemma A.3, Q is not computable in bounded memory. □

Lemma A.5 Let $Q(P)$ be a duplicate-preserving LTO query such that $|MaxRef(S)| \neq 0$ or $|MinRef(S)| \neq 0$ for some $S \in \mathcal{S}(Q)$. Then $Q(P)$ is not computable in bounded memory.

Proof We prove the case $|MaxRef(S_m)| \neq 0$ for some $S_m \in \mathcal{S}(Q)$. The proof for $|MinRef(S_m)| \neq 0$ is symmetric. By Definition 4.2 of $MaxRef$ there exists a nonredundant inequality join $S_n.B < S_m.A\ (n \neq m)$. Let $\mathcal{E}_a$ and $\mathcal{E}_b$ denote the equivalence classes induced by P for A and B respectively. Note that these two equivalence classes are distinct since P is satisfiable. Assume that neither $\mathcal{E}_a$ nor $\mathcal{E}_b$ is upper-bounded. (There is no loss of generality because the case when neither is lower-bounded is symmetric and the case where one is lower-bounded and the other upper-bounded is disallowed by the definition of $MaxRef$.) For notational convenience, let $<_m$ denote "less than by at least m". Let k_{max} represent the maximum constant occurring in Q, let $m =$|EQ(Q)|, and let $X = \{\mathcal{E}_a, \mathcal{E}_b\}$. Given any integer N define tuples $t_1^X, \ldots, t_N^X$ with t_i^X having $\mathcal{E}_a = k_{max} + 2im$ and $\mathcal{E}_b = k_{max} + (2i+1)m$. From these tuples defined on X we can use Lemma A.1 to construct valid tuples $t_1, \ldots, t_n$ defined on $\mathcal{A}(Q)$ such that:

$$k_{max} <_m t_1[\mathcal{E}_b] <_m t_1[\mathcal{E}_a] <_m t_2[\mathcal{E}_b] <_m t_2[\mathcal{E}_a] <_m \cdots <_m t_N[\mathcal{E}_b] <_m t_N[\mathcal{E}_a]$$

We construct 2^N presentations, $\mathcal{P}_1, \ldots, \mathcal{P}_{2^N}$, where each presentation consists (in any order) of a different subset of the set of tuples $\{t_1[S_m], \ldots, t_N[S_m]\}$ of stream S_m. We claim that any evaluation algorithm for the query Q has to be in a different memory state after seeing each of the above presentations. Since none of the presentations $\mathcal{P}_i$ contain tuples from streams other than S_m, the answer at the end of each presentation is empty. Consider any two presentations $\mathcal{P}_i$ and $\mathcal{P}_j\ (i \neq j)$. Let $t_k[S_m]$ be the tuple with largest value of k among all tuples appearing in one of $\mathcal{P}_i, \mathcal{P}_j$, but not in the other. Without loss of generality, assume $t_k[S_m]$ appears in $\mathcal{P}_i$ but not in $\mathcal{P}_j$. Note that all the tuples $t_l[S_m]\ (l > k)$ occur either in both $\mathcal{P}_i$ and $\mathcal{P}_j$ or in neither. Now consider the set of tuples $\Delta\mathcal{P} = \{t_k[S_x] : S_x \neq S_m, S_x \in \mathcal{S}(Q)\}$. Consider two input cases — the occurrence of $\Delta\mathcal{P}$ after $\mathcal{P}_i$ and the occurrence of $\Delta\mathcal{P}$ after $\mathcal{P}_j$. The answer in the former case is equal to the answer in the latter case with an additional tuple $t_k[L]$. By definition tuples in $\Delta\mathcal{P}$ join with $t_k[S_m]$ to produce $t_k[L]$ in the answer. Also, tuples in $\Delta\mathcal{P}$ do not join with any tuple $t_l[S_m]\ (l < k)$ since this would cause the predicate $B < A$ to be violated. It is possible that the tuples in $\Delta\mathcal{P}$ join with some tuples $t_l[S_m]\ (l > k)$, but by our choice of k, any such tuple $t_l[S_m]$ that occurs in either $\mathcal{P}_i$ or $\mathcal{P}_j$ also occurs in the other and thus will lead to the same answer in both the cases. Consequently any evaluation algorithm for Q that has the same memory state after seeing $\mathcal{P}_i$ and $\mathcal{P}_j$ will produce the wrong answer for at least one of the two inputs described above. Therefore any correct evaluation algorithm for Q needs at least $log\ 2^N = \Theta(N)$ bits of memory, so Q is not computable in bounded memory. □

Lemma A.6 Let $Q(P)$ be a duplicate-eliminating LTO query such that $|MaxRef(S)|_{eq} + |MinRef(S)|_{eq} > 1$ for some $S \in \mathcal{S}(Q)$. Then $Q(P)$ is not computable in bounded memory.

Proof We assume that there are no equality joins between unbounded attributes of different streams, or else Lemma A.4 applies. The proof is split into three cases. In each case we show that given any number N we

can generate tuples $t_1, \ldots, t_N$ that satisfy the conditions of Lemma A.3, proving that Q is not computable in bounded memory. We only generate values for some of the attributes for each tuple and use Lemma A.1 to "fill in" the values of the other tuples. As in Lemma A.5, let $<_m$ denote "less by at least m".

Case 1 ($|MinRef(S_m)|_{eq} > 1$ *for some* $S_m \in \mathcal{S}(Q)$). In this case there exist nonredundant atomic predicates $p_1 = (S_m.A < S_n.C)$ and $p_2 = (S_m.B < S_o.D), m \neq n, m \neq o$ in P. By Definition 4.2 of *MinRef*, attributes A and B are unbounded and belong to different equivalence classes. Since the attributes of any stream in an LTO query are totally ordered we can assume without loss of generality that $A < B \in P^+$.

We claim that all four attributes A, B, C, D are unbounded and belong to different equivalence classes. As noted above A and B are unbounded and belong to different equivalence classes. Attributes C and D must be unbounded since otherwise p_1 and p_2 would be redundant, which contradicts the definition of *MinRef*. Moreover, since we assume that there are no equality joins between unbounded attributes belonging to different streams, all the attributes in an unbounded equivalence class belong to the same stream. Thus neither C nor D can be in the same equivalence class as either A or B. Finally, C and D cannot be in the same equivalence class as each other since otherwise the predicates $A < B$ and $B < D$ would imply that p_1 is redundant. This proves the above claim.

Let $\mathcal{E}_a, \mathcal{E}_b, \mathcal{E}_c, \mathcal{E}_d$ be the equivalence classes of A, B, C, D respectively. Recall that we use the convention that the subscripts i of equivalence classes $\mathcal{E}_i$ are in topological sort order, meaning that $i < j$ if $\mathcal{E}_i < \mathcal{E}_j$. The predicates $A < C, A < B, B < D \in P^+$ imply that $a < c$ and $a < b < d$. It cannot be the case that $\mathcal{E}_b < \mathcal{E}_c$ since that would imply that $A < C$ is redundant. Therefore either $\mathcal{E}_c < \mathcal{E}_b$ or $\mathcal{E}_c$ and $\mathcal{E}_b$ are unrelated according to "$<$". In either case, we can construct a valid topological sort ordering satisfying $c < b$. Thus without loss of generality we can assume $a < c < b < d$.

We consider three subcases of Case 1 depending on whether the attributes A, B, C, D are lower-bounded or upper-bounded. Let k_{min} and k_{max} denote the smallest and largest constants occurring in Q, respectively.

Case 1a *(None of* A, B, C, D *are lower-bounded).* Given any N, select values for $t_i[\mathcal{E}_a]$, $t_i[\mathcal{E}_b]$, $t_i[\mathcal{E}_c]$, and $t_i[\mathcal{E}_d]$, for $i = 1, \ldots, N$, such that:

$$\begin{aligned} & t_1[\mathcal{E}_a] <_m t_1[\mathcal{E}_c] < t_2[\mathcal{E}_a] <_m t_2[\mathcal{E}_c] < \ldots < t_N[\mathcal{E}_a] <_m t_N[\mathcal{E}_c] \\ <_m\ & t_N[\mathcal{E}_b] <_m t_N[\mathcal{E}_d] < \ldots < t_1[\mathcal{E}_b] <_m t_1[\mathcal{E}_d] \\ <_m\ & k_{min} \end{aligned}$$

Using Lemma A.1, we can construct tuples $t_1, \ldots, t_N$ defined on $\mathcal{A}(Q)$ based on the above assignment of values to equivalence classes in the set $X = \{\mathcal{E}_a, \mathcal{E}_b, \mathcal{E}_c, \mathcal{E}_d\}$. The projections $t_i[S_m]$ onto the stream S_m are all distinct since $t_i[A] \neq t_j[A]$, if $i \neq j$. The tuple $\langle t_i[S_m], t_j[\mathcal{A}(Q) - \mathcal{A}(S_m)]\rangle$ is not valid since $t_i[\mathcal{E}_d] < t_j[\mathcal{E}_b]$ if $i < j$ (violating the predicate $B < D$) and $t_i[\mathcal{E}_a] > t_j[\mathcal{E}_c]$ if $i > j$ (violating the predicate $A < C$). Thus it follows from Lemma A.3 that query Q is not computable in bounded memory.

Case 1b *(All of* A, B, C, D *are lower-bounded).* In this case, the tuples $t_1, \ldots, t_N$ are constructed such that:

$$\begin{aligned} k_{max} <_m\ & t_1[\mathcal{E}_a] <_m t_1[\mathcal{E}_c] < t_2[\mathcal{E}_a] <_m t_2[\mathcal{E}_c] < \ldots < t_N[\mathcal{E}_a] <_m t_N[\mathcal{E}_c] \\ <_m\ & t_N[\mathcal{E}_b] <_m t_N[\mathcal{E}_d] < \ldots < t_1[\mathcal{E}_b] <_m t_1[\mathcal{E}_d] \end{aligned}$$

The remainder of the argument proceeds similarly to Case 1a.

Case 1c *(*A, C *are upper-bounded and* B, D *are lower-bounded).* For this case, $t_1, \ldots, t_N$ satisfy:

$$\begin{aligned} & t_1[\mathcal{E}_a] <_m t_1[\mathcal{E}_c] < t_2[\mathcal{E}_a] <_m t_2[\mathcal{E}_c] < \ldots < t_N[\mathcal{E}_a] <_m t_N[\mathcal{E}_c] \\ <_m\ & k_{min} \\ <_m\ & k_{max} \\ <_m\ & t_N[\mathcal{E}_b] <_m t_N[\mathcal{E}_d] < \ldots < t_1[\mathcal{E}_b] <_m t_1[\mathcal{E}_d] \end{aligned}$$

The remainder of the argument again parallels Case 1a. This completes the proof of Case 1.

Case 2 ($|MaxRef(S_m)|_{eq} > 1$). Symmetric to the proof of Case 1.
Case 3 ($|MaxRef(S_m)|_{eq} = |MaxRef(S_m)|_{eq} = 1$). In this case there exist nonredundant atomic predicates $p_1 = (S_m.A > S_n.C)$ and $p_2 = (S_m.B < S_o.D), m \neq n, m \neq o$ in P. We consider two subcases. In each subcase we specify only the relationships that need to hold between the values chosen for the projections of tuples $t_1, \ldots, t_n$ onto the set $X = \{\mathcal{E}_a, \mathcal{E}_b, \mathcal{E}_c, \mathcal{E}_d\}$. The remaining details are analogous to the proof for Case $1a$. Note that C and D can be the same attribute.
Case 3a (*Either none of A, B, C, D is lower-bounded or none is upper-bounded*). Assume A, B, C, D are all lower-bounded. The same proof applies when some attributes are neither upper-bounded nor lower-bounded, and the proof when all the attributes are upper-bounded is symmetric. We generate tuples $t_1, \ldots, t_N$ with the following property: for any t_i, t_j $(i < j)$, the values assigned by t_i to attributes A, B, C, D are all smaller than the values assigned by t_j to A, B, C, D, i.e., $t_i[X] < t_j[Y]$, for $X, Y \in \{A, B, C, D\}$. Further, each t_i assigns values to attributes A, B, C, D such that the predicates p_1 and p_2 are satisfied. For example, if A, B, C, D belong to distinct equivalence classes $\mathcal{E}_a, \mathcal{E}_b, \mathcal{E}_c, \mathcal{E}_d$ and $a > c > b > d$, then the tuples generated satisfy:

$$\begin{aligned} k_{max} <_m\ & t_1[\mathcal{E}_d] <_m t_1[\mathcal{E}_b] <_m t_1[\mathcal{E}_c] <_m t_1[\mathcal{E}_a] \\ <\ & t_2[\mathcal{E}_d] <_m t_2[\mathcal{E}_b] <_m t_2[\mathcal{E}_c] <_m t_2[\mathcal{E}_a] \\ <\ & \ldots \\ <\ & t_N[\mathcal{E}_d] <_m t_N[\mathcal{E}_b] <_m t_N[\mathcal{E}_c] <_m t_N[\mathcal{E}_a] \end{aligned}$$

Case 3b (*Some among A, B, C, D are lower-bounded and some upper-bounded*). Either A and C (respectively B and D) are both lower-bounded or are both upper-bounded, otherwise the predicate p_1 (respectively p_2) is redundant. Let us assume that A, C are upper-bounded and B, D lower-bounded. Let $\mathcal{E}_a, \mathcal{E}_b, \mathcal{E}_c, \mathcal{E}_d$ be the equivalence classes of A, B, C, D respectively. None of these equivalence classes can be the same or P would be unsatisfiable. For this case the tuples $t_1, \ldots, t_N$ satisfy:

$$\begin{aligned} & t_1[\mathcal{E}_c] <_m t_1[\mathcal{E}_a] < t_2[\mathcal{E}_c] <_m t_2[\mathcal{E}_a] < \ldots < t_N[\mathcal{E}_c] <_m t_N[\mathcal{E}_a] \\ <_m\ & k_{min} \\ \leq\ & k_{max} \\ <_m\ & t_1[\mathcal{E}_b] <_m t_1[\mathcal{E}_d] < t_2[\mathcal{E}_b] <_m t_2[\mathcal{E}_d] < \ldots < t_N[\mathcal{E}_b] <_m t_N[\mathcal{E}_d] \end{aligned}$$

The case when A, C are lower-bounded and B, D upper-bounded is symmetric. □

B Efficient Algorithm for Checking Bounded-Memory Computability

A naive algorithm for determining whether a query $Q(P)$ is computable in bounded memory enumerates all the LTO queries of Q and checks if each one is computable in bounded memory. This approach can be very expensive since there are an exponential number of LTO queries. We propose a simple polynomial algorithm that checks if a query Q is computable in bounded memory without explicitly checking each LTO query of Q.

The outline of the algorithm is shown below. The algorithm handles both duplicate-preserving and duplicate-eliminating SPJ queries.

Algorithm B.1 (Check Bounded-Memory Computability of an SPJ Query)
Input: SPJ query $Q = \pi_L(\sigma_P(S_1 \times S_2 \times \cdots \times S_n))$
Output: Yes, if Q is computable in bounded memory. No, otherwise.

1. If P is not satisfiable, or if $n = 1$ and Q is a duplicate-preserving query, 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, $\mathrm{IND}(X \cup \{k_{max}, k_{min}\}, P)$ 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 4.3 and 5.3,) return No.
5. Return Yes.

$\diamond$

Step 1 checks if Q is trivially computable in bounded memory. Steps 2 and 3 check if condition C1 or C2 of Theorem 4.2 or 5.2 is violated. Step 4 checks condition C3 in Theorem 4.2 (if Q is duplicate-preserving) or Theorem 5.2 (if Q is duplicate-eliminating). If none of the conditions C1–C3 are violated, Q is computable in bounded memory and Yes is returned in Step 5.

The correctness of steps 3 and 4 follows from the observation that an unbounded attribute in any LTO query of Q is unbounded in Q as well, and any unbounded attribute in Q is unbounded in some LTO query of Q. Suppose some LTO query Q_L of a duplicate-eliminating query Q violates condition C3. Then there exist two non-filter, nonredundant predicates p_1 and p_2 in Q_L that cause violation of condition C3. Let X be the set of (at most 4) attributes that occur in either p_1 or p_2. It can be shown that query Q' constructed from X in Step 4 is not computable in bounded memory. Thus our algorithm correctly returns No. Conversely, if there is some Q' that is not computable in bounded memory, it can be shown that there exists an LTO query of Q that violates condition C3 of Theorem 5.2. A similar argument holds for the duplicate-preserving case.

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 $\mathcal{A}(Q)$, Step 4 takes polynomial time. Thus Algorithm B.1 is polynomial in the size of the input query.

C Queries with Self-Joins

We extend our main results to queries containing 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 (natural) self-join of stream S with itself on attribute A. Note that unlike two different streams, there is an implicit constraint on self-joined streams: at any point of time, the instances of $S^{(j)}$ and $S^{(k)}$ are the same for any j, k.

For duplicate-preserving queries, all of our results in Section 4 (most importantly Theorems 4.2 and 5.2) carry over to queries with self-joins. Some modifications to the "only if" part of the proof of Theorem 4.2 are needed to accommodate self-joins; all other reasoning and proofs carry over directly. The efficient algorithm in Appendix B also applies, so it can be used to test whether a duplicate-preserving query with self-joins is computable in bounded memory.

Hereafter we consider duplicate-eliminating SPJ queries. Theorem 5.3 remains valid in the presence of self-joins. However, Theorem 5.2 does not hold for self-join LTO queries, as the following example illustrates.

Example C.1 Consider the following LTO query Q:

$$Q = \pi_{S^{(1)}.A}(\sigma_{S^{(1)}.A=10 \wedge S^{(1)}.A=S^{(2)}.A \wedge S^{(1)}.B=S^{(2)}.B \wedge S^{(1)}.B>10}(S^{(1)} \times S^{(2)}))$$

Attributes $S^{(1)}.B$ and $S^{(2)}.B$ are unbounded, which violates condition C2 of Theorem 5.2. However, query Q is equivalent to the query $Q' = \pi_A(\sigma_{A=10 \wedge B>10}(S))$, which is clearly computable in bounded memory. □

Conditions C1–C3 in Theorem 5.2 are still sufficient to ensure that an LTO query is computable in bounded memory, but they are not necessary, i.e., there exist LTO queries (e.g., query Q of Example C.1) that violate one or more of conditions C1–C3, but are computable in bounded memory. In our example, query Q with two occurrences of stream S is equivalent to the reduced query Q' with only one occurrence of S. Intuitively, one of the occurrences of S in Q was redundant. We generalize this observation to obtain a characterization of bounded-memory computability for duplicate-eliminating self-join queries.

Definition C.1 (**Redundant Stream**) Consider a duplicate-eliminating LTO query $Q(P)$. A stream $S_i^{(j)}$ in Q is said to be *redundant* if there exists a stream $S_i^{(k)} (j \neq k)$ such that:

1. The total ordering of elements $\mathcal{E}(S_i^{(j)})$ of $S_i^{(j)}$ is the same as the ordering of elements $\mathcal{E}(S_i^{(k)})$ of $S_i^{(k)}$.
2. If $S_i^{(j)}.A \in L$, where L is the list of projected attributes, then predicate $(S_i^{(j)}.A = S_i^{(k)}.A) \in P^+$.
3. If $(S_i^{(j)}.A \;\; Op \;\; S_j.B) \in P$, then $(S_i^{(k)}.A \;\; Op \;\; S_j.B) \in P^+$

We say that $S_i^{(k)}$ *covers* $S_i^{(j)}$ in Q. □

Let Q be a self-join LTO query such that $S_i^{(j)} \in \mathcal{S}(Q)$ is redundant. Let $S_i^{(k)}$ be a stream that covers $S_i^{(j)}$. The query Q' obtained from Q by eliminating $S_i^{(j)}$ and replacing every occurrence of attribute $S_i^{(j)}.A$ by $S_i^{(k)}.A$ is equivalent to Q.

Theorem 5.2 holds for LTO queries with self-joins provided they do not contain any redundant occurrences of a stream. Thus, to check bounded-memory computability of a duplicate-eliminating query Q with self-joins, we first eliminate all redundant occurrences of streams (e.g., using a greedy algorithm) to obtain an equivalent query Q'. We can then check the conditions of Theorem 5.2 for Q', or use the more efficient algorithm in Appendix B.