Optimizing Away Joins on Data Streams

Lukasz Golab
AT&T Labs – Research
lgolab@research.att.com

Theodore Johnson
AT&T Labs – Research
johnsont@research.att.com

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

Divesh Srivastava
AT&T Labs – Research
divesh@research.att.com

David Toman
University of Waterloo
david@cs.uwaterloo.ca

ABSTRACT

Monitoring aggregates on network traffic streams is a compelling application of data stream management systems. Often, streaming aggregation queries involve joining multiple inputs (e.g., client requests and server responses) using temporal join conditions (e.g., within 5 seconds), followed by computation of aggregates (e.g., COUNT) over temporal windows (e.g., every 5 minutes). These types of queries help identify malfunctioning servers (missing responses), malicious clients (bursts of requests during a denial-of-service attack), or improperly configured protocols (short timeout intervals causing many retransmissions). However, while such query expression is natural, its evaluation over massive data streams is inefficient.

In this paper, we develop rewriting techniques for streaming aggregation queries that join multiple inputs. Our techniques identify conditions under which expensive joins can be optimized away, while providing error bounds for the results of the rewritten queries. The basis of the optimization is a powerful but decidable theory in which constraints over data streams can be formulated. We show the efficiency and accuracy of our solutions via experimental evaluation on real-life IP network data using the AT&T Gigascope stream processing engine.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*Temporal logic*

General Terms

Theory, Algorithms, Performance

Keywords

Data stream integrity constraints, Data stream query rewriting, Data stream joins

1. INTRODUCTION

Network traffic monitoring is a compelling application of data stream management systems (DSMSs). For instance, the Gigascope DSMS has been developed at AT&T Labs and is now operationally used within AT&T's IP backbone [10, 17]. Applications of Gigascope include traffic analysis, performance monitoring, troubleshooting, and detection of network attacks.

We have examined a set of monitoring queries written by network analysts and found that many such queries compute aggregates that summarize various properties of the underlying packet stream. In particular, a common type of streaming aggregation requires joining and correlating multiple streams (or substreams of the same packet stream), corresponding to requests and responses or transmissions and acknowledgements. Representative examples include:

1. For every 5-minute interval, report the number of DNS requests in that interval (i.e., packets whose destination port equals 53) that do not have a matching response packet from the server within 5 seconds (i.e., one whose source port is 53, and destination IP address and port are equal to the source IP address and port of the request).

2. For every 5-minute interval, report the number of TCP SYN packets in that interval (i.e., requests for connection) that do not have a matching SYN-ACK packet within 5 seconds (i.e., responses from the server that establish the connections, such that the source IP address and port of the request equal the destination IP address and port, respectively, of the response, and vice versa).

3. For every 5-minute interval, report the number of SYN and SYN-ACK pairs in that interval (denoting establishment of a TCP connection) that do not have a matching FIN packet within 10 seconds (denoting connection teardown).

Queries 1 and 2 help identify malfunctioning servers (missing responses) and improperly configured protocols (requests time out too quickly and are retransmitted before the responses arrive). Query 3 helps detect "SYN flood" denial-of-service attacks, where malicious clients attempt to cripple a server with millions of SYN packets; once the connections are accepted and valuable resources are allocated by the server, the clients never continue or close the bogus connections.

The above queries may be expressed as joins on the IP address, port, and timestamp (e.g., within 5 or 10 seconds), followed by computation of aggregates over non-overlapping temporal windows (e.g., every 5 minutes). While such query expression is natural, its exact evaluation would require the DSMS to (a) store every request packet until it finds a matching response packet or the temporal join condition has elapsed, whichever comes first, and (b) match every response packet with a previous request packet that satisfies the temporal join condition. This is inefficient in practice, both from the storage and computational perspectives, especially over high-speed IP traffic streams with many connections among many different IP addresses/ports.

One way to improve performance is by sampling the streams and returning approximate query answers. In this paper, we develop query rewrites that are more efficient and accurate than sampling, whereby expensive joins are completely *optimized away*. For instance, our technique rewrites Query 1 from above as:

> *For every 5-minute interval, report the difference between the number of DNS requests and responses in that interval.*

Thus, rather than storing and correlating individual requests with matching responses, the rewritten query simply maintains two independent frequency counters. Of course, this optimized query is not necessarily equivalent to the original query—the reported differences may be lower or higher than the actual number of requests in that interval without matching responses. Intuitively, this error cannot be large under reasonable constraints on the arrival pattern of requests and responses.

Our work is also useful for the case when the user directly expresses the desired correlation by independently counting related events, as has been employed by network analysts in an ad-hoc fashion [20, 29]. In this case, our results formally identify the stream constraints that need to hold for the user's query expression to be meaningful.

Our contributions in this paper are as follows.

1. We define a powerful but decidable theory in which constraints over data streams can be formulated. The proposed theory is more powerful than temporal integrity constraints studied so far; for a more detailed discussion see Sections 3 and 6.

2. Based upon the above theory, we identify conditions under which joins can be eliminated from streaming aggregation queries. We also derive error bounds for the results of the rewritten queries.

3. Using a real-life IP packet stream and DSMS (Gigascope), we present experimental evidence of the efficiency and accuracy of our rewritings, as compared to joins over appropriately sampled streams. In particular, we show that to match the efficiency of the rewritten query, a sampling rate of less than 1% must be used for the original Query 3 above, but a sampling rate of 10% is already less accurate than the rewritten query.

The remainder of this paper is organized as follows. Section 2 gives a detailed motivating example. Sections 3 and 4 present the details behind the proposed stream integrity constraints and query transformations, respectively. Section 5 provides experimental results. Related work is presented in Section 6. Section 7 concludes the paper.

2. MOTIVATING EXAMPLE

We model data streams as relations with a fixed schema, in which tuples are timestamped according to their arrival times. Building upon Query 3 from the introduction, we consider three sub-streams of a TCP packet stream: SYN packets sent by clients that originate TCP connections, SYN-ACK packets sent by servers that acknowledge the original SYN packets, and FIN packets from clients or servers that terminate the connections[1]. Similarly, we can define DNS request and response sub-streams for Query 1. We represent the TCP sub-streams using three relational schemes, SYN, SYN-ACK and FIN, with attributes `ip` standing for *IP addresses* (for simplicity of exposition, we encapsulate the parts representing the source and destination addresses, ports, etc., in a single attribute), and `time` denoting the *timestamp* (the time of arrival of the tuple, in seconds, in the respective stream).

We would like to answer the query:

> *For each* 5*-minute interval, how many* SYN *packets in that interval have a matching* SYN-ACK *packet within* 5 *seconds, but do not have a matching* FIN *packet within* 10 *seconds?*

This query may be formulated (in SQL syntax) over the above schemes as follows:

```
SELECT tb, count(*) as cnt
FROM   SYN s, SYN-ACK sa
WHERE  s.ip = sa.ip
  AND  sa.time >= s.time
  AND  sa.time − s.time <= 5
  AND  NOT EXISTS
       ( SELECT *
         FROM   FIN f
         WHERE  sa.ip = f.ip
           AND  f.time >= sa.time
           AND  f.time − sa.time <= 10 )
GROUP BY s.time/300 as tb
```

Note that the above query requires expensive join and anti-join operations over the three streams. While efficient algorithms for data stream joins have been proposed [13, 19, 28], at high streaming speeds, such joins become infeasible.

In this paper, we pursue a more indirect approach to evaluating the above query, expressed as follows:

```
SELECT s.tb, min(s.cnt,sa.cnt)−f.cnt as cnt
FROM   ( SELECT   tb, count(*) as cnt
         FROM     SYN
         GROUP BY time/300 as tb ) s,
       ( SELECT   tb, count(*) as cnt
         FROM     SYN-ACK
         GROUP BY time/300 as tb ) sa
       ( SELECT   tb, count(*) as cnt
         FROM     FIN
         GROUP BY time/300 as tb ) f
WHERE  s.tb = sa.tb AND sa.tb = f.tb
  AND  min(s.cnt,sa.cnt) − f.cnt > 0
```

The rewritten query computes an arithmetic expression over independent counts of SYN, SYN-ACK and FIN packets in a 5-minute interval, provided this difference is positive. This query completely eliminates the join operation between individual SYN and SYN-ACK packets, and the (anti-)join operation between the result of the previous join operation and the FIN packets. The three counts can be computed on the fly, and the WHERE condition that equates the `tb` attributes merely states that every 5 minutes the

[1]Technically, a TCP connection may also be terminated by a RST (reset) packet. For simplicity, we assume that RST packets are part of the FIN sub-stream.

arithmetic expression over the counters is computed and the counters are reset.

However, in general, the two queries are not necessarily equivalent (or even close). Thus, we study *integrity constraints* that, when satisfied by data streams, make the above transformation possible. We use the following constraints, which can be obtained from TCP specifications [15] or using mining techniques [21].

- In the SYN, SYN-ACK and FIN streams, the `ip` attribute can serve as an identifier of a TCP connection (a key) for the duration of a connection. But, it is not a key in general as there may be multiple connections between a particular source-destination pair of IP addresses over time.

- For every normal TCP connection, there is a single SYN, a single SYN-ACK, and a single FIN (or RST) packet. However, in abnormal circumstances (e.g., during a SYN flood attack), some packets may be missing (e.g., FINs never produced by malicious clients, or fewer SYN-ACKs generated by the attacked server, which is unable to grant every connection request).

- The SYN, SYN-ACK, and FIN packets belonging to the same TCP flow are temporally co-located in the three substreams, with SYNs and SYN-ACKs appearing at most 5 seconds apart, and SYN-ACKs and FINs at most 10 seconds apart.

These constraints must have been known (at least intuitively) to the user when formulating the *original* query, in particular, when specifying maximum time intervals between matching SYN, SYN-ACK, and FIN packets. Also, note that the above constraints hold only approximately. In our example, this is mainly due to network latencies (late SYN-ACKs) and a small percentage of long-lasting TCP flows (late FINs). There are two ways to solve this problem:

- use a more complex but precise specification that accounts for the deviations, or

- use more intuitive constraints that are satisfied by most of the stream.

While the first solution may seem preferable from the theoretical point of view, the complexity of developing comprehensive descriptions that account for all possible deviations is prohibitive and the computational properties of such theories are often quite poor.

Continuing with our example, the errors induced by the rewritten query can be traced to two main sources:

1. **Boundary effects**: incurred by dividing the time line into 5-minute buckets—the (SYN, SYN-ACK) and (SYN-ACK, FIN) pairs that *cross* bucket boundaries are not accounted for. This error cannot be completely avoided by, e.g., shifting the 5 minute window for SYN-ACK and FIN packets by a few seconds, as we would incur the same error for the matching packets that arrive closer together.

2. **Approximate satisfaction of integrity constraints**: incurred, e.g., by asserting that all FIN packets arrive within 10 seconds of the corresponding SYN-ACK packets. If some FINs arrive later, but still in the same 5-minute bucket, then the late FINs will not be accounted for in the original query, but would in the rewritten query.

Note also that these errors cannot be eliminated altogether as the `ip` attribute serves as a TCP flow identifier for a limited period of time. Thus, on noisy networks, the exact accounting of lost/superfluous packets is not possible, save reproducing the whole TCP state machine [15] in the query.

The motivation behind this paper originates from Internet protocols that exhibit behaviors of the form: request-response, transmission-acknowledgement, or initiation-establishment-teardown. However, our solution is applicable in other situations where properties of a single (conceptual) entity are monitored on multiple data streams at different time instants. For instance, we can watch the *two-phase commit* (2PC) protocol for irregularities, e.g., attempts to commit transactions for which one of the participants did not vote *yes*.

We introduce stream integrity constraints next, and then describe how these are used in our query rewrites in Section 4.

3. STREAM INTEGRITY CONSTRAINTS

It is often the case that *high-level entities*, such as TCP connections, are decomposed into multiple data items (packets) before they can be transmitted over (possibly multiple) data streams. The decomposition and reassembly are governed by a transmission protocol.[2] For the purposes of stream query optimization, we extract the relevant constraints governing the arrival of data items in the data streams and represent them using *stream integrity constraints*. These constraints are defined by extending the standard SQL DDL for data streams as follows.

First, each data stream must have a *distinguished attribute* `time`, of type `TIMESTAMP`, that captures the arrival time of each data item. Second, to represent higher-level entities, we introduce *virtual attributes*, of type `VIRTUAL`, that capture the identity of these entities (e.g., TCP connections). The combination of timestamps and virtual attributes allows us to conveniently specify *stream integrity constraints* in terms of higher-level conceptual objects. Note that virtual attributes are solely used in the integrity constraints and neither the original nor the final optimized queries actually refer to them.

The *stream integrity constraints* themselves form a part of the *declaration of a stream* (`DEFINE STREAM` S). The constraints are specified using the following DDL clauses.

The Stream key clause. A *stream key* captures the idea that a certain attribute (or set of attributes) identifies higher-level concepts *across* several data streams in a particular time window. The syntax is an extension of the SQL DDL key specification:

`STREAM KEY` $(a_1, \ldots, a_k)$ `SPANS` S' `WINDOW` $[t_s, t_e]$

The constraint states that $(a_1, \ldots, a_k)$ uniquely identifies items in both streams S and S' in the window $[t_s, t_e]$ that is *relative* to the S item; for this declaration to be valid, both S and S' have to have a common signature.

The Foreign stream key clause. A *foreign key* is a simple extension of the SQL DDL constraint with a windowing construct:

`FOREIGN KEY` $(a_1, \ldots, a_k)$
    `REFERENCES` $S'(b_1, \ldots, b_k)$ `WINDOW` $[t_s, t_e]$

This constraint states that the $(a_1, \ldots, a_k)$ tuple in S must appear in S' as a $(b_1, \ldots, b_k)$ tuple in the $[t_s, t_e]$ window.

[2]In the case of network protocols, a state machine is commonly used for this purpose. The protocol specification itself can be thought of as a set of *integrity constraints*.

The Co-occurrence clause. Last, we can specify that items arriving in separate streams and satisfying certain conditions *can only appear* (co-occur) within a window:

```
COOCCURS (a_1,...,a_k)
         AND S'(b_1,...,b_k) WINDOW [t_s,t_e]
```

All the *window* parts of the declarations are always relative to the timestamp of the item in S.

Example 1 In our running example, the Stream DDL declaration for the SYN-ACK stream is as follows:

```
DEFINE STREAM SYNACK (
  time TIMESTAMP,
  id   VIRTUAL,
  ip   ADDRESS,
  PRIMARY KEY (id),
  STREAM KEY (id) SPANS FIN WINDOW [0,*],
  STREAM KEY (ip) SPANS FIN WINDOW [0,15],
  FOREIGN KEY (id,ip) REFERENCES SYN(id,ip)
                     WINDOW [-5,0]
)
```

We use * to denote unbounded windows. Note that the PRIMARY KEY declaration and its meaning are identical to the standard SQL DDL. In our example, the virtual attribute id identifies a TCP flow, and therefore is a primary key and stream key across all three sub-streams. On the other hand, ip is a stream key across the three sub-streams only for the duration of a single flow (within a 15-second window that starts when a SYN packet arrives). Figure 3 in Appendix A lists all the constraints in the SYN, SYN-ACK, and FIN streams.

The stream constraints are, however, not used for checking (constraint enforcement), but for *reasoning* about equivalence of stream queries. To this end, we need to define the notion of *constraint inference*.

Example 2 In our example, assuming that id is a key in the SYN stream as well, the co-occurrence constraint[3]

```
DEFINE STREAM SYNACK (
  COOCCURS (id) AND SYN(id) WINDOW [-5,0]
)
```

is a logical consequence of the explicit stream constraints (namely those which state that id is a primary key, stream key, and foreign key within a five-second window) and thus can be used for query optimization.

Existing theories on temporal integrity constraints do not allow constraints expressive enough to capture the properties of data streams needed to enable the desired query rewrites. The stream integrity constraints defined in this section are based on a novel underlying constraint theory developed in this paper. The proposed theory overcomes the shortcomings of existing temporal constraint theories, in particular their inability of expressing powerful equational constraints needed to capture *stream keys* and other *window-based* stream constraints. Our solution is based on a combination of a powerful logic for linear time, S1S [6], and generalized full dependencies, both tuple- and equality-generating [7]. A major contribution of this paper is showing that the combined theory is still decidable and that the complexity of the logical implication problem is, under reasonable assumptions, comparable to other constraints.

[3]We slightly abuse the DDL syntax here to be able to talk about *individual* constraints holding on a stream rather than about the complete specification for that stream.

Appendix A gives a formal proof for the following theorem, including the appropriate complexity bounds:

Theorem 3 *The theory of stream integrity constraints is decidable for full* FOREIGN KEY *constraints. The complexity of reasoning is identical to the standard relational case.*

4. QUERY TRANSFORMATIONS

We consider queries based on applying an aggregate operator (in particular, the count aggregate) on the result of join and/or anti-join operations. Satisfaction of stream integrity constraints is the prerequisite for each of the rewrites. We assume that we have been given stream constraints that describe the data streams involved in the queries.

The rewrites are formulated for a pair of streams (or substreams), S_1 and S_2, with a common schema. For simplicity, the schema uses three generic attributes, time, id, and a, standing for the timestamp, a (possibly virtual) identifier of tuples corresponding to the same entity, and *other* attributes in the tuples arriving in the streams (denoted here by a single attribute a). This arrangement simplifies the exposition of the rules without limiting their applicability.

4.1 Window Predicate Elimination

The first rewrite eliminates window predicates in the WHERE clause by rediscovering the *true* identities (e.g., of the actual TCP flows) to which individual items arriving on the stream(s) belong. Note that here we use the *virtual attribute* id to represent this identifier.

The window predicate elimination rule is defined as follows. The selection condition

```
WHERE  S_1.a = S_2.a AND S_2.time − S_1.time ≤ δ
```

that relates two data streams simplifies to

```
WHERE  S_1.id = S_2.id
```

if the following constraints

```
DEFINE STREAM S_1 (
  STREAM KEY (a) SPANS S_2 WINDOW [0,ϵ]
  STREAM KEY (id) SPANS S_2 WINDOW [0,*]
  COOCCURS (id) AND S_2(id) WINDOW [0,ϵ']
)
```

hold in S_1 for $\epsilon \geq \delta \geq \epsilon'$. Note that the third required constraint is only implied in our running example by stating, e.g., that there is only one FIN packet (id is a key for the FIN stream) and that the FIN packet arrives at most ten seconds after the matching SYN-ACK packet.

PROOF. Having two tuples that satisfy the first selection condition, their id attribute values must be the same due to the first constraint as $\epsilon \geq \delta$. On the other hand, two packets that agree on the id attribute and thus satisfy the second selection condition, must, by the second and third constraints, also satisfy the first selection condition, in particular the window condition as $\delta \geq \epsilon'$. □

In our running example, $\epsilon' = 5$ in the COOCCURS clause between the SYN and SYN-ACK streams, and $\epsilon' = 10$ in the COOCCURS clause between the SYN-ACK and FIN streams. Furthermore, $\epsilon = 15$ in both cases (ip is a key within a 15-second window). Therefore, the above rewriting is valid only if the values of δ in the join and anti-join predicates of the original query are between 5 and 15, and between 10 and 15, respectively.

4.2 Join Elimination

The join/anti-join elimination rules are used to completely remove the join/anti-join from the query and replace it by an arithmetic expression that involves independent counts over the involved streams. Note the crucial use of the conceptual `id` attribute in the integrity constraints enabling this rewrite.

The join elimination rule reads as follows. Let $G \in$ `a` be a set of grouping attributes appearing in both S_1 and S_2 (note that G is empty in our motivating example). Since `id` is the identifier attribute, it functionally determines `a`, and therefore tuples from S_1 and S_2 that join on `id` belong to the same group. Let $p(G)$ be a predicate that enforces that the values of each grouping attribute in G are equal in S_1 and S_2. The query

```
SELECT   tb, G, count(*) as cnt
FROM     S1, S2
WHERE    S1.id=S2.id
GROUP BY S1.time/k as tb, G
```

rewrites to

```
SELECT   tb, G, min(s1.cnt,s2.cnt) as cnt
FROM     ( SELECT   tb, G, count(*) as cnt
           FROM     S1
           GROUP BY time/k as tb, G ) s1,
         ( SELECT   tb, G, count(*) as cnt
           FROM     S2
           GROUP BY time/k as tb, G ) s2
WHERE    s1.tb=s2.tb AND p(G) AND
         min(s1.cnt,s2.cnt)>0
```

with a boundary error bounded by ϵ/k (assuming uniformity of packet arrival over k time units) if the following constraints hold

```
DEFINE STREAM S1 (
  PRIMARY KEY (id)
)
DEFINE STREAM S2 (
  PRIMARY KEY (id)
  FOREIGN KEY (id,a) REFERENCES S1(id,a)
                     WINDOW [−ε, 0]
)
```

PROOF. The `id` attribute is the key for both streams S_1 and S_2, thus there can be at most one pair of tuples that agree on `id`. The inclusion dependency constraint postulates that there indeed must be exactly one such pair for each tuple in S_1 (or S_2). Thus the minimal value of the independent counts is indeed equal to the count of tuples in the result of the join. □

We reiterate that the boundary error of ϵ/k assumes that packets on S_1 and S_2 arrive at a uniform rate throughout each window of size k. Given this assumption, only those S_1-tuples which arrive within the last ϵ time units of the current window may have matching S_2-tuples that arrive in the next group-by window and therefore are not included in the count over the current window. Clearly, no bound can be given in the general case, where an adversary may send all S_1-tuples without matching S_2-tuples near window boundaries. In practice, one way to eliminate this worst-case scenario is to issue a set of queries that all count the number of packets on each input stream, but whose group-by windows are offset from one another.

We also note that the above rewriting is consistent with the well-known rule for estimating the cardinality of a join of S_1 and S_2 as $\frac{|S_1||S_2|}{\max(V(S_1,id),V(S_2,id))}$, where $V(S,j)$ denotes the number of distinct values of the join attribute j in table S [12]. In the above rewriting, the join attribute `id` is a primary key, therefore $V(S_1,id) = |S_1|$ and $V(S_2,id) = |S_2|$. Moreover, `id` and `a` are foreign keys, therefore the assumption of containment of value sets needed for the above formula to hold is satisfied. Hence, the size of the above join is $\min(|S_1|,|S_2|)$.

4.3 Anti-Join Elimination

Similarly to the *join case* we can treat anti-joins:

```
SELECT   tb, G, count(*) as cnt
FROM     S1
WHERE    NOT EXISTS
         ( SELECT *
           FROM   S2
           WHERE  S1.id=S2.id )
GROUP BY S1.time/k as tb, G
```

rewrites to

```
SELECT tb, G, s1.cnt-s2.cnt as cnt
FROM   ( SELECT   tb, G, count(*) as cnt
         FROM     S1
         GROUP BY time/k as tb, G ) s1,
       ( SELECT   tb, G, count(*) as cnt
         FROM     S2
         GROUP BY time/k as tb, G ) s2
WHERE  s1.tb=s2.tb AND p(G) AND
       s1.cnt-s2.cnt>0
```

with a boundary error bounded by ϵ/k (again, assuming uniformity over k time units) if the following constraints hold

```
DEFINE STREAM S1 (
  PRIMARY KEY (id)
)
DEFINE STREAM S2 (
  PRIMARY KEY (id)
  FOREIGN KEY (id,a) REFERENCES S1(id,a)
                     WINDOW [−ε, 0]
)
```

PROOF. Since the `id` attribute is a key for both streams, for each S_1 tuple there is at most one S_2 and for each S_2 there is exactly one S_1 tuple in the streams and thus the difference of the independent counts is equal to the count of tuples in the set difference (note that here we use the *conceptual* `id` attribute in a crucial way as the *real* attributes of tuples arriving on the streams do not have this property). The first aggregate subquery gives the exact count of the S_1 tuples and the second one the count of matching S_2 tuples not accounting for tuples within the last ϵ time units. □

Note that the symmetric variant in the anti-join case is vacuous as it implies that the result is empty (due to the constraint stating that for every S_2 tuple there must be at least one corresponding S_1 tuple).

Note also that it is relatively easy to see that instances of streams that violate the integrity constraints make the above rewrites invalid.

4.4 Multiple Streams and Complex Queries

While the rules presented so far have been specified in terms of two data streams, it is easy to extend the technique to multiple streams whose items refer to the same conceptual entities (otherwise there is little point of asking queries that correlate the streams). Using a natural inductive argument, the rewrites can be extended to queries of the form

```
SELECT   tb, count(*) as cnt
FROM     Q'
GROUP BY S_1.time/k as tb
```

where Q' is an arbitrary combination of joins and anti-joins over a finite set of input streams, such that the integrity constraints required by the atomic rewrites in Sections 4.1–4.3 are satisfied for each operator in Q'. This allows applying the rules top-down, ultimately eliminating all joins and anti-joins. As the integrity constraints are preserved under common join reordering, we are not restricted to searching for a particular plan for Q' to perform the rewriting.

Example 4 The rewrite used in our example in Section 2 uses the composition of *window predicate elimination*, *join elimination* and *anti-join elimination* transformations.

In practice, however, the rules should be integrated in a DSMS query optimizer, where their application will be determined by the optimizer's search strategy and cost model.

5. EXPERIMENTS

5.1 Setting

We now present experimental results to illustrate the efficiency and accuracy of our rewrite rules in a practical setting. We used Gigascope to test four versions of the running example from Section 2, using default settings for Gigascope system parameters such as hash table sizes. The four queries are: the original version with a join and anti-join, the original version over sampled inputs with sampling rates of 1 or 10 percent, and our rewritten version, which counts the total number of SYNs, SYN-ACKs and FINs in each 5-minute interval. For brevity, we omit the results of experiments with other types of queries listed in the Introduction (e.g., DNS or HTTP requests and responses) as the results were similar. In order to maximize the accuracy of the sampled queries, we instructed Gigascope to perform per-connection rather than random sampling [24]. That is, if a SYN packet happened to be included in the sample, then the corresponding SYN-ACK and FIN packets (i.e., those having the same IP addresses and ports, and having timestamps within 5 or 10 seconds, respectively, of one another) were also included.

To ensure consistency across experiments, we captured one hour of IP packets from an AT&T data source and separately performed each experiment by replaying the captured stream. The average data rate was approximately 100,000 packets/sec (about 400 Mbits/sec) in each direction and the total number of packets in the captured data stream was over 700 million. In order to test the scalability of our solution, we also created a faster stream by merging five copies of the captured IP trace and perturbing the IP addresses to ensure that there were five times as many TCP connections. We label the original trace "slow" and the merged trace "fast".

The experiments were performed on a dual-core 3GHz Intel Xeon server with 4Gb of RAM, running Linux 2.4.21. Two properties were measured: efficiency and accuracy. Since streaming queries continuously monitor their inputs, it does not make sense to measure efficiency in terms of running time. Instead, we recorded the total CPU time taken to process the one-hour trace and calculated the average CPU utilization. In terms of accuracy, we report the relative error of the result returned by each technique (i.e., the number of TCP connections with missing FINs), as compared to the original query with joins and no sampling. Technically, even the original query incurs errors (e.g., due to TCP flows lasting longer than 15 seconds), which we ignore since we are only interested in the relative error of independent counting versus joining sampled streams. In practice, this error is small and can be estimated via profiling and incorporated into the application logic. For instance, we found that only approximately 2 percent of TCP connections in the tested packet stream were longer than 15 seconds; this observation is consistent with recent work on modeling TCP traffic [22]. Therefore, the application should not report a possible SYN flood attack unless the number of missing FINs is significantly higher than 2 percent.

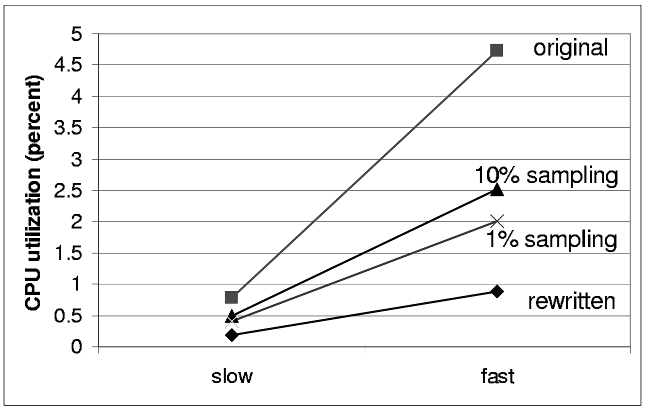

Figure 1: CPU utilization measurements

We summarize our findings as follows. *In order to match the efficiency of the rewritten query, the original query requires a sampling rate of less than one percent. However, once the sampling rate drops to ten percent, the original query is already less accurate than the rewritten query.*

5.2 Results

Figure 1 illustrates the CPU utilization over the slow and fast streams. The rewritten query is the fastest, followed by the original queries with 1 percent sampling, 10 percent sampling, and no sampling, respectively. On the slower stream, the rewritten query is approximately five times more efficient than the original query without sampling, though both use less than one percent of the CPU. However, the CPU usage of the original query grows to nearly 5 percent over the fast stream, which is roughly six times that of the rewritten query (i.e., simple aggregation is more scalable than a join). We hypothesize that the original query can be an order of magnitude slower than the rewritten query over very fast streams. In general, given that a DSMS is expected to run a large collection of streaming queries, even using 5 percent of the CPU for a single query may be excessive.

Next, we measured the errors incurred by the tested queries as compared to the original query without sampling. Each variant was tested with three time window sizes: 1 minute, 2 minutes, and 5 minutes. Intuitively, as the window size grows, the rewritten query should be more accurate since there are fewer error-inducing boundaries (recall Section 2). Similarly, the original queries (with sampling) should be more accurate over larger windows, which correspond to larger sample sizes. In either case, the window length must be small enough (e.g., up to 5 minutes) in order to allow new results (and alarms) to be generated in a timely manner.

Figure 2 shows the ranges of relative errors (per window) over the one-hour packet trace. In all cases, more accurate results are obtained over larger windows, as expected. Moreover, the rewritten query is more accurate than the 10-percent-sampled original query, and much more accurate than the 1-percent-sampled origi-

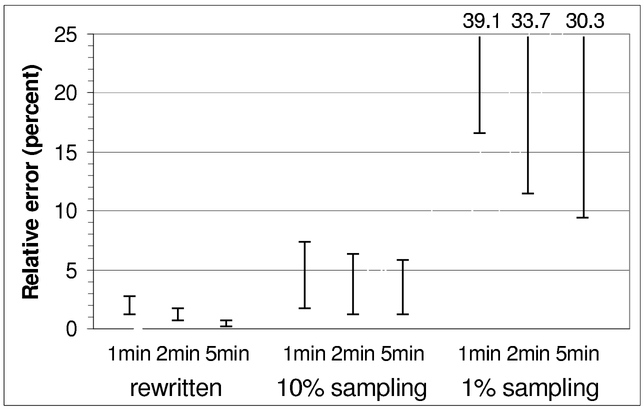

Figure 2: Accuracy measurements

nal query, both in absolute terms and in terms of the variance in the approximation error.

Notably, the error in the rewritten query is roughly an order of magnitude below the upper bound of ϵ/k described in Section 4. Here, $\epsilon = 15$ seconds and k is 1, 2, or 5 minutes, giving upper bounds of 25, 12.5, and 5 percent, respectively. On the other hand, the results in Figure 2 show that the approximation error of the rewritten query does not exceed 3, 1.5, and 0.5 percent, respectively. There are two main reasons why the boundary error of ϵ/k is a loose upper bound.

First, the upper bound assumes that all TCP connections are ϵ seconds long. That is, any SYN packet that arrives within ϵ seconds before the end of a window is assumed to have a matching FIN packet in the next window, thereby causing a boundary error. This is not true in our data set. In fact, we observed that for most time windows of size between one and 5 minutes, at least two thirds of the TCP connections were one or two seconds long. Furthermore, at least 90 percent of connections were at most 5 seconds long. Given this distribution of TCP flow lengths, we can tighten the error bounds by assuming that two thirds of the connections are exactly 2 seconds long, $(0.9 - \frac{2}{3})$ of the connections are exactly 5 seconds long, and the remaining ten percent are exactly 15 seconds long. Hence, all the two-second connections that start in the last two minutes before the end of a window have matching FINs in the next window, and so on. Given a window of size w, the proportion of connections causing boundary errors is:

$$\frac{(\frac{2}{3} \times 2) + ((0.9 - \frac{2}{3}) \times 5) + (0.1 \times 15)}{w}$$

Thus, the revised bounds are 6.7, 3.3, and 1.3 percent for window sizes of 1, 2, and 5 minutes, respectively.

The above numbers are still higher than the observed error bounds. Most of the remaining discrepancies between the predicted and observed errors may be explained as follows. Some missing FINs from a particular time window (that arrive in the next window) are "canceled out" by superfluous FINs that arrived earlier in this window (and belong to connections from a previous window).

6. RELATED WORK

The optimization framework presented in this paper generalizes those of Kompella *et al.* [20] and Wang *et al.* [29], who propose to monitor the difference between the number of SYNs and FINs in order to detect denial-of-service attacks. We have formalized this idea, showed how to integrate it into a DSMS, and experimentally verified its advantages over current DSMS query optimization strategies.

Integrity constraints for time-dependent data have been studied in the area of temporal databases. For example, *temporal functional dependencies* have been studied by Jensen *et al.* [16] and extended to accommodate granularities of time by Wang *et al.* [30] and Wijsen [31]. The main goal of these approaches was to develop tools for modeling of temporal databases, e.g., the ERVT data model [1] that supports time-stamping and evolution constraints, IS-A links, and disjointness and covering constraints. However, none of these approaches provides constraints expressive enough to allow capturing the properties of data streams enabling the rewrites proposed in this paper. The underlying constraint theory is a combination of a powerful logic for linear time, S1S [6], with generalized full dependencies, both tuple- and equality-generating [7]. The technique used to combine these theories is based, in part, on $\mathrm{Datalog}_{1S}$ [8].

Previous work on stream constraints includes the concept of join referential integrity, which is similar to our COOCCURS clause [4]. However, identifying higher-level entities across several streams, which enables the rewrites proposed in this paper, was not discussed. Furthermore, inference of constraints was not considered.

The (Anti-)Join-Count query rewriting rule proposed in this paper rule is fundamentally different from aggregate-join pushdown rules proposed, e.g., by Paulley&Larson [23], Gupta *et al.* [14], Srivastava *et al.* [25], or DeHaan *et al.* [11]: all the above use functional dependencies (under constraints with varying expressiveness) to identify whether a subquery of a join functionally determines all grouping attributes. This allows commuting the grouping-aggregation with the join. In our approach, the join operation is completely eliminated (replaced with a scalar operation) based on an inclusion dependency. Note also that the groups in our case are, in general, not functionally determined by either of the join subqueries.

Efficient techniques for evaluating joins over streaming data have been studied in the past [13, 19, 28]. Our solutions aim at eliminating the joins altogether. At gigabit speeds, this may be the only feasible option.

Also, the proposed technique is complementary to the work on optimization of streaming queries, based, e.g., on rate of delivery [2], utilization of resources [3, 26], or quality of service [27], as the application of the proposed optimization rules can be integrated with an appropriate search strategy and a cost function in the query optimizer.

7. CONCLUSIONS

In this paper, we developed rewriting techniques that eliminate joins from streaming aggregation queries in the presence of appropriate integrity constraints. We defined a theory in which various constraints may be expressed, presented a suite of rewrite rules, and showed the advantages of our solution in a real-world setting.

An interesting direction of future work is to analyze errors due to approximate satisfaction of constraints (e.g., as already mentioned, some TCP flows last longer than 15 seconds), and their propagation during query transformation, based on logical inference. We are also interested in expanding our constraint theory to find other types of useful rewritings. One example involves counting out-of-order packets and retransmissions in IP traffic streams. Rather than performing an explicit self-join and looking for multiple copies of the same packet, we want to formalize the necessary conditions and transformation rules for answering these types of queries using aggregation alone, e.g., by counting *gaps* between the sequence numbers of consecutive packets.

8. REFERENCES

[1] A. Artale, E. Franconi, and F. Mandreoli. Description Logics for Modelling Dynamic Information. In *Logics for Emerging Applications of Databases*. Lecture Notes in Computer Science, Springer-Verlag, 2003.

[2] A. Ayad and J. F. Naughton. Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams. In *ACM SIGMOD*, pages 419–430, 2004.

[3] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. In *ACM SIGMOD*, pages 253–264, 2003.

[4] S. Babu, U. Srivastava, and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3):545–580, 2004.

[5] C. Bettini, S. Jajodia, and X. S. Wang. *Time Granularities in Databases, Data Mining, and Temporal Reasoning*. Springer, 2000.

[6] J. R. Büchi. On a Decision Method in Restricted Second Order Arithmetic. In *International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11, 1962.

[7] A. K. Chandra, H. R. Lewis, and J. A. Makowsky. Embedded Implicational Dependencies and their Inference Problem. In *ACM STOC*, pages 342–354, 1981.

[8] J. Chomicki. Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding. *ACM Trans. Database Syst.*, 20(2):149–186, 1995.

[9] J. Chomicki and D. Toman. Temporal Databases. In *Handbook of Temporal Reasoning in Artificial Intelligence*. Elsevier, 2005.

[10] C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *ACM SIGMOD*, pages 647–651, 2003.

[11] D. DeHaan, D. Toman, and G. E. Weddell. Rewriting Aggregate Queries using Description Logics. In *Description Logics*, pages 103–112. CEUR-WS vol.81, 2003.

[12] H. Garcia-Molina, J. Ullman, and J. Widom. Database System Implementation. *Prentice Hall*, 2000.

[13] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, 2003.

[14] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *VLDB*, pages 358–369, 1995.

[15] Information Sciences Institute. RFC 793, 1981.

[16] C. S. Jensen, R. T. Snodgrass, and M. D. Soo. Extending Existing Dependency Theory to Temporal Databases. *IEEE TKDE*, 8(4), 1996.

[17] T. Johnson, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Streams, security and scalability. In *IFIP Data and Applications Security, LNCS 3654*, pages 1–15, 2005.

[18] F. Kabanza, J.-M. Stevenne, and P. Wolper. Handling Infinite Temporal Data. *J. Comput. Syst. Sci.*, 51(1):3–17, 1995.

[19] J. Kang, J. F. Naughton, and S. Viglas. Evaluating Window Joins over Unbounded Streams. In *ICDE*, pages 341–352, 2003.

[20] R. Kompella, S. Singh, and G. Varghese. On scalable attack detection in the network. *IEEE/ACM Transactions on Networking*, 15(1):14–25, 2007.

[21] F. Korn, S. Muthukrishnan, and Y. Zhu. Checks and Balances: Monitoring Data Quality Problems in Network Traffic Databases. In *VLDB*, pages 536–547, 2003.

[22] M. Mellia, I. Stoica, and H. Zhang. TCP Model for Short Lived Flows. *IEEE Communcations Letters*, 6(2):85–87, 2002.

[23] G. N. Paulley and P.-Å. Larson. Exploiting Uniqueness in Query Optimization. In *ICDE*, pages 68–79, 1994.

[24] V. Shkapenyuk, T. Johnson, S. Muthukrishnan, and O. Spatscheck. Query-aware sampling for data streams. In *Int. Workshop on Scalable Stream Processing Systems (SSPS)*, 2007.

[25] D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy. Answering queries with aggregation using views. In *VLDB*, pages 318–329, 1996.

[26] U. Srivastava and J. Widom. Memory-Limited Execution of Windowed Stream Joins. In *VLDB*, pages 324–335, 2004.

[27] N. Tatbul, U. Cetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *VLDB*, pages 309–320, 2003.

[28] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. In *VLDB*, pages 285–296, 2003.

[29] H. Wang, D. Zhang, and K. Shin. Change-point monitoring for detection of DoS attacks. *IEEE Trans. on Dependable and Secure Comp.*, 1(4):193–208, 2004.

[30] X. S. Wang, C. Bettini, A. Brodsky, and S. Jajodia. Logical Design for Temporal Databases with Multiple Granularities. *ACM Trans. Database Syst.*, 22(2):115–170, 1997.

[31] J. Wijsen. Temporal FDs on Complex Objects. *ACM Trans. Database Syst.*, 24(1):127–176, 1999.

APPENDIX

A. DECIDABILITY AND COMPLEXITY OF STREAM INTEGRITY CONSTRAINTS

For a fixed theory of linearly ordered time we define stream constraints as follows:

Definition 5 (Constraints) *Let $R_1, \ldots, R_k$, $S_1, \ldots, S_l$ be predicate symbols (not necessarily distinct), ψ and ψ' conjunctions of atomic equalities, and φ and φ' formulas in the theory of time. We define* stream integrity constraints *to be formulas of the form:*

$$\forall \mathbf{t} \forall \mathbf{x}. R_1(t_1, \mathbf{x}_1) \wedge \ldots \wedge R_k(t_k, \mathbf{x}_k) \wedge \varphi_{\mathbf{t}} \wedge \psi_{\mathbf{x}} \\ \rightarrow \left\{ \begin{array}{l} \exists \mathbf{t}'. \psi'_{\mathbf{t},\mathbf{t}'} \wedge S_1(t'_1, \mathbf{y}_1) \wedge \ldots \wedge S_l(t_l, \mathbf{y}_l) \\ \psi'_{\mathbf{x}} \\ \varphi'_{\mathbf{t}} \end{array} \right.$$

for $\mathbf{y}_1 \cup \ldots \cup \mathbf{y}_l \subseteq \mathbf{x} = \mathbf{x}_1 \ldots \mathbf{x}_k$ vectors of data variables, $\mathbf{t} = \{t_1, \ldots, t_k\}$, and $\mathbf{t}' = \{t'_1, \ldots, t'_l\}$ to be time variables. The subscripts of the φ and ψ formulas in the constraints indicate the sets of allowable free variables in these formulas.

Given a finite set of constraints Σ and a constraint C, an implication problem *$\Sigma \models C$ is a question whether C is true in all models of Σ.*

The above constraints can be thought of as *temporal* variants of functional dependencies and inclusion dependencies. Note however, that the temporal part of the constraints can use arbitrarily complex formulae in the theory of time—this arrangement makes this approach much more expressive than virtually any temporal integrity constraints proposed in the literature so far [5, 9, 16]. We also allow combining right hand sides of constraints with the same antecedent by conjunction; this is mere syntactic convenience.

The constraints present in our running example are shown in Figure 3 (we omit the external universal quantifiers). They specify that (i-iii) the virtual attribute `id` is a *true* key of each sub-stream, (iv-v) `id` is a *foreign* key, (vi-vii) `ip` is a key within 15-second windows, and (viii-ix) FIN packets arrive within 10 seconds after the corresponding SYN-ACK packet, and SYN-ACK packets arrive within 5 seconds after the corresponding SYN packet.

A.1 Correspondence Theorem

First we show how to convert a logical implication problem for stream constraints to a decision problem in the underlying theory of time. We need several (technical) definitions to simulate the effects of the data values in the constraints. In particular, for every predicate symbol $R(t, \mathbf{x})$ and every substitution $[\mathbf{a}/\mathbf{x}]$ of constants in A for the variables $\mathbf{x}$ we define a unary predicate symbol $R^{\mathbf{a}}(t)$; the collection of these unary predicates represents $R(t, \mathbf{x})$ in the result of the transformation.

Definition 6 *Let σ be a schema and A a finite set of constants. We define*

$$\sigma(A) = \{R^{\mathbf{a}}(t) : R(t, \mathbf{x}) \in \sigma, \mathbf{a} \in A^{|\mathbf{x}|}\}$$

$$\begin{aligned}\mathrm{AUX}^{\sigma}_{A} = \{\, & \mathrm{E}^{a,a}, \mathrm{E}^{a,b} \leftrightarrow \mathrm{E}^{b,a}, \mathrm{E}^{a,b} \wedge \mathrm{E}^{b,c} \rightarrow \mathrm{E}^{a,c}, \\ & \forall t.\mathrm{E}^{a,b} \rightarrow (R^{\mathbf{a}a\mathbf{b}}(t) \leftrightarrow R^{\mathbf{a}b\mathbf{b}}(t)) : \\ & \quad a, b, c \in A, R^{\mathbf{a}a\mathbf{b}}, R^{\mathbf{a}b\mathbf{b}} \in \sigma(A)\,\}.\end{aligned}$$

The additional propositions $\mathrm{E}^{a,b}$ simulate the effects of equality in the original formulae: whenever constants a and b are forced to be equal in a model of the original constraints, e.g., as a consequence of a functional dependency, the proposition $\mathrm{E}^{a,b}$ is true in the corresponding model on the transformed theory. The set $\mathrm{AUX}^{\sigma}_{A}$ captures the interactions between the $\mathrm{E}^{a,b}$ propositions and the remainder of the translation. The symbols defined above are used to transform constraints in the constraint theory Σ as follows:

Definition 7 *Let C be a constraint, A a finite set of constants, and θ is a substitution for variables $\mathbf{x}$ with values from A. We define*

$$C\theta = \forall \mathbf{t} R_1^{\mathbf{x}_1\theta}(t_1) \wedge \ldots \wedge R_k^{\mathbf{x}_k\theta}(t_k) \wedge \varphi_{\mathbf{t}} \wedge (\psi_{\mathbf{x}}\theta)$$
$$\rightarrow \left\{ \begin{array}{l} \exists \mathbf{t}'.\psi'_{\mathbf{t},\mathbf{t}'} \wedge S_1^{\mathbf{y}_1}(t'_1) \wedge \ldots \wedge S_l^{\mathbf{y}_l}(t'_l) \\ (\psi'_{\mathbf{x}}\theta) \\ \varphi'_{\mathbf{t}} \end{array} \right.$$

where $(\psi_{\mathbf{x}}\theta)$ is the formula ψ in which each atomic subformula $x_i = x_j$ is replaced by a proposition $\mathrm{E}^{x_i\theta, x_j\theta}$ and $R_i^{\mathbf{x}\theta}, S_j^{\mathbf{y}\theta} \in \sigma(A)$ are unary predicates in the theory of time.

Given a theory Σ for the schema σ, we define a set of formulas

$$\begin{aligned}\mathrm{SAT}_A(\Sigma) = \{C\theta : C \in \Sigma,\ & \theta \text{ a substitution} \\ & \text{for } \mathbf{x} \text{ with values from } A\}.\end{aligned}$$

For an implication problem $\Sigma \models C$, the consequent C is transformed into a contrapositive form by Skolemizing all quantifiers over data variables as follows:

Definition 8 *Let C be a constraint and $\{x_1, \ldots, x_k\}$ the set of (universally quantified) data variables in C. We define A_C to be the set $\{a_1, \ldots, a_k\}$ of distinct constants and*

$$\mathrm{NSAT}(C) = \neg(C[a_1/x_1, \ldots, a_k/x_k]).$$

This way $\Sigma \models C$ is transformed to a corresponding satisfiability problem in the theory of time.

Theorem 9 *Let Σ be a set of constraints and C a constraint over the schema σ. Then $\Sigma \models C$ if and only if $\mathrm{AUX}^{\sigma}_{A_C} \cup \mathrm{SAT}_{A_C}(\Sigma) \cup \{\mathrm{NSAT}(C)\}$ is not satisfiable.*

PROOF. *Consider first that $\mathrm{AUX}^{\sigma}_{A_C} \cup \mathrm{SAT}_{A_C}(\Sigma) \cup \{\mathrm{NSAT}(C)\}$ is satisfiable and thus has a model T with a domain $\mathbf{dom}_T$. In T, the propositions $\mathrm{E}^{a,b}$ define an equivalence relation on A_C. We designate a* canonical value *for each of the equivalence classes. We say that a substitution θ is* canonical *if it only uses canonical values in A_C.*

Now we construct a structure M as follows:

$$\begin{aligned} & R_i(t, \mathbf{x}\theta) \text{ is true in } M \iff \\ & \quad R_i^{\mathbf{x}\theta}(t) \text{ is true in } T \text{ for } \theta \text{ canonical and } t \in \mathbf{dom}_T. \end{aligned}$$

It is easy to verify that $M \models \Sigma$, assuming otherwise leads to a contradiction with $T \models \mathrm{AUX}^{\sigma}_{A_C} \cup \mathrm{SAT}_{A_C}(\Sigma)$. However, $M \not\models C$ as otherwise we would have $T \models \mathrm{NSAT}(C)$.

For the converse, consider a structure M such that $M \models \Sigma$ and $M \not\models C$. Then, for C to be falsified, there must be a substitution $[a_1/x_1, \ldots, a_k/x_k]$ that makes the precondition of C true and the consequent false in M. Let $A = \{a_1, \ldots, a_k\}$ and substitutions θ range over A. We construct a structure T setting

$$R_i^{\mathbf{x}\theta}(t) \text{ is true in } T \iff M \models R(t, \mathbf{x}\theta) \text{ is true in } M$$

for θ a substitution and $t \in \mathbf{dom}_M$. We also make $\mathrm{E}^{a,a}$ true and $E^{a,b}$, $a \neq b$, false in T for $a, b \in A$. Then $T \models \mathrm{AUX}^{\sigma}_{A_C}$ (trivially) and $T \models \mathrm{SAT}_{A_C}(\Sigma)$ since falsifying $C_i\theta \in \mathrm{SAT}_{A_C}(\Sigma)$ would also falsify $M \models C_i$. $T \models \mathrm{NSAT}(C)$ follows from the construction. $\square$

$\mathrm{AUX}^{\sigma}_{A_C} \cup \mathrm{SAT}_{A_C}(\Sigma) \cup \{\mathrm{NSAT}(C)\}$ is a monadic formula in the theory of time (with one quantifier alternation in addition to alternations present in the temporal parts of the constraints). Hence, using [6], we have:

Corollary 10 *Let the theory of time be the theory of one successor function (S1S). Then the logical implication problem is decidable.*

PROOF. *Immediate by observing that the second-order existential closure of the conjunction of formulas in $\mathrm{AUX}^{\sigma}_{A_C} \cup \mathrm{SAT}_{A_C}(\Sigma) \cup \{\mathrm{NSAT}(C)\}$ is a S1S sentence.* $\square$

The choice of a very powerful logic to serve as the basis for reasoning about time in the proposed stream constraints allows the user to specify (non first-order) properties of data streams, e.g., periodic events [18], time granularities [5], etc., in addition to the more common temporal keys and functional dependencies [31].

A.2 Complexity of Reasoning

In general, the constraints imposed on the temporal dimension can be arbitrary complex S1S formulas, yielding a (tight) non-elementary complexity bound even for deciding satisfiability of a single constraint. However, the size of individual constraints is fixed for the translations of streaming constraints and therefore we are mainly concerned with the *number of constraints* resulting from the *instantiation* by Skolem constants:

Lemma 11 *The size of $\mathrm{AUX}^{\sigma}_{A_C} \cup \mathrm{SAT}_{A_C}(\Sigma) \cup \{\mathrm{NSAT}(C)\}$ is exponential in the (maximal) number of variables in a constraint C and polynomial in the number of constraints.*

Thus, following the standard construction of a Büchi automaton [6] for S1S and assuming *constant* size of the automata for the individual constraints, we end with an automaton roughly exponential in the size of the constraint theory. This is no worse than other schema

$$\begin{array}{ll}
(i) & \mathsf{SYN}(t_1, i, a_1) \wedge \mathsf{SYN}(t_2, i, a_2) \rightarrow a_1 = a_2 \wedge t_1 = t_2 \\
(ii) & \mathsf{SYN\text{-}ACK}(t_1, i, a_1) \wedge \mathsf{SYN\text{-}ACK}(t_2, i, a_2) \rightarrow a_1 = a_2 \wedge t_1 = t_2 \\
(iii) & \mathsf{FIN}(t_1, i, a_1) \wedge \mathsf{FIN}(t_2, i, a_2) \rightarrow a_1 = a_2 \wedge t_1 = t_2 \\
(iv) & \mathsf{SYN}(t_1, i, a_1) \wedge \mathsf{SYN\text{-}ACK}(t_2, i, a_2) \rightarrow a_1 = a_2 \wedge t_2 \geq t_1 \\
(v) & \mathsf{SYN\text{-}ACK}(t_1, i, a_1) \wedge \mathsf{FIN}(t_2, i, a_2) \rightarrow a_1 = a_2 \wedge t_2 \geq t_1 \\
(vi) & \mathsf{SYN}(t_1, i_1, a) \wedge \mathsf{SYN\text{-}ACK}(t_2, i_2, a) \wedge (t_2 \geq t_1) \wedge (t_2 - t_1 \leq 15) \rightarrow i_1 = i_2 \\
(vii) & \mathsf{SYN\text{-}ACK}(t_1, i_1, a) \wedge \mathsf{FIN}(t_2, i_2, a) \wedge (t_2 \geq t_1) \wedge (t_2 - t_1 \leq 15) \rightarrow i_1 = i_2 \\
(viii) & \mathsf{FIN}(t_1, i, a) \rightarrow \exists t_2.\mathsf{SYN\text{-}ACK}(t_2, i, a) \wedge (t_1 \geq t_2) \wedge (t_1 - t_2 \leq 10) \\
(ix) & \mathsf{SYN\text{-}ACK}(t_1, i, a) \rightarrow \exists t_2.\mathsf{SYN}(t_2, i, a) \wedge (t_1 \geq t_2) \wedge (t_1 - t_2 \leq 5)
\end{array}$$

Figure 3: Constraints present in and across the SYN, SYN-ACK, and FIN sub-streams.

languages proposed for database systems. Note also that logical implication for the theory of full dependencies combined with functional dependencies alone, a non-temporal sub-theory of our constraint theory, is already EXPTIME-complete [7] and the temporal sub-theory, restricted to universally-quantified Horn clauses (Datalog$_{1S}$) is PSPACE-complete [8].