S-Store: A Streaming NewSQL System for Big Velocity Applications

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

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

ABSTRACT

First-generation streaming systems did not pay much attention to state management via ACID transactions (e.g., [3, 4]). S-Store is a data management system that combines OLTP transactions with stream processing. To create S-Store, we begin with H-Store, a main-memory transaction processing engine, and add primitives to support streaming. This includes triggers and transaction workflows to implement push-based processing, windows to provide a way to bound the computation, and tables with hidden state to implement scoping for proper isolation. This demo explores the benefits of this approach by showing how a naïve implementation of our benchmarks using only H-Store can yield incorrect results. We also show that by exploiting push-based semantics and our implementation of triggers, we can achieve significant improvement in transaction throughput. We demo two modern applications: (i) leaderboard maintenance for a version of "American Idol", and (ii) a city-scale bicycle rental scenario.

1. INTRODUCTION

Managing high-speed data streams generated in real time is an integral part of today's big data applications. In a wide range of domains from social media to financial trading, there is a growing need to seamlessly support incremental processing as new data is generated. At the same time, the system must ingest some or all of this data into a persistent store for on-demand transaction or analytical processing. In particular, this is true for applications in which high-velocity data updates a large persistent state such as leaderboard maintenance or online advertising. In these situations, there is need to manage streaming and non-streaming state side by side in a way that ensures transactional integrity and performance.

Today's stream processing systems lack the required transactional robustness, while OLTP databases do not provide native support for data-driven processing. In this work, our goal is to build a single, scalable system that can support both stream and transaction processing at the same time. We believe that modern distributed main-memory OLTP platforms, also known as *NewSQL* systems [5], provide a suitable foundation for building such a system, since (i) they are more lightweight than their traditional disk-based counterparts; (ii) like streaming engines, they offer lower latency via in-memory processing; and (iii) they provide strong support for state and transaction management. Thus, we introduce S-Store, a streaming OLTP system that realizes our goal by extending the H-Store DBMS [6] with streams.

We propose to demonstrate the S-Store streaming NewSQL system and several of its novel features that include:

Architecture: S-Store makes a number of fundamental architectural extensions to H-Store that generally apply to making any main-memory OLTP system stream-capable. Thus, the first goal of this demonstration is to highlight our architectural contributions.

Transaction Model: S-Store inherits H-Store's ACID transaction model and makes several critical extensions to it. The streaming nature of the data requires dependencies between transactions, and S-Store provides ACID guarantees in the presence of these dependencies. We will show how our extended model ensures transactional integrity.

Performance: S-Store's native support for streams not only makes application development easier and less error-prone, but also boosts performance by removing the need to poll for new data and by reducing the number of round-trips across various layers of the system. We demo these features by comparing H-Store and S-Store.

Applications: S-Store can support a wide spectrum of applications that require transactional processing over both streaming and non-streaming data. The demo will present a select set of these applications, highlighting different technical features of the system as well as its support for diverse workloads.

The rest of this paper provides an overview of the S-Store system and the details of our demo scenarios.

2. S-STORE SYSTEM OVERVIEW

S-Store belongs to a new breed of stream processing systems designed for high-throughput, scalable, and fault-tolerant processing over big and fast data across large clusters. Like its contemporaries such as Twitter Storm/Trident [9] or Spark Streaming [10], S-Store supports complex computational workflows over streaming and non-streaming data sets. S-Store is unique in that all data access in S-Store is SQL-based and fully transactional.

S-Store builds on the H-Store NewSQL system [6]. H-Store is a high-performance, in-memory, distributed OLTP system designed for shared-nothing clusters. It targets OLTP workloads with short-lived transactions, which are pre-defined as parameterized stored procedures (i.e., SQL queries embedded in Java-based control code) that are invoked by client requests at run time. As with most distributed database systems, a good H-Store design partitions the database in a way that processes most of the transactions in a single-sited manner, minimizing the number of distributed transactions and reducing the overhead of coordination across multiple partitions [8].

Proceedings of the VLDB Endowment, Vol. 7, No. 13

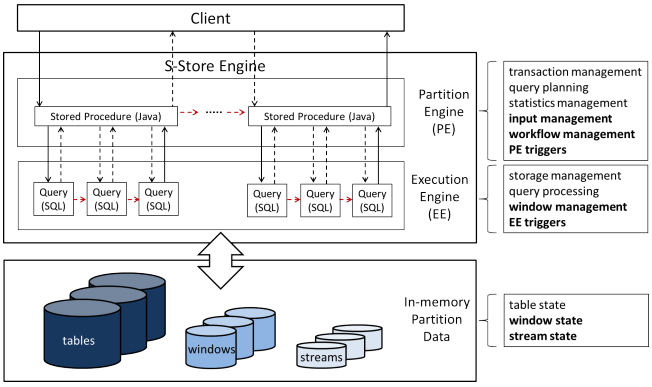

Figure 1: S-Store Architecture

In the single-sited case, H-Store runs all transactions serially, removing the need for fine-grained locks and latches. Fault tolerance is achieved through a technique that combines command logging and periodic snapshotting [7]. S-Store inherits all of these core features and extends them in non-trivial ways. We intend to demonstrate these extensions for the single-sited case.

S-Store is essentially a client-server system that follows H-Store's architecture. Each server node consists of two layers: the partition engine (PE) and the execution engine (EE). The PE is responsible for receiving and managing transaction requests from clients in the form of stored procedure invocations (each consisting of a name and a set of input parameter values). This includes tasks such as query planning and transaction management (i.e., scheduling, recovery, distributed transaction coordination). On the other hand, the EE is responsible for low-level storage management and query processing tasks such as indexing and operator execution.

Fig. 1 shows S-Store's high-level architecture. The base architecture is directly inherited from H-Store as described above. In addition, a number of extensions are made to enable native stream processing support in the engine (shown in boldface in Fig. 1). These include: (i) management of inputs from streaming clients and complex workflows of stored procedures at the PE layer, (ii) management of stream- and window-based queries at the EE layer, (iii) management of in-memory stream and window state.

In the following, we describe the key features of our design that enabled these architectural extensions:

Streams, Windows, Triggers, and Workflows: We have added four new constructs to H-Store: (i) *streams* to differentiate continuously flowing state from regular stored state; (ii) *windows* to define finite chunks of state over (possibly unbounded) streams; (iii) *triggers* to indicate computations to be invoked for newly generated data; and (iv) *workflows* to represent pipelines of dependent stored procedures triggered by downstream outputs.

Uniform State Management: H-Store's in-memory tables are used for representing all states including streams and windows, making state access both efficient and transactionally safe. Unlike regular tables, stream and window state has a short lifespan determined by the queries accessing it. To support this, S-Store provides automatic garbage collection mechanisms for tuples that expire from stream or window state.

Data-driven Processing via Triggers: Special insert triggers are defined on stream or window state in order to enable push-based, data-driven processing in S-Store. There are two types of triggers: EE triggers at the query level and PE triggers at the stored procedure level (shown as red dashed arrows in Fig. 1). The former enable continuous processing within a given transaction execution, while the latter do so across multiple transaction executions that are part of a common workflow. S-Store triggers differ from conventional SQL triggers in the sense that they react to the presence of data from a well-defined set of sources (i.e., they act as "control triggers" rather than "data triggers").

Stream-oriented Transaction Model: S-Store adopts H-Store's transaction model: transactions are defined using stored procedures and are executed with ACID guarantees. However, the inherent stream processing semantics in S-Store exposes data and processing dependencies among transaction executions that are not captured by this model. Thus, S-Store makes a number of extensions to capture them and maintain ACID properties in their presence. We sketch some of these below.

- An S-Store transaction is defined by two things: a stored procedure definition and a batch of input tuples. For this purpose, we distinguish between *border stored procedures* (BSP) and *interior stored procedures* (ISP). A BSP is one that sits at the input of the workflow (i.e., has no upstream stored procedure). All others are defined to be ISPs. Transaction executions for BSPs are defined by a batch of tuples as specified by the user (e.g., 2 tuples). A transaction based on an ISP is also defined by batches; however, that batch is generated and appears on the output stream of the immediate upstream stored procedure. A transaction commits when its input batch has been completely processed.
- A given stored procedure will be executed many times during the lifetime of a workflow. We call each of these a *transaction execution* (TE). A legal schedule of transaction executions must preserve the natural order. That is, the first transaction execution for stored procedure SP_i must precede the second transaction execution for SP_i, etc.
- For a given stored procedure, partial window state may carry over from one TE to the next. This is due to the continuous nature of streaming applications. A window in SP_i may contain state that was produced by previous TEs of SP_i. Such state must be protected from the access of arbitrary TEs. Thus, we introduce the notion of "scope of a transaction execution" to restrict window access to only consecutive TEs of a given stored procedure.
- The fact that a stored procedure ($SP1$) may precede another one ($SP2$) in a workflow leads to processing dependencies between their TEs. For a given input batch, the system must produce a schedule that is equivalent to executing $SP1$ first and $SP2$ second (a "serializable schedule" in S-Store). Furthermore, when there are shared writable tables along a workflow, S-Store requires a serial execution of the involved stored procedures.
- In addition to these extensions, we leverage H-Store's command logging mechanism to provide an upstream backup based fault tolerance technique for our streaming transaction workflows.

The native stream processing extensions described above allows S-Store to use the layered engine architecture in the most efficient way by embedding the required functionality into the proper component. This allows the system to avoid redundant computations (e.g. for windowing), communication across the layers, and the need to poll for new data. The latter two are illustrated with the horizontal red dashed arrows replacing the vertical black dashed arrows in Fig. 1. It is important to note that, although discussed within the context of H-Store in this paper, we believe that the architectural additions described above generally apply when extending any in-memory OLTP engine with streams. In fact, the two-layer engine is the only H-Store-specific element of our design, which essentially leads to having two levels of triggers in the engine rather than one.

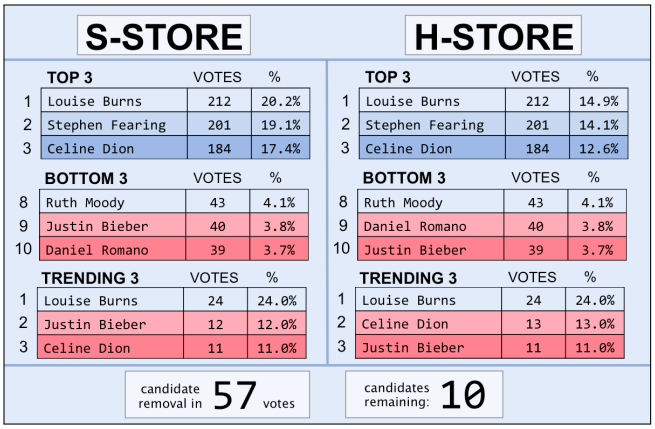

Figure 2: Real-time Display for Voter Leaderboard

3. DEMONSTRATION SCENARIOS

We will demonstrate the key features of the S-Store system using two classes of applications: (i) those that resemble classic OLTP workloads, but also involve streaming inputs and computations, and (ii) those that apply complex real-time computations over streams as in classic stream processing workloads, but also involve maintaining and sharing state with transactional guarantees.

Through these applications, we will show how the built-in streaming and transactional processing primitives in S-Store facilitate reliable application development, improve transaction throughput, and ensure correct executions and results when compared to a pure OLTP engine (e.g., H-Store).

In the following, we describe two selected application scenarios: one that is based on the Voter OLTP Benchmark [2], and another that models a sustainable green city application inspired by the New York City bike share program [1].

3.1 Voter with Leaderboard

A Canadian game show, *Canadian Dreamboat*, seeks to join the ranks of popular teen icon reality shows. At the beginning of the show, 25 candidates are presented to viewers, who then vote on which candidate is the most talented. Each voter may cast a single vote via text message. Once 100 total votes have been cast, the system removes the candidate with the lowest number of votes from the running, 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, several leaderboards are maintained: one representing the top three candidates, one representing the bottom three candidates, and one representing the top three trending candidates of the last 100 votes (Fig. 2). With each incoming vote, these leaderboards are to be updated with new statistics regarding the number of votes each candidate has received.

This workflow is divided into three stored procedures (Fig. 3). The first stored procedure ($SP1$) validates each vote (checks that the contestant exists, checks that the corresponding phone number has not yet voted, etc.) and records new votes as they arrive. Once $SP1$ has committed, the second stored procedure ($SP2$) is invoked. $SP2$ updates the leaderboards and maintains a running count of the total number of votes received. Once the count reaches 100, the third stored procedure ($SP3$) is invoked. $SP3$ removes the candidate with the lowest number of votes from the Contestants table, simultaneously removing all votes for that candidate from the Votes table and all leaderboards.

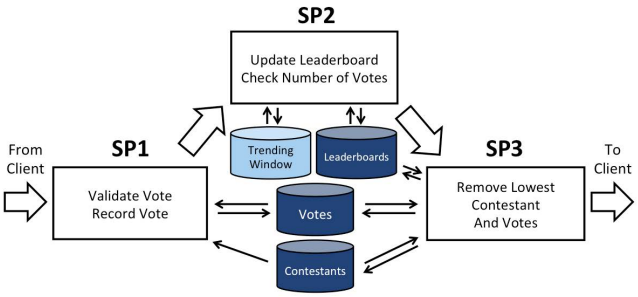

Figure 3: Leaderboard Maintenance Workflow

Because table state is shared among $SP1$, $SP2$, and $SP3$, they must run serially. Once the 100th vote is received and recorded, $SP3$ will immediately remove the candidate with the least number of votes before $SP1$ or $SP2$ is allowed to run again. This will guarantee that no votes will be processed out of order and lead to unexpected results.

H-Store makes no such guarantee. In H-Store, it is possible that additional votes will be processed by $SP1$ or $SP2$ before $SP3$ has a chance to run. As a result, potentially valid votes may be thrown away, or votes for an invalid candidate may be counted incorrectly. Take for instance a scenario in which candidate X is the contestant with the fewest votes once the 100th vote is received. Ideally, $SP3$ will recognize X as having the lowest vote total and remove him or her from the show before allowing any new votes. However, suppose that the next 20 new votes arrive, all cast for candidate X, pushing his total above candidate Y. If $SP3$ runs *after* those votes are counted, then candidate Y will be removed instead.

Furthermore, it is possible that an incorrect vote may be counted in H-Store due to a lack of ordering transactions by votes' arrival order. Suppose that a user submits a vote for candidate X, then another vote for candidate Y before the first has been recorded. Ideally, the vote for X should be counted, and the vote for Y rejected. However, if the ordering is not maintained, the vote for Y may be counted instead of the vote for X.

Because S-Store processes incoming requests in arrival order and triggers transactions in workflow order, the problems listed above do not occur. We intend to show the disparity between S-Store and H-Store by running the two systems side by side, using one client to send the same vote requests to each. As votes are counted, the inconsistencies between the two systems will lead to a difference in the recorded votes for each candidate. Eventually, this will lead to not only incorrect candidates being removed from the H-Store version but also the possibility for a false winner to be selected.

We also intend to show that S-Store is more efficient than H-Store at handling streaming workloads such as *Voter with Leaderboard*. The difference comes from a reduction of Client-to-PE round trips due to push-based workflow processing, as well as a reduction of PE-to-EE round trips due to native support for windowing. In both cases, H-Store requires additional communication to provide similar functionality. We intend to show this by running instances of S-Store and H-Store simultaneously, displaying the number of transactions per second that each is processing.

3.2 Bicycle Sharing

In this scenario, we examine a current trend common in many progressive cities: bicycle sharing. Take a hypothetical company, *BikeShare*, which allows its customers to rent a bike from one of its many stations throughout the city, ride it, and then return it to any available dock at any desired station.

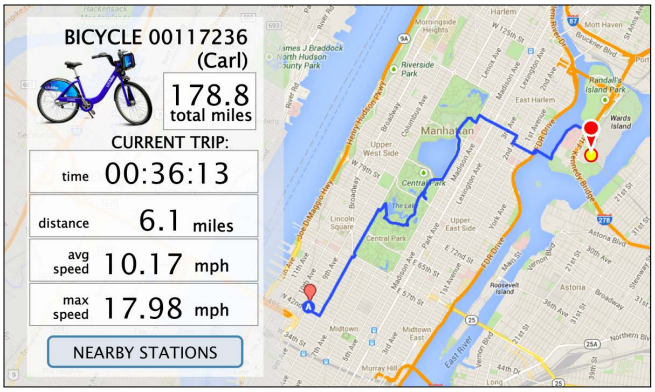

Figure 4: BikeShare Streaming Data of a Single Bike

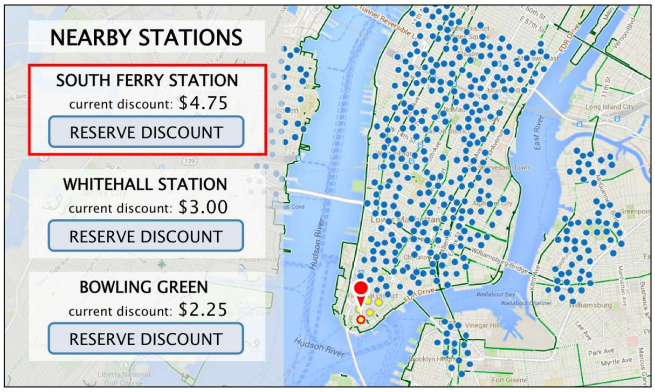

Figure 5: Map of BikeShare Stations and Nearby Discounts

A BikeShare member can check out or return a bike at any BikeShare station, effectively issuing an OLTP request to the BikeShare database through the company's app. This in turn updates the current status of the corresponding bike, dock, and station. A ride ends upon bike return and the member's credit card is charged based on the amount of the time the bike was rented.

Each bike contains a GPS unit, which reports its location at a high frequency (e.g., once a second) into the BikeShare database, where it is recorded and analyzed in real time. During a trip, the rider is able to gain information about the distance s/he has traveled, his or her average and maximum speeds throughout the trip, as well as other aggregate statistics such as calories burned. In addition, s/he will be able to track the path s/he traveled and gain information about specific portions of the ride including nearby stations. BikeShare is also able to use this information to collect usage statistics and detect anomalies (for instance, a bike traveling at 60 mph may indicate that the bike is on the back of a truck, likely being stolen).

In order to promote bike availability at all stations, BikeShare also offers real-time discounts for users who drop off their bikes at stations in need of bikes. The discounts for dropoff stations near the user are continuously updated depending on the available number of bikes at those stations as well as the current position of the user. The user may accept a discount offer for a nearby station of his or her choice by clicking on a button on the company's app. This in turn assigns the selected offer to the user for a period of 15 minutes, while removing it from the list of available discounts for that station. The discount will expire if the user does not make a bike return to the corresponding station within the allotted period, possibly making that discount available to other nearby users. Nearby discounts are dynamically calculated in real time based on all streaming bike positions, continuously changing the status of the stations as checkouts or returns take place, as well as the discount acceptances or expirations occurring for specific users. Transactional processing is required to ensure correct calculation of these discounts.

This workload involves pure OLTP (e.g., bike checkouts and returns), pure streaming (e.g., real-time statistics and alerts), and a combination of the two (e.g., real-time discount calculation). S-Store makes it possible to handle the entire workload within a single system. With this demonstration, we intend to show the versatility of S-Store, and the ease with which a developer may create an application featuring two traditionally disparate functionalities. The demo will feature several map-based GUIs. For example, one will interact with BikeShare users for displaying ride statistics and nearby stations with discounts, as well as receiving OLTP requests on bike checkouts, bike returns, and discount offer acceptances (Fig. 4 and Fig. 5). Another one will be used by the BikeShare company for real-time monitoring purposes (e.g., displaying BikeShare stations with the number of bikes and docks that are available at each, locations of all bikes that are currently being ridden, stolen bike alerts). The demonstration will feature virtual users borrowing and returning bikes, and real-time discounts being offered as one BikeShare station starts running out of bikes due to a sudden increase in checkouts.

4. SUMMARY

From our experience with stream processing applications, it is clear that they all require persistent state. We described S-Store, a new system that addresses the state management shortcomings of previous stream processing systems. In particular, it incorporates ACID transactions by building on H-Store, a main-memory OLTP DBMS. Our demo will illustrate this functionality through the lens of two applications that require transactional support. We also demonstrate the somewhat surprising result that, by using the streaming components, S-Store can achieve higher throughput than H-Store alone, while still providing transactional guarantees.

Acknowledgments. We thank the PSU BikeShare Capstone team and Hong Quatch from Intel/PSU for their contributions to the BikeShare application. This research was funded in part by the Intel Science and Technology Center for Big Data.

5. REFERENCES

[1] CitiBike. `http://www.citibikenyc.com/`.

[2] H-Store Supported Benchmarks. `http://hstore.cs.brown.edu/documentation/deployment/benchmarks/`.

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

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

[5] M. Aslett. How will the database incumbents respond to NoSQL and NewSQL? The 451 Group, 2011.

[6] R. Kallman et al. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. In *VLDB*, 2008.

[7] N. Malviya et al. Rethinking Main Memory OLTP Recovery. In *ICDE*, 2014.

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

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

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