Abstract of "Transactional Streaming: Managing Push-Based Workloads with Shared Mutable State in a Distributed Setting" by John Meehan, Ph.D., Brown University, May 2019.

In the past, streaming data management systems (SDMSs) have eschewed transactional management of shared mutable state in favor of low-latency processing of high-velocity data. Streaming workloads involving ACID state management have required custom solutions involving a combination of database systems, complicating scalability by adding blocking communication, which severely impedes performance. This dissertation presents a solution: a transactional streaming system, S-Store. Built on a main-memory OLTP system, S-Store implements a novel transaction model and gives best-in-class performance on hybrid streaming and OLTP workloads while maintaining the correctness guarantees of both worlds. To achieve optimal performance, S-Store distributes processing across a cluster of machines. This requires an implementation which solves the unique challenges to system design and partitioning that arise from dataflow state management. This dissertation describes heuristics for optimizing two overarching workload categories: those that use data parallelism and those that use pipeline parallelism. Transactional streaming has opened opportunities for new solutions to real-world problems, particularly in streaming data ingestion. Together, these contributions provide a foundation for a new class of streaming research in which ACID state management is an essential feature rather than an afterthought.

Transactional Streaming: Managing Push-Based Workloads with Shared Mutable State in a Distributed Setting

by

John Meehan

Department of Computer Science

Brown University

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island

May 2019

This dissertation by John Meehan is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date: _____ _____

Stanley B. Zdonik, Director
Brown University

Recommended to the Graduate Council

Date: _____ _____

Nesime Tatbul, Reader
Intel Labs, Massachusetts Institute of Technology

Date: _____ _____

Ugur Çetintemel, Reader
Brown University

Approved by the Graduate Council

Date: _____ _____

Andrew G. Campbell
Dean of the Graduate School

Curriculum Vitae

John Leonard Meehan is the son of Mike and Lynn Meehan. He graduated cum laude from the University of Notre Dame in 2009 with a Bachelors of Science in Computer Science and Film. He worked as a Database Engineer at SP Capital IQ until 2012, when he joined the Ph.D. program at the Brown University Computer Science Department. He earned his Masters of Science there in 2014, and successfully defended his dissertation in December 2018.

Acknowledgements

I owe all of my success in this project to Stan Zdonik and Nesime Tatbul, both of whom have been phenomenal advisors during my time in graduate school. Stan and Nesime have provided an enormous amount of invaluable support and advice, and were deeply involved in the intellectual creation and writing of the academic papers associated with the S-Store Project. Perhaps even more importantly, I feel very proud to call Stan and Nesime friends, and have received both encouragement and support throughout my career at Brown.

I have had the pleasure of working with a number of other faculty members during my time at Brown. I have received a great deal of advice from Ugur Çetintemel, who helped me to find my footing as I entered graduate school and provided much-needed perspective as the project progressed. Kristin Tufte, David Maier, and the Portland State team were extremely helpful in finding and developing S-Store's niche in the database world.

This work would not be possible without the entire S-Store team. In particular, I would like to thank Hawk Wang, who provided engineering expertise in the early S-Store codebase, and Jiang Du, who has contributed a great deal of code and input throughout S-Store's lifespan (particularly as relates to the BigDAWG project). Additionally, I have had the privilege to work with many extremely talented graduate and undergraduate students, each of which helped to build S-Store into the open-source system it is today. In particular, I owe a deep gratitude to students from Brown (Chenggang Wu, Shaobo Tian, Yulong Tian, Christian Mathiesen, Bikong Wang, Tommy Yao, Cansu Aslantas, and Eric Metcalf) and Portland State University (Hong Quach, Erik Sutherland, and Christopher Giossi). And of course, a special thanks to Andy Pavlo both for his help navigating the H-Store codebase, but also for creating the most well-developed and documented research software that I have encountered.

I would additionally like to thank Intel, which has funded the S-Store project though the ISTC

program. Intel has also provided a great deal of support to the S-Store project in the form of opportunities and exposure, particularly via other projects such as the BigDAWG Polystore.

Finally, I would like to thank my friends and family, particularly my parents, Lynn and Mike Meehan, who were extremely supportive of me from beginning to end. I would especially like to thank my girlfriend Alexis Jackson, who is the foremost expert in S-Store outside of the authors of these papers (despite being in a completely separate professional field). She has also been extremely supportive and patient with me throughout this process (which was frequently quite the feat).

This thesis, and Chapters 3, 4, and 5 in particular, is largely based on several papers that were jointly written with Stan Zdonik and Nesime Tatbul, as well as the many members of the S-Store project from Brown University, MIT, Portland State University, and the University of Toronto. Chapter 3 originates from the work in our S-Store VLDB 2015 paper [110] and our contribution to the IEEE Special Issue on Next-Generation Stream Processing [117]. Chapter 4 is largely based on work in submission to VLDB 2019, entitled *Transactional Stream Processing at Scale*. Chapter 5 is based on work published in CIDR 2017 [109] and HPEC 2016 [111].

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The priorities of data processing systems have dramatically shifted throughout their history. As computing hardware evolves and new technologies are made available, so too do the requirements for data processing. One workload may prioritize accessibility, another data accuracy, and still another raw computing performance.

Early database systems were primarily comprised of the relational model. Beginning in the 1970's systems such as System R [27] and INGRES [143] used the relational database model primarily for "business data processing" [142]. These systems and others were specifically designed for on-line transaction processing applications, otherwise known as *OLTP workloads*. OLTP workloads involve regular updates to the database via either individual transactions or bulk inserts [73]. These early databases included a number of common features, such as row storage in tables, B-trees for indexing, and ACID transaction properties.

As hardware evolved, so too did the quantities of data available, and the speed at which the data could be processed. It became increasingly clear that the relational model, while very effective at many workloads, was insufficient for several modern use cases [143]. A variety of high performance data-processing solutions emerged: column-store databases for column-based on-line analytical processing [140, 136], so-called NoSQL systems for large-scale Internet applications [45], and array databases designed to handle scientific datasets [40, 122]. Most relevantly, the concept of streaming data processing was developed to handle high-velocity data immediately upon arrival.

In the development of these specialized databases, some old database standards were eschewed in favor of increased performance. Specifically, the ACID transactions for concurrent state management fell by the wayside in several of the branching database lineages. When stream processing systems were created in the early 2000's, all state management would take place on-disk, making the process of modifying any state transactionally slow and computationally expensive relative to

any processing done in memory.

However, there are many workloads that can benefit from both the performance gains of a specialized database, while also maintaining the essential features of a relational database. As computer hardware improves, it becomes possible to re-integrate the missing functionalities, particularly ACID transactions, into specialized data processing models such as stream processing.

1.1 ACID Transactions and Serializable Scheduling

In traditional databases, state is assumed to be reliable at all times. Table state is able to be accessed and modified by a variety of clients and operations; this is known as *shared mutable state*. In order to protect the consistency of shared mutable state, operations that read and/or write to the database are confined to *transactions*.

One of the core features of relational database systems, and the focus of this dissertation, is the notion of *ACID transactions* [73]. ACID transactions consist of four elements, for which the acronym stands: 1) atomicity, 2) consistency, 3) isolation, and 4) durability [73, 131]. The specifics of each element are as follows:

Atomicity: All operations within a transaction are *atomic*, especially with regards to writes / changes. Either the entire transaction is completed (*commits*), or all of the operations fail (*abort*).

Consistency: At the end of the transaction, any transformations to the state are correct, or *consistent*. The operations do not violate any integrity constraints associated with the state, or other transformation rules.

Isolation: Transactions may execute concurrently, but each transaction must be able to execute in a *serializable* manner with regards to other transactions. This ensures that, from any transaction's perspective, state is *isolated* while it is being accessed.

Durability: Once a transaction has committed, any changed state is *durable* to failures. In the event of a system crash, it is able to be recovered.

Because all of these features are abstracted into a single programming model, developers are left with more flexibility when developing concurrent programs [68, 123]. The database itself ensures that operations are executed with transactional correctness, promising to protect the consistency of shared, mutable state. This is particularly true when multiple operations take place within a single transaction. For instance, a single transaction may contain two operations reading a value from

tables $T1$ and $T2$, and a third that updates a value in $T3$ based on the earlier reads. ACID transactions ensure the developer that these operations execute as a single unit, and that if the transaction commits, it will be without any external interference.

The most widely-accepted definition of "correct" concurrent execution of transactions is that they must be executed in an order that is *serializable* [34, 68]. This means that, regardless of the order in which a set of transactions actually execute, there exists a schedule in which those transactions can be executed serially, one after the other. Transactions can potentially be interleaved, but must be executable in a non-interleaved fashion (though not a particular serial order).

Traditional OLTP systems ensure serializability by *scheduling* transaction executions in a correct order [68]. The operations within a given transaction have a natural ordering, specified within the transaction definition itself. Additionally, there are *conflicting operations* between different transactions. Two operations are considered conflicting if they both access the same piece of state, and at least one of them updates that state. All operations are scheduled in a way that they could theoretically be re-ordered to create a serial execution of the transactions.

1.2 Push-based Processing: Streaming Systems

From the 1970's on into the early 2000's, database management systems were primarily designed for the computer hardware typically available: single computers with a single CPU core and a relatively small amount of main memory [123]. This meant that all permanent database state was stored on disk. The mechanisms required for accessing state transactionally (locking, logging, etc.) were slow, but compared to the latency of the disk access itself, they were negligible [76]. However, as workloads evolved, the database community began to branch from traditional OLTP architectures in favor of more specialized databases [142].

In the early 2000's, a need appeared for processing ever-changing, high-velocity data in near real-time. Database researchers began the development of stream data processing systems, query processors that specialized in static queries continuously running on high-velocity, ever-changing data streams [24, 48]. Early stream processing systems such as Aurora/Borealis [14, 13] focus on executing SQL-like operators on an unbounded and continuous stream of input data.

Unlike traditional OLTP databases, streaming databases are *push-based*. This means that data is processed by one operation, and then the result is then pushed to another operation, which is consequently triggered. Operators in streaming systems are connected in a directed acyclic graph called a *dataflow graph*, which maintains a specific order of operations on incoming data items.

The primary optimization goal of early streaming systems was reducing the latency of results. Stream processing was largely used for monitoring applications for which the timeliness of the data

processing is of the utmost importance, where the value of data quickly degrades over time [83]. For financial trading applications, for instance, decisions must be made as quickly as possible, and data that is most recent is most important [101]. To achieve these near real-time speeds, streaming applications were typically run almost entirely in main-memory, avoiding the extreme latency of disk access.

Many stream processing applications required archival storage, but due to the reduction of disk access, system-level support for these queries was limited and ad hoc. Streaming systems were largely not designed with storage in mind, instead relying on either tacked-on storage or a connection to an external data management system. In the case of tacked-on storage, traditional ACID guarantees in general, and atomicity and isolation in particular, were largely overlooked in favor of improved latency. External data management, on the other hand, comes at the cost of performance.

1.3 Advancements in Hardware and OLTP Technologies

Parallel to the development of stream processing, advancements in computer hardware provided the opportunity to significantly improve performance of transaction processing. Main memory became far less expensive, and individual machines fitted with multi-core processors. It became feasible for businesses and researchers to purchase small clusters of machines capable of holding large datasets in memory [144].

Simultaneously, transaction processing workloads became increasingly oriented towards low-latency, high-throughput transaction execution [123]. Similarities in workloads began to emerge. In many cases, only a small subset of the database was accessed in each transaction. Transactions were very short-lived, but frequently focused on writing new information to the database. Additionally, the workloads tended to be highly repetitive, but had a need for variations on the transactions to be repeated at a high frequency - often thousands of times per second.

With the hardware advancements and new workload requirements, it suddenly became both feasible and necessary to hold the entire database in main memory, eschewing disk access for everything but logging. It became clear that entirely new database architectures would be necessary to handle the modern workloads [144]. Mechanisms designed for transaction processing on disk, such as locking, latching, and buffer management, suddenly overwhelmed the meaningful work that transactions were able to accomplish [76]. Main-memory OLTP databases such as H-Store [85], Hekaton [57], and HyPer [89] emerged to tackle the low-latency, high-throughput requirements.

1.4 Motivation: Streaming Workloads with Shared Mutable State

The advent of main-memory OLTP now makes it possible to integrate ACID state management into streaming data management systems. Main-memory OLTP is able to complete transactions in milliseconds, which allows for the near real-time processing required for streaming applications. However, managing shared mutable state in the presence of a dataflow graph comes with additional complexities that are not able to be efficiently handled in either a pure OLTP workload or a pure streaming workload.

There are several common features of workloads that benefit from a hybrid streaming and OLTP system. These include:

- **Low-latency Execution (Streaming)** Like traditional streaming workloads, transactional streaming workloads typically depend on near real-time execution, as their results are highly time-dependent.
- **Time-series Datasets (Streaming)** Streaming systems are particularly well-suited for time-series data. Their push-based nature provides natural maintenance of time-stamp (or equivalent) ordering, a necessity for time-series data.
- **Shared Mutable State (OLTP)** State shared between multiple operations, or accessible by external means, is the primary motivation for adding transactions to stream processing. ACID guarantees in general, and Isolation and Consistency in particular, are missing from traditional streaming systems.
- **Disparate Operations (OLTP)** ACID transactions, by definition, isolate state for the entirety of the execution. However, some workloads benefit from separation of operations into a series of multiple transactions, each triggering the next. This allows for state to be accessed in between the transactions, while still providing an ordered execution to the transaction series.

To demonstrate the need for state isolation in streaming workloads, take the FIX (Financial Information eXchange) trading workload illustrated in Figure 1.1. This is a simplified version of intelligent order routing, based on customer experience at TIBCO StreamBase, Inc. 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, for instance, the history of a particular exchange with the given security. It likely also involves retrieving data from other databases not shown in the figure. Thus, it is modeled as a separate

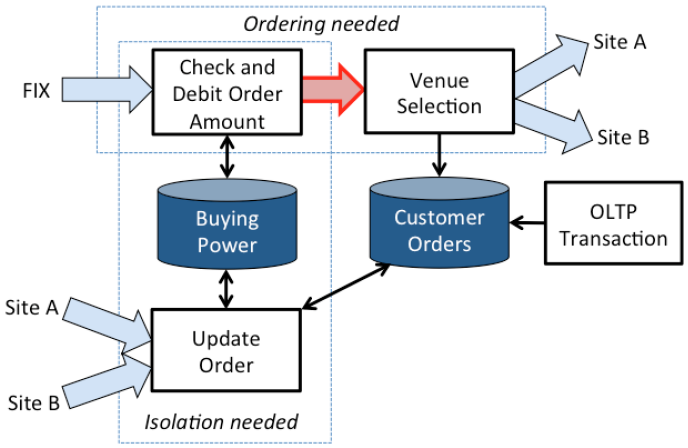


Figure 1.1: FIX Trading Dataflow Example

transaction so that the *Buying Power* database is available to other transactions before the *Venue Selection* transaction is complete.

Venue Selection in this example requires isolation, since it has to make its decision based on consistent state (for instance, 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 customer's 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 a pure stream processing system were used to implement this use case, it would be able to ensure ordering and push-based processing. However, the isolation requirements of the application would not be expressible. If a pure OLTP DBMS were used instead, isolation would be ensured, but the database would not 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. In many cases, a streaming system with integrated transaction processing can improve system performance and simplify the application code.

1.5 Goals and Contributions

This dissertation seeks to answer questions related to the combination of streaming and OLTP. The overall goal is to improve the capability to handle streaming workloads with shared mutable state by defining firm correctness criteria, creating a new data model for developers, and also improving the performance over the state-of the art. Specific contributions include:

Establish Correctness Requirements for Transactional Streaming
The goal is to create a system capable of handling correctness requirements of both streaming and OLTP workloads, so the first step is to define what exactly those requirements are. Each dataflow graph should be able to modify state without interference from external transactions, other dataflow graphs, or itself, with all ordering and durability requirements intact.

Define a Novel Streaming Transaction Model
There is a great deal of nuance in the specifics of how streaming transactions must interact with one another and external transactions. A novel streaming transaction model is necessary to dictate these interactions and ensure the correctness guarantees previously established.

Develop an Architecture with Improved Performance over the State-of-the-Art
By adding state management directly into dataflow graphs, it becomes possible to improve performance for target workloads over either an ad-hoc implementation on any single system, or a combination of two or more disparate systems. The architecture should additionally be able to scale to a small cluster of many nodes while keeping coordination costs to a minimum.

Establish Guidelines for Developing Scalable Workloads
Transactional streaming can bring a great deal of programming flexibility, but excessive communication can limit the ability to scale workloads onto multiple nodes. While the architecture limits coordination costs, intelligent workload planning and design can further improve performance. Here, we establish the guidelines for effective development and design of the target workloads, and two separate scalability strategies depending on the workload properties.

Invent and Implement Unique Use-Cases for Transactional Streaming
With the advent of the new architecture and model, opportunities arise for the creation of unique solutions to modern data problems. One specific use-case, streaming data ingestion, can benefit from not only transactional streaming, but also a generalized architecture for a variety of ingestion situations. This can further be tuned towards ingestion for polystores, a unification of several specialized databases.

The outline of the dissertation is as follows. In Chapter 2, we will provide a more thorough background on existing solutions for high-velocity workloads, and places where those systems fall short of our goals. Chapter 3 will provide a detailed description of what it means to have correct state management in a streaming environment, define a streaming transaction model, and establish a single-node implementation and evaluation. Chapter 4 extends the model into a distributed environment, establishing various coordination costs and design strategies to improve workload scalability. Chapter 5 explores use-cases for transactional streaming, particularly in the realm of streaming data ingestion. Chapter 6 describes preliminary work into the area of workload design, specifically targeting automatic database design for a distributed environment. Chapter 7 delves further into the related work of the field, Chapter 8 describes some avenues of future work, and Chapter 9 relays the conclusions of the dissertation.

Chapter 2

High-Velocity Workloads and Solutions

With the advent of the Internet and more advanced network connectivity in general, a need arose to be able to handle data not only from many different sources, but also with a high degree of haste. Such workloads can be categorized as *high-velocity* workloads, where one of the top priorities is minimizing the latency of executing a task. As technology improved, various techniques were applied to handle high-velocity data, branching into a variety of systems designed for specific use-cases.

There are several different categories of high-velocity data, and different solutions for each. This chapter will give the basics of three such workloads. First, Section 2.1 will describe workloads that benefit from push-based processing, applying a variety of operations onto data as it arrives. Section 2.2 describes main-memory transaction processing, used for modern OLTP workloads that require transactional correctness at very high speeds and a variety of sources. Finally, Section 2.3 describes workloads that require a hybrid of the previous two: push-based processing with transactional state management. This section also explains the shortcomings of the existing solutions, and why a new solution is required.

2.1 Data Stream (Push-based) Processing

One class of applications that arose for data management involves monitoring *data streams* - potentially unbounded sequences of data elements generated in rapid succession [29]. While some data streams have endings, they can potentially continue ad-infinitum as their various sources produce more and more new information. Some examples of data streams include various sensor data [44, 128], financial stock exchanges [162, 171], or social media feeds [97].

There are a number of shared traits among data stream workloads, including but not limited to

[146]:

- **Time-series Ordering -** Most data streams contain some *time-series* component to them, giving them a natural temporal ordering and necessitating an internal ordering to the processing. The time-series may or may not be related to wall-clock time.
- **Comparison Against Historic Values -** Aggregates over periods of time are quite frequent in data stream processing. Some efficient management of historic state is needed, keeping the most useful data items in main memory if possible.
- **Short-lived / Low-latency -** Data streams have a tendency to be extremely dependent on being processed very quickly, as the value of the results can diminish very quickly.
- **High Input Rates -** As with any high-velocity workload, a high input rate is to be expected. Systems designed to handle data streams must be able to manage a high throughput, and should preferably adapt to the load as necessary.
- **Emphasis on Data Movement -** Because data streams are constantly creating new information and low-latency processing is required, there is a requirement to be able to receive new data, send results, and potentially move/copy data items internally at very high rates.
- **Repetitive Computations -** By the nature of having static, continuous queries, the actual processing performed on the data items is extremely repetitive and predictable.

The needs for processing data streams are very different than those of a traditional DBMS [44, 14, 141]. First of all, most monitoring applications receive their data not from humans generating transactions, but instead from external automated sources. They typically require some history of values over a specific period of time, frequently called a *window*. Frequently they are trigger-oriented and alert-based; as new information arrives, it may be necessary to activate some other process. Due to the time-series component, processing the data in order is necessary, but the system must also be aware of out-of-order or missing data. Finally, most data stream processing requires some sort of *real-time* component, implying that the data's value is tied to its lifespan and needs to be processed at the lowest-possible latency [86].

2.1.1 Streaming Data Management Systems

As a result of the differing requirements listed above, Streaming Data Management Systems (SDMS) were created. SDMSs use *continuous queries* to handle data streams [29, 50, 48]. As the name implies, continuous queries involve queries which are constantly active, processing new data as it

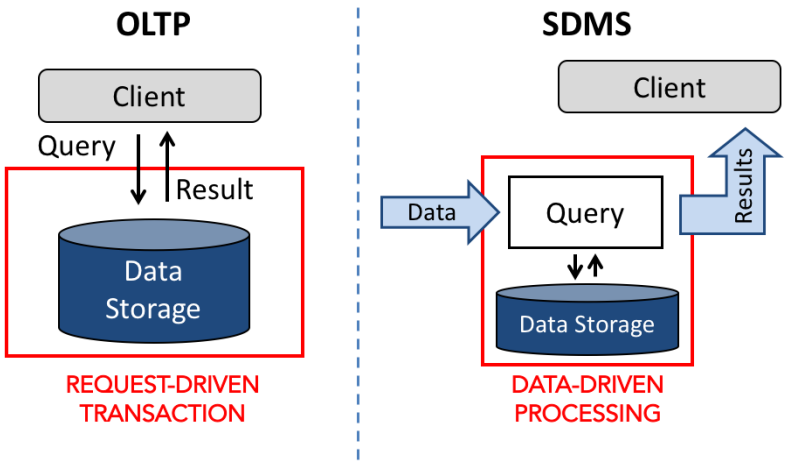


Figure 2.1: High-Level Comparison between OLTP and SDMS Architectures

arrives rather than only running when invoked. This led to a paradigm shift in processing data streams; rather than storing the data in a database and waiting for queries, SDMSs store queries in a query base, activating when new data arrives (Figure 2.1). This is also referred to as *push-based* processing, as opposed to the pull-based model of traditional DBMSs. Because this is driven by data as it arrives at the SDMS, it is also known as a DBMS-active, human-passive (DAHP) model [44, 14].

Several systems were created to handle more complex strings of push-based queries, with each operator (an independent continuous query) pushing its output to the next operator [14]. The data model varies by system, but frequently the operators are arranged in a directed acyclic graph called *dataflow graphs*. Upon an upstream operator completing execution on a data item, a downstream operator is subsequently *triggered*, maintaining the push-based semantics. Eventually, the last operator in the graph executes, pushing the output to any applications waiting for it.

In order to process many data items that were created nearly simultaneously, data that was created within a specific time range are frequently grouped into *batches*. These batches still maintain ordering relative to one another based on their time of creation, but in many SDMSs adopt a new timestamp which has been disconnected from wall-clock time for ease of reference and comparison [25].

2.1.2 Historical State Management in SDMSs

The majority of state management within traditional SDMSs is optimized for storing historical time series data. Because latency is such an important factor in streaming data management, access to the

disk is avoided whenever possible. Storing the time series data exclusively in memory is not always possible, especially considering the technical limitations when these systems were being developed. In many systems, disk storage was used as a data stream overflow from memory, using a circular buffer to coordinate the two and keep the most useful information in memory [14]. Attention was paid to the affect of disk access on the *quality of service* of the system, an evaluation of the accuracy and timeliness of the results [48]. At the time, use-cases for time series data largely have the benefit of not needing to mutate the data items. Additionally, in many cases only a single operator was concerned with the stored data at once, limiting the need for state isolation. As a result, transactional guarantees were mostly ignored for this state.

Additional support was occasionally supplied for more traditional data storage, which could be accessed by the outside world. However, support for this was largely limited and ad-hoc. Transactional support was not considered a priority, and thus not implemented. In many cases, an additional DBMS was employed for the purposes of storing this historical data.

At the crux of the data storage problem lies a simple truth: transactions are computationally expensive. When the original SDMS systems were developed, the technology simply did not exist to allow the throughput and latency needed to make transactional state management viable for managing data streams. That changed in the mid-2000s with the availability of multi-core processors, inexpensive memory, and the advent of main-memory OLTP systems.

2.2 High-Speed Transaction Processing (Main-Memory OLTP)

In addition to data streams, a concurrent need arose to execute transactions at an extremely high velocity. Rather than transactions being manually entered from a single human operator, advances in networks led to the possibility that transactions could come from a variety of sources remotely. Rather than a DBMS expecting tens or hundreds of transactions per second [62], thousands or more requests per second may be received instead.

In the mid 2000s, advancement in computer hardware drastically increased the potential power with which to process transactions. Inexpensive multi-core CPUs made it significantly easier for a single machine to process multiple transactions in parallel. Memory also drastically reduced in price, and it became much easier to create a relatively inexpensive cluster of machines.

Ideal workloads for high-velocity transaction processing tend to include the following properties [123]:

- **Small Footprint -** Each transaction is expected to only access a small portion of the database. Frequently, only a single data item is accessed within these lightweight transactions, and the

transaction can be pointed to that item using indexes and partitioning schemes.

- **Short-lived / Low-latency -** Because each transaction accesses so little of the database, they also have the potential to execute very quickly. If the transaction overhead is reduced to match the short lifespan of the transactions, this can ideally lead to high throughput.
- **High Input Rates -** Transactions are able to come from a variety of different sources, potentially at a very high input rate, and need to be handled at a comparatively high throughput rate.
- **Write-Heavy -** OLTP applications are inherently write-heavy as applications make updates to shared mutable state. Data tends to be added in small batches, often as small as a single data item. These writes need to be isolated from other transactions, and full ACID semantics are required.
- **Repetitive -** While transactions come at high velocity from a variety of sources, they tend to be performing the same basic functionality, only with differing parameters.

Workloads with these properties quickly overload the capacities of traditional OLTP systems that rely on disk access. To handle high-velocity transactions, a new class of OLTP processing became necessary.

2.2.1 Taking Transactions from Disk to Memory

To manage high-velocity transactional workloads, the logical step became to move OLTP systems from the disk directly into main memory. Hardware advances to CPU, memory, and networking made such a leap possible. However, early transaction mechanisms were constructed with the assumption that data was being stored on disk. With the disk access removed, suddenly transactional overheads became a significant portion of the OLTP workload. A different paradigm was necessary to maximize throughput of a main-memory OLTP system.

A study [76] was conducted into the transactional overheads on the CPU instructions in the Shore DBMS, an OLTP system with a traditional architecture. They found three core culprits: buffer pool management, concurrency control scheme, and recovery mechanisms. As shown in Figure 2.2, these three components comprised roughly 88% of the instructions of a sample class of transactions (specifically the NewOrder transaction in the TPC-C benchmark). This means only 12% of the instructions were actually executing the operation itself. Ideally, the overheads could be removed in favor of a more efficient architecture oriented towards main memory execution.

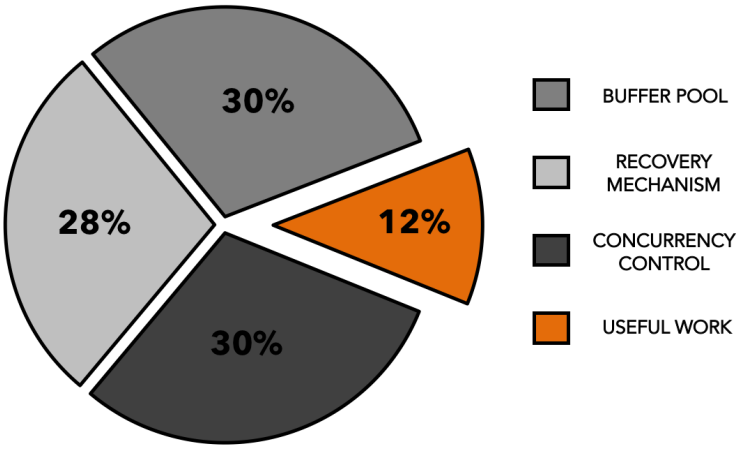


Figure 2.2: Breakdown of CPU instructions in the Shore DBMS (NewOrder transactions in the TPC-C benchmark) [76, 123]

In disk-based OLTP systems, the buffer pool is maintained as a cache for execution, with the DBMS maintaining two copies of the same disk block of data [131]. If the block is changed in the buffer, it is important that it is eventually changed on disk as well, and that future operations execute on the most up-to-date version. In a main-memory database, however, the data is entirely stored on disk, removing the need for two copies.

Similarly, recovery mechanisms in a traditional DBMS architecture are also computationally expensive. This can largely be attributed to the need to maintain a write-ahead log on disk for recovery, separate from the blocks being changed in the buffer [123, 113]. Every update to the state must be logged on disk before a transaction can fully commit. While logging cannot be removed entirely, there are strategies to consolidate the writes to disk, as will be described in the next section.

The execution of concurrent transactions is an extremely common occurrence in traditional DBMSs; however, they come at a cost. Some type of concurrency control is required in order to maintain serializable consistency (i.e. ensuring that all transactions execute as though they were executed one right after the other) [35]. *Pessimistic concurrency control* typically involves a multi-granularity two-phase locking protocol, similar to the one implemented in System R [27]. This strategy involves blocking transactions from executing if it is possible that a non-serializable schedule could exist [112, 123]. *Optimistic concurrency control* schemes (such as multi-version concurrency control [33]) take a different approach, processing transactions with potential conflicts and verifying that serializability was not compromised at the end [112]. In either case, the concurrency control scheme can take up a significant portion of the transaction execution time when disk access is not involved.

2.2.2 Shared-Nothing OLTP (H-Store Architecture)

With the hardware advances, a number of research and commercial systems arose which took a variety of approaches to main-memory OLTP. These include but are not limited to Hekaton (an extension for Microsoft SQL Server) [57], HyPeR [89], and Shore-MT [82], each of which will be discussed in more detail in Chapter 7. This section will focus on the approach of one particular main-memory OLTP system, H-Store [85], as its architecture is a highly relevant foundation of our work. H-Store was chosen because 1) it is the research parallel/foundation of VoltDB [12] (a modern enterprise main-memory database), 2) its use of small, lightweight transactions mirror streaming database needs, and 3) its distribution structure is ideal for linear scalability of transactional streaming.

H-Store uses main memory as its primary storage location [123, 85]. As such, rows are not stored in blocks as they would be on disk, but instead are accessible directly. H-Store does feature anti-caching, a method of storing colder data on disk when it is not needed, but even anti-caching only stores each data item in a single location to remove the need to synchronize multiple copies in buffer management [55].

Because state is primarily (or even exclusively) stored in memory, durability is extremely important. Because of the assumption that workloads are extremely repetitive, the majority of processing in H-Store is contained within SQL-based stored procedures, which use input parameters to specify the transactions [108]. These stored procedures are guaranteed to be deterministic (i.e. they will produce the exact same output given the same input, if executed in the same order). Thus, rather than logging each state change, H-Store uses command logging to create an ordered list of stored procedure executions that are stored on disk. This log can then be rerun in the event of a failure in order to recreate the state of the database. Transactions are fully committed to the system in batches, allowing the log to write many commands to disk at once (asynchronously of the actual transaction executions). Additionally, H-Store takes a periodic snapshot of the database which is also stored on disk. These allow the command logs to be truncated to the point of time at which the snapshot was taken.

The state in memory is divided into partitions across one or multiple nodes. H-Store employs a lightweight, pessimistic concurrency control technique in which state access is locked at the *partition* level (rather than the table or tuple level) [126]. Each memory partition is accessible only by a single thread, each of which executes transactions serially (i.e. without any concurrency). Due to their small footprint, transactions are typically expected to access only a single partition. They are queued at that location and execute one at a time, guaranteeing a serial schedule for single-parititon transactions.

While it is expected that most transactions will execute on a single partition, H-Store does allow for transactions to access multiple partitions in a distributed transaction. In this case, multiple partition locks will need to be obtained at once, and the transaction executions will be coordinated via two-phase commit protocol.

Shared-nothing architectures have been shown to be very effective at executing small-footprint transactions across a distributed system while maintaining roughly linear scalability as new nodes are added [124]. Because of the relatively low transaction overhead and the almost complete removal of disk latency, these transactions are able to execute on the order of milliseconds. More details about H-Store's architecture will be discussed as they become relevant to the development of our transactional streaming system.

2.3 The Need for Transactional Streaming

When comparing the two types of high-velocity workloads described here, there are some obvious similarities. There is the obvious expectation of high input rates, but also several other common features. Both data stream processing and main-memory OLTP workloads involve many smaller operations as requests arrive, as opposed to the long analytical queries of OLAP workloads. Both require extremely low latency, as timeliness is a factor in both cases. The constant copying of data throughout a stream processing system could be thought of as a write-heavy workload, similar to high-velocity transactions. Perhaps most importantly, both workloads tend to be extremely repetitive, a factor which can be exploited for better performance.

The key difference on each side is transparent. Data stream processing primarily concerns itself with time series, and with that comes a need for time series ordering throughout its execution. SDMS systems also utilize push-based processing through a series of operators in a dataflow graph, an element missing from OLTP execution. On the other hand, main-memory OLTP is primarily concerned with ACID execution at a high throughput and low latency, an element that has never been fully realized in a SDMS. Data isolation and strong recovery guarantees are both elements that SDMS systems tend to lack [110].

There are many workloads that require both push-based processing and ACID management of shared mutable state. Aside from the financial trading example given in Section 1.4, there are several other use-cases, many of which involve streaming data ingestion. These include:

- **Internet of Things (IoT) Monitoring and Archiving -** As network connectivity spans to an enormous variety of devices and sensors, so too does the need to unify the incoming data and process them in real-time. Any shared mutable state between sensors or devices requires

isolation during transformation to avoid damaging results.

- **Streaming ETL -** The ETL (or Extract Transform Load) process of ingesting data into a data warehouse has long been considered most efficient as a bulk-load batch process. Transforming it into a streaming process can greatly reduce the latency of results, but requires transactional state management.

- **Data Ingestion for Polystores -** A polystore is a system built on top of a variety of disparate database systems, utilizing each one's specialized architecture to solve corresponding data problems [61]. Such a system includes new challenges in data ingestion, involving the normalization, cataloging, and routing of new data as it arrives.

Each of these workloads will be further discussed in Chapter 5, along with the specifics of transactional streaming solutions.

The functionality of a transactional streaming system can potentially be attained by connecting multiple systems together (for instance a streaming database like Esper [5] and a main-memory OLTP database like HyPeR [89]), but this solution comes at a massive cost to performance. As we will show in future chapters, the use of multiple systems to accomplish the goal of storing time series data transactionally can severely limit performance, a major problem in a system oriented towards low-latency operation.

Rather than fitting a square peg into a round hole, a better strategy is to build upon both streaming data management systems and main-memory OLTP. Utilizing the lessons of both branches, we seek to create a hybrid *transactional streaming* system capable of handling data streams with shared mutable state, as well as fulfilling the capabilities of both a SDMS and a traditional OLTP system.

Chapter 3

Stream Processing with Shared Mutable State

While hardware advancements make it theoretically possible to add transactional state management to push-based stream processing systems, accomplishing this is another feat entirely. This chapter concerns the requirements and definitions of a transactional streaming system. As a system primarily concerned with the correctness of shared, mutable state, it is important to first clarify what it means for this state to be correct (Section 3.1). Those specifications must then be integrated into a computational model, including a novel transaction model (Section 3.2). The transaction model is then implemented into a main-memory streaming OLTP hybrid system known as S-Store. The details of the single-node implementation are discussed in Section 3.3. Finally, the performance of that implementation is compared to existing systems on hybrid streaming OLTP workloads in Section 3.4.

3.1 Correctness Guarantees

Stream processing with shared mutable state is all about ensuring consistent state in the presence of external transactions. The first step to accomplishing this is to define what is considered to be consistent in a transactional streaming environment. These rules for consistency are narrowed down to three *correctness guarantees*, each inherited from either the OLTP or streaming lineage.

As already discussed, transaction-processing systems normally provide ACID guarantees. These guarantees broadly protect against data corruption of two kinds: (i) interference of concurrent transactions, and (ii) transaction failures. Consistency and Isolation primarily address interference, while Atomicity and Durability address failures. While they do not feature ACID state management,

most stream processing engines protect against failure-induced data inconsistencies by incorporating failure-recovery facilities. However, any streaming computation that shares mutable data with other computations (e.g., a separate streaming dataflow graph) must guard against interference from those computations as in standard OLTP.

To protect data correctness, there are additional requirements from stream processing systems that must be considered. First, a transaction execution must conform to some logical order specified by the user. The scheduler should be free to produce a schedule that interleaves transactions in a variety of ways, but the results must be equivalent to the specified logical order. Secondly, in streaming systems, failures may lead to lost or duplicated tuples if operations are incorrectly repeated on recovery. Thus, streaming systems typically strive to provide exactly-once semantics as part of their fault-tolerance mechanisms. In order to correctly handle hybrid workloads, the system must provide efficient scheduling and recovery mechanisms that maintain three complementary correctness guarantees that are needed by both streaming and transactional processing.

3.1.1 ACID Guarantees

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

3.1.2 Ordered Execution Guarantees

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

correctness for complex dataflow graphs as a whole.

Traditional ACID-based models do not provide any order-related guarantees aside from timestamp serializability. In fact, transactions can be executed in any order as long as the result is equivalent to a serial schedule. Therefore, a transactional streaming system must provide an additional correctness guarantee that ensures that every transaction schedule meets the following two constraints:

Batch Order Constraint For a given streaming transaction T, atomic batches of an input stream S must be processed in order.

Dataflow Order Constraint For a given atomic batch of stream S that is input to a dataflow graph G, transactions that constitute G must be processed in a valid topological order of G.

For coarser-grained isolation, nested transactions may also be defined as part of a dataflow graph, which may introduce additional ordering constraints (further explored in Section 3.2.3). The system must take all of these constraints into account in order to create correct execution schedules.

3.1.3 Exactly-once Processing Guarantees

Failures in streaming applications may lead to lost state, and recovering from failures typically involves replicating and replaying streaming state. If not applied with care, replaying the streaming operations may lead to redundant executions and duplicated state. To avoid these problems, streaming systems provide fault tolerance mechanisms that will ensure "exactly-once" semantics. Note that exactly-once may refer either to external delivery of streaming results, or to processing of streams within the system. The former typically implies the latter, but the not necessarily the other way around. This work primarily focuses on exactly-once processing rather than delivery, as that is more directly relevant in terms of database state management.

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

that redoing a committed transaction does not lead to redundant invocations on others that depend on it.

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

Note that executing a streaming transaction may have an external side effect other than modifying the database state (e.g., delivering an output tuple to a sink that is external to the system). It is possible that such a side effect may get executed multiple times during recovery. Thus, the exactly-once processing guarantee is similar to systems such as Spark Streaming[168] in that it applies only to state that is internal to the transactional streaming system.

3.2 Computational Model

The transactional streaming model adopts well-accepted notions of OLTP and stream processing, and fuses them into one coherent model. All of the traditional notions of transactions, as discussed in previous sections, apply here as well.

Both OLTP and streaming transactions can share state and at the same time produce correct results. Three different kinds of state are supported: (i) public tables, (ii) windows, and (iii) streams. Furthermore, a distinction is made 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 [160]. 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.

3.2.1 Streaming Transactions & Dataflow Graphs

Data Model

Our stream data model is very similar to many of the stream processing systems of a decade ago [14, 24, 48]. A *stream* is an ordered, unbounded collection of tuples. Tuples have a timestamp [24] or, more generally, a batch-id [37, 81] 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) [24], or more recently, by Spark Streaming (group tuples into "mini batches" of small time intervals) [168].

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 [24, 168]) 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, 3, 18, 115, 118, 151, 168]. 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 3.2.3), 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 [37, 71], 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 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 3.1, 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 3.1) and *interior transactions* (others, e.g., T_2 in Figure 3.1). 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 3.1 also illustrates the different kinds of state accessed and shared by different transaction

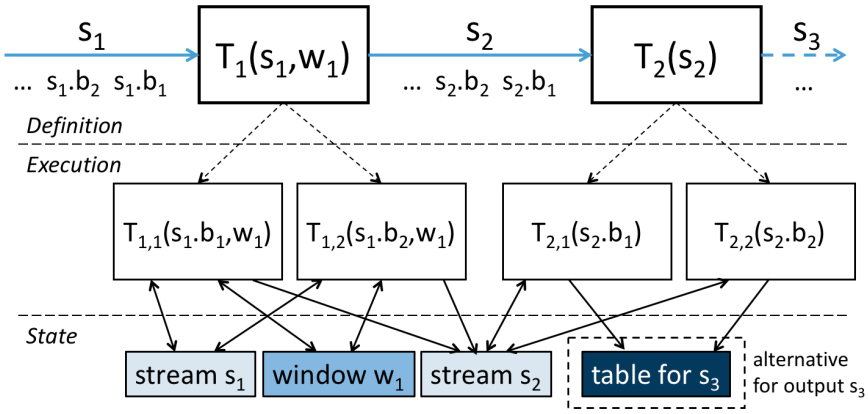


Figure 3.1: Transaction Executions in a Dataflow Graph

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 3.1): (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.

3.2.2 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 3.1. 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 3.1. 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 3.1, 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

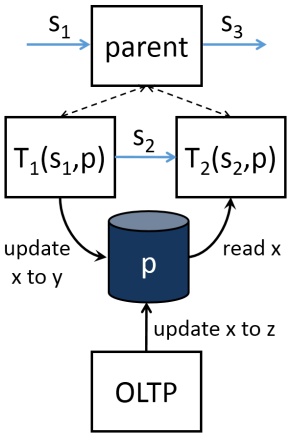


Figure 3.2: Nested Transaction Example

have:

1. *Dataflow 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. *Batch 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.

3.2.3 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 3.2. 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.

More generally, an S-Store nested transaction consists of two or more stored procedures with a partial order defined among them [137]. 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 straightforward 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.

3.2.4 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 3.1. 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 3.1). 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 3.1). This is similar to other exactly-once processing systems such as Spark Streaming [168].

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 3.2.2 and 3.2.3), 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).

3.3 S-Store Architecture

This section introduces **S-Store**, an implementation of the model described above. S-Store is built on top of the H-Store main-memory OLTP system [85]. This allows it to inherit H-Store's support for high-throughput transaction processing, thereby eliminating the need to replicate this complex functionality. S-Store also receives associated functionality that will be important for streaming OLTP applications, including indexing, main-memory operation, distribution mechanisms, 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. Figure 3.3 displays the single-node architecture S-Store, with streaming additions noted in bold font on the right-hand side. It is important to note that the architectural features that have been added to H-Store are conceptually applicable to any main-memory OLTP system.

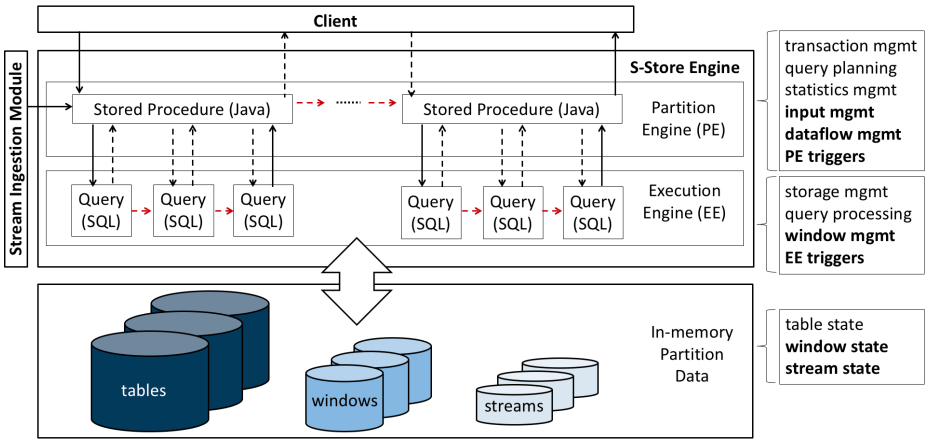


Figure 3.3: S-Store Architecture

3.3.1 H-Store Architecture Details

As discussed previously in Section 2.2.2, H-Store is an open-source, main-memory OLTP engine that was developed at Brown and MIT [85], and formed the basis for the design of the VoltDB NewSQL database system [12]. Here we will describe some implementation details relevant to the creation of S-Store.

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 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 [124], where a site corresponds to a CPU core. These sites are divided across a number of nodes, which can be located anywhere across the clusterThe available DRAM for a node is divided equally among the partitions it contains, 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.

In this chapter, we limit the scope of implementation to only a single site on a single node to remove the complications of distribution. Distribution of transactional streaming will be discussed in Chapter 4.

H-Store provides recovery through a checkpointing and command-logging mechanism [108]. 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.

3.3.2 Dataflow Graphs

Dataflow graphs in S-Store are composed of stored procedures able to trigger one another in a directed acyclic graph. As new data arrives, it is placed on an input stream which activates the transaction execution of a *border stored procedure*, meaning a stored procedure at the start of the dataflow graph. Dataflow stored procedures take stream data and a unique batch-id as input parameters, generating transaction executions identified by batch-id. Each dataflow stored procedure is assigned an *output stream*. Upon completion, a dataflow stored procedure places any output data onto its output stream, to be consumed by either a downstream stored procedure or an output client. If there is a downstream stored procedure in the graph, the output data then triggers the next transaction execution upon commit.

3.3.3 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 3.3, 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 3.3, the horizontal arrows between SQL queries inside the EE layer denote EE triggers.

3.3.4 Streams

S-Store implements a stream as a time-varying H-Store table, which makes stream state 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 3.3.3). When triggering streaming transactions, the current stream table serves as input for the corresponding downstream streaming transaction.

Unlike tables, data in streams are able to be moved from one node to another, in order to trigger downstream stored procedures across nodes. When an upstream stored procedure triggers a downstream one, the transaction call includes the corresponding stream data attached as parameters. Thus, the stream data is passed as message data to the downstream scheduler, removing the need to later pull that data from the original node during the transaction execution. The stream data is also maintained in persistent memory on the transaction's node of origin for recovery purposes.

3.3.5 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 3.1.1, special scoping rules are enforced for window state. A window table may only written by 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.

3.3.6 Single-Node Streaming Scheduler

Being an OLTP database that implements the traditional ACID model, the H-Store scheduler can execute transaction requests in any serializable order. On a single H-Store partition, transactions run in a serial fashion by design [85]. H-Store serves transaction requests from its clients in a FIFO manner by default.

As we discussed in Section 3.2.2, 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 3.2.2. Additionally, as discussed in Section 3.2.3, 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, the single-node scheduler implementation admits only one of them. Because there is no opportunity for parallelism in the single-node case, there is no discernible disadvantage to simply forcing the transactions to execute in a FIFO schedule. This ensures both batch and dataflow ordering and serializability with OLTP transactions, while

also being an extremely low-overhead solution. In Chapter 4, we will discuss more complicated scheduling solutions for a distributed environment.

3.3.7 Recovery Mechanisms

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, the system 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, the system turns PE triggers back on. At that point, it also checks 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). S-Store uses a technique similar to *upstream backup* [79] 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, the system 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, PE triggers do not need to be turned off during weak recovery. In fact, the replay relies 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.

3.4 Performance Comparisons (Single-Node)

The first experiments focus on S-Store's performance as a single-node system, particularly in the context of the three guarantees described in Section 3.1. In modern state-of-the-art systems, it is challenging to provide all three processing guarantees. More specifically, OLTP systems are usually able to process ACID transactions with high performance, but have no concept of dataflow graphs, and thus no inherent support for ordering or exactly-once processing. In contrast, stream processing systems are able to provide dataflow ordering and exactly-once processing, but do not support ACID transactions. Thus, in both cases, achieving all three guarantees with high performance is a major challenge.

3.4.1 Performance vs OLTP and Streaming

To test S-Store's performance in comparison to current state-of-the-art, we created a simple leaderboard maintenance benchmark. This benchmark mimics a singing competition in which users vote for their favorite contestants, and periodically, the lowest contestant is removed until a winner is selected. As shown in Figure 3.4, the benchmark's dataflow graph is composed of three stored procedures that each access shared table state, and thus requires data isolation (i.e., a nested transaction) across all three. For the purposes of simplifying comparison across systems, we considered a batch to be a single vote, and we record our throughput numbers in terms of "input batches per second."

The leaderboard maintenance benchmark requires all three of S-Store's processing guarantees to be executed correctly. We first compared S-Store's performance to its OLTP predecessor, H-Store. As an OLTP system, by default H-Store only provides the first guarantee, ACID, and thus maintains an impressive throughput (over 5000 input batches per second, as shown in the first row of Figure 3.5). However, the results it provides are incorrect; a wrong candidate may win the contest since

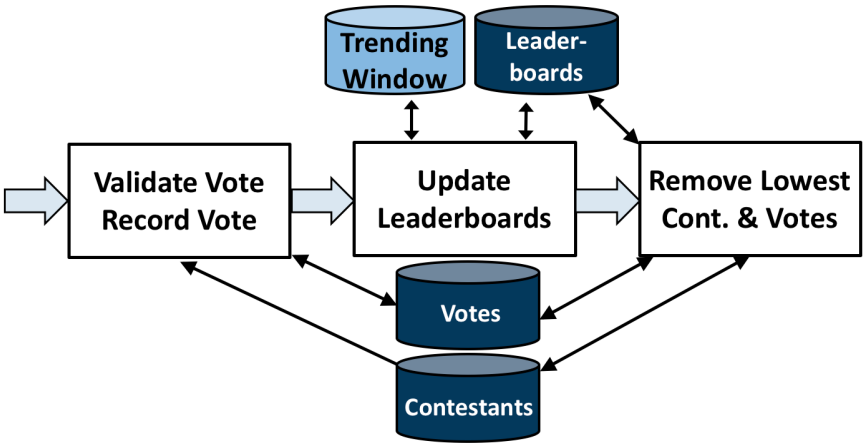


Figure 3.4: Leaderboard Maintenance Benchmark Dataflow

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

Figure 3.5: Voter Leaderboard Experiment Results (Relative to Guarantees)

votes may be processed in a different order than the one that is required by the benchmark. For H-Store to provide correct results, the ordering guarantee must also be provided.

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

Both single-node streaming engines (e.g., Esper [5]) and distributed stream processing engines (e.g., Storm [151]) also struggle to provide all three processing guarantees. In the case of streaming engines, dataflow graphs are considered fundamental structures, and the ordering guarantee is

provided. Exactly-once processing can also be added to many systems possibly with some loss in performance (e.g., Storm with Trident [11]). However, ACID transactions are not integrated into streaming systems. Instead, they must use an additional OLTP database to store and share the mutable state consistenly. For our experiments, we used VoltDB [12] (the commercial version of H-Store) to provide this functionality to Esper and Storm.

Similarly to H-Store, providing all three processing guarantees decimates throughput. To provide both ordering and ACID, the streaming systems must submit requests to the OLTP database and wait for the response to move on. Even with a main-memory OLTP system like VoltDB, this additional communication takes time and prevents the stream system from performing meaningful work in the mean time. As shown in Figure 3.5, both Esper and Storm with Trident were only able to manage about 600 input batches per second, when providing ACID guarantees through VoltDB.

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

3.4.2 Trigger Performance

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 these experiments, command-logging was disabled to emphasize the feature being measured.

EE Trigger Comparison

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 3.6). 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 3.6 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 3.7 shows how maximum throughput varies with the number of EE triggers. S-Store

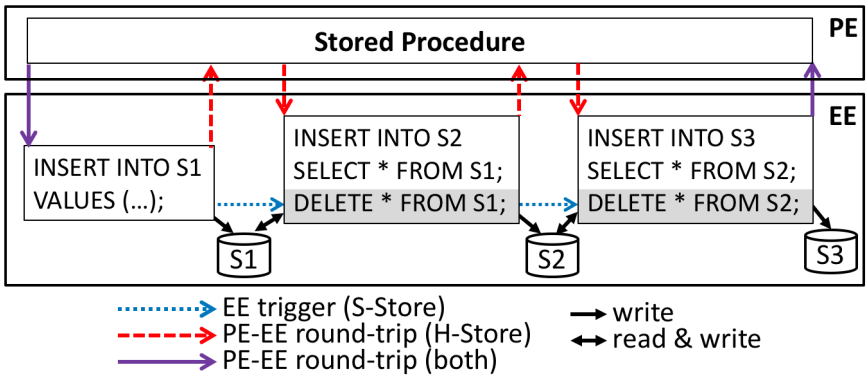


Figure 3.6: EE Trigger Micro-Benchmark

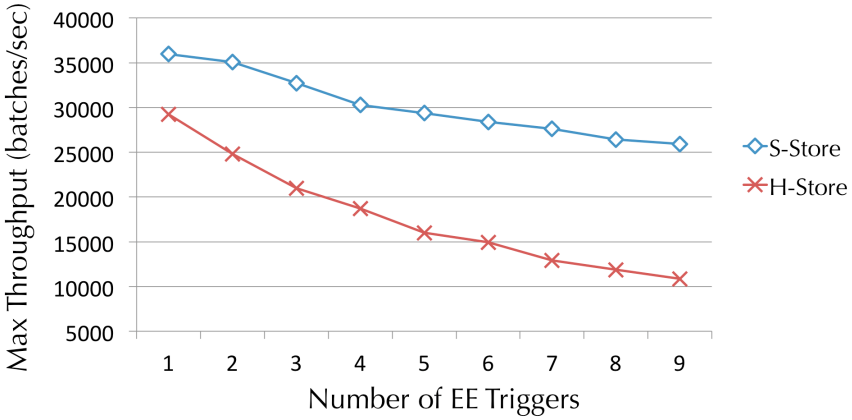


Figure 3.7: EE Trigger Result

outperforms H-Store in all cases, and its relative performance further increases with the number of EE triggers, reaching up to a factor of 2.5x for 9 triggers. This trend continues as more EE triggers are added.

PE Trigger Comparison

A second experiment was conducted to compare 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 3.8, 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 asyn-

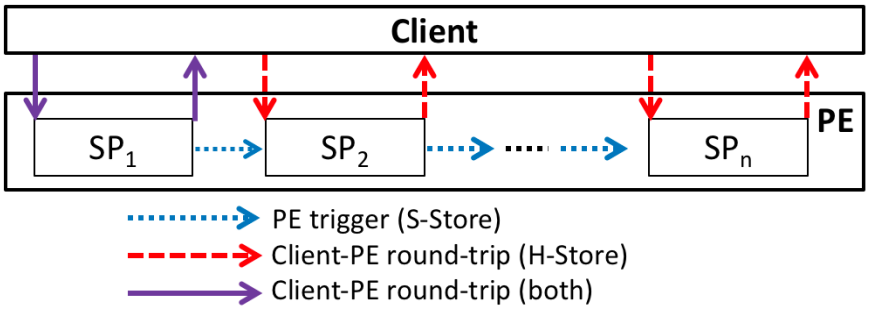


Figure 3.8: PE Trigger Micro-Benchmark

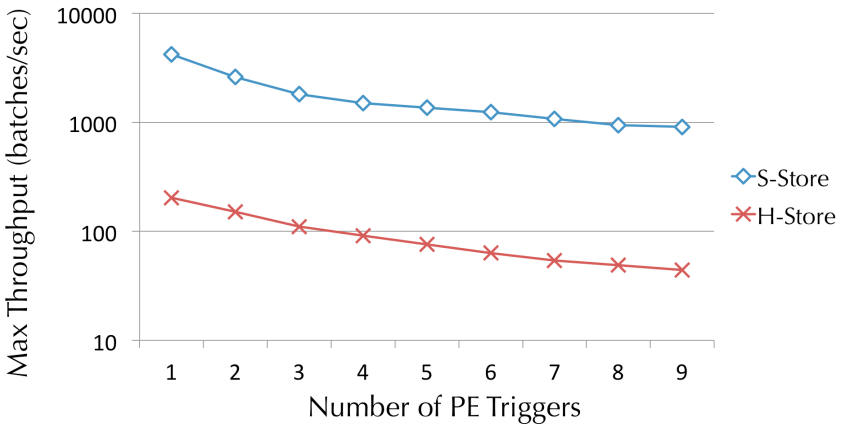


Figure 3.9: PE Trigger Result

chronously. 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 3.4.1). 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 3.9 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.

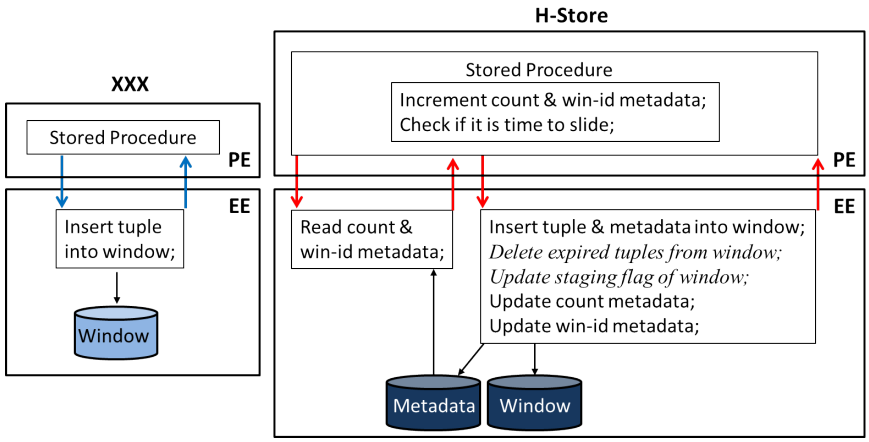
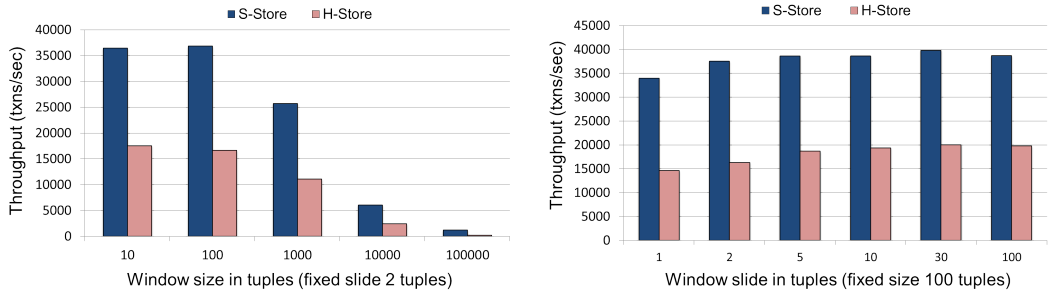


Figure 3.10: Window Micro-benchmarks



(a) Window Size Comparison

(b) Window Slide Comparison

Figure 3.11: Windowing Performance

3.4.3 Windowing Comparison

In this micro-benchmark, we compare native windowing support provided by S-Store within its EE to a comparable H-Store implementation without such built-in windowing support. As mentioned in Section 3.3.5, S-Store employs a tuple-wise ordering strategy and flags new tuples for ”staging” as they arrive, keeping them inactive until the window slides. The most fair strategy for H-Store is to use a similar technique, in which separate columns are maintained to indicate the absolute ordering of tuples, as well as a flag denoting which tuples are active within the window. If enough tuples have arrived to slide the window, then ”staged” tuple flags are removed and expired tuples are deleted.

Figure 3.10 depicts the micro-benchmark we used in this experiment, which simply inserts new tuples into a tuple-based sliding window and maintains the window as tuples continue to arrive. As seen on the left-hand side of the figure, S-Store can implement this micro-benchmark simply by defining the window with its desired parameters and issuing insertion queries over it. An equivalent

implementation in H-Store (shown on the right-hand side of Figure 3.10) would require the maintenance of both a window and a metadata table, with a two-staged stored procedure to manage the window state using a combination of SQL queries and Java logic.

Figure 3.11(a) shows the transaction throughput achieved by S-Store and H-Store as window size varies, while Figure 3.11(b) does the same as window slide varies. Our native windowing implementation grants us roughly a 2x throughput gain over the naive implementation of H-Store. This gain primarily comes from maintaining window statistics (total window size, number of tuples in staging, etc.) within the table metadata, allowing window slides and tuple expiration to happen automatically when necessary. It is also important to note that variations in window size provide a much larger contribution to the performance difference than window slide variations. This is because slide frequency only affects the frequency of two SQL queries associated with window management (shown in italic in Figure 3.10).

3.4.4 Recovery Performance

As described earlier in Sections 3.2.4 and 3.3.7, 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 3.4.2 (Figure 3.8), 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 3.12, 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 3.13, 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

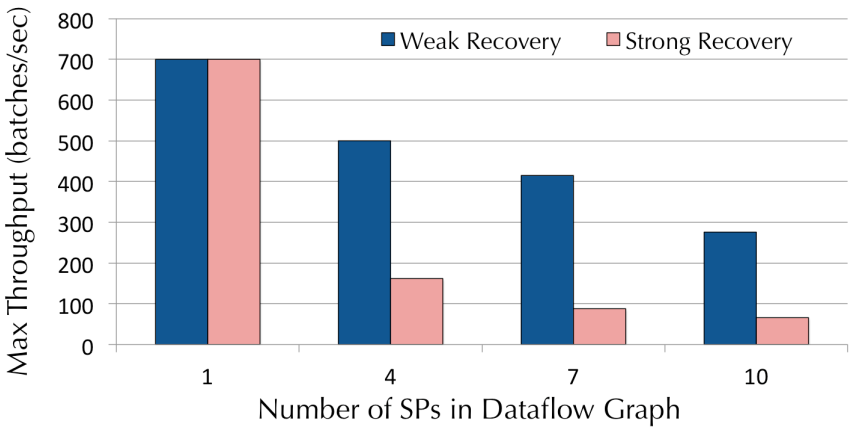


Figure 3.12: Logging Performance (Weak vs Strong)

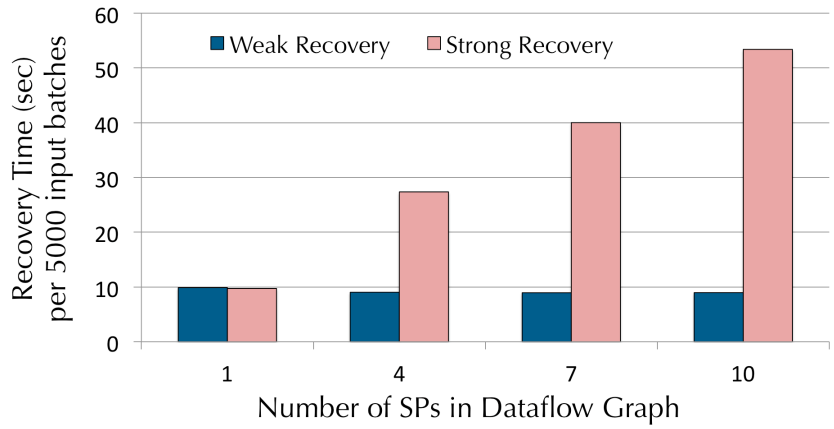


Figure 3.13: Recovery Performance (Weak vs Strong)

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.

3.5 Conclusions

A streaming transaction model integrates correct management of shared mutable state into a push-based dataflow environment. In this chapter, we have defined what "correct state management" means in this context, specifically addressing ACID transactional needs, dataflow and batch ordering, and exactly-once processing. Creating a thorough transaction model requires the consideration of all three correctness guarantees, while also allowing for interaction with separate OLTP transactions.

This chapter also 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 database management system in a straight-forward way.

We have shown performance benefits in even the single-node case. 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. This is primarily due to the minimization of communication requirements, preventing the need to delay processing to maintain one or more correctness guarantees.

Chapter 4

Streaming Transactions at Scale

While transactional streaming is a solution to some of the shortcomings of both stream processing and OLTP, it inherits the scalability challenges of both worlds as well. Providing full correctness guarantees in a distributed setting can be very expensive. A variety of coordination costs can limit performance, including distributed transactions, scheduling coordination costs, and data stream movement across nodes. Section 4.1.1 gives an overview of a shared-nothing architecture, and why it is a strong choice for transactional streaming. Database design can differ significantly from traditional OLTP workloads, however. The better tailored a database design is to a specific workload, the more effective the design will be. Sections 4.2 and 4.3 detail the specific design decisions recommended for workloads that are data-parallel and pipeline-parallel, respectively.

The distributed architecture for transactional streaming takes the features of shared-nothing systems and repurposes them for streaming workloads. Section 4.4 describes how distributed transaction coordination in OLTP can be utilized to maintain stream ordering. Section 4.5 explains our efficient implementation of streaming data movement across nodes. Finally Section 4.6 compares system performance of the various scaling techniques described in this chapter.

4.1 Sharding State and Processing (OLTP vs Streaming)

The goal of distribution in both OLTP and streaming workloads is very similar: maximize load balancing across a cluster while minimizing the coordination costs. However, the specifics of the two workload types are very different, and the mechanisms of achieving that balance differ as well. This section first explains the details of the shared-nothing distribution model (Section 4.1.1), then explains how that model is typically utilized for OLTP workloads (Section 4.1.2). We then discuss how to add batching, a fundamental component of streaming, into the shared-nothing model

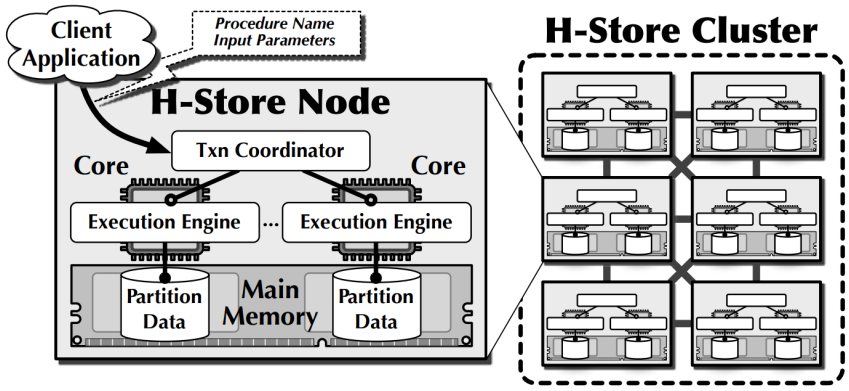


Figure 4.1: H-Store/Shared-Nothing Architecture (from [123])

(Section 4.1.3).

4.1.1 Shared-Nothing Distribution Model (H-Store)

As described in Chapter 3, our implementation of transactional streaming inherits its base distribution model from main-memory OLTP, specifically a shared-nothing architecture (illustrated in Figure 4.1). This architecture was chosen largely on the basis of limiting coordination costs and allowing the potential for a high-throughput transactional system. Here, we provide more details about the shared-nothing architecture [126, 123], particularly as it relates to H-Store (our base system of choice) and an implementation of distributed transactional streaming.

A group of *nodes* in a shared-nothing architecture is defined as a cluster. Nodes are independent units within the cluster with their own complete set of resources, including one or multiple CPU cores and an exclusive block of memory. A single machine in a shared-nothing cluster contains a single node by default, but can contain multiple independent nodes if desired.

Each node contains a number of *partitions*, a disjoint subset of the database [75, 172]. Each partition contains a state component, where database state is stored in-memory. Unless replication or high-availability is used, the state within a partition is unique. A partition is exclusively accessed by a single thread (ideally mapped to a unique CPU core). The single-threaded access to state is the Execution Engine from Chapter 3, and is what provides the data isolation to the partition's state.

Each node contains a transaction coordinator responsible for communicating with other nodes in the cluster, as well as managing the local transactions of the node. In H-Store, the transaction coordinator is decentralized, and all coordinators collectively maintain serializable transaction ordering and manage a queue of transaction executions [123]. Coordinators communicate with one another via commit protocol messages for distributed transactions and to perform system admin-

istrative functions. The transaction coordinator lives in the Partition Engine described in Chapter 3.

Ideally, transactions only access a single partition, and thus do not require coordination across the cluster. However, state from multiple partitions can be accessed together in a distributed transaction. In this case, each accessed partition must be locked simultaneously, as each partition's thread manages its own state access. Distributed transactions use two-phase commit (2PC) to ensure atomic coordination. While the redundant communication of 2PC is expensive in terms of latency, it assures that any coordination happens transactionally. 2PC is necessary not just for distributed transactions, but also for redirecting transactions to another location once they are scheduled and other communications across transaction coordinators.

4.1.2 Database Design for OLTP

Workloads suited for shared-nothing, main-memory OLTP tend to feature transactions that have a very *small footprint*, meaning that they access a very small quantity of state and are very short-lived [123]. If multiple data items are accessed in a single transactions, there are two possibilities: either they are located on the same partition (in which case the transaction remains single-sited), or they are on separate partitions and require a distributed transaction. The more partitions that are accessed in a single transaction, the higher the coordination cost (with a major step upwards for the first additional partition).

Sharding state based on a single key to minimize distributed transactions is easy, provided that the key represents how the data will typically be accessed. If a transaction accesses many different types of state simultaneously, however, it becomes increasingly difficult to shard that state such that it can all be co-located onto the same partition.

By the same token, state access on a partition requires computing resources. In a shared-nothing system, reading or writing a piece of state is executed by the processing thread attached to the same partition and node where the state resides. Data is typically not uniformly accessed, resulting in skew. OLTP systems take skew into account during the database design process [124]. "Hot" tuples that are accessed frequently tend to be isolated to allow for less-constricted access, while "cold" tuples can be grouped together. The goal of this exercise is to balance the processing load as evenly as possible across the distributed system, and the processing load is directly related to state access.

4.1.3 Batching with Minimal Coordination

Streaming systems (including transactional streaming) frequently involve batch processing, which is antithetical to the small footprint of data access that shared-nothing systems prefer. Batch pro-

cessing is used to group data items that arrive during an atomic unit of time, and enables that data to be processed at a higher-throughput than if they were handled individually. The latter point is particularly true when transactions are introduced, as each transaction comes with administration overhead.

While small-footprint transactions are easier to manage, larger transactions that touch more state can continue to be efficient in a shared-nothing system, so long as all of the accessed state is in the same location. When sharding state for batch processing, it is important that the state be partitioned in a way that it is easy to access together.

Say, for instance, the developer knows that a frequent transaction accesses some quantity of tuples from a single table and no other shared state. In the simplest case, the entirety of that table could be all located on the same partition. This would guarantee that all tuples that the transaction requires are in the same place, and the transaction will be single-sited. Obviously this may not always be possible, as a table may be too large or accessed too frequently to exist entirely on one partition. Even if the table must be divided among multiple partitions, however, the number of partitions could be kept as low as possible, and those partitions could all be located on the same node (or a limited number of nodes). Either of those decisions can reduce the coordination cost of batch processing, as it limits the number of communications for each transaction.

4.2 Database Design: Data-Parallelizable Workloads

In broad terms, workloads in transactional streaming can be broken down into two categories: those that are data-parallelizable and those that are pipeline-parallelizable. In the next two sections, we explore the differences between these scalability approaches, and the properties of the workloads that warrant their use. In this section, we describe data parallelization, typically associated with more traditional streaming workloads.

4.2.1 Properties of Workloads

A large number of traditional streaming workloads share a number of common elements, which tend to yield a particular approach to parallelization. Specifically, modern streaming workloads tend to be partitionable on a single attribute, such as device group or location. For instance, in many IoT scenarios, sensors are grouped together based on whether their values are relevant to one another. If two groups are disparate, then it is easy to partition their processing onto separate machines. With this partitioning scheme, it is possible to achieve near-linear scalability with the number of nodes in the cluster, as there is very little communication needed for each machine to process its portion.

This is sometimes known as *data parallelization*.

A similar strategy can be taken towards certain transactional streaming workloads, with the key difference being the need to partition shared mutable state in addition to the processing itself. There are specific properties that a workload must contain in order to fulfill the requirements of data parallelization. These include:

Majority of Workload Partitionable on a Single Key

The most important property for data parallelization in transactional streaming is that the majority of the shared mutable state accessed be partitionable on a single key. Without this property, the workload will require an abundance of communication between nodes, and the coordination costs will outweigh the data parallelism. We have discussed in previous chapters how expensive distributed transactions can be to the overall performance of the system, so these become the major bottleneck in such cases.

Take for instance a workload in which a group of sensors ”A” are located at several locations, and can be easily be partitioned by location. The values from these sensors can easily be stored in main memory, and assigned to specific nodes based on that partitioning key. However, suppose that the workload frequently called for data from these ”A” sensors to be compared with another group of ”B” sensors, and that these ”B” sensors are partitionable on a completely separate group of locations. Because the two groups have different natural partitioning schemes and there is frequent interaction between the groups, it is impossible to co-locate the ”A” sensor values with the ”B” sensor values they are compared against.

The one exception to all accessed state being partitionable on a single key is if the outlying state is replicable across nodes. This is particularly true for state in the workload that is “read-mostly.” If the same state is replicated to all nodes which require it, then cross-node communication can still be minimized.

Priority on Scalability

The reason that data parallelization is used so frequently by modern streaming systems is that it scales extremely well to the cluster environment it is given. Typically workloads that can be data parallelized will scale linearly with the number of nodes available.

Large, Infrequent Batches

The larger an individual batch is, the more benefit can be gained by distributing the work across as many nodes as are available. Similarly, larger batches tend to be sent with less frequency, as more of the time-series data is consolidated into fewer batches. This allows for more of the data to be distributed across nodes in bulk, minimizing communication costs.

Few Queries Across All Partitions

Because each type of state is parallelized across many partitions in these workloads, it becomes more costly to run a query across a range of data not limited on the partitioning keys. Such a query is likely to require a distributed transaction that reaches across many (if not all) nodes and partitions.

4.2.2 Example: Linear Road

A number of streaming or MapReduce workloads fit the description of a data-parallelizable workload, including various IoT sensor collections, many machine learning algorithms, and the infamous WordCount workload. For our purposes, we use the canonical stream processing benchmark, Linear Road [26].

Linear Road simulates a highway toll system for expressways (X-Ways) of a metropolitan area. Location data is collected for each car on the monitored expressways every thirty seconds, with tolls, potential traffic alerts, and other information collected and evaluated with each new data point. This data is formatted as tuples in a time-series, with each tuple containing a type-id that indicates which dataflow of continuous queries to evaluate. The majority of the dataflow queries are expressway-dependent, meaning that only one expressway need be queried per request. These queries include the collection of vehicle locations, the evaluation of daily expenditures for a particular vehicle, and the query of travel time. Additionally, there is an account balance query that is not partitionable on expressway, requiring an aggregate on a scan of all X-Ways to complete. There are separate historical queries as well, which are similar to OLTP queries rather than continuous queries.

Because the majority of the continuous queries in Linear Road are expressway-dependent, the benchmark is data-parallelizable on X-Way. Traditionally, the benchmark is evaluated on the number of X-Ways that it is a particular system can handle, determining how effective the addition of new nodes is at handling additional X-Ways.

4.2.3 Ideal Partitioning Strategy: Relaxation of Batch Atomicity

Partitioning the Linear Road benchmark (and other data-parallelizable workloads) in a transactional streaming system is on its face the same as in a traditional streaming system. The attribute that indicates the grouping is the one that the data should be partitioned on. In the case of Linear Road, that attribute is X-Way. Each partition in a transactional streaming system can contain one or more X-Ways, allowing the possibility for nodes to be added in order to accommodate additional X-Ways. In this scenario, any query that only accesses a specific X-Way (the bulk of the workload) will only access a single partition. A few, less-frequent queries will access multiple partitions in a distributed transaction.

The key problem to this partitioning strategy is that while the mutable state and processing

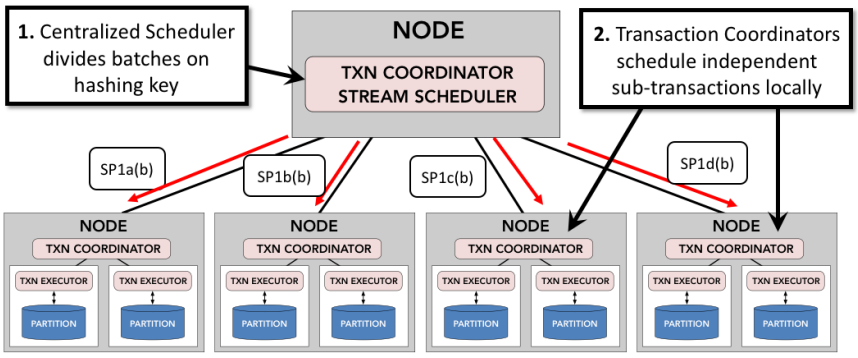


Figure 4.2: Data Parallelism: Division of a Batch into Sub-Transactions

can be easily partitioned in this manner, the *transactions* themselves contain a batch-id ordering component that spans partitions. By our correctness guarantees, a batch of tuples which contains information from multiple partitions must be executed as a distributed transaction, as they are a single atomic unit. This becomes a major problem, as effectively every transaction becomes a distributed transaction, greatly increasing communication cost and killing the performance benefits of the data parallelism.

The solution is to make a small modification to our correctness guarantees: relaxing our atomicity guarantees in order to divide the processing of a batch. In this scenario, data from an incoming batch can be hashed on a specific partitioning key, and divided into *subbatches*, one for each partition on which accessed data resides (Figure 4.2). Each subbatch contains a subset of the original batch, with each tuple residing in exactly one subbatch. A subbatch may be empty.

Each subbatch retains the properties of the original batch, but with only a fraction of the data (i.e. the portion of the data that matches the key hash). This includes the ”batch-id” property; each subbatch retains the original batch's batch-id, ensuring the same batch ordering as before. A transaction which executes on a subbatch also retains full ACID properties; specifically, subbatches are *atomic* with respect to their own tuples, but *not* with respect to the original batch.

The key difference between executing a single distributed transaction on the original batch versus a separate transaction for each subbatch lies in the execution strategy. Each subbatch transaction executes and commits independently of its *sibling* subbatches (i.e. subbatches which contain the same batch-id). This means that each subbatch transaction is able to execute as a single-sited transaction, without expensive communication overhead.

This also means that there are scenarios in which some sibling subbatch transactions may commit while others may fail. In the event that a subbatch transaction fails, the system will attempt to restart it in the same way that a typical transaction will be restarted. However, sibling subbatches

on other partitions will be completely agnostic to this restart and possible failure, resulting in only a portion of the original batch having actually committed. This is the primary consequence of dividing the original atomic batch.

It should be noted that the use of subbatches is a completely subjective decision on the part of the user. While they have potential to greatly improve performance under the right conditions, they also compromise our correctness guarantees in the ways described above. In workloads in which batch atomicity is a priority, it is *not* recommended that the user employ data parallelism in their design strategy.

4.2.4 Ideal Scheduling Strategy: Periodic Batch-Id Coordination

With the addition of subbatch transactions, the majority of dataflow graph executions in a data-parallelized setting should be limited to a single partition. All of the necessary data for a subbatch dataflow is co-located onto a single partition, so it stands to reason that the scheduling should be located on the same partition as well. Each node's transaction coordinator becomes responsible for scheduling its own partitions as though each procedure in the partitioned dataflow graph was exclusively located on it. This is very similar to how H-Store handles scheduling.

Unlike H-Store, coordination is required to ensure batch ordering is still maintained. While the scheduling on each node is kept mostly independent, there is a need for periodic coordination between nodes in order to synchronize the batch-ids of each partition's state. While most queries in a data parallelizable workload are partitionable, there are likely other streaming queries or OLTP queries on the state which require batch-id consistency across partitions.

Take for instance the "Account Balance" query in Linear Road. In this query, a user requests his or her toll balance at a given point in time (or batch-id). To gain this information, the system must aggregate together the sum total of account balances for that user across each expressway, each of which is stored on a separate partition. The shared state of each partition must have a matching batch-id, lest the batch-id consistency of the result be compromised. However, as stated earlier, each partition is independently executing its own subbatches without any regard for what the other nodes are doing. To provide a consistent view of the account balance as of a point in time, it is necessary to synchronize each node on batch-id.

On a regular basis (determined by a system administrator), a *periodic batch-id coordination* request is sent to each node's transaction coordinator, indicating that their batches should now be synchronized (Figure 4.3). These nodes will communicate among themselves in order to determine how far along each scheduler is in terms of batch-ids. Collectively they decide which transaction coordinator has scheduled the latest batch-id (batch b), and each coordinator determines that they

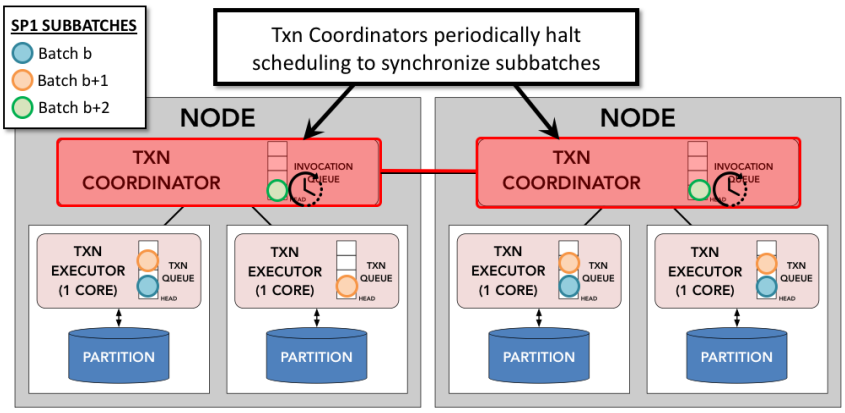


Figure 4.3: Data Parallelism: Periodic Subbatch Coordination

will stop scheduling transactions with a batch-id greater than that. One by one, each node stops scheduling transactions, and waits for all outstanding transactions to commit (or fail, as the case may be). Once all nodes have finished their outstanding streaming transactions, their state is synchronized on batch-id; all state on these nodes is processed up to the point of batch b. At this point, any distributed transactions requiring state from some or all of the partitions may run, and the result will be batch-wise consistent. Once these distributed transactions have finished, each node may resume scheduling subbatch transactions as normal. It should be noted that these coordinations must take place before a data snapshot can take place.

Such a complicated method of synchronizing subbatches across multiple nodes comes at an obvious heavy cost, as each node must completely halt streaming transaction executions for a notable amount of time. This leads to delays in normal streaming dataflow operation, as well as delays in distributed transaction execution. However, it is the assumption that in these cases, normal streaming dataflow operation performance is the priority, and that these distributed transactions are few and far between. The performance benefits and near-linear scalability gained from the data parallelism should outweigh the cost of these periodic batch-id coordination requests across nodes.

4.3 Database Design: Pipeline-Parallelizable Workloads

Unlike data parallelization, pipeline parallelization divides workloads based on their operators rather than hashing and dividing the incoming batches as they arrive. Each operator is assigned to a specific node, passing data between them throughout the course of the dataflow. This section describes the types of workloads that fit this pattern, and the design strategies that are employed to give them ideal performance.

4.3.1 Properties of Workloads

While many traditional streaming workloads are parallelizable on a single key, OLTP workloads involving shared mutable state frequently are not. As discussed in Sections 2.2 and 4.1, OLTP workloads in general and high-velocity OLTP workloads in particular tend to involve small, write-heavy operations on a very limited portion of the database [15, 123]. These databases tend to contain a variety of types of state, each with a partitioning key unique to its type.

Transactional streaming workloads that are more conducive to pipeline parallelism tend to include at least one, and frequently multiple, of the following traits:

Variety of State / Tables

Pipeline parallelism is crucial for workloads that include access to a wide variety of state. The more tables that a database contains, the more likely it is that the state should be sharded on different keys. This reduces the opportunity for data parallelism.

Transactionally Divisible Workload

Dividing a workload into multiple operations running on separate locations has the strongest performance impact if those operations are able to execute independently. Better performance is attainable if the lock on state is able to be released and the state made globally visible before the entire dataflow completes. Many workloads feature natural opportunities for transaction boundaries, allowing for better pipeline parallelism.

Need for Strong Transaction Guarantees

It goes without saying that transactional streaming workloads pair best with workloads that require ACID guarantees. Pipeline parallelism in particular is well-suited to this, as state is better able to be divided in a way as to minimize the need for extremely expensive distributed transactions.

Uneven Input Rate

Pipeline parallelism has been shown to increase resiliency in a stream processing system in the case of varying input rates [165]. A temporary increase in load to the system will be more evenly distributed in a pipeline-parallelized environment, whereas the load may disproportionately affect a subset of nodes if purely data parallelized.

4.3.2 Example: Streaming TPC-DI

Many streaming data ingestion workloads meet a variety of the characteristics listed above. One example of such a workload is modeled after a retail brokerage firm application, emulated by TPC-DI. TPC-DI is a data integration benchmark created by the TPC team to measure the performance of various enterprise-level ingestion solutions [129]. It focuses on the extraction and transformation

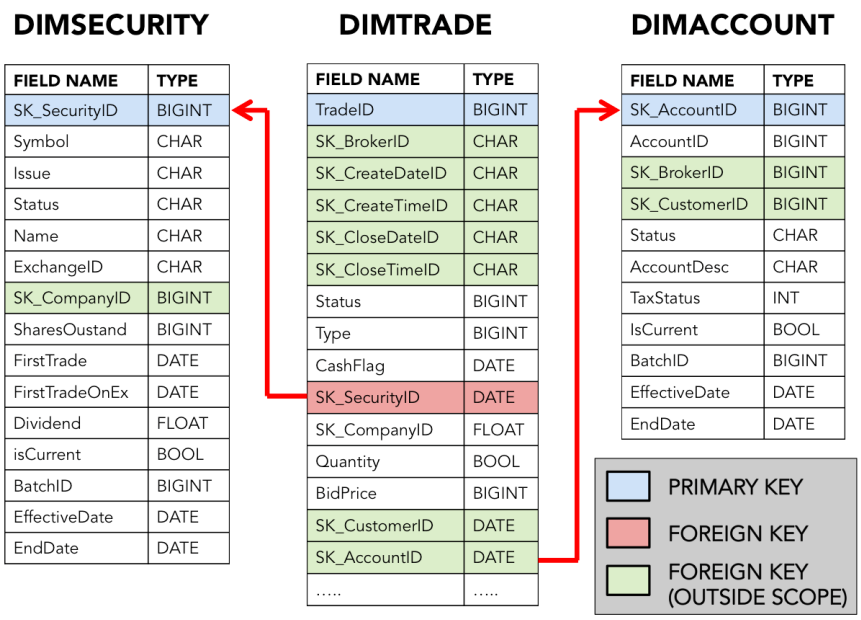


Figure 4.4: Partial Schema of TPC-DI

of data from a variety of sources and source formats (e.g. CSV, XML, etc.). These various flat files are processed in large batches, and the results are integrated as one unified data model in a data warehouse.

While TPC-DI was originally designed as a benchmark for traditional data ingestion, it can be re-imagined as a streaming ETL use case [109]. The obvious benefit to this conversion is quicker access to incremental results. Traditional ETL systems process large batches of data overnight, while a streaming version could process smaller microbatches throughout the day.

Assuming the ETL system outputs the same final results, there is no downside to taking a streaming approach. However, there are additional data dependencies that must be considered when breaking large batches into smaller ones. Take for instance the DimSecurity, DimAccount, and DimTrade tables, each of which are described in Figure 4.4. Note that the DimTrade table contains foreign keys on both the DimSecurity and DimAccount tables (other foreign keys also exist in these tables, but for simplicity, we will focus on this subset) [152]. When new rows are defined within the DimTrade table, reference must be made to the other two tables to assign the SK_SecurityID and SK_AccountID keys, which means that the corresponding rows must already exist in their respective tables.

Stream processing is an ideal way to handle these data dependencies. Assume a batching mechanism that groups the creation of security S, account A, and a trade request T from account A for

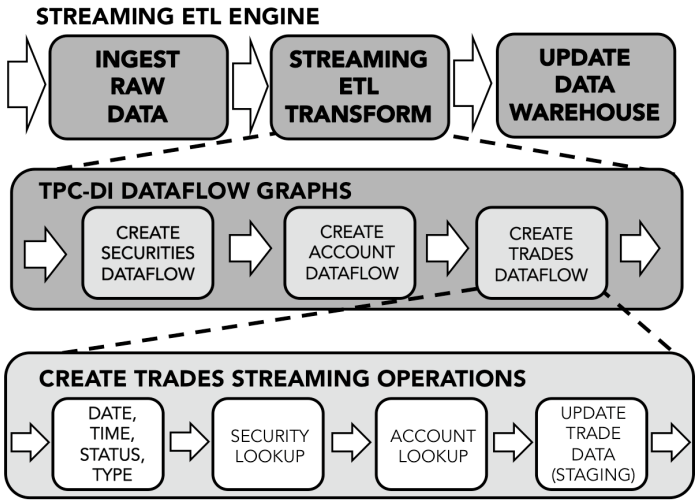


Figure 4.5: Partial Dataflow Graph Representation of TPC-DI

purchasing shares of security S. We can construct a dataflow graph of streaming operators that maintain the data dependencies of this batch while allowing processing flexibility. As illustrated in Figure 4.5, the batch is first processed by a $CreateSecurities$ dataflow graph, then $CreateAccount$, and finally, $CreateTrades$.

Such a workload contains all of the elements that encourage pipeline parallelism. There is a clear need for ACID transactions, as the state must be isolated as it is read from or written to the database. The read and write operations in the workload are each divisible into their own transactions, so long as their ordering is preserved. The variety of state makes it difficult to coordinate sharding keys, encouraging pipeline parallelism instead. Additionally, the stream input rate may be very uneven, as there are times when the financial industry is much busier than others (such as market open and close).

4.3.3 Ideal Partitioning Strategy and Dataflow Design

Transactional streaming adds a unique wrinkle to database design in its ability to partition not only state and processing, but also the workload itself into a dataflow graph. Frequently when stateful workloads are defined, specific transaction boundaries are left up to the developer. Designers of transactional streaming systems can exploit this workload flexibility to create dataflow graphs well-suited for distribution.

In general, stored procedures within a dataflow graph should follow a few simple rules to achieve best performance:

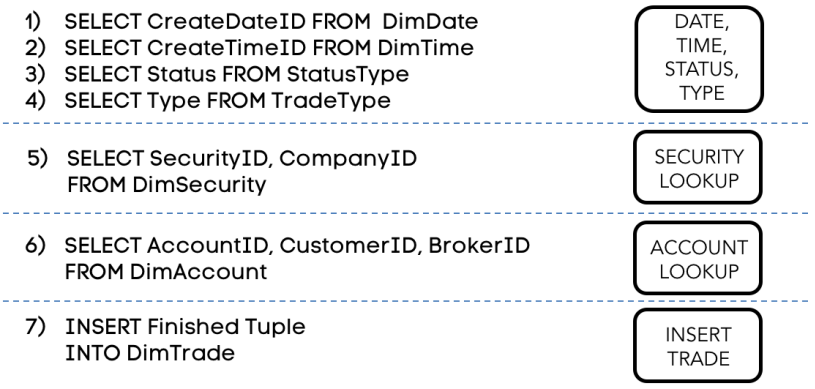


Figure 4.6: TPC-DI Operations Assigned to SPs

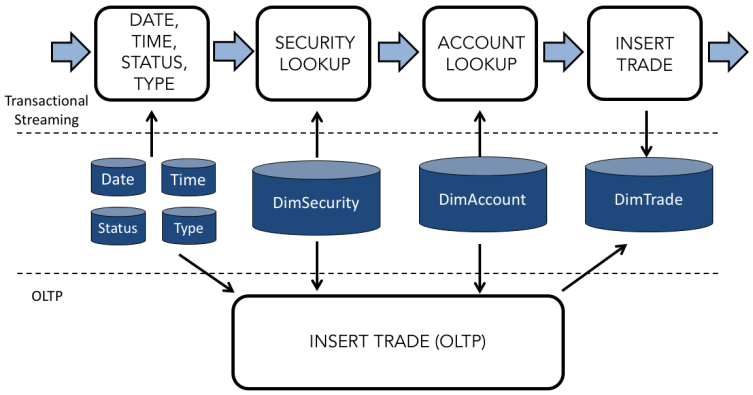


Figure 4.7: TPC-DI Dataflow Graph Compared to Single OLTP SP

1. To maximize distribution, workloads should be divided into granular units whenever possible. This maximizes the potential to balance load across the distributed system using pipeline parallelism.

2. By contrast, sequential operations to the same state (or multiple pieces of state that are co-located) should be confined to a single stored procedure if possible. This reduces the total number of transactions in available situations, each of which requires additional overhead, without causing inflexibility.

3. Stored procedures should access as few tables as possible. If possible, only one table should be accessed, or multiple tables partitionable on the same key. The fewer tables accessed by an operator, the easier it becomes to co-locate state to avoid distributed transactions.

Applying these rules to the $CreateTrades$ subset of TPC-DI results in the division of operations in Figure 4.6. Rules 1 and 2 lead to the workload being divided into four stored procedures, three of which are large dimension tables that could potentially be hashed on a single key. Separating these operations (Security Lookup, Account Lookup, Insert Trade) greatly increases options for sharding the data and processing across partitions. Rule 3, on the other hand, applies to the first SP (Date, Time, Status, Type). This SP access four tables, but all tables are small and read-only, and could easily be co-located or replicated. Thus, combining these operations into a single transaction is feasible, and reduces the overhead of creating four separate transactions when one will suffice. What results is the final dataflow graph seen in Figure 4.7, as opposed to the OLTP configuration seen at the bottom of the figure.

Determining stored procedure boundaries within a dataflow graph must entirely be left up to the user. While the listed rules can have a dramatic impact on performance, it is important to remember that these almost always change the semantics of the workload as well. A workload that is divided across many stored procedures is going to be very flexible for database design, but can also open the shared mutable state to external transactions mid-batch execution. The user must determine whether these semantics are acceptable, and if not, must find a way to protect the shared state throughout the batch execution. This is most often done via nested transactions (at the expense of performance).

4.4 Maintaining Global Ordering via Transaction Coordination

In the single-sited case, maintaining both a strong transaction isolation level and stream ordering constraints was relatively straightforward. Because there was no opportunity for parallelism, it was possible to simply schedule transactions from a singular dataflow graph in a serial manner, ensuring that each transaction executes in topological order without interference. This approach provided near-optimal throughput and latency in cases with only a single dataflow graph in a single-sited setting. Once parallelism is introduced, however, maintaining a global ordering of transactions becomes a much more interesting challenge.

Scheduling in a distributed transactional streaming system requires consideration of OLTP serializability in addition to global stream ordering. We will first discuss how transaction coordination is handled in distributed OLTP (Section 4.4.1). We then explore the additional global ordering requirements of a transactional streaming scheduler (Section 4.4.2). We explain how to efficiently combine global ordering with serailizability mechanisms that already exist (Section 4.4.3). Finally, we implement a distributed scheduler to S-Store's transaction coordinator, optimized to handle pipeline parallelized workloads (Section 4.4.4).

4.4.1 Transaction Coordination in H-Store

H-Store uses a decentralized transaction coordinator in order to minimize coordination across nodes [123]. While the coordinators do communicate transaction requests to one another and send heartbeat messages, the communication between one another is kept to a minimum.

As mentioned previously, H-Store executes transactions one at a time at each partition (i.e. serially), which reduces the need for heavy-weight concurrency control at each partition [32]. Instead, each transaction receives exclusive access to all data and indexes at each touched partition [123].

While inter-node coordination is kept at a minimum, it is still necessary to maintain a serializable isolation level across the distributed system. H-Store accomplishes this by using timestamp-based scheduling to determine transaction ordering [32, 123]. Once a transaction arrives at a node, the coordinator assigns a unique transaction identifier (or txn-id) based on its wall-clock arrival timestamp and its base partition ID. Assuming that the transaction is local, it is then placed in the appropriate partition queue as it waits to be executed. Local transactions are executed serially based on the order of their txn-ids.

Each partition contains a single lock managed by its node's coordinator [123]. Transactions receive a partition's lock in the order of their txn-id. In the event that a transaction with a larger txn-id has already been executed on the partition, it becomes necessary to abort the transaction and restart it with a new timestamp (and thus a larger txn-id). Similarly, if a transaction is scheduled to execute on one partition, but is revealed to instead need to be executed at a different partition, the transaction is aborted and redirected to the appropriate node (and reassigned a txn-id). Each transaction's execution is postponed by a few milliseconds in order to prevent the starvation of distributed transactions. This serial execution ordering at each partition also maintains serializability across the entire system.

4.4.2 Streaming Scheduler for Global Ordering

As described in Section 3.2.2, global ordering guarantees for transactional streaming refer to batch and dataflow ordering internal to each dataflow graph. In addition to global ordering, the serializable isolation constraint of OLTP must be met as well, providing limitations on both streaming and non-streaming transactions. Serializability is maintained between all transactions, but ordering guarantees provide an additional constraint to certain streaming transactions.

When considering the ordering of a transaction, it is always relative to another transaction. For the purposes of describing these interactions, we can divide them into two categories:

- **Global Ordering** - Global ordering transaction comparisons occur between two transactions from the same dataflow graph to which either dataflow or batch ordering constraints apply.

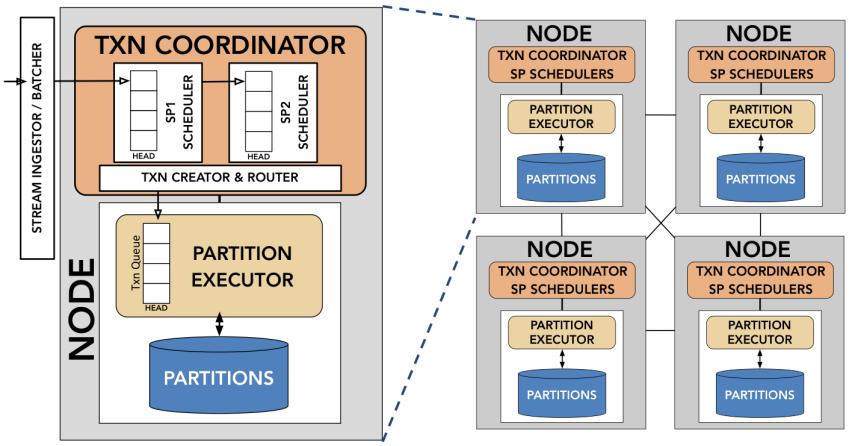


Figure 4.8: Distributed S-Store Architecture (with Decentralized Scheduler)

In the case of batch ordering, any transaction that originated from the same streaming stored procedure requires batch ordering considerations. For dataflow ordering, any transaction with the same batch-id originating from the same dataflow graph must be compared to one another via topological ordering. If neither of these cases apply, then the comparison is a normal serializable transaction comparison (even if the two transactions are both from the same dataflow graph). (Note that global ordering ensures serializability as well).

- **Serializable** - Any two transactions that do not require the more constrained global ordering must still be serializable. This applies to any comparison in which at least one of the transactions is an OLTP transaction, or comparing two transactions from the different dataflow graphs. Additionally, comparisons between two transactions from different stored procedures in the same dataflow graph fall under this category if they do not concern the same batch-id.

While serializability can be maintained on a partition-by-partition basis, global ordering must be maintained from a higher level. It is possible for a single stored procedure to produce multiple streaming transactions, each of which modifies state on a separate partition. It is important that, from the perspective of other transactions, these streaming transactions execute according to global ordering. Serializability maintained at the partition level will not necessarily provide that without additional mechanisms.

To maintain global ordering, a separate *streaming scheduler* component is created within the transaction coordinator in the partition engine. The streaming scheduler exists at the level of S-Store's transaction coordinator, and manages the ordering of transactions before they are put into one or more partition queues. It may either be *centralized* or *decentralized*, depending on the work-

load needs. In the decentralized case (Figure 4.8), communication between scheduling components becomes crucial to maintaining global ordering. Because the scheduler lives within the transaction coordinator, the same communication protocols are used. A TCP/IP connection is established between each coordinator, with Google's Protocol Buffer serialization library used to construct and decode the messages [123].

The streaming scheduler handles several responsibilities, including:

- **Maintaining Dataflow Information** - The streaming scheduler is the ideal location to maintain full knowledge of the topology of the dataflow graph(s), and which transactions have been invoked, are outstanding, or have aborted or committed. Performance statistics can also be managed from the streaming scheduler.
- **Maintaining Batch Ordering** - The batch ordering is maintained for each stored procedure individually. Any transaction that accesses state modified by streaming transactions will see a consistent snapshot of the data from the perspective of batch ordering.
- **Maintaining Dataflow Ordering** - The dataflow ordering is maintained for each batch relative to a dataflow graph's topology. Any transaction that accesses state modified by streaming transactions will see a consistent snapshot of the data from the perspective of batch ordering.
- **Invoking Triggered Transactions** - While transactions trigger their downstream counterparts without an extra component in the single-node case, the scheduling component is a natural location for this responsibility in a distributed setting.
- **Re-Routing and Moving Stream Data** - If a transaction needs to be invoked on another node, it is the scheduler's duty to route any necessary stream data to the appropriate location. This data movement must be transactional, but there are opportunities to optimize for minimal blocking communication (Section 4.5).

4.4.3 Combining Global Ordering with Timestamp-Based Serializability

There are two primary strategies for the streaming scheduler to maintain its ordering guarantees. Each strategy is not mutually exclusive; it is possible for different ordering guarantees to be handled scheduling strategies.

Pessimistic Scheduling Approach

In the pessimistic approach, the scheduler prevents the creation of a transaction until it has verified that its parent transactions have fully committed. The advantage to this approach is that it ensures

that the dependent transaction will never execute out of order, as it will not even be invoked before the commit. As a result, no contingencies are needed for if a transaction fails or attempts to execute out of order; if it fails, it can be restarted, and out-of-order execution becomes impossible.

The disadvantage to this approach is performance. Because dependent transactions are not even invoked until their parents have committed, there is no room for parallelization in the transaction invocation and queuing process. The pessimistic approach runs the risk of the scheduler not creating enough transactions to ensure that each partition is constantly processing a transaction (thus risking a loss of work cycles).

Optimistic Scheduling Approach

In the optimistic approach, the scheduler assumes that under normal circumstances, transactions will not attempt to execute out-of-order or fail. With that assumption, the scheduler is able to create a transaction (following its stream ordering) without verifying that its parent transactions have yet committed. The scheduler assumes that, because it ordered the transactions properly on its end, the partition components will also ensure that the ordering is maintained. This is not so different from speculative transaction execution, only it pertains to speculative transaction creation instead [16, 98].

The advantage to this approach is obvious: by scheduling transactions without upstream verifications, the scheduler can have a large quantity of transactions open at the same time, providing a constant supply of work for each partition. The downside, however, is that failures become significantly more expensive. As with optimistic concurrency control, any abort results not only in the undoing of the failed transaction, but also a cascading abort of all transactions dependent on it. Unlike OLTP, however, where dependencies are entirely serializability-based, transactional streaming must consider stream ordering dependencies as well. Cascading aborts involving many transactions can result. Thus, it is very important to have mechanisms in place which will minimize the number of transactions that need to be rolled back.

Take for instance a simple example illustrated in Figure 4.9. In this example, we have a stored procedure SP_1 within a dataflow graph, which creates transactions $SP_{1,1}$ and $SP_{1,2}$ on batches B_1 and B_2, respectively. Suppose that $SP_{1,1}$ exclusively accesses partition P_1, and $SP_{1,2}$ exclusively accesses partition P_2. If the two transactions are both invoked one after the other according to batch ordering yet exist simultaneously, it becomes possible for $SP_{1,2}$ to commit before $SP_{1,1}$. In normal runtime (i.e. both transactions commit), this is fine, so long as distributed transactions execute in a way that is serializable. Outside transactions are not allowed to execute while only $SP_{1,2}$ is committed, batch ordering can be maintained from the perspective of the state. However, if $SP_{1,1}$ aborts after $SP_{1,2}$ has committed, suddenly recovery becomes more difficult. In this simple

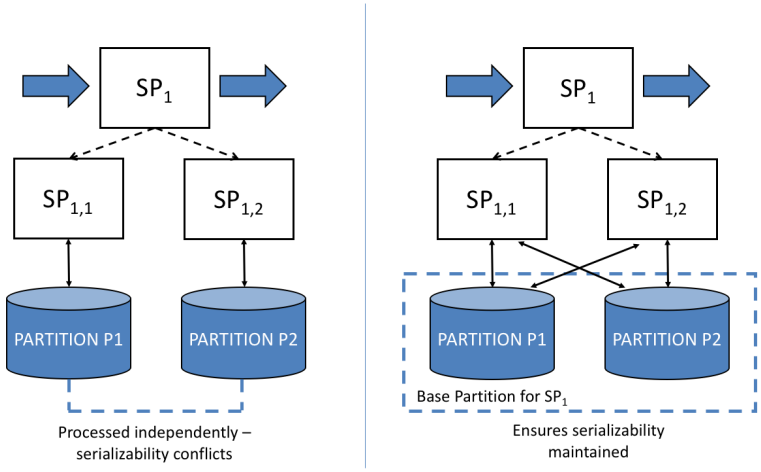


Figure 4.9: Serializability conflict example in optimistic scheduling (and base partition solution)

case, the work that $SP_{1,2}$ has achieved needs to be undone. However, if other transactions have committed that are dependent on either transaction, those will also need to be aborted and undone, and a potential chain reaction can result.

To minimize this coordination, we take advantage of the fact that streaming SPs have an extremely predictable and consistent workload, and execute on batches of data. For now we assume that each SP will always access the same subset of partitions, and thus each transaction of the SP will have the same base partition (shown on the right-hand side of Figure 4.9). This makes the communication across partitions unnecessary since all of them will necessarily be locked on the transaction execution.

4.4.4 Distributed Localized Scheduling: Implementation and Algorithm

To attain a balance of performance during normal runtime and upon abort, we take a pessimistic scheduling approach to dataflow ordering, and an optimistic scheduling approach to batch ordering. This combination allows for many transactions to be scheduled for the same stored procedure simultaneously while limiting the need for cascading aborts in the event of a failure. Additionally, this combination of approaches allows the streaming scheduling to take place not at the dataflow level, but instead at the stored procedure level. For that reason, ordering is handled in a *procedure scheduler* component. Each streaming stored procedure is assigned a unique procedure scheduler, each located on a single node. As mentioned earlier, we assume that each streaming stored procedure is assigned to a single base partition in order to minimize conflict on aborts. To minimize coordination costs, it is most effective to locate the procedure scheduler within the transaction co-

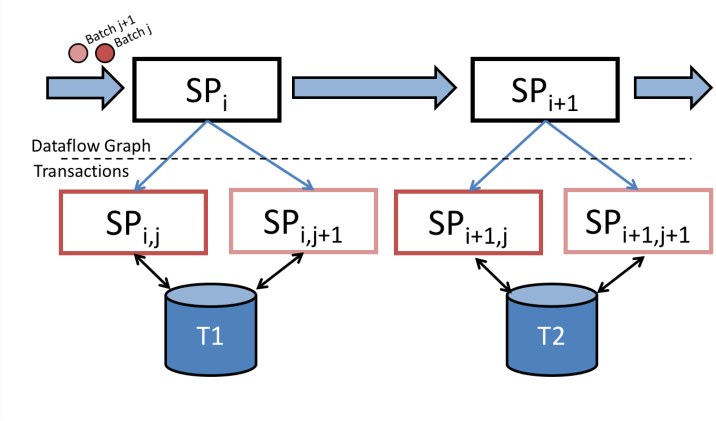


Figure 4.10: Distributed Localized Scheduling - 2 SP Example

ordinator of the same node as its base partition. For this reason, we call our approach *distributed localized scheduling*.

At the procedure scheduler level, new *transaction invocations* arrive from either the batch generator or from another procedure scheduler. Transaction invocations include a batch-id and a set of input tuples. The procedure scheduler works with the base partition to assure that batch ordering is maintained.

Below is a description of our distributed localized scheduling algorithm, based on a simple two-stored procedure example from Figure 4.10.

Maintaining Dataflow Ordering

In order to maintain dataflow ordering, we are looking to ensure that for a given batch j, any downstream transaction $SP_{i+n,j}$ does not commit before $SP_{i,j}$ commits (where $n > 0$). An illustration can be found in Figure 4.11.

Because each procedure scheduler is independent of one another, it is prudent to limit communication between them. The primary form of communication between procedure schedulers comes from passing a transaction invocation from upstream scheduler $Sched_i$ to downstream scheduler $Sched_{i+1}$ (associated with SP_i and SP_{i+1}, respectively).

We take a pessimistic approach to managing dataflow ordering, meaning that we do not schedule transaction invocations for a downstream procedure $SP_{i+1,j}$ until we have confirmed that $SP_{i,j}$ has *committed*. There are two reasons to take the cautious approach here. The first is that if we allow downstream transaction $SP_{i+1,j}$ to be created before its parent commits, we run the risk of compounding potential rollbacks due to failure, needing to abort all transactions from all downstream SP_{i+1} to SP_{i+n} on batch j and larger. If we take the pessimistic approach to dataflow ordering, we

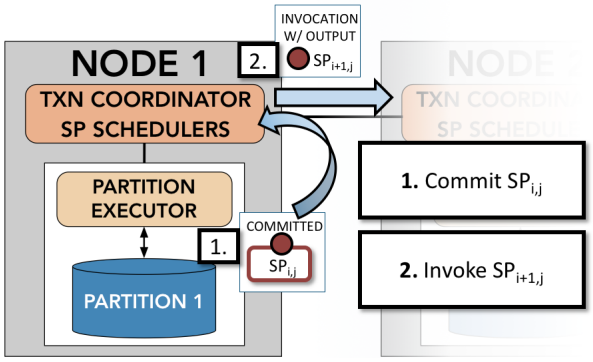


Figure 4.11: Maintaining Dataflow Ordering

can limit the rollback damage to only a single SP.

There is another reason that the pessimistic approach is logical in this case: transaction invocations require input data items in order to generate a full transaction, execute, and commit. This batch of input data for $SP_{i+1,j}$ is gained from the output data of $SP_{i,j}$, which is collected upon its commit. Thus, creating downstream transaction invocations becomes a natural barrier point for scheduling.

Because SP_{i+1}, j is not invoked until $SP_{i,j}$ is committed, $SP_{i,j}$ will commit before any $SP_{i+n,j}$ for all $n > 0$ by induction.

Maintaining Batch Ordering (Commits)

To maintain batch ordering, we look to ensure that for any stored procedure SP_i, an earlier batch j in transaction SP_i, j commits before any later batch $j + n$ commits in transaction $SP_{i,j+n}$ (for all $n > 0$). For this section, we will assume the case where all $SP_{i,j+n}$ commit. Failures will be addressed in the next section (4.4.4). An illustration can be found in Figure 4.12.

To mitigate the performance loss in dataflow ordering, batch ordering takes the optimistic approach. To do this, we take advantage of the existing serializability mechanisms, and assume that on any partition P, transactions will commit in the order that they were created (and will abort if an out-of-order transaction is attempted). So, for any TXN_x and TXN_{x+y} that both execute on partition P, where x and $x + y$ are timestamps and $y > 0$, TXN_x will commit before TXN_{x+y} as a given.

Additionally, we assume that all SP_i execute on a particular base partition set P_i. This is made possible by the fact that streaming workloads are highly rigid and predictable, and because of the batch processing restrictions discussed in Section 4.1.3, it is best for performance-based designs to

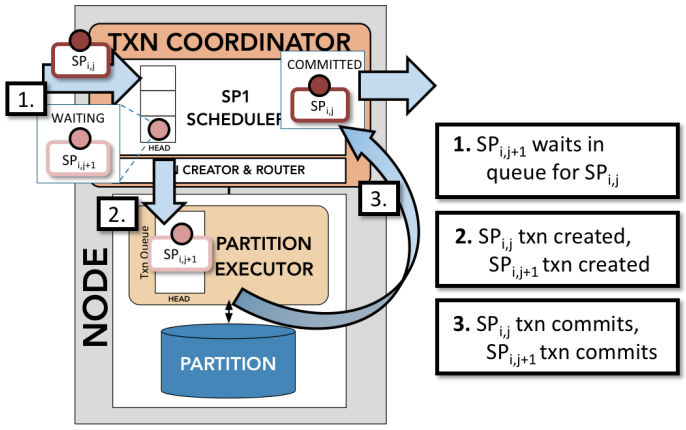


Figure 4.12: Maintaining Batch Ordering (Commits)

limit the partitions on which a procedure can execute.

$Sched_i$ creates all transactions of SP_i, and will not create any transaction $SP_{i,j+1}$ until it has created a transaction for $SP_{i,j}$. If a transaction invocation for $SP_{i,j+1}$ arrives before the invocation for $SP_{i,j}$, it will hold $SP_{i,j+1}$ in queue until $SP_{i,j}$'s invocation arrives and its transaction is created.

Assume that $SP_{i,j}$ and $SP_{i,j+1}$ are created, in that order, and assigned TXN_x and TXN_{x+y}, respectively. Both transactions are sent to the same base partition set P_i, and are placed in queue in order of their timestamps as per the earlier assumptions. In this case, we assume that both transactions commit. $SP_{i,j}$ commits first due to its earlier timestamp, and $SP_{i,j+1}$ commits afterwards. By induction, $SP_{i,j}$ will always commit before $SP_{i,j+n}$ for all $n > 0$, assuming no aborts.

Maintaining Batch Ordering (Aborts)

To maintain batch ordering, we look to ensure that for any stored procedure SP_i, an earlier batch j in transaction $SP_{i,j}$ commits before any later batch $j + n$ commits in transaction $SP_{i,j+n}$ (for all $n > 0$), even in the case of aborts. We make the same assumptions as the previous section. An illustration can be found in Figure 4.13.

Assume that $SP_{i,j}$ and $SP_{i,j+1}$ are once again created, in that order, and assigned TXN_x and TXN_{x+y}, respectively ($y > 0$). Both transactions are sent to the same base partition set P_i, and are placed in queue in order of their timestamps. This time, assume that $SP_{i,j}$ aborts. Post-abort, the transaction invocation is sent back to $Sched_i$ with an incremented restart counter $RS_i = 1$. $Sched_i$ immediately halts the creation of any new transactions on SP_i. At this point, some q transactions $SP_{i,j+1}$ through $SP_{i,j+q}$ are still in the partition queue(s) of P_i.

At abort time, P_i also flags SP_i as having aborted, and begins purging its queue of all SP_i

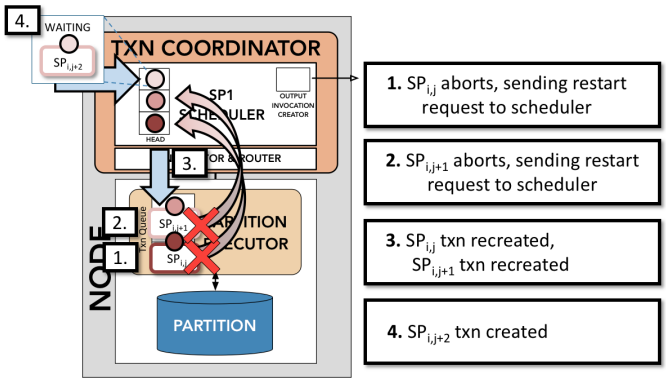


Figure 4.13: Maintaining Batch Ordering (Aborts)

transactions (incrementing their restart counters, and sending them back to $Sched_i$ for restart). This is done to ensure that no transaction from SP_i commit out of order. P_i will continue this until all q transactions have been removed from the queue and marked for restart.

Once P_i has finished, $Sched_i$ begins creating new transactions for its restart queue, starting with $SP_{i,j}$ (assigning it some TXN_{x+z} where $z > y$) and continuing in batch order so that $SP_{i,j+1}$ is assigned TXN_{x+a} where $a > z$. In the ideal case, $SP_{i,j}$ commits on the rerun, resulting in the conclusions of Section 4.4.4. If instead $SP_{i,j}$ aborts again, the process described in this section repeats with an incremented restart counter.

In the event that $SP_{i,j}$ continues to abort, this process repeats ad-infinitum. Obviously, this is likely to not be the desired outcome. The user has the option to apply a restart limit for transactions, which will force a permanent abort of the offending transaction in order to avoid falling too far behind real-time. In this case, $Sched_i$ will cease scheduling $SP_{i,j}$, and will notify any downstream schedulers $Sched_{i+1}$ of this decision. This violates our exactly-once and ordering rules, but in many cases is better than blocking new batches indefinitely.

The pseudocode for the procedure scheduler and corresponding base partition is provided in Algorithm 1 and Algorithm 2, respectively.

Algorithm 1 Pseudocode of Procedure Scheduler $Sched_i$ (for SP_i)

```
1:  PriorityQueue invocQueue                                   ▷ ordered on batchId
2:  PriorityQueue restartQueue                                 ▷ ordered on restartCount, batchId
3:  long lastStartedBatchId ← −1                               ▷ last batchId with a txn created for SP_i
4:  long lastRestartBatchId ← −1                               ▷ last batchId with a txn restarted for SP_i
5:  long lastCommittedBatchId ← −1                             ▷ last batchId with a txn committed for SP_i
6:
7:  function AUDITRESPONSE(txn,status)
8:      if status is COMMITTED then
9:          if lastCommittedBatchId < txn.batchid then
10:             lastCommittedBatchId ← txn.batchId
11:         if lastRestartBatchId < txn.batchId then
12:             lastRestartBatchId ← txn.batchId
13:         for each downstream SP_{i+x} of SP_i do
14:             procInvc ← new procInvocation(SP_{i+x},txn.outputData, 0)
15:             queue procInvc at Sched_{i+x} (must be confirmed)
16:     else if status is ABORTED and txn.restartCount is less than the restart threshold then
17:         procInvc ← new procInvocation(SP_i, txn.inputData, txn.restartCounter + 1)
18:         add procInvc to restartQueue
19:         if lastCommittedBatchId < txn.batchId then
20:             lastRestartBatchId ← lastStartedBatchId − 1
21:     else
22:         send error for txn and status
23:         update lastCommittedBatchId and notify downstream schedulers   ▷ Blocking
24:     if lastCommittedBatchId == lastStartedBatchId then
25:         reset the restartCounter at the base partition for SP_i        ▷ Blocking
26:
27: while S-Store is running do
28:     if restartQueue is not empty then
29:         if restartQueue.peek().batchId == lastRestartBatchId + 1 then
30:             procInvocation ← restartQueue.poll()
31:             create new transaction for procInvocation and queue in SP_i's base partition
32:             confirm transaction is in queue                            ▷ Blocking
33:     else if invocQueue.peek().batchId == lastStartedBatchId + 1 then
34:         procInvocation ← invocQueue.poll()
35:         create new transaction for procInvocation and queue in SP_i's base partition
36:         confirm transaction is in queue                                ▷ Blocking
37:         lastStartedBatchId ← procInvocation.currentBatchId
```

Algorithm 2 Pseudocode for Base Partition P

```
1: priorityQueue txnQueue                                   ▷ ordered by txnId
2: restartCounter for each SP for which P is a base partition
3:
4: while S-Store is running do
5:     txn ← txnQueue.poll()
6:     if restartCounter for txn.SP > txn.restartCounter then
7:         status ← abortTxn(txn)
8:     else
9:         status ← executeTxn(txn)
10:    if status is ABORTED and restartCounter for txn.SP < txn.restartCounter + 1 then
11:        restartCounter for SP ← txn.restartCounter + 1
12:    Call AUDITRESPONSE(txn,status) at the procedure scheduler for txn
```

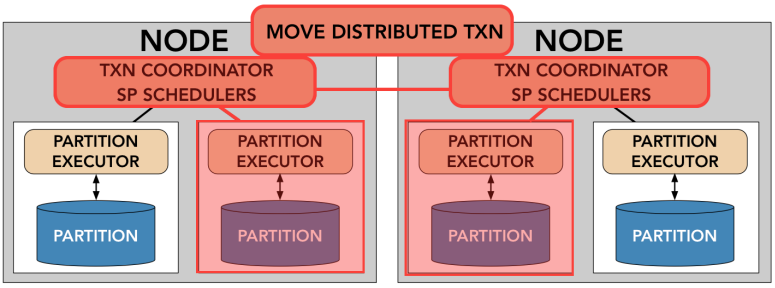
4.5 Efficient Movement of Stream State

In the transition between transactions in a dataflow graph, the output stream data of upstream transaction $SP_{i,j}$ must then be moved to the location where a downstream transaction $SP_{i+1,j}$ is going to execute. In the single node case, the stream data could simply be stored on the single partition and read directly from there. With multiple nodes introduced, it frequently becomes necessary to physically move the data between partitions (though only for cases in which SP_i and SP_{i+1} execute on separate partitions). Fortunately, this movement can be optimized due to the immutability of stream data. We call this a *MOVE* operation.

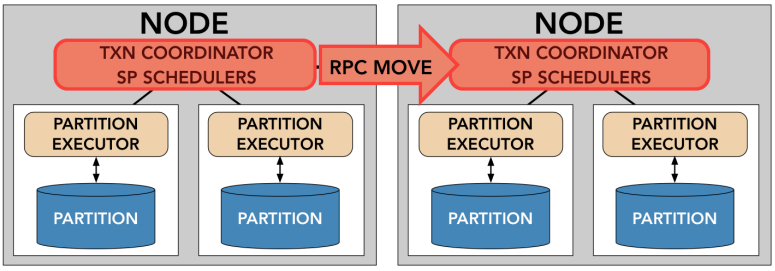
Distributed Transaction Implementation

In a naïve MOVE implementation, data is transferred using a dedicated MOVE system stored procedure (Figure 4.14(a)). The MOVE system stored procedure executes between any two consecutive streaming SPs in a dataflow graph which are executed on different partitions. For instance, take a dataflow graph in which SP_i (base partition P_1) precedes SP_{i+1} (base partition P_2) topologically. After $SP_{i,j}$ has committed but before $SP_{i+1,j}$ can begin, the MOVE operation $MOVE_{SP_{i,j},P_1,P_2}$ will be executed in order to migrate the output stream data of $SP_{i,j}$ from P_1 to P_2. Note that the naïve implementation will also remove the data from P_1 (i.e. it is a true movement of the data).

The MOVE operation is a full distributed transaction in this case, which will require significant coordination and lock both P_1 and P_2 for the entire duration. This has the advantage of guaranteeing full ACID correctness of the data migration, but is capable of grinding performance to a halt.



(a) MOVE: Distributed Txn Implementation



(b) MOVE: RPC Implementation

Figure 4.14: MOVE Implementations

This implementation is not practical for the low-latency, high-throughput workloads that demand transactional streaming.

Remote Procedure Call Implementation

One alternative to system stored procedures is to use *remote procedure calls* (RPCs) to move the data (Figure 4.14(b)). In this approach, the remote procedure calls include all of the stream data directly inside the downstream procedure invocation. In this implementation, the output stream data is written to the partition, but also pulled from the partition into the procedure scheduler upon commit, returning the entire result set as an output value. The procedure scheduler for SP_i on partition P_1 parses that output value and includes it in any downstream invocations. If the downstream procedure SP_{i+1} is located on another partition P_2, the procedure scheduler $Sched_i$ will then serialize the output stream for movement across a TCP connection along with the transaction invocation. When the procedure scheduler $Sched_{i+1}$ receives the procedure invocation, it deserializes the same input stream. When the transaction invocation is executed as a transaction, the data is passed as an input parameter.

The primary advantage to this approach is that most of the steps can be executed in parallel

to transactions being executed on P_1 and P_2. Once the result set is returned to the Partition Engine, the transaction coordinator can immediately begin executing the next transaction in its queue. The serializing, data transfer over the network, and deserializing can all take place parallel to the meaningful transaction work. Remote procedure calls are executed via two-phase commit protocol, ensuring that the data transfer is atomic and consistent. The output stream data is by definition immutable, which makes the approach possible and provides data isolation.

In order to ensure the durability of the stream movement in the face of system failure, it is necessary to additionally write the result set of each transaction to a stream object in the Execution Engine. This write takes place on the location of SP_i's execution. It is used to recreate transaction invocations that were lost in the event of a failure. As detailed by the recovery algorithms of Section 3.2.4, the batches in the output stream are compared with the executions that result from the transaction log, and any missing transaction invocations are reconstructed based on the output stream data.

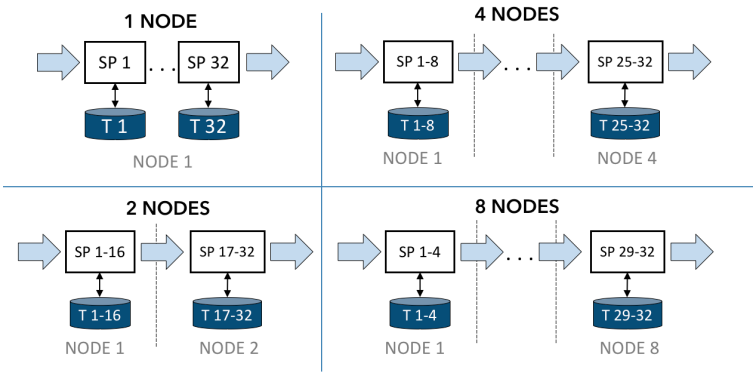
Note that this version of the MOVE operation could be considered more of a COPY, as the original batch remains in the stream object until garbage collection. Garbage collection takes place during a global data snapshot, as downstream transactions will no longer need to be reconstructed at that point. Garbage collection of a batch may also occur once that batch has reached completion of the entire dataflow graph.

Despite the additional overhead, the added parallelism opportunities of the remote procedure calls vastly improves both latency and throughput of a transactional streaming system. While MOVE operations still involve a performance cost, they are no longer prohibitive with this technique.

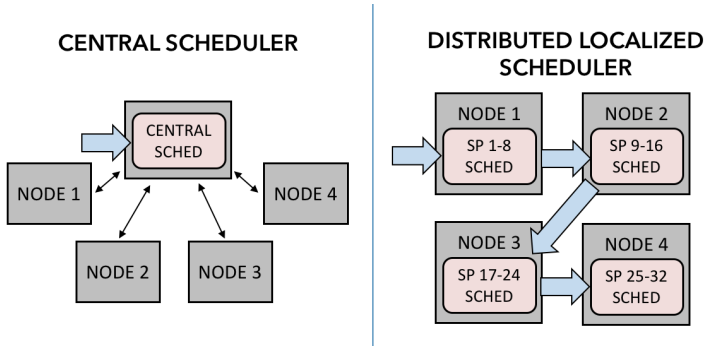
4.6 Performance Comparisons

We present the results of our experimental study comparing various configurations of distributed S-Store to one another, or against our main-memory OLTP predecessor H-Store. To evaluate performance, we compare maximum possible system throughput (at an equally-high input rate) using a latency bound indicating the average end-to-end time of a batch completing an entire dataflow graph. Unless otherwise specified, we measure the throughput in terms of *tuples per second*, use a latency bound of one second, and use a fixed batch size of 10 tuples per batch.

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. In some cases, we limit each machine to only a certain number of S-Store partitions allocated to any single machines, but we never limit the processing power of the machines.



(a) Dataflow Graph Distribution Configurations



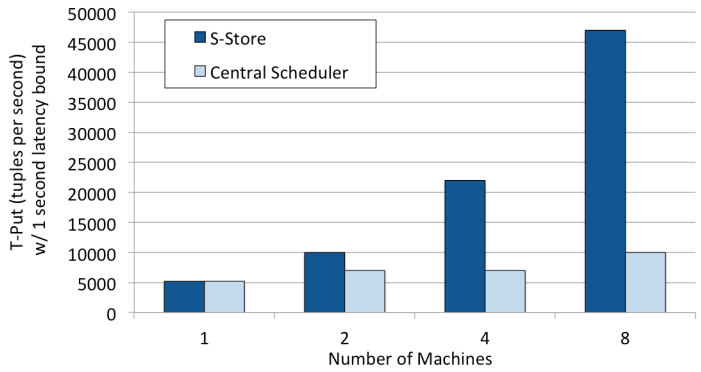
(b) Centralized Scheduler vs S-Store (DLS)

Figure 4.15: Scheduler Experiment Configurations (32 SP Workload)

4.6.1 Scheduling Performance: Pipeline-Parallelizable Workloads

To demonstrate the performance of our Distributed Localized Scheduler, we created a large toy benchmark designed to maximize the flexibility of distribution of a dataflow graph. The benchmark consists of 32 stored procedures, each connected in series in a dataflow graph (Figure 4.15(a)). Each stored procedure takes in a batch of tuples which feature a tuple-id/tuple-value pair. It accesses a single, unique table, that is entirely contained on a single S-Store partition. The stored procedure either reads from the table with 90% probability, or writes to the table with 10% probability (determined by the client when the input batch is created). Pipeline parallelism is employed by evenly distributing the tables across all available machines, and assigning the stored procedure schedulers to the optimal nodes in the DLS case (Figure 4.15(b)). Each machine was limited to four data partitions, meaning that the total number of partitions used for each experiment is four times the number of machines.

First, we compare our DLS solution to a centralized scheduler configuration within S-Store



(a) S-Store (DLS) vs Central Scheduling

(b) S-Store (DLS) vs Asynchronous H-Store Client

Figure 4.16: Scheduler Experiment (32 SP Workload)

(Figure 4.16(a)). The data points of 1, 2, 4, and 8 machines (4, 8, 16, and 32 partitions, respectively) were chosen because each is able to evenly divide the 32 stored procedure workload, thus highlighting the difference in scheduler performance. S-Store is able to gain significant performance improvement using the distributed localized scheduler rather than a centralized scheduler. The distributed scheduler allows S-Store to scale linearly as more machines are added (up to the point at which the workload is fully parallelized). By contrast, central scheduling becomes a major throughput bottleneck, as almost all transactions must be run remotely. Instead, the DLS configuration only communicates between schedulers for downstream invocations, and each scheduler is located on the same node as the data its SP accesses.

Additionally, we compare against an H-Store configuration which has a dataflow graph implemented in the client, but sends its transactions to the system *asynchronously* (Figure 4.16(b)). Because the transactions are sent asynchronously, batch ordering is *not* maintained in this scenario.

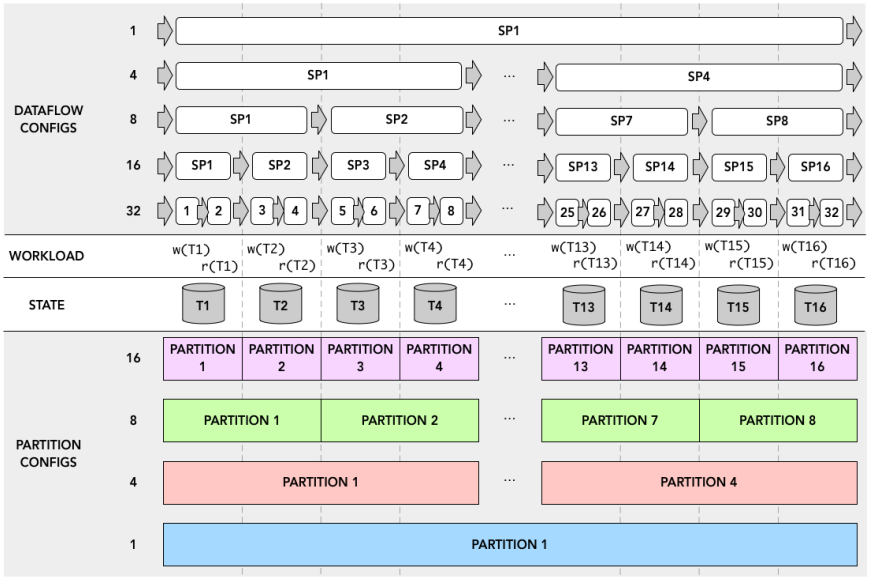


Figure 4.17: Procedure Boundaries Experiment Configurations

It is important to note that to provide batch ordering guarantees in H-Store, all transactions would need to be sent in series, destroying any opportunity for parallelism [110]. H-Store's client is able to asynchronously submit transactions to the engine without concern for the order in which they execute, so it might be expected that it would achieve a very high performance and scalability. However, because H-Store is still relying on the single location of its client for all of its streaming transaction scheduling, it incurs a similar communication cost to the central scheduler. Beyond roughly 4 machines, this becomes a bottleneck and limits scalability.

4.6.2 Dataflow Partitioning and Design for Pipeline-Parallelism

In workloads where transaction boundaries are flexible, intelligent division of reads and writes within operator boundaries is crucial for distributed performance. This provides better opportunities to shard state and processing in a way that minimizes communication costs and maximizes load balancing potential (Section 4.3.3).

To demonstrate this, we posit a workload that writes, then reads, from a single table, then performs the same actions on another table. This continues for a total of 16 separate tables, T1 through T16. First the value of a set of tuples is updated in T1 (by adding a specified amount), then the next set of tuple-ids is read from the same table. That set of tuple-ids is used to determine which tuples to update in T2, and so on. We assume that while the reads and writes must take place in a specified order, the specific transaction boundaries are fluid (i.e. all 16 tables do not need to be

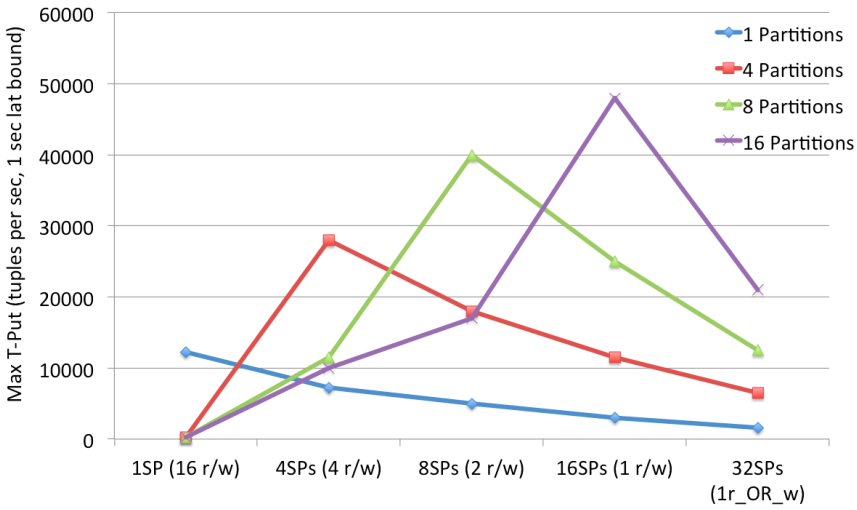


Figure 4.18: Procedure Boundaries Comparison

locked at the same time).

This experiment explores the performance difference between multiple dataflow configurations, as illustrated in Figure 4.17. The 16 writes and 16 reads can be organized in a variety of operator configurations, from all in a single stored procedure, to 16 stored procedures (each of which writes and reads a single table), to 32 stored procedures (each performing a single write or read). Similarly, a variety of table partitioning configurations can be used. All 16 tables can be placed on a single partition, they can be divided evenly across 4 partitions (4 tables apiece), or divided across 16 partitions. In all cases (besides the single-partition case), the partitions are evenly divided across 4 machines, and it is assumed that the procedure schedulers are optimally located on the same machine as the data they access.

As you can see in Figure 4.18, the best performance for each partitioning configuration is achieved when the number of stored procedures in the dataflow graph matches the number of partitions being accessed. When there are fewer stored procedures than partitions, each transaction will access multiple tables across multiple partitions, thus incurring a distributed transaction cost. By contrast, when the number of stored procedures exceeds the number of partitions, additional overhead is incurred for beginning and committing the additional transactions. Extra communication cost is also accrued as data is passed from one SP to the next. Ideally, the workload is divided across as many partitions as possible, with the number of procedures accessing the state matching the number of partitions. That way each procedure is single-sited, and each partition is able to perform transactions in parallel.

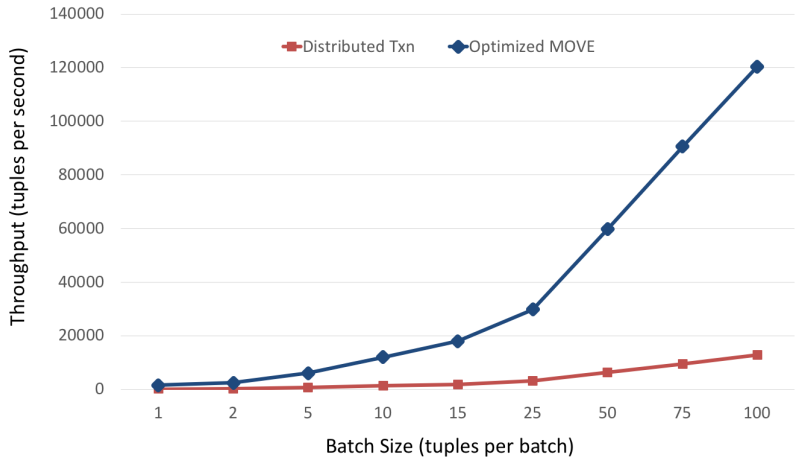


Figure 4.19: MOVE Op Comparison

4.6.3 Data Stream Movement

In order to demonstrate the performance benefit gained from our RPC MOVE implementation, we developed a micro-benchmark that almost exclusively MOVEs data from one node to another. The workload consists of two stored procedures SP_1 and SP_2, each of which does a write to its own partition P_1 and P_2, with a cross-node movement in between ($SP_1 \rightarrow SP_2$). In the "Distributed Txn" case, these stored procedures are separated by a third SP which deletes the data from the stream object on P_1 and writes it to P_2 in a distributed transaction (making the full dataflow graph $SP_1 \rightarrow MOVE \rightarrow SP_2$). The results are found in Figure 4.19.

In all cases, the RPC MOVE achieved a significantly better maximum throughput than the Distributed Txn case, but the difference became much more pronounced as the batch size was increased and more data was passed between nodes. While both configurations were able to improve their throughput (in terms of tuples per second) with an increased batch size, the RPC MOVE separated itself through its ability to move batches asynchronously between nodes, without the need to hold a lock on either touched partition during the MOVE.

4.6.4 TPC-DI Performance Comparison

To test the overall scalability and performance of a pipeline parallelized workload, we implemented a portion of the TPC-DI workload described in Section 4.3.2. Specifically, we implemented the dataflow graph associated with the ingestion of new Trade data items.

We implemented the design from Figure 4.20 with the assumption that each stored procedure runs on a separate partition containing the table(s) it requires. We compare the performance of this

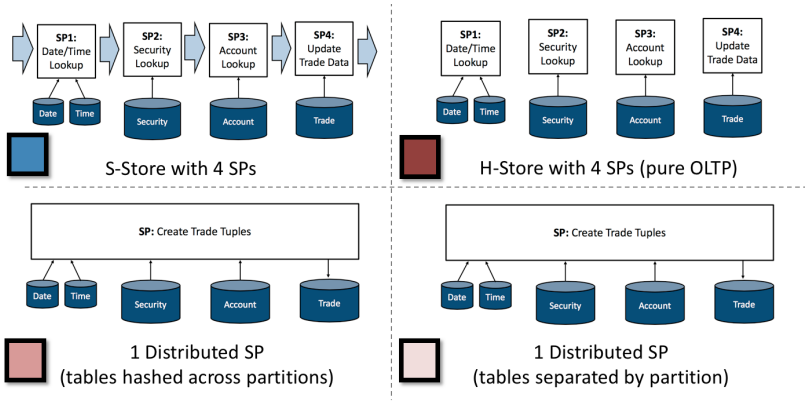


Figure 4.20: TPC-DI Trade.txt Ingestion Configurations

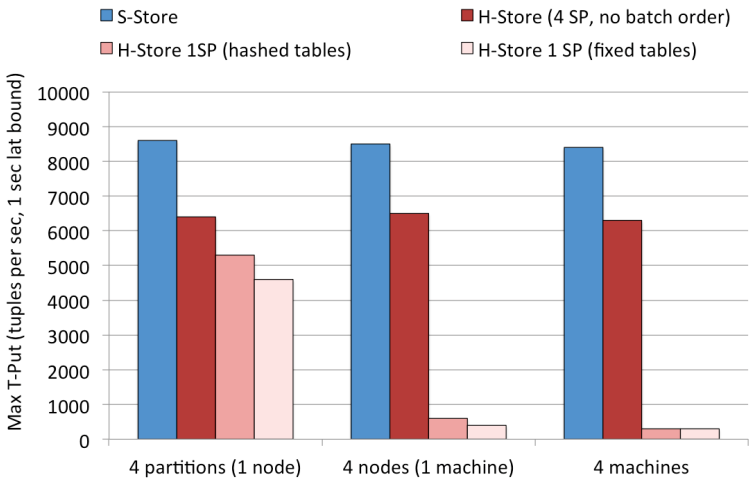


Figure 4.21: TPC-DI Performance Comparison

design against a variety of H-Store configurations: one using H-Store's default hashing scheme for each table and running all operations in a single distributed stored procedure, one matching the state configuration of S-Store (i.e. each table gets its own partition) and running a single stored procedure, and a third mimicking the S-Store configuration as best as possible and implementing the dataflow graph in the client. Each configuration is run on 4 partitions, with those partitions either being located on a single node, across multiple nodes running on the same machine, and across 4 separate machines. All performance was recorded in terms of throughput in tuples-per-second, with a 1 second latency bound for the entire dataflow.

As can be seen in Figure 4.21, S-Store is able to utilize pipeline parallelism to get strong performance regardless of configuration. Because the stream movement is negligible in terms of through-

put and each transaction is single-sited and locally scheduled, performance is consistent even across several machines. H-Store is able to achieve similar performance if the workload is similarly divided into four single-sited procedures, but because the dataflow is implemented in the client, additional communication is incurred at each stored procedure. Additionally, this configuration does not provide batch ordering guarantees. When the workload is instead implemented as a single distributed SP, performance suffers greatly. The distributed transaction is not horribly expensive when kept to partitions on a single node, but as soon as multiple nodes are involved, performance grinds to a halt.

Due to the combination of distribution techniques discussed in this chapter (pipelined division of operations, distributed localized scheduling, and optimized RPC stream data movement), S-Store is able to maintain practically the same parallelized performance across four partitions, regardless of whether those partitions are all located on the same node or across multiple machines.

4.7 Conclusions

Distribution of transactional streaming workloads can potentially require suffocating coordination costs if not implemented in a way that takes advantage of the workload being distributed. While distributed transaction execution mechanisms can be inherited from main-memory OLTP systems, maintaining global dataflow and batch ordering in a shared-nothing environment requires additional consideration. Global ordering can be maintained by utilizing the serializability guarantees of shared-nothing OLTP. A scheduler, either centralized or decentralized, can manage the transaction IDs and timestamps such that batches are executed in the expected order. Additional mechanisms allow the ability to abort and restart dependent transactions in case of a parent transaction's failure.

Streaming systems involve a lot of moving stream data state from one location to another, and transactional streaming is no exception. It is a high priority to be able to move immutable stream state transactionally without hindering workload performance. We accomplish this by passing state through asynchronous transaction invocations backed by two-phase commit.

Database design plays an enormous part into finding an efficient approach to distributing the workload. In cases where data parallelism is possible, it is most efficient to break a batch into multiple subbatches and centrally schedule each subbatch on the corresponding partition with the data it need. This relaxes the atomicity guarantee of batch processing, but comes at large gains for performance. For many other workloads, pipeline parallelism is most efficient. In these situations, it is more efficient to utilize distributed local scheduling, bringing the scheduling of each stored procedure to the node on which it will run.

System performance varies wildly depending on how well these elements are accounted for when designing the database for specific workloads. In Chapter 5, we discuss use-cases for dis-

tributed transactional streaming, and in Chapter 6, we discuss the potential for automating some of these decisions to prevent the need for expert advice.

Chapter 5

Streaming Data Ingestion

One of the core use-cases for transactional streaming is the opportunity it opens for streaming data ingestion. In this chapter, we explore how distributed transactional streaming can lead to the implementation of a new data ingestion architecture.

Data ingestion is the process of getting data from its source to its home system as efficiently and correctly as possible. This has always been an important problem and has been touched on by many previous research initiatives like data integration, deduplication, integrity constraint maintenance, and bulk data loading. This process sometimes goes under the name of Extract, Transform, and Load (ETL) [95, 154].

Modern applications, however, put new requirements on ETL. Traditional ETL is constructed as a pipeline of batch processes, each of which takes its input from a file and writes its output to another file for consumption by the next process in the pipeline, etc. The reading and writing of files is cumbersome and very slow. Older applications, such as data warehouses, were not that sensitive to the latency introduced by this process. They did not need the most absolutely current data. Newer applications like IoT (Internet of Things), on the other hand, seek to provide an accurate model of the real world in order to accommodate real-time decision making.

For modern applications in which latency matters, ETL should be conceived as a streaming problem. New data arrives and is immediately handed to processing elements that prepare data and load it into a DBMS. We believe that this requires a new architecture and that this architecture places some fundamental requirements on the underlying stream processing system. An analytics system for demanding applications like IoT are based on the existence of a data store that captures a picture of the world that is as accurate as possible. If correctness criteria are not met, the contents of the analytics system can drift arbitrarily far from the true state of the world.

5.1 Modern Data Ingestion Workloads

5.1.1 Internet of Things (IoT)

There is a strong need to support real-time data ingestion, particularly for demanding new applications such as IoT. Many of the standard problems of data ingestion (like data cleaning and data integration) remain, but the scale at which these tasks must operate changes the solution.

Take for instance self-driving vehicles as an example of an IoT deployment. Today's cars have many on-board sensors such as accelerometers, position sensors (e.g. GPS, phone), fuel consumption sensors and at least one on-board computer. In the future, cars will also be equipped with communication ability to send messages to other vehicles or to the cloud via sophisticated middleware (streaming system). In cities, it is easy to imagine that this may scale to over a million cars. Such a system will need to offer many services, including for example a warning and a list of nearby gas stations when the fuel tank level is below some threshold.

In this situation, it is easy to see why the traditional data integration process is not sufficient. The value of sensor data decreases drastically over time, and the ability to make decisions based on that data is only useful if the analysis is done in near real-time. There is a necessity to maintain the order of the time-series data, but to do so in a way that does not require waiting hours for a large batch to become available. Additionally, time-series data can become very large very quickly, particularly if sensor sample rates are high. This storage can become extremely expensive, and it is likely that the entirety of the time-series data does not need to be stored to extract the relevant analytics.

We postulate that with a completely fresh look at the data integration process, the analytics, cleaning, and transformation of this time-series data can all be performed by one system in near real-time.

5.1.2 Bulk-Loading ETL

While sensor data and other streaming data sources are a natural use-case, we believe that streaming ETL can have benefits for traditional data ingestion as well. One example is the retail brokerage firm application, streaming TPC-DI, which was described in Section 4.3.2. These types of workloads typically involve the extraction and transformation of data from a variety of sources and source formats (e.g. CSV, XML, etc.), and are usually processed in large batches. Once processed, all data is accessible within the data warehouse. The full data ingestion stack is illustrated in Figure 5.1.

Obviously, there is usually very heavy latency associated with this type of process. Frequently, data arrives in batches only a single time a day, and all data must be available before processing can

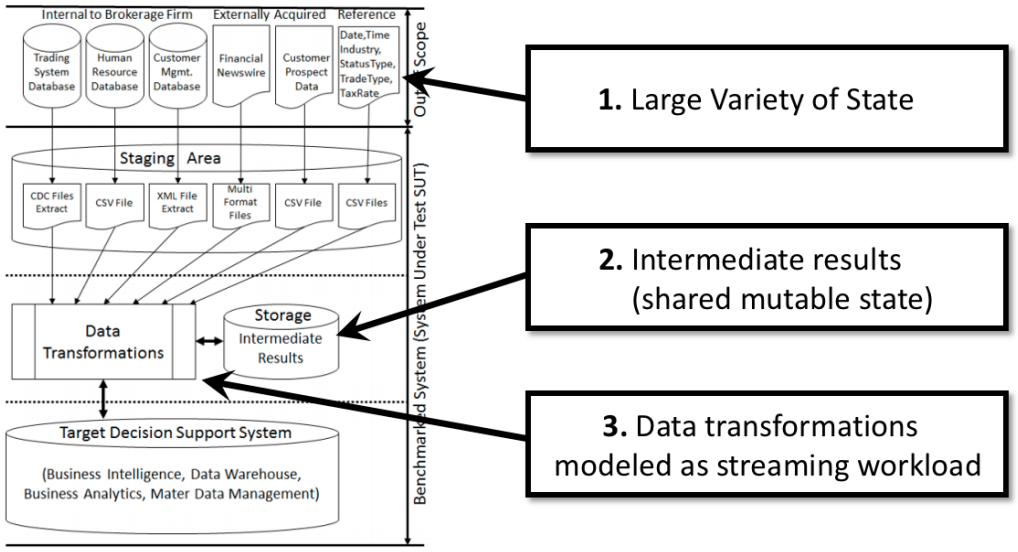


Figure 5.1: Typical Data Ingestion Stack from [129] (and potential applications of streaming)

begin. Incrementally processing data as it arrives in a streaming fashion can potentially have major performance benefits.

The main concern when transforming these processes into streaming workloads is the potential loss of state correctness. Without ACID transactional data mutation, there is a high likelihood of losing state consistency over time. With this problem eliminated by transactional streaming, it becomes a much more viable option.

Dividing ETL processing into small, ordered transactions has the added benefit of making the process more granular for the purposes of distributed processing. Ordinarily, ETL operations update state to a variety of relational tables, and may do so in single, large transactions. However, these transactions can require look-ups on data that are sharded on different keys, and thus likely require a large cross-node transaction in a distributed system. Because of its push-based ordered processing, streaming ETL could allow these large distributed transactions to be broken into several small single-sited transactions without damaging correctness.

5.1.3 Polystores

In the past few decades, database systems have gradually become more and more specialized for highly specific workloads. More recently, a trend has begun towards unifying multiple big data systems under a singular umbrella and unified data model. We call such architectures *polystores*. One major point of research in polystores is the notion of unified data ingestion and catalog maintenance.

Transactional streaming is ideal for such a use-case. This will be further discussed in Section 5.3.

5.2 Rethinking Data Ingestion

5.2.1 Requirements for Streaming ETL

As with any data integration system, streaming ETL must first-and-foremost be concerned with the correctness and predictability of its results. Simultaneously, a streaming ETL system must be able to scale with the number of incoming data sources and process data in as timely a fashion as possible. With these goals in mind, we can list the requirements for a streaming ETL system, which draw from elements of traditional ETL requirements, streaming requirements, and infrastructure/state management requirements (summarized in Table 5.1).

Data Collection

A typical example of an ETL workload generally includes a large number of heterogeneous data sources that may have different schemas and originate from a variety of sources. In the case of streaming data sources, data must be collected, queued, and routed to the appropriate processing channel. A data collection mechanism should have the ability to transform traditional ETL data sources (e.g. flat files) into streaming ETL sources. In this case, the traditional data sources need to be collected by streaming clients that can batch and create input streams. Data collection should scale with the number of data sources to avoid losing information or becoming a bottleneck for processing.

In addition to simply collecting data, some of the data cleaning computation can be pushed to the data collection network. For example, in an IoT application that collects data over a large network of sources, the gateway routers and switches can help with some of the computation via filtering and smoothing of signal data [135]. Another option when router programming is not readily available is to use field-programmable gate arrays (FPGAs) for the same functionality [41]. Some network interface cards also support embedded FPGAs. Even though the computational capabilities and memory sizes of these edge programming nodes are very limited, for large-scale applications such as IoT, the benefit of using this method is quick scalability with the network size.

Bulk Loading

It has long been recognized that loading large amounts of data into a DBMS should not be done in a tuple-at-a-time manner. The overhead of SQL is too high; furthermore, indexing the new data presents a significant issue for the warehouse.

While the ETL processes are revised to a streaming architecture, it must also support mutable shared storage (as mentioned below). Therefore, it should also have the ability to take over some

Requirement	Description	Trad. ETL	Streaming	State Mgmt
Data Collection	Collection and routing of data	yes	yes	no
Bulk Loading	Import data at high bandwidth	yes	no	yes
Out of Order	Timeseries arriving out-of-order	no	yes	no
Stream Order	Dataflow and batch ordering	no	yes	no
Exactly-Once	No loss or duplication	yes	yes	no
Local Storage	Staging and caching	yes	no	yes
ACID Txns	Full ACID state mutation	yes	no	yes
Heterogeneity	Variety in collection and storage	yes	no	yes
Scalability	Ability to grow with data	yes	yes	yes
Data Freshness	Low-latency ingestion	no	yes	yes

Table 5.1: Requirements for Streaming ETL and Their Origins

of the responsibility of the warehouse. For instance, it can store (cache) some of the data that is computed on the input streams. This data can be pulled as needed into the warehouse or as a part of future stream processing. For example, it is possible to store the head of a stream in the ETL engine and the tail in the warehouse. This is largely because recent data is more likely to be relevant for data ingestion than older data.

Another tactic to increase the overall input bandwidth of the system is to suppress the indexing and materialized view generation of the warehouse until some future time. The warehouse would have to be prepared to answer queries in a two-space model in which older data is indexed and newer data is not. This would require two separate query plans. The two spaces would be merged

periodically and asynchronously.

Out of Order Tuples

This topic relates to data cleaning, but has particular importance to IoT. Whenever one has many millions of devices talking simultaneously, it becomes very difficult to guarantee that the data's arrival order corresponds to the actual order of data generation. Tuples can be out of time-stamp order or they can be missing altogether. Waiting for things to be sorted out before proceeding can introduce an unacceptable level of latency.

One problem that arises in a network setting with disorder in tuple arrival is determining when a logical batch of tuples has completely arrived. In such a setting, some earlier solutions required that the system administrator specify a timeout value [14]. Timeout is defined as a maximum value for how long the system should wait to fill a batch. If the timeout value is exceeded, the batch closes and if any subsequent tuples for this batch arrive later, they are discarded. We can imagine that for streams that represent time-series (the vast majority of streaming sources in IoT), if a batch times out, the system could predict what that batch would look like based on historical data. In other words, we can use predictive techniques (e.g., regression) to make a good guess on the missing values.

Dataflow Ordering

As previously stated, data ingestion is traditionally accomplished in large batches. In the interest of improving performance, streaming data ingestion seeks to break large batches into much smaller ones. The same goes for operations; rather than accomplishing data transformation in large transactions, streaming data ingestion breaks operations into several smaller operations that are connected as a user-defined dataflow graph.

In order to ensure that these smaller operations on smaller batches still produce the same result as their larger counterparts, ordering constraints need to be enforced on how the dataflow graph executes. Streaming data management systems are no stranger to ordering constraints. Intuitively, batches must be processed in the order in which they arrive. Additionally, the dataflow graph must execute in the expected order for each batch. These constraints should be strict enough to ensure correctness while also providing enough flexibility to achieve parallelism.

Exactly-Once Processing

When a stream-processing system fails and is rebuilding its state via replay, duplicate tuples may be created to those generated before the failure. This is counter-productive and dangerous. Data ingestion attempts to remove duplicate records, and our system should not insert the very thing that deduplication addresses. Similarly, it is expected that no tuples are lost during either normal operation or failure.

Exactly-once guarantees apply to the activation of operations within a dataflow graph, as well as the messaging between engines that comprise the streaming ETL ecosystem. In either case, the absence of exactly-once guarantees can cause a loss or duplication of tuples or batches.

Local Storage

Any ETL or data ingestion pipeline needs to maintain local storage for temporary staging of new batches of data while they are being prepared for loading into the backend data warehouse. For example, in an IoT use case, a large number of streaming time-series inputs from distributed sources may need to be buffered to ensure their correct temporal ordering and alignment. Furthermore, in a distributed streaming ETL setting with multiple related dataflow graphs, there will likely be a need to support shared in-memory storage. While this raises the need for transactional access to local storage, it can also potentially provide a queryable, locally consistent view of the most recent data for facilitating real-time analytics at the OLAP backend.

In addition to temporary staging of new data, local storage may also be required for caching older data that has already made its way into the data warehouse. For example, in our TPC-DI scenario, each incoming batch of new tuples requires look-ups in several warehouse tables for getting relevant metadata, checking referential integrity constraints, etc. Performing these look-ups on a local cache would be more efficient than retrieving them from the backend warehouse every time.

ACID Transactions

ETL processes are fundamentally concerned with the creation of state. Incoming tuples are cleaned and transformed in a user-defined manner, and the output is assumed to be consistent and correct. A streaming ETL engine will be processing multiple streams at once, and each dataflow instance may try to make modifications to the same state simultaneously. Additionally, as previously discussed, staged tuples may be queried by the outside world, and cached tuples are maintained from a data warehouse. In all of these cases, data isolation is necessary to ensure that any changes do not conflict with one another.

It is expected that the atomicity of operations is maintained. ETL is executed in batches, and it would be incorrect to install a fraction of the batch into a data warehouse. Additionally, all state must be fully recoverable in the event of a failure. ACID transactions provide all of these guarantees, and are a crucial element to any ETL system.

Heterogeneity

Modern data ingestion architectures should provide built-in support for dealing with not only heterogeneous data sources, but also diverse target storage systems. While the former is a well-known problem and has been addressed by traditional ETL and data integration solutions, it is more challenging to apply these solutions at the scale of a large number of streaming data sources as in IoT.

Furthermore, the heterogeneity of the storage systems in today's big data ecosystem has led to the need for using multiple disparate backends or federated storage engines (e.g., the BigDAWG polystore [61]). The presence of multiple, heterogeneous targets calls for a data routing capability within the streaming ETL engine. Furthermore, if semantically-related batches are being loaded to multiple targets, it may be critical to coordinate their loading to help the data warehouse maintain a consistent global view.

Scalability

Most modern OLAP systems scale roughly linearly in order to accommodate the increasing size of datasets. Ideally, data ingestion should scale at the same rate as the OLAP system in order to avoid becoming a bottleneck. It is important that the data ingestion also be able to keep up with increasing quantities of data sources and items. This means providing the ability to scale up processing across many nodes and accommodating a variable number of connections. Disk and/or memory storage must also be able to scale to suit expanding datasets.

Data Freshness and Latency

One of the key reasons to develop a streaming ETL system is to improve the end-to-end latency from receiving a data item to storing it in a data warehouse. When running analytical queries on the data warehouse, we take into account the freshness of the data available in the warehouse. Data freshness can be measured with respect to the most recent data available to queries run in the data warehouse. The more frequently new data arrives, the fresher the warehouse data is. If the most recent data is only available in the data ingestion cache, a query's data freshness can be improved by pulling that data as it begins.

The end-to-end latency to ingest new data items is also related to data freshness. Refreshing the data frequently in the data warehouse will not help unless new data items can be ingested quickly. Often, achieving the best possible latency is not as crucial as obtaining an optimal balance between high throughput within a reasonable latency bound. Latency bounds can be variable degrees of strict. For example, it may be important that time-sensitive sensor data be processed immediately, as its value may quickly diminish with time. Traditional ETL, on the other hand, frequently has more relaxed latency bounds.

Together, these requirements provide a roadmap for what is expected from streaming ETL. Next, we explore a generic architecture to build a streaming ETL system for a variety of uses.

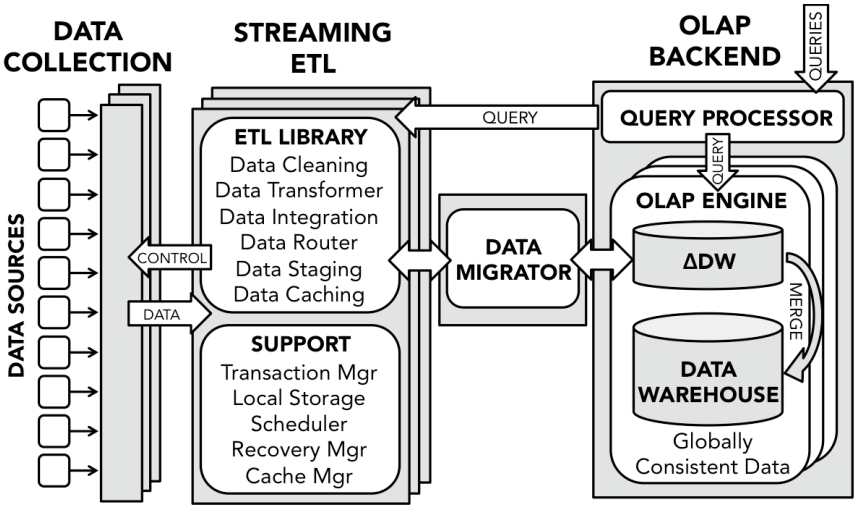


Figure 5.2: Streaming ETL Architecture

5.2.2 Streaming ETL System

The architecture of a streaming ETL engine is illustrated in Figure 5.2. We envision a three-tiered architecture for handling the ETL process, comprised of three primary components (data collection, streaming ETL, OLAP backend), as well as the durable migration between them.

Data Collection

In an IoT workload, one expects data to be ingested from thousands of different data sources at once. Each data source submits new tuples onto a stream (likely sending them through a socket), which are then received by a data collector mechanism. This data collector primarily serves as a messaging queue. It must route tuples to the proper destination while continuously triggering the proper ETL process as new data arrives. Additionally, the data collector must be distributed, scaling to accommodate more clients as the number of data sources increases. Fault tolerance is also required to ensure that no tuples are lost during system failure.

The data collector is responsible for assigning logical batches of tuples, which will be consumed together by the ETL engine. While time-series order is naturally maintained with respect to each data source, global ordering can be much more tricky in the presence of thousands of streams. It is the data collector's responsibility to ensure that the global ordering of the tuples is maintained.

Kafka is a good initial choice for handling the messaging infrastructure in our initial implementation. Apache Kafka is a highly-scalable publish-subscribe messaging system able to handle thousands of clients and hundreds of megabytes of reads and writes per second [96] . Kafka's combination of availability and durability makes it a good candidate for our data collection mechanism,

as it is able to queue new tuples to push to the streaming ETL engine.

Presently, in our implementation, Kafka serves exclusively as a messaging queue for individual tuples, each of which are routed to the appropriate dataflow graph within the streaming ETL engine. In contrast to our architecture description, all batching currently takes place within the streaming ETL component. In future iterations of the data ingestion stack, the data collection component can be extended to handle more complicated message handling, including the batching of near-simultaneous tuples and perhaps simple filtering operations.

Streaming ETL

Once the data has been batched, it is pushed to the streaming ETL engine. The streaming ETL engine features a full library of traditional ETL tools, including data cleaning and transformation operators. Following a user-defined dataflow graph of operators, incoming batches are massaged into normalized data ready for integration with a data warehouse. Frequently, reference will need to be made to existing data in the warehouse (e.g., to look-up and assign foreign keys). For these instances, the streaming ETL engine requires the ability to cache established data from the warehouse, as constant look-ups from the warehouse itself will quickly become very expensive.

Once the data has been fully cleaned and transformed, it remains staged in the streaming ETL engine until the data warehouse is ready to receive it. Because there may be multiple data warehouses storing a variety of data types, the streaming ETL engine must be able to route outgoing data to the appropriate warehouse, much like the data collection mechanism routes its output. Additionally, the streaming ETL engine must be scalable to support expanding amounts of data, and fault-tolerant to ensure that its results are recoverable in the event of a failure.

In addition to the ETL library, the streaming ETL engine also requires various support features found in most databases. For instance, to ensure consistency of the outgoing data, transactions are required. Local storage is needed for both staging and caching, and dataflow scheduling and recovery must be managed within the engine.

Transactional streaming in general, and S-Store in particular, is an ideal choice for the Streaming ETL component. It provides the dataflow ordering and exactly-once guarantees of stream processing, while also handling all of the ACID state management locally. Additionally, it is queryable from the outside world, and able to scale with the amount of data it receives.

OLAP Backend

The OLAP backend consists of a query processor and one or several OLAP engines. The need to contain several OLAP systems is rooted in variations in data type; some data are best analyzed in a row-store, some a column-store, some an array database, etc. Each OLAP engine contains its own data warehouse, as well as a delta data warehouse which stores any changes to be made to

the dataset. The delta data warehouse contains the same schema as the full warehouse, but may be missing indexes or materialized views in order to allow for faster ingestion. The streaming ETL engine writes all updates to this delta data warehouse, and the OLAP engine periodically merges these changes into the full data warehouse.

The OLAP backend also requires a query processor, preferably one that is able to access many of the underlying OLAP engines. If this is not possible, multiple query processors may be needed. When a data warehouse is queried, the corresponding delta table must also be queried (assuming that some of the results are not yet merged). Potentially, if the user is looking for the most recent data, the query processor may query the staging tables in the streaming ETL engine as well. The user should have the ability to choose whether to include staged results, as they may affect query performance.

Clearly the OLAP backend depends on the type of data being stored, but in the general case, a *polystore* is ideal for this component. More on polystores in the next section.

Durable Migration

Batches are frequently moved between the Data Collection, Streaming ETL, and OLAP Backend components, and mechanisms are needed to ensure that there is no data lost in transit. At the bare minimum, any failure as a batch is moved between components should result in the data being rolled back and restored to its original location. Additionally, the migration mechanism should be able to support the *most* strict isolation guarantees of its components. We believe that ACID state management is crucial for a Streaming ETL component, and therefore the migration mechanism should support fully ACID transactions.

Migration is expensive, and it is important that it primarily takes place at a time when the individual components are not overloaded. Take an example of migrating data between the Streaming ETL and OLAP Backend components. The OLAP Backend may frequently handle long-running queries that are both disk and CPU intensive. While the delta data warehouse can help with this, it is important to coordinate migration around those queries. Similarly, there may be scenarios in which the Streaming ETL engine is overworked. Developers should have the option to implement either a push- or pull-based migration model, depending on the structure of their ETL workload. If the Streaming ETL is the bottleneck, or if fresh data is a high priority, then new data should periodically pushed to the OLAP backend when it is ready. If long-running OLAP queries are the priority, then the backend should pull new data when there is downtime.

In our implementation, durable migration is provided via the messaging connection between S-Store and BigDAWG, our polystore OLAP backend.

5.3 Real-time Data Ingestion for Polystores

Big data problems are commonly characterized along multiple dimensions of complexity including volume, velocity, and variety. Earlier system solutions focused on each individual dimension separately, targeting different classes of computations or data types (e.g., batch/OLAP [2] vs. real-time/streaming [151], graphs [107] vs. arrays [40]). This led to a heterogeneous ecosystem, which has become difficult to manage for its users in terms of programming effort and performance optimization. Furthermore, large-scale big data applications rarely involve a single type of data or computation (e.g., [128]). As a result, integrated architectures (e.g., [10, 7]) and new hybrid systems (e.g., [4, 47, 19, 67]) have started emerging.

The polystore architecture and its first reference implementation BigDAWG represent a comprehensive solution for federated querying over multiple storage engines, each possibly with a different data and query model or storage format, optimized for a different type of workload [61]. One of the main design principles of BigDAWG is that it tightly integrates real-time and batch processing, enabling seamless and high-performance querying over both fresh and historical data.

We envision an architecture where all new data enters the polystore as a stream. Streams can arrive from multiple sources, at high rates, and must be reliably and scalably ingested into the system on a continuous basis. During this ingestion phase, various transformations that prepare the data for more sophisticated querying and storage can be applied. For example, raw input streams may be merged, ordered, cleaned, normalized, formatted, or enriched with existing metadata. The resulting streams can then be used as inputs for immediate, real-time analytics (e.g., detecting real-time alerts) and can be loaded to one or more backend storage systems for longer-term, batch analytics. Meanwhile, the polystore continues to process interactive queries that may involve both newly ingested and older data. Therefore, it is highly important that these queries can see a complete and consistent view of the data in a timely manner, regardless of where it is actually stored.

We believe that a stateful stream processing system with transactional guarantees and multi-node scalability support is a good fit for addressing the real-time processing requirements of a polystore system discussed above. Each ETL workflow can be represented as a dataflow graph consisting of ACID transactions as nodes and streams flowing between them as edges. Input streams chunked into well-defined atomic batches are processed through these dataflows in an orderly and fault-tolerant manner. The resulting output batches can then be incrementally loaded to backend stores with transactional guarantees. Furthermore, S-Store has its own in-memory storage engine that can handle adhoc queries and traditional OLTP transactions over shared tables, thereby providing consistent and fast access to most recent data and materialized views derived from it. S-Store's fast transactional store feature also enables unique optimizations in the BigDAWG polystore such as

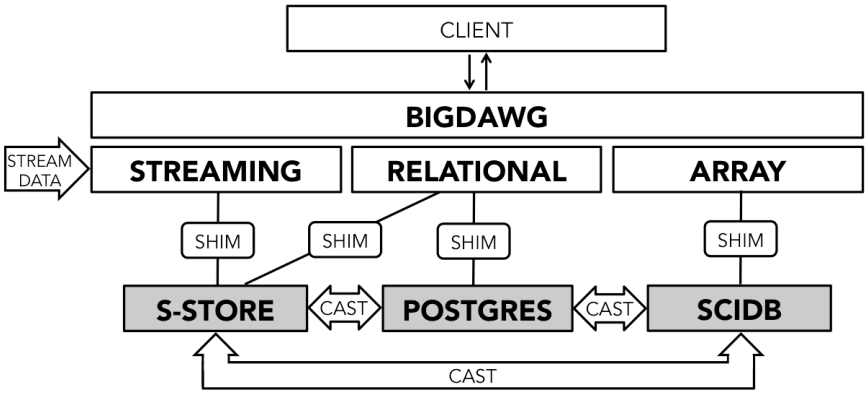


Figure 5.3: BigDAWG 1.0 Architecture

caching and anti-caching. Similarly, BigDAWG's extensible, island-based architecture enables the use of S-Store together with other messaging or streaming systems in a federated manner if needed.

5.3.1 BigDAWG Architecture and Streaming Island

The BigDAWG polystore system unifies multiple systems with varying use cases and storage requirements under a single architecture [61]. It is founded on the idea that "one size does not fit all," and thus unifying many different storage engines is the best method of handling a variety of specific tasks. The architecture of BigDAWG is illustrated in Figure 5.3.

Because these systems use very different query languages, BigDAWG uses a construct called *islands of information*, which are front-facing abstractions including a query language, data model, and *shims*, a set of connections to the underlying storage engines. Each island represents a category of database systems; for example, a relational island may contain traditional database systems such as Postgres or MySQL, while an array island may contain multi-dimensional array databases such as SciDB [40]. Individual systems may be included in multiple islands, if they fall under multiple categories.

Individual systems under BigDAWG may migrate data between one another using a *cast* operator. These operators transform the data from the host system into a serialized binary format which can then be interpreted by the destination system. The cast operator allows for an efficient method of moving data to a specific storage engine that may be more efficient at carrying out the operation. For instance, if the user is looking to join a table from Postgres to a SciDB array, it may be most efficient to migrate the array into Postgres (transforming it in the process) and perform the join there.

As with other data types, BigDAWG must be able to manage incoming streaming data, and

should provide the user with a unified method of querying those data streams. In addition to S-Store, BigDAWG should be able to support other contemporary streaming data management systems such as Spark Streaming [168] or Apache Storm [151]. As is the case with other categories of data types, BigDAWG has need of a streaming island in order to manage the unique needs of streaming data.

Due to the nature of streaming data, the streaming island must be substantially different than the islands described previously. While most islands are pull-based in nature, streaming island is inherently push-based. Multiple data sources can be connected to this streaming island, as well as multiple stream ingestion systems. One of the primary functions of BigDAWG's streaming island should be to direct streaming data into the proper ingestion system(s). In this way, streaming island serves as a publish-subscribe messaging module, and should perhaps be partially implemented using an engine that specializes in scalable messaging, such as Apache Kafka [96].

The second functionality required by streaming island is the ability to view and pass results from continuous queries. To propagate the push-based nature of streams, streaming island must be able to trigger other operations, including pull-based operations from non-streaming systems. One simple example of such an operation is a user-facing alert. Take, for instance, a MIMIC medical application that is monitoring heart rate in real time. If conditions are met that indicate abnormalities in the heart rate, the streaming application may need to send an alert to a doctor.

In addition to the push-based functionality, other non-streaming systems may need to be able to poll the results of a continuous query at any time. The streaming island should facilitate this as well, either by temporarily storing the results of the query, or simply serving as a pass-through for the pull-request to the appropriate streaming system.

5.3.2 Push vs Pull in ETL Migration Queries

Cross-system data storage and management is an obvious challenge in polystores. If the same data items are needed in separate queries, each of which can be best handled by different systems, then where should those data items be located for the best possible performance?

Streaming ETL is a strong example of such a problem. Let's consider a situation in which S-Store is performing streaming ETL for Postgres (illustrated in Figure 5.4). Ideally, S-Store is able to perform transformations on incoming data independently of Postgres. However, the transformation process will frequently require referencing existing data within the target system. An S-Store query that requires Postgres data can be executed in one of a few possible ways:

1. **Cross-System Query** - The required data remains in Postgres. The S-Store query must be executed as a cross-system transaction that accesses the target data a single time, and immediately forgets it once the transaction commits. This is very expensive, especially if a similar

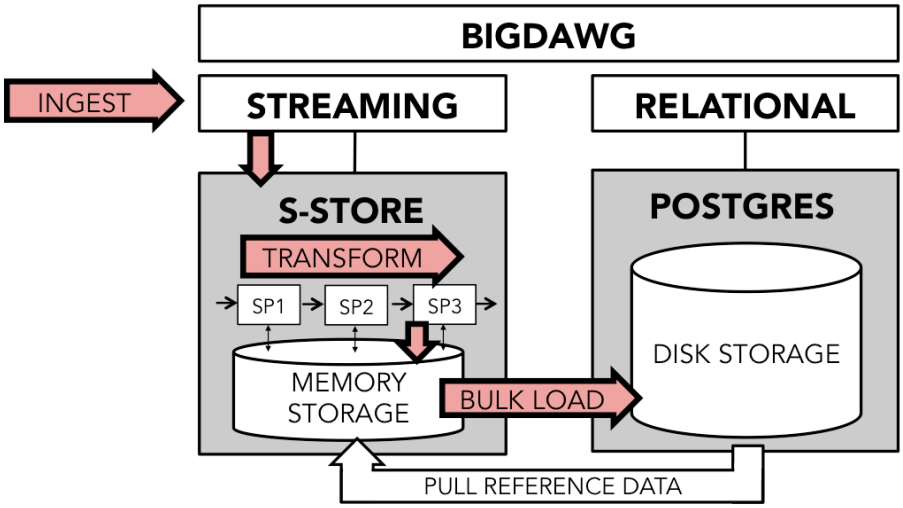


Figure 5.4: Streaming ETL Example

query will be run in the near future.

2. **Replication** - The required data is cached in S-Store from Postgres. After the copy is made, the S-Store query can be run locally and is inexpensive. However, maintaining the correctness of the S-Store cached copy is expensive in the event that the required data is modified in a Postgres query.

3. **Migration** - The required data can be moved into S-Store (and removed from Postgres). The S-Store query can be run locally and is inexpensive, especially in the event of repeated queries on the same data. However, if a Postgres query requires access to the data, it will need to be run as a cross-system query.

As this example illustrates, there are three primary solutions to the data locality problem: replication, migration, and cross-system querying. Each solution comes with benefits and drawbacks, and the optimal approach will always depend on the specific case. Further research has been put into exploring these trade-offs, and developing a cost-model which quantifies the options and informs a query planner about which approach is ideal for a given situation [59].

The introduction of streaming ETL and cross-system caching brings up important questions: what is the cost of moving data between systems? Is it more expensive to periodically update a copy of the data in another system, or to pull data across systems each time it is needed? Does pushing some of the query-planning into the island level improve cross-system query performance?

In order to compare potential query plans, we have constructed a simple experiment that compares two query plans for a UNION query. Let's assume that S-Store is being used as an ingestion

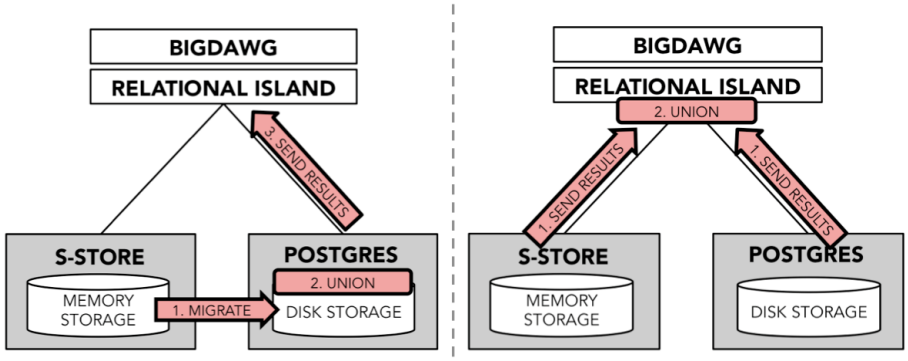


Figure 5.5: Migration Query Plan vs. UNION in Island Query Plan

engine for the ORDERS table of the TPC-C workload, eventually migrating tuples into Postgres [1] [149]. A user wishes to run a full table scan on the ORDERS table, using the simple query "SELECT * FROM ORDERS." Because S-Store is ingesting tuples into the ORDERS table and incrementally sending them to Postgres, a percentage of the ORDERS table lives in each system. To accomplish a full table scan on ORDERS, the tuples in S-Store must be combined with the tuples from Postgres, effectively making the query "(SELECT * FROM S-Store.ORDERS) UNION (SELECT * FROM Postgres.ORDERS)." Two methods of executing the UNION query (illustrated in Figure 5.5) are:

i. migrate the data from S-Store to Postgres, and perform the UNION in Postgres, or
ii. pull all resulting tuples into the Relational Island, and perform the UNION there.

This experiment was run on an Intel® XEON® processor with 40 cores running at 2.20 GHz. S-Store was deployed in single-node mode. Migration from S-Store to Postgres is implemented as a binary-to-binary migration. It is assumed that the pipe connection between the two systems is already open, and thus pipe set up time is ignored in the migration results. Queries are submitted to both systems via JDBC. A total of 200,000 tuples were contained within the ORDERS table, a percentage of which were stored in S-Store and the rest in Postgres.

As can be seen in Figure 5.6, the most efficient query plan depends on the amount of data being held in the S-Store and Postgres tables. The cost of migrating tuples from S-Store to Postgres increases linearly with the number of tuples being transferred. If 25% or fewer tuples are held in the S-Store table, then it is more efficient to migrate the tuples into Postgres and do the UNION there. However, by executing in this way, the data from S-Store is effectively being moved twice: once to Postgres, and then again to the client. Thus, if more than 25% of the tuples are in S-Store, then it becomes faster to instead transfer all results to the Relational Island and do the UNION there. This has the added benefit of being able to pull results from both systems in parallel. As a result, the

[1] While TPC-DI is a more realistic workload for streaming ETL than TPC-C, results were unavailable as of this writing.

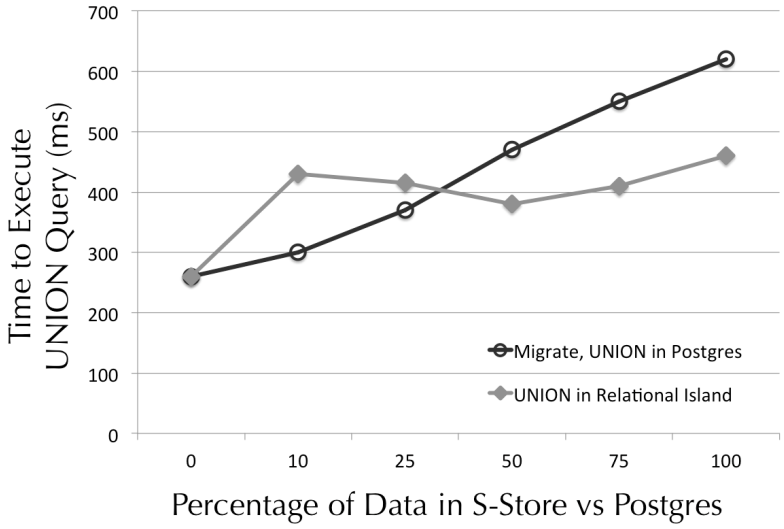


Figure 5.6: Migration Evaluation for UNION Queries

optimal query performance for this approach falls at a 50/50 data distribution between S-Store and Postgres.

There are additional aspects to consider with these preliminary results. For instance, if the query is repeated on a regular basis, then it becomes more efficient to migrate the tuples into Postgres, even if the initial migration is more expensive. In the case of streaming ETL, incremental loading is an effective method of spreading the cost of the migration over time and providing quicker access for Postgres queries.

Also note that the UNION operator is relatively inexpensive to perform. In the case of UNIONing within the relational island, result sets only need to be stored long enough to be concatenated and sent to the client. More complicated query plans, including joins and nesting, will increase the complexity of processing required within the Relational Island. It is likely that it is more efficient to perform complex queries within a mature specialized system such as Postgres, even if it means migrating large amounts of data. We will explore these kinds of issues in more detail as part of our ongoing research.

5.4 Conclusions

This chapter makes the case for streaming ETL as the basis for improving how up-to-date the data in a warehouse is, even when the implied ingestion rate is very high. We have discussed the major functional requirements such a streaming ETL approach should address, including embedded local

storage and transactional guarantees. We have then described a system architecture designed to meet these requirements using a transactional stream processing system as its base technology for ingestion.

Additionally, we described a polystore called BigDAWG and the role of a streaming engine in BigDAWG. S-Store supports transactional guarantees, making the system much more reliable for managing shared state. This prevents inconsistent updates from working their way into the other storage systems that constitute BigDAWG. We have briefly described how S-Store can also act as an ETL system, providing services such as data cleaning, data integration, and efficient data loading.

Chapter 6

Preliminary Work: Automatic Database Design

The distribution techniques of Chapter 4 and the target use cases of Chapter 5 strike a delicate balance between complicated workloads and the tools with which to scale them outward with high performance. Optimal database design is not necessarily intuitive, and performance will suffer greatly without it. Ideally, much of the database design is done automatically for the user, saving him or her a large amount of time and energy.

In this chapter, we explore preliminary research in automatic database design. We define what fundamental principles apply to good design in Section 6.1. We devise a model that represents the relationship between state and processing in Section 6.2. We then use that model to form a cost model for designs in Section 6.3 and then apply that cost model to an automated partitioning algorithm in Section 6.4

6.1 Principles of Good Design for Transactional Streaming

As with any database management system, there are principles of good design that must be followed in order to make the best use of the available resources. While simple designs can be created manually by hand, the addition of many stored procedures and shared state can quickly create an extremely large search space which becomes unwieldy even for an expert. To achieve best-possible scalability, database administrators (DBAs) need to keep in mind these best practices for placing both processing and state onto physical nodes in a way that achieves the best possible throughput.

6.1.1 State Co-Location

As with traditional shared-nothing database systems, the communication cost of moving data across nodes is the major bottleneck to scalability. Distributed transactions that request data across S-Store partitions are extremely expensive. In this respect, S-Store is very similar to any shared-nothing system, and data co-location is crucial to overall workload performance, as shown in Figure 6.1. If a transaction accesses multiple different types of state, it will execute much more efficiently if all of the state it needs lives on a single partition.

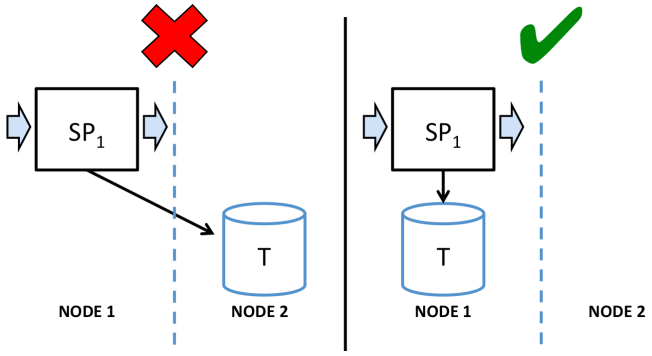


Figure 6.1: State Co-Location

Because S-Store dataflow graphs are composed entirely of user-defined stored procedures, nearly the entire workload is highly predictable (with the exception being ad-hoc OLTP transactions). Ideally, the entire workload is comprised of locally-run transactions such that data access across nodes is unnecessary. It is impossible to completely remove distributed transactions from the workload, but they can be minimized by co-locating state. There are a number of established techniques for co-locating state in a shared-nothing system (described in Section 7.7); the novelty of the challenge with respect to S-Store is weighing the cost of state co-location with other optimization concerns.

6.1.2 Processing Co-Location

In addition to remote data access, S-Store must also consider a second communication cost: the cost of triggering a downstream stored procedure and moving stream data from one site to another. This communication cost is very similar to the cost considered by traditional distributed stream processing systems, described in Section 7.7. It is directly related to the amount of data transferred over the stream (i.e. the selectivity of the stream) and the frequency with which the transfer takes place. As such, it is less expensive to co-locate two consecutive stored procedures in a dataflow graph on the same node, as shown in Figure 6.2.

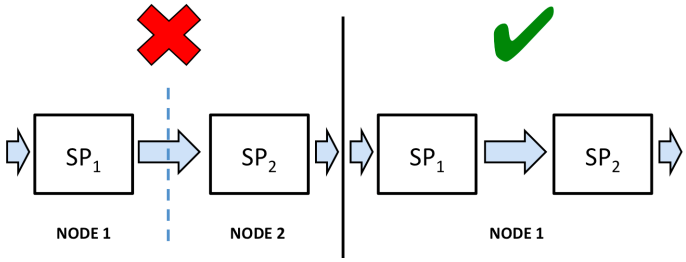


Figure 6.2: Processing Co-Location

MOVEs are specialized distributed transactions, with full transactional properties (most relevantly, fault tolerance). However, unlike the distributed transactions involved with remote data access, the MOVE operation can be heavily optimized as described in Section 4.5. This is due to the append-only nature of streams, and the exactly-once guarantees that S-Store provides. Again, the process of distributing a streaming data management system is a well-studied problem; the novelty introduced with S-Store is the consideration of state in addition to the processing.

6.1.3 Load Balancing

In S-Store's system design, because transactions are assigned to specific nodes, the workload bottleneck will be determined by whichever node and partition is most heavily overworked. While co-location is important to efficient operation, even distribution of the processing load is also crucial. In a distributed streaming system, a dataflow graph is only able to achieve the maximum throughput equal to the slowest operator in the chain. The maximization of the overall throughput of the workload requires the minimization of the processing bottleneck, as shown in Figure 6.3.

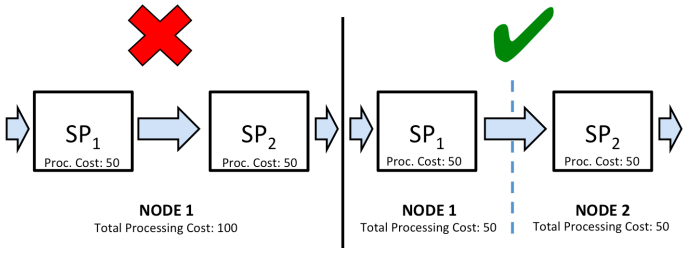


Figure 6.3: Load Balancing

There are two keys to minimizing the processing bottleneck, both of which rely on the natural predictability of a dataflow graph workload. First, assuming each transaction requires a predictable amount of processing time, each one can be assigned to a node in such a way that maximizes the processing potential of available resources. In addition to balancing processing, balancing state

placement across nodes is also necessary. Load balancing via state placement is particularly beneficial for handling skew; isolating "hot" data items (i.e. data items that are frequently accessed) can better distribute processing for the operations that access those items.

Frequently, load balancing is directly at odds with both state and processing co-location. Improving one may negatively impact the other. Thus, it is important that each is properly weighted when evaluating the best database design.

6.1.4 Dataflow Partitioning (Pipeline Parallelism)

When designing the database, it is important to consider not just the assignment of state and processing, but also the design of the dataflow graph itself. S-Store dataflows allow for transactions to trigger one another, but do not force triggered transactions to execute immediately. This allows for flexibility in both the assignment of processing across nodes and increases the opportunities for parallelism. Each transaction is made up of one or more operations, which may or may not access state. If these operations must be performed atomically, then they belong together in a transaction. If the atomicity between them is more flexible, however, then they can potentially be divided into two or more transactions, as shown in Figure 6.4.

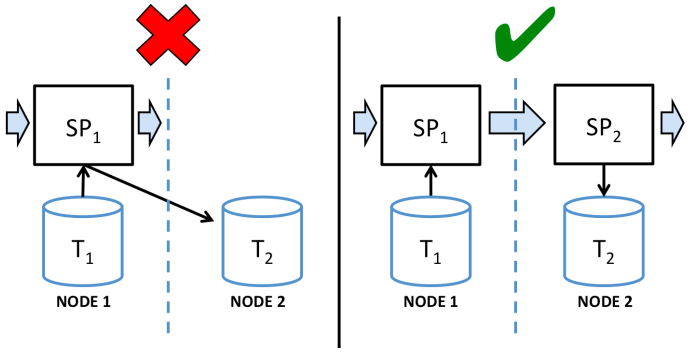


Figure 6.4: Dataflow Partitioning

Dividing operations into as many transactions as possible improves the possibility for pipeline parallelism within S-Store. Additionally, the less frequently state is accessed by a single transaction, the less likely it is that the transaction will need to be distributed across multiple partitions. It follows that when designing a dataflow graph, it is prudent to make each transaction as lightweight as possible in order to provide more scheduling options. S-Store can always execute two sequential transactions serially, but it cannot divide a single transaction in an attempt to improve performance. Separating operations into as many transactions as possible also improves the modularity of the

dataflow graph, allowing for flexibility in assigning and co-locating state and processing (discussed further in Section 4.3.3).

6.1.5 Procedure Partitioning (Data Parallelism)

When designing a distributed system, pipeline parallelism can only take you so far. In order to achieve near linear scalability, data parallelism is usually required. The difficulty in parallelizing the processing within S-Store is that ACID transactions are very rigid in their execution. How do we employ data parallelism for transactions that are executed on large batches that access a variety of partitioned data?

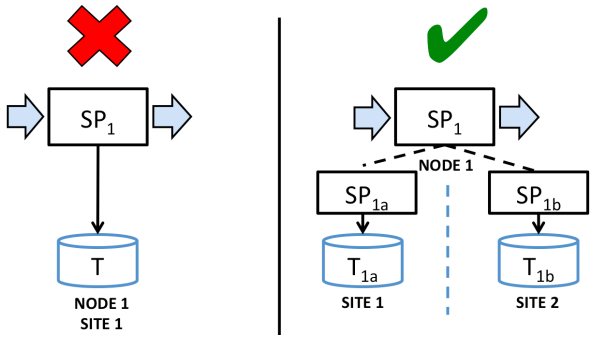


Figure 6.5: Procedure Partitioning

The answer is the creation of an optimized distributed transaction capable of executing in parallel but committing as a single transaction. A transaction has to meet two criteria. First, the shared state that is accessed must either be all partitioned on a single key or replicated. It is also possible that no shared state is accessed, as is the case with purely streaming workloads. Secondly, there must not be any dependencies between operations that access the data items.

If these criteria are met, then it is possible to divide the transaction into fragments, each of which can be executed on a separate partition in parallel, as shown in Figure 6.5. These fragments are created, distributed, and then commit together using two-phase commit. In the event that the conditions are not met, but data items from multiple partitions must still be accessed, then a typical distributed transaction must be used instead.

6.1.6 Interactions Between Design Principles

While each of the five design principles (State Co-Location, Processing Co-Location, Load Balancing, Dataflow Partitioning, and Procedure Partitioning) have clear benefits when examined in-

Principle	Origin	Advantages	Conflicts	Auto?
State Co-Location	OLTP	Avoids distrib txns	Load Balancing Processing Partitioning	Yes
Processing Co-Location	Streaming	Avoids MOVE ops	Load Balancing Processing Partitioning	Yes
Load Balancing	OLTP Streaming	Reduces bottlenecks	Processing Co-Location	Yes
Dataflow Partitioning	Streaming	State Co-Location Load Balancing	Processing Co-Location	No
Processing Partitioning	OLTP Streaming	Load Balancing	State Co-Location Processing Co-Location	No

Table 6.1: Principles of Good Design

dependently, individual design choices become much more difficult when all five principles are examined together. Many of the principles, such as Processing Co-Location and Load Balancing, are frequently directly at odds with each other, making it difficult to determine the best design for a specific situation without careful analysis. The interactions between these principles are represented in Table 6.1.

Additionally, the Dataflow and Procedure Partitioning design principles are very complicated and may influence the programming model. In the case of Procedure Partitioning, the user must be aware of a partitioning key on which the processing can be split into multiple independent processes, and the database must be designed such that the required state is located on the same node as the processing. Dataflow Partitioning becomes even more complicated, as splitting operations across two stored procedures fundamentally changes the transactional semantics of those operations. Both Procedure and Dataflow Partitioning require careful analysis of the internal operations of the stored procedures, and should be largely left to the users' discretion.

Putting the principles of good design into practice can be a tedious task for a developer. Each principle may be directly at odds with others, leading to a push-and-pull of fine-tuning designs to find the best possible performance. Ideally, the system is instead able to determine the best (or, at least, a near-optimal) design given a set of constraints. The main thrust of our proposed ongoing research lies in determining good database designs automatically, using the principles described above.

6.2 Modeling the Relationship between State and Processing

To model database designs, we take inspiration from Schism [53], an automatic database designer for OLTP workloads. Schism seeks to horizontally partition a database in a way that minimizes the number of costly distributed transactions. It accomplishes this by representing the relationship between tuples as a graph. Nodes of this graph represent pieces of state (tuples), and the edges represent whether those tuples are accessed together in a transaction according to a workload trace. Once this graph has been determined, Schism then uses a graph-partitioning algorithm to determine a state-partitioning scheme that results in the fewest number of distributed transactions.

We can follow a similar approach when partitioning the state being accessed by dataflow graphs, but with one important distinction: in addition to partitioning the state, we must also consider how best to partition our processing as well. To accomplish this, our graph includes more variables, but the fundamental goal of even partitioning at a minimum cost is maintained.

6.2.1 Workload Analysis

To build a representative graph, the first task must be to analyze the given workload. Specifically, information must be gathered on which stored procedures access which state, and which stored procedures trigger which downstream stored procedures. Depending on the granularity with which we are partitioning, the state access can be evaluated as coarsely as "procedure SP accesses table T", or as finely as "procedure SP accesses tuple U with likelihood L." This data is most easily collected via a workload trace, in which the dataflow graph runs for a set period of time and statistics are collected about state and processing access.

6.2.2 Graph Representation

Once we have collected statistics about which stored procedures are related to which state and other stored procedures, it is time to build our graph of connections between state and processing. Unlike Schism, which considers only state-to-state relationships, we consider two different types of

relationships: state-to-state and processing-to-state. This means that both state and processing must be represented as nodes in the graph, while their interactions are represented by weighted edges. Thus, we consider two different types of nodes in our graph representation:

State Nodes represent each piece of state in the system. Depending on how granular the statistics and eventual partitioning is, state nodes can represent any quantity of state, from a table to a tuple. Each state node has a *memory cost* associated with it, which represents the amount of space in memory the state occupies.

Processing Nodes represent each stored procedure in the dataflow graph(s). A processing node may either represent an entire stored procedure, or a portion of one in the case of partitionable procedures. Each processing node has a *processing cost* associated with it, which is an arbitrary representation of how much CPU processing resources are needed to complete a transaction.

These two types of nodes are then connected with two different types of edges:

Processing-to-Processing Edges represent the relationship between two stored procedures in a dataflow graph. If SP_n is connected to $SP_n + 1$ in a dataflow graph, then these nodes are connected via a processing-to-processing edge. Each P-to-P edge may have a weight associated to it, which represents the relative cost of separating the two nodes.

Processing-to-State Edges represent the relationship between a stored procedure and the state it accesses. Each P-to-S edge may have a weight associated to it, which represents the relative cost of remotely accessing the state that lives on another partition.

Note that state-to-state edges are unnecessary, as those relationships are evaluated implicitly through pairs of processing-to-state edges connected to a single processing node.

6.3 Cost Model

Once the dataflow graph is represented by a graph of nodes and edges, the partitioning of state and processing can be represented by partitioning that graph n ways, where n is the number of machines that are to be used. Additionally, at a finer level, the graph can be partitioned p ways, where p is the number of partitions in the cluster. The goal of the graph partitioning is to divide the workload as evenly as possible across machines while both meeting the specified constraints and minimizing the overhead of distributing state and processing.

In order to evaluate the quality of various designs, we first need to quantify the benefit of each of the core principles of design. The first three principles, state co-location, processing co-location, and load balancing are all relatively easy to quantify, so we begin there.

6.3.1 Quantifying State Co-Location

Measuring state co-location is about detecting the number of distributed transactions in a workload. In the graph representation, distributed transactions are modeled as edges between processing nodes and state nodes (P-to-S edges). These edges are given a weight value that represents how costly it is to separate the processing node from the state node, i.e. how costly the distributed transaction is. The weight may be calculated as a function of the probability the state is accessed by the processing (i.e. how likely is the tuple to be accessed by the stored procedure), or as a function of the selectivity of the query (i.e. how many tuples are read/written from the given stored procedure). For simplicity's sake, we start by considering an edge cost to be a binary value; either the state is accessed by the processing (value of 1), or it isn't (value of 0).

To quantify the relative cost of distributed state access, we compare the total cost of distributed state accesses (i.e. cut processing-to-state edges) to the total number of state accesses (i.e. processing-to-state edges, cut or uncut).

$$\text{state co-location cost} = (\sum \text{distributed state costs})/(\sum \text{all state costs})$$

Or, if we assume all distributed state access costs to be the same:

$$\text{state co-location cost} = \frac{\text{number of cut processing-to-state edges}}{\text{total number of processing-to-state edges}}$$

6.3.2 Quantifying Processing Co-Location

Measuring processing co-location involves quantifying the number of MOVE operations required for a given dataflow graph. MOVE operations are modeled as edges between processing nodes and other processing nodes (P-to-P edges). These edges are given some weight that represents how costly it is to separate two consecutive processing nodes (stored procedures), i.e. how costly the move operation is. The weight may be calculated as a function of the probability that the downstream stored procedure will be accessed, or as a function of how much stream state is passed from one procedure to the other. For simplicity's sake, we start by considering an edge cost to be a binary value; either two stored procedures are consecutive in a dataflow (value of 1), or it isn't (value of 0).

To quantify the relative cost of MOVE operations, we compare the total cost of all MOVEs (i.e. cut P-to-P edges) to the total number of connections between procedures (i.e. all P-to-P edges, cut or uncut).

$$\text{processing co-location cost} = (\sum \text{MOVE costs})/(\sum \text{all processing connection costs})$$

Or, if we assume all *move* operations are weighted identically:

$$\text{processing co-location cost} = \frac{\text{number of cut processing-to-processing edges}}{\text{total number of processing-to-processing edges}}$$

6.3.3 Quantifying Load Balancing

In addition to measuring the cost of distribution, we also need to measure the benefit of distributing load across machines. We define the processing load on each machine to be the sum of the *processing costs* of the processing nodes (stored procedures) assigned to that machine. Ideally, if the load is spread evenly, this value should be as close as possible from one machine to another. It is worth noting that the measure of how "overloaded" a machine is is taken relative to its maximum processing power; a machine capable of more CPU cycles than another should also be assign more work. For simplicity, for now we assume that all machines are identical and capable of the same amount of work.

While it isn't a perfect measure, we can roughly evaluate how evenly the load is distributed by measuring the difference between the most overloaded machine and the least overloaded machine. This value must be normalized by dividing it by the total processing load across all machines.

$$\text{load balancing cost} = \frac{\max(\text{processing costs on any machine}) - \min(\text{processing costs on any machine})}{\text{all processing costs}}$$

6.3.4 Combining Elements into a Cost Model

There is an obvious push-and-pull nature between balancing the load between machines and trying to minimize the number of distributed transactions and moves. Balancing the load means, at the bare minimum, that some processing elements will be separated across machines. To evaluate a design in the presence of our first three design principles, it is important to weigh the design aspects as is appropriate to the scenario. For instance, in a cluster in which the network between machines has low bandwidth, load balancing may not be as important as minimizing the number of distributed transactions or MOVE operations. Similarly, if the network bandwidth is not a large concern, then distributed transactions and MOVE operations become less expensive on the whole.

To evaluate the cost of all three elements at once, we assign each value a weight:

$$\text{design cost} = \alpha(\text{state co-location cost}) + \beta(\text{processing co-location cost}) + \gamma(\text{load balancing cost})$$

On a typical system, γ (the load balancing weight) is going to be the largest number of the three, as we want to weigh our cost model in favor of distribution. The second highest cost is typically α (the state co-location weight) is second highest, as distributed transactions are extremely expensive. Third highest is β, as optimized MOVE operations make processing co-location costs the least concerning. While this is the typical ranking of the three costs, the specific numbers can vary wildly from cluster to cluster depending on the architectural details.

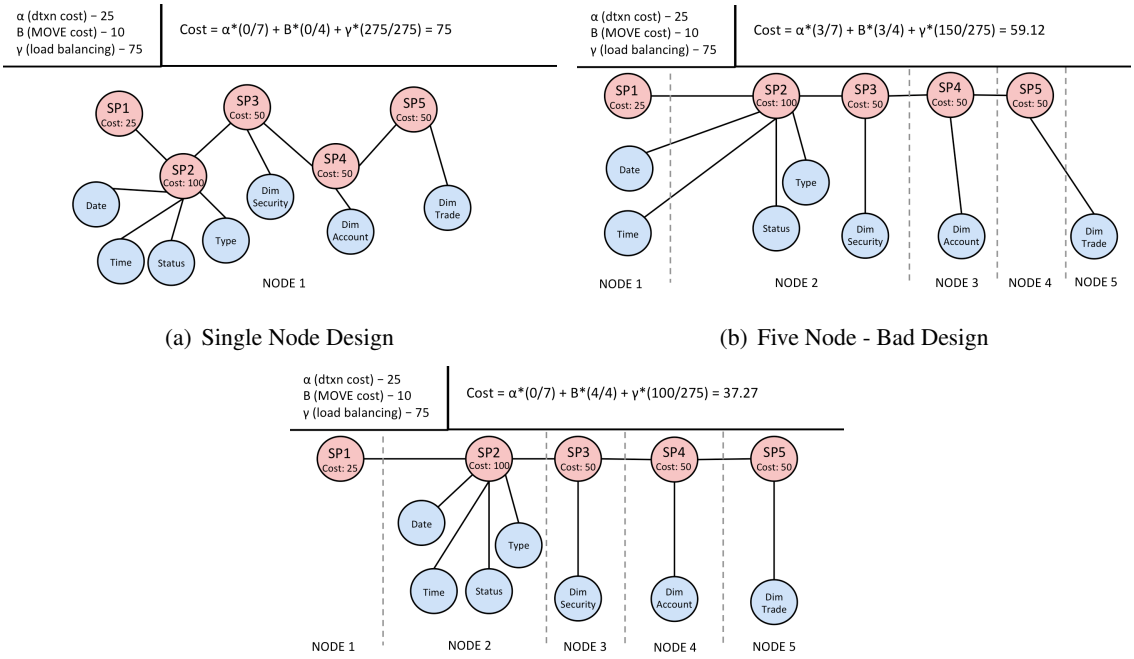
6.4 Searching for an Optimal Design

The process of automatically searching for an optimal design contains several steps, all of which happen offline before the database is deployed. First, the database administrator (DBA) must define the dataflow and procedures of the workload. We can then run a workload trace on the resulting dataflow graph, determining the selectivity and probability with which state and processing are connected to one another. From there, we can create a fully-weighted graphical representation, which can be partitioned using our cost model. Finally, we can experimentally evaluate elements of the cost model, make changes to the dataflow and procedure design if necessary, and rerun the automatic designer to iteratively find the best possible design.

6.4.1 Workload Trace

While processing-to-processing edges are defined by the dataflow graph, processing-to-state edges must be determined before the full partitionable graph can be built. Because S-Store is built on top of H-Store, it is able to use Horticulture out of the box as a design tool for suggestion of several elements, including horizontal partitioning key, table replication, and secondary indexes [124]. These suggestions are based on workload traces, and provide best-possible designs for a majority of scenarios. When considering optimal database designs for S-Store, we can take advantage of the fact that dataflow graphs provide very concrete, predictable workloads.

Horticulture is able to detect which stored procedures access which state, and with what probability, by using a workload trace. S-Store can use this same workload trace to determine the relative average runtimes/costs of individual stored procedures. This information can then be applied to the cost model, specifically to the edge weights in the case of state access, and processing node weight



(a) Single Node Design

(b) Five Node - Bad Design

(c) Five Node - Good Design

Figure 6.6: State and Processing Partitioning Schemes for DimTrade Ingestion

in the case of relative procedure runtimes. The workload trace must also detect how much information is being passed from SP to SP via stream, as that information is necessary for determining processing-to-processing edge weights.

A workload trace may or may not be required, depending on the granularity of the design. In the TPC-DI example, we already know which tables are accessed by which procedures, and which procedures trigger other procedures; it is directly expressed in the design. Therefore, for a coarse partitioning of state and processing, a workload trace is not needed. However, if state is to be partitioned on the tuple-level, or if edge weights are required for more effective partitioning, then it becomes necessary to know how frequently each stored procedure accesses individual tuples, and the selectivity of the data being retrieved by the database or passed between procedures.

6.4.2 Constrained Graph Partitioning

Once the edge weights between processing and state nodes are determined, S-Store's automatic designer can attempt to partition the graph in n ways, where n is the number of machines in the cluster. Constrained, k-way graph partitioning is a well-studied problem, and is known to be NP-complete. However, there are a number of available libraries that use heuristics to find optimal graph

partitioning, including [93, 90, 88, 87]. For instance, because the cost model can be expressed most simply as "balance the load between nodes as much as possible while cutting as few edges as possible," one obvious heuristic is to attempt to cut edges with a lower cost first. A proof-of-concept implementation has been created using METIS [87], but it is left to proposed work to create a full-fledged implementation using heuristics.

In the TPC-DI example, we first express the workload as a graph representation between state and processing (Figure 6.6(a)). In this example, we assume that all edges carry the same weight, with processing-to-processing cuts and processing-to-state edges being differentiated using the α and β variables in the cost model (Section 6.3.4). We assume that the state-colocation (α) cost carries a higher weight than the processing-colocation (β) cost, meaning distributed transactions in the design are more expensive than the MOVEs. We also assume that load-balancing (γ) carries the highest weight of the three.

When evaluating how to partition the Trade.txt dataflow of TPC-DI, there are two goals to consider: the designer is trying to 1) divide costs of processing nodes as evenly as possible while 2) cutting as few edges (edges with the smallest weights) as possible. At the same time, the algorithm must consider costs of memory nodes, ensuring that no node is taking on more state than it can handle (not shown in these examples). In the case of the Trade.txt dataflow example in Figure 6.6(a), we evaluate the cost of the naïve partitioning scheme of placing everything on a single node. While this incurs no distributed transaction or MOVE penalty, the price of placing everything on a single node makes the cost quite high. In the second partitioning example (Figure 6.6(b)), we arbitrarily partition the graph across five nodes without consideration of which cuts to make. While this has a lower cost than the first due to load distribution, it incurs a number of unnecessary co-location penalties. Meanwhile, the optimal design (Figure 6.6(c)) features a relatively low cost. This is because all state is co-located with its processing (easy to do, since state is not shared between the processing nodes), and the load is evenly distributed across the five nodes.

6.4.3 Evaluation and Iteration

Once an optimal design is found given the dataflow, procedure, and hardware constraints, it is important to then evaluate the actual performance of this design. This can be done by deploying the newly-generated design onto the cluster of machines and evaluating its performance on a sample workload. Similarly, this can be accomplished by simulating the design with the measured constraints, and roughly estimating the relative performance of each component. In either case, the goal is to find where the bottlenecks of the design are, e.g. nodes that are overworked, overly expensive MOVEs, or distributed transactions.

Using the newly-identified bottlenecks, the user can then evaluate whether or not to make changes to the original dataflow or procedure design. For instance, in the TPC-DI example, SP2 has become a processing bottleneck for the design, as Node 2 is the most overworked in this scenario. One solution to this problem may be to further partition SP2, either laterally into several separate SPs or vertically into a partitioned SP with optimized distributed transactions. In either case, the user would then rerun the automatic designer on the new design, iterating on the previous result until the design requirements are met.

6.5 Conclusions

In developing applications for a distributed transactional streaming system, there are a large number of variables to consider. These include state co-location, processing co-location, load balancing, dataflow partitioning, and procedure partitioning. For complicated workloads, the design space can quickly become overwhelming for a database developer.

In order to provide guidance and simplify the space, we provide some preliminary work into an automatic database designer. This designer uses a cost model that takes into account three of the more quantifiable elements: state co-location, processing co-location, and load balancing. By weighting these three elements according to the system hardware, we are able to calibrate the cost model. State and processing can be theoretically represented as a graph for which nodes and edges can be assigned values based on the cost model. By partitioning this graph in a way that maximizes balance while minimizing cuts, we can automatically find a good partitioning design. At that point, the user can iterate on the design by adjusting the dataflow and procedure partitioning elements, rerunning the algorithm again to hopefully improve the design further.

There is much more to explore in the space of automatic database design, both drawing from previous academic works (Chapter 7) and in making further developments to transactional streaming architecture (Chapter 8).

Chapter 7

Related Work

7.1 Data Streams and Streaming Data Management Systems

Data streams are defined as "a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items" [71]. The notion of data stream processing has been around since the 90s. Similar to data streams, the *Chronicle data model* involved *chronicles*: ordered, append-only sequences of tuples [80]. The *Tapestry* system introduced the notion of continuous queries - long-running processes that immediately executes on incoming data items as they arrive [148]. Some stream database managers, such as *Tribeca*, emerged in the mid 90s to provide limited querying ability over network packet streams [145]. Similar to the push-based SDMS systems, *event-condition-action* triggers were created in SQL databases in the *Alert* system [133]. These triggers turned a passive DBMS into an active one via append-only active tables.

In the late 90's and early 2000's, a number of database systems such as OpenCQ [106], NiagaraCQ [50], Stanford's STREAM system [29], and Brown/MIT's Aurora SDMS [14] were developed. These systems focused on taking in "internet-scale" data streams using *continuous queries*, long-running processes that immediately executes on incoming data items as they arrive.

The abstract architecture of a typical SDMS is shown in Figure 7.1 [71]. An input monitor regulates the incoming data items, managing overall system load. State itself is typically stored in three levels: temporary working storage for the queries themselves, summary storage for stream estimations, and static storage for catalog and meta-data. Continuous queries are stored in a repository, and executed as the data requires. All streaming output is buffered and sent to the corresponding data sinks.

In 2005, MIT and Brown established a list of eight requirements for real-time stream processing systems [141]. Real-time stream processing systems should provide the ability to:

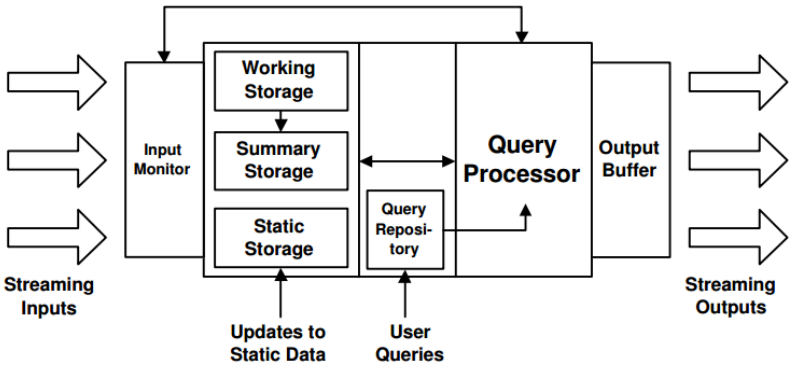


Figure 7.1: Abstract Architecture of a Streaming Data Management System (from [71])

1. Keep the data moving
2. Query using SQL, or a SQL-like language such as CQL [25]
3. Handle delayed, missing, or out-of-order data
4. Generate predictable outcomes
5. Integrate historical and streaming data
6. Guarantee recoverable and available data
7. Partition and scale applications, preferably automatically
8. Process and respond with extremely low latency

The next generation of stream processing systems were largely focused on scalability and industrial applications [72]. The Borealis system expanded on the earlier Aurora model, adding modules for load managing and shedding [13, 147]. Products such as Esper focused on *complex event processing* and streaming analytics [5]. Apache developed Storm, a real-time data processing system with a focus on parallelization of computation and fault-tolerance in the event of failover [151]. This was later expanded into Hereon [97]. Microsoft created StreamInsight, implementing continuous queries on their SQL Server [20, 21]. A number of modern streaming systems are inspired by the MapReduce model [170], further covered in the next section.

7.2 State Management in Streaming Systems

First-generation streaming systems provided relational-style query processing models and system architectures for purely streaming workloads [5, 14, 24, 48]. The primary focus was on low-latency processing over push-based, unbounded, and ordered data arriving at high or unpredictable rates. State management mostly meant efficiently supporting joins and aggregates over sliding windows,

and correctness was only a concern in failure scenarios [30, 79].

Botan et al. proposed extensions to the traditional database transaction model to enable support for continuous queries over both streaming and stored data sources [38]. While this work considered ACID-style access to shared data, its focus was limited to correctly ordering individual read/write operations for a single continuous query rather than transaction-level ordering for complex dataflow graphs as in S-Store.

More recently, a new breed of streaming systems has emerged, which aim to provide a Map-Reduce-like distributed and fault-tolerant framework for real-time computations over streaming data. Examples include S4 [118], Storm [151], Twitter Heron [97], Spark Streaming [4, 168], Samza [3], Naiad [115], Flink [1], and MillWheel [18]. These systems differ significantly in the way they manage persistent state and the correctness guarantees that they provide, but none of them is capable of handling streaming applications with shared mutable state with sufficient consistency guarantees as provided by S-Store.

S4, *Storm*, and *Twitter Heron* neither support fault-tolerant persistent state nor can guarantee exactly once processing. Storm when used with *Trident* can ensure exactly-once semantics, yet with significant degradation in performance [11]. Likewise, *Google MillWheel* can persist state with the help of a backend data store (e.g., BigTable or Spanner), and can deal with out of order data with exactly-once processing guarantees using a low watermark mechanism [18].

Several recent systems adopt a *stateful dataflow model* with support for in-memory state management. *SEEP* decouples a streaming operators state from its processing logic, thereby making state directly manageable by the system via a well-defined set of primitive scale-out and fault-tolerance operations [65, 66]. *Naiad* extends the MapReduce model with support for structured cycles and streaming based on a timely dataflow model that uses logical timestamps for coordination [115]. *Samza* isolates multiple processors by localizing their state and disallowing them from sharing data, unless data is explicitly written to external storage [3]. Like S-Store, all of these systems treat state as mutable and explicitly manageable, but since they all focus on analytical and cyclic dataflow graphs, they do not provide inherent support for transactional access to shared state.

There are a number of systems have explicitly been designed for handling *hybrid workloads* that include streaming. *Spark Streaming* extends the Spark batch processing engine with support for discretized streams (D-Streams) [168]. All state is stored in partitioned, immutable, in-memory data structures called Resilient Distributed Datasets (RDDs). Spark Streaming provides exactly-once consistency semantics, but is not a good fit for transactional workloads that require many fine-grained update operations. *Microsoft Trill* is another hybrid engine designed for a diverse spectrum of analytical queries with real-time to offline latency requirements [47]. Trill is based on a tempo-relational query model that incrementally processes events in batches organized as columns. Like

Spark Streaming, its focus lies more on OLAP settings with read-mostly state. Last but not least, the *Google Dataflow Model* provides a single unified processing model for batch, micro-batch, and streaming workloads [19]. It generalizes the windowing, triggering, and ordering models found in MillWheel [18] in a way to enable programmers to make flexible tradeoffs between correctness and performance.

7.3 Real-Time Databases

Real-time databases handle transactions that have some type of deadline or timing constraint [86, 121]. While traditional DBMS systems are evaluated on an "average" response time, real-time databases are typically evaluated by how frequently transactions are executed behind schedule [138]. Deadlines are defined by the application. "Real-time" does not necessarily involve a deadline of milliseconds; in some cases, deadlines may be seconds or even minutes [139]. Typical use-cases for real-time databases include control of laboratory experiments, flight control systems, robotics applications, etc.

In all cases, real-time transactions lose their value over a specified period of time [86], though the consequences may vary depending on the type of deadline [94, 121]. *Hard* transactions imply that missing a deadline absolutely must not happen. *Soft* transactions, on the other hand, will merely decrease in value after their deadlines, eventually reaching a value of zero at some point. *Firm* transactions have less disastrous consequences for missing deadlines than hard transactions; however, their value will drop to zero if they do.

If real-time databases provide the integrity constraints and serializable transaction model of a typical OLTP system, they are said to be *internally consistent* [103]. ACID properties apply in this case [121]. At times, however, it may be more valuable for a real-time database to instead be *externally consistent*. In this case, the database changes its values in accordance with the real world, and ACID properties or integrity constraints may be broken in the process.

Real-time databases that handle temporal data may demand that data values in the database have roughly the same age [105, 159]. Synchronizing data values with real-time constraints can require two different transaction consistency types [166]. External transaction consistency ensures that the difference between a timestamp for a transaction operation and the valid time of the data item it accesses is less than a given threshold [121]. Temporal transaction consistency, on the other hand, ensures that all data items read by the transaction have timestamps within a given threshold.

There are a few core differences between real-time databases and streaming data management systems. First and foremost, real-time databases manage a series of incoming transaction requests rather than a stream of incoming data items [163, 146]. Thus, real-time databases take a *pull-based*

approach to data processing rather than the *push-based* approach of SDMSs. The notion of dataflow graphs is not functionality that real-time databases include, and the ordering guarantees of stream processing systems do not apply. Similarly, exactly-once processing is not a concern.

Additionally, streaming data management systems frequently do not apply specific deadlines on when data must be processed, but instead prioritize ordering of incoming data items with as little latency as possible. Our definitions of consistency are entirely internal, prioritizing ACID transactions and serializability. Our stream ordering constraints are somewhat similar to temporal transaction consistency, ensuring that the state read by a given transaction has been accessed in order of batch-id.

7.4 Main-Memory OLTP

While main memory was relatively expensive until fairly recently, main-memory database systems have been an active topic of research since the early eighties [56, 63]. The University of Wisconsin introduced the MM-DBMS project, which introduced the use of pointers for direct record access in memory and optimized index strategies [100]. Recovery techniques for main-memory databases were explored in the MARS system, using two separate processors to copy updates into stable disk storage [60]. TPK implemented the technique of serially executing transactions via specialized threads in a multi-processor system [102]. A variety of commercial main-memory OLTP systems appeared in the nineties, including TimesTen [150], P*Time [46], and DataBlitz [31] [63].

Advances in multi-core parallelism, memory prices, and network technology in the 2000s rapidly changed the main-memory database landscape [63]. These enhancements led to a new class of distributed main-memory systems with new architectural needs (further discussed in Sections 2.2 and 2.2.1).

One of the earliest distributed main-memory databases was PRISMA/DB, which uses two-phase locking and 2PC to execute transactions over a distributed environment [91, 123]. HyPer was designed to be a combination OLTP and OLAP high-performance database system, with a distributed version known as ScyPer [89, 114]. ScyPer manages its OLTP and OLAP workload on separate nodes, with the data on the OLAP node being potentially stale. SAP's HANA takes a similar approach, but stores the data as both a row-store and a column-store (rather than being a direct copy) [99, 64].

The primary main-memory OLTP system discussed in this paper, H-Store, and its commercial counterpart VoltDB, are both examples of partitioned systems [85, 12, 63]. The serial execution of transactions on individual threads greatly simplifies concurrency control. Hekaton [57], HANA [99], MemSQL [51], and Oracle TimesTen [150] are all examples of non-partitioned systems. In a

non-partitioned system, all nodes are able to access all potential state, with multiple threads having access to state. This results in a more complex engine implementation and concurrency control, but leads to more flexible transactions and allows a greater variety of state access without a huge performance hit.

7.5 Distributed Stream Scheduling

Early streaming systems primarily featured continuous queries as long-running processes, and scheduling was oriented towards executing these queries in a way that maximized the use of resources. A major priority in all instances is recovery. For instance, Aurora used *train scheduling*, a set of scheduling heuristics that minimize I/O operations and unnecessary query executions [44]. Adaptive load-aware scheduling of continuous queries reduces resource usage during peak load times, and Chain scheduling optimizes memory usage at run-time for a variety of single-stream queries [28]. Real-time scheduling introduced the notion of adjusting internal scheduling to meet time-dependent Quality-of-Service requirements [132].

More recently, continuous operator scheduling is applied to more complicated distributed environments. Microsoft's StreamScope has introduced two abstractions, *rVertex* and *rStream* to manage complex scheduling within a distributed streaming system [104]. In this case, rStreams maintain the sequence of events with monotonically increasing identifiers, and the rVertexes ensure determinism on various input streams to ensure correctness during recovery. Storm with Trident [11], MillWheel [18], and TimeStream [130] are all similar examples of providing strong consistency and exactly-once guarantees with their distributed scheduling approaches.

Much of parallelized distributed streaming has moved to a *bulk-synchronous parallel* (BSP) model [153]. In this model, each parallel node performs a local computation followed by a blocking barrier period to allow the nodes to communicate with each other. The process is then repeated. This methodology is adopted by MapReduce, in which the *map* portion involves the parallel processing, and the *reduce* portion is the communication barrier period [54]. Many modern streaming systems implement this model with each operator corresponding to a cycle, using small *micro-batches* to reduce the latency as much as possible [156]. Some examples include Spark Streaming [168] and Google Dataflow with FlumeJava [19].

Some streaming schedulers seek to remove barriers from the BSP model whenever possible in order to improve efficiency and latency [157]. Drizzle schedules operations in groups in order to remove barriers for communication, instead opting to fetch input data as soon as prior tasks have finished [156]. Other systems such as Ciel use complex *dynamic task graphs* in order to choose which tasks to schedule when [116].

One contribution to efficient scheduling as it pertains to streaming with shared mutable state is the improvement of data locality. PACMan improves the performance of parallel processing jobs by improving the caching coordination across distributed caches [22]. Resilient Distributed Datasets (RDDs) similarly provide consistent shared memory access by restricting data transformations to coarse-grained changes involving immutable data items [167].

Other systems such as Borg [158] and Mesos [78] dynamically assign tasks across cluster computing frameworks, but do not support write-heavy transactions on mutable state. Other job schedulers such as Apollo [39] and Sparrow [120] also schedule short micro-tasks, but do not support dependencies between tasks.

7.6 Streaming Data Ingestion

There has been a plethora of research in ETL-style data ingestion [95, 154]. The conventional approach is to use file-based tools to periodically ingest large batches of new or changed data from operational systems into backend data warehouses. This is typically done at coarse granularity, during off-peak hours (e.g., once a day) in order to minimize the burden on both the source and the backend systems. More recently, there has been a shift towards micro-batch ETL (a.k.a., "near real-time" ETL), in which the ETL pipelines are invoked at higher frequencies to maintain a more up-to-date data warehouse [155]. It has been commonly recognized that fine-granular ETL comes with consistency challenges [84, 70]. In most of these works, ETL system is the main source of updates to the warehouse, whereas the OLAP system takes care of the query requests. Thus, consistency largely refers to the temporal lag among the data sources and the backend views which are used to answer the queries. In such a model, it is difficult to enable a true "real-time" analytics capability. In contrast, the architecture we propose in this paper allows queries to have access to the most recent data in the ETL pipeline in addition to the warehouse data, with more comprehensive consistency guarantees. Implementing the ETL pipeline on top of an in-memory transactional stream processing system is the key enabler for this.

Modern big data management systems have also looked into the ingestion problem. For example, AsterixDB highlights the need for fault-tolerant streaming and persistence for ingestion, and embeds data feed management into its big data stack so as to achieve higher performance than gluing together separate systems for stream processing (Storm) and persistent storage (MongoDB) [74]. Our architecture addresses this need by using a single streaming ETL system for streaming and storage with multiple guarantees that include fault tolerance.

Shen et al. propose a stream-based distributed data management architecture for IoT applications [135]. This has a three-layer (edge-fog-cloud) architecture similar to our Streaming ETL

architecture. However, the main emphasis is on embedding lightweight stream processing on network devices located at the edge layer (like our data collection layer) with support for various types of window joins that can address the disorder and time alignment issues common in IoT streams.

There has been a large body of work in the database community that relates to time-series data management. As an example for one of the earliest time-series database systems, KDB+ is a commercial, column-oriented database based on the Q vector programming language [9, 8]. KDB+ is proprietary and is highly specialized to the financial domain, and therefore, is not suitable for the kinds of IoT applications that we propose to study in this proposal. There are several examples of recent time-series databases, including InfluxDB [6], Gorilla [125], and OpenTSDB [164]. Each of these provide valuable insight into time-series databases but do not meet our ingestion requirements and are not a great fit for the kinds of IoT applications that we consider.

BigDAWG has many parallels with federated database systems like Garlic [42]. For example, in both cases, schema mapping and data movement between sites are important features. The main difference is that in a federated database, each site (component) was autonomous. Each site had a different owner with her own set of policies. It would not be possible to permanently copy data from one system to another. BigDAWG is really a database built out of heterogeneous databases. There is a single owner who determines things like data placement across systems.

Integrating real-time and batch processing has become an important need, and several alternative architectures have been adopted by big data companies, such as lambda [10] or kappa architecture [7]. In lambda, the same input data is fed to both a throughput-optimized batch and a latency-optimized real-time layer in parallel, whose results are then made available to the applications via a serving layer. Kappa in contrast feeds the input only to a streaming system, followed by a serving layer, which supports both real-time and batch processing (by replaying historical data from a logging system such as Kafka [96]). Fernandez et al. also propose Liquid - an extended architecture similar to kappa [67]. Our polystore architecture is similar to kappa and Liquid in that all new input is handled by a streaming system, but our serving layer consists of a more heterogeneous storage system. Also, our streaming system, S-Store, is a transactional streaming system with its own native storage, which facilitates ETL.

7.7 Distributed Database Design

Database design for stream processing in a distributed environment traditionally focuses on proper allocation of processing resources to provide best-possible latency and/or throughput. Frequently, this means dividing the workload to minimize communication channels across nodes while maximizing the CPU usage on each node. This approach involves splitting a workload in a way that

improves either data or pipeline parallelism.

Many distribution approaches choose to focus on data parallelism. IBM's System S team refers to this a *channelization*, and defines a *split/aggregation/join* architectural pattern that separates incoming data into physical streams, aggregates the result, and correlates the data back together [23]. This is very similar to the map/reduce architecture employed by batch-processing systems like Hadoop and Pig Latin, in which incoming data is hashed into categories, aggregated in parallel, and recombined into a single aggregate [161, 119]. One of the most popular modern streaming systems, Spark Streaming, is also batch-based in nature and follows the map/reduce paradigm [168]. These data parallelism techniques rely heavily on partitionable workloads, and typically are unconcerned about mutable state shared with other applications.

Other stream distribution techniques instead approach the scalability problem from a pipeline parallelism perspective. These techniques focus on assigning operators (or combinations of operators, called *processing elements*), to nodes in a way that best maximizes the utility of all nodes' resources. The challenge here is mapping a logical dataflow graph to physical nodes, which the System S team accomplishes first accomplishes via both code generation and graph partitioning [69, 92].

Note that none of these approaches strongly consider shared mutable state. Storage optimization for streaming systems has been considered by IBM; however, the research in this area focuses on deletion of old data to make room for the new (attempting to retain data for as long as possible) rather than considering state that may be accessed by multiple operators or OLTP requests [77].

Single-node OLTP database design has been well-studied from a variety of perspectives. Search heuristics and constraint programming are used for single-node index generation [49] and materialized views [17], and query-evaluation used in schema design and evaluation for data warehouses [127], to name a few.

In a distributed environment, database design can be traced to file assignment optimization that balances cost and performance [58]. Parallel shared-nothing designers typically seek to leverage the knowledge of partitioning attributes of relations in order to ship database operations to the nodes with the appropriate data [173]. Such techniques minimize the communication overhead caused by distributed transactions.

In an OLTP environment, it is important to determine which partitioning attributes are most important, and how to utilize other techniques (such as replication). There are a variety of approaches for this, most of which require a workload trace. Schism models database partitioning as a graph partitioning problem, establishing each tuple (or groups of related tuples) as graph nodes connected by queries that access related tuples together [53]. Similarly, Horticulture uses a Markov model to estimate which queries are likely to access which table, using large-neighborhood search

to determine horizontal partitioning, replication, indexes, and stored procedure routing [124].

These approaches do not consider the push-based nature of a streaming environment, and do not use minimal-yet-efficient movement of data across nodes to generate pipeline parallelism in their workflows. An automatic streaming database designer has the advantage of having thorough knowledge of its workload, as dataflow graphs of continuous queries will remain relatively static.

The notion of "breaking up" a transaction into multiple smaller transactions for performance purposes has been explored in active databases [43]. The event-driven nature allows the database to divide rules into multiple combinations of transactions in order to improve performance. Similarly, research on "transaction chopping" has shown that redefining transaction boundaries can have strong performance benefits [134]. Another example is Pyxis, which automatically partitions operations in a workload between the database and application layer [52]. This approach does not assume the ordering guarantees of streaming, which can provide more flexibility and constraints in determining transaction boundaries. In each case, maintaining serializability is the primary concern rather than stream processing semantics such as stream ordering and exactly-once processing.

Chapter 8

Future Work

While this dissertation has described a variety of initial research on the topic of transactional streaming, it is far from an exhaustive list of possibilities for the field. Quite the contrary, during our research there were a number of potential paths that were left unexplored due to time and resource constraints. There are many potential improvements in both model and implementation that show promise, some of which are listed here.

8.1 Improved Parallelization by Splitting Transactions

Chapter 4 outlined two potential paths to parallelization (pipeline and data parallelism), with each path applying to specific category of workload. Pipeline parallelization in particular is common for dataflows that involve shared mutable state, but presents a fairly rigid dataflow construction. However, it is possible to combine pipeline and data parallel processing into a single dataflow graph. As briefly explored in theory in Section 6.1.5, dataflow parallelization can be greatly improved by parallelizing the execution of individual transactions. This can be done using vertical *transaction splitting* [123].

Assuming that a stored procedure accesses state that is only partitioned on a single key (or on multiple keys, but all with the same partitioning strategy), it becomes possible to split a transaction into multiple *sub-transactions*, each of which may execute independently, only communicating when they are each ready to commit. This is not unlike the description of the data parallelism strategy described in Section 4.2, but with one important difference: each transaction must coordinate a full commit together rather than committing separately.

To illustrate this concept, suppose a dataflow graph of two stored procedures, $SP1$ and $SP2$, shown in Figure 8.1. $SP1$ reads from table $T1$, partitioned on a specific key $k1$ into $T1_a$ and $T1_b$,

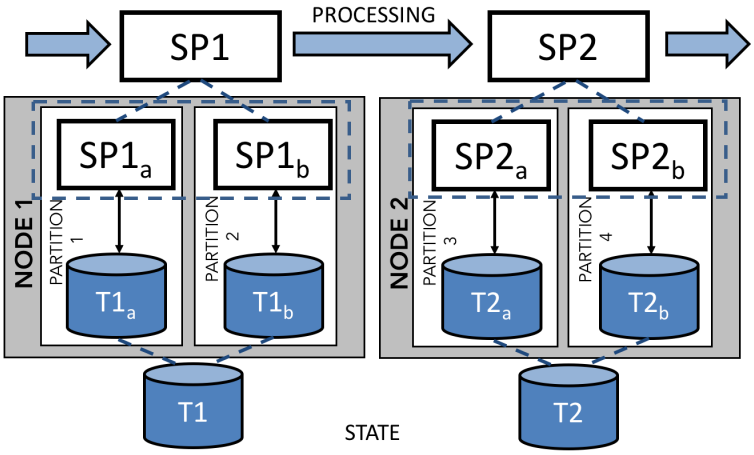


Figure 8.1: Split and Merge - Optimized Distributed Transactions

and is labeled as such. Similarly, $SP2$ writes to table $T2$, partitioned on a second key $k2$ into $T2_a$ and $T2_b$. $SP1$ receives a large batch of tuples with no particular shared trait, but containing keys $k1$ and $k2$. This batch can be *split* into *subbatches* on key $k1$ - one subbatch for each touched partition (in this case, $SP1_a$ and $SP1_b$). These subbatches execute as individual subtransactions, each of which is single-sited and need not communicate with one another during execution. However, once each subtransaction has finished, they all must commit or abort together as a single unit, much like a distributed transaction but with significantly less communication cost. If one subtransaction must abort, then all must abort. Once the full transaction has committed, the output of each subbatch is *merged* back into a single batch. The process can then be repeated for $SP2$, splitting this new batch on key $k2$ instead. This process is similar to a transactional version of the MapReduce paradigm, with the split representing the Map phase, and merge representing the Reduce.

The key difference between this strategy and the data parallelism strategy from Section 4.2 is that the subtransactions all commit together, and that the subbatches are merged back together before the downstream transaction can begin. As a result, all transactional guarantees remain intact (i.e. atomicity is maintained). The merging of the subbatches after execution also allows for the dataflow graph to contain many different stored procedures partitioned on different keys. In between the transaction executions, the batch can be reshuffled to align to the next partitioning key.

While communication is kept to a minimum, each subbatch and transaction invocation must be sent to the appropriate partition on which it is set to execute. As a result, it is best to limit the split transaction to a single node, across several partitions (and thus across multiple cores). This keeps the communication cost of the 2PC coordination to a minimum, preserving the performance gains of the transaction split.

8.2 Automatic Design for Dataflow and Procedure Partitioning

Chapter 6 describes the preliminary work for automatic database design that takes into account state co-location, processing co-location, and load balancing. However, it stops short of considering two other design considerations: dataflow and processing partitioning.

The automation of processing partitioning is certainly feasible, and partially accomplished through workload traces to determine which tables each procedure accesses. Ideally, a database designer could detect the partitioning key of the accessed tables, and match that to the schema of the input stream tuples. The primary difficulty comes in the drastic increase in the search space for optimal state and processing locations.

Automatic dataflow partitioning is a much more sticky subject. We delved into some general guidelines for this partitioning strategy in Section 4.3.3, but these rules assume that a database engineer is responsible for determining where these transaction boundaries need to exist for semantic purposes. Leaving such decisions completely to automation opens up many potential correctness issues about which operations need to be executed as an atomic unit. Attempting such a feat would make for a fascinating research direction. Likely, the strategy would involve the assignment of operations to the largest number of transactions possible (i.e. each operation gets its own transaction) unless the user explicitly declares that two or more operations *must* execute together.

8.3 Multiversion Concurrency Control

S-Store is presently implemented on a shared-nothing architecture in which tuples are mutated directly in memory. To do so, the entire partition is locked during the mutation in order to ensure that the state is isolated from other transactions. This single-threaded concurrency control scheme is extremely light-weight and optimized for coarse-grained partitioning and state locking, and operates under the assumption that the processing is evenly distributed in order to gain best-possible performance. While this is an efficient use of memory and can maintain high performance under the correct circumstances, there are other strategies of concurrency control that may allow more flexibility in database design.

One concurrency control scheme that may be particularly well-suited for transactional streaming is multiversion concurrency control (MVCC) [33]. In multiversion concurrency control, rather than directly modifying state in-place, each time an update occurs, a new version of that state is created. This new version is marked with a unique identifier, usually a timestamp, which indicates the order in which state was modified to ensure that each transaction accesses the correct version. Eventually, older versions can be cleaned up as they become irrelevant. The downside to this approach

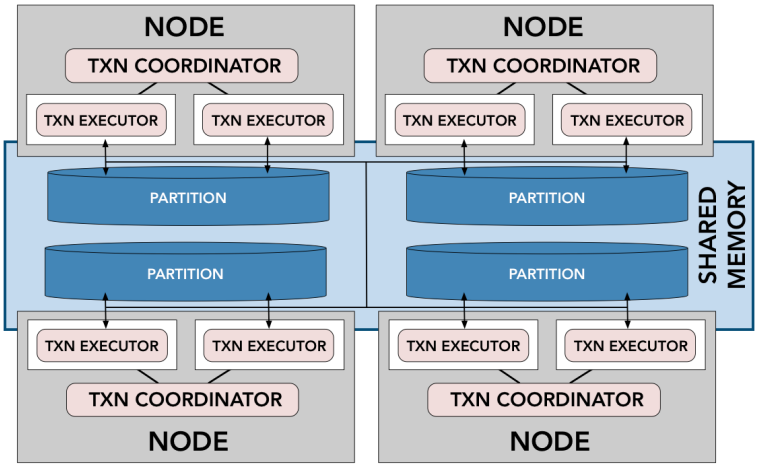


Figure 8.2: RDMA Architecture

is that these additional versions take additional space, and can make indexing and serializability implementation more difficult.

MVCC has additional advantages for transactional streaming. Correct execution of dataflow graphs relies not only on serializability of operations, but also on the proper order related to batch-id. When multiple procedures access the same state, our transaction model ensures that their transactions execute in dataflow and batch order. However, this does not guarantee that a piece of state is protected between transaction executions for the same batch on multiple stored procedures. To make such a guarantee, ordinarily one must place both stored procedures within a nested transaction. MVCC offers a new option for some situations, however: allow transactions for a given batch to access the older version of the state that matches its batch-id. Such a strategy may lead to interesting research directions to determine what constitutes "correctness" in this situation. For instance, if a downstream stored procedure updates a piece of state with batch-id b, and its upstream counterpart updates the same piece of state with batch-id $b + 1$, which update should persist? In our present transaction model, such an update is "first come first served;" the updates are executed serially in the order they arrive, regardless of batch-id. MVCC introduces flexibility, and with it comes the option to determine a different definition of correctness.

8.4 High-Speed Remote Memory Access (InfiniBand)

Shared-nothing architectures operate under the fundamental assumption that the network connections between machines are the core bottleneck to performance, and that network communication

should be avoided at all costs. However, new network technologies such as InfiniBand FDR 4x can place bandwidth at a similar level to memory access [36]. This has led to system redesigns and new architectures that take advantage of the network throughput in order to facilitate low-cost remote data access, removing the high performance barrier of distributed transactions [169] (Figure 8.2).

If streaming transactions were to be implemented on an architecture that takes advantage of low-cost remote memory access, there are a number of distributed design decisions that might fundamentally change. In such an environment, co-location of state that is accessed together becomes much less crucial. The cost model for distributing state and processing likely looks very different, and if throughput is stable no matter the distribution, more emphasis in minimizing latency may be necessary. The scheduling strategy is another element that may see significant changes in this scenario. All of these are interesting areas for exploration, and we would expect a very different architecture that could potentially yield significantly better performance for certain workloads.

Chapter 9

Conclusion

In this dissertation, we presented the motivation for creating transactional stream processing, a hybrid data model between streaming data management systems and main-memory OLTP systems. We proposed three core correctness requirements for transactional streaming: ACID, stream ordering, and exactly-once processing. We presented a novel transaction model able to integrate dataflow graphs into an OLTP model, allowing dataflow transactions from multiple sources and external transactions to all access the same shared mutable state. We implemented this data model in S-Store, a hybrid transactional streaming system built on the main-memory OLTP system H-Store. We demonstrated that integrating the two systems together provides much better performance on hybrid workloads with correctness requirements than multiple disparate systems combined.

With the merits of a single-node S-Store proven, we then moved on to the task of implementing a distribution model. S-Store inherits its distribution fundamentals from H-Store, but encounters new problems in maintaining performance with the presence of dataflow graphs. One of the biggest challenges is guaranteeing global dataflow ordering for shared state in the distributed environment. S-Store uses a novel transaction scheduling method called distributed localized scheduling in order to maintain dataflow ordering while retaining strong performance, particularly for pipeline parallelized workloads. By attaching immutable stream data to stored procedure invocations, the state can be moved transactionally in a way that is asynchronous, thus avoiding major performance overhead. We show that our implementation choices and design guidelines can lead to a scalable system that still provides our correctness guarantees.

Transactional streaming is particularly well-suited for streaming data ingestion systems. We discussed core requirements for such an ETL system in order to ensure correct state output. We also described a potential Streaming ETL architecture. Additionally, we implemented a streaming data ingestion component for the BigDAWG polystore, breaking down proper querying when data spans

both the ingestion and warehouse components within BigDAWG.

One of the more interesting areas of current work for S-Store is the creation of an automatic database designer. The first step is to recognize the principles of database design that provide best performance, and create a cost model that weighs those against one another. Using that cost model, it becomes possible to utilize graph partitioning algorithms to search for a near-optimal design that finds locations for state and processing. There is a variety of existing work on automatic database designers, but the unique combination of mutable state and push-based dataflow processing leads to interesting challenges.

S-Store also has a number of future avenues for research, particularly in regards to state-of-the-art database architectures and cluster hardware. While the initial distribution and database design decisions are largely based on a shared-nothing architecture, newer cluster hardware such as high-bandwidth remote memory access may challenge many assumptions and offer exciting performance opportunities. Similarly, multi-version concurrency control can lead to a reconsideration of our core correctness guarantees, with potential to introduce a second generation of modeling transactional streaming.

Together, this dissertation collects what is hopefully the beginnings of a new, fascinating class of transactional stream processing systems, which can provide the best of both the worlds of streaming and OLTP.

Bibliography

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

[2] Apache Hadoop. http://hadoop.apache.org.

[3] Apache Samza. http://samza.apache.org/.

[4] Apache Spark. http://spark.apache.org/.

[5] Esper. http://www.espertech.com/esper/.

[6] InfluxDB. https://www.influxdata.com/.

[7] Kappa Architecture. https://www.oreilly.com/ideas/questioning-the-lambda-architecture.

[8] Kdb+ Database and Language Primer. https://a.kx.com/q/d/primer.htm.

[9] Kx Systems. http://www.kx.com/.

[10] Lambda Architecture. http://lambda-architecture.net.

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

[12] VoltDB. http://www.voltdb.com/.

[13] ABADI, D., AHMAD, Y., BALAZINSKA, M., ÇETINTEMEL, U., CHERNIACK, M., HWANG, J., LINDNER, W., MASKEY, A., RASIN, A., RYVKINA, E., TATBUL, N., XING, Y., AND ZDONIK, S. The Design of the Borealis Stream Processing Engine. In *CIDR* (2005).

[14] ABADI, D., CARNEY, D., ÇETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal 12*, 2 (2003).

[15] ABADI, D. J. *Query Execution in Column-oriented Database Systems*. PhD thesis, Cambridge, MA, USA, 2008. AAI0820132.

[16] ADYA, A., GRUBER, R., LISKOV, B., AND MAHESHWARI, U. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. *SIGMOD Rec. 24*, 2 (May 1995), 23–34.

[17] AGRAWAL, S., CHAUDHURI, S., AND NARASAYYA, V. R. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB* (2000), vol. 2000, pp. 496–505.

[18] AKIDAU, T., BALIKOV, A., BEKIROGLU, K., CHERNYAK, S., HABERMAN, J., LAX, R., MCVEETY, S., MILLS, D., NORDSTROM, P., AND WHITTLE, S. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *PVLDB 6*, 11 (2013).

[19] AKIDAU, T., BRADSHAW, R., CHAMBERS, C., CHERNYAK, S., FERNÁNDEZ-MOCTEZUMA, R. J., LAX, R., MCVEETY, S., MILLS, D., PERRY, F., SCHMIDT, E., AND WHITTLE, S. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *Proc. VLDB Endow. 8*, 12 (Aug. 2015), 1792–1803.

[20] ALI, M. An Introduction to Microsoft SQL Server StreamInsight. In *Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research & Application* (2010), ACM, p. 66.

[21] ALI, M., CHANDRAMOULI, B., GOLDSTEIN, J., AND SCHINDLAUER, R. The Extensibility Framework in Microsoft StreamInsight. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering* (Washington, DC, USA, 2011), ICDE '11, IEEE Computer Society, pp. 1242–1253.

[22] ANANTHANARAYANAN, G., GHODSI, A., WANG, A., BORTHAKUR, D., KANDULA, S., SHENKER, S., AND STOICA, I. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 20–20.

[23] ANDRADE, H., GEDIK, B., WU, K.-L., AND PHILIP, S. Y. Scale-up Strategies for Processing High-Rate Data Streams in System S. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on* (2009), IEEE, pp. 1375–1378.

[24] ARASU, A., BABCOCK, B., BABU, S., CIESLEWICZ, J., DATAR, M., ITO, K., MOTWANI, R., SRIVASTAVA, U., AND WIDOM, J. STREAM: The Stanford Data Stream Management System. In *Data Stream Management: Processing High-Speed Data Streams* (2004).

[25] ARASU, A., BABU, S., AND WIDOM, J. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal The International Journal on Very Large Data Bases 15*, 2 (2006), 121–142.

[26] ARASU, A., CHERNIACK, M., GALVEZ, E., MAIER, D., MASKEY, A. S., RYVKINA, E., STONEBRAKER, M., AND TIBBETTS, R. Linear Road: A Stream Data Management Benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30* (2004), VLDB Endowment, pp. 480–491.

[27] ASTRAHAN, M. M., BLASGEN, M. W., CHAMBERLIN, D. D., ESWARAN, K. P., GRAY, J. N., GRIFFITHS, P. P., KING, W. F., LORIE, R. A., MCJONES, P. R., MEHL, J. W., PUTZOLU, G. R., TRAIGER, I. L., WADE, B. W., AND WATSON, V. System R: Relational Approach to Database Management. *ACM Trans. Database Syst. 1*, 2 (June 1976), 97–137.

[28] BABCOCK, B., BABU, S., MOTWANI, R., AND DATAR, M. Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2003), SIGMOD '03, ACM, pp. 253–264.

[29] BABU, S., AND WIDOM, J. Continuous Queries over Data Streams. *SIGMOD Rec. 30*, 3 (Sept. 2001), 109–120.

[30] BALAZINSKA, M., BALAKRISHNAN, H., MADDEN, S. R., AND STONEBRAKER, M. Fault-tolerance in the Borealis Distributed Stream Processing System. *ACM TODS 33*, 1 (2008).

[31] BAULIER, J., BOHANNON, P., GOGATE, S., GUPTA, C., AND HALDAR, S. DataBlitz Storage Manager: Main-memory Database Performance for Critical Applications. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1999), SIGMOD '99, ACM, pp. 519–520.

[32] BERNSTEIN, P. A., AND GOODMAN, N. Timestamp-based Algorithms for Concurrency Control in Distributed Database Systems. In *Proceedings of the Sixth International Conference on Very Large Data Bases - Volume 6* (1980), VLDB '80, VLDB Endowment, pp. 285–300.

[33] BERNSTEIN, P. A., AND GOODMAN, N. Multiversion Concurrency Control - Theory and Algorithms. *ACM Trans. Database Syst. 8*, 4 (Dec. 1983), 465–483.

[34] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[35] BERNSTEIN, P. A., SHIPMAN, D. W., AND WONG, W. S. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Trans. Softw. Eng. 5*, 3 (May 1979), 203–216.

[36] BINNIG, C., CROTTY, A., GALAKATOS, A., KRASKA, T., AND ZAMANIAN, E. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow. 9*, 7 (Mar. 2016), 528–539.

[37] BOTAN, I., DERAKHSHAN, R., DINDAR, N., HAAS, L., MILLER, R. J., AND TATBUL, N. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. *PVLDB 3*, 1 (2010).

[38] BOTAN, I., FISCHER, P. M., KOSSMANN, D., AND TATBUL, N. Transactional Stream Processing. In *EDBT* (2012).

[39] BOUTIN, E., EKANAYAKE, J., LIN, W., SHI, B., ZHOU, J., QIAN, Z., WU, M., AND ZHOU, L. Apollo: Scalable and Coordinated Scheduling for Cloud-scale Computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 285–300.

[40] BROWN, P. G. Overview of sciDB: Large Scale Array Storage, Processing and Analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 963–968.

[41] BROWN, S. D., FRANCIS, R. J., ROSE, J., AND VRANESIC, Z. G. *Field-Programmable Gate Arrays*, vol. 180. Springer Science & Business Media, 2012.

[42] CAREY, M. J., HAAS, L. M., SCHWARZ, P. M., ARYA, M., CODY, W., FAGIN, R., FLICKNER, M., LUNIEWSKI, A. W., NIBLACK, W., PETKOVIC, D., ET AL. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *Research Issues in Data Engineering, 1995: Distributed Object Management, Proceedings. RIDE-DOM'95. Fifth International Workshop on* (1995), IEEE, pp. 124–131.

[43] CAREY, M. J., JAUHARI, R., AND LIVNY, M. On Transaction Boundaries in Active Databases: A Performance Perspective. *IEEE Transactions on Knowledge and Data Engineering 3*, 3 (Sept 1991), 320–336.

[44] CARNEY, D., ÇETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., SEIDMAN, G., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. Monitoring Streams: A New Class of Data Management Applications. In *Proceedings of the 28th International Conference on Very Large Data Bases* (2002), VLDB '02, VLDB Endowment, pp. 215–226.

[45] CATTELL, R. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec. 39*, 4 (May 2011), 12–27.

[46] CHA, S. K., AND SONG, C. P*TIME: Highly Scalable OLTP DBMS for Managing Update-intensive Stream Workload. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (2004), VLDB '04, VLDB Endowment, pp. 1033–1044.

[47] CHANDRAMOULI, B., GOLDSTEIN, J., BARNETT, M., DELINE, R., FISHER, D., PLATT, J. C., TERWILLIGER, J. F., AND WERNSING, J. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. *PVLDB 8*, 4 (2014).

[48] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S., RAMAN, V., REISS, F., AND SHAH, M. A. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR* (2003).

[49] CHAUDHURI, S., DATAR, M., AND NARASAYYA, V. Index Selection for Databases: A Hardness Study and a Principled Heuristic Solution. *IEEE Transactions on Knowledge and Data Engineering 16*, 11 (2004), 1313–1323.

[50] CHEN, J., DEWITT, D. J., TIAN, F., AND WANG, Y. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD* (2000).

[51] CHEN, J., JINDEL, S., WALZER, R., SEN, R., JIMSHELEISHVILLI, N., AND ANDREWS, M. The MemSQL Query Optimizer: A Modern Optimizer for Real-time Analytics in a Distributed Database. *Proc. VLDB Endow. 9*, 13 (Sept. 2016), 1401–1412.

[52] CHEUNG, A., MADDEN, S., ARDEN, O., AND MYERS, A. C. Automatic Partitioning of Database Applications. *Proc. VLDB Endow. 5*, 11 (July 2012), 1471–1482.

[53] CURINO, C., JONES, E., ZHANG, Y., AND MADDEN, S. Schism: A Workload-driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010), 48–57.

[54] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[55] DEBRABANT, J., PAVLO, A., TU, S., STONEBRAKER, M., AND ZDONIK, S. Anti-caching: A New Approach to Database Management System Architecture. *Proc. VLDB Endow. 6*, 14 (Sept. 2013), 1942–1953.

[56] DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., AND WOOD, D. A. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1984), SIGMOD '84, ACM, pp. 1–8.

[57] DIACONU, C., FREEDMAN, C., ISMERT, E., LARSON, P.-A., MITTAL, P., STONECIPHER, R., VERMA, N., AND ZWILLING, M. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD* (2013).

[58] DOWDY, L. W., AND FOSTER, D. V. Comparative Models of the File Assignment Problem. *ACM Computing Surveys (CSUR) 14*, 2 (1982), 287–313.

[59] DU, J., MEEHAN, J., TATBUL, N., AND ZDONIK, S. Towards Dynamic Data Placement for Polystore Ingestion. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics* (New York, NY, USA, 2017), BIRTE '17, ACM, pp. 2:1–2:8.

[60] EICH, M. H. Parallel Architectures for Database Systems. IEEE Press, Piscataway, NJ, USA, 1989, ch. A Classification and Comparison of Main Memory Database Recovery Techniques, pp. 417–424.

[61] ELMORE, A., DUGGAN, J., STONEBRAKER, M., BALAZINSKA, M., CETINTEMEL, U., GADEPALLY, V., HEER, J., HOWE, B., KEPNER, J., KRASKA, T., MADDEN, S., MAIER, D., MATTSON,

T., PAPADOPOULOS, S., PARKHURST, J., TATBUL, N., VARTAK, M., AND ZDONIK, S. A Demonstration of the BigDAWG Polystore System. *The Proceedings of the VLDB Endowment (PVLDB) 8*, 12 (August 2015).

[62] ET AL, A., BITTON, D., BROWN, M., CATELL, R., CERI, S., CHOU, T., DEWITT, D., GAWLICK, D., GARCIA-MOLINA, H., GOOD, B., GRAY, J., HOMAN, P., JOLLS, B., LUKES, T., LAZOWSKA, E., NAUMAN, J., PONG, M., SPECTOR, A., TRIEBER, K., SAMMER, H., SERLIN, O., STONEBRAKER, M., REUTER, A., AND WEINBERGER, P. A Measure of Transaction Processing Power. *Datamation 31*, 7 (Apr. 1985), 112–118.

[63] FAERBER, F., KEMPER, A., KE LARSON, P., LEVANDOSKI, J., NEUMANN, T., AND PAVLO, A. Main Memory Database Systems. *Foundations and Trends in Databases 8*, 1-2 (2017), 1–130.

[64] FÄRBER, F., CHA, S. K., PRIMSCH, J., BORNHÖVD, C., SIGG, S., AND LEHNER, W. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Rec. 40*, 4 (Jan. 2012), 45–51.

[65] FERNANDEZ, R. C., MIGLIAVACCA, M., KALYVIANAKI, E., AND PIETZUCH, P. Integrating Scale-out and Fault-tolerance in Stream Processing using Operator State Management. In *SIGMOD* (2013).

[66] FERNANDEZ, R. C., MIGLIAVACCA, M., KALYVIANAKI, E., AND PIETZUCH, P. Making State Explicit for Imperative Big Data Processing. In *USENIX ATC* (2014).

[67] FERNANDEZ, R. C., PIETZUCH, P., KREPS, J., NARKHEDE, N., RAO, J., KOSHY, J., LIN, D., RICCOMINI, C., AND WANG, G. Liquid: Unifying Nearline and Offline Big Data Integration. In *CIDR* (2015).

[68] FRANKLIN, M. J. Concurrency Control and Recovery, 1997.

[69] GEDIK, B., ANDRADE, H., AND WU, K.-L. A Code Generation Approach to Optimizing High-Performance Distributed Data Stream Processing. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management* (2009), ACM, pp. 847–856.

[70] GOLAB, L., AND JOHNSON, T. Consistency in a Stream Warehouse. In *CIDR* (2011).

[71] GOLAB, L., AND OZSU, T. Issues in Data Stream Management. *ACM SIGMOD Record 32*, 2 (2003).

[72] GORAWSKI, M., GORAWSKA, A., AND PASTERAK, K. A Survey of Data Stream Processing Tools. 295–303.

[73] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[74] GROVER, R., AND CAREY, M. Data Ingestion in AsterixDB. In *EDBT* (March 2015).

[75] HAMMER, M., AND NIAMIR, B. A Heuristic Approach to Attribute Partitioning. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1979), SIGMOD '79, ACM, pp. 93–101.

[76] HARIZOPOULOS, S., ABADI, D. J., MADDEN, S., AND STONEBRAKER, M. OLTP Through the Looking Glass, and What We Found There. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2008), SIGMOD '08, ACM, pp. 981–992.

[77] HILDRUM, K., DOUGLIS, F., WOLF, J. L., YU, P. S., FLEISCHER, L., AND KATTA, A. Storage Optimization for Large-Scale Distributed Stream-Processing Systems. *ACM Transactions on Storage (TOS) 3*, 4 (2008), 5.

[78] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 295–308.

[79] HWANG, J.-H., BALAZINSKA, M., RASIN, A., CETINTEMEL, U., STONEBRAKER, M., AND ZDONIK, S. High-Availability Algorithms for Distributed Stream Processing. In *ICDE* (2005).

[80] JAGADISH, H. V., MUMICK, I. S., AND SILBERSCHATZ, A. View Maintenance Issues for the Chronicle Data Model (Extended Abstract). In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (New York, NY, USA, 1995), PODS '95, ACM, pp. 113–124.

[81] JAIN, N., MISHRA, S., SRINIVASAN, A., GEHRKE, J., WIDOM, J., BALAKRISHNAN, H., ÇETINTEMEL, U., CHERNIACK, M., TIBBETTS, R., AND ZDONIK, S. Towards a Streaming SQL Standard. *PVLDB 1*, 2 (2008).

[82] JOHNSON, R., PANDIS, I., HARDAVELLAS, N., AILAMAKI, A., AND FALSAFI, B. Shore-MT: A Scalable Storage Manager for the Multicore Era. 2435.

[83] JOHNSON, T., MUTHUKRISHNAN, M. S., SHKAPENYUK, V., AND SPATSCHECK, O. Query-aware Partitioning for Monitoring Massive Network Data Streams. In *SIGMOD* (2008).

[84] JORG, T., AND DESSLOCH, S. Near Real-Time Data Warehousing using State-of-the-art ETL Tools. In *BIRTE Workshop* (August 2009).

[85] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P. C., MADDEN, S., STONEBRAKER, M., ZHANG, Y., HUGG, J., AND ABADI, D. J. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *PVLDB 1*, 2 (2008).

[86] KAO, B., AND GARCIA-MOLINA, H. An Overview of Real-Time Database Systems. Technical Report 1993-6, Stanford University, 1993.

[87] KARYPIS, G., AND KUMAR, V. METIS – Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0.

[88] KARYPIS, G., AND KUMAR, V. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on scientific Computing 20*, 1 (1998), 359–392.

[89] KEMPER, A., AND NEUMANN, T. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on* (2011), IEEE, pp. 195–206.

[90] KERNIGHAN, B. W., AND LIN, S. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal 49*, 2 (1970), 291–307.

[91] KERSTEN, M. L., APERS, P. M., HOUTSMA, M. A., VAN KUYK, E. J., AND VAN DE WEG, R. L. A Distributed, Main-Memory Database Machine. In *Database Machines and Knowledge Base Machines* (1988), vol. 43 of *The Kluwer International Series in Engineering and Computer Science (Parallel Processing and Fifth Generation Computing)*, Springer.

[92] KHANDEKAR, R., HILDRUM, K., PAREKH, S., RAJAN, D., WOLF, J., WU, K.-L., ANDRADE, H., AND GEDIK, B. Cola: Optimizing stream processing applications via graph partitioning. In *Middleware 2009*. Springer, 2009, pp. 308–327.

[93] KHANDEKAR, R., RAO, S., AND VAZIRANI, U. Graph Partitioning Using Single Commodity Flows. In *Proceedings of the Thirty-eighth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2006), STOC '06, ACM, pp. 385–390.

[94] KIM, Y.-K., AND SON, S. H. An Approach Towards Predictable Real-Time Transaction Processing. *Fifth Euromicro Workshop on Real-Time Systems* (1993), 70–75.

[95] KIMBALL, R., AND CASERTA, J. *The Data Warehouse ETL Toolkit*. Wiley Publishing, 2004.

[96] KREPS, J., NARKHEDE, N., AND RAO, J. Kafka: A Distributed Messaging System for Log Processing. In *NetDB Workshop* (2011).

[97] KULKARNI, S., BHAGAT, N., FU, M., KEDIGEHALLI, V., KELLOGG, C., MITTAL, S., PATEL, J. M., RAMASAMY, K., AND TANEJA, S. Twitter Heron: Stream Processing at Scale. In *SIGMOD* (2015).

[98] KUNG, H. T., AND ROBINSON, J. T. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst. 6*, 2 (June 1981), 213–226.

[99] LEE, J., MUEHLE, M., MAY, N., FRBER, F., SIKKA, V., PLATTNER, H., AND KRUEGER, J. High-Performance Transaction Processing in SAP HANA. 28–33.

[100] LEHMAN, T. J., AND CAREY, M. J. Query Processing in Main Memory Database Management Systems. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1986), SIGMOD '86, ACM, pp. 239–250.

[101] LERNER, A., AND SHASHA, D. The Virtues and Challenges of Ad Hoc + Streams Querying in Finance. *IEEE Data Engineering Bulletin 26*, 1 (2003).

[102] LI, K., AND NAUGHTON, J. F. Multiprocessor Main Memory Transaction Processing. In *Proceedings of the First International Symposium on Databases in Parallel and Distributed Systems* (Los Alamitos, CA, USA, 1988), DPDS '88, IEEE Computer Society Press, pp. 177–187.

[103] LIN, K. Consistency Issues in Real-Time Database Systems. In *[1989] Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume II: Software Track* (Jan 1989), vol. 2, pp. 654–661 vol.2.

[104] LIN, W., FAN, H., QIAN, Z., XU, J., YANG, S., ZHOU, J., AND ZHOU, L. STREAMSCOPE: Continuous Reliable Distributed Processing of Big Data Streams. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2016), NSDI'16, USENIX Association, pp. 439–453.

[105] LIU, J. W. S., LIN, K. J., SHIH, W. K., YU, A. C., CHUNG, J. Y., AND ZHAO, W. Algorithms for Scheduling Imprecise Computations. *Computer 24*, 5 (May 1991), 58–68.

[106] LIU, L., PU, C., AND TANG, W. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering 11*, 4 (July 1999), 610–628.

[107] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 135–146.

[108] MALVIYA, N., WEISBERG, A., MADDEN, S., AND STONEBRAKER, M. Rethinking Main Memory OLTP Recovery. In *2014 IEEE 30th International Conference on Data Engineering* (March 2014), pp. 604–615.

[109] MEEHAN, J., ASLANTAS, C., ZDONIK, S., TATBUL, N., AND DU, J. Data Ingestion for the Connected World. In *CIDR* (2017).

[110] MEEHAN, J., TATBUL, N., ZDONIK, S., ASLANTAS, C., CETINTEMEL, U., DU, J., KRASKA, T., MADDEN, S., MAIER, D., PAVLO, A., STONEBRAKER, M., TUFTE, K., AND WANG, H. S-Store: Streaming Meets Transaction Processing. *PVLDB 8*, 13 (2015), 2134–2145.

[111] MEEHAN, J., ZDONIK, S., TIAN, S., TIAN, Y., TATBUL, N., DZIEDZIC, A., AND ELMORE, A. Integrating Real-Time and Batch Processing in a Polystore. In *IEEE HPEC Conference* (September 2016).

[112] MENASC, D. A., AND NAKANISHI, T. Optimistic versus Pessimistic Concurrency Control Mechanisms in Database Management Systems. *Information Systems 7*, 1 (1982), 13 – 27.

[113] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst. 17*, 1 (Mar. 1992), 94–162.

[114] MÜHLBAUER, T., RÖDIGER, W., REISER, A., KEMPER, A., AND NEUMANN, T. ScyPer: Elastic OLAP Throughput on Transactional Data. In *Proceedings of the Second Workshop on Data Analytics in the Cloud* (New York, NY, USA, 2013), DanaC '13, ACM, pp. 11–15.

[115] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A Timely Dataflow System. In *SOSP* (2013).

[116] MURRAY, D. G., SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. CIEL: A Universal Execution Engine for Distributed Data-flow Computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 113–126.

[117] N. TATBUL, S. ZDONIK, J. MEEHAN, C. ASLANTAS, M. STONEBRAKER, K. TUFTE, C. GIOSSI, H. QUACH. Handling Shared, Mutable State in Stream Processing with Correctness Guarantees. *IEEE Data Engineering Bulletin, Special Issue on Next-Generation Stream Processing 38*, 4 (December 2015).

[118] NEUMEYER, L., ROBBINS, B., NAIR, A., AND KESARI, A. S4: Distributed Stream Computing Platform. In *KDCloud* (2010).

[119] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2008), SIGMOD '08, ACM, pp. 1099–1110.

[120] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 69–84.

[121] OZSOYOGLU, G., AND SNODGRASS, R. T. Temporal and Real-Time Databases: A Survey. *IEEE Trans. on Knowl. and Data Eng. 7*, 4 (Aug. 1995), 513–532.

[122] PAPADOPOULOS, S., DATTA, K., MADDEN, S., AND MATTSON, T. The TileDB Array Data Storage Manager. *Proc. VLDB Endow. 10*, 4 (Nov. 2016), 349–360.

[123] PAVLO, A. *On Scalable Transaction Execution in Partitioned Main Memory Database Management Systems*. PhD thesis, Brown University, Providence, RI, 5 2014.

[124] PAVLO, A., CURINO, C., AND ZDONIK, S. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *SIGMOD* (2012).

[125] PELKONEN, T., FRANKLIN, S., TELLER, J., CAVALLARO, P., HUANG, Q., MEZA, J., AND VEERARAGHAVAN, K. Gorilla: A Fast, Scalable, In-memory Time Series Database. *PVLDB 8*, 12 (2015), 1816–1827.

[126] PHILIP CHARLES JONES, E., J. ABADI, D., AND MADDEN, S. Low Overhead Concurrency Control for Partitioned Main Memory Databases. 603–614.

[127] PHIPPS, C., AND DAVIS, K. C. Automating Data Warehouse Conceptual Schema Design and Evaluation. In *DMDW* (2002), vol. 2, Citeseer, pp. 23–32.

[128] PHYSIONET. MIMIC II Data Set. `https://physionet.org/mimic2/`.

[129] POESS, M., RABL, T., JACOBSEN, H.-A., AND CAUFIELD, B. TPC-DI: The First Industry Benchmark for Data Integration. *Proc. VLDB Endow. 7*, 13 (Aug. 2014), 1367–1378.

[130] QIAN, Z., HE, Y., SU, C., WU, Z., ZHU, H., ZHANG, T., ZHOU, L., YU, Y., AND ZHANG, Z. TimeStream: Reliable Stream Computation in the Cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 1–14.

[131] RAMAKRISHNAN, R., AND GEHRKE, J. *Database Management Systems*, 3 ed. McGraw-Hill, Inc., New York, NY, USA, 2003.

[132] SCHMIDT, S., LEGLER, T., SCHALLER, D., AND LEHNER, W. Real-time Scheduling for Data Stream Management Systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)* (July 2005), pp. 167–176.

[133] SCHREIER, U., PIRAHESH, H., AGRAWAL, R., AND MOHAN, C. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. In *Proceedings of the 17th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1991), VLDB '91, Morgan Kaufmann Publishers Inc., pp. 469–478.

[134] SHASHA, D., LLIRBAT, F., SIMON, E., AND VALDURIEZ, P. Transaction Chopping: Algorithms and Performance Studies. *ACM Trans. Database Syst. 20*, 3 (Sept. 1995), 325–363.

[135] SHEN, Z., ET AL. CSA: Streaming Engine for Internet of Things. *IEEE Data Engineering Bulletin, Special Issue on Next-Generation Stream Processing 38*, 4 (December 2015).

[136] SIDIROURGOS, L., GONCALVES, R., KERSTEN, M., NES, N., AND MANEGOLD, S. Column-store Support for RDF Data Management: Not All Swans Are White. *Proc. VLDB Endow. 1*, 2 (Aug. 2008), 1553–1563.

[137] SILBERSCHATZ, A., KORTH, H. F., AND SUDARSHAN, S. *Database System Concepts*. McGraw-Hill, 2010.

[138] SONG, X., AND LIU, J. W. S. How Well Can Data Temporal Consistency Be Maintained? In *IEEE Symposium on Computer-Aided Control System Design* (March 1992), pp. 275–284.

[139] STANKOVIC, J. A. Misconceptions about Real-time Computing: A Serious Problem for Next-Generation Systems. *Computer 21*, 10 (Oct 1988), 10–19.

[140] STONEBRAKER, M., ABADI, D. J., BATKIN, A., CHEN, X., CHERNIACK, M., FERREIRA, M., LAU, E., LIN, A., MADDEN, S., O'NEIL, E., O'NEIL, P., RASIN, A., TRAN, N., AND ZDONIK, S. C-store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases* (2005), VLDB '05, VLDB Endowment, pp. 553–564.

[141] STONEBRAKER, M., ÇETINTEMEL, U., AND ZDONIK, S. The 8 Requirements of Real-time Stream Processing. *SIGMOD Rec. 34*, 4 (Dec. 2005), 42–47.

[142] STONEBRAKER, M., AND CETINTEMEL, U. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Data Engineering* (Washington, DC, USA, 2005), ICDE '05, IEEE Computer Society, pp. 2–11.

[143] STONEBRAKER, M., HELD, G., WONG, E., AND KREPS, P. The Design and Implementation of INGRES. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 189–222.

[144] STONEBRAKER, M., MADDEN, S., ABADI, D. J., HARIZOPOULOS, S., HACHEM, N., AND HELLAND, P. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases* (2007), VLDB '07, VLDB Endowment, pp. 1150–1160.

[145] SULLIVAN, M. Tribeca: A Stream Database Manager for Network Traffic Analysis. In *Proceedings of the 22th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1996), VLDB '96, Morgan Kaufmann Publishers Inc., pp. 594–.

[146] TATBUL, E. N. *Load Shedding Techniques for Data Stream Management Systems*. PhD thesis, Brown University, 2007.

[147] TATBUL, N., CETINTEMEL, U., AND ZDONIK, S. Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing. In *In VLDB* (2007).

[148] TERRY, D., GOLDBERG, D., NICHOLS, D., AND OKI, B. Continuous Queries over Append-only Databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1992), SIGMOD '92, ACM, pp. 321–330.

[149] THE TRANSACTION PROCESSING COUNCIL. TPC-C Benchmark (Revision 5.9.0). `http://www.tpc.org/tpcc/`, 2007.

[150] TIMESTEN TEAM, C. In-memory Data Management for Consumer Transactions: The TimesTen Approach. *SIGMOD Rec. 28*, 2 (June 1999), 528–529.

[151] TOSHNIWAL, A., TANEJA, S., SHUKLA, A., RAMASAMY, K., PATEL, J. M., KULKARNI, S., JACKSON, J., GADE, K., FU, M., DONHAM, J., BHAGAT, N., MITTAL, S., AND RYABOY, D. V. Storm @Twitter. In *SIGMOD* (2014).

[152] TRANSACTION PROCESSING PERFORMANCE COUNCIL (TPC). TPC Benchmark DI (Version 1.1.0). `http://www.tpc.org/tpcdi/`, Nov. 2014.

[153] VALIANT, L. G. A Bridging Model for Parallel Computation. *Commun. ACM 33*, 8 (Aug. 1990), 103–111.

[154] VASSILIADIS, P. A Survey of Extract-Transform-Load Technology. *International Journal of Data Warehousing and Mining 5*, 3 (July 2009).

[155] VASSILIADIS, P., AND SIMITSIS, A. Near Real-Time ETL. In *New Trends in Data Warehousing and Data Analysis*, S. Kozielski and R. Wrembel, Eds. Springer, 2009.

[156] VENKATARAMAN, S., PANDA, A., OUSTERHOUT, K., ARMBRUST, M., GHODSI, A., FRANKLIN, M. J., RECHT, B., AND STOICA, I. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 374–389.

[157] VERMA, A., CHO, B., ZEA, N., GUPTA, I., AND CAMPBELL, R. H. Breaking the MapReduce Stage Barrier. In *Cluster Comput* (2013), vol. 16, Springer US, pp. 191–206.

[158] VERMA, A., PEDROSA, L., KORUPOLU, M. R., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale Cluster Management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (Bordeaux, France, 2015).

[159] VRBSKY, S., AND LIN, K.-J. Recovering Imprecise Transactions with Real-time Constraints, 11 1988.

[160] WEIKUM, G., AND VOSSEN, G. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.

[161] WHITE, T. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[162] WHITNEY, A., AND SHASHA, D. Lots o' Ticks: Realtime High Performance Time Series Queries on Billions of Trades and Quotes. *SIGMOD Record 30*, 2 (2001), 617.

[163] WINGERATH, W., GESSERT, F., WITT, E., FRIEDRICH, S., AND RITTER, N. Real-Time Data Management for Big Data. In *EDBT* (2018).

[164] WLODARCZYK, T. W. Overview of Time Series Storage and Processing in a Cloud Environment. In *IEEE CloudCom Conference* (2012), pp. 625–628.

[165] XING, Y., HWANG, J.-H., ÇETINTEMEL, U., AND ZDONIK, S. Providing Resiliency to Load Variations in Distributed Stream Processing. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (2006), VLDB '06, VLDB Endowment, pp. 775–786.

[166] YU, P. S., WU, K.-L., LIN, K.-J., AND SON, S. H. On Real-time Databases: Concurrency Control and Scheduling. *Proceedings of the IEEE 82*, 1 (Jan 1994), 140–157.

[167] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 2–2.

[168] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., AND STOICA, I. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP* (2013).

[169] ZAMANIAN, E., BINNIG, C., HARRIS, T., AND KRASKA, T. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow. 10*, 6 (Feb. 2017), 685–696.

[170] ZHAO, X., GARG, S., QUEIROZ, C., AND BUYYA, R. A Taxonomy and Survey of Stream Processing Systems. *Book chapter* (2017), 177–200.

[171] ZHU, Y., AND SHASHA, D. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *Proceedings of the 28th International Conference on Very Large Data Bases* (2002), VLDB '02, VLDB Endowment, pp. 358–369.

[172] ZILIO, D. C. *Physical Database Design Decision Algorithms and Concurrent Reorganization for Parallel Database Systems*. PhD thesis, Toronto, Ont., Canada, Canada, 1998. AAINQ35386.

[173] ZILIO, D. C., JHINGRAN, A., AND PADMANABHAN, S. *Partitioning Key Selection for a Shared-Nothing Parallel Database System*. IBM TJ Watson Research Center, 1994.