Integrating Real-Time and Batch Processing in a Polystore

John Meehan, Stan Zdonik
Shaobo Tian, Yulong Tian
Brown University
{john,sbz}@cs.brown.edu

Nesime Tatbul
Intel Labs & MIT
tatbul@csail.mit.edu

Adam Dziedzic, Aaron Elmore
University of Chicago
{ady,aelmore}@cs.uchicago.edu

Abstract—**This paper describes a stream processing engine called S-Store and its role in the BigDAWG polystore. Fundamentally, S-Store acts as a frontend processor that accepts input from multiple sources, and massages it into a form that has eliminated errors (data cleaning) and translates that input into a form that can be efficiently ingested into BigDAWG. S-Store also acts as an intelligent router that sends input tuples to the appropriate components of BigDAWG. All updates to S-Store's shared memory are done in a transactionally consistent (ACID) way, thereby eliminating new errors caused by non-synchronized reads and writes. The ability to migrate data from component to component of BigDAWG is crucial. We have described a migrator from S-Store to Postgres that we have implemented as a first proof of concept. We report some interesting results using this migrator that impact the evaluation of query plans.**

I. Introduction

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 [1] vs. real-time/streaming [2], graphs [3] vs. arrays [4]). 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., [5]). As a result, integrated architectures (e.g., [6], [7]) and new hybrid systems (e.g., [8], [9], [10], [11]) 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 [12]. 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. In this paper, we describe how we realize this principle using a novel transactional stream processing system called S-Store [13]. We illustrate several important roles a streaming system such as S-Store can generally play in the heterogeneous setting of a polystore architecture such as BigDAWG, and provide an overview of our ongoing research and preliminary results in this area.

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.

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. S-Store, the first such system that we have been building at the ISTC for Big Data [13], [14], [15], has been designed for streaming applications with shared mutable state, such as real-time ETL. 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 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.

In the rest of this paper, we first provide a more detailed background on BigDAWG, S-Store, and the basic querying and migration primitives that we implemented to integrate them together. We then discuss our ongoing research topics

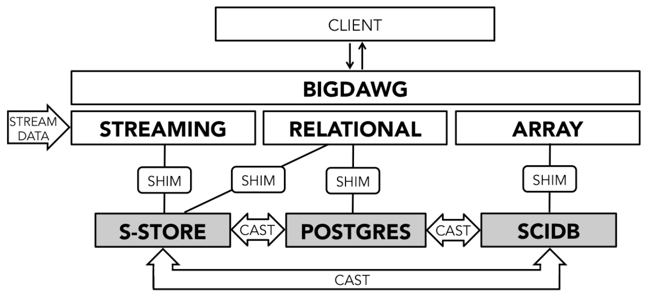


Fig. 1. BigDAWG 1.0 Architecture

and long-term motivations. Finally, we present preliminary experimental results evaluating the initial performance of the querying and migration primitives.

II. Background

A. BigDAWG Architecture

The BigDAWG polystore system unifies multiple systems with varying use cases and storage requirements under a single architecture [12]. 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 1.

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 [4]. 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.

Streaming systems play a very unique role in BigDAWG, as they manage any stream data that is pushed to the polystore. They perform continuous queries on all new data, and push information and notifications to a variety of output sources. All streaming systems fall under a streaming island, which is further described in Section III-A. We believe that streaming systems also have an important role in data ingestion in the general case, which we describe in Section III-B.

B. S-Store

Traditional stream processing systems were first created over a decade ago with the purpose of handling ever-changing ordered data in near-real-time [16], [17], [18]. These systems chained variants of standard relational operators that had been altered to handle unbounded streams. Due to the real-time nature of these systems, latency was of the highest priority, since the value of the results often degraded with time. As a result, disk access was minimized wherever possible, and in our view early streaming systems did not properly address storage-related issues. Streaming applications often require support for storage and historical queries, in which case an additional data storage engine needs to be used for strong data consistency guarantees.

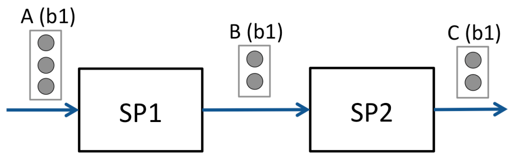


Fig. 2. A Simple S-Store Dataflow Graph

S-Store satisfies both requirements, providing low-latency, push-based processing seamlessly integrated with ACID data management. To accomplish this, S-Store ensures that all state, be it stream state, windows, or relational tables, may only be accessed within the context of a transaction. Streaming workloads are divided into *dataflow graphs*, directed acyclic graphs of disparate *stored procedures* (or SPs), atomic units of processing attached to both an input and output stream. A *transaction execution* (or TE) is defined as the execution of a stored procedure on an incoming atomic *batch* of input tuples. Each batch contains its own unique *batch-id*, which determines the order in which they can be processed. As a transaction execution commits, its output tuples are given the same batch-id and are placed onto its output stream. These can then serve as an input batch of the downstream stored procedure of the dataflow graph.

A very simple dataflow graph is shown in Figure 2. Here we have two stored procedures, $SP1$ and $SP2$, linked by a stream. $SP1$ takes as input batch $b1$, labeled $A(b1)$, and starts a transaction. This transaction produces $B(b1)$ as an output. $SP2$ then takes $B(b1)$ as input, and produces a transaction that has $C(b1)$ as an output. While $A(b1)$, $B(b1)$, and $C(b1)$ all have the same batch-id ($b1$), they will contain different tuples (transformations on the original batch).

S-Store's contributions are best summarized through the combination of its three data processing guarantees:

1. ***ACID*** guarantees for individual transactions (both OLTP and streaming). Like conventional OLTP systems, each transaction (OLTP or streaming) takes the database from one consistent state to another.
2. ***Ordered Execution*** guarantees for dataflow graphs of streaming transactions. Streaming data contains an inherent order, and processing on streaming data requires multiple consecutive steps. Streaming transactions must be scheduled in a way that preserves those orderings.
3. ***Exactly-Once Processing*** guarantees for streams (i.e., no

loss or duplication). This is particularly relevant in the case of failure to ensure that the same stored procedure does not execute on the same batch multiple times, and no transactions are lost in the failure.

Because lightweight transactions are a must for a streaming system with ACID guarantees, S-Store is built on top of H-Store, a high-throughput main-memory OLTP engine [19]. S-Store inherits the distributed, shared-nothing architecture of H-Store, as well as its command log recovery mechanism. All state (including streams and windows) are implemented as relational tables. Dataflow graphs of stored procedures are implemented via *partition engine triggers,* or PE triggers. When a new batch of tuples is inserted onto a stream with a PE trigger, a new transaction execution of the downstream stored procedure is invoked on that batch. A *streaming scheduler* ensures that S-Store's ordering guarantees are maintained while coordinating parallel processing on as many transactions as possible.

S-Store fills two roles within the BigDAWG polystore. As a streaming system that ingests data and runs continuous queries, S-Store clearly fits under the streaming island. However, because S-Store is SQL-based at its core, it is also able to serve as a main-memory OLTP engine for BigDAWG under the relational island. Thus, S-Store uses shims to connect to both islands, as illustrated in Figure 1.

C. Example Use-Cases

1) MIMIC: MIMIC II is an ICU data set containing clinical data obtained from hospitals and physiological vital sign data for ICU patients [5]. Due to the diverse nature of the data set, MIMIC II is one initial use case for the BigDAWG Polystore System.

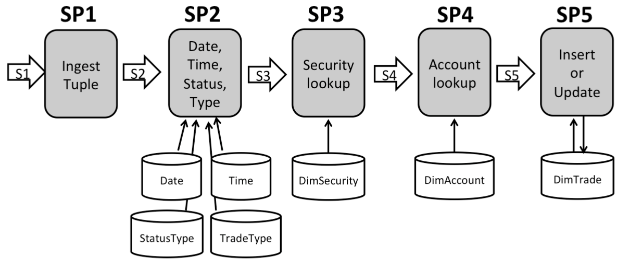
One particularly interesting aspect of this data set is the time series signal data representing patient vital signs. If this signal data is captured and analyzed in real-time, it is possible to perform interactive queries that simulate emergency situations. Using S-Store, it is trivial to construct a dataflow graph capable of detecting unusual shifts or patterns in the signal data. For instance, if the weighted average of a patient's Pulmonary Arterial Pressure (PAP) is detected to be under a specific threshold, S-Store can create an alert for medical professionals. More complicated queries, such as detecting irregular patterns in ECG signal data, are also possible.

In addition to these real-time alerts, S-Store is capable of cleaning and formatting incoming tuples for future ingestion into a long-term storage engine. Oftentimes time-series data is best stored in an array database such as SciDB, as it is easy to consider the patient, type of waveform, and time information each as its own dimension. S-Store can transform incoming tuples to suit the needs of SciDB, and use BigDAWG's migration functionality to bulk load those tuples into disk-based array storage.

2) TPC-DI: Data integration is a requirement for any database system when ingesting new information, often referred to as Extraction-Transformation-Loading (ETL) processes. ETL processes are primarily responsible for 1) the extraction of data from a variety of sources, 2) the transformation of raw source data to match the structure of the data in the target system, and 3) the loading of the altered data into the target system [20].

```
SP1:  INGEST Stream Tuple(s) FROM Input Stream
SP2:  SELECT CreateDateID FROM  DimDate
      SELECT CreateTimeID FROM DimTime
      SELECT Status FROM StatusType
      SELECT Type FROM TradeType
SP3:  SELECT SecurityID, CompanyID FROM DimSecurity
SP4:  SELECT AccountID, CustomerID, BrokerID
        FROM DimAccount
SP5:  INSERT Finished Tuple INTO DimTrade
```

(a) Pseudo-SQL of DimTrade Ingestion [21]



(b) Dataflow Graph of DimTrade Ingestion

Fig. 3. Ingestion of DimTrade Table in TPC-DI Benchmark

Traditionally, data integration involves loading flat files into a database system, allowing large quantities of data to be collected before bulk loading into the system. However, there are obvious latency benefits to instead ingesting the data as it arrives.

TPC-DI is a data integration benchmark created to measure the performance of various enterprise-level integration solutions [22]. The benchmark mimics a retail brokerage firm, and focuses on extracting and combining data from a variety of sources and source formats (e.g. CSV, XML), transforming them into one unified data model and loading the results into a data store.

While TPC-DI is designed with traditional ETL in mind, it can be easily modified to represent a streaming ETL workload instead. Take the queries associated with ingesting the DimTrade table, for example (Figure 3(a)). If these tuples arrive in batches on a regular basis rather than a full flat file, then the queries can be modeled as a streaming workload. Ordinarily, running the workload as a single transaction in a shared-nothing database would require one large distributed transaction. This is because the queries involved retrieve data from multiple tables, each of which must be partitioned on a different key. However, by dividing the process into five operations (Figure 3(b)), S-Store can instead perform the operations incrementally, processing each tuple in five smaller single-sited transactions while still providing the correct result. This results in quicker access to incremental portions of the end result and provides opportunities for parallelism, while keeping correctness intact.

III. Ongoing Research

Both S-Store and BigDAWG are ongoing research projects, with several areas of active development. Below we describe some areas of future work.

A. *The Role of Streaming Island in BigDAWG*

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 [23] or Apache Storm [24]. 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 [25].

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.

B. *Streaming ETL*

A polystore such as BigDAWG provides an opportunity to reconsider the entire data ingestion process. Historically, data ingestion and ETL is an under-served portion of data storage and analytics, and we believe that there are improvements to be made by integrating a streaming system into the process. Typically, ETL is performed in large batches. Data is collected throughout the day, and stored in flat files to be loaded all at once. A series of operations are then performed on this incoming data in order to mold it into the data schema of the target system. One obvious drawback to this approach is that the data is not available on the target system until an entire batch has been collected, which can take hours or even days

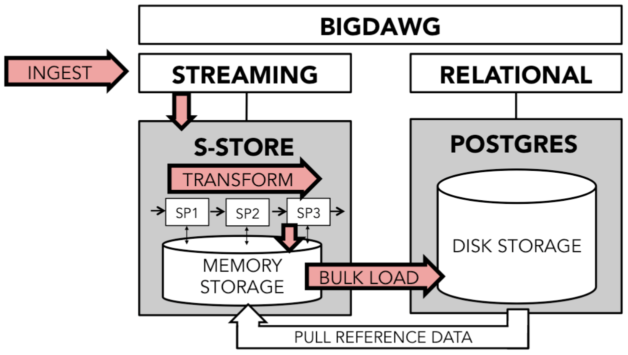


Fig. 4. Streaming ETL Example

[20]. By instead using a streaming system to model the ETL process, it is possible to do the data cleaning and migration as new data arrives rather than waiting to complete everything in bulk.

Streaming ETL divides the data into a sequence of well-defined chunks of configurable size. Each chunk should be processed and loaded as an atomic unit, i.e., partial chunks should not be visible to backend engines for querying. Furthermore, chunks should be durably maintained until at least their backend loading is successfully completed and acknowledged by the target system. Transformations on the chunks may also require reading and writing shared tables in a concurrent manner. Last but not least, for ensuring correct streaming semantics, processing chunks exactly once, in the right order is required. Therefore, transactional processing support is crucial for streaming ETL.

Due to its transactional properties, S-Store in particular is very well-suited to streaming ETL. Data ingestion naturally involves shared, mutable state, since references to previously ingested state is required for many operations. For instance, one common operation in the relational data transformation process is to look up and populate a foreign key reference to a related table. For this operation to succeed, the foreign key row must already be populated in the secondary table, and to guarantee correctness, all state reads and writes must be done through ACID transactions.

S-Store is scalable, and thus can serve as a streaming ingestion engine for many diverse systems under the BigDAWG polystore. As data is transformed, it can then be incrementally migrated in batches to the appropriate engine using BigDAWG's cast operators. From beginning to end, the entire process can be done as push-based operations, meaning that the destination systems will automatically be fed new data as it becomes available.

C. *Cross-System Data Movement and Caching*

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 4). Ideally, S-Store is able to perform transformations on incoming data independently of Postgres. However, as mentioned in Section III-B, 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:

(i) **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 query will be run in the near future.

(ii) **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.

(iii) **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. One future avenue of research is 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.

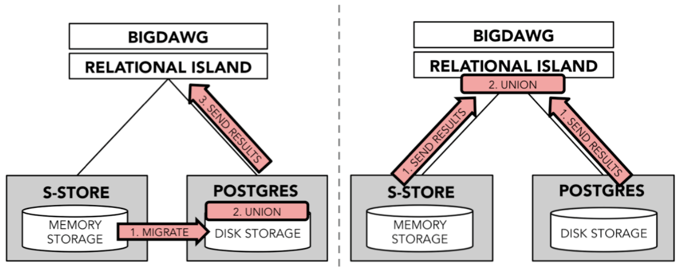
IV. Initial Results

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?

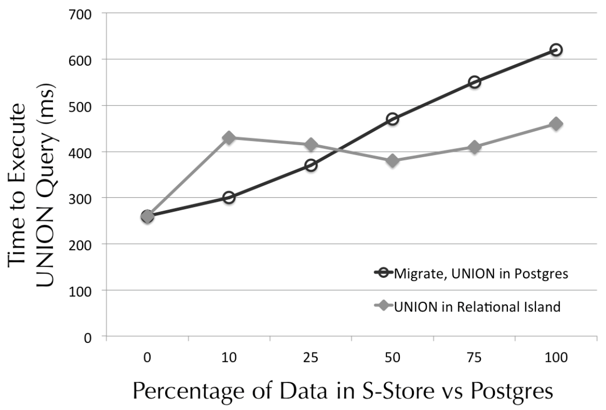
In order to compare potential query plans, we have constructed a simple experiment that compares two query plans for a UNION query. Let's assume that S-Store is being used as an ingestion engine for the ORDERS table of the TPC-C workload, eventually migrating tuples into Postgres [1] [26]. A user wishes 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(a)) 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.

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



(a) Migration Query Plan vs. UNION in Island Query Plan



(b) Comparison for UNION Queries

Fig. 5. Migration Evaluation for UNION Queries

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(b), 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 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.

V. Related Work

BigDAWG has many parallels with federated database systems like Garlic [27]. 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.

ETL systems have been around for many years. S-Store should be responsible for this important function in BigDAWG. Traditional ETL is typically done as a series of batch processing steps, each of which dumps its results to a file that is accessed by the next element in the pipeline [20]. This writing of files is very slow. S-Store processes tuples as they arrive, and pushes these intermediate results downstream to the next element without writing these results to a file. This results in near-real-time ETL.

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 [6] 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 [25]). Fernandez et al. also propose Liquid - an extended architecture similar to kappa [11]. 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.

There are a number of systems that have explicitly been designed for handling *hybrid workloads* that include real-time processing. Examples include Spark Streaming [23], Microsoft Trill [9], and Google Dataflow [10]. These all support batch, micro-batch, and streaming workloads in general, but in a more homogeneous setting compared to S-Store. Also, being analytical systems, they provide weaker transactional guarantees than S-Store.

VI. Conclusions

In this paper, we described a polystore called BigDAWG and the role of a streaming engine in BigDAWG. We also describe S-Store as a particular streaming engine that has a sophisticated model of shared memory. S-Store supports transactional guarantees, making the system much more reliable for managing shared state. This is important so that inconsistent updates do not work 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. Data movement between the component systems of a polystore is quite fundamental. Thus, we have concentrated on the topic of data migration in this paper. We have shown a simple experiment that supports the idea that the cost of data migration depends strongly on the amount of data that is moved.

VII. Acknowledgments

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

References

[1] "Apache Hadoop," http://hadoop.apache.org.
[2] "Apache Storm," http://storm.apache.org.
[3] G. Malewicz *et al.*, "Pregel: A System for Large-scale Graph Processing," in *SIGMOD*, 2010.
[4] M. Stonebraker *et al.*, "SciDB: A Database Management System for Applications with Complex Analytics," *Computing in Science and Engineering*, vol. 15, no. 3, pp. 54–62, 2013.
[5] PhysioNet, "MIMIC II Data Set," https://physionet.org/mimic2/.
[6] "Lambda Architecture," http://lambda-architecture.net.
[7] "Kappa Architecture," https://www.oreilly.com/ideas/questioning-the-lambda-architecture.
[8] "Apache Spark," http://spark.apache.org.
[9] B. Chandramouli *et al.*, "Trill: A High-Performance Incremental Query Processor for Diverse Analytics," *PVLDB*, vol. 8, no. 4, pp. 401–412, 2014.
[10] T. Akidau *et al.*, "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing," *PVLDB*, vol. 8, no. 12, pp. 1792–1803, 2015.
[11] R. C. Fernandez *et al.*, "Liquid: Unifying Nearline and Offline Big Data Integration," in *CIDR*, 2015.

[12] A. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, S. Madden, D. Maier, T. Mattson, S. Papadopoulos, J. Parkhurst, N. Tatbul, M. Vartak, and S. Zdonik, "A Demonstration of the BigDAWG Polystore System," *The Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 12, August 2015.

[13] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang, "S-Store: Streaming Meets Transaction Processing," *PVLDB*, vol. 8, no. 13, pp. 2134–2145, 2015.

[14] U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, E. Sutherland, N. Tatbul, K. Tufte, H. Wang, and S. Zdonik, "S-Store: A Streaming NewSQL System for Big Velocity Applications (Demonstration)," in *International Conference on Very Large Data Bases (VLDB'14)*, Hangzhou, China, September 2014.

[15] N. Tatbul *et al.*, "Handling Shared, Mutable State in Stream Processing with Correctness Guarantees," *IEEE Data Engineering Bulletin*, to appear.

[16] D. Abadi *et al.*, "Aurora: A New Model and Architecture for Data Stream Management," *VLDB Journal*, vol. 12, no. 2, 2003.

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

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

[19] R. Kallman *et al.*, "H-Store: A High-Performance, Distributed Main Memory Transaction Processing System," *PVLDB*, vol. 1, no. 2, 2008.

[20] P. Vassiliadis, "A Survey of Extract-Transform-Load Technology," *IJDWM*, vol. 5, no. 3, pp. 1–27, 2009.

[21] Transaction Processing Performance Council (TPC), "TPC Benchmark DI (Version 1.1.0)," http://www.tpc.org/tpcdi/, Nov. 2014.

[22] M. Poess, T. Rabl, H.-A. Jacobsen, and B. Caufield, "TPC-DI: The First Industry Benchmark for Data Integration," *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1367–1378, Aug. 2014.

[23] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: Fault-tolerant Streaming Computation at Scale," in *SOSP*, 2013, pp. 423–438.

[24] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy, "Storm @Twitter," in *SIGMOD*, 2014, pp. 147–156.

[25] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A Distributed Messaging System for Log Processing," in *NetDB Workshop*, 2011.

[26] The Transaction Processing Council, "TPC-C Benchmark (Revision 5.9.0)," http://www.tpc.org/tpcc/, 2007.

[27] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic *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*. IEEE, 1995, pp. 124–131.