Flexible Time Management in Data Stream Systems*

Utkarsh Srivastava
Stanford University
usriv@cs.stanford.edu

Jennifer Widom
Stanford University
widom@cs.stanford.edu

ABSTRACT

Continuous queries in a Data Stream Management System (DSMS) rely on time as a basis for windows on streams and for defining a consistent semantics for multiple streams and updatable relations. The system clock in a centralized DSMS provides a convenient and well-behaved notion of time, but often it is more appropriate for a DSMS application to define its own notion of time—its own clock(s), sequence numbers, or other forms of ordering and timestamping. Flexible application-defined time poses challenges to the DSMS, since streams may be out of order and uncoordinated with each other, they may incur latency reaching the DSMS, and they may pause or stop. We formalize these challenges and specify how to generate *heartbeats* so that queries can be evaluated correctly and continuously in an application-defined time domain. Our heartbeat generation algorithm is based on parameters capturing skew between streams, unordering within streams, and latency in streams reaching the DSMS. We also describe how to estimate these parameters at run-time, and we discuss how heartbeats can be used for processing continuous queries.

1. INTRODUCTION

There has been considerable recent interest in the problem of processing continuous queries over data streams [4, 10, 11]. Commonly cited applications include monitoring of network traces, sensor data, stock quotes, web usage logs, call records, and others. In most of these applications, data is generated at distributed *sources*, then streamed to a central server where a *Data Stream Management System* (DSMS) is running. *Continuous queries* registered with the DSMS are evaluated over the incoming stream data.

The semantics for continuous queries in a data stream system typically assumes *timestamps* on data stream elements (hereafter *tuples*). For example, time-based sliding windows, common in stream applications [18], are defined based on timestamps [1, 12, 21]. A consistent semantics for multiple streams and updatable relations relies on timestamps [1]. To achieve semantic correctness, the DSMS query processor usually needs to process tuples in increasing timestamp order. That is, the query processor should never receive a tuple with a lower timestamp than any previously received tuple; we refer to this property as the *ordering requirement*.

There are two common ways in which timestamps may be assigned to stream tuples:

1. Tuples are timestamped on entry to the DSMS using the DSMS system time. In this case the ordering requirement presents no difficulties.

2. Sources timestamp the tuples before sending them to the DSMS. Such timestamps define a notion of *application time* and are referred to as application timestamps.

As an example of application timestamps, consider monitoring sensor readings to correlate changes in temperature and pressure. Each tuple consists of a sensor reading and an application timestamp affixed by the sensor, denoting the time at which that reading was taken. In general there may not be any relationship between the time at which the reading is taken (the application timestamp) and the time at which the corresponding stream tuple reaches the DSMS (the system timestamp). Other similar examples with application timestamps are analysis of stock trades, and monitoring packet traces from network routers. Note that application time may simply be a sequence number, e.g., orders processed linearly.

When stream tuples carrying application timestamps arrive at a DSMS from one or more, possibly distributed sources, their arrival may not be in increasing timestamp order, due to a number of possible factors:

1. Unsynchronized application clocks at the sources

2. Different network latencies from different sources to the DSMS

3. Data transmission over a non-order-preserving channel

To handle out-of-order arrival, we assume an architecture that includes an *input manager* (see Figure 1, explained in Section 2) whose function is to buffer out-of-order stream tuples, presenting them to the query processor in increasing timestamp order.

*This work was supported by the National Science Foundation under grant IIS-0118173 and IIS-9817799 and by a Stanford Graduate Fellowship from Sequoia Capital.

PODS 2004 June 14-16, 2004, Paris, France.

It is not always straightforward to decide how long a tuple should be buffered by the input manager before it can be presented to the query processor. For example, consider our sensor application and suppose there are two separate streams for temperature and pressure readings. Further, suppose that the sensors are programmed to generate a tuple only when the new reading differs from the previously reported reading. If the pressure is changing while the temperature is constant, there is a steady arrival of tuples on one stream while the other stream is silent or "paused". However, it is not clear at the DSMS whether temperature tuples are delayed, or (as is the case) there is no temperature data to transmit while application time for the temperature stream nevertheless advances. Without this information, the input manager cannot decide whether the tuples on the pressure stream can be moved to the query processor without violating the ordering requirement, or whether they should be buffered until the arrival of further tuples on the temperature stream.

In general, each tuple must eventually be moved from the input manager to the query processor without violating the ordering requirement, i.e., no tuple should be stalled indefinitely in the input manager. We refer to this property as the *progress requirement*. To meet the progress requirement we propose the use of *heartbeats*. Informally, at any instant, a heartbeat τ for a set of streams provides a guarantee to the DSMS that all tuples arriving on those streams after that instant will have a timestamp greater than τ. If the sources themselves do not provide heartbeats, the DSMS needs to deduce them, and doing so is a primary topic of this paper. When a heartbeat τ arrives or is generated, the input manager can move all tuples with timestamp $\leq \tau$ to the query processor.

Our approach to heartbeat generation is to quantify certain properties of the environment as *parameters* and generate heartbeats based on these parameters, along with the stream data seen so far. For example, there may be a bound on the skew between application clocks at different sources, or the network latency for stream tuples to reach the DSMS may be bounded. Our algorithm is simple, efficient, and general enough to be applicable in a wide variety of environments. When parameter values cannot be specified in advance, we provide a technique for estimating them from the stream data seen so far.

The main contributions of this paper are:

- We formalize the problem and define the parameters needed for heartbeat generation (Section 2).
- Under the assumption that parameter values have been specified, we give an algorithm for heartbeat generation and prove its correctness (Section 3).
- We demonstrate our approach on example data stream application environments (Section 4).
- We discuss implementation and scalability issues of our heartbeat generation algorithm (Section 5).
- For the case when parameter values cannot be specified in advance, we describe how they may be estimated based on the stream arrival pattern so far (Section 6).
- We show how heartbeats can be used in continuous query plans for efficient execution (Section 7).

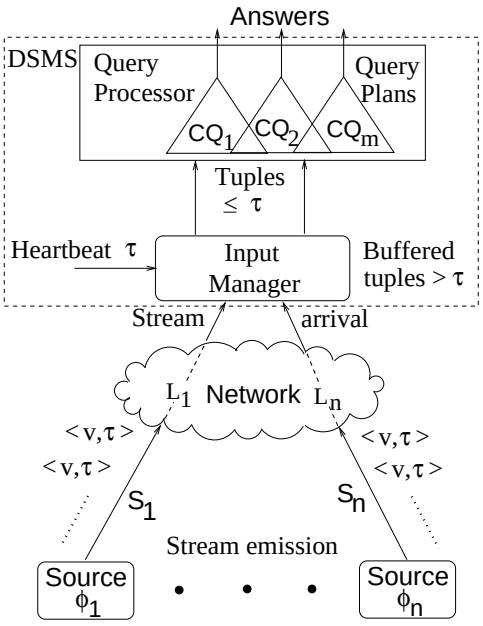

Figure 1: The basic environment we consider

Finally we survey related work in Section 8 and conclude in Section 9.

2. PRELIMINARIES AND PARAMETERS

Figure 1 depicts an abstraction of the environment we consider. Continuous queries $(CQ_1, CQ_2, \ldots, CQ_m)$ are registered with the DSMS. These are executed over input data streams $(S_1, S_2, \ldots, S_n)$ and stored relations (not shown). We assume relation updates also are streamed to the system, and hereafter do not consider them separately. Streams are generated by possibly distributed sources $\phi_1, \phi_2, \ldots, \phi_n$. Each stream tuple is comprised of a value v along with an application-defined timestamp τ from some discrete, ordered domain. Streams are transmitted from their respective sources to the DSMS over a network which may have some transmission latency, upper-bounded by $L_1, L_2, \ldots, L_n$ respectively for each stream. Although a DSMS can be expected to handle multiple arbitrary time domains, we assume a given query conforms to just one.

We will need to consider two different notions of time—actual or "wall-clock" time, and application-defined time. We use different notations for them for clarity: We use τ to denote application timestamps and δ to denote differences in application timestamps. An instant of wall-clock time is denoted by c and intervals of wall-clock time by t. Network latency is also an interval of wall-clock time and is denoted by L. Without loss of generality, for implementation we assume the DSMS system clock emulates wall-clock time.

Since stream tuples may not arrive at the DSMS in increasing timestamp order, there is an *input manager* to buffer tuples until they can be moved to the query processor without violating the ordering requirement (Section 1). The decision as to when a tuple can be moved to the query processor is based on *heartbeats*, and our general goal is to deduce heartbeats for the set of streams $S_1, S_2, \ldots, S_n$ at the DSMS.

> A heartbeat for a set of streams $S_1, S_2, \ldots, S_n$ at wall-clock time c is defined as the maximum application timestamp τ such that all tuples arriving on $S_1, S_2, \ldots, S_n$ after time c must have timestamp $> \tau$.

To have any hope of deducing heartbeats, we need to assume that the generation and transmission of stream tuples satisfies certain properties. For example, application clock skew at the sources, out-of-order stream tuples, and network latency all must be bounded or we cannot deduce heartbeats in general. We propose the following general framework to specify these bounds.

1. *Skew bound*

   We capture the maximum application-time skew between two sources ϕ_i and ϕ_j by a pair (t_{ij}, δ_{ij}) $(t_{ij} \geq 0, \delta_{ij} \geq 0)$ which is interpreted as follows: If at time c, ϕ_i emits a tuple with timestamp τ, then all tuples emitted by ϕ_j after time $c + t_{ij}$ shall have timestamp $> \tau - \delta_{ij}$.

   Intuitively, bound (t_{ij}, δ_{ij}) states that source ϕ_j lags behind source ϕ_i by at most δ_{ij} units of application time, but this guarantee is delayed by t_{ij} units of wall-clock time. This notion of "delayed" guarantees is a novel feature of our framework. It allows us to capture a number of environments precisely, which would not be possible with more naïve methods of skew bound specification. We provide examples in Section 4.

   Frequently one of these two values may be 0. However, we will see in Section 3.1 that in some common cases we do need both parameters to capture an environment accurately.

   In some cases, it may be more appropriate to use a tuple-based rather than a time-based t_{ij}. Also, for extreme generality we could specify a set of pairs (t_{ij}, δ_{ij}) instead of a single pair. These modifications and generalizations are elaborated in Section 3.1.

2. *Out-of-order generation*

   How out-of-order a stream S_i is at emission can be captured conveniently as the application-time skew of source ϕ_i with respect to itself, specified by (t_{ii}, δ_{ii}).

   Using the definition for skew bound above, $(0, \delta_{ii})$ specifies that when source ϕ_i emits a tuple with timestamp τ, all tuples it emits subsequently must have timestamp $> \tau - \delta_{ii}$. That is, the reordering of timestamps is bounded by δ_{ii}. Note that if a stream is in timestamp order with no duplicate timestamps, then $\delta_{ii} = 0$. If there may be duplicate timestamps but no reordering, then $\delta_{ii} = 1$. Again, a nonzero t_{ii} gives a delayed bound on the reordering.

3. *Latency bound*

   The bound on transmission latency from a source ϕ_i to the DSMS is specified as L_i units of wall-clock time: If any tuple from ϕ_i takes t units of wall-clock time to be transmitted to the DSMS, then $0 \leq t \leq L_i$.

In this paper, we are assuming that deterministic bounds are available on the skew, out-of-order generation, and network latency. An alternative, more sophisticated approach would be to work with probabilistic bounds having a certain confidence. For example, we could say L_i is a bound on the network latency with confidence $1 - \epsilon$ if the probability that the latency of a tuple from source ϕ_i to the DSMS exceeds L_i is at most ϵ. Network latency distributions are typically light-tailed [14], so we may be able to have a small high-confidence bound even when the deterministic bound is very large. However, with probabilistic bounds, some tuples may arrive at the DSMS after the corresponding heartbeat has been generated. To preserve the ordering requirement, these tuples must be dropped. However, if the bounds have sufficiently high confidence, the fraction of tuples that are dropped will be very small. In the rest of this paper, we assume deterministic bounds.

Skew and out-of-order bounds can together be represented succinctly as a single *skew bound matrix* B. The entries B_{ij} $(i \neq j)$ bound the skew between sources ϕ_i and ϕ_j, and the diagonal entries B_{ii} bound how out-of order stream S_i is at emission. When no bound can be guaranteed, the corresponding matrix entry is empty. In general skew bounds are not symmetric—source ϕ_i certainly need not lag behind source ϕ_j in the same way that ϕ_j lags behind ϕ_i.

We have considered the case when streams may become further reordered in the network before reaching the DSMS. If it is known that streams cannot be reordered en route, we can incorporate that fact into our heartbeat generation algorithm, but for generality we assume the possibility of further reordering. Note that tuples that are lost in the network during transmission (network latency $= \infty$) do not affect our problem. Since these tuples never reach the DSMS, they do not contribute to query results and cannot violate the ordering requirement.

The skew bound matrix and latency bounds are treated as parameters to our heartbeat generation algorithm, specified in the next section. There are two ways in which these parameters may be obtained. If exact or sufficiently accurate information about the sources and the network is available, then these parameters may be specified by the system administrator(s). (For modularity and ease of specification, our framework provides a clean separation between parameters that depend only on the sources and those that depend only on the network.) However, when information about the sources or the network is insufficient, or properties vary over time, then we need to estimate these parameters based on the stream data so far. In this case, we use the parameters in a slightly different form, in which we do not separate source-dependent and network-dependent parameters.

We assume parameters are known for now, then we cover parameter estimation in Section 6.

3. HEARTBEAT GENERATION

Let us consider a system as depicted in Figure 1, and assume we have an $n \times n$ skew bound matrix B and a set of latency bounds $L_1, L_2, \ldots, L_n$ capturing the parameters of the environment as specified in the previous section. Our goal is to generate heartbeats for the set of streams $S_1, S_2, \ldots, S_n$. Recall that a heartbeat is the maximum (application-time) timestamp τ such that all future tuples arriving on any stream $S_1, S_2, \ldots, S_n$ are guaranteed to have timestamp $> \tau$.

Instead of maintaining a single heartbeat for $S_1, S_2, \ldots, S_n$, we maintain something stronger: a separate heartbeat τ_i for each stream S_i. The heartbeat for $S_1, S_2, \ldots, S_n$ is then

Input: Skew-bound matrix B, Latency bounds $L_1, L_2, \ldots, L_n$
Output: Heartbeats for S_i in array τ_i

1. $\tau_i[0] =$ minimum application time $- 1 \quad \forall i \in \{1, \ldots, n\}$
2. When a tuple with timestamp τ arrives on S_i at time c:
3. for $j = 1$ to n do
4. $\quad \tau_j[c + t_{ij} + L_j] = \max(\tau_j[c + t_{ij} + L_j], \tau - \delta_{ij})$

Figure 2: Heartbeat generation algorithm

$\min(\tau_1, \tau_2, \ldots, \tau_n)$. Generating separate heartbeats for different streams, rather than a single heartbeat over a set of streams, can make query plan execution more efficient as outlined in Section 7.

The basic heartbeat generation algorithm is specified in Figure 2. We assume wall-clock time is measured relative to a time 0 at which the DSMS is started. $\tau_i[c]$ denotes the heartbeat on stream S_i at wall-clock time c. For ease of presentation, the algorithm assumes the existence of an infinite array τ_i for each stream S_i, so that $\tau_i[c]$ can be used to store the heartbeat on S_i for any time instant c. A memory-efficient implementation of array τ_i is discussed in Section 5. Also, the algorithm does not set $\tau_i[c]$ for every stream for every instant c. By definition, heartbeats cannot decrease over time. Thus, if $\tau_i[c]$ is not set for some instant c, we assume $\tau_i[c] = \tau_i[c-1]$.

The algorithm "generates" a heartbeat at a particular instant of time by assigning the appropriate array value. In an actual system, heartbeats may be implemented as interrupts, or the query processor might perform periodic heartbeat lookups.

3.1 Extensions

We present two extensions to our basic skew-bound specification framework that allow us to capture a wider variety of environments:

1. *Tuple-based skew bounds*

   The skew bound (t_{ij}, δ_{ij}) for sources ϕ_i and ϕ_j can instead be specified as (n_{ij}, δ_{ij}), where $n_{ij} \geq 0$ denotes a number of tuples. The meaning is nearly identical, except that the interval of t_{ij} time units is replaced by a tuple-based interval consisting of emission of n_{ij} tuples by ϕ_j.

2. *Skew bound as a set*

   For full generality, the skew bound for sources ϕ_i and ϕ_j could be specified a set of (t_{ij}, δ_{ij}) pairs, instead of just a single pair. For example, it may be that after a small delay t_{ij} we can guarantee a fairly large δ_{ij}, and then after a longer delay t_{ij} we can tighten our δ_{ij} bound. We also can include both time-based and tuple-based bounds in the set.

To see why these extensions might be useful, consider a source ϕ_i that emits tuples in timestamp order but with up to k duplicates of each timestamp. This behavior is captured by the following two tuple-based specifications constituting B_{ii}:

- (0,1): captures sorted order
- (k,0): captures that there can be at most k duplicates

These constraints cannot be captured by time-based specifications, thus justifying the first extension. Further, both the above specifications are required or some heartbeats may be delayed: If only (0,1) is present, a heartbeat τ will be emitted only when a tuple with timestamp $\tau + 1$ has been seen, even if we have already seen k tuples with timestamp τ. If only (k,0) is present, a heartbeat τ will be emitted only after k subsequent tuples after the first τ have been seen, even if the next tuple has timestamp $\tau + 1$.

All of our discussion and algorithms extend to include tuple-based skew bounds. Hence, for presentation we shall assume that B consists only of time-based skew specifications. When B_{ij} is a set, we simply execute line 4 of the algorithm in Figure 2 for each $(t_{ij}, \delta_{ij}) \in B_{ij}$. All other details remain unchanged.

3.2 Proof of Correctness

We prove the correctness of our heartbeat generation algorithm. Assume all parameters are adhered to. Recall $\tau_i[c]$ is the heartbeat on stream S_i at time c. To prove the correctness of our heartbeat generation algorithm, we must prove that for every stream S_i and for every instant c, the following two properties hold for $\tau_i[c]$:

1. After time instant c, no future tuples on S_i can have timestamp $\leq \tau_i[c]$.
2. At time c, $\tau_i[c]$ is the maximum timestamp that the DSMS can infer to satisfy property (1).

Properties (1) and (2) clearly hold when the heartbeats are initialized in step 1 of our algorithm. We show that they remain true as time progresses.

First let us consider when the DSMS obtains new information and may be able to infer a new heartbeat. Given that the data sources and tuple transmission satisfy only the three bounds specified in Section 2, and there is no notion of application-time progress with wall-clock time, a new heartbeat on any stream can be inferred only when a new tuple arrives at the DSMS.

Suppose a tuple is emitted by source ϕ_i at time c and arrives at the DSMS on stream S_i at time c'. This arrival places guarantees on stream S_j using the skew bounds $(t_{ij}, \delta_{ij}) \in B_{ij}$. Since (t_{ij}, δ_{ij}) only gives guarantees on timestamps of tuples emitted by source ϕ_j after time $c + t_{ij}$, the DSMS must wait until it is ensured that all tuples emitted by ϕ_j before or at time $c + t_{ij}$ have arrived on S_j. The following lemma shows that this guarantee cannot be placed earlier than $c' + t_{ij} + L_j$.

LEMMA 3.1. *If a tuple s emitted by ϕ_i at time c arrives on stream S_i at time c', a tuple s' emitted by ϕ_j before or at $c + t_{ij}$ can reach the DSMS up to time $c' + t_{ij} + L_j$ and no later.*

PROOF. The maximum interval between the arrivals of s and s' is obtained when s' is emitted at $c' + t_{ij}$ and takes maximum possible time to reach the DSMS, while s takes the minimum possible time to reach the DSMS. This interval between their arrivals at the DSMS is given by $t_{ij} + L_j - 0 = t_{ij} + L_j$. Thus, if s arrives at time c', s' can arrive up to time $c' + t_{ij} + L_j$ and no later. □

THEOREM 3.2. *If a tuple with timestamp τ arrives on stream S_i at time c, the changes in heartbeat are given by $\tau_j[c + t_{ij} + L_j] = max(\tau_j[c + t_{ij} + L_j], \tau - \delta_{ij})$ for each $(t_{ij}, \delta_{ij}) \in B_{ij}$.*

PROOF. By Lemma 3.1, the guarantee due to any pair $(t_{ij}, \delta_{ij}) \in B_{ij}$ can be given at time $c + t_{ij} + L_j$ and no earlier. Further, by the definition of the skew bound, the guarantee that can be given is that all subsequent timestamps on stream S_j shall be $> \tau - \delta_{ij}$. Thus, heartbeat updates applied by our algorithm are valid, and by Lemma 3.1 the changes cannot be applied any earlier. □

3.3 Ensuring Progress

Recall that progress is defined as the ability of the input manager to eventually move every tuple to the query processor without violating the ordering requirement. Let us assume that all entries in B are nonempty, although bound values may be arbitrarily large. Then, as long as at least one stream is sending tuples, our algorithm uses skew bounds to ensure that all heartbeats are advancing, and progress is ensured. However, if all streams pause, heartbeats will cease and some tuples buffered in the input manager may remain stalled indefinitely.

As a simple example, consider a system with two sources, and let B include:

$$B = \begin{pmatrix} (-,0) & (-,5) \\ (-,-) & (-,0) \end{pmatrix}$$

(Wall-clock time components in skew bounds are not relevant to the example.) Let τ be the maximum timestamp seen on S_1 and $\tau - 5$ be the maximum on S_2. Suppose that both streams stop at this time. The strongest heartbeats that can eventually be inferred on S_1 and S_2 are τ and $\tau - 5$ respectively. Thus the overall heartbeat for S_1, S_2 is $\min(\tau, \tau - 5) = \tau - 5$, and all tuples on S_1 with timestamp in the interval $(\tau - 5, \tau]$ cannot be moved to the query processor until at least one stream resumes.

To ensure progress even when all streams have paused, we assume a user-specified timeout such that if no tuple arrives on any stream for $t_{timeout}$ units of time, then the DSMS can assume that any tuple arriving subsequently on any stream will have a timestamp greater than any seen so far. Thus, if no tuple arrives on any stream for time $t_{timeout}$, a heartbeat equal to the maximum timestamp seen so far is emitted for all streams, and all tuples can be moved to the query processor.

The following theorem exactly characterizes the situations in which a timeout is not required.

THEOREM 3.3. *Let $\delta_{ij}^{min} = \min\ \{\delta_{ij} | (t_{ij}, \delta_{ij}) \in B_{ij}\}$. A timeout is not needed iff $\forall\ i, j :\ \delta_{ij}^{min} = 0$.*

PROOF. For any stream S_i, let τ_i^{max} be the maximum timestamp seen on S_i when all streams pause. Suppose $\delta_{ij}^{min} > 0$ for some i, j. Let $\tau_k^{max} \leq \tau_i^{max} - \delta_{ij}^{min}$ for all $k \neq i$. The strongest heartbeat that can be inferred on S_j due to a tuple on S_k $(k \neq i)$ is at most τ_k^{max} (since in any skew bound pair $(t, \delta), \delta \geq 0$). Also, the strongest heartbeat that can be inferred on S_j due to tuples on S_i is $\tau_i^{max} - \delta_{ij}^{min}$. Thus the heartbeat τ_j (and hence the overall heartbeat for $S_1, S_2, \ldots, S_n$) is no greater than $\tau_i^{max} - \delta_{ij}^{min}$. This means that all tuples on S_i with timestamp in the interval $(\tau_i^{max} - \delta_{ij}^{min}, \tau_i^{max}]$ cannot be processed until a stream resumes. Thus a timeout is needed.

Now suppose $\delta_{ij}^{min} = 0$ for all i, j. Let τ^{max} be the maximum timestamp seen when the streams pause, and let it be seen on S_i. Since $\delta_{ij}^{min} = 0$ for all j, eventually a heartbeat of τ^{max} can be inferred on each S_j. Thus a timeout is not needed in this case. □

4. EXAMPLE SYSTEMS

In this section we illustrate our approach by applying it to two example systems. In one system, sources obtain timestamps from a shared global counter. In the other, sources are sensors with deviating clocks.

4.1 Global Counter

The basic setup is the same as shown in Figure 1. Additionally, there is some central source from which sources $\phi_1, \phi_2, \ldots, \phi_n$ obtain tokens or counters and use them to timestamp their output tuples. A new token is requested by ϕ_i only after the present token has been processed, and the central source issues tokens in monotonically increasing order. Let t_i^{min} and t_i^{max} bound the time that ϕ_i can spend processing each token and emitting the corresponding tuple, which includes any delay in obtaining the token from the central source. We assume the bounds on network latency are obtained from network characteristics. The other parameters required for heartbeat generation in this system can be specified as follows:

1. *Skew bound*: The skew bound for sources ϕ_i and ϕ_j $(i \neq j)$ is $B_{ij} = \{(\max(t_j^{max} - t_i^{min}, 0), 0)\}$. A tuple-based skew bound of $(1, 0)$ also can be added to B_{ij}.

   The maximum skew occurs when ϕ_j obtains a token τ_1 at time c followed immediately by ϕ_i obtaining a token $\tau_2 > \tau_1$. Let ϕ_i take its minimum time and emit τ_2 at time $c + t_i^{min}$, while ϕ_j takes its maximum time and emits τ_1 at time $c + t_j^{max}$. If $t_j^{max} > t_i^{min}$ the tuple emitted by ϕ_j has a lower timestamp (τ_1) than the tuple emitted by ϕ_i earlier (τ_2). Also, all subsequent tuples emitted by ϕ_j will have a greater timestamp than τ_2 (because a new token will be requested by ϕ_j). This skew is specified by $(t_j^{max} - t_i^{min}, 0)$. If $t_j^{max} < t_i^{min}$, then ϕ_j can never emit a tuple with lower timestamp than one already emitted by ϕ_i. In this case $(0, 0) \in B_{ij}$.

   In addition, if ϕ_i emits a tuple with timestamp τ_1 at time c and ϕ_j has emitted at least one tuple after time c, all subsequent tuples by ϕ_j will have a timestamp $> \tau_1$. Thus we can infer a tuple-based skew specification $(1, 0) \in B_{ij}$.

2. *Out-of-order generation*: Since each source is emitting in strictly increasing order, $B_{ii} = \{(0, 0)\}$ for all i.

3. *Timeout*: Since the conditions of Theorem 3.3 are satisfied, a timeout is not needed. Intuitively, if all streams pause we will see the latest token given by the central source, which eventually becomes the heartbeat.

This example illustrates the utility of our novel notion of "delayed" guarantees. Consider if we could only specify δ_{ij} corresponding to $t_{ij} = 0$. Suppose ϕ_j obtains a token τ at time c and takes time t_j^{max} to process it. At the same time, all other sources start obtaining tokens and processing them

at their fastest possible rate. When ϕ_j finishes and is ready to emit its tuple, the other sources may be far ahead of it. In time t_j^{max}, ϕ_k ($k \neq j$) can process up to $\lceil t_j^{max}/t_k^{min} \rceil$ tokens. Thus $\delta_{ij} = \sum_{k \neq j} \lceil t_j^{max}/t_k^{min} \rceil$ for all i. Since the values of t_k^{min} may be quite small, even very close to zero, with $t_{ij} = 0$ we get a very weak bound that is likely to delay heartbeats considerably.

4.2 Distributed Sensors

Consider a system in which the sources $\phi_1, \phi_2, \ldots, \phi_n$ are sensors that use local clocks to timestamp tuples. Let us assume the local clocks deviate from a global application clock by only a bounded amount due to periodic synchronization. We do not assume any relationship between wall-clock time and the global application clock. Let d_i be the maximum deviation of source ϕ_i's local clock from the global application clock. We assume, as before, that the bounds on network latency are obtained from network characteristics. The other parameters required for heartbeat generation in this system can be specified as follows:

1. *Skew bound*: The skew bound for sources ϕ_i and ϕ_j ($i \neq j$) is specified by $B_{ij} = \{(0, d_i + d_j)\}$. This bound corresponds to the worst case when both ϕ_i and ϕ_j deviate from the global clock by their respective maximum amounts, ϕ_i is ahead of the global clock, and ϕ_j is behind.

2. *Out-of-order generation*: Whether a particular source can emit tuples out of order depends on how periodic clock synchronization is implemented. If a local clock that has drifted ahead of the global clock resets itself to a smaller value, then the source may emit tuples out of order. Since the clock at source ϕ_i can reset itself by an amount no greater than d_i, we have $B_{ii} = \{(0, d_i)\}$.

3. *Timeout*: Since the conditions of Theorem 3.3 are not satisfied in this case, a timeout $t_{timeout}$ must be specified to ensure progress. However, the timeout is applied only when all streams pause, a highly unlikely event in an application involving a large number of sensors.

   However, suppose we know additionally that the application clock at any source ϕ_j advances at a rate of at least r_j per some unit of wall-clock time. Then we can add to B_{ij} ($\forall\, i, j,\ i \neq j$) the pair $(\lceil (d_i + d_j)/r_j \rceil, 0)$ and to B_{jj} ($\forall\, j$) the pair $(\lceil d_j/r_j \rceil, 0)$. Now a timeout is unnecessary.

5. COMPLEXITY AND SCALABILITY

In this section we address several remaining issues. The time and space complexity of our algorithm is analyzed in Section 5.1. In Section 5.2 we propose implementation techniques that enable our approach to scale to very large numbers of sources. Throughout this section, let k be the maximum number of pairs in any skew bound ($k = \max_{i,j}(|B_{ij}|)$, let t_{max} be the maximum wall-clock time component in any skew bound pair in B ($t_{max} = \max_{i,j}\ \{t_{ij} \mid (t_{ij}, \delta_{ij}) \in B_{ij}\}$). Let L_{max} be the maximum latency bound across sources ($L_{max} = \max_i(L_i)$). Recall that n is the number of streams.

5.1 Time and Space Complexity

We consider the time and space complexity of processing a single arriving stream tuple in the algorithm in Figure 2. For any tuple arrival, we set at most k heartbeats on each stream. Thus the time complexity is $O(kn)$ at each arrival. Recall that $k = \max_{i,j}(|B_{ij}|)$, so it is expected to be very small, usually 1.

The matrix B needs to be stored, thus contributing $O(kn^2)$ to space complexity. However, in most applications (as the examples seen in Section 4), the entries of B are computed from individual values for each source, so only $O(n)$ information is needed. We also require an array for each stream to store the heartbeats. For simplicity the algorithm in Figure 2 assumes an infinite array, but at any time c we do not need heartbeats for any times earlier than c. Also, the algorithm cannot set the heartbeat for any time later than $c + t_{max} + L_{max}$. Thus only $t_{max} + L_{max}$ contiguous array entries are used at any point in time. This gives a total space complexity of $O(kn^2 + n(t_{max} + L_{max}))$.

Notice that the unit of wall-clock time for heartbeats is at our discretion. We can reduce the space complexity by performing heartbeat generation using a more coarse-grained unit of wall-clock time. Suppose our original unit was one clock tick and we increase it to x clock ticks. Then t_{max} and L_{max} both drop by a factor of x and the space complexity reduces accordingly. However, now a heartbeat may be delayed unnecessarily by x wall-clock ticks in the worst case.

Of course these time and space complexities for heartbeat generation are completely unacceptable when n may be very large, as in typical sensor applications. In the next section we discuss techniques for scaling in these environments.

5.2 Scaling for Large Numbers of Sources

Our approach so far does not scale well in number of sources: the time and space complexity is linear in n even if we ignore the space overhead of storing B. The linear complexity arises from needing to generate one heartbeat per stream. One possibility is to generate just one heartbeat for the set of streams $S_1, S_2, \ldots, S_n$ instead of n separate stream-level heartbeats. While this approach may work well in some environments, it prevents us from scheduling parts of query plans independently as outlined in Section 7.

We can adopt an intermediate approach to lower the time and space complexity, and still have some flexibility in scheduling. We aggregate streams into groups, then treat each group as a single stream, maintaining a single heartbeat for it. The granularity at which we aggregate determines the number of groups and can be fixed depending on how much overhead is acceptable. For lowest overhead, we aggregate all streams together into a single group, which results in a single heartbeat for the set of all streams.

The skew bounds for aggregated streams can be derived offline from the skew bounds for the individual streams. Let our original n streams be aggregated into m groups $G_1, G_2, \ldots, G_m$. Given our usual $n \times n$ skew bound matrix B, we wish to arrive at an $m \times m$ matrix $\hat{B}$ with skew bounds for the aggregated streams. Intuitively, entry $\hat{B}_{ab}$ should specify the weakest guarantee provided by the source of any stream in G_a on the source of any stream in G_b. We then use $\hat{B}$ instead of B in our heartbeat generation algorithm.

To arrive at a formal expression for $\hat{B}$, we first need some

notation. Let S be a set of skew bound pairs (t, δ). Define S^+ to be the set S with redundant pairs added so that the strongest guarantee for each time $t \in \{0, 1, \ldots, t_{max}\}$ is mentioned explicitly. For example, if we have (1,2) and (3,1) in S, we add (2,2) to S. Let S' and S'' be two skew bound sets with redundant guarantees added as above. Then we define:

$$S' \sqcup S'' = \{(t, \delta) \mid (t, \delta') \in S', (t, \delta'') \in S'', \delta = \max(\delta', \delta'')\}$$

Intuitively $S' \sqcup S''$ gives, at every time step, the weaker of the guarantees given by either S' or S''. Thus the entries in $\hat{B}$ are given by the expression:

$$\hat{B}_{ab} = \bigsqcup_{S_i \in G_a, S_j \in G_b} (B_{ij})^+$$

Let there be n_a streams in G_a and n_b in G_b. Then we need to combine $n_a n_b$ entries of B to get $\hat{B}_{ab}$, which can be done in $O(kn_a n_b)$ time. Note again that $\hat{B}$ is computed offline. This method applies to the diagonal entries of $\hat{B}$ as well, providing the bound on out-of-order generation within an aggregated stream. The bound on network latency for aggregated stream G_a is simply $L_{G_a} = \max_{S_i \in G_a} L_i$.

By aggregating streams together and maintaining a single heartbeat for them, some heartbeats may be delayed compared to individual stream heartbeats. It is an interesting open problem to decide the best way to partition the n streams into $m < n$ groups such that heartbeat delay is minimized. Intuitively, sources having similar characteristics should be grouped together so that the corresponding streams are likely to have similar heartbeats anyway. The sensor environment is a natural fit for this approach, for example, where a large number of sensors may have identical properties.

6. PARAMETER ESTIMATION

So far we have assumed that the parameter values used by our algorithm can be specified statically in advance. However, obtaining such information may be difficult in reality. Thus, we need an approach in which the DSMS itself estimates the parameter values according to the history of the streams seen so far, and generates heartbeats according to those.

One possible method is to estimate the skew bound matrix B and the bounds on network latency, then apply the algorithm in Figure 2 as is. However, the framework of Section 2 was meant only for ease of parameter specification by providing a separation between the network-dependent and source-dependent parameters. In reality, the skew observed by the DSMS is a combined effect of skew at the sources and network latency. Thus, while estimating the parameters for heartbeat generation, it is not necessary to estimate these two factors separately—we need only estimate the resulting skew at the DSMS.

For this purpose, we define a modified skew bound matrix B' where each entry B'_{ij} $(i \neq j)$ gives the maximum skew between streams S_i and S_j *at the time of arrival* at the DSMS, rather than at the time of emission as given by B_{ij}. Similarly, the diagonal entries of B' give bounds on out-of-order arrival at the DSMS for single streams. As motivated above, the skew bounds in B' incorporate network latency while the skew bounds in the original matrix B do not. Given B', the heartbeat generation algorithm is a slight modification of the one given in Figure 2: Whenever a tuple with timestamp τ arrives on S_i at time c, then for each $(t_{ij}, \delta_{ij}) \in B'_{ij}$, we set $\tau_j[c + t_{ij}] = \tau - \delta_{ij}$.

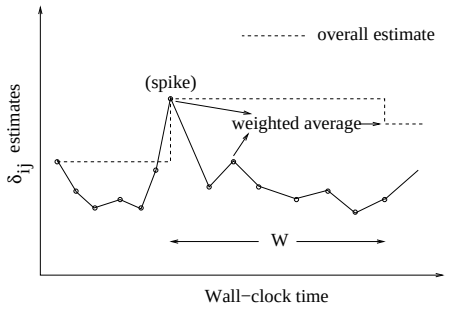

Figure 3: Adaptive estimation of skew bounds: a schematic view

Of course we still need a user-defined timeout to ensure progress in the case when all streams pause, as considered in Section 3.3. We next consider how B' can be estimated based on the stream data seen so far.

6.1 Estimating B'

Now let us discuss how B' is estimated. We first fix a bound t_{max} on the maximum time into the future at which we would like to provide a guarantee. Consider any $i, j \in \{1, 2, \ldots, n\}$. For each $t \in \{0, 1, \ldots, t_{max}\}$, we estimate $\delta_{ij}[t]$ such that $(t, \delta_{ij}[t]) \in B'_{ij}$. We start out with an initial estimate of $\delta_{ij}[t] = 0$. Let $\tau_i^{max}[c]$ denote the maximum application timestamp seen on S_i up to time c. If a tuple with timestamp τ arrives on S_j at time c', then:

$$\delta_{ij}[t] \geq \max(\tau_i^{max}[c' - t] - \tau + 1, 0)$$

We simply maintain the maximum of such estimates over time as our estimate of $\delta_{ij}[t]$.

For computing the above estimate, we need to store $\tau_i^{max}[c]$ for the last t_{max} time units. This technique operates according to the maximum skew seen so far. After initial settings and stabilization, heartbeats are generated too soon only when we see skew higher than has ever been seen before. We will consider a more adaptive approach in the next section.

Let us consider the time and space complexity of estimating B'. When a tuple arrives on S_j, we compute new estimates for each $\delta_{ij}[t]$, for a time complexity of $O(t_{max}n)$ per arrival. We need $O(t_{max}n^2)$ for storing B' and $O(t_{max}n)$ space for storing the τ_i^{max}'s for the past t_{max} time units in a wraparound fashion. These complexities are essentially the same as those required for heartbeat generation, thus parameter estimation does not add any asymptotic overhead. If n is large, we can again apply the idea of stream aggregation discussed in Section 5.2. We can also reduce the complexity by using a more coarse-grained unit of wall-clock time, as discussed in Section 5.1.

6.2 An Adaptive Approach

So far we have considered heartbeat generation based on prespecified or estimated parameters capturing the worst case, e.g., the maximum possible skew or latency. In some applications it may be preferable to generate heartbeats earlier, even if some inaccuracy results, particularly when the

Estimate$\delta(W, \alpha)$
1. initialize $num_{ij}[t] = 0, \delta_{ij}^{win}[t] = \delta_{ij}[t] = 0$
   $\forall i, j \in \{1, 2, \ldots, n\}, t \in \{0, 1, \ldots, t_{max}\}$
2. Whenever a tuple with timestamp τ arrives on stream S_j
   at time c:
3. for $i = 1$ to n do
4. for $t = 0$ to t_{max} do
5. $num_{ij}[t] = num_{ij}[t] + 1$
6. $\delta_{ij}^{est} = \max(\tau_i^{max}[c - t] - \tau + 1, 0)$
7. if $(\delta_{ij}^{est} > \delta_{ij}[t])$
8. $\delta_{ij}[t] = \delta_{ij}^{est}, \delta_{ij}^{win}[t] = 0, num_{ij}[t] = 0$
9. else
10. if $(\delta_{ij}^{est} > \delta_{ij}^{win}[t])$
11. $\delta_{ij}^{win}[t] = \delta_{ij}^{est}$
12. if $(num_{ij}[t] > W)$
13. $\delta_{ij}[t] = \alpha\delta_{ij}[t] + (1 - \alpha)\delta_{ij}^{win}[t]$
14. $\delta_{ij}^{win}[t] = 0$, $num_{ij}[t] = 0$

Figure 4: Algorithm for adaptive estimation of skew bounds

worst case occurs as relatively uncommon spikes (a common phenomenon in network latency, for example [5]).

To address these environments, we consider *adaptive* estimation of skew bounds. The estimated bounds should reflect the skew the DSMS is currently seeing rather than the worst it has ever seen. We modify our method of estimating $\delta_{ij}[t]$ so that recent estimates are given more importance, and if a spike was seen long ago, its effect on our current estimate is minimized.

The algorithm for estimating $\delta_{ij}[t]$ adaptively is given in Figure 4 and depicted graphically in Figure 3. We maintain $\delta_{ij}[t]$ as the overall estimate, and $\delta_{ij}^{win}[t]$ which is the maximum over a window consisting of the last $num_{ij}[t]$ estimates. If the new estimate (δ_{ij}^{est}) is larger than the current estimate, $\delta_{ij}[t]$ is immediately updated to this new estimate and the window is reset. If $num_{ij}[t]$ exceeds a threshold W, it signifies that many tuples have been observed without our estimate being exceeded, an indication that our estimate of $\delta_{ij}[t]$ may be too high. Hence we lower our estimate by taking a weighted average of $\delta_{ij}[t]$ with $\delta_{ij}^{win}[t]$: $\alpha\delta_{ij}[t] + (1 - \alpha)\delta_{ij}^{win}[t]$.

Parameters W and α are a measure of the adaptivity of our algorithm and can be adjusted to trade off latency against accuracy.

- W: How many low estimates of skew should be observed before we lower our estimate of $\delta_{ij}[t]$. $W = \infty$ corresponds to the nonadaptive algorithm of the previous section. W can also be time-based instead of tuple-based.

- α: Weight given to the previous estimate when we are lowering our estimate of $\delta_{ij}[t]$. If $\alpha = 0$ the estimate is updated immediately to the new, lower value. $\alpha = 1$ yields the nonadaptive algorithm.

We can also incorporate easily the notion of "adaptive adaptivity". We observe how often our new estimate δ_{ij}^{est} exceeds $\delta_{ij}[t]$, and if this rate is higher than a threshold we increase W. Similarly, if our estimate of $\delta_{ij}[t]$ has been continuously decreasing over many windows, we decrease W. Finally, notice that we use a rolling window instead of a sliding window so that the state required by the algorithm for estimating a single $\delta_{ij}[t]$ is reduced from $O(W)$ to $O(1)$. Thus, adaptivity does not increase overall time and space complexity. Detailed experimentation with the adaptive approach is left for future work.

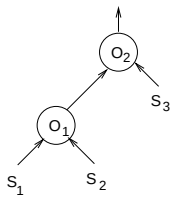

Figure 5: Example query plan

7. HEARTBEATS IN QUERY PLANS

At Stanford we are developing a general-purpose DSMS [19]. In this section we show how heartbeats can be incorporated deeply into our query processor to decrease latency while preserving semantics.

When a continuous query is registered with the DSMS, it is compiled into a query plan for execution. A query plan consists of *operators* and *inter-operator queues*. Operators are similar to traditional DBMS operators: Each operator reads tuples from one or more input queues, processes them according to operator semantics, and writes to a single output queue. Inter-operator queues connect various operators and define the paths along which tuples flow as they are being processed.

7.1 Query-Level Heartbeats

The most straightforward way to incorporate heartbeats is the method we have suggested so far in the paper: Let τ_i denote the heartbeat on stream S_i. We maintain a *query-level heartbeat* $\tau = \min(\tau_1, \tau_2, \ldots, \tau_n)$. All tuples with timestamp greater than τ are buffered in the input manager. When the query-level heartbeat τ increases, any buffered tuples whose timestamps are now $\leq \tau$ are presented to the query processor in timestamp order. (Recall Figure 1.)

However, this naïve method does not take into account the structure of the query plan, or the fact that we have per-stream heartbeats. Consider the query plan in Figure 5 and suppose S_1 and S_2 are almost synchronized with each other while S_3 is lagging far behind in terms of application time, i.e. $\tau_1 \approx \tau_2 \gg \tau_3$. Operator O_1 can process tuples independent of τ_3 (up to timestamp $\min(\tau_1, \tau_2)$) without violating the ordering requirement. However, this naïve scheme of using a query-level heartbeat will not permit O_1 to process tuples with timestamp $> \tau_3$ although $\min(\tau_1, \tau_2) \gg \tau_3$. Thus the system may be idle even when O_1 could be scheduled. Not allowing O_1 to be scheduled as soon as it is ready can cause two problems:

- It can reduce overall processor utilization in the case when multiple queries are being executed in the system [9].

- It can increase latency in producing the result tuples: When τ_3 "catches up", both O_1 and O_2 need to run,

while if O_1 had been scheduled earlier, only O_2 needs to execute to produce result tuples.

7.2 Operator-Level Heartbeats

The scenario in Section 7.1 suggests that heartbeats should be handled at the operator level rather than at the query level. Assume $\tau_{O_1}, \tau_{O_2}, \ldots, \tau_{O_m}$ are heartbeats on the input streams of operator O. We maintain an *operator-level heartbeat* $\tau_O = \min(\tau_{O_1}, \tau_{O_2}, \ldots, \tau_{O_m})$. Conceptually, each operator can now be thought of as having its own input manager. Similar to the query-level case, all tuples on the operator input streams with timestamp greater than τ_O are buffered in the operator's input manager, while those with timestamp $\leq \tau_O$ are presented to the operator in timestamp order for processing.

For this scheme, we must have heartbeats on inter-operator queues as well as on input streams. Consider a leaf operator O with input streams $S_1, S_2, \ldots, S_m$. Operator O can process tuples up to timestamp $\min(\tau_1, \tau_2, \ldots, \tau_m)$. Once it has processed all tuples up to timestamp τ, it can emit a heartbeat τ on its output stream. The same approach propagates up the plan tree.

With this new scheme consider the execution of the plan in Figure 5. Operator O_1 will have an operator-level heartbeat of $\min(\tau_1, \tau_2)$. Thus O_1 can be scheduled independent τ_3, thereby remedying the problem of Section 7.1.

7.3 Latency-Memory Tradeoff

Operator-level heartbeats aim to reduce latency by allowing operators to be scheduled earlier, but they may result in higher memory consumption as illustrated by the following example. Consider again the query plan of Figure 5 with operator-level heartbeats and $\tau_1 \approx \tau_2 \gg \tau_3$. Further, suppose that O_1 is "proliferative", i.e., the size of the result it produces is greater than the size of the input it consumes. As mentioned in Section 7.1, with operator-level heartbeats, the intermediate results produced on O_1's output queue cannot be used until τ_3 catches up. With a query-level heartbeat, O_1 would not proliferate tuples until O_3's heartbeat was ready to process them. Thus, with query-level heartbeats, the peak memory requirement would have been lower, at the cost of higher latency.

Some of these problems may be alleviated by using a sophisticated memory-aware scheduling algorithm such as *Chain* [2]. The detailed relationship between Chain and operator-level heartbeats is a subject of future work. However, if a simple non-memory-aware scheduling policy such as round-robin or FIFO is being used, we give a scheme for operator-level heartbeats that reduces latency over using a query-level heartbeat while not increasing memory usage. The general idea is to not always use the maximum operator-level heartbeat possible at each operator, but to artificially delay the heartbeats so as to keep the memory consumption in check. For this, each operator O determines its lowest ancestor A in the query-plan tree, such that the combined chain of operators from O to A is nonproliferative (i.e., selective). Then, instead of O using its own operator-level heartbeat, it uses A's heartbeat instead. Intuitively, the effect is that O does not produce tuples until they are ready to propagate up the tree at least as high as A.

7.4 Implementing a Memory-Aware Strategy

The challenge in implementing the memory-aware strategy outlined above, is to make it local, i.e., after some initial global configuration of the query plan, each operator should be able to decide its maximum heartbeat independently, without having to communicate with other operators. This is desirable for efficiency and ease of implementation.

AssignLabels()
Input: A query plan tree with memory links added
Output: A label R_i for each operator O_i
1. AssignLabelsSubtree(O_{root})

AssignLabelsSubtree(O_i)
1. if $\exists$ a memory-link from within T_i to some ancestor of O_i
2. $\quad O_{parent}$ = parent of O_i
3. $\quad R_i = R_{parent}$
4. else
5. $\quad R_i = U_i$
6. for all O_j such that O_j is a child of O_i do
7. $\quad$ AssignLabelsSubtree(O_j)

Figure 6: Assigning labels to obtain operator heartbeats locally

Suppose we are given, for each operator O, its lowest level ancestor A_O such that the entire chain of operators from O to A_O is selective. It is a common approach to gather such selectivity information at runtime (see [2] for a method to characterize the selectivity of multi-input stream operators). We want O to use a heartbeat no greater than that used by A_O. Also, since A_O is an ancestor of O, its heartbeat can be no greater than that of O. This implies that the heartbeat of each operator along the path from O to A_O must be equal.

To allow local computation of such heartbeats, we assign a label R_i consisting of a set of streams ($R_i \subseteq \{S_1, S_2, \ldots, S_n\}$) to each operator O_i . The maximum heartbeat to be used by O_i is obtained locally as $\tau_{O_i} = \min_j\{\tau_j | S_j \in R_i\}$. Recall that τ_j is the heartbeat on stream S_j.

Let O_{root} be the output node which is also the root of the query plan tree. Let T_i be the subtree of the plan rooted at operator O_i and U_i be the set of streams ($\subseteq \{S_1, S_2, \ldots, S_n\}$) which are input in the subtree T_i. If $R_i = U_i$, O_i uses its maximum possible operator-level heartbeat. But in general $R_i \supseteq U_i$ and the operator uses a lower heartbeat.

The algorithm for assigning these labels is given in Figure 6. First, we add imaginary links called "memory-links" to the query plan which join each operator O_i to the corresponding A_{O_i}. We proceed in a top-down fashion. Since the output node is assumed to have selectivity 0, it can always use its maximum heartbeat. Hence we start by setting $R_{root} = U_{root} = \{S_1, S_2, \ldots, S_n\}$.

For any other operator O_i, if there is no memory-link from within T_i to any ancestor of O_i, it means that the entire subtree T_i can be scheduled independently of the other parts of the plan without memory concerns. Thus we allow O_i to use its maximum heartbeat by setting $R_i = U_i$.

However, if there is such a memory link from within T_i to some ancestor $O_{ancestor}$ of O_i, we use the observation that in the presence of a memory-link, the heartbeats used by each operator along the corresponding path must be equal.

Thus, in this case we set $R_i = R_{parent}$ where O_{parent} is the parent of O_i.

Consider the execution of the above algorithm on the sample plan of Figure 5. Let O_1 be proliferative. Thus there is a memory link from O_1 to O_2. First O_2 is assigned a label $R_2 = \{S_1, S_2, S_3\}$. Then, due to the memory-link, O_1 is assigned the same label as O_2. Thus, in this case, it reduces to the query-level heartbeat execution strategy given in Section 7.1.

7.5 Order-Insensitivity

We have assumed that input stream tuples must be ordered before being processed by an operator. Operators such as window-based constructs require ordered input streams. However some operators, such as selection and projection, are *order-insensitive*: they can process tuples in any order without violating semantics. Operator-level heartbeats are not necessary for order-insensitive operators. We can use this fact to move tuples through operators earlier, potentially decreasing latency. Only when an entire path from leaf to root in a query plan consists of order-insensitive operators can we avoid eventually ordering the tuples. Whether we prefer to order sooner rather than later most likely depends on selectivity of operators and overall memory use; a formal investigation of this topic is planned for future work.

8. RELATED WORK

The problem of application-defined time and stream synchronization surfaces in nearly any continuous query processor that handles externally generated input streams and has a sufficiently rich query language. A number of ways of specifying skew bounds have been proposed in literature. For example, in the *Aurora* project [21], operators have a *slack* parameter to deal with out-of-order streams. Essentially, the slack parameter instructs its operator to wait a certain period of time before closing each window. Thus, a slack of t units of wall-clock time can be thought of as a skew bound of $(t, 1)$ in our scheme. Operators also have a *timeout* associated with them that is somewhat similar to our timeout (Section 3.3). The fact that in certain cases a timeout is not needed (as in our Section 4.1 example) has not been considered. Aurora's timeout and slack parameters together cannot accurately capture either of the scenarios covered in Section 4. Overall, our scheme is more general and expressive than that of Aurora. It allows skew bounds to be accurately specified, thus enabling the strongest possible guarantees to be placed. Besides, skew is clearly a stream-level property rather than an operator-level one.

Our framework for specifying skew bounds can also express other types of stream constraints that have been proposed in the context of memory-efficient continuous query processing [3]. For example a tuple-based k-ordered arrival constraint is equivalent to a tuple-based skew bound of (k,0).

In *Niagara* [7], the proposed solution is based on *punctuations* [20]. Punctuations define arbitrary predicates over streams. Thus, heartbeats can be thought of as special types of punctuations. However, the focus there has been more on using, rather than generating, punctuations. Punctuations are either assumed to be provided within streams, or if generated, are based on simple notions of slack as in Aurora. A characterization of queries that can benefit from punctuations has been provided in [20].

In *TelegraphCQ* [12], time is considered as a partial order, with the result of algebraic operators being defined only for timestamps comparable under this partial order [6]. A partial order does not alleviate the requirement for synchronizing streams, since the comparable timestamps still must be processed in order, yielding a problem no easier than in the totally ordered case. To the best of our knowledge, this problem has been recognized but not yet addressed in the Telegraph project.

Gigascope discusses the importance of time in data stream query processing [8]. They introduce application-generated timestamps as special attributes that can be used for windows, grouping, and operator unblocking. However, they do not discuss the kinds of issues we address in this paper: handling skew between distributed sources, latency, processing in the presence of "paused" streams, heartbeat generation, and using heartbeats in query plans.

In the context of distributed systems, Lamport [13] demonstrated how to impose a consistent total order on events using only distributed unsynchronized clocks. This work is not directly relevant to ours, since in our system we assume that the application-defined timestamps already provide a consistent total order according to which queries have to be processed.

In *sequence* [16] and *temporal* [17] databases, there is no notion of tuple arrival or reordering—time is incorporated as part of the static state of the data. However, note that the notions of *valid* and *transaction* time in temporal databases are somewhat similar to our notions of application time and wall-clock time respectively. In *real-time* databases [15], data has a certain validity period within which it must be processed. However this period is measured in terms of wall-clock time and there is no separate notion analogous to application time. Consequently there are no issues related to out-of-order arrival or synchronization between different sources.

9. CONCLUSIONS

We have proposed flexible and efficient techniques to handle application-defined time in a DSMS. The main obstacles we need to address are reordered streams, skew between streams, and "paused" streams. We introduce *heartbeats* as the primary method to deal with these obstacles, and focus on how heartbeats can be generated when the sources themselves do not provide any. We identify the parameters needed for heartbeat generation and give a novel framework in which they can be specified. Our framework is expressive enough to precisely capture a variety of scenarios. It also provides a clean separation between parameters that depend only on sources and those that depend only on the network. When parameter values are not specified by the application, we provide conservative as well as adaptive methods to estimate them. We provide techniques for scaling our heartbeat generation algorithm to a large number of streams. Finally, we provide initial ideas on how heartbeats can be used in query plans to decrease latency while still preserving semantics. As discussed in Section 7, more work is to be done on incorporating heartbeats into query plans in the best possible way, particularly as heartbeats interact with operator scheduling. We are pursuing this line of research, as well as implementing the approach proposed in this paper as part of the STREAM prototype DSMS under development at Stanford.

Acknowledgments

We are especially grateful to Bruce Lindsay, who first suggested that heartbeats might form the basis of a solution to the time problem in data stream query processing. We are also grateful to the entire STREAM group at Stanford for ongoing valuable discussions on time, heartbeats, query plans, and related issues. Finally, we thank Chris Olston for numerous excellent suggestions on an earlier draft of this paper.

10. REFERENCES

[1] A. Arasu, S. Babu, and J. Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *Proc. of the Ninth Intl. Conf. on Database Programming Languages*, September 2003.

[2] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator scheduling for memory minimization in data stream systems. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 253–264, June 2003.

[3] S. Babu, U. Srivastava, and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. Technical report, Stanford University Database Group, November 2002. Available at `http://dbpubs.stanford.edu/pub/2002-52`.

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

[5] J. Bolot. End-to-end packet delay and loss behavior in the internet. In *Proc. of the 1993 ACM SIGCOMM*, pages 289–298, September 1993.

[6] S. Chandrasekaran, S. Krishnamurthy, et al. Windows Explained, Windows Expressed. Available at `http://www.cs.berkeley.edu/~sirish/research/streaquel.pdf`.

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

[8] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. The Gigascope Stream Database. *IEEE Data Engineering Bulletin*, 26(1):27–32, March 2003.

[9] M. Garey, R. Graham, and D. Johnson. Performance guarantees for scheduling algorithms. *Operations Research*, 26(1):3–21, January 1978.

[10] J. Gehrke. Special issue on data stream processing. *IEEE Data Engineering Bulletin*, 26(1), March 2003.

[11] L. Golab and M. Ozsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, June 2003.

[12] S. Krishnamurthy, S. Chandrasekaran, et al. TelegraphCQ: An Architectural Status Report. *IEEE Data Engineering Bulletin*, 26(1):11–18, March 2003.

[13] L. Lamport. Time, clocks, and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, July 1978.

[14] A. Mukherjee. On the dynamics and significance of low-frequency components of internet load. *Internetworking: Research and Experience*, 5(4):163–205, December 1994.

[15] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1(2):199–226, 1993.

[16] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *Proc. of the 1994 ACM SIGMOD Intl. Conf. on Management of Data*, pages 430–441, May 1994.

[17] R. Snodgrass and I. Ahn. A taxonomy of time in databases. In *Proc. of the 1985 ACM SIGMOD Intl. Conf. on Management of Data*, pages 236–245, 1985.

[18] SQR – A Stream Query Repository. http://www-db.stanford.edu/stream/sqr.

[19] The STREAM Group. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, March 2003.

[20] P. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):1–14, June 2003.

[21] S. Zdonik, M. Stonebraker, et al. The Aurora and Medusa Projects. *IEEE Data Engineering Bulletin*, 26(1):3–10, March 2003.