Data Ingestion for the Connected World

John Meehan
Cansu Aslantas
Stan Zdonik
Brown University
{john,cpa,sbz}@cs.brown.edu

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

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

ABSTRACT

In this paper, we argue that in many "Big Data" applications, getting data into the system correctly and at scale via traditional ETL (Extract, Transform, and Load) processes is a fundamental roadblock to being able to perform timely analytics or make real-time decisions. The best way to address this problem is to build a new architecture for ETL which takes advantage of the push-based nature of a stream processing system. We discuss the requirements for a streaming ETL engine and describe a generic architecture which satisfies those requirements. We also describe our implementation of streaming ETL using a scalable messaging system (Apache Kafka), a transactional stream processing system (S-Store), and a distributed polystore (Intel's BigDAWG), as well as propose a new time-series database optimized to handle ingestion internally.

1. INTRODUCTION

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 targeted by many previous research initiatives, such as data integration, deduplication, integrity constraint maintenance, and bulk data loading. Data ingestion is frequently discussed under the name of Extract, Transform, and Load (ETL).

Modern applications put new requirements on ETL. Traditionally, 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 particularly 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.

We argue that 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 time-sensitive 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.

While this paper is primarily about data ingestion at scale, we note that for many modern applications, the data in question are time-series. We point out that paying special attention to this important data type can have strong benefits, and we are currently pursuing this line of work.

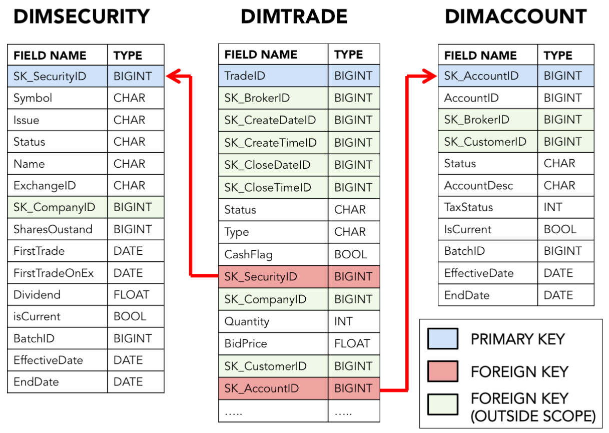
2. MOTIVATING EXAMPLES

2.1 Internet of Things

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 in these new applications, but the scale at which these tasks must operate changes the way we must conceive of the solution.

For instance, take 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 millions of 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. Storing this data can become extremely expensive,



DIMSECURITY

FIELD NAME	TYPE
SK_SecurityID	BIGINT
Symbol	CHAR
Issue	CHAR
Status	CHAR
Name	CHAR
ExchangeID	CHAR
SK_CompanyID	BIGINT
SharesOustand	BIGINT
FirstTrade	DATE
FirstTradeOnEx	DATE
Dividend	FLOAT
isCurrent	BOOL
BatchID	BIGINT
EffectiveDate	DATE
EndDate	DATE

DIMTRADE

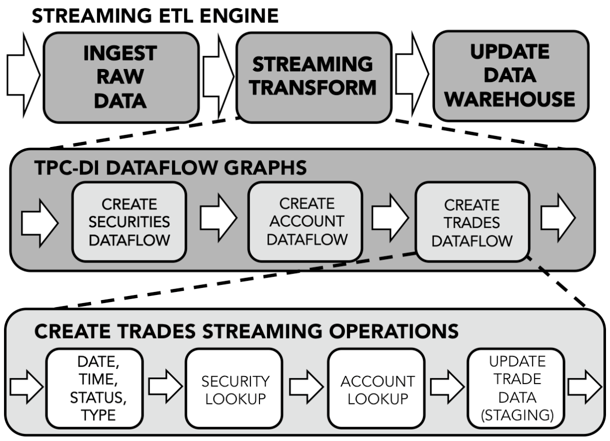
FIELD NAME	TYPE
TradeID	BIGINT
SK_BrokerID	BIGINT
SK_CreateDateID	BIGINT
SK_CreateTimeID	BIGINT
SK_CloseDateID	BIGINT
SK_CloseTimeID	BIGINT
Status	CHAR
Type	CHAR
CashFlag	BOOL
SK_SecurityID	BIGINT
SK_CompanyID	BIGINT
Quantity	INT
BidPrice	FLOAT
SK_CustomerID	BIGINT
SK_AccountID	BIGINT
.....	.....

DIMACCOUNT

FIELD NAME	TYPE
SK_AccountID	BIGINT
AccountID	BIGINT
SK_BrokerID	BIGINT
SK_CustomerID	BIGINT
Status	CHAR
AccountDesc	CHAR
TaxStatus	INT
IsCurrent	BOOL
BatchID	BIGINT
EffectiveDate	DATE
EndDate	DATE

🟦	PRIMARY KEY
🟥	FOREIGN KEY
🟩	FOREIGN KEY (OUTSIDE SCOPE)

(a) Partial Schema



(b) Partial Dataflow Graph

Figure 1: TPC-DI

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 time-series data can all be performed by one system in near real-time.

2.2 Streaming TPC-DI

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. Take for instance 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 [18]. It focuses on the extraction and transformation 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. The obvious benefit of the conversion to streaming is quicker access to incremental results. Traditional ETL systems process large batches of data overnight, while a streaming version could process smaller micro-batches throughout the day.

Assuming the ETL system outputs the same final results, there is no downside to taking a streaming approach. However, the act of breaking large batches into smaller ones introduces new data dependencies. Take for instance the DimSecurity, DimAccount, and DimTrade tables, each of which are described in Figure 1(a). 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) [21]. 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. The operations for managing additions to the DimSecurity, DimAccount, and DimTrade tables can be arranged in a specific order by using a streaming dataflow graph. For example, assume a batching mechanism groups the creation of security S, account A, and a trade request T from account A for purchasing shares of security S. The dataflow graph illustrated in Figure 1(b) maintains the data dependencies of this batch while allowing processing flexibility. The batch is first processed by a *CreateSecurities* dataflow graph, then *CreateAccount*, and finally, *CreateTrades*.

Dividing ETL processing into small, ordered transactions has the added benefit of making the process more granular for the purposes of distributed processing. Ordinarily, the operations creating the trade, company, and security relations may take place in a single, large transaction. 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. Streaming ETL could allow these large distributed transactions to be broken into several small single-sited transactions without damaging correctness due to its push-based, ordered processing. Thus, the transactions creating the trade, company, and security relations can each run separately on different nodes, independent of one another save for their order of execution.

3. STREAMING ETL REQUIREMENTS

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 divide the requirements for a streaming ETL system into three categories: ETL requirements, streaming requirements, and infrastructure requirements.

3.1 ETL Requirements

3.1.1 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. While heterogeneity in data collection 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 required for a large number of streaming data sources, as in IoT. 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 [19]. Another option when router programming is not readily available is to use field-programmable gate arrays (FPGAs) for the same functionality. 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.

3.1.2 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 cost of inserting data using SQL is significantly higher than inserting data in bulk. Furthermore, indexing the new data presents a significant issue for the warehouse. Therefore, a streaming ETL engine must have the ability to bulk load freshly transformed data into the data warehouse.

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.

3.1.3 Heterogeneous Data Types

Modern data ingestion architectures should provide built-in support for dealing with diverse target storage systems. 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 [7]). The presence of multiple heterogeneous destinations 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.

3.2 Streaming Requirements

3.2.1 Out-of-Order and Missing Tuples

This topic relates to data cleaning, but has particular importance to IoT. When receiving data from 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. With so many sources and incoming tuples, 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 [5]. 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 the missing values in 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.

3.2.2 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 DAG known as a 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.

3.2.3 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 endangers the integrity of the data. Data ingestion attempts to remove duplicate records, but without exactly-once processing, our system may 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. Any data migration to and from the streaming ETL engine must also occur once and only once. In both inter- and intrasystem messaging, the requirement of exactly-once processing in a streaming ETL engine prevents the loss or duplication of tuples or batches in case of recovery.

3.3 Infrastructure Requirements

3.3.1 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.

The streaming ETL engine should also have the ability to take over some 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.

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 (provided that the data in the cache is read-mostly to keep data consistency inexpensive).

3.3.2 ACID Transactions

ETL processes are fundamentally concerned with the creation of state, and transactions are crucial for maintaining correctness of that 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 discussed in Section 3.3.1, 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.

Similarly, it is expected that the atomicity of operations is maintained in an ETL system. ETL is executed in batches, and it would be incorrect to install a fraction of the batch into a data warehouse.

Perhaps most importantly, all state must be fully recoverable in the event of a failure. This is relevant to individual ETL operations; in the event of a failure, the system should be able to recover to its most recent consistent status. Recovery also pertains to migration between the various components in an ETL stack. If the data ingestion and warehousing are handled by different components, then the migration between the two must also be fully recoverable.

ACID transactions provide all of these guarantees, and are a crucial element to any ETL system.

3.3.3 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.

3.3.4 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.

4. THE NEW ARCHITECTURE

In developing the architecture of a streaming ETL engine, we wished to create a generic design that can suit a variety of data ingestion situations. This generic architecture is illustrated in Figure 2. We envision four primary components: data collection, streaming ETL, OLAP backend, and a data migrator that provides a reliable connection between the ETL and OLAP components.

4.1 Data Collection

Data may be collected from one or many sources. In an IoT workload, for example, data is ingested from thousands of different data sources at once. Each 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 (Section 3.1.1). 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 trickier in the presence of thousands of streams (Section

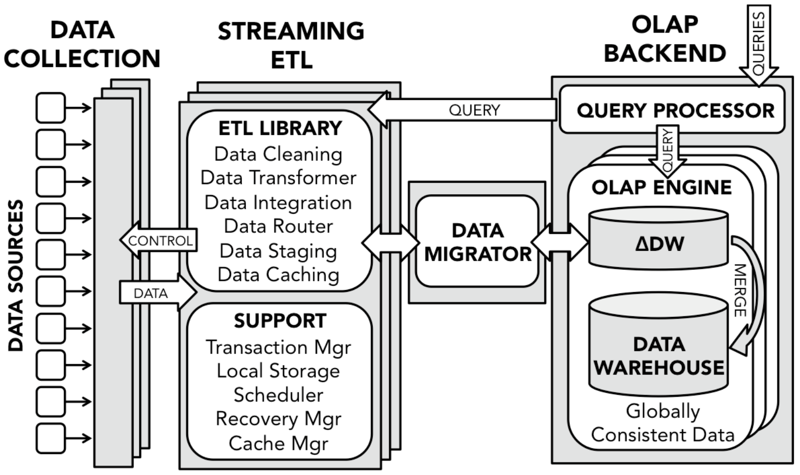


Figure 2: Streaming ETL Architecture

3.2.1). It is the data collector's responsibility to ensure that the global ordering of the tuples is maintained.

4.2 Streaming ETL

Once data has been batched in the data collector, 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. These ETL operators are largely dependent on the use-case that the system is designed to handle. For instance, traditional ETL may require full SQL support in order to select and compare against historic table data, while a time-series workload is more likely to require tools for signal processing, such as fast Fourier transformations.

Through 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 it is ready to be migrated. The migration may occur when the data warehouse is ready to receive the data and pulls it, or when the streaming engine is ready to push to the warehouse. 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 (Sections 3.3.3 and 3.2.3).

In addition to the ETL library, the streaming ETL engine also contains various support features found in most databases. For instance, to ensure consistency of the outgoing data, a transaction manager is included (Section 3.3.2). Local storage is needed for both staging and caching, and dataflow scheduling and recovery are managed within the engine (Sections 3.3.1 and 3.2.2).

4.3 OLAP Backend

The OLAP backend consists of a query processor and one or several OLAP engines. The need to support 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 (Section 3.1.3). 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 (Section 3.1.2). 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 all 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. Replication between the streaming ETL and OLAP engine is also an option, but it comes at the cost of needing to maintain two consistent versions of the same data items.

4.4 Durable Migration

Batches are frequently moved between the Streaming ETL and OLAP Backend components, and a mechanism is needed to ensure that there is no data lost in transit. Failures that occur 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 fully support ACID transactions as well (Section 3.3.2).

Migration is expensive, and it is important that it primarily takes place at a time when the individual components are not overloaded. For instance, the OLAP Backend may frequently handle long-running queries that are both disk and CPU intensive. Similarly, there may be scenarios in which the Streaming ETL engine is overworked. In order to avoid reduced performance of either the data ingestion or warehouse analytics, flexibility in both the timing and quantity of data in bulk loading is necessary. Assuming that the ingestion and analytics take place in different locations, there are two options for choosing when to migrate between the two:

Push. Data may be periodically pushed from the ingestion storage to the data warehouse. This may either be done at a regular frequency, or can occur when the workload on the data ingestion is lighter than usual.

Pull. The data warehouse itself can periodically pull fresh data from the data ingestion storage. This would occur either directly before an analytical query (if optimal data freshness is the priority) or at periods in which fewer analytical queries are running (if query performance is the priority).

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 be pushed to the OLAP backend when it is ready (Section 3.3.4). If long-running OLAP queries are the priority, then the backend should pull new data when there is downtime.

5. RELATIONAL ETL IMPLEMENTATION

When implementing our streaming ETL architecture, we wanted to consider the two primary use-cases listed in Section 2: relational ETL (TPC-DI) and time-series ETL (IoT). While the architecture in Figure 2 can be used for either, the implementation may vary significantly.

Relational data ingestion requires the transformation of incoming time-series data to suit the needs of one or several OLAP backends that are highly specialized. Typically these use-cases involve the mutation of relational data in a way that maintains constraints on the data, such as foreign keys. In this section, we discuss the implementation of our streaming ETL architecture for use in a relational ETL context. We use three research technologies to implement our core streaming ETL components - Apache Kafka [13], S-Store [15], and Intel's BigDAWG polystore [7]. The implementation is illustrated in Figure 3.

5.1 Components

5.1.1 Data Collection - Kafka

A streaming system needs a messaging infrastructure at its entrance to get data into the system. Kafka is a good initial choice. 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 [13] . 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

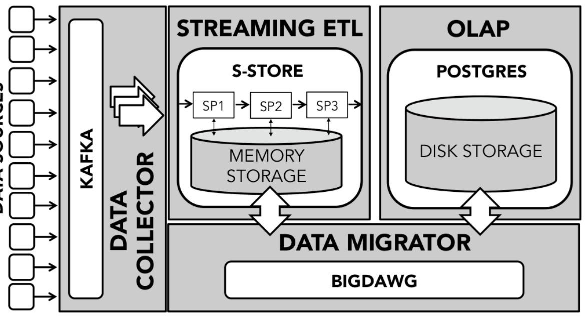


Figure 3: Relational ETL Implementation

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.

5.1.2 Streaming ETL - S-Store

Choosing a streaming data management system that best fits streaming ETL is complicated, as there are advantages and disadvantages to each. Few single systems meet all requirements described in Section 3. For instance, Spark Streaming [26] and Twitter Heron [14], two modern streaming systems, offer highly scalable processing with low latency. They provide data-driven processing and streaming primitives, and either seems like a natural choice for the streaming ETL component. However, when it comes to state management, neither system has a strong story. Neither provide support for shared, mutable state, so both would need some sort of external storage component. The addition of another database system for state management would introduce an extreme amount of overhead, since most operations in traditional ETL involve reading or writing persistent data on a regular basis.

S-Store, on the other hand, is a new streaming database system built explicitly to handle shared, mutable state [15]. Unlike traditional streaming systems, S-Store models dataflow graphs as a series of transactions, each of which ensure a consistent view of the modified state upon commit. S-Store is built on top of the main-memory OLTP system H-Store [11], and integrates streaming functionality such as streams, windows, triggers, and dataflows. It provides three fundamental guarantees which together are exclusively available in S-Store: ACID transactions, dataflow ordering, and exactly-once processing [20]. These guarantees in conjunction with its streaming functionality make S-Store an ideal fit for a streaming ETL engine.

As an OLTP system at its core, S-Store uses relational tables as its primary method of storage. It provides many of the standard elements to be expected in a DBMS, including indexes and materialized views. S-Store uses user-defined stored procedures as its transactional operations. Each stored procedure is defined using a mixture of Java and SQL. This allows for a good deal of flexibility, as data cleaning operations can be quite complex.

While S-Store is well-suited for the task of managing streaming ETL, it does require careful database design in order to

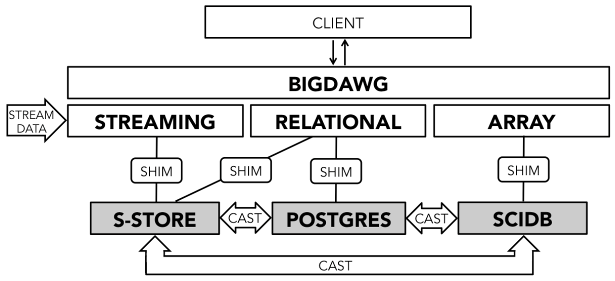


Figure 4: BigDAWG 1.0 Architecture

achieve scalability. As with most shared-nothing architectures, S-Store performs best when data items that are accessed together are co-located as often as possible, as distributed transactions are extremely expensive. Incoming data is batched in order to improve performance; typically the larger the batch, the better the performance in terms of raw throughput. However, large batches come with a trade-off of increased latency, as the system must wait for more data to arrive before processing a full batch. S-Store database design applies to the transactions within a dataflow graph as well. Typically an S-Store dataflow graph is composed of many transactions with only a few operations apiece rather than few transactions with many operations. By breaking up large operations into smaller components, it becomes possible to run several single-sited transactions in place of a large distributed transaction. This is particularly relevant when data is needed from multiple tables, each of which is hashed on a different key.

5.1.3 *OLAP Backend and Migration - Postgres and BigDAWG*

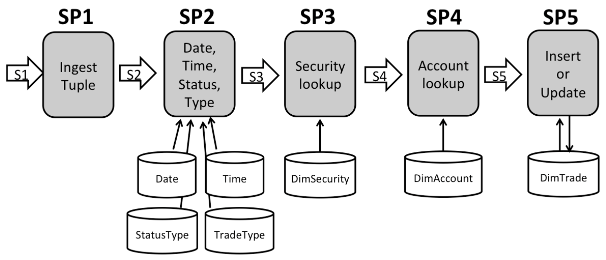
Because S-Store provides local storage, OLAP operations could, in theory, be integrated into the streaming ETL component. However, most OLAP engines are specialized and are able to perform much more sophisticated analytics operations with much better performance. Because this paper is primarily focused on ingestion rather than analytics, Postgres was chosen as a backend database for simplicity's sake. While using Postgres as an OLAP database is straightforward, the complication is the migration of data between the streaming ETL and OLAP engines. To address this concern, we chose the built-in data migrator provided by the BigDAWG polystore.

Intel's BigDAWG is a polystore of multiple disparate database systems, each of which specializes in one type of data (e.g., relational, array, streaming, etc.) [7]. BigDAWG provides unified querying and data migration across each of its databases, presenting them to the user as a single system. To facilitate this, BigDAWG includes several "islands of information", each of which contains multiple systems that share a common query language. For instance, all relational databases are connected to "relational island", which is queried using standard SQL. The architecture of BigDAWG is shown in Figure 4.

BigDAWG provides the flexibility needed to handle multiple OLAP data warehouses under one roof. Because S-Store is also integrated with BigDAWG, migration between S-Store and any OLAP system supported by the polystore is easy to implement and efficient [16]. Most importantly, the migration between both components is transactional. By using two-phase commit and maintaining open transactions on

```
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

Figure 5: DimTrade Ingestion in TPC-DI

both S-Store and Postgres until both sides of the migration have completed, BigDAWG is able to ensure that a batch of data is either fully migrated or completely rolled back, thereby avoiding data loss or duplication issues.

It is important to note that while data can be migrated from S-Store to BigDAWG or vice versa, we chose not to implement cross-system replication at this time. This is to ensure that we are only maintaining a single copy of the data at once, thereby avoiding data consistency issues associated with replication. We leave cross-system replication to future work. However, queries sent by the user can be automatically routed to the correct OLAP warehouse, and may include UNIONed data from a delta warehouse or from the staging storage of the streaming ETL engine. Complex queries that require data from both S-Store and Postgres can be performed by first migrating the relevant data fully into one engine or the other, and then performing the operation there.

5.2 Migration Experiments (TPC-DI)

5.2.1 *Experiment Setup*

One of the key questions facing a streaming ETL system is how frequently data should be migrated to the data warehouse. As discussed in Section 4.4, there are two methods of determining when this migration occurs: either the ingestion engine periodically *pushes* the data to the warehouse, or the warehouse *pulls* the data from the ingestion engine when it is needed. Thus, we devised an experiment to test the pros and cons of each.

To test the performance of our inaugural system, we implemented a portion of the TPC-DI workload described in Section 2.2. In the streaming ETL engine (S-Store), we implemented the operations required to ingest tuples from Trade.txt into the DimTrade table, as described in Figure 5. The necessary transformations can be broken down into a handful of SQL operations, several of which perform lookups on other dimension tables which are each partitioned on a different key (Figure 5(a)). These operations can be divided into five disparate transactions, each of which accesses a different table (Figure 5(b)). That way, each transaction is able to run single-sited, preventing expensive distributed transactions. The stream ordering constraints on the five-transaction dataflow ensures that although the operations are separated and frequently able to run in parallel, the end result of the workload is still correct.

The tables in the TPC-DI database were also created in Postgres to serve as our data warehouse and populated with historic data for the purposes of running analytical queries. With varying frequency (once every 100 ms to 10 seconds), an analytical query is run on the data available in Postgres. TPC-DI is focused solely on ingestion and does not provide a suitible analytical query for testing purposes. To compensate, we use a pricing summary report query inspired by Q1 from the TPC-H benchmark [22]. This query scans the DimTrade table, and for each type of trade calculates aggregates such as the total and average quantity, and the total and average pricing difference between bid and trade.

We run the experiment under two configurations, one push-based and one pull-based. In both configurations, the streaming ETL engine periodically pushes newly-processed data to the warehouse at an interval amount of time. In the second configuration, we allow the warehouse to pull all fully-processed data available in the streaming ETL engine at the beginning of each analytical query. This allows the query to run with the freshest data available. In both configurations, we measure the effects of frequent or sparse migration on 1) freshness of data available to the OLAP queries, 2) duration of the OLAP query runtime, and 3) maximum latency incurred by the ingestion engine.

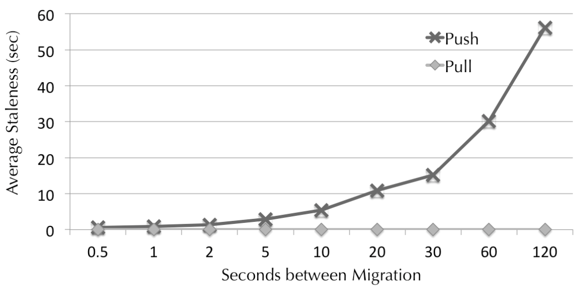
These experiments were run on an Intel® Core™i7 machine with 8 virtual cores and 8 GB of memory. For simplicity, S-Store, BigDAWG, and Postgres were all run on a single node. S-Store was run in single-partition mode, and Postgres used the default settings. Batches are composed of a single tuple per batch.

5.2.2 Results

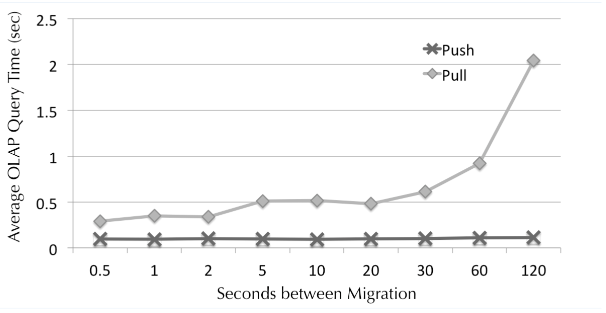
Figure 6 shows the simultaneous effects of the experiment described in Section 5.2.1.

We define data staleness as the amount of time since new data has been migrated into the data warehouse at the time of an analytical query (i.e., the opposite of data freshness). As shown in Figure 6(a), when data is simply pushed from the data ingestion engine to the data warehouse on a regular basis, the data staleness seen by an OLAP query is solely dependent on the transfer interval. As the time between migrations increases, the data staleness for analytical queries does as well. The OLAP query runs at random intervals, and thus may or may not be executed directly after a migration takes place. When the query is repeated multiple times, the staleness averages out to be roughly half the duration between migrations. Meanwhile, if the data warehouse automatically pulls all of the most recent data upon issuing the OLAP query, then the staleness will always be kept at zero. Pulling new data with each query is the clear best option if staleness is the priority.

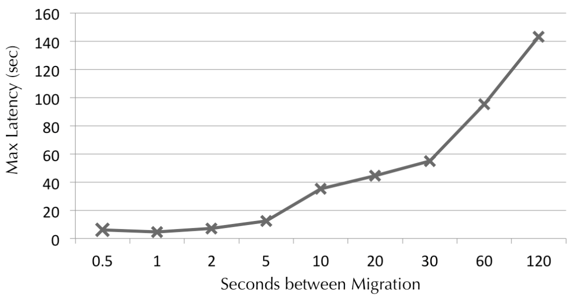
One tradeoff to data freshness involves the run time of the OLAP query in question (Figure 6(b)). In the push case, the analytical query run time is hardly affected at all by migration. Postgres is able to multitask the migration and



(a) OLAP Query Staleness



(b) OLAP Query Run Time



(c) Max Ingestion Latency

Figure 6: Push vs. Pull (at ingestion saturation)

the query well enough that any effect is negligible, and the query runs efficiently. In the pull case (which also features periodic data pushing), performance is affected significantly by the frequency of the push. Each time the query runs, a connection must be established with S-Store in order to receive the latest tuples. There is a baseline overhead in establishing that connection, so even if very little data is transferred, the effect is noticeable in terms of performance. However, if the period between data pushes is large, then the analytical query ends up pulling significantly more data. When this happens, the query time quickly balloons, since the workload is dominated by data migration. If analytical query performance is a top priority, then exclusively pushing data from the ingestion engine (rather than pulling before each query) is ideal.

Data migration also has a significant effect on the performance of the data ingestion engine. Figure 6(c) shows the latency consequences of data migration at various intervals. In this case, latency is measured as the period between a data item first arriving at the ingestion engine and being fully migrated to the data warehouse. Note that there is only a single line in this graph. This is because the effect on the data ingestion is the same regardless of push or pull; only the interval between migrations is important. At saturation, new incoming data items must wait for migration to finish before they can be ingested (because this particular ingestion engine implementation requires a full partition lock during migration). In cases where the interval between migrations is high, more data must be migrated at once, and therefore the maximum amount of time a batch may wait is extremely high. While this effect may be less pronounced in an implementation with looser locking constraints, ingestion performance would still be affected.

In both scenarios (push vs. pull) and all three contexts (staleness, run time, and max ingestion latency), the situation is significantly improved by smaller, more frequent migrations. Setting the time between migrations to be somewhere between one and five seconds gives optimal results for each context. Frequent migrations also mean that data pulls for queries that require high degrees of data freshness have good performance as well.

5.3 IoT Proof-of-Concept (MIMIC II)

In addition to our TPC-DI experiments for relational ETL, we are also able to create a workload that imitates an IoT use-case on our current streaming ETL system implementation. As described in our previous work, we have implemented a real-time alert monitoring system over streaming patient waveforms from an ICU [20]. This is based on the MIMIC II dataset, which includes a variety of ICU data such as different types of heart-rate data [4].

In our demonstration, we ingest ECG and PAP heart-rate data in real-time and run two queries to search for abnormal patterns. One query ensures that PAP levels are above a given threshold, while the second looks for specific patterns within the ECG waveform. In the event of either case, an alert is sent to medical devices intended to notify professionals. The MIMIC dataflow also includes transformation and ingestion into SciDB [6], an array database, via BigDAWG. There, the waveform is stored in a warehouse for later analysis.

While we were able to successfully implement the workload, we noticed some limitations. Though S-Store is designed to handle ordered batch data (which share many properties with time-series data) in a streaming context, its query support is extremely rooted in relational databases. It is not truly optimized to perform complex time-series-specific operations. SciDB contains a similar problem; while time-series can be stored as a one-dimensional array, an array database lacks support for complicated time-series queries. Also, because both S-Store and SciDB center around strong support for mutable state rather than append-heavy time-series workloads, there are other missed opportunities for optimization.

Our MIMIC II implementation led us to the conclusion that a new, single system could be created to both ingest and analyze time-series at scale. We discuss our plans for such a system in the next section.

6. FUTURE WORK: TIME-SERIES ETL

While ETL for relational data involves mutation of shared state, IoT datasets primarily require consideration of time-series data, which tends to be very insert-heavy and requires operators capable of efficiently analyzing waveforms. We believe that the system we built in this paper can be further improved and optimized for an IoT use-case. Using the requirements and architecture that we have established for a streaming ETL system, we can create an all-in-one ingestion and analytics engine specifically for time-series data. We call this system Metronome.

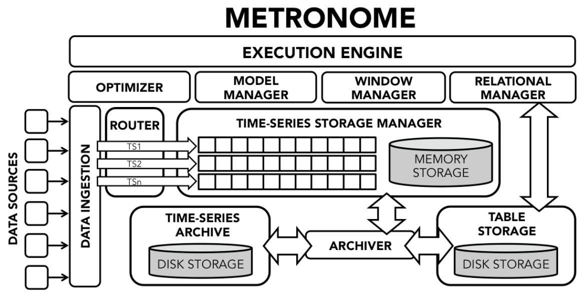


Figure 7: Metronome Architecture

There are a wide variety of use-cases which prominently feature time-series. Classically, time series are used for forecasting (i.e., prediction) and pattern matching (e.g., outlier detection). Additionally, when treated as a signal, time series can participate in FFTs, convolutions, and filtering. We believe that by treating time-series as the primary data type, a single data model can be used for all of these situations. We will then be able to specialize the generic data model by implementing operations that are specific to each time-series use-case.

Unlike most stream processing systems, Metronome's fundamental data type is a finite sequence of values (a window) that represents a contiguous piece of a time-series. Rather than relational tables, time-series store data in a named sequence of values indexed by time. A time-series is primarily updated by appending new values at higher time-index positions, with tertiary support for updating values in specific cases (such as error correction).

Because both the streaming ETL operations and the online analytics processing are concerned with time-series data, we believe that Metronome has the capability to handle an integrated streaming ETL architecture which includes both real-time data ingestion and time-series analytics. Figure 7 illustrates the basic architecture of Metronome. Each of the functionalities from Section 4 are supported. However, because the entire dataflow stack is optimized within a single system, components of the streaming ETL architecture do not directly map one-to-one with components of Metronome.

Streaming data ingestion for time-series shares many similarities to that of relational data, and has the same requirements listed in Section 3. However, with the change in fundamental data type, the priorities for these requirements change substantially. For instance, relational data prioritizes functionality that allows for update-in-place, such as ACID transactions. For time-series however, workloads are append-mostly; as new data arrives, it gets appended to the end of an existing time-series. As a result, the isolation guarantees of ACID can be relaxed without significant risk to correctness and at the benefit of performance. Traditional ETL requirements such as bulk loading and heterogeneity can also be de-prioritized, as the incoming time-series data is always stored as a time-series and that storage is local to Metronome.

While some requirements can be relaxed, the need for others becomes even more pronounced. Streaming requirements, in particular, are extremely important. For instance, when receiving information from thousands of devices in an IoT workload, out-of-order tuples are a frequent occurrence that needs to be directly addressed by the data collection mechanism. Each sensor likely generates its own timestamps, and even if the clocks are extremely accurate, they are almost certainly offset from one another. Additionally, data items may become delayed in-transit to Metronome. Thus, the data collection mechanism must be responsible for synchronizing incoming tuples into simultaneous batches. It is also unreasonable for a full batch to be held from processing as the system waits for tuples that it knows are missing or lost. Instead, the data collection mechanism should be able to use historic data to predict what those missing tuples are whenever possible.

As discussed in Section 3.3.3, scalability is also a crucial factor when considering data collection and analysis in a single system. Because resources will be directly shared between the two, it is very important that long-running analytics queries do not strongly affect ingestion performance, or vice versa. To accomplish this, it is necessary to intelligently determine when to migrate data between components based on resource availability, in much the same way as the prototype implementation.

Using the lessons learned in our implementation of streaming ETL, we believe that Metronome can be a novel, next-generation approach to the ingestion and analysis of time-series data at an IoT scale.

7. RELATED WORK

There has been a plethora of research in ETL-style data ingestion [12, 23]. 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 [24]. It has been commonly recognized that fine-granular ETL comes with consistency challenges [10, 8]. 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) [9]. 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 [19]. This a three-layer (edge-fog-cloud) architecture like ours. 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 [3, 2]. 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 [1], Gorilla [17], and OpenTSDB [25]. 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.

8. CONCLUSIONS

This paper makes the case for streaming ETL as the basis for improving the currency of the warehouse, 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. Our proof-of-concept implementation using the S-Store system [15] shows the viability of this approach in real-world use cases such as brokerage firm ingestion (TPC-DI) and medical sensor monitoring (MIMIC II).

IoT points out how in many applications the streams can be viewed as time-series. We believe that high-speed ingestion support is necessary, but that we can do even better when the input streams are known to be time-series. Thus, our future research directions include evaluating and comparing the performance of our approach against other ETL alternatives using IoT-scale benchmarks as well as enhancing our approach with time-series data management support.

9. 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.

10. REFERENCES

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

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

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

[4] MIMIC II Data Set. https://physionet.org/mimic2/.

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

[6] P. G. Brown. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *SIGMOD*, June 2010.

[7] A. Elmore et al. A Demonstration of the BigDAWG Polystore System. *PVLDB*, 8(12), August 2015.

[8] L. Golab and T. Johnson. Consistency in a Stream Warehouse. In *CIDR*, January 2011.

[9] R. Grover and M. Carey. Data Ingestion in AsterixDB. In *EDBT*, March 2015.

[10] T. Jorg and S. Dessloch. Near Real-Time Data Warehousing using State-of-the-art ETL Tools. In *BIRTE Workshop*, August 2009.

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

[12] R. Kimball and J. Caserta. *The Data Warehouse ETL Toolkit*. Wiley Publishing, 2004.

[13] J. Kreps, N. Narkhede, and J. Rao. Kafka: A Distributed Messaging System for Log Processing. In *NetDB Workshop*, June 2011.

[14] S. Kulkarni et al. Twitter Heron: Stream Processing at Scale. In *SIGMOD*, June 2015.

[15] J. Meehan et al. S-Store: Streaming Meets Transaction Processing. *PVLDB*, 8(13), September 2015.

[16] J. Meehan et al. Integrating Real-Time and Batch Processing in a Polystore. In *IEEE HPEC Conference*, September 2016.

[17] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. Gorilla: A Fast, Scalable, In-memory Time Series Database. *PVLDB*, 8(12):1816–1827, 2015.

[18] M. Poess et al. TPC-DI: The First Industry Benchmark for Data Integration. *PVLDB*, 7(13), August 2014.

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

[20] N. Tatbul et al. 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.

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

[22] Transaction Processing Performance Council (TPC). TPC Benchmark H (Version 2.17.1). http://www.tpc.org/tpch/, Nov. 2014.

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

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

[25] T. W. Wlodarczyk. Overview of Time Series Storage and Processing in a Cloud Environment. In *IEEE CloudCom Conference*, pages 625–628, 2012.

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