Handling Shared, Mutable State in Stream Processing with Correctness Guarantees

Nesime Tatbul[1,2], Stan Zdonik[3], John Meehan[3], Cansu Aslantas[3],
Michael Stonebraker[2], Kristin Tufte[4], Chris Giossi[4], Hong Quach[4]
[1]Intel Labs [2]MIT [3]Brown University [4]Portland State University
{tatbul,stonebraker}@csail.mit.edu, {sbz,john,cpa}@cs.brown.edu, {tufte,cgiossi,htquach}@pdx.edu

Abstract

S-Store is a next-generation stream processing system that is being developed at Brown, Intel, MIT, and Portland State University. It is designed to achieve very high throughput, while maintaining a number of correctness guarantees required to handle shared, mutable state in streaming applications. This paper explores these correctness criteria and describes how S-Store achieves them, including a new model of stream processing that provides support for ACID transactions.

1 Introduction

Stream processing has been around for a long time. Over a decade ago, the database community explored the topic of near-real-time processing by building a number of prototype systems [6, 9, 15]. These systems were based on a variant of the standard relational operators that were modified to deal with the unbounded nature of streams.

Additionally, streaming applications require support for storage and historical queries. In our view, the early systems did not properly address storage-related issues. In particular, they largely ignored the handling of shared, mutable state. They were missing the guarantees that one would expect of any serious OLTP DBMS. These correctness guarantees are needed in addition to those that streaming systems typically provide, such as exactly-once processing (which requires that, upon recovery, the system will not lose or duplicate data).

We believe that it is time to take a look at streaming through the lens of these processing guarantees. In this paper, we present S-Store, which is designed to address the correctness aspects of streaming applications. We show that it is possible to support correctness without serious performance degradation. We also show that the only way to achieve good performance is by tightly integrating storage management with the streaming infrastructure. Some modern streaming systems require the use of an external storage manager to provide needed services [2, 3, 22, 27]. As we will show, using external storage comes at a cost.

We begin with describing a motivating use case, and proceed to discuss S-Store's correctness guarantees, computational model and implementation to achieve these guarantees, followed by an experimental comparison with the state of the art.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

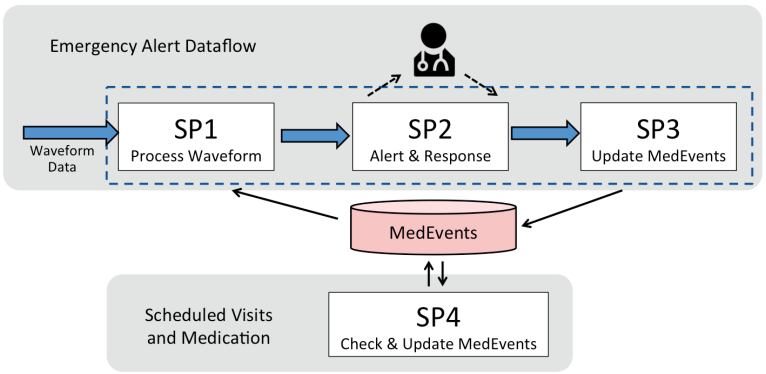


Figure 1: Multiple Streaming Dataflows Sharing State: A MIMIC-based Example [16, 26]

2 Example Use Case

In this section, we present a selected use case of S-Store, based on a recent demonstration of the BigDAWG polystore system [16], in which S-Store was used for real-time alert monitoring over streaming patient waveforms from an ICU (intensive care unit). This use case illustrates the need for consistently managing state shared among multiple streaming dataflows.

In a hospital environment, proper tracking of patient medications is critical to avoid overdoses and bad drug interactions. Studies have estimated that preventable "adverse drug events" (with patient injury) in hospitals to be between 380,000 and 450,000 per year [20]. We observe that different types of events may trigger medication administration: an emergency alert, a periodic doctor visit, or a periodic medication administration schedule. These events all require reading and updating of the list of medication administrations for a patient. In the MIMIC ICU data set [26], this data is stored in a medication events (*MedEvents*) table. Thus, separate dataflow graphs must update a single shared table, which requires transactional consistency to avoid patient injury.

Figure 1 diagrams potential S-Store dataflow graphs that update the *MedEvents* table. The upper dataflow represents an emergency alert notification, while the lower dataflow represents periodic doctor visits and medication administrations. In the emergency alert dataflow, a stored procedure (*SP1*) reads incoming patient waveform data (e.g., Pulmonary Arterial Pressure (PAP)), and calculates a windowed average over it. When this average is elevated, a doctor must be notified and medication may be recommended; however, medication must not be recommended if the medication has been recently administered. The doctor then either accepts or declines the recommendation, and the *MedEvents* table is updated appropriately. In the periodic-visits dataflow, a doctor or a schedule decides which medication is advisable. Before administering the medication, the caregiver enters the medication to be administered. The system then checks for potential drug interactions with recent medication administrations using the *MedEvents* table, and then updates *MedEvents* appropriately. This simpler dataflow is similar in nature to an OLTP transaction.

For ensuring correct semantics, this example requires ordered execution of its dataflows and transactional (ACID) access to the *MedEvents* table. More specifically, *SP1* must read the *MedEvents* table before an alert is sent to the doctor; the *MedEvents* table needs to remain locked so that other updates – such as from *SP4* – cannot interfere. Thus, *SP1*, *SP2*, and *SP3* must be part of an ordered dataflow within a single nested transaction. Furthermore, *SP1*, *SP2*, *SP3* cannot be a single stored procedure due to the human interaction. Note that this example could be extended with other similar emergency-alert dataflows, as different types of analysis are needed on different waveform streams, e.g., cardiac anomalies to be detected from ECG-waveform streams.

A similar workload pattern can be found in other domains such as transportation, wherein one or more shared tables must be read and updated by multiple dataflows, as might be seen in the display of messages on Variable Message Signs and Adaptive Signal Control. In this case, transactional processing support would be

required to avoid inconsistencies, garbled messages, and incorrect signal timing. We note that in most of these examples, the dataflows are fully automated (i.e., human-in-the-loop is not necessarily a critical requirement as in the medical setting).

3 Correctness

Transaction-processing systems normally provide *ACID (Atomicity, Consistency, Isolation, and Durability)* guarantees. These guarantees broadly protect against data corruption of two kinds: (i) interference of concurrent transactions, and (ii) transaction failures. Consistency and Isolation primarily address interference, while Atomicity and Durability address failures. It is widely understood that failures can cause data inconsistencies. Thus, most stream processing engines also cover this case by incorporating failure-recovery facilities. However, it is less widely acknowledged that any streaming computation that shares mutable data with other computations (e.g., a separate streaming dataflow graph) must guard against interference from those computations as in standard OLTP.

In addition to ACID, there are other correctness requirements from stream processing that must be considered. First, a transaction execution must conform to some logical *order* specified by the user. The scheduler should be free to produce a schedule that interleaves transactions in a variety of ways, but the results must be equivalent to the specified logical order. Secondly, it has been shown that, in streaming systems, failures may lead to lost or duplicated tuples. It puts a burden on the application to detect and react to such problems appropriately. Thus, streaming systems typically strive to provide *exactly-once* semantics as part of their fault-tolerance mechanisms.

For correctly handling hybrid workloads, S-Store provides efficient scheduling and recovery mechanisms that maintain three complementary correctness guarantees that are needed by both streaming and transactional processing. In what follows, we discuss these guarantees.

3.1 ACID Guarantees

We regard a transaction as the basic unit of computation. As in conventional OLTP, a transaction T must take a database from one consistent state to another. In S-Store, the database state consists of streaming data (*streams* and *windows*) in addition to non-streaming data (*tables*). Accordingly, we make a distinction between two types of transactions: (i) *OLTP transactions* that only access tables, and are activated by explicit transaction requests from a client, and (ii) *streaming transactions* that access streams and windows as well as tables, and are activated by the arrival of new data on their input streams. Both types of transactions are subject to the same interference and failure issues discussed above. Thus, first and foremost, S-Store strives to provide ACID guarantees for individual OLTP and streaming transactions in the same way traditional OLTP systems do. Furthermore, access to streams and windows require additional isolation restrictions, in order to ensure that such streaming state is not publicly available to arbitrary transactions that might endanger the streaming semantics.

3.2 Ordered Execution Guarantees

Stream-based computation requires ordered execution for two primary reasons: (i) streaming data itself has an inherent order (e.g., timestamps indicating order of occurrence or arrival), and (ii) processing over streaming data has to follow a number of consecutive steps (e.g., expressed as directed acyclic dataflow graphs as illustared in Figure 1). Respecting (i) is important for achieving correct semantics for order-sensitive operations such as sliding windows. Likewise, respecting (ii) is important for achieving correctness for complex dataflow graphs as a whole.

Traditional ACID-based models do not provide any order-related guarantees. In fact, transactions can be executed in any order as long as the result is equivalent to a serial schedule. Therefore, S-Store provides an ad-

ditional correctness guarantee that ensures that every transaction schedule meets the following two constraints: (i) for a given streaming transaction T, atomic batches of an input stream S must be processed in order (a.k.a., stream order constraint), and (ii) for a given atomic batch of stream S that is input to a dataflow graph G, transactions that constitute G must be processed in a valid topological order of G (a.k.a., dataflow order constraint).

For coarser-grained isolation, S-Store also allows the user to define nested transactions as part of a dataflow graph (e.g., see the *Emergency Alert Dataflow* in Figure 1), which may introduce additional ordering constraints [23]. S-Store's scheduler takes all of these constraints into account in order to create correct execution schedules.

3.3 Exactly-once Processing Guarantees

Failures in streaming applications may lead to lost state. Furthermore, recovering from failures typically involves replicating and replaying streaming state, which, if not applied with care, may lead to redundant executions and duplicated state. To avoid these problems, streaming systems strive to provide fault tolerance mechanisms that will ensure "exactly-once" semantics. Note that exactly-once may refer either (i) to external delivery of streaming results, or (ii) to processing of streams within the system. The former typically implies the latter, but the latter not necessarily implies the former. In this work, we have so far mainly focused on the latter (i.e., exactly-once processing, not delivery), as that is more directly relevant in terms of database state management.

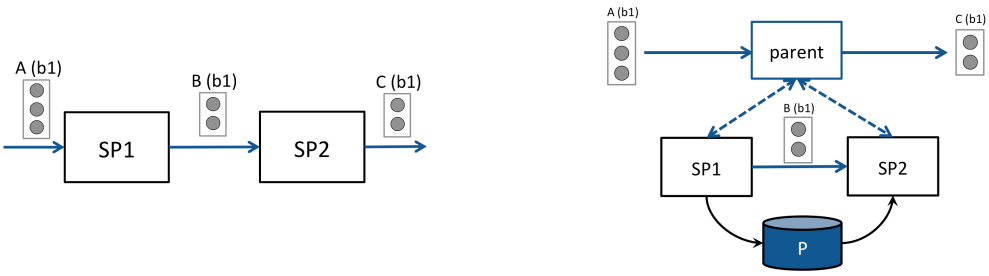
Exactly-once processing is not a concern in traditional OLTP. Any failed transaction that was partially executed is undone (Atomicity), and it is up to the user to reinvoke such a transaction (i.e., the system is not responsible for loss due to such transactions). On the other hand, any committed transaction that was not permanently recorded must be redone by the system (Durability). State duplication is not an issue, since successful transactions are made durable effectively only once. This approach alone is not sufficient to ensure exactly-once processing in case of streaming transactions, mainly because of the order and data dependencies among transaction executions. First, any failed transaction must be explicitly reinvoked to ensure continuity of the execution without any data loss. Second, it must be ensured that redoing a committed transaction does not lead to redundant invocations on others that depend on it.

S-Store provides exactly-once processing guarantees for all streaming state kept in the database. This guarantee ensures that each atomic batch on a given stream S that is an input to a streaming transaction T is processed exactly once by T. Note that such a transaction execution, once it commits, will likely modify the database state (streams, windows, or tables). Thus, even if a failure happens and some transaction executions are undone or redone during recovery, the database state must be "equivalent" to one that is as if S were processed exactly once by T.

Note that executing a streaming transaction may have an external side effect other than modifying the database state (e.g., delivering an output tuple to a sink that is external to S-Store). It is possible that such a side effect may get executed multiple times during recovery. Thus, our exactly-once processing guarantee applies only to state that is internal to S-Store. This is similar to other exactly-once processing systems such as Spark Streaming [28]. Exactly-once delivery might also be important in some application scenarios (e.g., dataflow graphs that involve a human-in-the-loop computation as in the medical use case described in Section 2). We plan to investigate this guarantee in more detail as part of our future work.

4 Model Overview

We now describe our model, which allows us to seamlessly mix OLTP transactions and streaming transactions. The basic computational unit in S-Store is a transaction, and all transactions are pre-declared as stored procedures. A stored procedure is written in both SQL (to interact with tables that store database state) and in Java (to allow arbitrary processing logic). Streaming transactions are those that take finite batches of tuples from streams as input and may produce finite batches of tuples as output. As one would expect, all transactions (streaming or



(a) Processing of Atomic Batches

(b) Nested Transactions

Figure 2: Example Dataflows

not), preserve the standard ACID properties of OLTP database systems.

As mentioned earlier, S-Store manages three kinds of state: (i) streams, (ii) windows, and (iii) tables. S-Store models a stream as an unbounded sequence of tuples. These tuples arrive in some order and are processed in chunks (called *atomic batches*). An atomic batch is a contiguous, non-overlapping subsequence of a stream in which all tuples in the batch share a common batch-id. A typical example is to group tuples with a common application timestamp or time-interval into the same batch [9, 28]. We assume that batches over a stream should be processed in ascending order of their batch-id's; however the order of tuples within a single batch does not matter since each batch is always processed as an indivisible atomic unit.

A window over a stream is also a contiguous subsequence of that stream, but different from atomic batches, windows come with a set of rules for deriving a new window from an old one. Windows are defined in units of batches (as opposed to time or tuple count), and can slide and tumble much as in previous generations of streaming systems [11], so, we will not go into detail here. It is important to note that defining windows in batch units ensures that windows are processed in a deterministic way, avoiding the "evaporating tuples" problem discussed in previous work [9, 12].

Streams, windows, and tables differ in terms of which transactions are allowed to access them. Tables can be publicly read or written by any transaction, while windows are private to the transaction in which they are defined, and streams are private to their "producer" and "consumer" transactions.

Steaming systems typically push data from input to output. This arrangement reduces latency, since there is no need to poll the inputs to determine if the next input batch is ready. S-Store, like other systems, captures the notion of the next action to perform through a dataflow graph. In the case of S-Store, the actions are transactions, making the dataflow graph a DAG with transaction definitions as nodes, and a directed arc from node T_i to node T_j if T_j should follow T_i in the processing order. That is to say, when T_i commits, T_j should be triggered next.

Figure 2a shows a two-stored-procedure (i.e., *SP1* and *SP2*) dataflow graph. The batch of tuples labeled *A* is the input to *SP1*, all with the same batch-id *b1*. *SP1* begins execution as a transaction with the 3-tuple batch as input. Suppose that *SP1* commits with the batch labeled *B* as output. The tuples in batch *B* would be assigned the batch-id of the inputs that they were derived from (*b1*), and the process repeats with batch *B* as input to *SP2* and batch *C* as the output batch for *SP2*.

Stored procedures that take multiple streams as input or emit multiple streams as output are processed in a similar way. In this case, a stored procedure begins execution with atomic batches from all of its input streams with a common batch-id and the same batch-id carries over to any output batches that result from this execution.

For each transaction definition, there could be many transaction executions (TEs). If stream *S* is the input to transaction *T*, a TE is created every time a new batch of tuples arrives on stream *S*. Windows are created in TEs. Since they are the principal data structure that reacts to the unbounded nature of a stream, the i^{th} TE for a transaction *T* will inherit any window state that is active in the $(i-1)^{st}$ TE for *T*. Aside from this exception,

windows are private and cannot be shared with TEs for other transactions, since that would break the isolation requirement for ACID transactions. Similarly, streams can only be shared by the TE's of their producer and consumer transactions in a dataflow (e.g., only TE's of *SP1* and *SP2* can share the stream that flows between them in Figure 2a).

We also provide a nested transaction facility that allows the application programmer to build higher-level transactions out of smaller ones, giving her the ability to create coarser isolation units among transactions, as illustrated in Figure 2b. In this example, two streaming transactions, *SP1* and *SP2*, in a dataflow graph access a shared table *P*. *SP1* writes to the table and *SP2* reads from it. If another OLTP transaction also writes to *P* in a way to interleave between *SP1* and *SP2*, then *SP2* may get unexpected results. Creating a nested transaction with *SP1* and *SP2* as its children will isolate the behavior of *SP1* and *SP2* as a group from other transactions (i.e., other OLTP or streaming). Note that nested transactions also isolate multiple instances of a given streaming dataflow graph (or subgraph) from one another.

S-Store transactions can be executed in any order as long as this order obeys the ordering constraints imposed by: (i) the relative order of atomic batches on each input stream, (ii) the topological ordering of the stored procedures in the dataflow graph, (iii) any additional constraints due to nested transactions. Assuming that transaction definitions themselves are deterministic, this is the only source of potential non-determinism in S-Store transaction schedules. For example, for the simple dataflow in Figure 2a, both of the following would be valid schedules: $[TE1(b1); TE1(b2); TE2(b1); TE2(b2)]$ or $[TE1(b1); TE2(b1); TE1(b2); TE2(b2)]$. On the other hand, for the dataflow in Figure 2b, the former schedule would not be allowed due to the nesting.

A more detailed description of our model can be found in a recent publication [23].

5 Implementation

Our S-Store implementation seeks to prove that we can provide all of the correctness guarantees mentioned above without major loss of performance. Implementation of the mechanisms to provide these guarantees must be native to the system to minimize overhead.

S-Store is built on top of H-Store [21], a high-throughput main-memory OLTP system, in order to take advantage of its extremely light-weight transaction mechanism. Thus, like H-Store, S-Store follows a typical two-layer distributed DBMS architecture (see Figure 3). Transactions are initiated in the partition engine (PE), which is responsible for managing transaction distribution, scheduling, coordination, and recovery. The PE manages the use of another layer called the execution engine (EE), which is responsible for the local execution of SQL queries. As mentioned earlier, transactions are predefined as stored procedures which are composed of Java and SQL statements. When a stored procedure is invoked with input parameters, a transaction execution (TE) is instantiated and runs to completion before committing. A client program connects to the PE via a stored-procedure execution request. If the stored procedure requires SQL processing, then the EE is invoked with those sub-requests.

While we chose H-Store to serve as our foundation, our architectural extensions and mechanisms could be implemented on any main-memory OLTP engine, thereby directly inheriting the required ACID guarantees discussed in Section 3.1. We are able to achieve our desired correctness guarantees due to the implementation additions described in the following subsections.

5.1 ACID Implementation

In order to maintain the transactional properties inherited from H-Store, we implement our dataflow graph as a series of stored procedures connected by streams. All streaming state, including both **streams** and **windows**, are implemented as time-varying tables, which are accessed within stored procedures. Thus, it is impossible to access streaming state in a non-transactional manner.

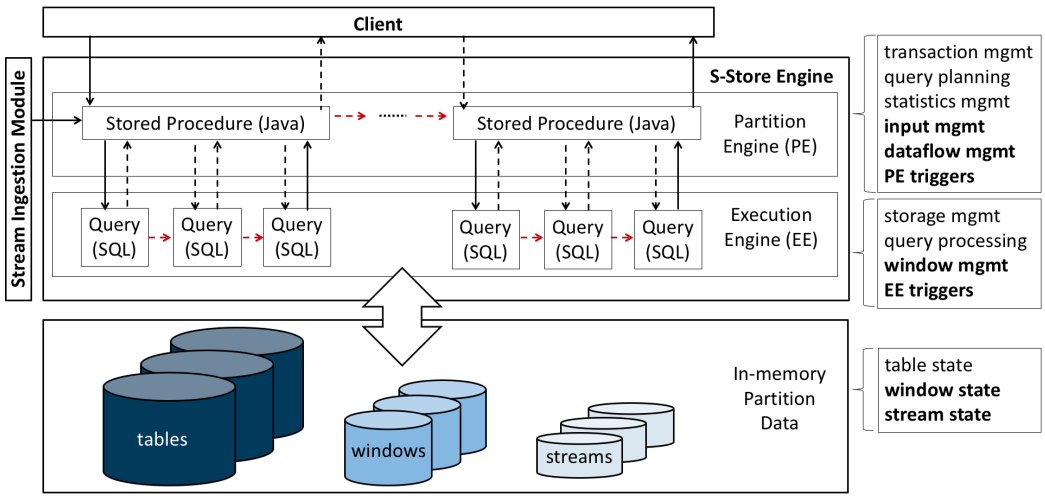


Figure 3: S-Store Architecture

The stored procedures within the dataflow are connected by streams, and activated via **partition engine (PE) triggers**. When a transaction commits and places a new batch of tuples into its output stream, any downstream transactions in the dataflow are immediately scheduled for execution using that output batch as their input.

In addition to PE triggers, S-Store includes **execution engine (EE) triggers**. These allow SQL statements to be invoked upon the availability of a new tuple in a stream or the slide of a window. Unlike PE triggers, EE triggers execute within the same transaction as the insertion that activated them.

5.2 Ordering Implementation

Because S-Store breaks down a dataflow into multiple discrete stored procedures, multiple simultaneous transaction requests must be scheduled in such a way that ordering is maintained between stored procedures within a dataflow, and between dataflow instantiations. S-Store provides such a **streaming scheduler**.

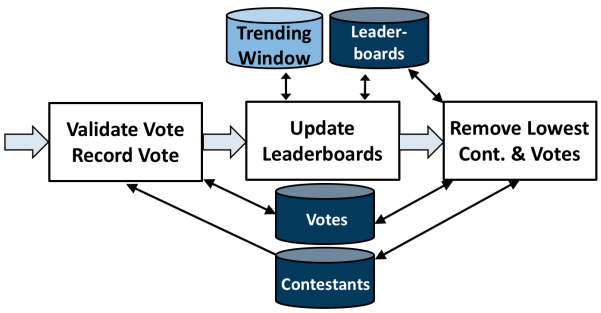
In single-node S-Store, transactions are scheduled serially, meaning that a batch will be processed to completion within a dataflow graph before the next batch is considered. This simple scheduling policy ensures that both stream and dataflow order constraints will always be satisfied for a given dataflow graph. In our ongoing work, we are extending the streaming scheduler to operate over multiple nodes.

5.3 Exactly-Once Implementation

Within single-node S-Store, our primary concern regarding exactly-once processing lies within internal message passing via streams, so we provide the guarantee primarily through fault tolerance. We provide two alternative fault-tolerance mechanisms, both of which guarantee exactly-once processing semantics.

In **strong recovery**, each transaction execution is logged using H-Store's command-logging mechanism. When recovering in this mode, the original execution order of the transactions will be replayed in exactly the same way as in the log. To ensure the exactly-once processing guarantee, PE triggers are turned off during recovery; all transactions are replayed from the log, but no transactions will be repeated.

In **weak recovery**, only "border" transactions (i.e., transactions that begin a dataflow graph) are command-logged. Upon recovery, these transactions are re-executed, but with PE triggers kept turned-on. The streaming scheduler will execute the full dataflow graph in a *legal* order according to ordering and data isolation rules, but not necessarily in the *exact* order that they were originally executed before the failure. This alternative recovery mode improves both run-time and recovery performance, while still providing the ordered execution (via the



(a) Leaderboard Maintenance Dataflow

System	ACID	Order	Exactly-Once	Max Tput (batches/sec)
H-Store (async)	✓	✗	✗	5300
H-Store (sync)	✓	✓	✗	210
Esper + VoltDB	✓	✓	✗	570
Storm + VoltDB	✓	✓	✓	600
S-Store	✓	✓	✓	2200

(b) Experiment Result

Figure 4: Performance vs. Correctness Guarantees

scheduler) and exactly-once processing guarantees.

For more information about the implementation of S-Store, please refer to our PVLDB paper [23].

6 State-of-the-Art Comparison

When evaluating S-Store's performance, it is once again important to consider the three guarantees described in Section 3. In modern state-of-the-art systems, it is challenging to provide all three processing guarantees. More specifically, OLTP systems are able to process ACID transactions with high performance, but have no concept of dataflow graphs, and thus no inherent support for ordering or exactly-once processing. In contrast, stream processing systems are able to provide dataflow ordering and exactly-once processing, but do not support ACID transactions. Thus, in both cases, achieving all three guarantees with high performance is a major challenge.

To test S-Store's performance in comparison to current state of the art, we created a simple leaderboard-maintenance benchmark. This benchmark mimics a singing competition in which users vote for their favorite contestants, and periodically, the lowest contestant is removed until a winner is selected. As shown in Figure 4a, the benchmark's dataflow graph is composed of three stored procedures that each access shared table state, and thus requires data isolation (i.e., a nested transaction) across all three. For the purposes of simplifying comparison across systems, we considered a batch to be a single vote, and we record our throughput numbers in terms of "input batches per second."

The leaderboard-maintenance benchmark requires all three of S-Store's processing guarantees to be executed correctly. We first compared S-Store's performance to its OLTP predecessor, H-Store. As an OLTP system, by default H-Store only provides the first guarantee, ACID, and thus maintains an impressive throughput (over 5000 input batches per second, as shown in the first row of Figure 4b). However, the results it provides are incorrect; a wrong candidate may win the contest since votes may be processed in a different order than the one that is required by the benchmark. For H-Store to provide correct results, the ordering guarantee must also be provided.

We can force H-Store to provide an ordering guarantee across the dataflow graph by insisting that H-Store process the whole dataflow graph serially. In this case, the client has to manage the order in which the transactions are executed, by waiting for a response from the engine before it can submit the next transaction request (i.e., submitting requests in a synchronous manner). As one would expect, performance suffers drastically as a result. H-Store's throughput plummets to around 200 input batches per second, when ordering constraints are enforced via synchronous requests.

Both single-node streaming engines (e.g., Esper [3]) and distributed stream processing engines (e.g., Storm [27]) also struggle to provide all three processing guarantees. In the case of streaming engines, dataflow graphs

are core functionality, and the ordering guarantee is provided. Exactly-once processing can also be added to many systems possibly with some loss in performance (e.g., Storm with Trident [4]). However, ACID transactions are not integrated into streaming systems. Instead, they must use an additional OLTP database to store and share the mutable state consistenly. For our experiments, we used VoltDB [5] (the commercial version of H-Store) to provide this functionality to Esper and Storm.

Similarly to H-Store, providing all three processing guarantees degrades throughput. To provide both ordering and ACID, the streaming systems must submit requests to the OLTP database and wait for the response to move on. Even with a main-memory OLTP system such as VoltDB, this additional communication takes time and prevents the stream system from performing meaningful work in the meantime. As shown in Figure 4b, both Esper and Storm with Trident were only able to manage about 600 input batches per second, when providing ACID guarantees through VoltDB.

By contrast, S-Store is able to maintain 2200 input batches per second on the same workload, while natively providing all three processing guarantees. S-Store manages both dataflow graph ordering and consistent mutable state in the same engine. This combination allows S-Store to handle multiple asynchronous transaction requests from the client and still preserve the right processing order within the partition engine. Meanwhile, each operation performed on any state is transactional, guaranteeing that the data is consistent every time it is accessed – even in presence of failures.

7 Related Work

First-generation streaming systems provided relational-style query processing models and system architectures for purely streaming workloads [3, 6, 9, 15]. The primary focus was on low-latency processing over push-based, unbounded, and ordered data arriving at high or unpredictable rates. State management mostly meant efficiently supporting joins and aggregates over sliding windows, and correctness was only a concern in failure scenarios [10, 19].

Botan et al. proposed extensions to the traditional database transaction model to enable support for continuous queries over both streaming and stored data sources [13]. While this work considered ACID-style access to shared data, its focus was limited to correctly ordering individual read and write operations for a single continuous query rather than transaction-level ordering for complex dataflow graphs as in S-Store.

More recently, a new breed of streaming systems has emerged, which commonly aim at providing a MapReduce-like distributed and fault-tolerant framework for real-time computations over streaming data. Examples include S4 [25], Storm [27], Twitter Heron [22], Spark Streaming [28], Samza [2], Naiad [24], Flink [1], and MillWheel [7]. These systems significantly differ in terms of the way they manage persistent state and the correctness guarantees that they provide, but none of them is capable of handling streaming applications with shared mutable state with sufficient consistency guarantees as provided by S-Store.

S4, *Storm*, and *Twitter Heron* neither support fault-tolerant persistent state nor can guarantee exactly once processing. Storm when used with *Trident* can ensure exactly-once semantics, yet with significant degradation in performance [4]. Likewise, *Google MillWheel* can persist state with the help of a backend data store (e.g., BigTable or Spanner), and can deal with out-of-order data with exactly once processing guarantees using a low-watermark mechanism [7].

Several recent systems adopt a *stateful dataflow model* with support for in-memory state management. *SEEP* decouples a streaming operators state from its processing logic, thereby making state directly manageable by the system via a well-defined set of primitive scale-out and fault-tolerance operations [17, 18]. *Naiad* extends the MapReduce model with support for structured cycles and streaming based on a timely dataflow model that uses logical timestamps for coordination [24]. *Samza* isolates multiple processors by localizing their state and disallowing them from sharing data, unless data is explicitly written to external storage [2]. Like S-Store, all of these systems treat state as mutable and explicitly manageable, but since they all focus on analytical and cyclic

dataflow graphs, they do not provide inherent support for transactional access to shared state.

There are a number of systems have explicitly been designed for handling *hybrid workloads* that include streaming. *Spark Streaming* extends the Spark batch processing engine with support for discretized streams (D-Streams) [28]. All state is stored in partitioned, immutable, in-memory data structures called Resilient Distributed Datasets (RDDs). Spark Streaming provides exactly-once consistency semantics, but is not a good fit for transactional workloads that require many fine-grained update operations. *Microsoft Trill* is another hybrid engine designed for a diverse spectrum of analytical queries with real-time to offline latency requirements [14]. Trill is based on a tempo-relational query model that incrementally processes events in batches organized as columns. Like Spark Streaming, its focus lies more on OLAP settings with read-mostly state. Last but not least, the *Google Dataflow Model* provides a single unified processing model for batch, micro-batch, and streaming workloads [8]. It generalizes the windowing, triggering, and ordering models found in MillWheel [7] in a way to enable programmers to make flexible tradeoffs between correctness and performance.

8 Conclusion

In this paper, we have described an approach to stream processing for applications that have shared, mutable state. These applications require guarantees for correct execution. We discussed ACID guarantees as in OLTP systems. We also described the idea of exactly-once processing, exactly-once delivery, and transactional workflows that obey ordering constrains as expressed in a dataflow graph. The paper also describes how we implement these guarantees on top of the H-Store OLTP main-memory database system.

In the future, we intend to look at extending our single-node prototype to run in a multi-node environment. This, of course, will preserve the guarantees mentioned above. We will re-examine recovery for our distributed extensions.

We are also studying how to adapt S-Store to effectively act as a real-time ETL system. Rather than loading data from flat files, S-Store will accept batches of tuples and install them transactionally in a persistent data store (either within S-Store or externally). During this process, its stored procedures can perform data cleaning and alerting. Each batch, possibly from multiple sources, must be processed to completion or not all. Furthermore, as tuples are being loaded, other transactions should not be allowed to see a partially loaded state. S-Store's ability to manage shared state makes it an ideal candidate for real-time ETL.

Acknowledgments. This research was funded in part by the Intel Science and Technology Center for Big Data, and by the NSF under grants NSF IIS-1111423 and NSF IIS-1110917.

References

[1] Apache Flink. `https://flink.apache.org/`.

[2] Apache Samza. `http://samza.apache.org/`.

[3] Esper. `http://www.espertech.com/esper/`.

[4] Trident Tutorial. `https://storm.apache.org/documentation/Trident-tutorial.html`.

[5] VoltDB. `http://www.voltdb.com/`.

[6] D. Abadi et al. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2), 2003.

[7] T. Akidau et al. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *PVLDB*, 6(11), 2013.

[8] T. Akidau et al. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *PVLDB*, 8(12), 2015.

[9] A. Arasu et al. STREAM: The Stanford Data Stream Management System. In *Data Stream Management: Processing High-Speed Data Streams*, 2004.

[10] M. Balazinska et al. Fault-tolerance in the Borealis Distributed Stream Processing System. *ACM TODS*, 33(1), 2008.

[11] I. Botan et al. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. *PVLDB*, 3(1), 2010.

[12] N. Jain et al. Towards a Streaming SQL Standard. *PVLDB*, 1(2), 2008.

[13] I. Botan et al. Transactional Stream Processing. In *EDBT*, 2012.

[14] B. Chandramouli et al. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. *PVLDB*, 8(4), 2014.

[15] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[16] A. Elmore et al. A Demonstration of the BigDAWG Polystore System. *PVLDB*, 8(12), 2015.

[17] R. C. Fernandez et al. Integrating Scale-out and Fault-tolerance in Stream Processing using Operator State Management. In *SIGMOD*, 2013.

[18] R. C. Fernandez et al. Making State Explicit for Imperative Big Data Processing. In *USENIX ATC*, 2014.

[19] J.-H. Hwang et al. High-Availability Algorithms for Distributed Stream Processing. In *ICDE*, 2005.

[20] Institute of Medicine of the National Academies. Preventing Medication Errors . https://iom.nationalacademies.org/̃media/Files/Report%20Files/2006/Preventing-Medication-Errors-Quality-Chasm-Series/medicationerrorsnew.pdf.

[21] R. Kallman et al. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *PVLDB*, 1(2), 2008.

[22] S. Kulkarni et al. Twitter Heron: Stream Processing at Scale. In *SIGMOD*, 2015.

[23] J. Meehan et al. S-Store: Streaming Meets Transaction Processing. *PVLDB*, 8(13), 2015.

[24] D. G. Murray et al. Naiad: A Timely Dataflow System. In *SOSP*, 2013.

[25] L. Neumeyer et al. S4: Distributed Stream Computing Platform. In *KDCloud*, 2010.

[26] PhysioNet. MIMIC II Data Set. `https://physionet.org/mimic2/`.

[27] A. Toshniwal et al. Storm @Twitter. In *SIGMOD*, 2014.

[28] M. Zaharia et al. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP*, 2013.