Physically Independent Stream Merging

Badrish Chandramouli[#], David Maier[*], Jonathan Goldstein[$]

[#]*Microsoft Research, Redmond, Washington, USA*

[*]*Portland State University, Portland, Oregon, USA*

[$]*Microsoft Corporation, Redmond, Washington, USA*

badrishc@microsoft.com, maier@cs.pdx.edu, jongold@microsoft.com

Abstract—**Several desired capabilities in a *data stream management system* (*DSMS*), such as query-plan switching and high availability, can be considerably simplified using a facility to merge equivalent data streams. One can logically view a data stream as a temporal table of events, each associated with a *lifetime* (time interval) over which the event contributes to output. In many applications, the "same" logical stream may present itself physically in multiple physical forms, for example, due to disorder arising during transmission or from combining multiple sources; and modifications or deletions of earlier events. Merging such streams correctly is challenging when the streams may differ physically in timing, order, and composition. This paper introduces a new stream operator called *Logical Merge* (*LMerge*) that takes multiple logically consistent streams as input and outputs a single stream that is compatible with all the inputs. LMerge can handle the dynamic attachment and detachment of input streams. We present a range of algorithms for LMerge that can exploit compile-time stream properties for efficiency. Experiments with StreamInsight, a commercial DSMS, show that LMerge is sometimes orders-of-magnitude more efficient than enforcing determinism on inputs, and that there is benefit to using specialized algorithms when stream variability is limited. We also show that LMerge and its extensions can provide performance benefits in several real-world applications.**

I. Introduction

A *data stream management system* (*DSMS*) [6, 11, 12, 13, 22, 23, 25] supports long-running *continuous queries* (*CQs*) in real time over streams of incoming data. We encounter many scenarios where we need to combine multiple copies of a data stream into a single output stream, for reliability, availability, and performance This problem has a trivial solution if all the input streams present the same elements in exactly the same order – just keep a count on each input, and let the output follow the stream with the largest count. In real DSMSs, however, the problem is not so simple:

1) ***Disorder***: Data streams from a source can get disordered arbitrarily during transmission to a destination where a CQ executes. Consider a scenario where CQs monitor measurement data collected from distributed machines located in geographically dispersed data-centers. Network congestion and other delays can cause data to arrive out-of-order at a destination. If we have multiple destinations for the same data, the degree and nature of disorder can be different at each location. Further, when we gather data from multiple sources (e.g., machines in the data-center example) into a single stream using a Union operator, the result can be disordered even if each input stream arrives in order.

One could buffer incoming tuples in order to eliminate disorder, but doing so can introduce significant query latency and incur high memory overhead. A CQ often contains data-reducing operators, such as aggregation and sampling, and memory needs are minimized if we can move stream elements through the query to such operators without ordering them [7].

2) ***Revisions***: Some scenarios require later stream elements that can modify or cancel earlier elements [10]. Streaming sources often have to contend with noise or data entry errors; for example, commercial stock ticker feeds issue revision tuples to amend previously issued tuples. Sources of data may also wish to completely modify earlier events to replace an earlier value, for e.g., with a more accurate value when performing *online aggregation* [24]. In addition to revisions at data sources, the DSMS engine may itself, for performance and latency reasons, let stream elements flow out of order, and allow operators to provide early answers with possible later revisions [10, 22]. *Progress markers* (such as heartbeats [6], punctuation [1, 2], and CTIs [22]) are used to constrain future elements and avoid arbitrary disorder.

Continuing our data-center example, suppose we are interested in tracking successful OS process executions on each machine. Here, we model each process as an event, with a lifetime corresponding to the process lifetime. When a process starts executing, we do not know its precise end-time a priori. The source may not wish to incur the latency of waiting for the process to end before sending the event, and therefore may issue an initial event with the process-start time (as an insert [4, 12, 22], I-stream [13], or positive tuple [11]). It may subsequently revise that event to provide the process-end time or cancel the event if the process is aborted (with a revision [4, 10, 22], D-stream [13], or negative tuple [11]). Further, consider a CQ that produces a running aggregate of successful process counts. A conservative aggregate operator may wait for a process event to end (or "finalize") before updating the count. An aggressive operator may instead reduce latency and memory usage by emitting an updated count as soon as it sees a process start event, and adjusting the result when the process event is later revised or canceled.

3) ***Processing Variations and Non-Determinism:*** As in databases, CQs are usually expressed declaratively [23] and can have many physical plans. For example, a temporal join of three streams A, B, and C can be processed using two-way joins as $A \bowtie (B \bowtie C)$, $B \bowtie (A \bowtie C)$, etc. or using one three-way join operator. Further, the same operator may use different algorithms – e.g., an aggregate operator may be aggressive or conservative as discussed earlier. A DSMS

operator may, under different run-time conditions, produce different physical presentations for the same underlying logical stream. For example, a multi-input operator such as join, union, or set-difference can produce a different sequence of output elements in two identical copies of a CQ, due to differences in the relative arrival of input events. In summary, output (and intermediate) streams from equivalent plans, while semantically identical, can physically look quite different.

A. Logical and Physical Streams

The above characteristics of real-world streams imply that copies of logically identical streams may look quite different at run time. In order to formalize the concept of "identical streams" in the presence of issues as described above, researchers have proposed separating the notions of physical and logical streams [4, 5, 11, 21, 22]. In general, a logical stream can be viewed as a *temporal database* (*TDB*) that consists of a set of *events*, each associated with a lifetime (interval in application time) and a payload. The lifetime indicates a period of time over which the event contributes to output. A physical stream, on the other hand, is a sequence of *stream elements* that can be *reconstituted* into a TDB instance. Such a distinction between physical and logical streams is observed (either implicitly or explicitly) in many DSMSs, both in academia [4, 5, 6, 7, 10, 11] and industry [22, 25].

Physically divergent streams can be logically equivalent (i.e., have the same TDB), as the following example demonstrates.

Example 1 (Logical/Physical Streams): Consider one possible model of physical streams, with three types of stream elements:

- *a(value, start, end)*, that adds a new event with *value* as payload and duration from *start* to *end*;
- *m(value, start, newEnd),* that modifies an existing event with a given *value* and *start* to have a new end time; and
- *f(time),* that finalizes (freezes from further modifications) every event whose current end is earlier than *time*.

Table 1 (left) shows two physical streams (Phy1 and Phy2), that are different in terms of ordering, event finalization, and lifetime changes. The rows of this table represent increasing instants of system time. These physical streams logically correspond to the same TDB shown in Table 1 (right). Note that prefixes of the two physical streams are not always logically equivalent, but are compatible, i.e., they can still become equivalent in the future.

TABLE I
PHYSICAL AND LOGICAL STREAMS

Phy1	Phy2
	a(A, 6, 7)
	a(B, 8, 15)
a(B, 8, ∞)	m(A,6,12)
a(A, 6, 12)	
m(B, 8, 10)	m(B, 8, 10)
f(11)	
f(∞)	f(∞)

Two Physical Streams

Payload	Interval
A	[6, 12)
B	[8, 10)

Equivalent Logical TDB

B. Logical Merge

Multiple scenarios (see Section II) require combining equivalent logical streams, either temporarily or indefinitely. However, merging multiple streams cleanly – with no loss or duplication of events – is challenging when the streams can differ physically in timing, order, and composition. This paper introduces the *Logical Merge* (*LMerge*) operator that provides logically equivalent output over physically diverse input streams. Some key issues are:

1) ***Disorder and Revisions***: The presence of disorder and revisions means that the final content of a stream event may be arrived at differently in two streams (e.g., streams Phy1 and Phy2 in Example 1), making the design of correct LMerge algorithms challenging. Note that simply choosing one of the input streams to follow can prevent the timely output of events that another input stream has already produced, and cause correctness issues if the chosen input fails, or incur memory overhead (discussed below).

2) ***Punctuation***: LMerge algorithms must be careful when propagating progress markers (punctuation), so that they can stay consistent with future updates on the input streams. We use Example 1 to illustrate that this problem is non-trivial. Assume that our LMerge operator has chosen to propagate elements a(A, 6, 7) and a(B, 8, 15) from Phy2 to its output. When it then sees f(11) from stream Phy1, this element cannot immediately be propagated to the output because: (1) it would "freeze" payload A to have lifetime of [6, 7) which cannot later be adjusted to end at 12; (2) it would freeze all end times earlier than 11, which would prevent later adjustment of the end time of payload B down to 10.

3) ***Memory***: We want to minimize the state that LMerge maintains for correct operation. For example, simply forcing the output to follow one arbitrary stream can result in significant buffering of events from other input streams to LMerge, if the chosen input lags behind the others (to handle the case where the chosen stream detaches or fails). Even if we follow the "fastest" input stream, the possibility of disorder and revisions (even if not actually present in the stream) exacerbates memory overhead, as we need to track what has been output from the different inputs. Further, buffering events per input without sharing can waste significant memory.

4) ***Failures***: Individual input streams can detach or re-attach to LMerge during runtime, e.g., due to machine failures or query plan migration from one virtual machine to another in a Cloud setting. The addition and removal of streams must be carried out carefully to avoid repeating past elements or omitting elements by advancing the output too soon. Consider an input stream that detaches and then re-attaches because its query instance fails and restarts. The new stream might miss some events present on the other inputs, or re-produce prior events because it reprocesses some data. LMerge must deal with such gaps and duplications. Interestingly, the trivial counting merge outlined earlier for simple streams does not work correctly when failures exist.

5) ***Stream Properties***: A fully general LMerge—that can handle unconstrained inputs—can be demanding of memory

and CPU. We want to take advantage of enforced or deduced stream properties, to allow optimized LMerge algorithms. For example, a data source may guarantee that it produces events in order. If a stream with non-decreasing timestamps passes through an aggregate operator (e.g., counting OS processes), we can infer that the output has strictly increasing timestamps. If the aggregation is *grouped* (e.g., performed for each machine ID), we can infer that the combination (payload, timestamp) is unique in the output stream. The static inference of such properties can significantly reduce the complexity and overhead of LMerge.

6) ***Stream Chattiness***: LMerge needs to select policies that balance responsiveness of output against "chattiness" – the need to issue additional output elements to modify previous elements. We provide an example of this last point.

Example 2 (Stream Chattiness): Recall the element types introduced in Example 1. Table 2 below shows two input streams, In1 and In2, and three alternative output streams. As before, the rows of this table represent increasing instants of system time. Out1 is the most aggressive, propagating every change from the inputs as it is seen. Out2 is more conservative, delaying elements until it knows they are final. It thus produces fewer elements than Out1, but produces them later, in general. Out3 is between the two. It outputs the first element it sees with a given payload and start, but saves any modifications until they are known to be final.

TABLE II
EXAMPLE INPUT AND OUTPUT STREAMS

In1	In2	Out1	Out2	Out3
a(A, 6, 10)		a(A, 6, 10)		a(A, 6, 10)
	a(A, 6, 12)	m(A, 6, 12)		
	a(B, 7, 14)	a(B, 7, 14)		a(B, 7, 14)
m(A, 6, 15)		m(A, 6, 15)		
	m(A, 6, 15)			
	f(16)	f(16)	a(A, 6, 15) a(B, 7, 14) f(16)	m(A, 6, 15) f(16)

C. Contributions

This work makes the following contributions.

- We characterize LMerge in a way that applies to many DSMSs, dealing with variations in stream semantics and representation. We formalize the requirements for LMerge output to correctly track its inputs, and propose alternative output policies that meet those requirements (Sec. III).
- We present efficient algorithms for LMerge under different assumptions on input stream properties, and discuss how such properties may be derived from query plans (Sec. IV). We also discuss policy choices for LMerge, handling missing elements, and for attaching and detaching streams (Sec. V).
- We implemented our LMerge algorithms in Microsoft StreamInsight [22], a commercial DSMS, and report their performance relative to different stream characteristics. We further show that a more general LMerge algorithm can have orders-of-magnitude better memory, latency, and throughput features than the strategy of enforcing input stream properties and using a more specialized algorithm (Sec. VI).
- We discuss several applications where LMerge as a building block adds significant value – high availability, fast availability with dynamic plan selection, and query jumpstart or cutover. Experiments show that LMerge can provide significant benefits to such applications, by reducing stream-rate variability and increasing throughput (Secs. II and VI).
- We propose a general scheme for "fast-forwarding" slower CQ plans under LMerge using *feedback signals,* and show that such feedback signals can improve performance by several factors over regular LMerge (Secs. II, V, and VI).

II. APPLICATIONS OF LOGICAL MERGE

The LMerge operator is fully composable with existing operators, and does not require modifications to the DSMS, except if feedback is employed. LMerge opens up convenient solutions to several important stream problems, and enables seamless adaptive CQ processing. Unlike a database, DSMS CQs can last for days or weeks. Tasks such as recovery, re-optimization, and load balancing are easier if individual queries are short lived: re-run a failed query from scratch, re-plan it between executions, launch new queries on a less-loaded node. However, providing these capabilities on queries while they continue to execute is harder.

1) High Availability: Consider providing *high availability* (*HA*) for a continuous stream query that involves a window of, say, 24-hour duration. Simply restarting such a query on failure requires a day for it to "spin-up" and start delivering correct answers. Avoiding such an outage means having redundant copies of the query running and being able to pull results from whichever one or ones have not failed (and to connect up a new copy of the query once it has spun up). We can achieve resilience against *n-1* simultaneous failures by instantiating *n* copies of a query on different machines, feeding into an LMerge operator located at the consumer. LMerge provides a steady stream of output events as long as at least one copy of the query is active. As LMerge is a composable operator, we can also achieve resiliency on a query-fragment level by deploying a hierarchy of LMerge operators – one for each replicated query fragment.

2) Fast Availability: There is also a need for "fast availability" for queries – obtaining output results as soon as possible. Using LMerge to combine (1) identical copies of a query running on machines with independent processor or network resources; or (2) different but semantically identical plans that respond differently to shifts in data distributions, allows answers to be reported from whichever copy is performing better at a given instant. Interestingly, if we need to run multiple copies anyway for HA, we may choose to run different plans to also get fast availability.

In Section VI, we show how LMerge can smoothly switch between streams that experience temporary congestion (due to network or CPU contention) in order to maintain nearly steady

throughput. Section VI also demonstrates how LMerge can "smooth out" variability in stream rate, which may arise because of load fluctuations, scheduling differences, and queuing delays.

3) Plan Fast-Forward: When LMerge is used to combine plans, one plan might lag behind the rest during periods when it is suboptimal or when the machine it runs on suffers resource contention. The work such a plan performs is mostly wasted. It is beneficial if such work can be avoided and that plan can catch up with the rest. In Section V-D, we introduce *feedback signals*, and show how LMerge can leverage such signals to "fast-forward" slower plans and avoid unnecessary work. Section 6 shows that such a technique can provide several times better throughput than running a single plan or performing LMerge without feedback.

4) Query Jumpstart: Another use of LMerge is to aid the process of "jumpstarting" query execution (e.g., in Cloud settings). Stream queries often hold long-lived elements as part of their internal state. In our process-monitoring example, a join or aggregate operator might hold elements for all active processes, including ones that have been running for days or weeks. If we spin up such a query using only current events in the real-time stream, it may take an extended period for the query to rebuild its state (or even be impossible). We may instead wish to "seed" query state using, for example, checkpoint information stored on disk or provided by a running copy of the query. LMerge can be used to seamlessly merge such state with real-time streams in order to get the query operational sooner.

5) Query Cutover: LMerge is also useful in "cutting over" from one query instance to a newly instantiated one with a possibly different plan, without the user or application being explicitly aware of such a switch. This capability can aid dynamic query optimization [14] and is particularly attractive in Cloud-based scenarios where one may wish to move executing queries based on current workload conditions.

III. Theory of Logical Merge

A. Stream Basics

We view a stream as a representation of a (potentially unbounded) temporal database (TDB) that is presented incrementally. The TDB may take different forms in different stream systems. One example is a sequence of snapshots of a relational table, a second is a collection of ⟨tuple, timestamp⟩ pairs. For the algorithms and implementations we present here, the TDB is a multiset of *events*, each of which consists of relational tuple p (which we term the *payload*), along with an associated *validity interval* denoted by a validity start time V_s and a validity end time V_e, which define a half-open interval $[V_s, V_e)$. V_e is permitted to be $+\infty$. One can think of V_s as representing the event's timestamp, while the validity interval is the period of time over which the event is active and contributes to output.

A stream is a potentially unbounded sequence of *elements* (some of which may resemble TDB events). While the kinds of events and their ordering constraints can vary between stream systems, we assume that any finite prefix of a stream can be *reconstituted* into a TDB instance [5]. Let $S = e_1, e_2, \ldots$ be a stream, with $S[i]$ being the prefix $e_1, \ldots, e_i$. We posit a *reconstitution function* $tdb(S, i)$ that produces the TDB instance corresponding to $S[i]$.[1]

It would be useful to have a version $tdb(S)$ of the reconstitution function that interprets the whole of S. One approach to defining $tdb(S)$ is as the limit of $tdb(S, 1), tdb(S, 2), \ldots$. If S behaves well – say it satisfies the monotonicity property $tdb(S, i) \subseteq tdb(S, i+1)$ – then this limit is well defined. But there can be pathological cases where S does not converge to a particular TDB instance. For example, if S can contain a stream element that cancels a previous stream element (in the sense of removing it from the TDB rather than curtailing its lifetime), then a stream such as

e, cancel(e), e, cancel(e), e, cancel(e), e, cancel(e), …

has no definite limit. For specific cases we consider later, $tdb(S)$ will be guaranteed to exist, though sometimes via stream properties that are weaker than monotonicity.

In most DSMSs, there are multiple stream instances that represent the same TDB. For example, if each stream element carries an explicit timestamp, then it can happen that $tdb(S, i) = tdb(U, i)$ even though $S[i]$ and $U[i]$ are distinct prefixes, because of different orderings. If $tdb()$ removes duplicates, then it is possible that $tdb(S, i) = tdb(U, j)$ for $i \neq j$. We say that prefixes $S[i]$ and $U[j]$ are *equivalent* if $tdb(S, i) = tdb(U, j)$, written $S[i] \equiv U[j]$. Streams S and U are *equivalent*, written $S \equiv U$, if $tdb(S)$ and $tdb(U)$ are well defined and equal. There can be many streams that represent the same TDB, just as there can be many physical structures that represent a given logical table in a relational database. The range of possible descriptions of the same TDB in a given stream system depends both on what kinds of elements are permitted in a stream and on the constraints (or lack thereof) on the order of elements. For example, there can be stream elements that serve to "adjust" a previously seen event, such as by altering its lifetime or updating data values in it. As an example of an ordering constraint, most stream systems support some form of punctuations that limit stream elements that can appear later.

Example 3 (Open and close elements): Consider a simple stream representation in which there are two kinds of stream elements, open(p, V_s) and close(p, V_e), where open() indicates the start time of an event with payload p and close() indicates the end of the event with payload p. (We assume here that there can only be one event with payload p active at a time.) Open and close elements correspond to I-Streams and D-Streams in Oracle CEP [25] and STREAM [13], or positive and negative tuples in Nile [11]. The following stream prefixes are equivalent, each representing the TDB:

p	Vs	Ve
A	1	4
B	2	5
C	3	∞

S[5]: open(A, 1), open(B, 2), open(C, 3), close(A, 4), close(B, 5)

[1] In some DSMSs, events are assumed to arrive in batches [17], so it may only make sense to apply $tdb()$ to selected prefixes of S.

`U[5]: open(A, 1), close(A, 4), open(B, 2), close(B, 5), open(C, 3)`

`W[6]: open(B, 2), close(B, 6), open(A, 1), open(C, 3), close(A, 4), close(B, 5)`

Note that `close(B, 5)` in stream prefix `W[6]` serves to revise the previous `close(B, 6)`.

B. Definition of Logical Merge

If input streams never fail, the definition of Logical Merge is straightforward. It takes a set of equivalent input streams $I_1, \dots, I_n$ and produces an equivalent output stream O. That is, $I_1 \equiv \cdots \equiv I_n \equiv O$. In practice, however, input streams can fail (or detach), so different inputs will not be equivalent. We adopt the weaker notion of *mutual consistency* for input streams, which intuitively means there is some complete "reference stream" of which each input represents a segment. We want to express this condition in terms of stream prefixes, since that is all we have to work with at any finite point in time. Formally, stream prefixes $\{I_1[k_1], \dots, I_n[k_n]\}$ are *mutually consistent* if there exist finite sequences E_i and F_i, $1 \leq i \leq n$ such that $E_1 : I_1[k_1] : F_1 \equiv \cdots \equiv E_i : I_i[k_i] : F_i \equiv \cdots \equiv E_n : I_n[k_n] : F_n$. Here, $A : B$ denotes the concatenation of A with B. We say $\{I_1, \dots, I_n\}$ are *mutually consistent* if all finite prefixes of them are mutually consistent. Stream O represents the *Logical Merge* (LMerge) of mutually consistent streams $\{I_1, \dots, I_n\}$ if $\{I_1, , \dots, I_n, O\}$ are mutually consistent without extending O, and that O is minimal. In other words, there is no other mutually consistent O' with $tdb(O') \subset tdb(O)$. For simplicity in the sequel, we assume that all inputs start at the same point (the E_i's are empty). While this assumption will not necessarily hold in practice, we can treat an input stream that starts late as having a consistent prefix that was skipped over.

The LMerge definition above is *abstract* – in terms of mutual consistency of entire streams, not prefixes. However, while we usually wish to propagate inputs to the output eagerly, we need to also ensure that, at any given point in time, the output is able to follow future additions to the inputs. Thus, we need to ensure that the output can "track" any additional elements that show up on the inputs. We say that output-stream prefix $O[j]$ *is compatible with* input-stream prefix $I[k]$ if, for any extension $I[k] : E$ of the input prefix, there exists an extension $O[j] : F$ of the output sequence that is equivalent to it. Stream prefix $O[j]$ *is compatible with* the mutually consistent set of input stream prefixes $\boldsymbol{I} = \{I_1[k_1], \dots, I_n[k_n]\}$ if for any set of extensions $E_1, \dots, E_n$ that makes $I_1[k_1] : E_1, \dots, I_n[k_n] : E_n$ equivalent, there is an extension $O[j] : F$ of the output sequence that is equivalent to them all.

The specific criteria for guaranteeing compatibility between inputs and the output of LMerge depends on the kinds of stream elements allowed and any stream properties guaranteed on the inputs (or enforced on the output). We will assume that the kinds of elements and the properties are the same for all inputs and the output, though one could obviously relax this constraint.

C. Stream Properties and LMerge

We are interested in properties that a given stream S might satisfy in terms of element sequences it allows and the state of its TDB. Such properties will affect how the TDB can evolve, and may lead to simpler or less space-intensive methods for LMerge. Examples:

- Stream elements are ordered on some time attribute. In Example 3, `S[5]` has this property, but neither `U[5]` nor `W[6]` does. With this property, once time has advanced to point t, we know we have seen all payloads with $V_s \leq t$. Further, no event in the TDB with a finite V_e can get shorter.
- There can be at most one close() element for any open() element. `S[5]` and `U[5]` satisfy this condition, but not `W[6]`. With this condition, we know that once we see a close() element, the corresponding TDB event will be present forever.
- The pair $\langle p, V_s \rangle$ is a key for every instance of the TDB. Such a property might arise if p consisted of a sensor id and a reading, where no sensor reports more than once per time period. Such a constraint can simplify matching up corresponding events across inputs to an LMerge operator.

While such properties might be stipulated by input sources, they usually arise through compile-time analysis of query plans. For example, the last condition above holds on the output of any aggregate operator, since the subset of p corresponding to the grouping attributes are in fact a key at any point in time. The formulation of input-output compatibility for a given situation depends on what properties hold, as the following examples show.

Example 4 (Stream Properties and Compatibility): Consider streams with open() and close() elements and the property that each open() has at most one corresponding close(). Then output $O[j]$ is compatible with input $I[k]$ if $O[j] \subseteq I[k]$. In that case, there exists an extension F such that $O[j] : F \equiv I[k]$. So, $O[j] : (F : E) \equiv I[k] : E$ for any extension E of the input. Furthermore, the condition $O[j] \subseteq I[k]$ is necessary for compatibility. Suppose $O[j]$ contains $\text{open}(p, V_s) \notin I[k]$. Then, there is no way to extend $O[j]$ to be equivalent with $I[k] : \emptyset$. So, all the open events in $O[j]$ must be in $I[k]$. If $O[j]$ contains $\text{close}(p, V_e) \notin I[k]$, then there is no way to extend $O[j]$ to be equivalent with $I[k]$:close$(p, V_e + 1)$, since $O[j]$ already contains a close element for p. In the case of a set of mutually consistent inputs $\boldsymbol{I} = \{I_1[k_1], \dots, I_n[k_n]\}$, $O[j]$ is compatible with $\boldsymbol{I}$ exactly when $O[j] \subseteq (\cup \boldsymbol{I})$.

Example 5 (Compatibility for StreamInsight): This example corresponds to StreamInsight, for which we implemented the detailed algorithms defined in Section 4. StreamInsight has three kinds of elements:

- insert(p, V_s, V_e): Adds an event to the TDB with payload p whose lifetime is the interval $[V_s, V_e)$. V_e can be $+\infty$.
- adjust(p, V_s, V_{old}, V_e): Change the event $\langle p, V_s, V_{old} \rangle$ to be $\langle p, V_s, V_e \rangle$. If $V_e = V_s$, the event $\langle p, V_s, V_{old} \rangle$ is removed. For example, the sequence of elements: `insert(A, 6, 20)`, `adjust(A, 6, 20, 30)`, `adjust(A, 6, 30, 25)` is equivalent to the single element: `insert(A, 6, 25)`.

- stable(V_c): A statement that the portion of the TDB before time V_c is *stable*: There can be no future insert(p, V_s, V_e) element with $V_s < V_c$, nor can there be an adjust element with $V_{old} < V_c$ or $V_e < V_c$.

The TDB model for StreamInsight is a collection of events of the form $\langle p, V_s, V_e \rangle$.

For our prototypes of LMerge, we consider the following range of restrictions that can improve performance if they hold. In Section 4, we present algorithms for each point in this spectrum, and discuss how stream properties can be derived and used to choose an appropriate algorithm.

R0. There are only insert() and stable() elements with strictly increasing V_s times. Hence, the stream has deterministic order with no duplicate events.

R1. The input steams consist only of insert() and stable() elements, V_s is non-decreasing, and the order among elements with equal V_s is deterministic.

R2. Same as R1, except order for elements with the same V_s can differ across inputs. Further, for any stream prefix $S[i]$, $\langle p, V_s \rangle$ forms a key for $tdb(S, i)$.

R3. All kinds of elements are permitted and there is no constraint on time order, except as imposed by stable() elements. As with R2, for any stream prefix $S[i]$, $\langle p, V_s \rangle$ forms a key for $tdb(S, i)$.

We will use **R4** to signify the "no additional restrictions" case where all three kinds of elements are permitted, elements need not be in timestamp order, and the TDB is a multi-set (hence there can be more than one event with the same payload and lifetime).

In order to understand the correctness of our algorithm for the R3 case, we find it useful to think of a stable(V_c) element as "freezing" certain parts of the TDB. A TDB event $\langle p, V_s, V_c \rangle$ is *half frozen* (*HF*) if $V_s < V_c \leq V_e$ and *fully frozen* (*FF*) if $V_e < V_c$. If $\langle p, V_s, V_e \rangle$ is half frozen, we know there will be some event $\langle p, V_s, V \rangle$ in the TDB henceforth. If $\langle p, V_s, V_e \rangle$ is fully frozen, no future adjust() event can alter it, and so it will be in all future version of the TDB. Any TDB event that is neither half frozen nor fully frozen is *unfrozen* (*UF*).

D. Correctness for the R3 Case

Before presenting the precise conditions for input-output compatibility for R3, we provide examples of possible outputs for given inputs to LMerge. Both input and output streams are described by their TDBs; our discussion is applicable to any input stream that reconstitutes to a given input TDB, and allows the output of any stream that reconstitutes to a given output TDB. For each of the TDBs below, *last* is the latest value V such that a stable(V) element has been seen. The annotation to the right of each event indicates its "freeze" status.

I1 (last:14)

p	Vs	Ve	
A	2	16	**HF**
B	3	10	**FF**
C	4	18	**HF**
D	15	20	**UF**

I2 (last:11)

p	Vs	Ve	
A	2	12	**FF**
B	3	10	**FF**
C	4	18	**HF**
E	17	21	**UF**

O1 (last:11)

p	Vs	Ve	
A	2	∞	**HF**
B	3	10	**FF**
C	4	∞	**HF**

O2 (last:14)

p	Vs	Ve	
A	2	16	**HF**
B	3	10	**FF**
C	4	18	**HF**
D	15	20	**UF**
E	17	21	**UF**

O3 (last:13)

p	Vs	Ve	
A	2	12	**FF**
C	4	18	**HF**
D	15	20	**UF**

Consider the LMerge of streams corresponding to `I1` and `I2`.

`O1` is compatible with `I1` and `I2`. It has a TDB that might result from a conservative tracking policy that outputs only information that must be in the output eventually. `O1` will only require adjustments to end times.

`O2` represents a more aggressive policy, but it is still compatible with `I1` and `I2`. It contains events corresponding to all input events seen, even if those events are unfrozen. `O2` may have to issue later elements to completely remove some events.

`O3` is not compatible with `I1` and `I2` for two reasons. First, although `<A, 2, 12>` matches an event in `I2`, it contradicts the contents of `I1`, from which we can tell the end time will be no less than 14. As this event is fully frozen in `O3`, there is no subsequent stream element that can correct it. Second, `O3` lacks the event `<B, 3, 10>`, which is fully frozen in the input streams but cannot be added to `O3` given its stable point.

We now describe (and justify) the exact conditions for compatibility in the R3 case.

Assume $\{I_1, \ldots, I_n\}$ are mutually consistent input streams and O is the output stream. Suppose at some instant we have seen prefixes $\{I_1[k_1], \ldots, I_n[k_n]\}$ of the input streams and emitted prefix $O[j]$ on the output stream. Let $\text{TDB}_m = tdb(I_m, k_m)$ and $\text{TDB}_O = tdb(O, j)$. Assume that stable(L_m) was the most recent stable() event on I_m, and stable(L) was the most recent stable event on O. We must have the following conditions.

C1. L is no greater than the maximum of the L_m. (If it were, then it is possible for an event to appear in one of the inputs and be fully frozen there without being able to add it to O.)

The other two conditions concern what events *may* be in TDB_O (Condition C2) and what events *must* be in TDB_O (Condition C3) for given combination of p and V_s.

C2. TDB_O may contain at most one event for a given p and V_s.

- If that event is UF, there is no constraint on it (as it can be completely removed).
- If TDB_O contains $\langle p, V_s, V_e \rangle$ that is HF, then there must be some TDB_m containing $\langle p, V_s, V_m \rangle$ where either the event is HF and $L_m \leq L$ (so the output event can be adjusted to match any changes in TDB_m) or the event is FF and $L \leq V_m$ (so it is still possible to adjust TDB_O to match TDB_m).
- If TDB_O contains $\langle p, V_s, V_e \rangle$ that is FF, then there must be some TDB_m containing $\langle p, V_s, V_e \rangle$ that is FF (so we know that event is definitely in the output).

C3. TDB_O must have an event for p and V_s in either of two cases:

1) There is an event $\langle p, V_s, V_e \rangle$ in some TDB_m that is FF and either

- $L \leq V_s$ (in this case, the event can still be added to TDB_O), or
- $V_s < L \leq V_e$ and TDB_O has $\langle p, V_s, V_O \rangle$ that is HF (note that since $L \leq V_e$, this event can be adjusted to $\langle p, V_s, V_e \rangle$), or
- $V_e < L$ and TDB_O contains $\langle p, V_s, V_e \rangle$.

2) No input contains a FF event for p and V_s, but one or more inputs contain a HF event of the form $\langle p, V_s, \_ \rangle$. Let I_m be the input with such an event with the largest L_m. Then either:

- $L \leq V_s$ (in which case an appropriate event can still be added to TDB_O), or
- $V_s < L \leq L_m$ and TDB_O has $\langle p, V_s, V_O \rangle$ that is HF (which can be adjusted to match future changes to the event in the input).

(Note that an UF event $\langle p, V_s, V_e \rangle$ in any input places no constraint on TDB_O.)

These conditions are simplified if L tracks the largest L_m. In that case, the requirement is that TDB_O and TDB_m have the same set of FF events, and that their sets of HF events match on p and V_s.

Compatibility in the R3 case leaves room for a wide range of policies on how loosely or tightly the output of LMerge tracks the input. A very liberal policy would be to allow arbitrary unfrozen events in the output, even if there is no support among the inputs for such an event. This policy is likely unwise, since such events would almost surely be adjusted, unless there were a robust model for predicting future inputs. A more reasonable policy is to allow only half frozen and fully frozen events in the output where each event has support in the TDB of one of the inputs. That support might take the form of an exactly matching event, or, for half-frozen events in the output TDB, a half-frozen event in some input TDB with the same payload and valid start values. A conservative policy might only allow an element in the output if it is supported by a fully frozen event in one of the input TDBs. These different policies tend to trade latency for "chattiness" of the output: how many adjust() elements might need to be issued to bring the output into line with adjustments on the input. A second aspect of output policy is when to issue a stable () element on the output. Our experience is that we want to keep the output at the maximum stable point of all the inputs to minimize the memory requirements of LMerge, though there might be cases where lagging a bit behind the maximum would avoid some adjust() elements in the output.

Compatibility in the R4 case, where there can be multiple events with the same p and V_s, has more complicated conformance conditions. If L, the maximum stable point of the output O, tracks the maximum L_m, then TDB_O must contain all the FF events from TDB_m, and an equal number of HF events, for that p and V_s.

IV. Algorithms for Logical Merge

In this section, we provide algorithms for different variants of LMerge optimized for specific stream properties described in Section III-C. Section IV-F shows how we may use stream properties to decide which algorithm to use, given a CQ and input streams.

A. LMerge Algorithm for Case R0

We start with case R0, where input streams have elements with strictly increasing `Vs` values, hence no duplicate timestamps. In this case, it turns out that we need only two pieces of information: the maximum `Vs` (`MaxVs`) and the maximum stable() timestamp (`MaxStable`) seen across all input streams. Refer to Algorithm R0. When we see the insertion of element `e` on stream `s`, we can discard the element if it does not increase `MaxVs`, and output it otherwise (Lines 3-5). Note that `s` is simply the identifier (for e.g., an integer) of a specific input stream. Similarly, a stable() element is output if it increases `MaxStable` (Lines 9-11). Note that in the R0 case, stable() elements are in a sense redundant, since the stable point advances with each new insert(), though a system might include them to signal progress in the presence of lulls.

Algorithm R0: Logical Merge for Case R0

```
 1 MaxStable = MaxVs = −∞;
 2 Insert(element e, stream s)
 3   if (e.Vs > MaxVs)
 4     MaxVs = e.Vs;
 5     OutputInsert(e);
 6 Adjust(element e, stream s)
 7   Error("Not supported");
 8 Stable(timestamp t, stream s)
 9   if (t > MaxStable)
10     MaxStable = t;
11     OutputStable(t);
```

B. LMerge Algorithm for Case R1

The next algorithm considers case R1, an insert-only case with non-decreasing `Vs`. Here, we may have duplicate `Vs` timestamps, but such elements are presented in deterministic order (e.g., sorted on a field in the payload). This condition holds in scenarios such as Top-k aggregation, where elements with the same `Vs` are presented in rank order. Here, all we need to maintain (in addition to `MaxStable` and `MaxVs`) is an array with one counter for each input stream, which counts the number of elements on that stream with `Vs` = `MaxVs`. Refer to Algorithm R1. On an insert element that increases `MaxVs`, we reset this array to zeros (Lines 5-7). If the insert on stream s increases the counter for `s` beyond the old maximum counter value across all streams, the insert is sent as output (Lines 8-10). A stable() element is handled as before.

```
Algorithm R1: Logical Merge for Case R1
 1 MaxStable = MaxVs = −∞;
 2 SameVsCount[1 ... #inputs] = 0;
 3 Insert(element e, stream s)
 4   if (e.Vs < MaxVs) return;
 5   if (e.Vs > MaxVs)
 6     SameVsCount[1 ... #inputs] = 0;
 7     MaxVs = e.Vs;
 8   if (MAX(SameVsCount) == SameVsCount[s])
 9     OutputInsert(e);
10   SameVsCount[s]++;
11 Adjust(element e, stream s)
12   Error("Not supported");
13 Stable(timestamp t, stream s)
14   if (t > MaxStable)
15     MaxStable = t;
16     OutputStable(t);
```

C. LMerge Algorithm for Case R2

Next in complexity is case R2, a non-decreasing, insert-only case where we may have duplicate `Vs` values, and elements with the same `Vs` may be presented in different orders by different streams. We assume that (`Vs`, `Payload`) is a key in any prefix of the TDB. (The relaxation to handle duplicates is straightforward and omitted.) Refer to Algorithm 2. Our algorithm uses a hash table in addition to `MaxStable` and `MaxVs`. The hash table indexes (using `Payload` as key) all elements with `Vs` = `MaxVs`. When we receive an insert element, we check the hash table – if the corresponding payload exists, we are done. Otherwise, we update the hash table and output the element (Lines 8-10). An element that increases `Vs` beyond `MaxVs` clears the hash table (Lines 5-7) so that it can track elements with the new `MaxVs`.

```
Algorithm R2: Logical Merge for Case R2
 1 MaxStable = MaxVs = −∞;
 2 hash = new Hashtable();
 3 Insert(element e, stream s)
 4   if (e.Vs < MaxVs) return;
 5   if (e.Vs > MaxVs)
 6     hash.Clear();
 7     MaxVs = e.Vs;
 8   if (!hash.Contains(e))
 9     hash.Add(e);
10     OutputInsert(e);
11 Adjust(element e, stream s)
12   Error("Not supported");
13 Stable(timestamp t, stream s)
14   if (t > MaxStable)
15     MaxStable = t;
16     OutputStable(t);
```

D. LMerge Algorithm for Case R3

We now tackle case R3, where inserts, adjusts, and stable elements may be presented in any order, and (`Vs`, `Payload`) is a key in the TDB for any stream prefix. See Algorithm R3; we propose a new index structure called *in*^{2}*t* (for *index-2-tier*) depicted in Figure 1 (left). The top tier of *in*^{2}*t* is a red-black-tree keyed by (`Vs`, `Payload`), where each node consists of an event and points to a second tier index implemented as a hash table. The hash table contains, for each input stream `s`, the current `Ve` value for that stream indexed by key `s`. An additional hash table entry with special key ∞ is also maintained for the output.

On an insert() element in stream s, we lookup *in*^{2}*t* for a node with the same (`Vs`, `Payload`). If such a node does not exist (Lines 5-10), we add the node and produce output. In the hash table, we add an entry for stream s as well as for the output. An exception is when `Vs` is less than `MaxStable` (Line 6), which indicates that the corresponding entry previously existed and has been removed from *in*^{2}*t*. Otherwise (Line 12), we simply add an entry to the hash table and return. An adjust() element is handled similarly (Lines 14-16), except that output is not produced as a result of an adjust.

Finally, consider the processing of a stable() element `e`. We only need to handle stable() elements that increase `MaxStable`. We first find nodes that are going to become half frozen in *in*^{2}*t*; i.e., nodes whose `Vs` is less than `e`'s timestamp. For each such node, we check if there is a mismatch between the output and the input, where a compatibility violation is going to occur as a result of outputting e. There are three cases of compatibility violations:

- There is no input event for (`Vs`, `Payload`) in stream s, but there is an output event (due to some other input stream).
- The currently output event will become fully frozen due to `e`, but the corresponding input is not fully frozen.
- The input event will become fully frozen, but the current output is not fully frozen.

In all cases, we adjust the output so that it matches the input (Lines 24-27). This choice – of correcting output only to avoid irrecoverable divergence between output and input – represents one out of several policies discussed in Section 5.1. Finally, if the input becomes fully frozen, we delete the corresponding node from *in*^{2}*t* (Lines 28-29), update MaxStable, and output a stable() element (Lines 30-31).

```
Algorithm R3: Logical Merge for Case R3
 1 MaxStable = −∞;
 2 index = new in2t();
 3 Insert(element e, stream s)
 4   node f = index.SameVsPayload(e);
 5   if (!exists(f))
 6     if (e.Vs < MaxStable) return;
 7     f = index.AddNode(e);
 8     OutputInsert(e);                    2
 9     f.AddHashEntry(∞, e.Ve); // hash entry for o/p
10   f.AddHashEntry(s, e.Ve); // hash entry for i/p
11 Adjust(element e, stream s)
12   node f = index.SameVsPayload(e);
13   if (!exists(f)) return;
14   f.UpdateHashEntry(s, e.Ve);           1
15 Stable(timestamp t, stream s)
16   if (t <= MaxStable) return;
17   iterator it = index.FindHalfFrozen(t);
18   while (node f = it.Next())
19     InVe = f.GetHashEntry(s);
20     if (!exists(InVe)) InVe = f.GetEvent().Vs;
21     OutVe = f.GetHashEntry(∞);
22     if (InVe != OutVe and
23         (InVe < t or OutVe < t))
24       OutputAdjust(f.GetEvent(), Ve: InVe);
25       f.UpdateHashEntry(∞, InVe);
26     if (InVe < t) // fully frozen
27       index.DeleteNode(f);
28   MaxStable = t;
29   OutputStable(t);
```

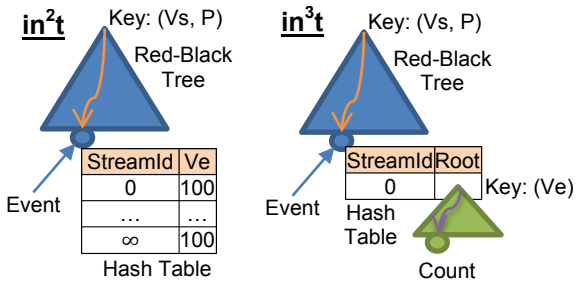

Fig 1: Data structures for cases R3 (in^2t) and R4 (in^3t) of LMerge

E. LMerge Algorithm for Case R4

The main challenge with case R4 is that many elements in a stream can have the same (`Vs`, `Payload`), with different `Ve` values. Further, there could be duplicates in the stream. Hence, we propose a new index structure – shown in Figure 1 (right) – called *in^3t* (for *index-3-tier*), where we replace the single `Ve` value in each entry of the lower-level hashtable of *in^2t* with a small index (red-black-tree) on `Ve`, where each `Ve` is associated with its count (to handle duplicates). See Algorithm R4; during insert and adjust, the output is updated lazily as before. When processing a stable() element, we ensure future compatibility before producing a stable() element as output. The invariants we maintain are more subtle:

- (Lines 9-11) The output TDB contains no more events for a particular (`Vs`, `Payload`) than the maximum number of events in any input TDB, for that (`Vs`, `Payload`). While not necessary, this condition helps limit output chattiness.
- (Lines 20-22) When an incoming stable() element has a timestamp greater than some `Vs` (i.e., that `Vs` becomes half frozen), we ensure that, for each (`Vs`, `Payload`) that is getting half frozen, there are exactly as many output elements with a value of (`Vs`, `Payload`) as there are in the input. This invariant needs to be met before we can propagate the stable() element, in order to guarantee future convergence. The method `AdjustOutputCount()` determines the exact procedure for meeting this invariant; briefly, it involves producing new output elements or "canceling" prior output elements for that (`Vs`, `Payload`) combination. We discuss this method below.
- (Lines 23-26) For a particular (`Vs`, `Payload`), if some `Ve` becomes fully frozen as a result of an incoming stable() element, we need to ensure that our output TDB contains the same number of events with that (`Vs`, `Payload`, `Ve`), before propagating the stable() element. The `AdjustOutput()` method achieves this invariant; briefly, it involves adjusting the `Ve` of events output earlier with the same (`Vs`, `Payload`). We discuss this method below.

When the stable() timestamp moves beyond the largest `Ve`, for a particular (`Vs`, `Payload`), the corresponding node can be deleted from the top tier of in^3t (Lines 27-28).

Adjusting Output to Meet Invariants The method `AdjustOutputCount()` ensures that, when a particular `Vs` gets

```
Algorithm R4: Logical Merge for Case R4
 1 MaxStable = −∞;
 2 index = new in3t();
 3 Insert(element e, stream s)
 4   node f = index.SameVsPayload(e);
 5   if (!exists(f))
 6     if (e.Vs < MaxStable) return;
 7     f = index.AddNode(e);
 8   f.IncrementCount(s, e.Ve);
 9   if ((e.Vs>=MaxStable) and (f.GetCount(s)>f.GetCount(∞)))
10     OutputInsert(e);
11     f.IncrementCount(∞, e.Ve);
12 Adjust(element e, stream s)
13   node f = index.SameVsPayload(e);
14   if (!exists(f)) return;
15   f.IncrementCount(s, e.Ve); f.DecrementCount(s, e.Vold);
16 Stable(timestamp t, stream s)
17   if (t <= MaxStable) return;
18   iterator it = index.FindHalfFrozen(t);
19   while (node f = it.Next())
20     if (f.Vs >= MaxStable) // element getting half frozen
21       // ensure #o/p events=#i/p events for that (Vs, P)
22       AdjustOutputCount(f);
23     iterator itIn = f.FindAllVe(s);
24     iterator itOut = f.FindAllVe(∞);
25     // Make o/p reflect i/p for all FF (Ve < t) nodes
26     AdjustOutput(f, t, itIn, itOut);
27     if (f.GetMaxVe(s) < t) // Done processing that (Vs, P)
28       index.Delete(f);
29   MaxStable = t;
30   OutputStable(t);
```

half frozen, there are exactly as many output elements with a value of (`Vs`, `Payload`) as there are in the input. There are two possibilities. If there are more output events with the given value of (`Vs`, `Payload`), we delete output elements until the counts are identical. If there are more input events, we output new insert() elements for (`Vs`, `Payload`) with `Ve` values that have been seen on the input stream that is getting half frozen. During this process, we may also choose to update existing output Ve values so that the input and output streams match exactly for that (`Vs`, `Payload`) combination. This may be useful if we expect half frozen elements to rarely get updated in the future.

When an element with a particular (`Vs`, `Payload`, `Ve`) combination would become fully frozen due to a stable() element, the `AdjustOutput()` method ensures that we have the same count of output elements for that (`Vs`, `Payload`, `Ve`) combination as we have in the corresponding half frozen input (if the element is an output element that is not half frozen in the input, it can be deleted). Assuming that the total count invariant for half frozen elements was met earlier, and that input streams are logically identical, we are guaranteed to find existing output elements with the same (`Vs`, `Payload`) combination that we can adjust to achieve the above. Thus, depending on whether the output count for (`Vs`, `Payload`, `Ve`) is smaller or larger that the input count, `AdjustOutput()` has to either (1) change the `Ve` of existing output elements with a larger `Ve` to the current `Ve` value; or (2) expand the current `Ve` value to a larger value, either ∞ or a `Ve` present in the input for that (`Vs`, `Payload`) combination.

F. Space and Runtime Complexity of LMerge

We analyze the complexity of the LMerge algorithms on the basis of *runtime stream properties* that characterize the nature of input streams to LMerge. These properties can be measured as statistics during runtime, although some may be determined statically based on operators in the plan. Let s denote the number of input streams to LMerge. Consider the set of events that are "alive", i.e., not fully frozen at any given instant. Let w denote the number of unique (`Vs`, `Payload`) values, and d denote the number of elements with the same (`Vs`, `Payload`). Further, let g denote the number of events with the same `Vs`, and let h represent the number of distinct half-frozen (`Vs`, `Payload`) values. Finally, let c be the number of events that become fully frozen due to a stable() element, and let p denote payload size. Based on these properties, the complexity of the various LMerge algorithms is shown in Table IV.

TABLE IV
RUNTIME AND SPACE COMPLEXITY OF LMERGE.

Case	Runtime Complexity			Space Complexity
	Insert	***Adjust***	***Stable***	
R0	O(1)	n/a	O(1)	O(1)
R1	O(s)	n/a	O(1)	O(s)
R2	O(s)	n/a	O(1)	O($g \cdot p$)
R3	O($\lg w$)	O($\lg w$)	O($c \cdot \lg w + h$)	O($w(p+s)$)
R4	O($\lg w + \lg d$)	O($\lg w + \lg d$)	O($c \cdot \lg w + h \cdot d$)	O($w(p + s \cdot d)$)

G. Choosing the Right LMerge Algorithm

Given a range of algorithms for LMerge, a question that naturally arises is: How do we choose the right version of LMerge for a given set of input streams and query plan? We derive and reason about *compile-time stream properties* in order to answer this question. We do not give a detailed formalism of stream properties here, but we provide several examples to illustrate how stream properties are used for this purpose:

1) Every input stream publishes properties that indicate whether the stream is ordered, has adjust() elements, or has duplicate timestamps. If we are merging such input streams directly, we can use such properties to choose an algorithm.

2) The DSMS may have special operators that enforce certain properties. For example, many systems have a reordering or cleansing operator that accepts disordered input, buffers it and outputs an in-order stream. Such a stream can be annotated at compile-time in order to choose an appropriate LMerge algorithm.

3) Certain operators or groups of operators produce streams with a certain property. For example, an in-order stream fed into a windowed aggregate (e.g., count) outputs one event per strictly increasing timestamp, leading to a choice of algorithm R0.

4) If each input to LMerge results from an in-order stream fed into a sliding window multi-valued aggregate such as Top-k, we would choose algorithm R1, due to duplicate timestamps.

5) If each query under LMerge performs a grouped aggregation (e.g., a count for every machine in a data center) over an ordered stream, we would use algorithm R2 since the order for elements with the same `Vs` is non-deterministic.

6) If each query instead performs a grouped aggregation (e.g., count) over a disordered stream, we would use algorithm R3.

V. DISCUSSION AND EXTENSIONS

A. LMerge Policy Choices

Under the basic requirement of LMerge maintaining "compatible" output, we can implement various policies. For example, Algorithm R3 (Section IV) highlights two locations where we have such freedom to choose different policies. In location 1, we choose to never output incoming adjust events, instead preferring to retain the current output for every unique `Vs`. We issue adjust() elements to ensure that output is compatible with inputs, only when we process a stable() element. This policy limits chattiness of LMerge, as the following theorem indicates.

Theorem 1 (Non-chattiness) Algorithm R3 outputs no more insert() or adjust() elements than the total number of insert() elements received. Further, R3 outputs no more stable() elements than the total number of stable() elements received. Some alternatives here include:

- We can reflect every adjust() element at the output. This choice makes LMerge more "chatty", but allows listeners to process such changes earlier if they are interested.
- Force LMerge to "follow" a particular input stream, for example, the stream with the currently maximum stable() timestamp (called the *leading stream*). This choice may be appropriate when one stream is usually ahead of the others. However, if the leading stream keeps changing, this policy can incur significant overhead in re-adjusting output. Note that even in this case, LMerge needs to track information from other inputs in order to handle the case where the leading stream detaches.

Another point for choosing a different policy is location 2. When we process the first insert element for a particular `Vs`, we reflect it at the output immediately. While this policy ensures that output is maximally responsive, as before, we may choose other variants:

- We can output an insert only if it is produced by the leading stream, or the stream with the highest insert() timestamp or the maximum number of unfrozen elements.
- We can avoid sending an element as output until it gets half frozen on some input stream. This policy ensures that we never fully remove an element that we place on the output, at the expense of higher latency.

A hybrid choice may be to wait until some fraction of the input streams have produced an element for each `Vs`, before sending it to the output. If input streams are physically different, this policy may reduce the probability of producing spurious output that later needs to be fully deleted.

B. Joining and Leaving Input Streams

We need to handle joining and leaving streams. When a stream leaves LMerge, it is simply marked as "leaving". Eventually, our algorithms guarantee that it will no longer be

considered during LMerge. A joining stream provides a timestamp t such that it is guaranteed to produce the correct TDB for every point starting from t (i.e., every event in the TDB with `Ve` $\geq t$). We can mark the stream as "joined" as soon as `MaxStable` reaches t, since from this point forwards, LMerge can tolerate the simultaneous failure or removal of all the other streams.

C. Handling Missing Elements

We would like to handle the case where individual input stream may contain missing elements. One may expect that our goal should be that the output must contain an element as long as some input stream reports it. However, it is easy to see that this requirement forces LMerge to progress (issue stable() elements) only as fast as the slowest progressing input stream. (Consider an element that is missing from every stream other than the slowest-progressing one.) This option is highly undesirable in practice.

Instead, Algorithms R0, R1, and R2 output elements missing in some stream S as long as some other stream delivers the missing elements to LMerge before S delivers an element with higher `Vs`. These algorithms optimistically track only the latest `Vs` across all inputs (`MaxVs`) in order to minimize state and achieve high performance. Algorithms R3 and R4 output an element `e` as long as the stream that increases `MaxStable` beyond `e.Vs` produces element e.

D. Feedback to Signal Progress

An interesting application of LMerge is combining several alternative, equivalent query plans that behave differently under different conditions, such as data-value distributions or arrival rates. Alternatively, we may be executing identical plans on machines with varying resources such as CPU. LMerge can select results from whichever plan is producing output the soonest at a given point in time. Under such conditions, much of the work of the other plans is wasted, as LMerge ignores their outputs.

We propose a modification to LMerge, where LMerge signals to its input plans that elements before a certain time t are no longer of interest. This modification can allow the slower plans to avoid sending such elements. Particular operators may also be able to avoid performing unnecessary computations and purge state to save memory, though they must retain enough information to potentially produce output after time t, if required.

We have implemented feedback signaling for LMerge (cf. Section VI-E). Operators in the slower plan react to the feedback signal in order to avoid work, and purge state when possible, and propagate the signal further upstream in the plan. This capability enables more efficient dynamic selection among plans at run time by allowing the slower plans to "fast-forward" in order to catch up. Note that more general exploitation of such signals is possible, along the lines of feedback punctuation [8].

VI. Evaluation

We approach the evaluation of LMerge in three phases:

1) We demonstrate the behavior of LMerge over streams generated using query fragments over disordered input. Further, we compare the algorithm variants, some of which are relevant only for certain stream properties hold.
2) We compare the strategy of enforcing stream properties in order to use the simpler versions of LMerge, against directly using a more general version of LMerge.
3) We apply LMerge for solving several real applications: fast availability, network-congestion masking, and dynamic plan selection with feedback signals.

A. Setup and Implementation

We use StreamInsight to implement our algorithms. We perform our experiments on an 8-core machine with two 2.33GHz processors and 16GB main memory running Windows Server 2008 R2. We evaluated all our proposed LMerge variants (see Section IV for details):

1) LM^{R4}: This operator is the most general LMerge variant (case R4), and uses the in^3t data structure.
2) LM^{R3+}: This operator implements the in^2t based algorithm, and is the preferred algorithm for case R3.
3) LM^{R3-}: This variant uses a simpler algorithm for case R3 of LMerge, where events from each input stream are maintained in a *separate* index, with another index used to hold output events. The output index is required: (1) to check whether an element was previously output; (2) to perform adjustments to prior output before propagating a stable() element. While this algorithm is simpler to implement, it duplicates event information across input streams and requires multiple tree lookups at runtime.
4) LM^{R2}: In case R2, events with the same non-decreasing `Vs` may arrive in different orders at different inputs. Here, we only need one index to maintain, for the latest `Vs` across all inputs, all the events seen for that `Vs`.
5) LM^{R1}: In case R1, `Vs` is in non-decreasing order, and events with the same `Vs` are in deterministic order. We only need to maintain (1) the latest `Vs` seen across all inputs; and (2) a counter per input, which tracks the number of events seen with this value of `Vs` on that input.
6) LM^{R0}: Input streams in case R0 are in strictly increasing order of `Vs`. Thus, we only need to maintain the latest `Vs` and the latest stable() timestamp seen across all inputs.

We also evaluated the combination of LMerge with a *Cleanse operator* (called C+LM) to enforce stream properties a priori (see Section VI-D). Finally, we added support in StreamInsight for feedback signals (cf. Sections II, V-D, and VI-E).

B. Metrics and Workloads

We track: (1) *Throughput*, which measures the number of events produced at the output per second; (2) *Memory*, which measures the main memory used by an operator, including elements, payloads, and index structures; and (3) *Output Size*, which measures the number of adjust() elements produced. This metric quantifies the chattiness of the stream.

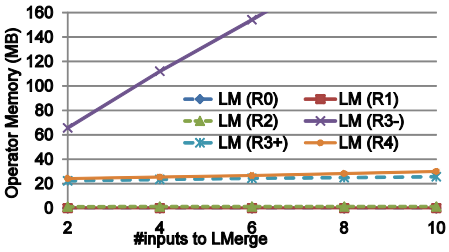

Fig. 2 Memory, in-order input streams.

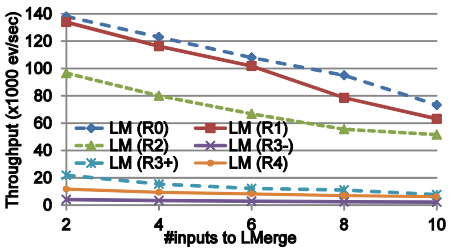

Fig. 3 Throughput, in-order streams.

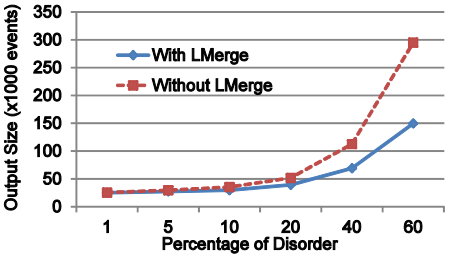

Fig. 4 Output size, increasing disorder.

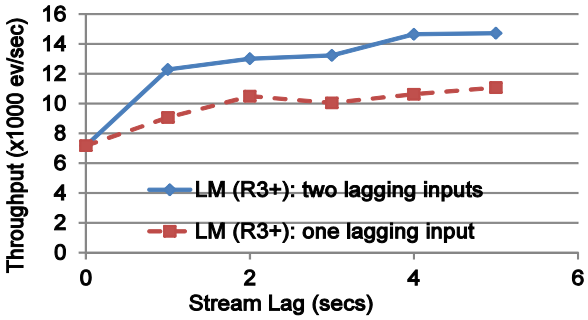

Fig. 5 Throughput, increasing stream lag.

Our evaluation mostly used synthetically generated datasets.[2] We use a commercial-grade test stream generator [26] to produce data. Each event has two fields, an integer in the interval [0, 400] and a randomly generated 1000-byte string. The event generator produces between 200K and 400K elements, based on a set of supplied parameters, including:

- *StableFreq*: The probability that an element in the stream is a stable() element. We ensure that at least one insert() is generated between consecutive stable() elements. The default value of this parameter is 1%.
- *EventDuration*: The lifetime of each event. By default, lifetime is set so that, on average, around 10K elements are "active" (contributing to output) at any point in time.
- *MaxGap*: The maximum application-time gap between consecutive elements. The gap is chosen randomly from the range [0, MaxGap]. We set MaxGap to 20 seconds.
- *Disorder*: The fraction of disordered elements. Disorder is created by moving V_s values back by some amount. Disorder is generated on a best-effort basis (e.g., we cannot have 100% disorder with StableFreq=1). The default value is 20%.

Our generated streams have disorder but no adjust() elements. Such elements are naturally produced during query processing, and hence we use sub-queries over the stream-generator output in order to generate them. A simple example of such a sub-query is aggregate (count) followed by a lifetime modification.

C. Investigating LMerge Behavior

We investigate the performance of the different LMerge algorithms as we vary different stream characteristics.

1) LMerge over Ordered Streams We use an ordered stream without adjust() elements, and thus can evaluate all the variants of LMerge. Figure 2 shows the memory usage of LMerge, as we increase the number of input streams. We see that LM^{R0} and LM^{R1} have negligible memory usage. LM^{R2} is slightly higher as it maintains all events with the current highest `Vs`. (The lines in Figure 1 for LM^{R0}, LM^{R1}, and LM^{R2} overlap as they perform similarly.) LM^{R3+} incurs slightly more memory than the simpler versions due to its generality, but the cost is almost independent of the number of inputs, as it shares evhent payloads across inputs. In contrast, LM^{R3-} requires much more memory due to duplication of data, and degrades linearly with the number of input streams.

We compare the algorithms in terms of throughput in Figure 3. As expected, the simpler algorithms provide higher throughput. Between LM^{R3-} and LM^{R3+}, we see that LM^{R3+} does much better than LM^{R3-} due to the optimized data structure and algorithm.

2) Output Size, Increasing Disorder We introduce disorder in the input stream, and feed it into a sub-query that generates many adjust() elements. Figure 4 compares the output of LMerge to the output without LMerge, as we increase the percentage of disorder. We see that when disorder increases, the number of adjusts increases significantly at the output. However, our specific output policy controls chattiness by limiting the production of intermediate adjusts that may not be present in the final TDB.

3) Throughput, Increasing Stream Lag We feed LMerge three input streams with 20% disorder each, with StableFreq set at 0.1%. Element lifetimes are kept at 40 seconds. We simulate lag on two of the input streams by delaying event generation by a fixed amount of time. Figure 5 shows throughput as we increase lag from 0 to 5 seconds. We observe that as lag increases, LMerge performance improves since it can directly drop tuples from the lagging streams. LMerge hides the lag on the slower streams by following the "fastest" stream. Further, throughput gains are higher if more streams are lagging, as long as at least one stream is able to keep up with the workload (We experiment further with this phenomenon in Section VI-E.)

4) Memory and Throughput, Varying StableFreq We measure the effect of StableFreq on throughput and memory of LMerge. As we increase StableFreq from 0.001% to 1%, we see in Figure 6 (left) that memory usage decreases as

[2] We also tested LMerge with real stock ticker data mined from Yahoo! Finance (with no problem). However, the synthetic data generator gave us finer control over stream properties of interest.

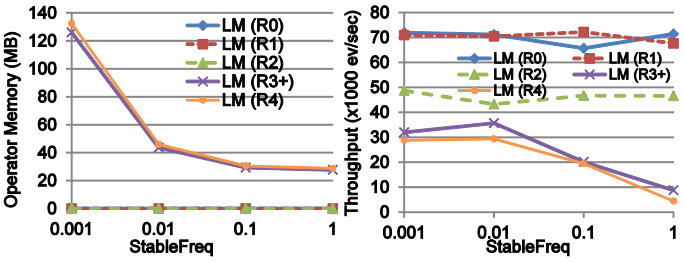

Fig. 6 Memory and throughput, increasing StableFreq.

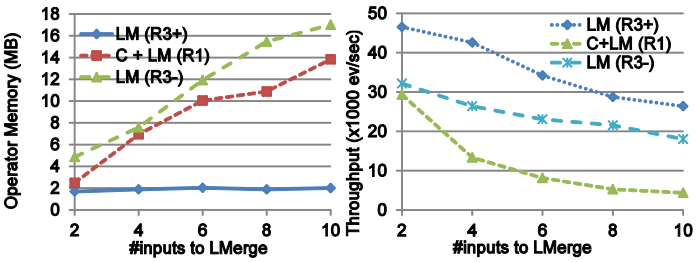

Fig. 7 Memory and throughput, enforcing stream properties.

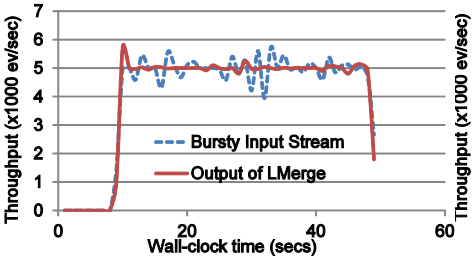

Fig. 8 Handling bursty streams.

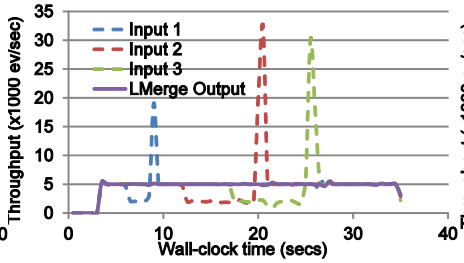

Fig. 9 Masking network congestion.

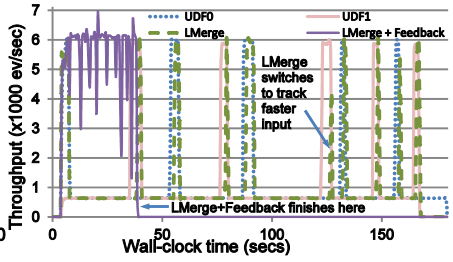

Fig. 10 Plan switching with fast-forward.

expected, due to more frequent cleanup. On the other hand, the throughput for LM^{R3+} and LM^{R4} decreases as shown in Figure 6 (right), as we need to perform more frequent compatibility checks. Note that the throughput for simpler schemes is not affected since they have significantly simpler algorithms for stable() elements.

D. Enforcing Stream Properties

Noting that the LMerge algorithms are significantly simplified for special cases where the stream satisfies specific properties, we investigate the possibility of enforcing these properties before feeding streams to the simpler versions of LMerge tailored for such properties. Timestamp ordering is enforced by a special *Cleanse* operator, which accepts a disordered stream and buffers elements until a stable() element is received, at which point it releases (in timestamp order) all fully frozen elements. We enforce ordering by placing a Cleanse at each input to LM^{R1}, which has constant memory requirement and is very efficient; this scheme is referred to as $C+LM^{R1}$. We use an input stream with 50% disorder, and pass it through an aggregate operator. The output of this query fragment contains 36% adjust() elements, with a 0.1% chance of seeing a stable() element.

1) Memory Consumption As we increase the number of inputs to LMerge from 2 to 10, we see from Figure 7 (left) that our optimized LM^{R3+} algorithm performs best, and its memory usage is almost independent of the number of input streams. However, the Cleanse-based solution ($C+LM^{R1}$) suffers linear degradation due to the overhead of ordering each stream separately – the overhead is nearly 7X more than LM^{R3+} for 10 inputs. We also see that LM^{R3-} degrades linearly with number of inputs due to no sharing of payloads across inputs.

2) Throughput Figure 7 (right) depicts throughput as we increase the number of input streams. Our solution (LM^{R3+}) outperforms the Cleanse-based solution ($C+LM^{R1}$). The relative improvement increases as we add more inputs because $C+LM^{R1}$ suffers from having to execute several Cleanse operators (one for each input) along with an LMerge operator (LM^{R1}) for the final merge. As before, LM^{R3-} does not perform well due to its naïve data structure.

3) Latency With $C+LM^{R1}$, the Cleanse operator buffers elements and produces output only when fully frozen. Thus, the latency of $C+LM^{R1}$ will grow with event lifetimes and the amount of potential disorder, since in order to maintain strict ordering, it needs to hold on to an element until stable() crosses `Ve`. Using LM directly, on the other hand, incurs latency in milliseconds (120ms on average for LM^{R3+}). Even if event lifetimes and the amount of potential disorder are a few seconds, the Cleanse solution will incur orders-of-magnitude higher latency than using LM directly.

In summary, applying LMerge directly on streams with disorder/revisions is superior (for memory, latency, and throughput) to ordering streams and doing a simpler merge.

E. Evaluating LMerge Applications

We next report on experiments that reflect different real-world situations where one might apply LMerge in practice.

1) Handling Bursty Data We generate four bursty streams with 20% disorder, each having an average event rate of 5000 elements/sec (this rate does not result in CPU overload under normal conditions). Bursty streams may exist in real applications due to several reasons such as CPU load and resource variations on machines, garbage collection, scheduling vagaries, and queue buildup between operators. We model burstiness by inserting random delays between tuples in a stream with a small probability (between 0.3 and 0.5%). The delays are chosen from a truncated normal distribution with mean 20 and standard deviation 5. Since elements arrive from sources at a constant rate, such delays result in temporary event build-up in queues, and cause subsequent compensating spikes in throughput. Figure 8

shows one of the input streams, along with the output of LMerge. Each stream is bursty, but LMerge smooths out the burstiness because it chooses to follow the best input at any given instant. Note that with many inputs to LMerge, the probability of all inputs behaving in a bursty manner at the same instant is greatly reduced.

2) Masking Network Congestion We use the same streams as before, presented at a rate of 5000 elements/sec. We model network congestion at different points in time in each of three streams, by introducing normally distributed delays between elements during the congested period. Network congestion results in temporary low throughput, followed by a spike in throughput when conditions return back to normal. Figure 9 shows the input streams as well as the output of LMerge. We see that the output of LMerge is unaffected by such congestion, as it is able to produce output as long as at least one input is not lagging. Note that at around 18 seconds, two inputs are simultaneously congested, but LMerge is unaffected as expected. Thus, we are able to completely mask the effect of such congestion using LMerge.

3) Dynamic Plan Switching with Fast-Forward We investigate the advantage of using LMerge for workload-based plan switching (see Sections II and V-D). We instantiate two alternate plans for the same query, both of which perform a user-defined selection function (UDF) on the data. The first plan (UDF0) is expensive for small values of X (a payload field), while the second plan (UDF1) is expensive for large values of X. We feed a stream with 200K elements, where alternating sequences (batches) of events have low and high values of X. The batch size is varied randomly between 10K and 30K elements. Thus, the "optimal" plan switches 9 times during execution. We show the performance of these queries individually (without LMerge) in Figure 10, where UDF0 and UDF1 finish in 176 and 163 seconds respectively. We next place LMerge (LM^{R3+}) above the two queries. One may expect LMerge to benefit from plan switching, but adding LMerge is not very useful because, while it tracks the faster input at any point, the total work performed in both queries is identical. Thus, the total processing time for LMerge is around 163 seconds.

We then let LMerge send feedback signals as described in Section V-D, to fast-forward the slower plan. This scheme, called LM+Feedback, allows LMerge to follow the faster plan, at the same time fast-forwarding the slower plan so that it can be immediately tracked by LMerge when it becomes optimal in the future. Overall, LM+Feedback completes execution in around 34 seconds, and is nearly 5X faster than LM^{R3+} without feedback.

VII. Related Work

Stream and Temporal Models A wide range of stream and temporal models have been proposed in research and adopted by industry. The model of STREAM [13], one of the early DSMSs, is adopted by Oracle CEP [25]. Aurora/Borealis [12] was commercialized as StreamBase. The CEDR project [4] proposed an interval-based algebra, motivated by early research on temporal databases [21], and forms the basis of StreamInsight [22]. NiagaraST [5] uses an interval-based model, but does not support speculation. In Nile [11], positive tuples begin new events while negative tuples expire older events. In Section III, we present the theory of LMerge as a general operator that can be used with any of these stream models – we discuss open and close elements (that are similar to I-streams and D-streams or positive and negative tuples) in Example 3. Our specific algorithms and implementations in this paper adopt the interval-based temporal model [4, 5, 21, 22], although other models can be handled with modifications.

High Availability High-availability in stream processing systems is a well-studied topic. Most techniques for high availability assume a primary copy of the query, and a backup copy that takes over when the primary fails. Hwang et al. [15] give a good overview of high availability schemes proposed for streams. Hwang et al. [9] propose a high-availability solution for wide-area networks that uses a duplicate-elimination operator for insert-only disordered streams. This algorithm can be classified as falling between R2 and R3 in our classification. In contrast, we focus on LMerge as a general primitive over a broad class of stream definitions, propose a suite of algorithms for LMerge, leverage stream properties and feedback for efficiency, and use LMerge for many new applications beyond high availability.

Dynamic Plan Switching Yang et al. [18] present an approach to switching between plans for a running stream query, which follows up on earlier seminal work by Zhu et al. [20]. Their approach involves determining a split time, where the old plan delivers all results before that time, and the new plan after. Such a cut-over involves a certain determinism in streams that would be hard to meet in the presence of disorder or element modifications. LMerge, in contrast, can cope with both queries running at once and producing distinct physical streams. Heinz et al. [19] use this cut-over technique to switch among plans when input statistics change significantly. We note that LMerge can provide a similar capability by running the alternative plans together and using feedback signaling to suppress work on slower plans.

Eddies [16] allows the choice of query plan to be chosen on a fine per-tuple granularity, but does not target temporal streams. LMerge, on the other hand, is a general operator that allows plan switching as one of its applications. Feedback signals sent from LMerge to fast-forward slow plans can be viewed as a novel application of feedback punctuation [8], which has been proposed and used in a different context.

VIII. Conclusions

We introduced the Logical Merge (LMerge) operator as a general duplicate-eliminating Union over input streams that are physically divergent and fallible. We defined LMerge in such a way that it can apply to any DSMS in which a stream (implicitly or explicitly) represents a temporal database. We discussed how input stream properties can affect LMerge, and presented a range of algorithms that deal with progressively more general cases.

We implemented our LMerge variants as operators in Microsoft StreamInsight, as well as feedback signals to reduce work in input queries. We first evaluated the different algorithms, over input streams that satisfied the strictest case, and then explored the response of specific algorithms to stream characteristics such as disorder and lag. We examined the benefit of a general LMerge relative to explicitly enforcing the stricter input properties that the more constrained versions require. Here, we found that using a general LMerge can sometimes provide orders-of-magnitude better memory and latency features, along with higher throughput. Finally, we showed the suitability of LMerge for applications with bursty events and congestion, where LMerge can smooth out variability. We also examined merging plans with different responses to data patterns, and showed that using feedback signals to fast-forward slower plans can significantly improve overall throughput.

REFERENCES

[1] U. Srivastava, J. Widom: *Flexible Time Management in Data Stream Systems*. PODS 2004: 263-274.

[2] P. Tucker et al.: *Exploiting Punctuation Semantics in Continuous Data Streams*. IEEE TKDE 15(3): 555-568 (2003).

[3] J. Li et al.: *Semantics and Evaluation Techniques for Window Aggregates in Data Streams*. SIGMOD 2005: 311-322.

[4] R. Barga et al.: *Consistent Streaming Through Time: A Vision for Event Stream Processing*. CIDR 2007: 363-374.

[5] D. Maier, J. Li, P. Tucker, K. Tufte, V. Papadimos: *Semantics of Data Streams and Operators*. ICDT 2005: 37-52.

[6] T. Johnson et al.: *A Heartbeat Mechanism and Its Application in Gigascope*. VLDB 2005: 1079-1088.

[7] J. Li et al.: *Out-of-order Processing: A New Architecture for High-Performance Stream Systems.* PVLDB 1(1):274-288 (2008).

[8] R. Fernandez-Moctezuma, K. Tufte, J. Li: *Inter-Operator Feedback in Data Stream Management Systems via Punctuation.* CIDR 2009.

[9] J. Hwang, U. Cetintemel, S. Zdonik: *Fast and Reliable Stream Processing over Wide Area Networks*. ICDE 2007: 604-613.

[10] E. Ryvkina et al.: *Revision processing in a stream processing engine: A high-level design*. ICDE 2006: 141.

[11] M. Hammad et al.: *Nile: A Query Processing Engine for Data Streams.* ICDE 2004: 851.

[12] D. Abadi et al.: *The design of the Borealis stream processing engine.* CIDR 2005.

[13] B. Babcock et al.: *Models and issues in data stream systems.* PODS 2002: 1-16.

[14] Y. Xing, S. Zdonik, J. Hwang: *Dynamic load distribution in the Borealis stream processor.* ICDE 2005: 791-802.

[15] J. Hwang et al.: *High-Availability Algorithms for Distributed Stream Processing.* ICDE 2005: 779-790.

[16] S. Madden, M. Shah, J. Hellerstein, V. Raman: *Continuously adaptive continuous queries over streams.* SIGMOD 2006: 49-60.

[17] I. Botan et al.: *SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems*. VLDB 2010: 232-243.

[18] Y. Yang et al.: *HybMig: A Hybrid Approach to Dynamic Plan Migration for Continuous Queries*. IEEE TKDE: 398–411 (2007).

[19] C. Heinz et al.: *Toward Simulation-Based Optimization in Data Stream Management Systems*. ICDE 2008: 1580-1583.

[20] Y. Zhu et al.: *Dynamic Plan Migration for Continuous Queries Over Data Streams.* SIGMOD 2004: 431-442.

[21] C. Jensen, R. Snodgrass: *Temporal Specialization*. ICDE 1992.

[22] Microsoft StreamInsight. http://tinyurl.com/4awexam.

[23] B. Gedik et al.: *SPADE: The System S Declarative Stream Processing Engine.* SIGMOD 2008: 1123-1134.

[24] J. Hellerstein, P. Haas, H. Wang: *Online Aggregation*. SIGMOD 1997: 171-182.

[25] Oracle CEP. http://tinyurl.com/4gjlrkh.

[26] A. Raizman et al.: *An Extensible Test Framework for the Microsoft StreamInsight Query Processor*. DBTest 2010.