Towards Dynamic Data Placement for Polystore Ingestion

Jiang Du
University of Toronto
jdu@cs.toronto.edu

John Meehan
Brown University
john@cs.brown.edu

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

Stan Zdonik
Brown University
sbz@cs.brown.edu

ABSTRACT

Integrating low-latency data streaming into data warehouse architectures has become an important enhancement to support modern data warehousing applications. In these architectures, heterogeneous workloads with data ingestion and analytical queries must be executed with strict performance guarantees. Furthermore, the data warehouse may consists of multiple different types of storage engines (a.k.a., polystores or multi-stores). A paramount problem is data placement; different workload scenarios call for different data placement designs. Moreover, workload conditions change frequently. In this paper, we provide evidence that a dynamic, workload-driven approach is needed for data placement in polystores with low-latency data ingestion support. We study the problem based on the characteristics of the TPC-DI benchmark in the context of an abbreviated polystore that consists of S-Store and Postgres.

ACM Reference Format:
Jiang Du, John Meehan, Nesime Tatbul, and Stan Zdonik. 2017. Towards Dynamic Data Placement for Polystore Ingestion. In *Proceedings of BIRTE '17, Munich, Germany, August 28, 2017,* 8 pages.
https://doi.org/10.1145/3129292.3129297

1 INTRODUCTION

In many modern applications such as the Internet of Things (IoT), time-sensitive data generated by a large number of diverse sources must be collected, stored, and analyzed in a reliable and scalable manner. This is critical to supporting accurate and timely monitoring, decision making, and control. Traditional data warehousing architectures that have been based on separate subsystems for managing operational (OLTP), data ingestion (ETL), and analytical (OLAP) workloads in a loosely synchronized manner are no longer sufficient to meet these needs. As a result, new approaches such as data stream warehousing [14], near real-time warehousing [27], lambda/kappa architectures [1, 2, 12], and HTAP systems [4] have recently emerged. While these approaches architecturally differ from one another, low-latency data ingestion (a.k.a., streaming or near real-time ETL) is seen as a critical component of the solution in all of them.

In our recent work, we have designed and built one-of-a-kind transactional stream processing system called *S-Store* [20]. S-Store is a scalable main-memory system that supports hybrid OLTP+streaming workloads with well-defined correctness guarantees including ACID, ordered execution, and exactly-once processing [26]. While S-Store can be used as a stand-alone system to support streaming applications with shared mutable state [6], we have also shown, within the context of the BigDAWG polystore system [11], how S-Store can uniquely enhance OLAP-style data warehousing systems with near real-time capabilities [21].

We believe that streaming ETL in particular stands out as the killer app for S-Store [19]. More specifically, S-Store can easily be programmed to continuously ingest configurable-size batches of newly added or updated data from a multitude of sources, and apply the necessary cleaning and transformation operations on them using its dataflow-based computational model. Furthermore, it provides the necessary scalable system infrastructure for processing ETL dataflows with transactional guarantees. A crucial component of this infrastructure is the database-style local in-memory storage. S-Store's storage facilities can be used for multiple different purposes, including: (i) temporary staging of newly ingested batches and any intermediate data derived from them as they are being prepared for loading into the back-end data warehouse, (ii) caching copies of older data fragments from the warehouse that will need to be frequently looked up during ETL, (iii) serving as the primary storage for data fragments which are subject to frequent updates. In general, since our streaming ETL engine has all the capabilities of an in-memory OLTP database, it can take over some of the responsibility of the back-end warehouse. For example, it can be directly queried to provide fast and consistent access to the freshest data.

Figure 1 shows a high-level overview of the streaming ETL architecture that we envision. All data newly collected from the sources (time-ordered, append-only streams as well as arbitrary insertions in general) and requests for in-place updates or deletions on older data are ingested through a transactional streaming engine (S-Store). The streaming engine in turn populates a back-end OLAP engine with updates on a frequent basis, through a data migration component. The migration component is bi-directional, i.e., data can be copied or moved between the two engines transactionally, in both directions. Meanwhile, all OLAP query requests to the system

BIRTE '17, August 28, 2017, Munich, Germany
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5425-7/17/08…$15.00
https://doi.org/10.1145/3129292.3129297

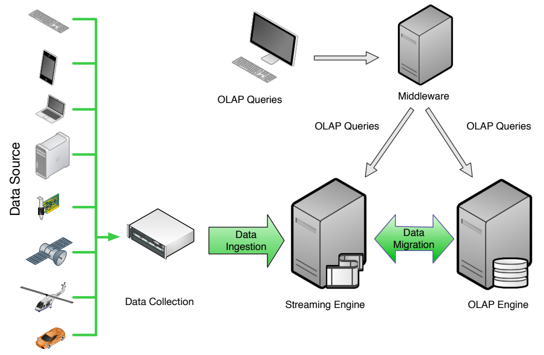

Figure 1: Architectural Overview

are received by a middleware layer that sits on top of the two engines. This layer maintains a global system catalog, which keeps track of all the data fragments and where they are currently stored. Based on this information, it determines where to forward the query requests for execution. Query results received from the engines are then returned back to the user. We have built a working prototype for this architecture based on Kafka [17] (data collection), S-Store (streaming ETL engine), Postgres (OLAP engine), and BigDAWG (migration + middleware layer) [19]. This architecture raises a number of interesting research issues in terms of cross-system optimization.

In this paper, we study how different data placement strategies perform in the presence of mixed (read and write) ETL workloads: Given a data fragment (i.e., the lowest level of data granularity - part of a relation), it can be stored in the streaming engine, in the OLAP engine, or in both. While the main-memory streaming engine can generally handle look-ups and updates faster, it has a limited memory budget. In contrast, the OLAP engine has larger storage capacity, but is slower to access. Furthermore, both engines are subject to dynamically changing workloads which consist of ingestion and query requests. Thus, given a mixed workload with different types of data, operations, and performance needs, data ingestion is affected greatly by the decisions of (i) which data fragments to store in the streaming engine, and (ii) whether to copy or move the data fragments between the database engines. As we will illustrate based on preliminary experiments on the TPC-DI benchmark [23], this decision can have significant impact on ETL latency.

2 BACKGROUND

2.1 BigDAWG Polystore

When it comes to database systems, it is commonly believed that "one-size no longer fits all" [25]. Specialized databases have become the norm. Some systems are designed specifically for unique types of data such as arrays or graphs [7, 9]. Others specialize in data formatting such that analytical queries can run extremely quickly [24]. Many workloads, however, require multiple of these specializations to execute efficiently.

Intel's BigDAWG represents a polystore of multiple disparate database systems, each of which specializes in one type of data (e.g., relational, array, streaming, etc.) [11]. BigDAWG provides

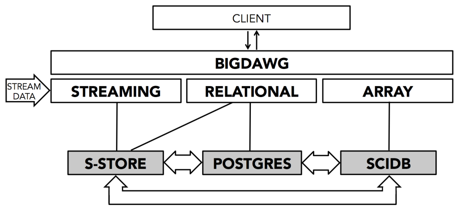

Figure 2: BigDAWG 1.0 Architecture

the user with querying and data migration across these systems, essentially abstracting the individual systems into one unified front-end from the user's perspective. BigDAWG accomplishes this by separating databases into several "islands of information", each of which contains multiple systems that share a common query language. For instance, relational databases such as Postgres and MySQL are connected to "relational island," which is queried via SQL statements (Figure 2).

While operations are delegated to the appropriate specialized system, BigDAWG also contains the ability to run queries on one engine which requires data from another engine. To facilitate this, BigDAWG contains the ability to efficiently migrate data from one engine to another. One specific scenario that data migration makes possible is the ingestion of streaming data into an analytical data warehouse. Such a scenario is best served by a data stream management system performing data cleaning operations on the streaming data before migrating the data to the OLAP engine, as discussed in Section 1.

2.2 S-Store

S-Store is a streaming system that specializes in the correct management of shared, mutable state [20]. S-Store models its dataflow graphs as a series of transactions, each of which has full ACID properties inherited from OLTP. S-Store also provides the ordering and exactly-once guarantees of a modern streaming system, ensuring correctness from the perspective of the dataflow graph.

S-Store is built on top of the main-memory OLTP system, H-Store [16]. Transactions are parameterized user-defined stored procedures, each of which passes output data along a stream in a dataflow graph to be used as the input parameters of the next downstream stored procedure. Each transaction executes on an atomic batch of tuples, the size of which is defined by the user. Batches are ordered by their time of arrival, and that order is maintained throughout the dataflow graph. Transactions in a dataflow graph typically execute independently, meaning locks on shared state are released between consecutive transactions in the graph.

3 THE DATA PLACEMENT PROBLEM

In our data warehousing setting, the workload consists of a mix of ingest requests and query requests. These requests must be served on a continuous basis with low latency and high throughput. Ingest requests may generally consist of any changes to the data in the warehouse including insertions, in-place updates, or deletions. Feeds from streaming data sources (e.g., sensors, stock market) are typically in the form of appends (i.e., only time-ordered insertions). Ingest requests are primarily served by S-Store, whereas query requests can be served by both S-Store or Postgres depending on the location of the needed data fragments.

S-Store transactionally processes ingest requests in small, atomic batches of tuples in order to apply ETL transformations. The resulting data fragments are then asynchronously migrated from S-Store to Postgres. This migration can be in the form of periodic pushes from S-Store to Postgres, or on-demand pulls by Postgres. Since S-Store has its own local storage, it can store a data fragment temporarily until migration, or even thereafter if that fragment will be needed by S-Store. While the natural direction of migration for newly ingested data is from S-Store to Postgres, we also support migrations in the opposite direction. It can be useful to bring older data fragments back to S-Store, such as when an ETL transformation needs to look up older data for validating new data or when a certain data fragment starts receiving a burst of in-place updates on it.

In general, while a data fragment is being migrated from one engine (source) to another (destination), there are two options with respect to data placement:

(1) ***Move.*** Delete the migrated data fragment from the source engine as part of the migration transaction (i.e., the destination engine becomes the one and only location for the fragment).

(2) ***Copy.*** Continue to keep a copy of the migrated data fragment at the source engine (i.e., the fragment gets replicated at the destination engine).

These options can have advantages or disadvantages under different circumstances. The Move option keeps a single copy of a data fragment in the system, which avoids redundant storage and, more importantly, removes the need to maintain transactional consistency across multiple copies in case of updates to the fragment. However, data access may take more time if the desired fragment is not available in local storage. On the other hand, the Copy option incurs an overhead for storing and maintaining multiple replicas of a data fragment in the system. This is largely due to the transactional overhead of two-phase commit. However, efficient, local data access to the fragment is guaranteed at all times.

While these are generic tradeoffs between Move and Copy for any given pair of engines, there are additional considerations specific to our setting. More specifically, our front-end processor, S-Store, and back-end warehouse, Postgres, differ in their system characteristics. Being a main-memory system, S-Store can provide fast data access, but has limited storage capacity. Furthermore, it is optimized for short-lived, read and update transactions (e.g., no sophisticated techniques for large disk scans). Postgres is disk-based, and can support large-scale, read-intensive OLAP workloads better than S-Store.

In order to achieve high performance for a given mix of ingest and query workload, data fragments must be placed carefully. Both workload characteristics (e.g., frequency of reads vs. updates) as well as the tradeoffs discussed above must be taken into account. Furthermore, as the workload dynamically changes, placement of data fragments should be adjusted accordingly. Next, we will illustrate the problem using an example scenario taken from the TPC-DI benchmark [23].

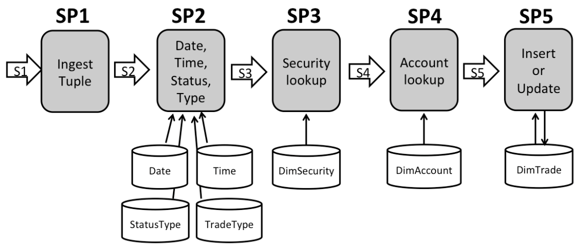

Figure 3: TPC-DI Trade Data Ingestion Dataflow

4 AN EXAMPLE: STREAMING TPC-DI

TPC-DI is a data integration benchmark for evaluating the performance of traditional ETL tools [23]. It models a retail brokerage firm that needs to extract and transform data from heterogeneous data sources. The original benchmark does not involve streaming data. However, some of the data sources are incremental in nature and can be modeled as such. For example, Figure 3 shows the trade data ingestion portion of the benchmark remodeled as a streaming ETL scenario. In this dataflow, new trade tuples go through a series of validation and transformation procedures before they can be loaded into the *DimTrade* table of the warehouse.

A Case for Copy. One of those procedures (*SP4*) involves establishing foreign key dependencies with the *DimAccount* table. More specifically, when new rows are defined within the *DimTrade* table, reference must be made to the *DimAccount* table to assign the *SK_AccountID* key along with a few other fields. In other words, for each new batch of trade tuples to be ingested, the ETL dataflow must perform a lookup operation in the *DimAccount* table. Assume that an initial load for the *DimAccount* table has already been performed via S-Store (the ETL engine) before the Trade dataflow starts executing. In other words, *DimAccount* already resides in Postgres (the OLAP engine). Unless a copy of the *DimAccount* fragments were kept in S-Store after this initial load, *SP4*'s lookups would require migrating the relevant *DimAccount* fragments from Postgres to S-Store. This in turn would incur high latency for the trade ingestion dataflow. In this scenario, keeping a local copy of *DimAccount* fragments to be referenced by the trade dataflow in S-Store would be a good data placement decision.

A Case for Move. Next, assume that S-Store occasionally ingests update requests for the *DimAccount* table. For *DimAccount* fragments that are replicated in S-Store, such updates must be transactionally applied on both engines in order to ensure mutual consistency. In this scenario, Move might be a more desirable data placement strategy for frequently updated *DimAccount* fragments than the Copy option. This way, S-Store would only have to locally update a single copy for those fragments.

OLAP Queries. Now further assume that, while the above ingestion scenarios on *DimTrade* and *DimAccount* are taking place, a query request on the *DimTrade* table arrives. The query requires the system to scan the whole *DimTrade* table and calculate the total and average difference between bid and trade price values for each trade. Trade data is streaming into the system at high frequency and is being ingested into Postgres through S-Store via periodic, push-based migration. As such, the larger portion of *DimTrade* (which

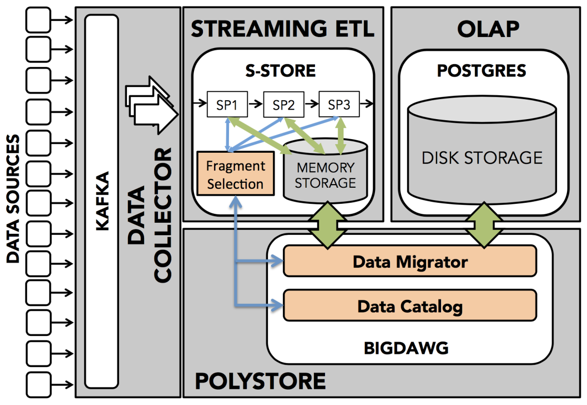

Figure 4: System Architecture

accounts for older trades) is stored in Postgres, while the smaller, recently ingested portion is stored in S-Store. Therefore, this query cannot be answered in its entirety on either Postgres or S-Store. The query planner has multiple different options to ensure the most complete (and thus, the freshest) answer to this query. For example, Postgres can issue a pull request to migrate *DimTrade* fragments from S-Store and then execute the OLAP query in Postgres, or the middleware layer can execute the OLAP query on S-Store and Postgres in parallel and merge their results into the full answer. We have analyzed some of these options in recent benchmark studies [19, 21]. The main takeaway is that data placement can also have significant impact on query performance. Therefore, the ETL workload must be considered in conjunction with the query workload in making data placement decisions.

5 SYSTEM ARCHITECTURE

We have created a prototype for streaming data ingestion [19]. This prototype uses a combination of BigDAWG and S-Store, in conjunction with Kafka (a publish-subscribe messaging system [17]) and the relational database Postgres (Figure 4). New tuples arrive from a variety of data sources and are queued in Kafka. These tuples are batched and pushed to S-Store. As a streaming system with ACID state management, S-Store is particularly well-suited for streaming data ingestion workloads. Streaming data can be ingested and transformed in a dataflow graph, with intermediate state being maintained in a transactional manner.

For each stored procedure that requires access to data, S-Store checks the data catalog in BigDAWG through a **fragment selection** module. Data catalog in BigDAWG maintains all information about data fragments including the data placement. If the required data fragment only exists in Postgres, the fragment selection module will instruct the data migrator in BigDAWG to migrate the fragment from Postgres to S-Store. Meanwhile, the fragment selection module is also responsible for deciding whether the migration should be **Move** or **Copy**, and if the total size of the fragments exceeds the storage limit of S-Store, which fragment(s) should be evicted.

Once the final tuples have been created, they can then either be stored directly in S-Store, or migrated to a data warehouse.

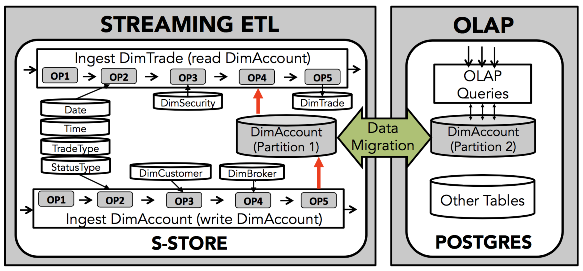

Figure 5: Experimental Setup (*DimAccount* in TPC-DI)

6 PRELIMINARY EXPERIMENTS

6.1 Setup

To evaluate the effect of Copying and Moving data in the presence of multiple database systems, data must be distributed between at least two engines. Our experiments simulate a streaming data ingestion workload, and thus we use our streaming data ingestion prototype described in Section 5. In the implementation, data can be migrated between S-Store (the streaming ETL engine) and Postgres (the OLAP engine), and queries can be run on either system.

We executed the experiments on an Intel® Xeon® machine with 64 virtual cores and 132GB memory. S-Store is co-located on the same node as Postgres for ease of communication and migration. We warmed up the S-Store cache for ten seconds before collecting statistics.

To motivate the use of multiple systems in tandem, we implemented a subset of TPC-DI as a streaming workload (Section 4). Specifically, our experiments involve the ingestion of the *DimTrade* and *DimAccount* tables from their respective flat files. Each of these ingestion processes was modeled as its own dataflow graph consisting of multiple SQL statements. These SQL statements perform lookups on a variety of other tables to retrieve normalized foreign keys before inserting the finished tuple into a final table (Figure 5).

In the case of *DimAccount*, incoming tuples represent in-place updates to existing rows in the database. *DimTrade* tuples, on the other hand, are always inserts, but require a lookup on the *DimAccount* table to generate a foreign key. S-Store is configured as single-sited and single-partitioned. Since there are no distributed transactions for this configuration, we chose to implement the ingestion of *DimTrade* and the update of *DimAccount* each in one stored procedure.

We generated heterogeneous workloads that contain changes to both *DimAccount* and *DimTrade*. In each experiment, the workload varies in terms of the percentage of operations that write or read from *DimAccount*. We partition *DimAccount* into ten fragments of equi-width. We notice that the ingestion of *DimTrade* from the *Trade.txt* flat file only accesses five of the ten fragments. For the update to *DimAccount*, we randomly generate a sequence of in-place updates by selecting the *account-id* that falls into the five fragments that are read during the ingestion of *DimTrade*. We then mix the in-place updates with the data ingestion source from *Trade.txt*. We measure the average latency of each operation (in-place update or ingestion) for the heterogeneous workload.

For simplicity, most tables in this experiment are considered to be cached in S-Store. The *DimTrade* table is considered to be entirely located in S-Store. The *DimAccount* table, on the other

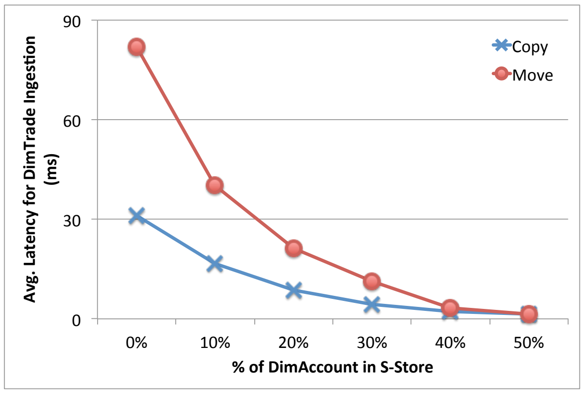

Figure 6: A Read-Only Workload (100% Read + 0% Write)

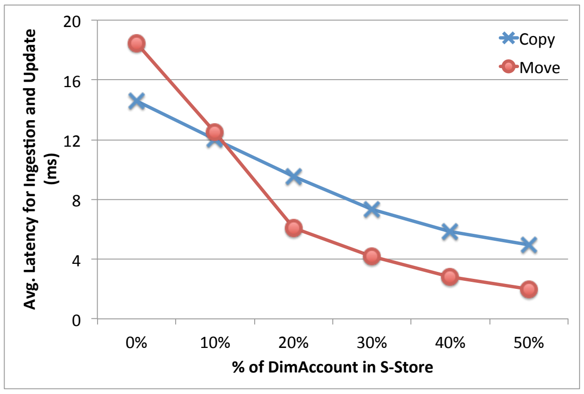

Figure 7: A Write-Intensive Workload (1% Read + 99% Write)

hand, is primarily located in Postgres. In the following experiments, a percentage of *DimAccount* is either Copied or Moved to S-Store, depending on the scenario. For all of the experiments, we measure the average latency of each operation (ingestion or update) in the workload that includes necessary data migrations.

6.2 Results

6.2.1 Read-intensive Workloads. As shown in Figure 5, the ingestion of *DimTrade* contains five operations, with OP4 retrieving the *account-id*, *broker-id* and *customer-id* from table *DimAccount*. Executing this lookup process locally in S-Store (i.e., Copying *DimAccount* from Postgres to S-Store) can generally improve the performance, but S-Store only has limited storage space (or *cache* as we call it in this paper). In this experiment, we study how the storage limit of S-Store affects the performance for lookups (to table *DimAccount*) during data ingestion (of table *DimTrade*). The workload we generate for this experiment contains only data ingestion to *DimTrade* and no update to *DimAccount*, i.e., this workload contains 100% reads and 0% writes to *DimAccount*.

Figure 6 demonstrates the benefit of Copying tables to S-Store when there are lookups in the ETL tasks. As we clarified in 6.1, the y-axis represents the average latency of the ingestion, including necessary data migrations. When the cache size in S-Store is 0, for each lookup to *DimAccount* during the ingestion to *DimTrade*, the fragment of *DimAccount* that contains the key (*account-id*) must be migrated from Postgres to S-Store. Typically the migration incurs prohibitive cost. When the cache size increases, the fragments that have been migrated to S-Store can be stored locally for future lookups during the ingestion, reducing the number of migrations. We employed least recently used (LRU) to evict fragments from S-Store when the size of Copied fragments exceeds the cache limit. When S-Store has a large enough storage and is able to cache all the fragments of *DimAccount* table that are required in the ingestion of *DimTrade* (in this experiment, five out of the ten fragments are accessed during the ingestion of *DimTrade*), the latency of the ETL ingestion to *DimTrade* is minimized.

In this scenario, the average latency of the workload for Moving is more expensive than Copying for most cache sizes. The reason is that when the cache in S-Store is full, and a fragment required is not in the cache, a new migration must be issued. Copy only has to migrate the required fragment from Postgres to S-Store, while Move has an additional step to migrate the evicted fragment from S-Store back to Postgres.

6.2.2 Write-intensive Workloads. When migrating data from Postgres to S-Store, Move implies that there is always only one copy of the data in the database engines, while Copy implies that there are two copies of the data in the system: one in S-Store, and another in Postgres. The workload in this experiment contains 99% in-place updates to *DimAccount* and 1% ingestions to *DimTrade*, generated as described in Section 6.1. In order to guarantee the transactional safety, the updates are executed synchronously in S-Store and Postgres. When an update is issued to S-Store, if the fragment of the data that this update accesses exists only in S-Store, the update is executed and finished in S-Store. If the fragment of the data exists in Postgres, S-Store will issue the update to Postgres for execution and stalls until Postgres finishes the execution.

S-Store is built on top of H-Store, a system that is designed to speed up transactionally safe updates, and hence for such operations, S-Store has a much lower latency compared to Postgres. Figure 7 shows that when the cache size increases, more fragments are migrated to S-Store, and since Move only keeps one copy of a fragment in the system, Moved fragments exist only in S-Store. Thus, there are no additional steps for updating the data in Postgres, which would increase the cost. Therefore, for a write-intensive workload where updates are the majority, the average latency decreases quickly when the cache size increases if we choose to Move the data between S-Store and Postgres.

On the contrary, Copy keeps the data in both S-Store and Postgres. As we have explained, the cost of updates in Postgres dominates the synchronized update process, and thus the curve of the average latency for Copy does not change much when the cache size increases. We also notice that when the cache size is less than 10% of the size of the *DimAccount* table, Copy performs better than Move. It is not difficult to see that in such cases, a large amount of migrations are conducted because of cache misses (i.e., the required fragment is not in the cache). For each cache miss, Copy only has to migrate the required fragment from Postgres to S-Store (and delete the evicted fragment from S-Store), while Move must execute two

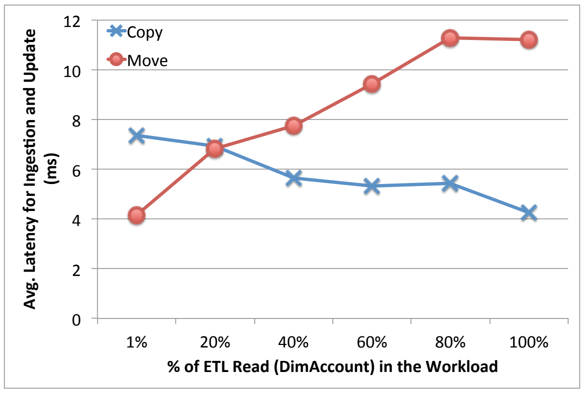

Figure 8: Heterogeneous Workloads

migrations, one for the required fragment from Postgres to S-Store, another one for the evicted fragment from S-Store to Postgres.

6.2.3 Heterogeneous Workloads. We have seen that for read-intensive workloads, Copy often has better average latency, and for write-intensive workloads, Move usually has better average latency. Here, we experiment with workloads that are heterogeneous. In this experiment, we generate a series of workloads for which read operations (ingestion to *DimTrade*) make up from 1% to 100% of the total workload. We fix the cache size to 30% of *DimAccount*.

First, Figure 8 confirms our previous observation that when the percentage of reads is relatively small ($< 20\%$) where the workload is dominated by writes (transactional updates to *DimAccount*), the latency for Move is lower than that for Copy, and when the workload is dominated by reads (ETL lookups to *DimAccount*), the latency for Copy is better than that for Move. Secondly, in our experiments, the cost of reads is much cheaper than that of writes (in-place updates); thus, for the Copy scenario, the average latency of all reads and writes to *DimAccount* in the workload decreases as the percentage of read in the workload increases. Thirdly, we notice that the average latency for Move increases when the percentage of read in the workload increases. This is because for cache misses when the cache is full, Move is much more expensive than Copy, as we have explained above. The additional migration cost for Move offsets the benefit brought by cheaper read. The figure shows that the curves for Copy and Move meet in a workload that contains about 20% of read operations in this setting as a confluence of the factors cited above.

6.2.4 Takeaway Messages. We notice that the migration cost between database engines (S-Store and Postgres) is very expensive compared to the cost of local reads and writes in a workload. The migration cost is frequently not negligible during the execution of a mixed workload. For instance, although local reads are much cheaper than writes in S-Store in our settings, an ETL read may incur a data migration from Postgres to S-Store, and it may increase the average latency for data ingestion by up to two orders of magnitude. This implies that for certain circumstances, migration cost could be the dominating cost for a workload, and minimizing this migration cost may be a good enough objective function for an approximated optimized design. For other circumstances, considering only migration cost is probably not enough. For instance, for a workload that contains only writes, it may make sense to migrate the data from Postgres to S-Store, so the transactional writes are executed faster in S-Store, even if it means paying the additional cost for migrating data between the database engines.

7 RELATED WORK

Data placement was previously studied in traditional multi-database systems, including federated and distributed databases. In federated databases like Garlic [15], the focus is on unified, cost-based querying over heterogeneous databases, each of which is autonomous in their internal data placement and query processing policies. This is in contrast to our polystore setting, where on-demand data migration across engines in the form of Move and Copy operations can be utilized to optimize data placement. In distributed databases, distribution design involves both fragmentation (i.e., how to divide tables into partitions) and allocation (i.e., how to place fragments onto nodes) [22]. For best results, the two should be tackled together, which makes the problem more complex. In our case, due to volatile and mixed nature of streaming ETL+OLAP workloads, we focus on a dynamic, workload-aware data placement solution.

There is extensive literature on physical database design tuning, including online approaches that are sensitive to workload changes ([5, 10]). The focus of these works has mainly been on tuning indexes, materialized views, or partitions of a single database system. More recently, new techniques have been proposed for physical design tuning in multi-store systems. For example, MISO determines where to store data in an HDFS-RDBMS hybrid storage system based on materialized views [18]. Our work extends these efforts further by considering data placement requirements of near real-time ingestion in a polystore environment.

Caching has been used as an optimization technique in many settings including databases. In IBM's DBCache, data from back-end database servers is replicated on front-end database servers in order to enhance data access performance of dynamic web applications [3]. Memcached is a distributed memory caching system to speed up dynamic database-driven web services [13]. Anti-caching is a technique to move colder data to disk in main-memory OLTP databases [8]. Our Copy-based migration is similar to caching in spirit, whereas our Move-based migration resembles anti-caching in that the system maintains a single copy of data with hot fragments residing in main-memory S-Store and colder ones residing in disk-based Postgres.

As we briefly mentioned earlier, our near real-time data ingestion support for data warehouses has also been followed by others in different contexts [2, 4, 12, 14, 27]. Stream warehouses such as ATT's DataDepot focus on ingesting append-only streams into a historical warehouse focusing on leveraging temporal semantics for consistency and data partitioning [14]. Near real-time warehouses focus on micro-batch ETL by invoking ETL pipelines at higher frequencies to maintain a more up-to-date data warehouse [27]. More recently, big data companies have proposed new architectures that integrate near real-time and batch processing in a way to ensure low latency for the former and high throughput for the latter [2, 12]. Finally, HTAP systems such as IBM's Wildfire tightly integrate

OLTP and OLAP support in order to support near real-time analytics including ingestion [4]. Our polystore environment contains elements from each of these different approaches, but emphasizes the use of a transactional streaming database infrastructure for near real-time ingestion and a polystore backend for OLAP, in which dynamic data placement plays a critical role.

8 SUMMARY AND ONGOING WORK

In this paper, we have discussed the problem of data placement and caching in a distributed polystore. Our belief is that data ingestion in this setting presents some unique challenges that require further study. Our solution involves an integration of a stream processing system with an analytics back-end provided by the BigDAWG polystore. This paper is a first step in that direction.

While caching and data placement are not new ideas, the context of a polystore changes their performance characteristics in such a way as to require a complete rethinking. In this paper, we have considered Copying results in the ingestion engine to make subsequent reads faster. Copying requires making or retaining a copy in the streaming engine or in the home storage system of the data. Any update to that data would have to be realized in all locations, making writes very expensive. To address this, we also allow Moving the data, which simply moves the data to a new location (including perhaps the ingestion engine). This paper has studied the problem of how to best match the workload to the appropriate Moves and Copies.

While the work described in this paper involves two systems that each run on a single machine, the data placement problem is further complicated once the individual systems within the polystore are distributed across multiple machines. In this scenario, it is not enough to consider the system-level location of the data, but also the location of the data on physical hardware. It is likely that distribution properties of individual systems must be considered when determining data locations at the polystore level. Additionally, while this paper focuses on only two systems for simplicity, the data placement problem becomes much more difficult in a configuration space of three or more systems. As the number of systems in a polystore increases, so do the possible trade-offs considered when deciding when to Move or Copy data. These are interesting research problems, and we leave them as future work.

In the future, we envision a system that dynamically Moves and Copies the data in response to a particular workload. In order to accomplish this, we will need several things.

- A cost model that can estimate a relative cost for various placement plans. The cost model will be used to compare the effectiveness of multiple plans and must account for the extreme expense that is incurred when migrating data from one system to another.
- A more robust distributed catalog that, among other things, keeps track of where a particular piece of data currently resides.
- A tighter integration with the BigDAWG query optimizer that uses the cost model to choose the best query plan, which here amounts to picking the best copy and using it at the best point in a distributed query plan.

We are currently working on these research issues and intend to perform a thorough experimental evaluation using several benchmarks, including TPC-DI and an application furnished to us by a major credit card processing company.

Acknowledgments. We thank Renee J. Miller for reviewing the work. We also thank the anonymous reviewers for their helpful suggestions. This research is funded in part by a Bell Canada Fellowship, NSERC, the Intel Science and Technology Center for Big Data, and the NSF under grant NSF IIS-1111423.

REFERENCES

[1] Kappa Architecture. https://www.oreilly.com/ideas/questioning-the-lambda-architecture.
[2] Lambda Architecture. http://lambda-architecture.net.
[3] M. Altinel, C. Bornhovd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. Cache Tables: Paving the Way for an Adaptive Database Cache. In *VLDB*, pages 718–729, 2003.
[4] R. Barber, M. Huras, G. M. Lohman, C. Mohan, R. Muller, F. Ozcan, H. Pirahesh, V. Raman, R. Sidle, O. Sidorkin, A. J. Storm, Y. Tian, and P. Tozun. Wildfire: Concurrent Blazing Data Ingest and Analytics. In *SIGMOD*, pages 2077–2080, 2016.
[5] N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *ICDE*, pages 826–835, 2007.
[6] 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. *PVLDB*, 7(13):1633–1636, 2014.
[7] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. A Demonstration of SciDB: A Science-oriented DBMS. *PVLDB*, 2(2):1534–1537, 2009.
[8] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. Anti-caching: A New Approach to Database Management System Architecture. *PVLDB*, 6(14):1942–1953, 2013.
[9] Neo4J Developers. Neo4j. https://neo4j.com/, 2012.
[10] J. Du, B. Glavic, W. Tan, and R. J. Miller. DeepSea: Progressive Workload-Aware Partitioning of Materialized Views in Scalable Data Analytics. In *EDBT*, pages 198–209, 2017.
[11] 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. *PVLDB*, 8(12):1908–1911, 2015.
[12] R. C. Fernandez et al. Liquid: Unifying Nearline and Offline Big Data Integration. In *CIDR*, 2015.
[13] B. Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 124:5–5, 2004.
[14] L. Golab et al. Stream Warehousing with DataDepot. In *SIGMOD*, pages 847–854, 2009.
[15] V. Josifovski, P. Schwarz, L. Haas, and E. Lin. Garlic: A New Flavor of Federated Query Processing for DB2. In *SIGMOD*, pages 524–532, 2002.
[16] R. Kallman et al. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *PVLDB*, 1(2):1496–1499, 2008.
[17] J. Kreps, N. Narkhede, and J. Rao. Kafka: A Distributed Messaging System for Log Processing. In *NetDB Workshop*, 2011.
[18] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey. MISO: Souping Up Big Data Query Processing with a Multistore System. In *SIGMOD*, pages 1591–1602, 2014.
[19] J. Meehan, C. Aslantas, S. Zdonik, N. Tatbul, and J. Du. Data Ingestion for the Connected World. In *CIDR*, 2017.
[20] 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*, 8(13):2134–2145, 2015.
[21] J. Meehan, S. Zdonik, S. Tian, Y. Tian, N. Tatbul, A. Dziedzic, and A. Elmore. Integrating Real-Time and Batch Processing in a Polystore. In *IEEE HPEC*, 2016.
[22] M. T. Özsu and P. Valduriez. Distributed Database Systems: Where Are We Now? *IEEE Computer*, 24(8):68–78, 1991.
[23] M. Poess et al. TPC-DI: The First Industry Benchmark for Data Integration. *PVLDB*, 7(13):1367–1378, 2014.
[24] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A Column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[25] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE*, pages 2–11, 2005.

[26] N. Tatbul, S. Zdonik, J. Meehan, C. Aslantas, M. Stonebraker, K. Tufte, C. Giossi, and 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), 2015.

[27] P. Vassiliadis and A. Simitsis. Near Real-Time ETL. In S. Kozielski and R. Wrembel, editors, *New Trends in Data Warehousing and Data Analysis*, pages 1–31. Springer, 2009.