The Thirty-Third International
FLAIRS Conference (FLAIRS-33)

Bounded-Memory Criteria for Streams with Application Time

Simon Schiff, Özgür L. Özçep
Institute of Information Systems (IFIS)
University of Lübeck
Lübeck, Germany
{schiff,oezcep}@ifis.uni-luebeck.de

Abstract

Bounded-memory computability continues to be in the focus of those areas of AI and databases that deal with feasible computations over streams—be it feasible arithmetical calculations on low-level streams or feasible query answering for declaratively specified queries on relational data streams or even feasible query answering for high-level queries on streams w.r.t. a set of constraints in an ontology such as in the paradigm of Ontology-Based Data Access (OBDA). In classical OBDA, a high-level query is answered by transforming it into a query on data source level. The transformation requires a rewriting step, where knowledge from an ontology is incorporated into the query, followed by an unfolding step with respect to a set of mappings. Given an OBDA setting it is very difficult to decide, whether and how a query can be answered efficiently. In particular it is difficult to decide whether a query can be answered in bounded memory, i.e., in constant space w.r.t. an infinitely growing prefix of a data stream. This work presents criteria for bounded-memory computability of select-project-join (SPJ) queries over streams with application time. Deciding whether an SPJ query can be answered in constant space is easier than for high-level queries, as neither an ontology nor a set of mappings are part of the input. Using the transformation process of classical OBDA, these criteria then can help deciding the efficiency of answering high-level queries on streams.

Introduction

The potential infinity and velocity of stream data is a big challenge for designing streaming engines that are going to be used in an agent, in a data stream management system (DSMS) or in any other system that has to process streams. This holds true regardless of whether one considers engines for doing arithmetical calculations on low-level streams (such as sensor percepts in agents), answering queries in a DSMS, answering high-level queries w.r.t. a set of constraints in an ontology such as in the paradigm of Ontology-Based Data Access (OBDA), or considering a stream of actions and belief states in the agent paradigm.

Usually, stream queries are registered at a stream engine at some point and then evaluated continuously on an ever growing prefix of one or more input streams. Efficient algorithms that evaluate registered queries continuously are indispensable. However, deciding whether a query can be evaluated efficiently is a non-trivial task. And even if it is known that a query is computable efficiently it does not automatically lead to a procedure that generates an algorithm evaluating that query efficiently. If one could find criteria to identify such queries and a procedure that generates an algorithm respectively, then this would mean a real benefit for efficient stream processing.

In this paper, we are going to focus on one aspect of efficient stream processing dealt under the term "bounded-memory computation", namely keeping space consumption during the evaluation as low as possible, in particular keeping it constant in the size of the prefixes of the streams processed so far. A stream engine in an agent or a DSMS has only a bounded amount of space available, regardless of hard disk storage or main memory. If more than constant space is required, for instance linear space with respect to the ever growing prefix of one or more input streams, then a system will run out of memory at some point. In that case, it is not possible to evaluate such a registered query correctly.

Recent efforts where made to temporalize and streamify classical OBDA for processing streams of data (Baader, Borgwardt, and Lippmann 2013). High-level queries are written with respect to a signature of an ontology and answered by transforming them into queries on data source level. The ontology is a knowledge base and can be maintained by an expert of a specific domain such as an engineer. Mappings map ontology predicates into a query on data source level. Therefore, mappings are defined and maintained by an IT-specialist. Such an OBDA approach is also of interest for AI research on rational agents whose knowledge on the environment, e.g., is encoded in an ontology.

Given an OBDA setting, deciding whether a high-level query is bounded-memory computable (bm-computable) can be difficult. However, in case of classical OBDA, a high-level query can be transformed into a query on data source level such as the structured query language (SQL). Deciding bounded-memory computability for SQL queries is easier than for high-level queries as no ontology or set of mappings is part of the input. Nevertheless, some assumptions made on the ontology level need to be considered on the data source level. These assumptions and many more, as presented later, heavily influence the criteria for testing whether a SQL query is bm-computable.

We present criteria for bounded-memory computability of SQL queries over relational data streams with a specific attribute for application time. We assume that the SQL queries are transformation outputs of high-level queries in a streamified OBDA scenario and that they may contain constraints from the ontology and the high-level execution model.

Preliminaries

Queries that can be evaluated in constant space with respect to an ever growing prefix of one or more input streams are said to be bm-computable (Definition 1).

Definition 1 (Definition 3.1 in (Arasu et al. 2004))**.** 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 of the input streams of the query.

An instance of a stream is at any point of time a bag of tuples seen so far. Only a bounded amount of tuples can be stored in memory as otherwise, more than constant space is required during query evaluation. The bounded amount of tuples can be seen as a representation for the instance of a stream. Such a representation is later referred to as a synopsis and results have to be the same at any point of time regardless of whether query Q is applied on the synopsis or the instance of a stream.

At any time, the instance of the output stream is defined as the result of applying query Q on the database instance that contains all tuples ever received. Therefore, we have to restrict the class of queries to monotonic ones. As usual, a query Q over a schema R is called monotonic iff for every two instances I, J of R: $I \subseteq J \rightarrow Q(I) \subseteq Q(J)$. In particular, an SQL query is monotonic, if it does not contain negation or aggregation. This is the case for the class of select-project-join (SPJ) queries and their (polyadic) unions, which are in the focus of this paper.

If the projection of a query is duplicate preserving, then the polyadic union operator is duplicate preserving and if the projection is duplicate eliminating, then the polyadic union operator is duplicate eliminating. It can be shown that a set of SPJ queries, combined by a polyadic union operator, is bm-computable if every SPJ query in the set is bm-computable. Moreover, not every SPJ query in a set is necessarily bm-computable if the union of every SPJ in the set is bm-computable. SPJ queries have static relations (Definition 2), infinite streams and finite streams as input.

Definition 2. A *static relation* R consists of a finite bag of tuples having the same schema.

We draw a distinction between an *infinite stream* and a *finite stream* (Definition 3). The distinction is necessary for defining criteria testing whether a transformed query is bm-computable or not.

Definition 3. An *infinite (finite) stream* S (F) is an infinite (finite) sequence of relational tuples having the same schema.

A finite or infinite stream is referred to as a stream in the following, except when there is a notable difference.

We assume that the SPJ queries that are going to be tested for bounded-memory computability are given in a specific normal form as described in (Definition 4).

Definition 4 (Extension of Subsection 4.4 in (Abiteboul, Hull, and Vianu 1995))**.** An SPJ algebra query is in normal form iff it has the form:

$$\Pi_L \left(\bigtimes_{i=1}^{m} \{\langle a_i \rangle\} \times \sigma_P \left(\bigtimes_{i=1}^{k} R_i \times \bigtimes_{i=1}^{l} S_i \times \bigtimes_{i=1}^{p} F_i \right) \right)$$

where $\Pi \in \{\dot{\pi}, \pi\}$ is a duplicate-preserving projection operator $\dot{\pi}$ or a duplicate-eliminating projection operator π, $L = \{j_1, \dots, j_n\}$, $n \geq 0$ is the list of attributes projected out, $a_1, \dots, a_m \in$ **dom** a set of constants in the domain, $m \geq 0$, $\{1, \dots, m\} \subseteq \{j_1, \dots, j_n\}$, $R_1, \dots, R_k$ are static relation names (repeats permitted), $S_1, \dots, S_l$ are infinite stream names (no repeats permitted), $F_1, \dots, F_p$ are finite stream names (no repeats permitted), and P is a set of atoms of the form $(X \text{ op } Y)$ where op ranges over $\{<, =, >, \neq\}$ (Abiteboul, Hull, and Vianu 1995). X is an attribute and Y is either an attribute, integer or timestamp. The comparison of constants is omitted, as they evaluate either to true or false. No timestamp attribute is in the project list L. Each infinite stream or finite stream contains a timestamp attribute. The comparison of timestamp and integer attributes is forbidden. We assume that the size of a query is bounded by a constant.

We forbid the comparison of timestamps and integers as this would require a conversion of timestamps to integers, and there is possibly more than one way of converting a timestamp into an integer (and vice versa).

An important notion for our criteria is that of boundedness of attributes. Arasu and colleagues (Arasu et al. 2004) define the boundedness of attributes in queries that refer only to infinite streams. The boundedness of attributes in queries, which are the results of a transformation in a OBDA scenario considered in this paper, differ, as transformed queries refer additionally to static relations and finite streams.

Definition 5. If selection P^+ contains an equality join of the form $(S_i.A = R_j.B)$ or $(S_i.A = k)$ for some constant k, then attribute A of stream S_i is *lower-bounded* and *upper-bounded*. An attribute A is *lower-bounded (upper-bounded)* if the selection P^+ contains an inequality join of the form $(S_i.A > R_j.B)$ or $(S_i.A > k)$ ($(S_i.A < R_j.B)$ or $(S_i.A < k)$). If an attribute is lower-bounded and upper-bounded, then it is *bounded*, otherwise it is *unbounded*.

As mentioned in the beginning, a high-level query Q_H written with respect to an ontology can be transformed into a query on data source level, such as a SPJ query, depending on the expressiveness Q_H and the ontology language.

Usually, classical OBDA query answering is defined under set semantics and transformed queries have a duplicate eliminating operator. However, it is possible and reasonable to define OBDA query answering under bag-semantics (Nikolaou et al. 2017). Therefore, we present bounded-memory requirements for queries with a duplicate eliminating as well as duplicate preserving operators.

Execution Model

According to Definition 3, an infinite stream S_j $(1 \leq j \leq l)$ is an infinite sequence of relational tuples having the same schema. The domain of each attribute is the set of integers or timestamps from a flow of time $(\mathfrak{T}, \leq_{\mathfrak{T}})$, where $\mathfrak{T}$ has 0 as minimum, has no last element, is discrete, and $\leq_{\mathfrak{T}}$ is non-branching (i.e. linear). Exactly one attribute of a schema is from the domain of timestamps. The amount of tuples having the same timestamp is bounded by a constant.

A finite stream F_j $(1 \leq j \leq p)$ is a stream except that $\mathfrak{T}$ has a last element (Definition 3). We assume that the last element of $\mathfrak{T}$ is known before processing starts.

All streams are synchronized with respect to the timestamps and every tuple with the same timestamp fits into memory. An evaluation plan can assume that tuples of a stream arrive at the system with monotonic increasing timestamps and the order of every tuple per timestamp is random.

Static relations R_j $(1 \leq j \leq k)$ consist of a finite bag of tuples having the same schema (Definition 2). The domain of each attribute is the set of integers. R_j does not change over time while processing a query (which makes R_j *static*). Every attribute A of a static relation is lower-bounded by $\min\{r_j[A]\}$, upper-bounded by $\max\{r_j[A]\}$, and therefore always bounded.

All tuples of a stream having the same timestamp are received and cached until a tuple with a new timestamp arrives at the system. A marker denotes the arrival of such a tuple that has a timestamp different from those timestamps of the cached tuples. The query is executed as soon as a marker is received, results are written into the output, and the cache is emptied. Figure 1 visualizes two synchronized streams S, T, where a marker denotes the arrival of a new timestamp.

Example 1. A DSMS receives two synchronized and ordered streams S, T of temperature values produced by temperature sensors. The streams are synchronized, because the arrival of a new marker denotes the arrival of a tuple with a timestamp that is different from the timestamps of the cached tuples, and tuples arrive in order with respect to their timestamp. In this example, the tuples $S(-1°\mathrm{C}, 0s)$ and $T(-2°\mathrm{C}, 0s)$ arrive at the DSMS and are written into a cache. A marker triggers the re-evaluation of a query that is registered at the system. The cache is emptied and new tuples arrive at the system that have another timestamp than the already processed tuples. All tuples that arrive at the DSMS between two markers fit into memory, as the amount of tuples between two markers is bounded by a constant.

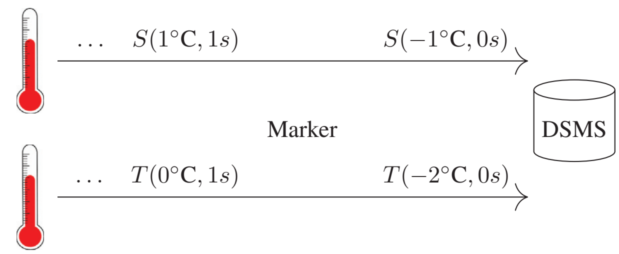

Figure 1: Execution Model

Preprocessing of Queries

Our criteria are defined for queries in a special form. Due to this, queries that consist of SPJ queries combined by a polyadic union operator need to be preprocessed in four steps in such a way that criteria testing whether the query is bm-computable can be applied:

1. Split each SPJ query, with selection containing inequality join predicates of the form $(S_i.A \neq S_j.B)$ with $i \neq j$, into multiple SPJ queries combined by a polyadic union operator, until no SPJ query contains a selection with a disjunction. Then, a SPJ query has $(S_i.A > S_j.B)$ and another one $(S_i.A < S_j.B)$ in its selection.
2. Join streams S_i with timestamp attribute I and S_j with timestamp attribute J into a single stream S with fresh timestamp K, if both are part of a SPJ query, where $(I = J)$ is in the selection P^+. The fresh schema of stream S contains every attribute from the streams S_i, S_j from the domain of integers and attribute K with $K = I = J$ from the domain of timestamps.
3. Rename an infinite stream S_i with timestamp attribute I into finite stream F_i if I is upper-bounded in the selection of a SPJ query.
4. Derive a set of locally totally ordered (LTO) queries from each SPJ query and combine them by a polyadic union operator. A LTO query is derived from a SPJ query Q by adding filter predicates to the selection P of Q. Adding different filter predicates to P of Q results in a finite number of different LTO queries. Whether a query is a LTO query or not, depends on the transitive closure P^+ of selection P. The transitive closure P^+ is the set of all atomic predicates that can be logically inferred by the predicates in P, involving only elements of P (Arasu et al. 2004).

For every input stream S_i of a LTO query, the set of integer attributes of S_i and constants contained in the query are totally ordered (Definition 6).

Definition 6 (Definition 4.3 in (Arasu et al. 2004))**.** A set of elements E (attributes, integers, constants) 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^+.

A preprocessed query is here referred to as a modified LTO query (Definition 7).

Definition 7. A *modified LTO query* has the following properties:

1. The selection does not contain an inequality join of the form $(S_i.A \neq S_j.B)$.
2. Does not refer to any two streams S_i with timestamp attribute I and S_j with timestamp attribute J together with an equality join predicate of the form $(S_i.I = S_j.J) \in P^+$, with $i \neq j$.
3. Does not refer to a stream S_i where the timestamp attribute is upper-bounded.

4. Each set of integer attributes of every SPJ query is totally ordered.

Set $\mathcal{A}(S)$ contains all attributes that are in the schema of stream S, and set $\mathcal{S}(Q)$ contains all input streams of query Q. A dependency graph $\mathcal{G}(Q) = (V, E)$ induced by a modified LTO query Q has the vertices $V = S(Q)$ and edges $E = \{(S_i, S_j) \mid S_i, S_j \in V \wedge (S_i.I > S_j.J) \in P^+ \wedge I, J$ are timestamp attributes$\}$. For each stream $S \in S(Q)$, graph $\widetilde{\mathcal{G}}(S, Q)$ is the connected component in $\mathcal{G}(Q)$, in that stream S is contained. Graph $\mathcal{G}(S, Q)$ is the spanning tree of $\widetilde{\mathcal{G}}(S, Q)$. If $\mathcal{G}(S, Q)$ forms a tree, then $d(\mathcal{G}(S, Q))$ denotes the distance of stream S to the root node of a tree. The children of a parent node in a tree are those with a distance of one to the parent node.

Our criteria depend on the potential redundancy of inequality predicates (Definition 8), and two sets *MaxRef*, *MinRef* (Definition 9).

Definition 8 (Definition 4.1 in (Arasu et al. 2004))**.** An inequality predicate $(e_1 < e_2) \in P$ is said to be *redundant* in P iff 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 an integer constant k such that $(e_1 = k) \in P^+$ and $(k < e_2) \in P^+$; (3) there exists an integer constant k such that $(e_1 < k) \in P^+$ and $(e_2 = k) \in P^+$.

Definition 9 (Definition 5.4 in (Arasu et al. 2004))**.** *MaxRef*(S_i) is the set of all unbounded integer attributes A of S_i that participate in a non-redundant inequality join $(S_j.B < S_i.A)$, with $i \neq j$, in P^+ and MinRef(S_i) is the set of all unbounded integer attributes A of S_i that participate in a non-redundant inequality join $(S_i.A < S_j.B)$, with $i \neq j$, in P^+.

Example Queries

In this section, we illustrate (the proofs for) our criteria for bounded-memory computability with an extensive example. The example query Q has input streams $S(A, I)$, $T(B, J)$, and $U(C, K)$, where A, B, and C are integer attributes, and I, J, and K are timestamp attributes:

$$\Pi_{A,B}(\sigma_{(I>J)\wedge(J>K)\wedge(A>B)\wedge(0<B)\wedge(B<5)}(S \times T \times U))$$

The query is bm-computable in the duplicate preserving, but not in the duplicate eliminating case. In the duplicate preserving case, synopses $Syn(t)$, $Syn(u)$ are created for streams T and U. A synopsis, such as $Syn(t)$ contains for the current instance t of stream T tuples that "represent" t so that $Q(t) = Q(Syn(t))$. In case of duplicate preserving queries, synopsis $Syn(t)$ contains two sets s_n^T, s_p^T. Set s_n^T in $Syn(t)$ of stream T contains values with respect to the current time step, and s_p^T in $Syn(t)$ of stream T values that were inserted into s_n^T in the past. Sets s_n^T and s_n^U contain only values with respect to the current time step and the size of them is always finite as only a bounded amount of tuples arrive at a DSMS at each time step. Set s_p^T is always finite as only distinct values of bounded attribute B between 0 and 5 are stored in the synopsis and set s_p^U only contains an empty tuple with a counter as presented later. Therefore, the size of $Syn(t)$ and $Syn(u)$ is bounded by a constant.

Assume tuples from streams S, T, and U arrive at a DSMS as visualized in Figure 2. The arrows denote that $(I > J) \in P$ or $(J > K) \in P$ are satisfied. At time step $0s$, the tuples $(42, 0s)$ and $(7, 0s)$ arrive at the DSMS. Tuple $(42, 0s)$ is discarded as $(I > J) \in P$ can never be satisfied. All tuples arrive with an increasing timestamp value and no tuple with a timestamp value less than $0s$ was ever received in the past or will be received in the future. The same holds for tuple $(7, 0s)$ with respect to $(J > K) \in P$ and additionally $(B < 5) \in P$ is not satisfied. In the next time step, tuple $(1, 1s)$ arrives at the DSMS and is not discarded, as $(J > K) \in P$ is possibly satisfied in the future. Value 1 is not stored in memory, as attribute C is not in the project list L or part of a predicate in selection P of Q. Instead, an empty value $()$ is stored in s_n^U together with a counter $\langle 1 \rangle$. Conceptually, $()\langle 1 \rangle \in s_n^U$ denotes that a tuple was received from stream U at the current time step. In the next time step, $()\langle 1 \rangle$ is moved from set s_n^U into s_p^U. Now, $()\langle 1 \rangle \in s_p^U$ denotes that a tuple was received from stream U in the past. Tuple $(2, 2s)$ is not discarded, as $(J > K) \in P$ and $\{(0 < B), (B < 5)\} \subseteq P$ are satisfied, and $(I > J) \in P$ is possibly satisfied in the future. Value 2 is stored in memory, as B is in the project list L and $()\langle 1 \rangle \in s_p^U$ denotes, that $(2, 2s)$ was received after exactly one tuple (here $(1, 1s)$) from stream U in the past. Therefore, $(2)\langle 1 \rangle$ is stored in s_n^T and moved into set s_p^T at the next time step. Counter $\langle 1 \rangle$ of value 2 denotes, that $(2, 2s)$ was received exactly once after a tuple was received from stream U in the past. In the next time step, tuple $(3, 3s)$ arrives at the DSMS and again, an empty value together with counter $\langle 1 \rangle$ is stored in s_n^U. At time step $4s$, $()\langle 1 \rangle \in s_n^U$ is moved into set $s_p^U = \{()\langle 1 \rangle\}$ and then merged by adding the counters of the empty values. Element $()\langle 2 \rangle \in s_p^U$ denotes that two tuples where received from stream U in the past. Tuple $(1, 4s)$ arrives at the DSMS from stream T and $(1)\langle 2 \rangle$ is stored in s_n^T denoting that $(1, 4s)$ was received exactly two times after two tuples where received from stream U in the past. In the next time step, $(1)\langle 2 \rangle$ is moved from set s_n^T into s_p^T and tuples $(42, 5s)$ and $(3, 5s)$ arrive at the DSMS. Value 3 is stored together with counter $\langle 2 \rangle$ in s_n^T as described before and value 42 is not stored in memory as synopsis $Syn(s)$ does not exist. All tuples received from stream S are only needed at the current time step to compute results and never in the future. Therefore, a synopsis $Syn(s)$ is unnecessary. Tuple $(42, 5s)$ from stream S is received after tuple $(2, 2s)$ and $(2, 2s)$ after $(1, 1s)$ as depicted in Figure 2. Thus, $(42, 2)$ needs to be written into the output stream, as attributes A and B are in the project list L. However, the tuples $(42, 5s), (2, 2s), (1, 1s)$ are not stored in memory and results need to be computed from the values stored in the synopsis $Syn(t)$. As depicted in Figure 2, $(2)\langle 1 \rangle$ is stored in s_p^T denoting, that value 2 was received from stream T in the past after exactly a single tuple was received from stream U in the past. Therefore, $(42, 2)$ can be derived as a result from synopsis $Syn(t)$. Additionally, $(42, 5s)$ was received after $(1, 4s)$ from stream T and $(1, 4s)$

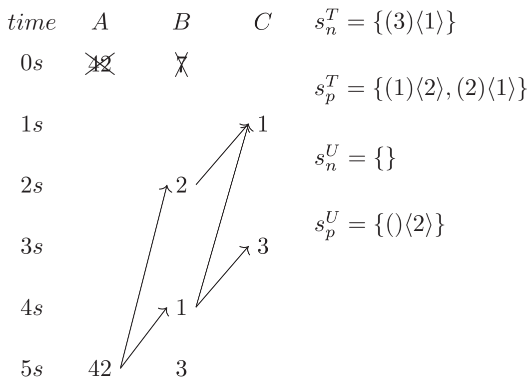

Figure 2: Example instances of streams S, T, and U

after $(3, 3s)$ and $(1, 1s)$ from stream U. Thus, $(42, 1)$ needs to be written twice into the output stream. Again, this result can be derived from $(1)\langle 2\rangle \in s_p^T$ as before.

Query Q is not bm-computable in the duplicate eliminating case as attribute A is in the project list L. Any evaluation strategy that evaluates Q has to check whether a tuple was already written into the output stream. Therefore, an evaluation strategy has to keep track of every distinct tuple that was written into the output stream. However, attribute A is not bounded and arbitrary many distinct tuples of stream S might arrive at the system, where $Syn(t)$ and $Syn(u)$ are not empty. An evaluation strategy, that necessarily has to keep track of every distinct value of attribute A, stores an unbounded amount of tuples in memory. Therefore, Q is not bm-computable in the duplicate eliminating case.

Duplicate Preserving Queries

This section presents in Theorem 1 a sufficient and necessary criterion for bounded-memory computability of duplicate-preserving modified LTO queries (Definition 7).

Theorem 1. *Let*

$$\dot{\pi}_L \left(\bigtimes_{i=1}^{m} \{\langle a_i \rangle\} \times \sigma_P \left(\bigtimes_{i=1}^{k} R_i \times \bigtimes_{i=1}^{l} S_i \times \bigtimes_{i=1}^{p} F_i \right) \right)$$

be a modified LTO query Q, $k, p \geq 0$, and $l > 1$. Q is bm-computable iff all of the following conditions are fulfilled:

C1: For every stream S_i, the graph $\mathcal{G}(S_i, Q)$ forms a tree, with $i = 1, \dots, l$.

C2: Graph $\mathcal{G}(S_i, Q)$ forms a tree and for every integer join of the form $(S_i.A$ op $S_j.B)$, with $i \neq j$ and op ranges over $\{<, =, >\}$, if $\mathcal{G}(S_i, Q) = \mathcal{G}(S_j, Q)$, then S_i, S_j have either the same stream as parent, S_i is the parent of S_j, or S_j is the parent of S_i, else if $\mathcal{G}(S_i, Q) \neq \mathcal{G}(S_j, Q)$, then $d(\mathcal{G}(S_i, Q)) = d(\mathcal{G}(S_j, Q)) = 0$.

C3: Graph $\mathcal{G}(S_i, Q)$ forms a tree and for every integer attribute $A \in \mathcal{A}(S_i)$ in project list L, with $i = 1, \dots, l$, $d(\mathcal{G}(S_i, Q)) \leq 1$, and if $|\{\mathcal{G}(S_i, Q) \mid 1 \leq i \leq l\}| > 1$, then A is bounded, else if $|\{\mathcal{G}(S_i, Q) \mid 1 \leq i \leq l\}| = 1$ and $d(\mathcal{G}(S_i, Q)) = 1$, then A is bounded.

C4: Graph $\mathcal{G}(S_i, Q)$ forms a tree and for every integer equality join predicate $(S_i.A = S_j.B)$, with $i \neq j$, $S_i.A$ and $S_j.B$ are both bounded, except for $|\{\mathcal{G}(S_i, Q) \mid 1 \leq i \leq l\}| = 1$ and either $d(\mathcal{G}(S_i, Q)) = 0$ or $d(\mathcal{G}(S_j, Q)) = 0$. If $|\{\mathcal{G}(S_i, Q) \mid 1 \leq i \leq l\}| = 1$ and $d(\mathcal{G}(S_i, Q)) = 0$ then $S_j.B$ is bounded and $d(\mathcal{G}(S_j, Q)) = 1$. If $|\{\mathcal{G}(S_i, Q) \mid 1 \leq i \leq l\}| = 1$ and $d(\mathcal{G}(S_j, Q)) = 0$ then $S_i.A$ is bounded and $d(\mathcal{G}(S_i, Q)) = 1$.

C5: Graph $\mathcal{G}(S_i, Q)$ forms a tree and $|MaxRef(S_i)| + |MinRef(S_i)| = 0$, with $i = 1, \dots, l$, except for $|\{\mathcal{G}(S_i, Q) \mid 1 \leq i \leq l\}| = 1$ and $d(\mathcal{G}(S_i, Q)) = 0$.

We shortly motivate each of the conditions in the criterion. As $\mathcal{G}(S_i, Q)$ forms a tree by *C1*, counters in $s_n^{S_i}$ of parent streams S_i can be updated depending on child streams S_j counters in $s_p^{S_j}$ each time a tuple from stream S_i is received. By condition *C2*, an inequality join of the form $(S_i.A$ op $S_j.B)$ is only allowed if S_i, S_j have either the same stream as parent, S_i is the parent of S_j or S_j is the parent of S_i, where op ranges over $\{\leq, =, \geq\}$. This ensures that each update of a counter can be computed with respect to any join between the attributes of S_i and S_j or child streams S_j of S_i at the current time step where it can be guaranteed that all values in $s_p^{S_j}$ of child streams S_j where received in the past with respect to the values in $s_n^{S_i}$ of the parent stream S_i of child streams S_j.

At each time step, a counter denotes how often values of an attribute A in project list L need to be written into the output stream. However, attributes are only written into the output stream if all $s_n^{S_i}$ of streams S_i that are in the root of $\mathcal{G}(S_i, Q)$ are not empty and at least one tuple is received from one of those streams S_i at the current time step. Then, only the counters in $s_n^{S_i}$ of streams S_i in the root of $\mathcal{G}(S_i, Q)$ or counters in $s_p^{S_i}$ of streams S_j that are a child of streams S_i in $\mathcal{G}(S_i, Q)$ are up to date and therefore, only attributes $A \in A(S_i) \cup A(S_j)$ are by condition *C3* in project list L.

Conditions *C4* and *C5* ensure that every attribute that influences the output of Q is bounded. If an attribute A influences the output of Q, then all values of attribute A need to be stored in memory. However, storing all values ever received would require an unbounded amount of memory. Therefore, only distinct values together with a counter are stored in memory which requires only a bounded amount of memory if attribute A is bounded.

Duplicate Eliminating Queries

This section presents in Theorem 2 a sufficient criterion for bounded-memory computability of duplicate eliminating modified LTO queries (Definition 7).

Theorem 2. *Let*

$$\pi_L \left(\bigtimes_{i=1}^{m} \{\langle a_i \rangle\} \times \sigma_P \left(\bigtimes_{i=1}^{k} R_i \times \bigtimes_{i=1}^{l} S_i \times \bigtimes_{i=1}^{p} F_i \right) \right)$$

be a modified LTO query, $k, p \geq 0$, and $l \geq 1$. Q is bm-computable if all of the following conditions are fulfilled:

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

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

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

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

The criterion for queries with a duplicate eliminating operator is less restrictive than for queries with a duplicate preserving operator as counters are unnecessary. However, all attributes influencing the output of Q need to be bounded to keep track of all tuples that where already written into the output stream. That is necessary to prevent duplicates being written into the output stream. Graph $\mathcal{G}(S_i, Q)$ does not have to form a tree as no counters need to be updated.

Related Work

Our criteria on bounded-memory computability are motivated by those of Arasu and colleagues (Arasu et al. 2004). The main difference is that Arasu and colleagues provide general criteria for a subclass of SQL queries over infinite streams without considering additional constraints on specific attributes such as that of a time attribute. Our results show that additional constraints such as that of having a time domain with a linear order and a starting time point have a strong influence on bounded-memory computability.

Pushing the idea of constraints on streams further leads to considering constraints specified in a knowledge base / ontology. In this respect, our work is related to (in fact, motivated by) stream processing within the OBDA paradigm (Calbimonte et al. 2012) or stream processing w.r.t. Datalog knowledge bases (Beck, Dao-Tran, and Eiter 2018; Walega, Kaminski, and Grau 2019). In particular we considered queries resulting from the transformation (Schiff, Özçep, and Möller 2018) of OBDA queries in STARQL (Özçep, Möller, and Neuenstadt 2014).

Bounded-memory computability is a general feature not restricted to the realm of streams. Indeed, historically, it has been considered in the first place in the realm of temporal databases where the focus is on finding bounded-history encodings in order to check temporal integrity constraints—as described in the classic paper of Chomicki (Chomicki 1995). Moreover, under the term “incremental maintainability” a generalized form of bounded memory processing is discussed in dynamic complexity (Patnaik and Immerman 1997). The aim is to solve problems that are not captured by some logic L (for example, calculating the transitive closure of a graph is not definable as a first order logic formula) by allowing an incremental update of formulas in L. The underlying incremental update model extends the idea of updating the values in registers which underlies our execution model. An early description for a stream execution model over first-order logic structures are stream abstract state machines (Gurevich, Leinders, and Van Den Bussche 2007).

Conclusion and Future Work

We made a step towards coping with the infiniteness of streams by finding criteria for testing whether a SPJ query over streams with application time and static relations can be evaluated in constant space. Our model is sufficiently general in order to capture realistic scenarios, as those described in (Schiff, Özçep, and Möller 2018), with non-trivial criteria for bounded-memory computability. Though non-trivial, those criteria are easy to check so that queries computable in constant space can be identified. Concerning the generality of our approach we note further that the domain of attributes is not restricted to integers but can be any discrete structure (cf. (Arasu et al. 2004, chp. 9)).

We currently work on extending criteria for queries with optional negation. Queries with negation are not monotonic and therefore a new execution model is required where tuples are not only appended to the instance of the output stream. Additionally it would be interesting to find criteria for queries that allow the comparison of timestamp attributes with non-timestamp attributes.

References

Abiteboul, S.; Hull, R.; and Vianu, V., eds. 1995. *Foundations of Databases*. Addison-Wesley.

Arasu, A.; Babcock, B.; Babu, S.; McAlister, J.; and Widom, J. 2004. Characterizing memory requirements for queries over continuous data streams. *ACM Transactions on Database Systems (TODS)* 29(1):162–194.

Baader, F.; Borgwardt, S.; and Lippmann, M. 2013. Temporalizing ontology-based data access. In *International Conference on Automated Deduction*, 330–344. Springer.

Beck, H.; Dao-Tran, M.; and Eiter, T. 2018. Lars: A logic-based framework for analytic reasoning over streams. *Artificial Intelligence* 261:16–70.

Calbimonte, J.-P.; Jeung, H.; Corcho, O.; and Aberer, K. 2012. Enabling query technologies for the semantic sensor web. *Int. J. Semant. Web Inf. Syst.* 8(1):43–63.

Chomicki, J. 1995. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.* 20(2):149–186.

Gurevich, Y.; Leinders, D.; and Van Den Bussche, J. 2007. A theory of stream queries. In *Proceedings of the 11th International Conference on Database Programming Languages*, DBPL'07, 153–168. Berlin, Heidelberg: Springer-Verlag.

Nikolaou, C.; Kostylev, E. V.; Konstantinidis, G.; Kaminski, M.; Grau, B. C.; and Horrocks, I. 2017. The bag semantics of ontology-based data access. *arXiv preprint arXiv:1705.07105*.

Özçep, Ö. L.; Möller, R.; and Neuenstadt, C. 2014. A stream-temporal query language for ontology based data access. In *KI 2014*, volume 8736 of *LNCS*, 183–194. Springer.

Patnaik, S., and Immerman, N. 1997. Dyn-fo: A parallel, dynamic complexity class. *Journal of Computer and System Sciences* 55(2):199–209.

Schiff, S.; Özçep, Ö. L.; and Möller, R. 2018. Ontology-based data access to big data. *Open Journal of Databases (OJDB)* 6:21–32. Postproceeding of Hidest'18.

Walega, P. A.; Kaminski, M.; and Grau, B. C. 2019. Reasoning over streaming data in metric temporal datalog. In *AAAI-19*.