S-Store: Streaming Meets Transaction Processing

John Meehan[1], Nesime Tatbul[2,3], Stan Zdonik[1], Cansu Aslantas[1],
Ugur Cetintemel[1], Jiang Du[4], Tim Kraska[1], Samuel Madden[3], David Maier[5],
Andrew Pavlo[6], Michael Stonebraker[3], Kristin Tufte[5], Hao Wang[3]

[1]Brown University [2]Intel Labs [3]MIT [4]University of Toronto [5]Portland State University [6]CMU

ABSTRACT

Stream processing addresses the needs of real-time applications. Transaction processing addresses the coordination and safety of short atomic computations. Heretofore, these two modes of operation existed in separate, stove-piped systems. In this work, we attempt to fuse the two computational paradigms in a single system called S-Store. In this way, S-Store can simultaneously accommodate OLTP and streaming applications. We present a simple transaction model for streams that integrates seamlessly with a traditional OLTP system, and provides both ACID and stream-oriented guarantees. We chose to build S-Store as an extension of H-Store - an open-source, in-memory, distributed OLTP database system. By implementing S-Store in this way, we can make use of the transaction processing facilities that H-Store already provides, and we can concentrate on the additional features that are needed to support streaming. Similar implementations could be done using other main-memory OLTP platforms. We show that we can actually achieve higher throughput for streaming workloads in S-Store than an equivalent deployment in H-Store alone. We also show how this can be achieved within H-Store with the addition of a modest amount of new functionality. Furthermore, we compare S-Store to two state-of-the-art streaming systems, Esper and Apache Storm, and show how S-Store can sometimes exceed their performance while at the same time providing stronger correctness guarantees.

1. INTRODUCTION

A decade ago, the database research community focused attention on stream data processing systems. These systems [10, 16], including our own system, Aurora/Borealis [7, 8], were largely concerned with executing SQL-like operators on an unbounded and continuous stream of input data. The main optimization goal of these systems was reducing the latency of results, since they mainly addressed what might be called monitoring applications [28, 30]. To achieve this, they were typically run in main memory, thereby avoiding the extreme latency caused by disk access.

While essentially all of the monitoring applications that we encountered had a need for archival storage, the system-level support for this was limited and ad hoc. That is, the systems were largely not designed with storage in mind; it was tacked on after the fact. Thus, there was no support for things like ACID transactions, leaving applications open to potential inconsistencies with weak guarantees for isolation and recovery.

These first-generation streaming systems could be viewed as real-time analytics systems. After all, the input was made up of an infinite stream of new tuples. The notion of some of these tuples representing updates of previously viewed tuples (or causing updates to other stored data that is related) was not made explicit in the model. This is fine if time is the key. In this case, if each tuple is given a unique timestamp, the update pattern is append-only. However, there are cases when the identifying attribute is something else. Consider a stock ticker application in which stock symbol is the key. Here a new tuple for, say, IBM is really an update to the previously reported price. Traders want to see the current stock book as a consistent view of the 6000 stocks on the NYSE, with all prices reported in a consistent way. Thus, these applications introduce the need for *shared mutable state* in streaming systems.

We are beginning to see the rise of second-generation streaming systems [1, 2, 9, 32, 33, 37, 40]. These systems do not enforce a relational view on their users. Instead, they allow users to create their own operators that are invoked and managed by a common infrastructure. Note that it is reasonable to have libraries of common operators (including relational) that manipulate tables. The infrastructure enforces some model of failure semantics (e.g., at-least-once or exactly-once processing), but still ignores needs of proper isolation and consistent storage in the context of updates.

Meanwhile, the advent of inexpensive, high-density RAM has led to a new generation of distributed on-line transaction processing (OLTP) systems that store their data in main memory, thereby enabling very high throughput with ACID guarantees for workloads with shared mutable state (e.g., [6, 18, 29]). However, these systems lack the notion of stream-based processing (e.g., unbounded data, push-based data arrival, ordered processing, windowing).

Many applications that involve shared mutable state in fact need aspects of both streaming and transaction processing. In this paper, we propose to combine these two computational paradigms in a single system called S-Store.

1.1 Example Use Cases

Applications that benefit from this kind of hybrid system generally include those that use the streaming facilities to record persistent state or views in shared tables (in near real-time), and at the same time use the transactional facilities to ensure a consistent representation or summary of this state (e.g., dashboards or leaderboards [14]). We now describe two selected use cases as examples.

Real-Time Data Ingestion. An analytics warehouse must be updated periodically with recent activity. It was once the case that this was done once a day (typically at night) when there was little

Proceedings of the VLDB Endowment, Vol. 8, No. 13

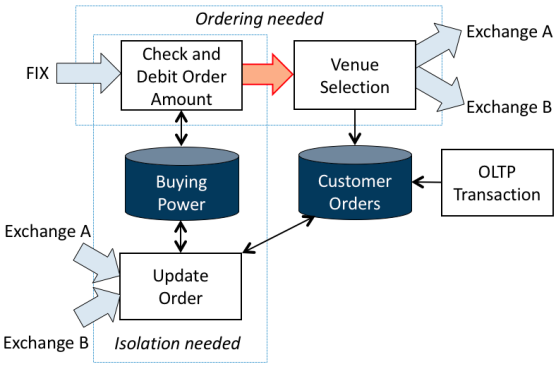


Figure 1: FIX Trading Example

to no load on the system. Nowadays, systems must be available at all times and the latency window for loading new data is quickly shrinking. Also, new data must be added to the warehouse in a consistent fashion (e.g., groups of updates must be added atomically) [23]. This suggests that a transaction mechanism is needed. Even more interesting is the fact that incoming data is typically in different formats and is often dirty. ETL tools can address some of the problems of data cleaning and integration, but they work with files of bulk updates. This is slow and cumbersome, and cannot load the warehouse in near real time. Thus, there is a need for something similar to ETL that instead works on streaming data. S-Store is well-positioned to satisfy this need, and in fact is already being used for this purpose in the BigDAWG system [19].

Shared Mutable State. S-Store is useful beyond real-time ETL, as illustrated in the example depicted in Figure 1. In the figure, rectangles represent transactions; oil drums represent stored, shared data; skinny arrows represent reads and writes of stored data; and block arrows represent streams. This example is based on customer experience at TIBCO StreamBase, Inc [4]. It is a simplified version of intelligent order routing with FIX (Financial Information eXchange) data.

Notice that FIX data arrives on a stream and is processed by a transaction (*Check and Debit Order Amount*) that checks the buyer's account balance and puts a temporary hold on the funds involved in that transaction in the *Buying Power* database. When this is successful, the *Venue Selection* transaction determines to which exchange the order is to be sent. This can be a complex process that involves checking, e.g., the history of a particular exchange with the given security, and may involve retrieving data from other databases not shown in the figure. Thus, it is modeled as a separate transaction so that the *Buying Power* database is available to other transactions, before the *Venue Selection* transaction is complete.

Also, *Venue Selection* requires isolation, since it has to make its decision based on a consistent state (e.g., there may be other, independent OLTP transactions accessing the *Customer Orders* database as shown in the figure). The bold red arrow that connects these two transactions expresses a dependency between them which requires that for a particular FIX input, *Check and Debit Order Amount* must precede *Venue Selection*. This illustrates the need for transaction ordering. Moreover, when *Check and Debit Order Amount* commits, *Venue Selection* needs to be triggered (push-based processing). At the bottom of the figure, the *Update Order* transaction takes input from the exchanges, and confirms or denies previously placed orders. In the case of a failed order, it will return funds to the customers account. This can obviously conflict with new orders from the same customer. Thus, *Check and Debit Order Amount* and *Update Order* must both be transactions to guarantee consistency through isolation.

If we used only a pure stream processing system to implement this use case, we would be able to ensure ordering and push-based processing. However, the isolation requirements of the application would not be expressible. If we used a pure OLTP DBMS instead, we would be able to ensure isolation, but would be unable to take advantage of push-based processing. Transaction ordering would need to be managed at the client, requiring unnecessary context switches and a need to poll the interface for new data. Today, use cases like this are implemented with in-memory data structures, careful custom coding, and recovery based on replaying message logs. We believe that a platform like S-Store reduces user code complexity.

1.2 Contributions and Outline

This paper introduces the design and implementation of S-Store, a single system for processing streams and transactions with well-defined correctness guarantees. Our approach to building such a system is to start with a fully transactional OLTP main-memory database system and to integrate additional streaming functionality. By doing so, we are able to leverage infrastructure that already addresses many of the implementation complexities of transaction processing. This choice is very natural, since streaming systems largely run in main memory to achieve low latency. More specifically, this work makes the following key contributions:

Model. We define a novel, general-purpose computational model that allows us to seamlessly mix *streaming transactions* with ordinary OLTP transactions. Stream processing adds additional semantics to an OLTP engine's operating model. In particular, stream processing introduces the notion of *order* to the transaction mix. That is, it is possible to say that one transaction must precede another, something that is missing from the non-deterministic semantics of a standard transaction model. Further, since streams are unbounded and arrive on a continuous basis, there is a need to add the necessary primitives for bounding computation on streams, such as batch-based processing [10, 27] and windowing [12, 24]. Finally, streaming transactions support a push-based processing model, whereas OLTP transactions access state in a pull-based manner. Our hybrid model provides uniform access to state for all transactions.

Architecture and Implementation. We show how our hybrid computational model can be cleanly and efficiently implemented on top of a state-of-the-art main-memory OLTP engine (H-Store [29]). Our architectural extensions are general enough to be applied to any main-memory OLTP engine, and include: (i) streams and windows represented as time-varying state, (ii) triggers to enable push-based processing over such state, (iii) a streaming scheduler that ensures correct transaction ordering, and (iv) a variant on H-Store's recovery scheme that ensures exactly-once processing for streams. Note that the discussion in this paper is confined to the single-node case; multi-node S-Store is the topic for follow-on research.

Performance. We provide a detailed study of S-Store's performance characteristics, specifically the benefits of integrating transactional state processing with push-based processing. For streaming workloads that require transactional state, S-Store demonstrates improved throughput over both pure OLTP systems and pure streaming systems. In both cases, the advantage is a direct result of integrating state and processing, removing blocking during communication between the dataflow manager and the data-storage engine.

The rest of this paper is organized as follows: We first describe our computational model for transactional stream processing in Section 2. Section 3 presents the design and implementation of the S-Store system, which realizes this model on top of the H-Store main-memory OLTP system [29]. In Section 4, we present an experimental evaluation of S-Store in comparison to H-Store, as well

as to two representative stream processing systems - Esper [3] (first generation) and Storm [37] (second generation). We discuss related work in Section 5, and finally conclude the paper with a summary and a discussion of future research directions in Section 6.

2. THE COMPUTATIONAL MODEL

In this section, we describe our computational model for transactional stream processing. This model allows us to support hybrid workloads (i.e., independent OLTP transactions and streaming transactions) with well-defined correctness guarantees. As we will discuss in more detail shortly, these guarantees include:

1. ***ACID*** guarantees for individual transactions (both OLTP and streaming)
2. ***Ordered Execution*** guarantees for dataflow graphs of streaming transactions
3. ***Exactly-Once Processing*** guarantees for streams (i.e., no loss or duplication)

S-Store acquires ACID guarantees from the traditional OLTP model (Sections 2.1 and 2.2), and adds ordered execution guarantees to capture stream-based processing semantics (Sections 2.3 and 2.4) and exactly-once processing guarantees for correctly recovering from failures (Section 2.5).

2.1 Overview

Our model adopts well-accepted notions of OLTP and stream processing, and fuses them into one coherent model. We assume that the reader is already familiar with the traditional notions, and strive to keep our model description brief and informal for them.

We assume that both OLTP and streaming transactions can share state and at the same time produce correct results. S-Store supports three different kinds of state: (i) public tables, (ii) windows, and (iii) streams. Furthermore, we make a distinction between *OLTP transactions* that only access public tables, and *streaming transactions* that can access all three kinds of state.

For OLTP transactions, we simply adopt the traditional ACID model that has been well-described in previous literature [39]. A *database* consists of unordered, bounded collections (i.e., sets) of tuples. A *transaction* represents a finite unit of work (i.e., a finite sequence of read and write operations) performed over a given database. In order to maintain integrity of the database in the face of concurrent transaction executions and failures, each transaction is executed with *ACID guarantees*.

Each transaction (OLTP or streaming) has a definition and possibly many executions (i.e., instances). We assume that all transactions are predefined as *stored procedures with input parameters*. They are predefined, because: (i) OLTP applications generally use a relatively small collection of transactions many times (e.g., Account Withdrawal), (ii) streaming systems typically require predefined computations. Recall that it is the data that is sent to the query in streaming systems in contrast to the standard DBMS model of sending the query to the data. The input parameters for OLTP transactions are assigned by the application when it explicitly invokes them ("pull"), whereas streaming transactions are invoked as new data becomes available on their input streams ("push").

For the purpose of granularity, the programmer determines the transaction boundaries. Course-grain transactions protect state for a longer period, but in so doing, other transactions may have to wait. Fine-grained transactions are in general preferred when they are safe. Fine-grained transactions make results available to other transactions earlier. Said another way, in dataflow graphs with transactions, we can commit stable results when they are ready and then continue processing as required by the dataflow graph.

2.2 Streaming Transactions & Dataflow Graphs

Data Model. Our stream data model is very similar to many of the stream processing systems of a decade ago [7, 10, 16]. A *stream* is an ordered, unbounded collection of tuples. Tuples have a timestamp [10] or, more generally, a batch-id [12, 27] that specifies simultaneity and ordering. Tuples with the same batch-id b logically occur as a group at the same time and, thus, should be processed as a unit. Any output tuples produced as a result of this processing are also assigned the same batch-id b (yet they belong to a different stream). Furthermore, to respect the inherent stream order, batches of tuples on a given stream should be processed in increasing order of their batch-id's. This batch-based model is very much like the approaches taken by STREAM (group tuples by individual timestamps) [10], or more recently, by Spark Streaming (group tuples into "mini batches" of small time intervals) [40].

In our model, the above-described notion of a "batch" of tuples in a stream forms an important basis for transaction atomicity. A streaming transaction essentially operates over non-overlapping "atomic batches" of tuples from its input streams. Thus, an *atomic batch* corresponds to a finite, contiguous subsequence of a stream that must be processed as an indivisible unit. Atomic batches for input streams must be defined by the application programmer, and can be based on timestamps (like in [10, 40]) or tuple counts.

Processing Model. Stream processing systems commonly define computations over streams as dataflow graphs. Early streaming systems focused on relational-style operators as computations (e.g., Filter, Join), whereas current systems support more general user-defined computations [1, 2, 9, 32, 33, 37, 40]. Following this trend and consistent with our OLTP model, we assume that computations over streams are expressed as dataflow graphs of user-defined stored procedures. More formally, a *dataflow graph* is a directed acyclic graph (DAG), in which nodes represent streaming transactions (defined as stored procedures) or nested transactions (described in Section 2.4), and edges represent an execution ordering. If there is an edge between node T_i and node T_j, there is also a stream that is output for T_i and input for T_j. We say that T_i precedes T_j and is denoted as $T_i \prec T_j$.

Furthermore, given the unbounded nature of streams, all stream processing systems support windowing as a means to restrict state and computation for stateful operations (e.g., Join, Aggregate). A *window* is a finite, contiguous subsequence of a stream. Windows can be defined in many different ways [12, 24], but for the purposes of this work, we will restrict our focus to the most common type: sliding windows. A *sliding window* is a window which has a fixed size and a fixed slide, where the slide specifies the distance between two consecutive windows and must be less than or equal to the window size (if equal to window size, it has been called a *tumbling window*). A sliding window is said to be *time-based* if its size and slide are defined in terms of tuple timestamps, and *tuple-based* if its size and slide are defined in terms of the number of tuples. Note that atomic batches and tumbling windows are similar in definition, but their use is orthogonal: batches are external to a streaming transaction T and are mainly used to set atomic boundaries for T's instances, whereas windows are internal to T and are used to bound computations defined inside T.

Atomic batches of tuples arrive on a stream at the input to a dataflow graph from push-based data sources. We adopt the data-driven execution model of streams, where arrival of a new atomic batch causes a new invocation for all the streaming transactions that are defined over the corresponding stream. We refer to execution of each such transaction invocation as a *transaction execution* (TE). (In the rest of this paper, we use the terms "transaction" and "stored procedure" interchangeably to refer to the definition of a

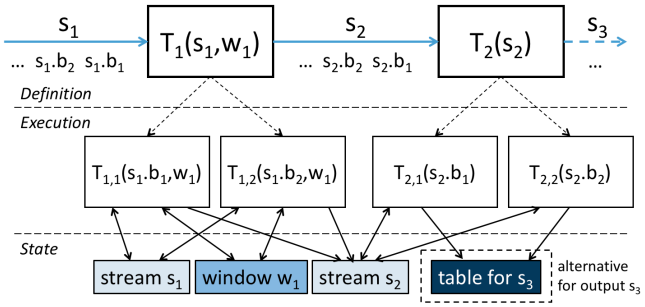


Figure 2: Transaction Executions in a Dataflow Graph

transaction, whereas we use the term "transaction execution" (TE) to refer to a specific invocation of that definition). A TE essentially corresponds to an atomic batch and its subsequent processing by a stored procedure. For example, in Figure 2, a dataflow graph with two stored procedures (i.e., T_1 and T_2) are defined above the dashed line, labeled "*Definition*", but each of those are executed twice for two contiguous atomic batches on their respective input streams (i.e., $s_1.b_1$, $s_1.b_2$ for T_1, and $s_2.b_1$, $s_2.b_2$ for T_2), yielding a total of four TE's shown below the dashed line, labeled "*Execution*" (i.e., $T_{1,1}$, $T_{1,2}$, $T_{2,1}$, and $T_{2,2}$). Note, $s_1.b_2$ denotes the second batch on stream $s1$ and $T_{1,2}$ denotes the second execution of T_1 on that batch.

Given a dataflow graph, it is also useful to distinguish between *border transactions* (those that ingest streams from the outside, e.g., T_1 in Figure 2) and *interior transactions* (others, e.g., T_2 in Figure 2). Border transactions are instantiated by each newly arriving atomic batch (e.g., $s_1.b_1$, $s_1.b_2$), and each such execution may produce a group of output stream tuples labeled with the same batch-id as the input that produced them (e.g., $s_2.b_1$, $s_2.b_2$, respectively). These output tuples become the atomic batch for the immediately downstream interior transactions, and so on.

Figure 2 also illustrates the different kinds of state accessed and shared by different transaction instances (shown below the dashed line, labeled "*State*"). T_1 takes as input the stream s_1 and the window w_1, and produces as output the stream s_2, whereas T_2 takes as input the stream s_2 and produces as output the stream s_3. Thus, TE's of T_1 (i.e., $T_{1,1}$ and $T_{1,2}$) share access to s_1, w_1, and s_2, whereas TE's of T_2 (i.e., $T_{2,1}$ and $T_{2,2}$) do so for s_2 and s_3. Note, there are two ways to output final results of a dataflow graph (e.g., s_3 in Figure 2): (i) write them to a public table, or (ii) push them to a sink outside the system (e.g., a TCP connection).

In order to ensure a correct execution, shared state accesses must be properly coordinated. We discuss this issue in more detail next.

2.3 Correct Execution for Dataflow Graphs

A standard OLTP transaction mechanism guarantees the isolation of a transaction's operations from others'. When a transaction T commits successfully, all of T's writes are installed and made public. During T's execution, all of T's writes remain private.

S-Store adopts such standard transaction semantics as a basic building block for its streaming transactions (thus ensuring ACID guarantees in this way); however, the ordering of stored procedures in the dataflow graph as well as the inherent order in streaming data puts additional constraints on allowable transaction execution orders. As an example, consider again the dataflow graph shown in Figure 2. The four TE's illustrated in this example can be ordered in one of two possible ways: $[T_{1,1}, T_{2,1}, T_{1,2}, T_{2,2}]$ or $[T_{1,1}, T_{1,2}, T_{2,1}, T_{2,2}]$. Any other orderings would not lead to a correct execution. This is due to the precedence relation between T_1 and T_2 in the graph as well as the ordering of the atomic batches on their input streams. This requirement is in contrast to most OLTP transaction processors which would accept any serializable schedule (e.g., one that is equivalent to any of the 4! possible serial execution schedules if these were 4 independent transactions).

Note that we make no ACID claims for the dataflow graph as a whole. The result of running a dataflow graph is to create an ordered execution of ACID transactions.

Furthermore, in streaming applications, the state of a window must be shared differently than other stored state. To understand this, consider again the simple dataflow graph shown in Figure 2. Let us assume for simplicity that the transaction input batch size for T_1 is 1 tuple. Further, suppose that T_1 constructs a window of size 2 that slides by 1 tuple, i.e., two consecutive windows in T_1 overlap by 1 tuple. This means that window state will carry over from $T_{1,1}$ to $T_{1,2}$. For correct behavior, this window state must not be publicly shared with other transaction executions. That is, the state of a window can be shared among consecutive executions of a given transaction, but should not be made public beyond that. Returning to Figure 2, when $T_{1,1}$ commits, the window in $T_{1,1}$ will slide by one and will then be available to $T_{1,2}$, but not to $T_{2,1}$. This approach to window visibility is necessary, since it is this way of sharing window state that is the basis for continuous operation. Windows evolve and, in some sense, "belong" to a particular stored procedure. Thus, a window's visibility should be restricted to the transaction executions of its "owning" stored procedure.

We will now describe what constitutes a *correct execution* for a dataflow graph of streaming transactions more formally. Consider a dataflow graph D of n streaming transactions T_i, $1 \leq i \leq n$. D is a directed acyclic graph $G = (V, E)$, where $V = \{T_1, \ldots, T_n\}$ and $E \subseteq V \times V$, where $(T_i, T_j) \in E$ means that T_i must precede T_j (denoted as $T_i \prec T_j$). Being a DAG, G has at least one topological ordering. A topological ordering of G is an ordering of its nodes $T_i \in V$ such that for every edge $(T_i, T_j) \in E$ we have $i < j$. Each topological ordering of G is essentially some permutation of V.

Without loss of generality: (i) Let us focus on one specific topological ordering of G and call it O; (ii) For ease of notation, let us simply assume that O corresponds to the identity permutation such that it represents: $T_1 \prec T_2 \prec .. \prec T_n$.

T_i represents a transaction definition $T_i(s_i, w_i, p_i)$, where s_i denotes all stream inputs of T_i (at least one), w_i denotes all window inputs of T_i (optional), p_i denotes all table partition inputs of T_i (optional). Similarly, $T_{i,j}$ represents the j^{th} transaction execution of T_i as $T_{i,j}(s_i.b_j, w_i, p_i)$, where $s_i.b_j$ denotes the j^{th} atomic batches of all streams in s_i.

A dataflow graph D is executed in rounds of atomic batches $1 \leq r < \infty$, such that for any round r, atomic batch r from all streaming inputs into D generates a sequence of transaction executions $T_{i,r}(s_i.b_r, w_i, p_i)$ for each T_i. Note that this execution generates an *unbounded schedule*. However, as of a specific round $r = R$, we generate a *bounded schedule* that consists of all $R * n$ transaction executions: $1 \leq r \leq R, 1 \leq i \leq n, T_{i,r}(s_i.b_r, w_i, p_i)$.

In the traditional ACID model of databases, any permutation of these $R * n$ transaction executions would be considered to be a valid/correct, serial schedule. In our model, we additionally have:

1. *Dataflow graph order constraint:* Consider the topological ordering O of G as we defined above. Then for any given execution round r, it must hold that:
   $T_{1,r}(s_1.b_r, w_1, p_1) \prec \ldots \prec T_{n,r}(s_n.b_r, w_n, p_n)$
2. *Stream order constraint:* For any given transaction T_i, as of any execution round r, the following must hold:
   $T_{i,1}(s_i.b_1, w_i, p_i) \prec \ldots \prec T_{i,r}(s_i.b_r, w_i, p_i)$

(1) follows from the definition of a dataflow graph which specifies a precedence relation on its nodes, whereas (2) is to ensure that atomic batches of a given stream are processed in order.

Any bounded schedule of D that meets the above two ordering constraints is a *correct schedule*. If G has multiple topological orderings, then the dataflow graph order constraint must be relaxed to accept any of those orderings for any given execution round of D.

2.4 Correct Execution for Hybrid Workloads

S-Store's computational model allows OLTP and streaming transactions to co-exist as part of a common transaction execution schedule. This is particularly interesting if those transactions access shared public tables. Given our formal description of a correct schedule for a dataflow graph D that consists of streaming transactions, any OLTP transaction execution $T_{i,j}(p_i)$ (defined on one or more public table partitions p_i) is allowed to interleave *anywhere* in such a schedule. The resulting schedule would still be correct.

We have also extended our transaction model to include *nested transactions*. Fundamentally, this allows the application programmer to build higher-level transactions out of smaller ones, giving her the ability to create coarser isolation units among stored procedures, as illustrated in Figure 3. In this example, two streaming transactions, T_1 and T_2, in a dataflow graph access a shared table partition p. T_1 writes to the table and T_2 reads from it. If another OLTP transaction also writes to p in a way to interleave between T_1 and T_2, then T_2 may get unexpected results. Creating a nested transaction with T_1 and T_2 as its children will isolate the behavior of T_1 and T_2 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. We describe such a scenario in Section 4.1.1.

More generally, an S-Store nested transaction consists of two or more stored procedures with a partial order defined among them [36]. The stored procedures within a nested transaction must execute in a way that is consistent with that partial order. A nested transaction will commit, if and only if all of its stored procedures commit. If one or more stored procedures abort, the whole nested transaction will abort.

Nested transactions fit into our formal model of streaming transactions in a rather straight-forward way. More specifically, any streaming transaction T_i in dataflow graph D can be defined as a nested transaction that consists of children $T_{i1}, \ldots, T_{im}$. In this case, $T_{i1}, \ldots, T_{im}$ must obey the partial order defined for T_i for every execution round r, $1 \leq r < \infty$. This means that no other streaming or OLTP transaction instance will be allowed to interleave with $T_{i1}, \ldots, T_{im}$ for any given execution round.

2.5 Fault Tolerance

Like any ACID-compliant database, in the face of failure, S-Store must recover all of its state (including streams, windows, and public tables) such that any committed transactions (including OLTP and streaming) remain stable, and, at the same time, any uncommitted transactions are not allowed to have any effect on this state. A TE that had started but had not yet committed should be undone, and it should be reinvoked with the proper input parameters once the system is stable again. For a streaming TE, the invocation should also take proper stream input from its predecessor.

In addition to ACID, S-Store strives to provide exactly-once processing guarantees for all streams in its database. This means that each atomic batch $s.b_j$ on a given stream s that is an input to a streaming transaction T_i is processed exactly once by T_i. Note that such a TE $T_{i,j}$, once it commits, will likely modify the database state (streams, windows, or public tables). Thus, even if a failure happens and some TE's are undone / redone during recovery, the database state must be "equivalent" to one that is as if s were processed exactly once by T_i.

For example, consider the streaming transaction $T_1(s_1, w_1)$ in Figure 2. If a failure happens while TE $T_{1,1}(s_1.b_1, w_1)$ is still executing, then: (i) $T_{1,1}$ should be undone, i.e., any modifications that it may have done on s_1, w_1, and s_2 should be undone; (ii) $T_{1,1}$ should be reinvoked for the atomic batch $s_1.b_1$. Similarly, if a failure happens after TE $T_{1,1}(s_1.b_1, w_1)$ has already committed, then all of its modifications on s_1, w_1, and s_2 should be retained in the database. In both of these failure scenarios, the recovery mechanism should guarantee that $s_1.b_1$ is processed exactly once by T_1 and the database state will reflect the effects of this execution.

Note that a streaming TE 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, as shown for s_3 at the top part of Figure 2). Such a side effect may get executed multiple times due to failures. Thus, our exactly-once processing guarantee applies only to state that is internal to S-Store (e.g., if s_3 were alternatively stored in an S-Store table as shown at the bottom part of Figure 2). This is similar to other exactly-once processing systems such as Spark Streaming [40].

If the dataflow graph definition allows multiple TE orderings or if the transactions within a dataflow graph contain any non-deterministic operations (e.g., use of a random number generator), we provide an additional recovery option that we call *weak recovery*. Weak recovery will produce a correct result in the sense that it will produce results that could have been produced if the failure had not occurred, but not necessarily the one that was in fact being produced. In other words, each atomic batch of each stream in the database will still be processed exactly once and the TE's will be ordered correctly (as described in Sections 2.3 and 2.4), but the final database state might look different than that of the original execution before the failure. This is because the new execution might follow a different (but valid) TE ordering, or a non-deterministic TE might behave differently every time it is invoked (even with the same input parameters and database state).

3. ARCHITECTURE & IMPLEMENTATION

We chose to build S-Store on top of the H-Store main-memory OLTP system [29]. This allows us to inherit H-Store's support for high-throughput transaction processing, thereby eliminating the need to replicate this complex functionality. We also receive associated functionality that will be important for streaming OLTP applications, including indexing, main-memory operation, and support for user-defined transactions.

In this section, we briefly describe the H-Store architecture and the changes required to incorporate S-Store's hybrid model described in the previous section. Nevertheless, we believe that the architectural features that we have added to H-Store are conceptually applicable to any main-memory OLTP system.

3.1 H-Store Overview

H-Store is an open-source, main-memory OLTP engine that was developed at Brown and MIT [29], and formed the basis for the design of the VoltDB NewSQL database system [6].

All transactions in H-Store must be predefined as stored procedures with input parameters. The stored procedure code is a mixture of SQL and Java. Transaction executions (TEs) are instantiated by binding input parameters of a stored procedure to real values and running it. In general, a given stored procedure definition will, over time, generate many TEs. TEs are submitted to H-Store, and

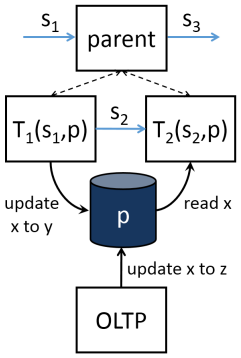


Figure 3: Nested Transactions

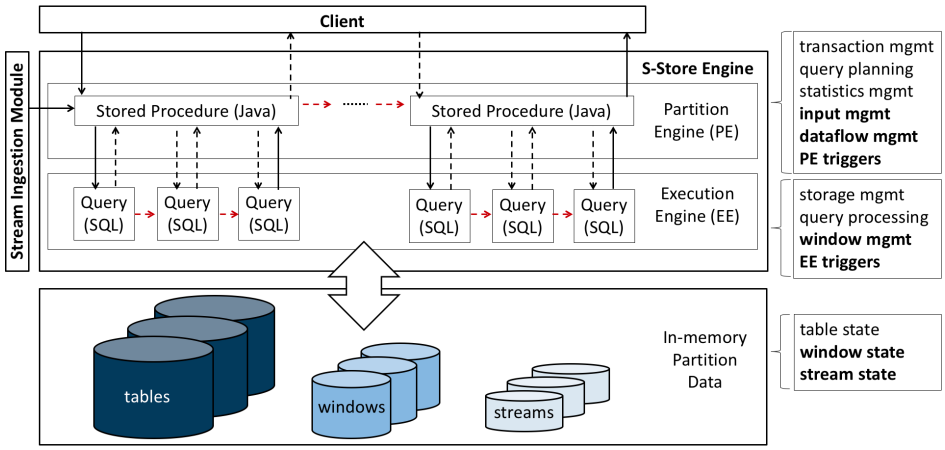


Figure 4: S-Store Architecture

the H-Store scheduler executes them in whatever order is required to provide ACID guarantees.

H-Store follows a typical distributed DBMS architecture in which a client initiates the transaction in a layer (in H-Store, called *the partition engine (PE)*) that is responsible for managing transaction distribution, scheduling, coordination, and recovery. The PE manages the use of another layer (in H-Store, called *the execution engine (EE)*) that is responsible for the local execution of SQL queries. This layering is very much like the transaction manager / transaction coordinator division of labor in a standard distributed DBMS architecture.

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 these sub-requests.

An H-Store database is partitioned across multiple sites [34], where a site corresponds to a CPU core. The available DRAM for a node is divided equally among the partitions, and each stores a horizontal slice of the database. A transaction is executed on the sites that hold the data that it needs. If the data is partitioned carefully, most transactions will only need data from a single site. Single-sited transactions are run serially on that site, thereby eliminating the need for fine-grained locks and latches.

H-Store provides recovery through a checkpointing and command-logging mechanism [31]. Periodically, the system creates a persistent snapshot or checkpoint of the current committed state of the database. Furthermore, every time H-Store commits a transaction, it writes a command-log record containing the name of that stored procedure along with its input parameters. This command-log record must be made persistent before its transaction can commit. In order to minimize interactions with the slow persistent store, H-Store offers a group-commit mechanism.

On recovery, the system's state is restored to the latest snapshot, and the command-log is replayed. That is, each command-log record causes the system to re-execute the same stored procedures with the same arguments in the same order that it did before the failure. Note that an undo-log is unnecessary, as neither the previous checkpoint nor the command-log will contain uncommitted changes.

3.2 S-Store Extensions

The high-level architecture of S-Store, directly adapted from H-Store, is shown in Figure 4. S-Store makes a number of extensions to H-Store to enable stream processing in the engine (shown in boldface in Figure 4). These include management of: (i) inputs from streaming clients and dataflow graphs of stored procedures at the PE layer, (ii) triggers at both the PE and the EE layers, (iii) stream- and window-based queries at the EE layer, (iv) in-memory stream and window state.

3.2.1 Streams

S-Store implements a stream as a time-varying H-Store table. Using this approach, stream state is persistent and recoverable. Since tables are unordered, the order of tuples in a stream is captured by timestamps. An atomic batch of tuples is appended to the stream table as it is placed on the corresponding stream, and conversely, an atomic batch of tuples is removed from the stream table as it is consumed by a downstream transaction in the dataflow. The presence of an atomic batch of tuples within a stream can activate either a SQL plan fragment or a downstream streaming transaction, depending on what "triggers" are attached to the stream (described in Section 3.2.2). In case of the latter, the current stream table serves as input for the corresponding downstream streaming transaction.

3.2.2 Triggers

Triggers enable push-based, data-driven processing needed to implement S-Store dataflow graphs. A trigger is associated with a stream table or a window table. When new tuples are appended to such a table, downstream processing will be automatically activated. The alternative to triggers would be polling for newly-arriving tuples, which would reduce throughput.

There are two types of triggers in S-Store to reflect the two-layer design of H-Store and of many other distributed database systems:

Partition engine (PE) triggers can only be attached to stream tables, and are used to activate downstream stored procedures upon the insertion and commit of a new atomic batch of tuples on the corresponding streams. As the name implies, PE triggers exist to create a push-based dataflow within the PE by eliminating the need to return back to the client to activate downstream stored procedures. In Figure 4, the horizontal arrows between stored procedures inside the PE layer denote PE triggers.

Execution Engine (EE) triggers can be attached to stream or window tables, and are used to activate SQL queries within the EE. These triggers occur immediately upon the insertion of an atomic batch of tuples in the case of a stream, and upon the insertion of an atomic batch of tuples that also cause a window to slide in the case of a window. The SQL queries are executed within the same transaction instance as the batch insertion which triggered them, and

can also activate further downstream EE triggers. EE triggers are designed to eliminate unnecessary communication between the EE and PE layers, for example when the execution of downstream processing is conditional. In Figure 4, the horizontal arrows between SQL queries inside the EE layer denote EE triggers.

3.2.3 Windows

Windows are also implemented as time-varying H-Store tables. A window is processed only when a new complete window state is available. For a sliding window, a new full window becomes available every time that window has one slide-worth of new tuples. Therefore, when new tuples are inserted into a window, they are flagged as "staged" until slide conditions are met. Staged tuples are not visible to any queries on the window, but are maintained within the window. Upon sliding, the oldest tuples within the window are removed, and the staged tuples are marked as active in their place. All window manipulation is done at the EE level, and output can be activated using an EE trigger.

Due to the invisible "staging" state of a window table as well as the transaction isolation rules discussed earlier in Section 2.3, special scoping rules are enforced for window state. A window table must not be accessed in general by TE's other than those of the stored procedure that defined it. In fact, a window table must only be visible to consecutive TE's of the stored procedure that contains it. As a consequence, one is not allowed to define PE triggers on window state, but only EE triggers. In other words, windows must be contained within the TE's of single stored procedures and must not be shared across other stored procedures in the dataflow graph.

S-Store provides automatic garbage collection mechanisms for tuples that expire from stream or window state, after any triggers associated with them have all been fired and executed.

It should be noted that some optimizations, such as incremental window processing, have been left as future work.

3.2.4 Streaming Scheduler

Being an OLTP database that implements the traditional ACID model, the H-Store scheduler can execute transaction requests in any order. On a single H-Store partition, transactions run in a serial fashion by design [29]. H-Store serves transaction requests from its clients in a FIFO manner by default.

As we discussed in Section 2.3, streaming transactions and dataflow graphs require TE's for dependent stored procedures to be scheduled in an order that is consistent with the dataflow graph (i.e., not necessarily FIFO). This is, of course, true for other streaming schedulers, but here we must obey the rules defining correct schedules as stated earlier in Section 2.3. Additionally, as discussed in Section 2.4, the application can specify (via defining nested transactions) additional isolation constraints, especially when shared table state among streaming transactions is involved. The simplest solution is to require the TE's in a dataflow graph for a given input batch to always be executed in an order consistent with a specific topological ordering of that dataflow graph.

Although our ordering rules described earlier would allow transaction schedules that are "equivalent" to any topological ordering of the dataflow graph, our current scheduler implementation admits only one of them. We have found this approach to be practical in that it is amenable to a low-overhead implementation in H-Store and good enough to support all the S-Store use cases and benchmarks that we have so far studied (see Section 4). As we consider scaling to larger collections of workloads and nodes going forward, issues of fairness and locality may require more sophisticated approaches, such as flow-based scheduling [26].

3.2.5 Recovery Mechanisms

As described in Section 2.5, S-Store provides two different recovery options: (i) *strong recovery*, which is guaranteed to produce exactly the same state as was present before the failure (note that this guarantee is feasible only if the workload does not contain any non-determinism), and (ii) *weak recovery*, which will produce a legal state that could have existed, but is not necessarily the exact state lost. Both of these options leverage periodic checkpointing and command-logging mechanisms of H-Store. However, they differ in terms of which transactions are recorded in the command-log during normal operation and how they are replayed during crash recovery.

Strong Recovery. S-Store's strong recovery is very similar to H-Store's recovery mechanism. All committed transactions (both OLTP and streaming) are recorded in the command-log along with their input arguments. When a failure occurs, the system replays the command-log starting from the latest snapshot. The log is replayed in the order in which the transactions appear, which is the same as the order they were originally committed. This will guarantee the reads-from and the writes-to relationships between the transactions are strictly maintained.

There is one variation on H-Store's recovery, however. Before the log replay, we must first disable all PE triggers so that the execution of a stored procedure does not redundantly trigger the execution of its successor(s) in the dataflow graph. Because every transaction is logged in strong recovery, failing to do this would create duplicate invocations, and thus potentially incorrect results. Once triggers are disabled, the snapshot is applied, and recovery from the command-log can begin.

When recovery is complete, we turn PE triggers back on. At that point, we also check if there are any stream tables that contain tuples in them. For such streams, PE triggers will be fired to activate their respective downstream transactions. Once those transactions have been queued, then the system can resume normal operation.

Weak Recovery. In weak recovery, the command-log need not record all stored procedure invocations, but only the ones that ingest streams from the outside (i.e., border transactions). We then use a technique similar to *upstream backup* [25] to re-invoke the other previously committed stored procedures (i.e., interior transactions). In upstream backup, the data at the inputs to a dataflow graph are cached so that in the event of a failure, the system can replay them in the same way that it did on first receiving them in the live system. Because the streaming stored procedures in an S-Store dataflow have a well-defined ordering, the replay will necessarily create a correct execution schedule. While transactions may not be scheduled in the exact order that took place on the original run, some legal transaction order is ensured.

When recovering using weak recovery, we must first apply the snapshot, as usual. However, before applying the command-log, S-Store must first check existing streams for data recovered by the snapshot, and fire any PE triggers associated with those streams. This ensures that interior transactions that were run post-snapshot but not logged are re-executed. Once these triggers have been fired, S-Store can begin replaying the log. Unlike for strong recovery, we do not need to turn off PE triggers during weak recovery. In fact, we rely on PE triggers for the recovery of all interior transactions, as these are not recorded in the command-log. Results are returned through committed tables.

Weak recovery is a light-weight alternative to strong recovery, since it need not log all committed transactions. Section 4.2.3 provides an experimental comparison of our strong and weak recovery mechanisms.

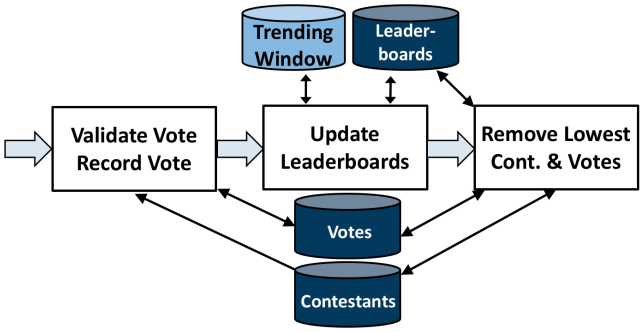


Figure 5: Leaderboard Maintenance Benchmark

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

Table 1: Guarantees vs Max Tput (Leaderboard Maintenance)

4. EXPERIMENTS

In this section, we present the results of our experimental study that evaluates S-Store with respect to existing alternatives in OLTP and stream processing. First, we demonstrate the benefits of integrating state management with push-based processing in Section 4.1. Specifically, we compare S-Store to H-Store, Esper, and Storm in terms of overall throughput on a transactional stream processing workload. Then, Section 4.2 further explores a number of micro-benchmarks that focus on evaluating specific architectural features of S-Store in comparison to its base system H-Store (i.e., EE triggers, PE triggers, and recovery modes).

To properly evaluate streaming workloads, we record throughput in terms of "input batches per second". This number represents the number of input batches that are processed to completion, regardless of the number of transactions executed. In order to simplify comparison to other systems, these experiments set the batch size to be a single tuple. For example, if any system processes 1,000 tuples / sec, we consider it to be processing 1,000 batches / sec.

All experiments were run on a cluster of machines using the Intel® Xeon® E7-4830 processors running at 2.13 GHz. Each machine contains a total of 64 cores and 264 GB of memory. Because we focus on single-node S-Store deployments in this paper and due to the partitioned architecture of S-Store, effectively only a single core is used for data access. In order to create data isolation for an apples-to-apples comparison, we limit data access to a single core on all featured systems. The experiments were run using a single non-blocking client which asynchronously sends requests to the system. Command-logging was enabled unless otherwise stated.

4.1 State-of-the-Art Comparison

In order to provide the best comparison between S-Store and state-of-the-art systems, we chose to implement a Leaderboard Maintenance benchmark that exercises all of the architectural additions of S-Store described in Section 3. We measure S-Store's performance against a main-memory OLTP system (H-Store [29]), a traditional single-node CEP engine (Esper [3]), and a modern distributed streaming system (Storm [37]).

4.1.1 Leaderboard Maintenance Benchmark

Consider a TV game-show in which viewers vote for their favorite candidate. Leaderboards are periodically updated with the number of votes each candidate has received.

Each viewer may cast a single vote via text message. Suppose the candidate with the fewest votes will be removed from the running every 20,000 votes, as it has become clear that s/he is the least popular. When this candidate is removed, votes submitted for him or her will be deleted, effectively returning the votes to the people who cast them. Those votes may then be re-submitted for any of the remaining candidates. This continues until a single winner is declared. During the course of the voting, each incoming vote needs to be validated and recorded. Furthermore, several leaderboards are maintained: one representing the top-3 candidates, another for the bottom-3 candidates, and a third one for the top-3 trending candidates of the last 100 votes. With each incoming vote, these leaderboards are updated with new statistics regarding the number of votes each candidate has received.

As shown in Figure 5, the dataflow graph contains three separate stored procedures: one to validate and insert a new vote, a second to maintain the leaderboard, and a third to delete a candidate if necessary. In order to ensure the correctness of the result in the presence of shared tables, as well as to maintain consistency of the tables across the dataflow graph, these three stored procedures must execute in sequence for each new vote.

4.1.2 OLTP Systems (H-Store)

As discussed at the beginning of Section 2, S-Store provides three primary guarantees: *ACID*, *ordered execution*, and *exactly-once processing*. When evaluating S-Store against an OLTP system (H-Store), it is important to consider which of these guarantees are being provided.

By default, H-Store provides only one of the three processing guarantees of S-Store: ACID. H-Store has no ordering guarantees, as it has no concept of a dataflow graph. It can instead choose any serializable transaction schedule (Section 3.1). In fact, we have previously shown that, in a workload in which multiple stored procedures within a dataflow share state like the one in Figure 5, H-Store may produce incorrect results [14]. H-Store also does not guarantee that a dataflow will be fully processed exactly once in the event of a system failure (again due to the lack of concept of a dataflow graph).

Because ordering guarantees are not considered, H-Store can asynchronously queue transactions for the engine to process. Thus, H-Store can send a transaction request and immediately send another without waiting for the response. The queue provides the system with a continuous supply of work, meaning H-Store is almost constantly doing transactional work. As a result, H-Store is able to process an impressive 5,300 input batches per second, as can be seen in Table 1.

By comparison, S-Store is able to achieve 2,200 input batches per second, while providing all three correctness guarantees. The primary performance difference lies within the *ordered execution* guarantee. To provide this, S-Store's scheduler must determine the proper order in which to run the transactions in its queue (discussed in Section 3.2.4). This scheduling does reduce the number of transactions per second that S-Store is able to process, but it is necessary to ensure correct results.

It is possible to execute the Leaderboard Maintenance benchmark on H-Store in a way that provides ordering guarantees. This is accomplished by designing a pseudo-"dataflow-graph" within the client. The parameters of a downstream procedure depend on the result from an upstream procedure, and transaction ordering must be ensured by the client. As a result, all procedures are forced to be invoked synchronously, meaning that a response must be received before the next request can be made.

This method ensures that the end results of the benchmark are correct, but performance suffers severely in the process. H-Store is only able to process 210 input batches per second when ordering is enforced by the client (see Table 1). Because all transaction calls are synchronous, H-Store's transaction queue never holds more than one transaction at a time. As a result, the client and the PE of H-Store must constantly wait for each other, severely hindering performance. S-Store, on the other hand, provides all three correctness guarantees while maintaining reasonable throughput.

4.1.3 *Streaming Systems (Esper and Storm)*

To compete with pure streaming systems, S-Store's performance must be comparable to both first-generation, single-node CEP engines as well as second-generation, distributed real-time streaming systems. We chose Esper and Storm as representative systems for their respective categories.

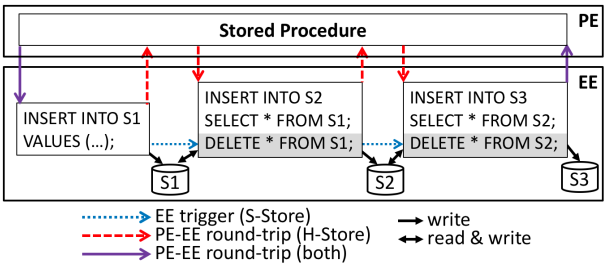
As further discussed in Section 5, neither Esper nor Storm are transactional. In order to provide comparable (though not comprehensive) guarantees to S-Store, only serialized tuple processing was allowed. All of Esper's default delivery ordering guarantees remain activated, meaning each tuple must run to completion before the next tuple may begin processing. For the Storm implementation, we opted to use Trident [5], an extension of Storm that supports stateful stream processing and exactly-once semantics. Data durability in both systems is provided by command-logging each of the three atomic processing units in the dataflow graph.

On stateless, pure streaming workloads that do not require transactional guarantees, both Esper and Storm would easily outperform S-Store. However, shared state management is key to many workloads, including our Leaderboard Maintenance benchmark. Like many stream processing systems, both Esper and Storm rely on external data storage for durable, shared state.

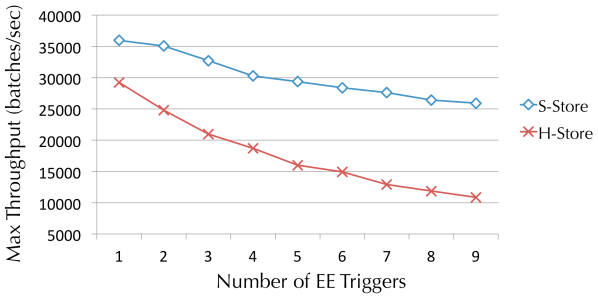
We added VoltDB[6], a main-memory, transactional database, as the backend for both Esper and Storm. VoltDB is an optimized, commercial version of H-Store, making the comparison with S-Store fair. Esper and Storm serve as the driving push-based engines, choosing when to access state based on the results received from the database. To maximize VoltDB's potential and batch requests from Esper / Storm to the database, we compile the three operations in Leaderboard Maintenance as VoltDB stored procedures. Each streaming system sends stored procedure requests via JDBC. Command-logging was unavailable in the open-source version of VoltDB, so asynchronous command logging was implemented in Esper and Storm.

After adding VoltDB, both Esper and Storm with Trident provide comparable guarantees to S-Store, outlined in Table 1. Esper (+VoltDB) provides two of the three processing guarantees of S-Store (ACID and ordered execution guarantees), but has no support for exactly-once semantics. Storm with Trident (+VoltDB) provides all three correctness guarantees.

As shown in Table 1, both Esper and Storm with Trident achieve roughly 600 batches per second, with data access being the significant bottleneck. At all times, either Esper or Storm is waiting for VoltDB, or vice-versa. Because tuples must be processed sequentially, only a single transaction request can be sent to VoltDB at a time, and the database must at a minimum wait for a full round-trip to and from the streaming system before it can process more work. Meanwhile, Esper and Storm must wait for VoltDB to process its transaction request before evaluating the response and continuing to process the dataflow graph.



(a) EE Trigger Micro-Benchmark



(b) EE Trigger Result

Figure 6: Execution Engine Triggers

By contrast, S-Store processes 2,200 batches per second. S-Store is able to handle multiple asynchronous transaction requests from the client and still preserve the tuple processing order. This is because all of the transaction ordering is handled directly by the S-Store partition engine. By combining the push-based semantics and fully-integrated state management, S-Store avoids the costly blocking communication between the streaming system and the database.

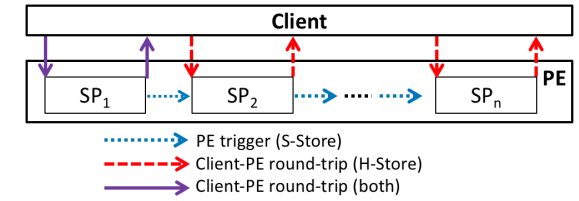
4.2 Micro-Benchmarks

A number of micro-experiments were performed to evaluate the optimizations achieved by S-Store over its predecessor, H-Store, in the presence of transactional stream processing workloads. For the experiments in Sections 4.2.1 and 4.2.2, command-logging was disabled to emphasize the feature being measured.

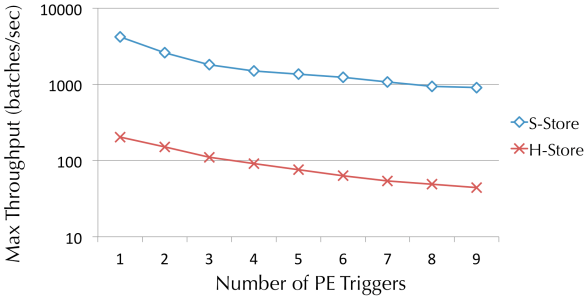
4.2.1 *Execution Engine Triggers*

In this experiment, we evaluate the benefit of S-Store's EE triggers. The micro-benchmark contains a single stored procedure that consists of a sequence of SQL statements (Figure 6(a)). In S-Store, these SQL statements can be activated using EE triggers such that all execution takes place inside the EE layer. H-Store, on the other hand, must submit the set of SQL statements (an insert and a delete) for each query as a separate execution batch from PE to EE. Figure 6(a) illustrates the case for 3 streams and 3 queries. S-Store's EE triggers enable it to trade off trigger execution cost for a reduction in the number of PE-to-EE round-trips (e.g., 2 triggers instead of 2 additional round-trips). Note also that the DELETE statements are not needed in S-Store, since garbage collection on streams is done automatically as part of our EE trigger implementation.

Figure 6(b) shows how maximum throughput varies with the number of EE triggers. S-Store outperforms H-Store in all cases, and its relative performance further increases with the number of



(a) PE Trigger Micro-Benchmark



(b) PE Trigger Result

Figure 7: Partition Engine Triggers

EE triggers, reaching up to a factor of 2.5x for 9 triggers. This trend continues as more EE triggers are added.

4.2.2 *Partition Engine Triggers*

This experiment compares the performance of S-Store's PE triggers to an equivalent implementation in H-Store, which has no such trigger support in its PE. As illustrated in Figure 7(a), the micro-benchmark consists of a dataflow graph with a number of identical stored procedures (SPs). Each SP removes tuples from its input stream, and then inserts these tuples into its output stream. We assume that the dataflow graph must execute in exact sequential order. In H-Store, the scheduling request of a new transaction must come from the client, and because the dataflow order of these transactions must be maintained, transactions cannot be submitted asynchronously. Serializing transaction requests severely limits H-Store's performance, as the engine will be unable to perform meaningful work while it waits for a client request (as discussed in Section 4.1.2). In S-Store, a PE trigger can activate the next transaction directly within the PE and can prioritize these triggered transactions ahead of the current scheduling queue using its streaming scheduler. Thus, S-Store is able to maintain dataflow order while both avoiding blockage of transaction executions and reducing the number of round-trips to the client layer.

Figure 7(b) shows how throughput (plotted in log-scale) changes with increasing dataflow graph size (shown as number of PE triggers for S-Store). H-Store's throughput tapers due to the PE's need to wait for the client to determine which transaction to schedule next. S-Store is able to process roughly an order of magnitude more input batches per second thanks to its PE triggers. Our experiments show that this benefit is independent of the number of triggers.

4.2.3 *Recovery Mechanisms*

As described earlier in Sections 2.5 and 3.2.5, S-Store provides two methods of recovery. *Strong recovery* requires every committed transaction to be written to the command-log. *Weak recovery*, on the other hand, is a version of upstream backup in which only committed border transactions are logged, and PE triggers allow interior transactions to be automatically activated during log replay. We now investigate the performance differences between these two methods, both during normal operation as well as recovery.

For the command-logging experiment, we use the same micro-benchmark presented in Section 4.2.2 (Figure 7(a)), using a dataflow with a variable number of SPs. Ordinarily in H-Store, higher throughput is achieved during logging by group-committing transactions, writing their log records to disk in batches. In S-Store, we have found that for trigger-heavy workloads, weak recovery can accomplish a similar run-time effect to the use of group commit. As shown in Figure 8(a), without group commit, logging quickly becomes a bottleneck in the strong recovery case. Each committed transaction is logged, so the throughput quickly degrades as the number of transactions in the dataflow graph increases. By contrast, weak recovery logs only the committed border transactions, allowing up to 4x the throughput as it writes a smaller fraction of log records to disk.

For the recovery experiment, we ran 5,000 input batches through the same PE micro-benchmark, recording logs for both weak and strong recovery. We then measured the amount of time it took S-Store to recover from scratch using each command-log.

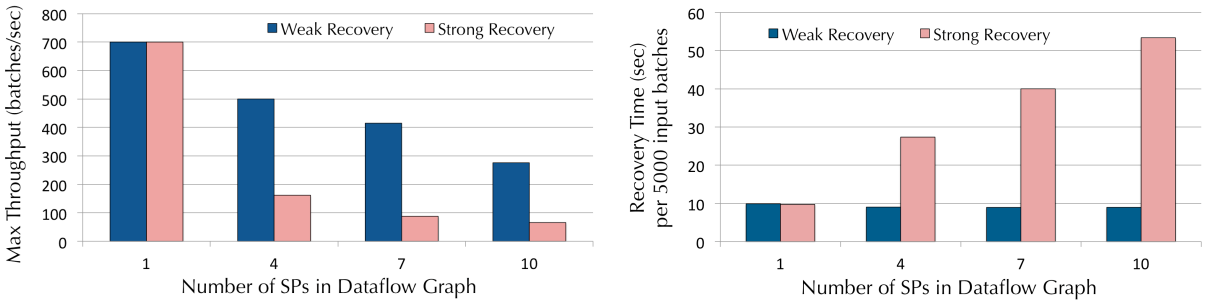
As shown in Figure 8(b), weak recovery not only achieves better throughput during normal operation, but it also provides lower recovery time. Typically during recovery, the log is read by the client and transactions are submitted sequentially to the engine. Each transaction must be confirmed as committed before the next can be sent. Because weak recovery activates interior transactions within the engine, the transactions can be confirmed without a round-trip to the client. As a result, recovery time stays roughly constant for weak recovery, even for dataflow graphs with larger numbers of stored procedures. For strong recovery, recovery time increases linearly with the size of the dataflow graph.

As previously stated, we expect the need for recovery to be rare, and thus prioritize throughput at run time over total recovery time. However, in real-time systems in which recovery time can be crucial, weak recovery can provide a significant performance boost while also improving run-time throughput.

5. RELATED WORK

In the early 2000's, there was a lot of interest in the database community for stream processing. The main goal of this work was to process continuous queries with low latency as data streamed into the system. This was largely inspired by the emergence of sensor-based applications. Many academic prototypes (Aurora / Borealis [7, 8], STREAM [10], TelegraphCQ [16], NiagraCQ [17]) were built, and several commercial products were spawned as a result of this work (e.g., TIBCO StreamBase, CISCO Truviso, SAP Coral8 / ESP, IBM InfoSphere Streams, Microsoft StreamInsight, Oracle CEP, Esper). With the exception of STREAM and Coral8, these systems did not support an explicit notion of transactions. STREAM did not directly claim to have transactions, but its execution model was based on logical timestamps which could be interpreted as transaction IDs. Batches of tuples with the same timestamp were executed atomically. While this could be used to provide isolation, recovery was not discussed. Furthermore, modeling transactions as the execution of an entire query graph did not allow finer-grained transaction definitions. Similarly, Coral8 provided so-called "atomic bundles" as a configurable isolation/recovery unit embedded in its execution model, but did not provide any transactional guarantees beyond "at least once" for processing events. Furthermore, none of these early systems considered integrating stream processing with traditional OLTP-style query processing.

Fault tolerance issues have been investigated as stream processing systems have been moved into distributed settings [8, 16]. A few fundamental models and algorithms have been established by



(a) Logging (b) Recovery

Figure 8: Recovery Mechanisms

this work [11, 25, 35], including the upstream backup technique that we leverage in our weak recovery mechanism [25].

There have also been several efforts in addressing specific transactional issues that arise in stream processing settings. For example, Golab et al. have studied the concurrency control problem that arises when a sliding window is advanced (write) while it is being accessed by a query (read) [22]. This work proposes subwindows to be used as atomic access units and two new isolation levels that are stronger than conflict serializability. Such a problem never arises in S-Store, since window state is accessed by a single TE at a time (and never by TEs of different SPs). As another example, Wang et al. have considered concurrency issues that arise when adding active rule support to CEP engines in order to monitor and react to streaming outputs [38]. In this case, the rules may require accessing state shared with other queries or rules. This work defines a stream transaction as a sequence of system state changes that are triggered by a single input event, and proposes a timestamp-based notion of correctness enforced through appropriate scheduling algorithms. S-Store investigates transactional stream processing in a more general context than active CEP engines.

Botan et al.'s work was the first to recognize the need for an explicit transaction model to support queries across both streaming and stored data sources [13]. This work proposed to extend the traditional page model [39] to include streams of events (as time-varying relations) and continuous queries (as a series of one-time queries activated by event arrivals). As a result, each one-time query execution corresponds to a sequence of read/write operations, and operations from one or more such sequences can be grouped into transaction units based on the application semantics. Transactions must then be executed in a way to ensure conflict serializability and event arrival ordering. Thus, this work focused on the correct ordering of individual read/write operations for a single continuous query, and not so much on transaction-level ordering for complex dataflow graphs like we do.

Recently, a new breed of stream processors has emerged. Unlike the majority of the earlier-generation systems, these do not adopt a select-project-join operator environment. Instead, they expect the user to supply their own operators (UDF's), and the system controls their execution in a scalable fashion over a cluster of compute nodes. Typically, these systems provide fault tolerance and recoverability, but do not support fully-ACID transactions. Essentially, they all aim at providing a MapReduce-like framework for real-time computations over streaming data. Representatives include Storm [37], Spark Streaming [40], Samza [2], Naiad [32], Flink [1], MillWheel [9], and S4 [33].

Storm provides two types of semantic guarantees: *at-least-once* and *at-most-once*. For *at-least-once*, each tuple is assigned a unique message-id and its lineage is tracked. For each output tuple t that is successfully delivered by a topology, a backflow mechanism is used to acknowledge the tasks that contributed to t with the help of a dedicated *acker bolt*. The data source must hold the tuple until a positive ack is received and the tuple can be removed (similar to upstream backup [25]). If an ack is not received within a given timeout period, then the source will replay the tuple again. Storm can only provide the weaker *at-most-once* semantics when the ack mechanism is disabled. Trident provides a higher-level programming abstraction over Storm which provides a stronger, *exactly-once* processing guarantee based on automatic replication [5]. While these guarantees ensure some level of consistency against failures, they are not sufficient to support atomicity and isolation as in the case of ACID guarantees. Furthermore, Storm focuses on purely streaming topologies and thus lacks support for dealing with persistent state and OLTP transactions.

Spark Streaming extends the Spark batch processing engine with support for discretized streams (D-Streams) [40]. Analytical computations are divided into a series of stateless, deterministic transformations over small batches of input tuples. Like STREAM, tuples are processed atomically within each of these batches. All state in Spark Streaming is stored in in-memory data structures called Resilient Distributed Datasets (RDDs). RDDs are partitioned and immutable. Like Storm+Trident, Spark Streaming provides *exactly-once* consistency semantics. Furthermore, the RDD-based state management model incurs high overhead for transactional workloads that require many fine-grained update operations (due to maintaining a large number of RDDs and managing their lineage).

Several of the new-generation streaming systems adopt a stateful dataflow model with support for in-memory state management. SEEP decouples a streaming operator's 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 [20]. SEEP has also been extended to support iterative cyclic computations [21]. Naiad extends the MapReduce model with support for structured cycles and streaming [32]. Naiad's timely dataflow model uses logical timestamps for coordination. 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, thus their consistency guarantees are weaker than S-Store's.

Microsoft Trill is a new analytics engine that supports a diverse spectrum of queries (including streaming, historical, and progressive/exploratory) with real-time to offline latency requirements [15]. Trill is based on a tempo-relational query model that incrementally processes events in batches organized as columns. Trill's adaptive batching and punctuation mechanisms enable trading off throughput for latency in case of higher loads. Both Trill and S-Store target hybrid workloads that include streaming, strive to maximize throughput while controlling latency, and are capable of in-memory

processing of events in adjustable batch granularity. However, S-Store focuses more on OLTP settings with shared mutable state, whereas Trill focuses more on OLAP settings with read-mostly state. Therefore, S-Store pays more attention to providing correctness guarantees in the face of concurrent access, processing dependencies, and failures without sacrificing performance.

6. SUMMARY & FUTURE DIRECTIONS

This paper has defined a new model of transactions for stream processing. We have presented the design and implementation of a novel system called S-Store that seamlessly combines OLTP transaction processing with our transactional stream processing model. We have also shown how this symbiosis can be implemented in the context of a main-memory, OLTP DBMS in a straight-forward way. S-Store is shown to outperform H-Store, Esper, and Storm on a streaming workload that requires transactional state access, while at the same time providing stronger correctness guarantees.

Future work includes extending S-Store to operate on multiple nodes. We plan to address a number of research issues including data and workload partitioning, distributed recovery, and distributed transaction scheduling. We also plan to investigate handling of dynamic and hybrid (OLTP+streaming) workloads.

Acknowledgments. We thank Richard Tibbets for sharing his experience about StreamBase use cases, as well as Chenggang Wu and Hong Quach for their contributions. This research was funded in part by the Intel Science and Technology Center for Big Data, by the NSF under grants NSF IIS-1111423 and NSF IIS-1110917, and by the Maseeh Professorship in Emerging Technologies.

7. REFERENCES

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

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

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

[4] TIBCO StreamBase. `http://www.streambase.com/`.

[5] Trident Tutorial. `http://storm.apache.org/documentation/Trident-tutorial.html`.

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

[7] D. Abadi et al. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):120–139, 2003.

[8] D. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, pages 277–289, 2005.

[9] T. Akidau et al. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *PVLDB*, 6(11):1033–1044, 2013.

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

[11] M. Balazinska et al. Fault-tolerance in the Borealis Distributed Stream Processing System. *ACM TODS*, 33(1):3:1–3:44, 2008.

[12] I. Botan et al. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. *PVLDB*, 3(1):232–243, 2010.

[13] I. Botan et al. Transactional Stream Processing. In *EDBT*, pages 204–215, 2012.

[14] U. Cetintemel et al. S-Store: A Streaming NewSQL System for Big Velocity Applications (Demonstration). *PVLDB*, 7(13):1633–1636, 2014.

[15] B. Chandramouli et al. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. *PVLDB*, 8(4):401–412, 2014.

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

[17] J. Chen et al. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, pages 379–390, 2000.

[18] C. Diaconu et al. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*, pages 1243–1254, 2013.

[19] A. Elmore et al. A Demonstration of the BigDAWG Polystore System (Demonstration). *PVLDB*, 8(12):1908–1919, 2015.

[20] R. C. Fernandez et al. Integrating Scale-out and Fault-tolerance in Stream Processing using Operator State Management. In *SIGMOD*, pages 725–736, 2013.

[21] R. C. Fernandez et al. Making State Explicit for Imperative Big Data Processing. In *USENIX ATC*, pages 49–60, 2014.

[22] L. Golab et al. On Concurrency Control in Sliding Window Queries over Data Streams. In *EDBT*, pages 608–626, 2006.

[23] L. Golab and T. Johnson. Consistency in a Stream Warehouse. In *CIDR*, pages 114–122, 2011.

[24] L. Golab and T. Ozsu. Issues in Data Stream Management. *ACM SIGMOD Record*, 32(2):5–14, 2003.

[25] J.-H. Hwang et al. High-Availability Algorithms for Distributed Stream Processing. In *ICDE*, pages 779–790, 2005.

[26] M. Isard et al. Quincy: Fair Scheduling for Distributed Computing Clusters. In *SOSP*, pages 261–276, 2009.

[27] N. Jain et al. Towards a Streaming SQL Standard. *PVLDB*, 1(2):1379–1390, 2008.

[28] T. Johnson et al. Query-aware Partitioning for Monitoring Massive Network Data Streams. In *SIGMOD*, pages 1135–1146, 2008.

[29] R. Kallman et al. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *PVLDB*, 1(2):1496–1499, 2008.

[30] A. Lerner and D. Shasha. The Virtues and Challenges of Ad Hoc + Streams Querying in Finance. *IEEE Data Engineering Bulletin*, 26(1):49–56, 2003.

[31] N. Malviya et al. Rethinking Main Memory OLTP Recovery . In *ICDE*, pages 604–615, 2014.

[32] D. G. Murray et al. Naiad: A Timely Dataflow System. In *SOSP*, pages 439–455, 2013.

[33] L. Neumeyer et al. S4: Distributed Stream Computing Platform. In *KDCloud*, pages 170–177, 2010.

[34] A. Pavlo et al. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *SIGMOD*, pages 61–72, 2012.

[35] M. A. Shah et al. Highly Available, Fault-tolerant, Parallel Dataflows. In *SIGMOD*, pages 827–838, 2004.

[36] A. Silberschatz et al. *Database System Concepts*. McGraw-Hill, 2010.

[37] A. Toshniwal et al. Storm @Twitter. In *SIGMOD*, pages 147–156, 2014.

[38] D. Wang et al. Active Complex Event Processing over Event Streams. *PVLDB*, 4(10):634–645, 2011.

[39] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.

[40] M. Zaharia et al. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP*, pages 423–438, 2013.