Physically Independent Stream Merging

Badrish Chandramouli[†] David Maier[‡] Jonathan Goldstein[*]

[†]*Microsoft Research, Redmond, USA* [‡]*Portland State University, Portland, USA* [*]*Microsoft Corporation, Redmond, USA*

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

***Abstract*—A facility for merging equivalent data streams can support multiple capabilities in a *data stream management system* (*DSMS*), such as query-plan switching and high availability. 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 in transmission or from combining multiple sources; and modifications 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 of them. 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. Unlike a traditional database, DSMS CQs can last for weeks. Tasks such as recovery, re-optimization, and load balancing are easier when individual queries are short lived (as in a transaction-processing system): 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 CQs while they continue to run is harder. We find that adding support in a DSMS for many such features is considerably simplified by introducing an operator – called *LMerge* (for *logical merge*) – to combine compatible versions of input event streams into a single equivalent output stream.

1) High Availability: Consider providing *high availability* (*HA*) [9, 15] for a CQ that involves a window of, say, 24-hour duration. Simply restarting the CQ 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 CQ running and being able to obtain 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 transparent resilience against $n-1$ simultaneous failures by instantiating n copies of a CQ (or CQ fragment) on different machines, feeding into an LMerge operator located at the consumer. LMerge provides a continuous stream of output events as long as at least one copy of the CQ is active, and allows input streams to be attached or detached as necessary.

2) Fast Availability: There is a need for "fast availability" for queries – obtaining output results with minimum latency. Using a carefully designed LMerge to combine (1) identical copies of a CQ 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 us to report answers from whichever copy is performing better at a given instant. LMerge can also hide burstiness, temporary performance degradations, and variability (e.g., due to network or CPU contention) in individual streams. Interestingly, if we need to run multiple copies of a CQ anyway for HA, we may choose to run different plans to get faster availability for "free".

3) Query Jumpstart: Another use of LMerge is to aid the process of "jumpstarting" query execution. Stream queries often hold long-lived events as part of their internal states. For example, a join operator might hold events for the current window (of 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 to do so). 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 new query operational sooner.

4) 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 multi-tenant execution, where we may wish to move CQs frequently based on SLAs and current workload conditions. We can place an LMerge operator (that allows us to attach and detach input streams dynamically) between the CQ and the user; making migration transparent to users as we attach a new CQ and later detach the old CQ from LMerge.

A. Challenges

LMerge is trivial if all the input streams present the same events 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.

Example 1 (DHCP Leases): Consider an example application for an ISP that tracks and reports *DHCP leases* of IP addresses to end users. Assume that DHCP allocations are tracked by two independent CQs on separate nodes, based on logs received from multiple distributed servers, either periodically or in real time. Based on the actual query plan, and the content and timing of data received, the two CQ outputs may appear physically different. Table I depicts leases for users A and B, as tracked by two streams Phy1 and Phy2. Here, *alloc(userId, start, end)* allocates a new lease to *userId*, for a duration from *start* to *end*, and *modify(userId, start, newEnd)* modifies – due to renewal or early termination – a current DHCP lease for *userId* to have an end time of *newEnd*. Rows of this table represent increasing instants of system time.

TABLE I
TRACKING DHCP LEASES

Phy1	Phy2
	alloc(A, 6, 7)
	alloc(B, 8, 15)
alloc(B, 8, ∞)	modify(A,6,12)
alloc(A, 6, 12)	
modify(B, 8, 10)	modify(B, 8, 10)

Two Streams Tracking DHCP Leases

UserId	Lease
A	[6, 12)
B	[8, 10)

Effective DHCP Lease Summary

Phy1 and Phy2 are *logically equivalent*, i.e., they report the same effective or eventual lease summary for users A and B, which is shown to the right in Table I. However, the streams are *physically different,* due to several reasons:

*1) **Disorder***: Phy1 reports a lease for user B followed by user A, whereas Phy2 first reports a lease for user A. Disorder is common in real streams [4, 6, 7, 28]; it may occur at the source, due to network congestion, transient router failures, or when combining events from multiple (possibly in-order) sources into a single output. For example, data for users A and B may have been collected at different servers before being sent to the two nodes hosting CQs for Phy1 and Phy2.

*2) **Revisions***: Phy1 directly reports the exact lease $[6,12)$ for A, whereas Phy2 reports a lease $[6,7)$ that it later revises to $[6,12)$. This situation can arise because the logs for user A may have been merged (or batched) before being sent to Phy1, whereas Phy2 happened to receive the individual allocations for user A. Revisions are common in practice due to noise, data-entry errors, and when improving accuracy during *online aggregation* [24]. Further, some CQs may not wish to incur the latency of waiting for a DHCP lease to end before reporting it, and may instead separately report the start (as I-streams [13], positive tuples [11], or inserts [4, 12, 22]), and later revise the report to include the correct end (as D-streams [13], negative tuples [11], or revisions/retractions [4, 10, 22]).

*3) **Processing Variations:*** If the sources are equivalent CQs with different physical plans (e.g., using different operators or join orderings), they may produce different outputs streams that eventually arrive at the same DHCP lease summary.

Phy1 and Phy2 show that a simple duplicate-eliminating set union cannot be used to merge equivalent streams. LMerge needs to be *physically independent*, i.e., unaffected by streams being physically different, as long as they are logically equivalent. Further, simply choosing to follow one of the input streams can prevent the timely output of events that another input stream has already produced, and can affect correctness if the chosen input fails. In order to be useful for our applications, LMerge should address additional challenges:

*1) **Avoiding Redundant Work***: The presence of LMerge usually involves the execution of redundant CQs at its inputs. One input might lag behind the others during periods when it is suboptimal or when its node suffers resource contention. If such redundant work can be avoided, then slower plan can catch up with the others.

*2) **Handling Failures***: Individual input streams can detach or re-attach to LMerge during runtime, e.g., due to machine failures or query-plan migration. The addition and removal of streams must be carried out carefully to avoid repeating or omitting events. Interestingly, the trivial counting merge outlined earlier does not work correctly when failures exist.

*3) **Other Challenges:*** Progress markers (such as heartbeats [6] and punctuation [1, 2, 22]) complicate the merging problem – we cannot propagate them without careful checks. Further, the volume of events output by LMerge needs to be carefully controlled. (Section II discusses both these issues in detail.)

Given the non-triviality of merging equivalent streams, one might consider enforcing order or applying revisions before feeding streams to LMerge. Unfortunately, this solution can affect throughput, memory and latency, sometimes by orders-of-magnitude. That said, a fully general LMerge can be demanding of CPU and memory. Hence, we wish to leverage *compile-time stream properties* of CQs to allow optimized LMerge algorithms. For example, a data source might guarantee in-order events. If a stream with non-decreasing timestamps passes through an aggregate (e.g., counting DHCP leases), we can infer that the output has strictly increasing timestamps. If the aggregation is *grouped* (e.g., performed for each user), we can infer that (userId, timestamp) is unique in the output stream. Static inference of such properties can significantly reduce the complexity and overhead of LMerge.

B. Contributions of this Paper

- We characterize LMerge in a general way that applies to many DSMSs, dealing with variations in stream semantics and representation. We formalize the requirements for correct LMerge, and propose output policies that meet those requirements (Secs. II & III).
- We present and analyze efficient algorithms for LMerge under different input-stream properties, and discuss how such properties may be derived from CQ plans. We also discuss policy choices for LMerge, handling missing events, and attaching or detaching streams (Secs. IV & V).
- We implemented our LMerge algorithms in Microsoft StreamInsight [22] and measured 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 simpler LMerge (Sec. VI).
- LMerge can easily support DSMS capabilities such as high availability, fast availability, query jumpstart, and query cutover. We show how LMerge can smoothly switch between streams that experience temporary congestion, to provide nearly steady throughput. We also show how LMerge can hide stream-rate variability, which

may arise due to load fluctuations, scheduling differences, and queuing delays (Sec. VI).

- We introduce *feedback signals* into LMerge, and show how LMerge can leverage such signals to "fast-forward" slower inputs and avoid unneeded work. We find that fast-forward with feedback can provide several times higher throughput than either LMerge without feedback, or running just a single CQ plan (Secs. V & VI).

II. Stream Formalism

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 our algorithms, 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 elements 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 the specific cases we consider later, $tdb(S)$ is 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 (just as there can be many physical structures that represent a given logical table in a database). 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. The range of possible descriptions of the same TDB in a given DSMS 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. As an example of an ordering constraint, most stream systems support some form of punctuations that limit stream elements that can appear later. We next give examples of two DSMS models in terms of elements supported.

Example 2 (Open and Close Elements): Consider a stream with 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. (We assume here that there can only be one event with payload p active at a time.) Open and close elements roughly correspond to I-Streams and D-Streams in Oracle CEP [25], 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)
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)`.

Example 3 (StreamInsight): Microsoft StreamInsight, the basis for our detailed algorithms (Section IV) and implementation (Section VI), 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 $\langle p, V_s, V_e\rangle$.

TABLE II
Physical and Logical Streams in StreamInsight

Phy1	Phy2
	ins(A, 6, 7)
	ins(B, 8, 15)
ins(B, 8, ∞)	adj(A,6,7, 12)
ins(A, 6, 12)	
adj(B, 8, ∞, 10)	adj(B, 8, 15, 10)
stable(11)	
stable(∞)	stable(∞)

Two Physical Streams

Payload p	Interval $[V_s, V_e)$
A	[6, 12)
B	[8, 10)

Equivalent Logical TDB

Abbreviations Used
ins → insert
adj → adjust

[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.

Table II above shows the two physical streams from Example 1 (DHCP leases), and the corresponding TDB. As before, the rows of this table represent increasing instants of system time.

Additional Challenges with LMerge

1) 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 Table II to illustrate that this problem is non-trivial. Assume that LMerge has chosen to only propagate elements insert(A, 6, 7) and insert(B, 8, 15) from Phy2 to its output. When it later sees stable(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.

2) Stream Chattiness: LMerge needs to select output policies that balance responsiveness against "chattiness" – the need to issue additional output elements to modify previous elements.

Example 4 (Stream Chattiness): Table III shows two input streams, In1 and In2, and three alternative output streams for LMerge. 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 stable. 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 stable.

TABLE III
CHATTINESS: INPUT AND OUTPUT STREAMS

In1	In2	Out1	Out2	Out3
ins(A, 6, 10)		ins(A, 6, 10)		ins(A, 6, 10)
	ins(A, 6, 12)	adj(A,6,10,12)		
	ins(B, 7, 14)	ins(B, 7, 14)		ins(B, 7, 14)
adj(A,6,10,15)		adj(A,6,12,15)		
	adj(A,6,12,15)			
	stable(16)	stable(16)	ins(A, 6, 15) ins(B, 7, 14) stable(16)	adj(A,6,10,15) stable(16)

III. THEORY OF LOGICAL MERGE

A. 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 portion. 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, $1 \le i \le n$ 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 may not 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 assume that the element kinds and the properties are the same for all inputs and output, although one could obviously relax this constraint.

A. Stream Properties and Logical Merge

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 2, `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 $V_s \le t$. Further, no TDB event 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*: This 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 are detected through compile-time analysis of CQ plans (Section IV-D has more details). 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 example shows.

Example 5 (Stream Properties and Compatibility): Consider streams with open() and close() elements and the simple 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 $\text{close}(p, V_e) \notin I[k]$, there is no way to extend $O[j]$ to be equivalent with $I[k]: \text{close}(p, V_e + 1)$, since $O[j]$ already contains a close element for p. In the case of a set of mutually consistent inputs $\boldsymbol{I} = \{I_1[k_1], \ldots, I_n[k_n]\}$, $O[j]$ is compatible with $\boldsymbol{I}$ exactly when $O[j] \subseteq (\cup \boldsymbol{I})$.

B. Property-Based Restrictions and Compatibility

For our prototypes of LMerge, we consider the following range of restrictions that can be used to improve performance if they hold. In Section IV, 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. Thus, 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 element kinds 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 $tdb(S, i)$.

R4. R4 is the "no restrictions" case where all three element kinds are permitted, elements need not be in timestamp order, and the TDB is a multi-set (hence there can be multiple events with the same payload and lifetime).

In order to understand the correctness of our algorithms, we find it useful to think of a $\text{stable}(V_c)$ element as "freezing" certain parts of the TDB. A TDB event $\langle p, V_s, V_c \rangle$ is *half frozen* (*HF*) if $V_s < V_c \le V_e$ 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*).

Compatibility 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 applies 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 $\text{stable}(V)$ 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	**HF**
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 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 $\text{stable}(L_m)$ was the most recent stable() event on I_m, and $\text{stable}(L)$ 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 have 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 $L_m \le L$ (so the output event can be adjusted to match any changes in TDB_m) or the event is FF and

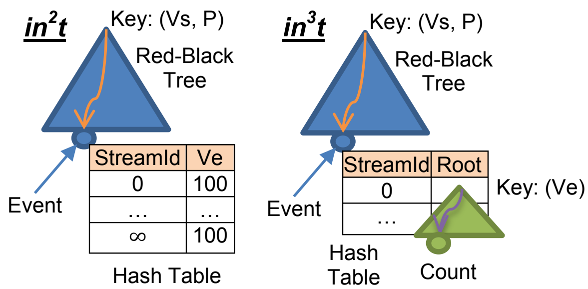

Fig. 1. Data structures for cases R3 (in^2t) and R4 (in^3t) of LMerge

$L \leq V_m$ (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 when either:

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

- $L \leq V_s$ (thus, 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 (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:

- $L \leq V_s$ (so 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 input with 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 allow arbitrary UF events in the output, even if there is no support among the inputs for such events. This policy is likely unwise, since such events would almost surely be adjusted, absent a robust model for predicting future inputs. A more reasonable policy is to output only HF and FF events that have support in the TDBs of the inputs. That support might take the form of an exactly matching event, or, for HF events in the output TDB, a HF 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 FF event in some input TDB. These different policies tend to trade latency for "chattiness" of the output: how many adjust() elements might be needed to align the output with adjustments on the inputs. A second aspect of output policy is when to issue a stable() element on the output. Our experience suggests keeping the output at the maximum stable point of all the inputs to minimize the memory requirements of LMerge, though lagging a bit behind the maximum can 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

This section provides algorithms for different variants of LMerge optimized for stream properties R0-R4 described in Section III-B. Section IV-D shows how to use stream properties to decide which algorithm to use for a given CQ.

A. LMerge Algorithms for Cases R0, R1, and R2

For space reasons, we describe briefly the simpler cases of R0, R1, and R2 (our technical report [27] has the detailed algorithms for these cases).

In case R0, input streams have elements with strictly increasing `Vs` values. 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. When we see an insert() element, we can discard the element if it does not increase `MaxVs`, and output it otherwise. A stable() element is output if it increases `MaxStable`.

Recall R1 is the 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, we just need to maintain (in addition to `MaxStable` and `MaxVs`) an array with one counter for each input stream, which counts the number of elements on that stream with `Vs = MaxVs`. On an insert element that increases `MaxVs`, we reset this array to zeros. If the insert on stream `r` increases the counter for `r` beyond the old maximum counter value across all streams, the insert is sent as output. A stable() element is handled as before.

Case R2 resembles R1, except elements with the same `Vs` may be in different orders in different inputs. We assume that (`Vs`, `Payload`) is a key of the TDB for any stream prefix. (The relaxation to handle duplicates is straightforward and omitted.) Here, we use 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. An element that increases `Vs` beyond `MaxVs` clears the hash table so that it can track elements with the new `MaxVs`.

B. 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^2t (for *index-2-tier*) depicted in Figure 1 (left). The top tier of in^2t is a red-black-

```
Algorithm R3: Logical Merge for Case R3
 1 MaxStable = −∞;
 2 index = new in2t();
 3 Insert(element y, stream r)
 4   node f = index.SameVsPayload(y);
 5   if (!exists(f))
 6     if (y.Vs < MaxStable) return;
 7     f = index.AddNode(y);          2
 8     OutputInsert(y);
 9     f.AddHashEntry(∞, y.Ve); // hash entry for o/p
10   f.AddHashEntry(r, y.Ve); // hash entry for i/p
11 Adjust(element y, stream r)
12   node f = index.SameVsPayload(y);
13   if (!exists(f)) return;               1
14   f.UpdateHashEntry(r, y.Ve);
15 Stable(timestamp t, stream r)
16   if (t <= MaxStable) return;
17   iterator it = index.FindHalfFrozen(t);
18   while (node f = it.Next())
19     InVe = f.GetHashEntry(r);
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   // update MaxStable and output a stable() element
29   MaxStable = t;
30   OutputStable(t);
```

```
Algorithm R4: Logical Merge for Case R4
 1 MaxStable = −∞;
 2 index = new in3t();
 3 Insert(element y, stream r)
 4   node f = index.SameVsPayload(y);
 5   if (!exists(f))
 6     if (y.Vs < MaxStable) return;
 7     f = index.AddNode(y);
 8   f.IncrementCount(r, y.Ve);
 9   if ((y.Vs>=MaxStable) and (f.GetCount(r)>f.GetCount(∞)))
10     OutputInsert(y);
11     f.IncrementCount(∞, y.Ve);
12 Adjust(element y, stream r)
13   node f = index.SameVsPayload(y);
14   if (!exists(f)) return;
15   f.IncrementCount(r, y.Ve); f.DecrementCount(r, y.Vold);
16 Stable(timestamp t, stream r)
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(r);
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(r) < t) // Done processing that (Vs, P)
28        index.Delete(f);
29   MaxStable = t;
30   OutputStable(t);
```

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 `r`, the current corresponding `Ve` value for that stream indexed by key `r`. An additional hash table entry with special key ∞ is also maintained for the output.

On an insert() element in stream `r`, we lookup in^2t 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 `r` 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^2t. 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 `y`. We only need to handle stable() elements that increase `MaxStable`. We first find each node that is going to become half frozen in in^2t; i.e., a node whose `Vs` is less than `y`'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 `y`. There are three cases of compatibility violations:

- There is no input event for (`Vs`, `Payload`) in stream `r`, 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 V-A. Finally, if the input becomes fully frozen, we delete the corresponding node from in^2t (Lines 28-29), update `MaxStable`, and output a stable() element (Lines 30-31).

C. *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 hash table 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. Our invariants for case R4 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 a (`Vs`, `Payload`) that is getting half frozen, the "total count" of output elements with a value of (`Vs`, `Payload`) equals the count in the input. This invariant must be met before propagating the stable() element, to guarantee future convergence. The

method `AdjustOutputCount()` determines the exact procedure for meeting this invariant (see [27] for details); briefly, it involves producing new output elements or "canceling" prior output elements for that (`Vs`, `Payload`).

- (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 the output TDB contains the same number of (`Vs`, `Payload`, `Ve`) events as the input, before propagating the stable() element. The `AdjustOutput()` method (covered in our technical report [27]) achieves this invariant; briefly, it involves adjusting the `Ve` of events previously output with the same (`Vs`, `Payload`). Note that the "total count" invariant mentioned earlier ensures that such an adjustment is always possible.

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).

D. Choosing the Right LMerge Algorithm

Given a range of LMerge algorithms, 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* to answer this question. We do not give a detailed formalism of stream properties here, but provide several examples of how they 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 operators to enforce particular 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 algorithm.

3) Certain operators or groups of operators produce streams with a particular property. For example, an in-order stream fed into a windowed aggregate 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 per user of DHCP leases) 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.

E. Runtime and Space 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 n 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(n)	n/a	O(1)	O(n)
R2	O(n)	n/a	O(1)	O($g \cdot p$)
R3	O($\lg w$)	O($\lg w$)	O($c \cdot \lg w + h$)	O($w(p+n)$)
R4	O($\lg w + \lg d$)	O($\lg w + \lg d$)	O($c \cdot \lg w + h \cdot d$)	O($w(p + n \cdot d)$)

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 are free 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. Some alternatives include:

- We can reflect every adjust() element at the output. This choice makes LMerge more "chatty", but allows a listener to process such changes earlier if it is interested.
- Force LMerge to "follow" a particular input stream, for example, the stream with the currently maximum stable() timestamp (the *leading stream*). This choice may be appropriate when one stream leads for long periods. However, if the leading stream keeps changing, this policy can incur significant overhead in re-adjusting output. Even in this case, LMerge must track information from other inputs to handle the case where the leading stream detaches.

Another point for choosing a different policy is location 2 in Algorithm R3. 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. Handling Joining and Leaving Input 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 in Input Streams

If we require that LMerge output contains an element if some input stream reports it, LMerge is forced to progress (issue stable() elements) only as fast as the most slowly 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 `y` as long as the stream that increases `MaxStable` beyond `y.Vs` produces element `y`.

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 can modify LMerge to signal its input plans that elements before a certain time t are no longer of interest. This modification permits 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 can exploit feedback signal locally, and optionally propagate the signal further upstream in the plan. This capability allows a slower plan to "fast-forward", possibly becoming the leading stream. 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, if the requisite stream properties hold.
2) We compare the strategy of enforcing stream properties and using simpler versions of LMerge, against directly using a more general version of LMerge.
3) We apply LMerge in several motivating scenarios: fast availability, network-congestion masking, and dynamic plan selection with feedback signals.

A. Setup and Implementation

We implement our algorithms in StreamInsight. We perform our experiments on an 8-core machine with two 2.33GHz processors and 16GB main memory running Windows Server 2008 R2. We implemented and evaluated all our proposed LMerge variants. LM^{R0}, LM^{R1}, LM^{R2}, LM^{R3+}, and LM^{R4} correspond to operators that implement the algorithms for cases R0, R1, R2, R3, and R4 respectively, from Section IV. We also evaluate a simpler algorithm for case R3, called LM^{R3-}, in which events from each input stream are held in a *separate* index, with another index for 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 simpler to implement, it duplicates event information across input streams and requires multiple tree lookups at runtime.

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 (Sections I, V, 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 last metric quantifies the chattiness of the stream.

Our evaluation mostly used synthetically generated datasets [2] from our commercial-grade test-stream generator [26]. Each event has two fields, an integer in the interval [0, 400] and a random 1000-byte string. The event generator produces between 200K and 400K elements, based on a set of supplied parameters (see [27] for more details), including:

- *StableFreq*: The probability that an element in the stream is a stable() element. The default value 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. The default value for this parameter 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-

[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.

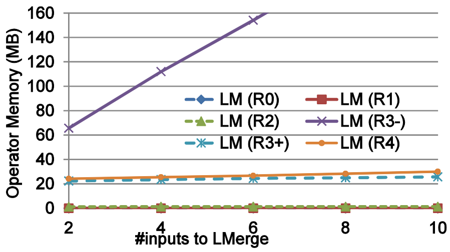

Fig. 2. Memory, in-order input streams

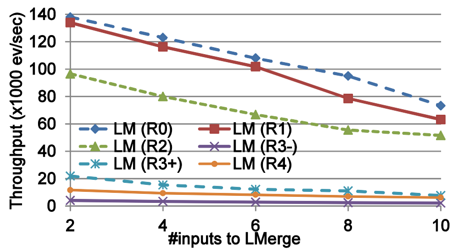

Fig. 3. Throughput, in-order streams

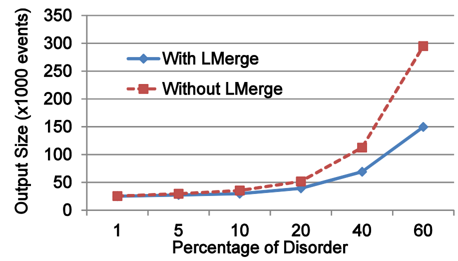

Fig. 4. Output size, increasing disorder

generator output in order to generate them, such as an 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 Using an ordered stream without adjust() elements, we can compare 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 2 for LM^{R0}, LM^{R1}, and LM^{R2} overlap as they perform similarly.) LM^{R3+} incurs slightly more memory than the simpler versions, but the cost is almost independent of the number of inputs, as it shares event payloads across inputs. In contrast, memory usage of LM^{R3-} degrades linearly with the number of input streams, due to duplication across 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 also experimented with introducing lag (or delay) in some of the input streams to LMerge. Our technical report [27] has the details; briefly, as lag increases, LMerge throughput improves since it can directly drop tuples and thus hide the lag in the slower streams (We also 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 5 (left) that memory usage decreases as expected, due to more frequent cleanup. On the other hand, the throughput for LM^{R3+} and LM^{R4} decreases as shown in Figure 5 (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*

Since LMerge algorithms are significantly simplified for special cases where the stream satisfies specific properties, we investigate enforcing these properties before feeding streams to the simpler versions of LMerge tailored to 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 6 (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 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 6 (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.

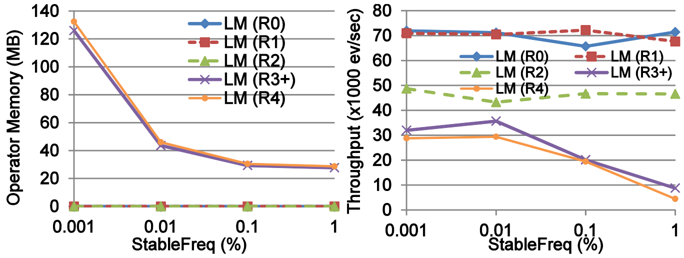

Fig. 5. Memory and throughput, increasing StableFreq

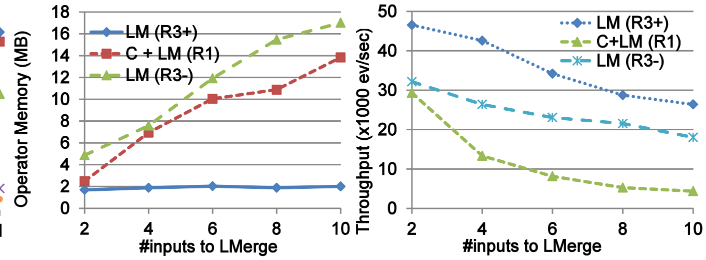

Fig. 6. Memory and throughput, enforcing stream properties

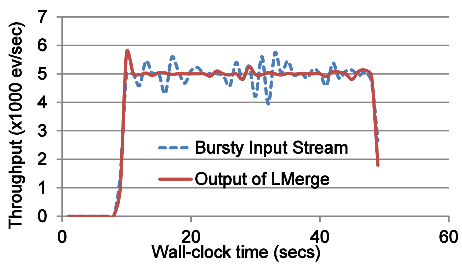

Fig. 7. Handling bursty streams

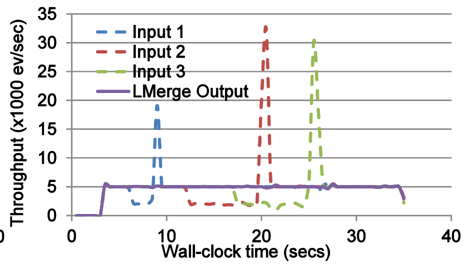

Fig. 8. Masking network congestion

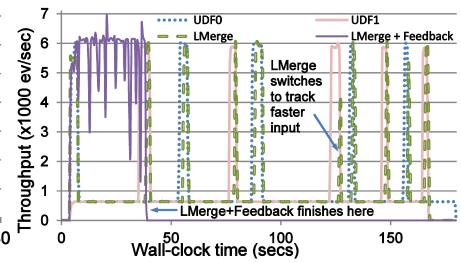

Fig. 9. Plan switching with fast-forward

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

E. Evaluating LMerge Scenarios

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 because of CPU load and resource variations on machines, garbage collection, scheduling vagaries, or 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 7 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 having a burst 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 when conditions return back to normal. Figure 8 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 can mask 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 I 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, so the "optimal" plan switches 9 times during execution. We show the performance of these queries individually (without LMerge) in Figure 9, 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 ~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 in 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 for windows, but single timestamps on events. In Nile [11], positive tuples begin new events while negative tuples expire older events. In Sections II and III, we presented 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 2. 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 real-world stream models, propose a suite of algorithms for LMerge, leverage stream properties and feedback signals for efficiency, and examine LMerge in scenarios 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 seminal work by Zhu et al. [20]. Their approach determines 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 is hard to satisfy under 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 provides a similar capability by running the alternative plans together with 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 to combine equivalent input streams that are physically divergent and fallible. Our LMerge definition applies to any DSMS in which a stream represents (implicitly or explicitly) 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 and showed how to leverage stream properties to choose the right variant for a given query. We proposed a new technique to fast-forward slower inputs to LMerge using feedback signals.

A detailed evaluation demonstrated the differences between the LMerge variants in terms of throughput and memory, as well as their response to various stream characteristics. We also found that it is beneficial to use a general LMerge instead of explicitly enforcing the stricter input properties that the more constrained LMerge variants require. We demonstrated the utility of LMerge for scenarios with bursty input, where LMerge can smooth out variability. We also used LMerge for fast availability and found that using feedback signals to fast-forward slower plans can significantly improve 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.

[27] B. Chandramouli, D. Maier, J. Goldstein: *Physically Independent Stream Merging.* Technical Report, Microsoft Research (MSR-TR-2011-82). http://research.microsoft.com/apps/pubs/?id=151428.

[28] M. Liu et al.: *Sequence pattern query processing over out-of-order event streams.* ICDE, 2009.