Liquid: Unifying Nearline and Offline Big Data Integration

Raul Castro Fernandez*, Peter Pietzuch
Imperial College London
{rc3011, prp}@doc.ic.ac.uk

Jay Kreps, Neha Narkhede, Jun Rao
Confluent Inc.
{jay, neha, jun}@confluent.io

Joel Koshy, Dong Lin, Chris Riccomini, Guozhang Wang
LinkedIn Inc.
{jkoshy, dolin, criccomini, gwang}@linkedin.com

ABSTRACT

With more sophisticated data-parallel processing systems, the new bottleneck in data-intensive companies shifts from the back-end data systems to the *data integration stack*, which is responsible for the pre-processing of data for back-end applications. The use of back-end data systems with different access latencies and data integration requirements poses new challenges that current data integration stacks based on distributed file systems—proposed a decade ago for batch-oriented processing—cannot address.

In this paper, we describe *Liquid*, a data integration stack that provides low latency data access to support near real-time in addition to batch applications. It supports incremental processing, and is cost-efficient and highly available. Liquid has two layers: a *processing layer* based on a stateful stream processing model, and a *messaging layer* with a highly-available publish/subscribe system. We report our experience of a Liquid deployment with back-end data systems at LinkedIn, a data-intensive company with over 300 million users.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

Data integration, stateful stream processing, Big Data processing

1. INTRODUCTION

Web companies such as Google, Facebook and LinkedIn generate value for their users by analyzing ever-increasing amounts of data. Higher user-perceived value means better user engagement, which, in turn, generates even more data. While this high volume of append-only data is invaluable for organizations, it becomes expensive to integrate using proprietary, often hard-to-scale data warehouses. Instead, organizations create their own *data integration stacks* for storing data and serving it to back-end data processing systems. Today's data integration stacks are frequently based on a MapReduce (MR) model [5]—they run custom ETL-like MR jobs on commodity shared-nothing clusters with scalable distributed file systems (DFS) such as GFS [10] or HDFS [32] in order to produce data for back-end systems [22].

*Work produced while doing an internship at LinkedIn

While in the past processing performance was limited, a new breed of data-parallel systems such as Spark [42] and Storm [36] has helped mitigate processing bottlenecks in back-end systems. As a result, the pendulum has swung back, making the data integration stack performance critical. For example, for *nearline* data processing systems, i.e. back-end systems that are stream-oriented and therefore require low-latency, high-throughout data access, the use of a DFS as the storage layer increases data access latency, thus impacting the performance of applications.

Many organizations today use a MR/DFS stack for data integration: the storage layer uses a DFS to store data in a cost-effective way, sharding it over nodes in a cluster; the processing layer executes batch-oriented MR jobs, which clean and normalize the data, perform pre-filtering or aggregation, before the data is used by back-end systems. Such a design for a data integration stack, however, has several limitations, affecting performance and cost:

1. Intermediate results of MR jobs are written to the DFS, resulting in **higher latencies** as job pipelines grow in length. Despite many efforts to reduce access latencies [43, 7], the coarse-grained data access of a MR/DFS stack is only appropriate for batch-oriented processing, limiting its suitability for low-latency nearline back-end systems.
2. To avoid re-processing all data after updates, back-end systems must support **incremental processing**. This requires the data integration stack to offer fine-grained access to the data and maintain explicit incremental state. A MR/DFS stack lacks these features—it can only offer coarse-grained access for batch-oriented processing, and does not handle transient computation state. While this facilitates failure recovery, it makes the implementation of efficient incremental processing challenging [4].
3. A data integration stack typically provides the raw "source-of-truth" data that many back-end systems consume through ETL-like jobs. As a result, the stack executes many jobs concurrently, and it must thus guarantee **resource isolation** so that faulty jobs cannot compromise the infrastructure. Despite recent efforts to control resources in clusters [16, 38], it is challenging to maintain resource isolation during low latency processing without over-provisioning of the infrastructure.

To address the above challenges, a common approach is for back-end systems to execute on their own duplicate copies of the "source-of-truth" data [35, 25]. This approach breaks the single "source-of-truth" abstraction and requires the handling of divergent data replicas. It also introduces substantial engineering effort to provide point-to-point channels from all data sources to all back-end systems, which does not scale with the number of back-end systems.

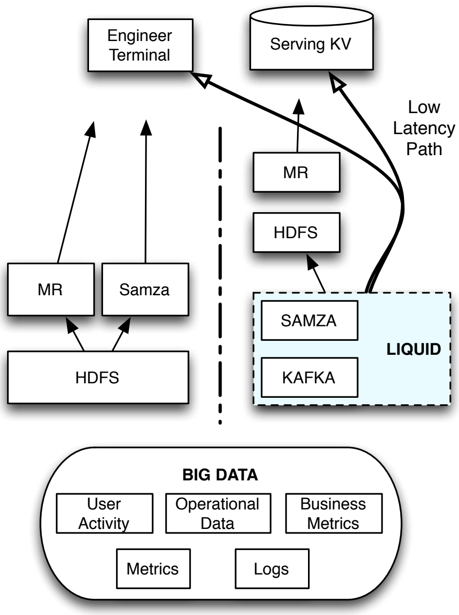

Figure 1: Data integration with Liquid

Finally, it also increases the hardware footprint and adds operational challenges, leading to a more brittle infrastructure.

Instead, we argue that it is time to rethink the architecture of a data integration stack. Rather than using the legacy MR/DFS model for data integration shown on the left of Figure 1, we describe **Liquid**, a new nearline data integration stack used at LinkedIn, as shown on the right. The Liquid stack has the following properties:

(i) Low latency. Fast data access to new data should be the default behavior. Liquid provides online data access to satisfy the stringent requirements of nearline back-end systems, and it also pushes data to batch-processing systems for offline processing.

(ii) Incremental processing. Liquid supports efficient incremental processing. It *annotates* data with metadata such as timestamps or software versions, which back-end systems can use to read from a given point. This *rewindability* property is a crucial building block for incremental processing and failure recovery.

(iii) High availability. Liquid provides the "source-of-truth" data that different back-end systems consume. For this reason, the data must be highly available for both reads and writes under common cluster failures [39].

(iv) Resource isolation. Liquid permits ETL-like jobs to execute centrally, instead of being managed by independent product teams in an organization. This allows for easier control of job resources and service guarantees.

(v) Cost effectiveness. Liquid must store a high volume of incoming data with high throughput, while maintaining low operational cost. This means that Liquid cannot keep all data in RAM to reduce access latency [24].

As shown in Figure 1, Liquid consists of two cooperating layers in order to achieve the above properties:

1. a **messaging layer** provides data access based on metadata, which permits back-end systems to read data from specific points in time. It is distributed to scale to high data volumes while remaining highly available;
2. a **processing layer** executes ETL-like jobs for back-end systems, guaranteeing low-latency data access. This layer can perform arbitrary data processing before passing data to back-end systems, ranging from data cleaning and normalization, to the computation of aggregate statistics or the detection of anomalies in the data.

As a result of this design, Liquid inherits the beneficial separation of storage and computation from the MR/DFS stack, resulting in two advantages: (i) it can scale storage and computation independently; and (ii) when fault tolerance is hard to achieve in the processing layer, e.g. with explicit state as part of the computation, it is possible to fall back to the highly-available messaging layer.

The **messaging layer** uses Apache Kafka [19], a highly-available distributed publish/subscribe messaging system. Data is stored persistently in distributed commit logs, which are replicated for high availability. For high-throughput data access, the messaging layer exploits the default "anti-caching" behavior [6] of modern OS file system caching, in which data is kept in RAM by default and evicted to disk if not used for a period of time. Since commit logs are append-only, it becomes possible to define accurately the period of time after which data should be flushed to disk, improving performance for back-end systems reading the head of the log.

The **processing layer** is implemented using Apache Samza [30], a distributed stream processing framework that follows a stateful processing paradigm [3, 4]. It executes ETL-like jobs efficiently and with low latency, representing state explicitly as part of the computation. The processing layer stores annotated data in the messaging layer as metadata, which permits jobs to choose input data streams dynamically.

In the next section, we motivate the need for a new nearline data integration stack in more detail. We explain the design of Liquid in §3 and its implementation in §4. In §5, we describe our experience running Liquid at LinkedIn as part of production systems with 300 million users. §6 discusses related work, and §7 concludes.

2. MOTIVATION

Modern web companies have specialized data processing systems that can be classified as *online*, *nearline*, and *offline*. At LinkedIn, the resource distribution for back-end systems according to these classes is approximately 25%, 25%, and 50%, respectively. Online systems, e.g. for processing business transactions, are handled by front-end relational databases and serve results within milliseconds. Nearline systems typically operate in the order of seconds, and offline systems within minutes or hours. As companies realize the increased business and user value of analyzing data with low latency, the trend is towards an increased number of resources for nearline back-end systems.

Both nearline and offline systems ingest data from a data integration stack. Next we discuss some requirements of these back-end systems, highlighting the problems of current stacks.

2.1 Requirements

Nearline applications. Applications that query a social graph, search data, normalize data, or monitor change are classified as *nearline*: the sooner they provide results, the higher the value to end users. As processing pipelines have more stages, the end-to-end latency also increases, which makes nearline processing more challenging. Fundamentally, DFS-based stacks do not support low-latency processing because they have a high overhead per stage: they are designed for coarse-grained data reads and writes.

Offline applications. Typical offline applications in web companies are recommendation systems, OLAP queries and batch-oriented MR-style jobs, e.g. for the generation of hourly reports or the training of machine-learning models.

For example, user-tracking data, such as click streams and actions, is generated by front-end web servers and stored in a DFS. ETL-like jobs transform the data according to the requirements of different back-end systems and deliver it for processing. After the required results are obtained, e.g. recommendations for individual users, the result data is loaded into serving layers, such as key-value stores, from which applications can access it with low latency.

When the input data changes, back-end systems require the latest data, which typically means that the processing pipeline must *re-execute from scratch*. This includes the final loading of result data into serving layers, which incurs a high overhead. To avoid that users are presented with increasingly stale results as data amounts grow, the hardware footprint of the infrastructure increases to reduce the re-execution time of the pipeline. In practice, different product teams therefore introduce their own incremental processing systems, which have to be maintained independently.

Additional requirements. Most back-end systems accessing a data integration stack have additional requirements. For certain jobs, it is important to have access to the *data lineage*, i.e. information about how the data was computed. In addition, *access control* is necessary to ensure that no faulty or misconfigured back-end systems can compromise the data of other applications. Finally, the data integration stack should be responsible for executing jobs with adequate service guarantees through *resource isolation*, which we refer to as "ETL-as-a-service".

2.2 Current approaches

When attempting to support nearline back-end systems, there are two common approaches that organizations adopt to improve data access latency, both of which come at the cost of an increased hardware footprint:

1. they *bypass* the data integration stack, sending data directly to back-end systems. This has two problems: (i) each system requires a new point-to-point connection to the data, which increases operational complexity; and (ii) since data is not stored in the stack, it is not available to other applications.
2. they *duplicate* the data so that back-end systems can access their own copies [35]. This incurs the complexity of handling replica divergence, and it does not guarantee low latency access if processing pipelines are complex.

As organizations have discovered the limitations of today's data integration stacks, the above approaches have led to new architectural patterns:

Lambda architecture [23]. In this pattern, input data is sent to both an offline and an online processing system. Both systems execute the same processing logic and output results to a service layer. Queries from back-end systems are executed based on the data in the service layer, reconciling the results produced by the offline and online processing systems.

This pattern allows organizations to adapt their current infrastructures to support nearline applications [21]. This comes at a cost, though: developers must write, debug, and maintain the same processing code for both the batch and stream layers, and the Lambda architecture increases the hardware footprint.

Kappa architecture [20]. In this pattern, a single nearline system, e.g. a stream processing platform, processes the input data. To re-process data, a new job starts in parallel to an existing one. It re-processes the data from scratch and outputs the results to a service layer. After the job has finished, back-end systems read the data loaded by the new job from the service layer. This approach only requires a single processing path, but it has a higher storage footprint, and applications access stale data while the system is re-processing data.

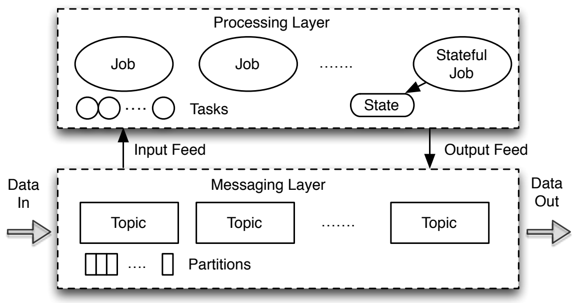

Figure 2: Architecture of the Liquid data integration stack

In summary, implementing the above architectural patterns in current MR/DFS-based data integration stacks introduces a range of problems, including an increased hardware footprint, data and processing duplication and more complex management. To overcome these issues and provide more flexibility for the new requirements of nearline applications, we describe a new data integration stack that acts as a more efficient substrate for back-end systems by providing low-latency data access by default.

3. LIQUID DESIGN

We describe the design of *Liquid*, a nearline data integration stack with two independent, yet cooperating layers that achieve the above requirements, as shown in Figure 2. A *processing layer* (i) executes ETL-like jobs for different back-end systems according to a stateful stream processing model [3]; (ii) guarantees service levels through resource isolation; (iii) provides low latency results; and (iv) enables incremental data processing. A *messaging layer* supports the processing layer. It (i) stores high-volume data with high availability; and (ii) offers rewindability, i.e. the ability to access data through metadata annotations.

The two layers communicate by writing and reading data to and from two types of *feeds*, stored in the messaging layer (see Figure 2): *source-of-truth feeds* represent primary data, i.e. data that is not generated within the system; and *derived data feeds* contain results from processed source-of-truth feeds or other derived feeds. Derived feeds contain *lineage information*, i.e. annotations about how the data was computed, which are stored by the messaging layer. The processing layer must be able to access data according to different annotations, e.g. by timestamp. It also produces such annotations when writing data to the messaging layer.

Back-end systems read data from the input feeds, after Liquid has pre-processed them to meet application-specific requirements. These jobs are executed by the processing layer, which reads data from input feeds and outputs processed data to new output feeds.

The division into two layers is an important design decision. By keeping both layers separated, producers and consumers can be decoupled completely, i.e. a job at the processing layer can consume from a feed more slowly than the rate at which another job published the data without affecting each other's performance. In addition, the separation improves the operational characteristics of the data integration stack in a large organization, particularly when it is developed and operated by independent teams: separation of concerns allows for management flexibility, and each layer can evolve without affecting the other.

3.1 Messaging layer

Next we describe the design of the messaging layer, which is based on a topic-based publish/subscribe communication model.

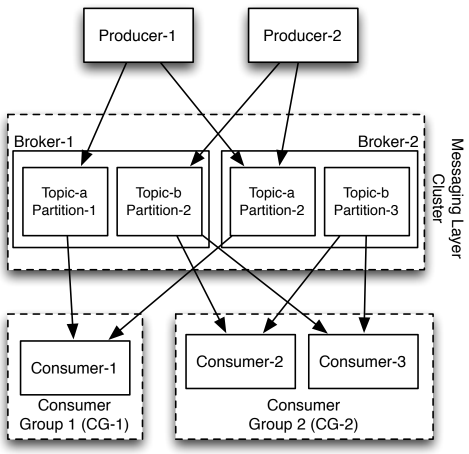

Figure 3: Architecture of the messaging layer

The goal of data integration is to make data available to an entire organization. Each team in an organization is only interested in consuming part of that data, and they only need to produce data of certain types. Therefore they should be able to do this while being unaware of other irrelevant data. A publish/subscribe model [8] fits this requirement: teams can *subscribe* to particular *topics*—types of data according to a schema or semantics—in order to access data, which is offered as a function by the data integration stack.

Although other communication models provide similar decoupling between producers and consumers, a publish/subscribe model has the advantage of abstracting data delivery, making it independent of application usage. This allows the messaging layer to be operated as a service, e.g. identifying misbehaving applications or deciding which data is requested more for load-balancing purposes. The use of publish/subscribe communication by the messaging layer thus supports (i) high throughput for reads and writes; (ii) high-availability for data access; and (iii) arbitrary data access based on metadata.

Topic-based publish/subscribe. In a topic-based publish/subscribe model, data is divided into *messages*, which are stored under different *topics*. Topics separate data into categories that are meaningful to applications, e.g. according to the system that produced the data. Clients of the messaging layer are called *producers* and publish data to different topics, such as front-end applications generating events. The example in Figure 3 shows two producers: *Producer-1* publishes data to *Topic-a* and *Producer-2* publishes to both *Topic-a* and *Topic-b*.

Consumers subscribe to topics, reading data from the messaging layer, and are typically back-end applications. Figure 3 shows three consumers: *Consumer-1* is subscribed to *Topic-a*, and *Consumer-2* and *Consumer-3* are subscribed to *Topic-b*. Typically hundreds of clients can produce and consume data to and from a topic.

To achieve high write/read throughput, topics are divided into *partitions*, which are distributed on a cluster of *brokers*. Each broker runs on a different physical machine that handles topics and the partitions for these topics by answering requests from clients, i.e. producers and consumers.

Figure 3 shows how the two topics have two partitions each (*Partition-1* and *Partition-2* for *Topic-a*, and *Partition-2* and *Partition-3* for *Topic-b*), which are stored across the two brokers. A single broker may handle partitions for different topics, such as *Broker-1* handling *Topic-a* and *Topic-b*. Producers can choose to which partition to publish data in a round-robin fashion or according to a hash function for load-balancing or semantic routing.

Consumers are divided into *consumer groups*, which gives further flexibility in how clients consume data. At the level of consumer groups, the messaging layer behaves as a publish/subscribe system, i.e. each consumer group receives messages from topics to which it is subscribed. However, only one consumer within each consumer group receives a given message, i.e. the system behaves as a queue for the consumers within a consumer group.

In Figure 3, there are two consumer groups (CGs): *CG-1* is subscribed to *Topic-a* and *CG-2* to *Topic-b*. When a new message is published to *Topic-b*, only *Consumer-1* or *Consumer-2* receive that message. All consumers in *CG-2* read data from brokers as if it was a queue, which helps load-balance the load across the consumers in a consumer group. Now consider a case in which more than one consumer group is subscribed to *Topic-a*. When a new message is published to *Topic-a*, one consumer of each subscribed consumer group is guaranteed to receive the message.

Distributed commit log. Each topic is realized as a *distributed commit log*, in which each partition is append-only and keeps an ordered, immutable sequence of messages with a unique identifier called an *offset*. An append-only log therefore keeps both messages and offsets in the natural order in which they were appended.

The distributed commit log is a crucial design feature of the messaging layer: (i) its simplicity helps create a scalable and fault-tolerant system; and (ii) its inherent notion of order allows for fine-grained data access. While the append-only log only guarantees a total order of messages per topic-partition but not across partitions, we observe that, in practice, this is sufficient for most back-end applications.

Clients of the messaging layer use offsets to keep track of the latest consumed data per partition. Consumers pull data from brokers by providing a set of offsets. After a pull request, brokers return the latest data after the specified offsets. This approach makes it efficient to maintain the latest consumed data, i.e. it requires only storing a single integer per partition.

Metadata-based access. The messaging layer uses a highly-available, logically-centralized *offset manager* to maintain annotations on the data, which can be queried by clients. For example, consumers can *checkpoint* their last consumed offsets to save their progress; after failure, they can ask for the last data that they processed. To re-process data, clients can include metadata, such as timestamps, with the offsets and retrieve data according to these previously-stored timestamps. We discuss other use cases for the offset manager in §4.2.

3.2 Processing layer

As shown in Figure 2, a *job* in the processing layer embodies computation over *streams*, which are the input data coming from feeds in the messaging layer. The job processes messages from an *input feed* and produces output messages, which can be published to a derived *output feed*. For parallel processing, a job is divided into *tasks* that process different partitions of a topic. The data for a *stateless* job is entirely contained in the input stream, while a *stateful* job has explicit state that evolves as part of the computation.

Jobs can communicate with other jobs, forming a *dataflow* processing graph. All jobs are decoupled by writing to and reading from the messaging layer, which avoids the need for a back-pressure mechanism. This is an important design decision that improves the operational robustness of the system.

Stateful processing. Stateful jobs access state locally for efficiency. State can be represented as arbitrary data structures, e.g. a window of the most recent stream data, a dictionary of statistics or an inverted index used for search queries.

Failure recovery is more challenging for stateful jobs. Our solution is for the processing layer to publish state updates to a *changelog*, which is a derived feed stored by the messaging layer. After failure, state is reconstructed from the changelog.

Incremental processing. The processing layer can process data incrementally by exploiting explicit state and the functionality of the metadata manager. A job can periodically checkpoint the offsets that it has consumed and maintain a summary of the input data as its state. When new input data becomes available, the job can thus ignore already processed data.

As a consequence, incremental processing reduces the load on the back-end systems because only the latest results are incorporated, as opposed to bulk loading all data from scratch. This is particularly important in scenarios in which only a small percentage of data changes periodically, such as user profile updates.

ETL-as-a-service. The processing layer executes multiple ETL-like jobs submitted by different back-end systems, and therefore can control the resources used by each at a fine granularity. To isolate resources on a per-job basis, the processing layer can use standard resource isolation mechanisms such as container-based OS isolation (see §4.4).

4. IMPLEMENTATION

This section describes a range of implementation features of Liquid, which enable it to achieve the desired requirements: (i) low-latency data access; (ii) incremental processing; (iii) high availability; (iv) resource isolation; and (v) cost effectiveness. Liquid is formed by Apache Kafka (approx. 25,000 lines of code), Apache Samza (approx. 7,000 lines of code), and integration code between both.

4.1 Low-latency data access

Despite accessing large amounts of data, Liquid must achieve high-throughput and low-latency data reads and writes. It uses two strategies to achieve this goal: efficient data distribution, i.e. parallel at both the messaging and processing layers, and fast storage. We described its distributed design in §3.1—in this section, we focus on a single node and explain how Liquid achieves fast reads and writes when using append-only disk-backed logs to store data.

Append-only log. Data is written to an append-only log that is persisted to disk for durability. Since this is at odds with the need for high-throughput writes and reads in the messaging layer, we explain some implementation decisions that improve performance.

To achieve *high-throughput writes*, the messaging layer relies on OS-level file system caching: the OS maintains data in RAM first and flushes it to disk after a configurable timeout parameter, as inspired by recent "anti-caching" designs [6]. The messaging layer also exploits knowledge about the layout of data on disk to define when data is evicted. Since the append-only log has a natural sequential order, it becomes possible to determine which data has to be flushed. This permits the head of the log to be maintained in memory for back-end systems that need low-latency access.

For *high-throughput reads*, brokers maintain an incrementally-built index file that is used to select the chunks of the log at which requested offsets are stored. Once these are located, the initial reads are slower due to the OS loading pages into RAM; after typically a few seconds, successive reads become fast due to prefetching. Note that prefetching is only needed for random access reads; sequential reads can retrieve data from the file system cache.

The append-only design of the log has another important advantage in a production environment: *read/write throughput remains constant independent of log size*. This allows for larger logs, leading to good resource utilization—fewer machines can store more data, which achieves cost-effective storage. As a result, data can be kept for longer in the messaging layer, typically in the order of weeks to months.

Log retention. To put a bound on the amount of data that is stored, a *retention* period is configured per topic. This period is usually expressed in terms of time, e.g. one month worth of data, but for operational reasons it may also be configured as a maximum log size. The disk-based message store maintains messages for longer in a cost-effective way, which offers back-end systems a margin to decide whether they need to consume data or not, thus enhancing their capacity for rewindability.

Log compaction. Some applications only require the last update for a specific key, e.g. a given user. In this case, there is a further opportunity to reduce the total space consumed by the log through *compaction*, which increases storage utilization. The log is scanned asynchronously, de-duplicating messages with the same key and keeping only the most recent data for each key.

Log compaction is particularly useful in order to reduce the size of the changelogs that store state checkpoints (see §3.2). The state used by the processing layer in the form of map, matrices or arrays is typically keyed on an attribute. Therefore it is sufficient to keep the latest update for each key to recover the state after failure. In this case, performing log compaction not only reduces the changelog size, but it also allows for faster recovery.

4.2 Incremental processing

Liquid facilitates incremental processing by permitting the processing layer to annotate incoming streams from the messaging layer with arbitrary metadata. All annotations are stored by the offset manager (see §3.1), which maintains a map of offsets to the metadata, such as the software version that consumed a given offset, or the timestamp at which data was read. With this mechanism, back-end systems can query for the latest offsets and retrieve the newest data, thus only updating the state in the processing layer.

Consider the problem of maintaining statistics about the data for a given topic that is periodically updated, e.g. every few hours. In this case, reading all data each time that it changes would be infeasible—the required time would increase linearly with data size.

Instead, the processing layer can read the available data, compute such statistics and maintain them as state. After consuming some data, the processing layer checkpoints the offsets in the offset manager. When new data arrives, it fetches the offsets from the offset manager and reads only the new data, appending new results to its state. In the general case, such offsets are cached in the processing layer, and fetching them from the offset manager is only necessary after a failure in the processing layer.

4.3 High availability

Data is persisted in log files, but brokers may fail, leaving the data unavailable. To avoid this situation, all partitions handled by a *lead broker* are replicated across *follower brokers*. If a lead broker fails, a hand-over process selects a new leader among its followers.

A follower broker acts as a normal consumer, reading data from its lead broker and appending it to its local log. This means that the followers for a given partition may not have incorporated all data from the lead broker when it fails. A coordination service (Apache Zookeeper [17]) is used to maintain a set of *in-sync-replicas* (ISRs), which are the subset of followers that are above a configurable minimum up-to-date threshold. After a broker failure, a re-election mechanism chooses a new leader from the set of ISRs.

This design guarantees that the messaging layer can tolerate up to *N*-1 failures with *N* brokers in the set of ISRs. Maintaining a large *N*, however, exposes a performance/durability trade-off: the maximum durability is achieved when a lead broker sends data to all followers and waits for all acknowledgments; the minimum durability is obtained if acknowledgments are returned to clients immediately after receiving a message. The chosen durability level impacts the throughput and latency of the data integration stack.

Delivery guarantees. In the current implementation, the messaging layer provides *at-least-once* delivery semantics. There is no built-in support to detect duplicates that can occur after a failure when replying data from an already processed offset. This is sufficient for applications that only handle keyed data with idempotent updates, because duplicates can be detected easily by the application. This is not the case for all applications, however, and there is an ongoing effort to design and implement support for *exactly-once* semantics.

4.4 Resource isolation

To have Liquid offer ETL-as-a-service, the processing layer must guarantee that all jobs achieve a minimum level of service. This is challenging for two reasons: (i) resource-intensive jobs may affect other jobs running on the same infrastructure; and (ii) the implementation executes in a Java VM, which means that garbage collection impacts performance.

The Liquid implementation addresses these challenges as follows. In the messaging layer, partitions are load-balanced across all available clusters, which achieves a better balance of jobs at the processing layer. The processing layer uses OS-level resource isolation, as realized by Linux containers in Apache YARN [38], thus restricting the memory and CPU resources of each job.

For stateful processing jobs, the bulk of the memory consumption is due to the maintained state. To avoid frequent invocation of the garbage collector in the JVM when state is managed, the processing layer allocates the state off-heap by using RocksDB [29], a persistent key-value store.

4.5 Cost effectiveness

Keeping resource and operational costs low is an important requirement for a system that is expected to store and process an ever-increasing amount of data. Although the cost of RAM has decreased in recent years, it is still not cost-effective to store all data of the data integration stack in memory. To achieve low-latency access even when data is stored on disk, the messaging layer exhibits an anti-caching behavior, therefore supporting high throughput reads and writes while keeping data durable.

Multi-tenancy is the norm in data-intensive organizations. Multiple independent teams may be executing different applications on the same cluster, leading to resource contention. To retain a given quality-of-service per application, while maintaining a high cluster utilization, Liquid uses a resource management layer that isolates resources on a per-application basis.

5. EXPERIENCE WITH LIQUID

A Liquid deployment is pervasive across the back-end systems at LinkedIn. The messaging layer, based on Apache Kafka, runs in 5 co-location centers, spanning different geographical areas. It ingests over 50 TB of input data and produces over 250 TB of output data daily (including replication). For this, it uses around 30 different clusters, comprised of 300 machines in total that host over 25,000 topics and 200,000 partitions.

The processing layer, based on Apache Samza, spans across 8 clusters with over 60 machines. Overall, Liquid is deployed on more than 400 machines that perform data integration and adaptation for back-end and front-end systems.

5.1 Real-world use cases

Next we describe a range of applications built on top of Liquid. We also discuss some of the encountered problems and challenges.

Data cleaning and normalization. A crucial task in many organizations is to clean and normalize user-generated content. This is typically done by specialized algorithms that, e.g. disambiguate entities or detect synonyms in text data. To achieve best results, algorithms must operatore on the latest content, which is challenging because (i) users continuously generate new content; and (ii) engineers continuously optimize their processing algorithms.

These two challenges require different system properties: when users generate new content, the cleaning pipeline must have low-latency, so that new information is incorporated quickly, e.g. appearing when users search the website; when the source code of the cleaning pipeline changes, it is necessary to re-process data with the new algorithm so that all data was cleaned with the same algorithm.

Before the deployment of Liquid, there were two different sub-systems for data cleaning, one for the nearline case (i.e. new content from users) and another for the batch case (i.e. changes in the pipeline code). This meant that each time that new cleaning code was written, it had to be tested against both cases, which was time-consuming and error-prone. Even worse, these sub-systems were shared by different teams, making resource isolation impossible: bugs in one sub-system, affected the other.

The use of Liquid brought several benefits: it achieved (i) more efficient re-processing, i.e. it is now easier to integrate the latest user-generated data with current results, or to clean past data with new algorithms; (ii) resource isolation, i.e. multiple algorithms can execute in parallel (e.g. as required for A/B testing), without affecting each others performance; and (iii) lower data access latency, which allows back-end systems to serve freshly cleaned data.

Site speed monitoring. To improve user experience, web companies monitor the page loading times by tracking client-generated events, often referred to as *real user monitoring* (RUM). Events are stored first and analyzed later to detect anomalies and performance problems in the loading times. A fundamental issue with this approach is that problems are not detected promptly, which prevents corrective actions to be issued in real-time. For example, if the root cause of a page loading problem is quickly isolated to a particular CDN, traffic can be re-routed to different servers.

With Liquid, when a client visits a webpage, an event is created that contains a timestamp, the page or resource loaded, the time that it took to load, the IP address location of the requesting client and the content delivery network (CDN) used to serve the resource. These events are consumed by Liquid, which groups them by location, CDN, or other dimensions.

Based on this data, Liquid can feed back-end applications that detect anomalies: e.g. CDNs that are performing particularly slowly, or increased loading times from specific client locations. Back-end applications can consume already pre-processed data that divides user events per session. As a result, back-end applications can detect anomalies within minutes as opposed to hours, permitting a rapid response to incidents.

Call graph assembly. At LinkedIn, dynamic web pages are built from thousands of REST calls, which are executed by distributed machines. Each call can subsequently trigger other calls, and the responses of all these calls constitute the generated web page. This makes it important to detect slow calls, which indicate problems with a particular service.

Before the Liquid deployment, the usual procedure was to analyze all logs after they were stored in the DFS, i.e. a batch job constructed a *call graph* hours after an incident was logged. Liquid enabled to move such processing earlier in the pipeline, reducing latency and identifying potential problems within seconds rather than hours. Other organisations use purpose-built systems for this task, such as Dapper [33] at Google, or Zipkin [37] at Twitter.

At LinkedIn, the call graph assembly is an application running on top of Liquid. Liquid records each event produced by the REST calls and stores them in the messaging layer with a unique *id* per user call, as assigned by the front-end system, i.e. all REST calls for a given request share the same *id*. The processing layer processes these events to assemble the call graph. The call graph is used in production to monitor the site in real-time, and to inform capacity planning decisions.

Operational analysis. Analyzing operational data, such as metrics, alerts and logs, is crucial to react to potential problems quickly, avoiding further damage. Not only malfunctioning software or physical machines but also fraud attempts require prompt action. The volume of data grows with the number of monitoring metrics and logs, and increases due to new features and hardware resources. Previously all this data was stored in the DFS, which meant that it was retrieved and analyzed only after a problem was detected.

At LinkedIn, an internal service presents a range of business, operational and user metrics as visualizations that help different teams understand the current infrastructure status. With Liquid, integrating new data, such as crash reports from mobile phones, is straightforward: all data is transported by the messaging layer, which only needs to produce a new metric. The processing layer helps prepare data for visualizations and provides aggregate values to facilitate analysis.

6. RELATED WORK

Data warehouses have been used traditionally to maintain data in an organization under a single global schema that can be queried by applications. With the rise of big data, systems such as HBase [15] and Hive [34] focus on scale. None of these solutions, however, are suitable for nearline processing [13] because they all store data in a traditional DFS. More recently proposed systems exploit the memory of a cluster to speed up query processing [40], and optimize access to the DFS [18]. While such optimizations improve performance, they do not address issues such as the complexity of ETL, end-to-end latency or a large hardware footprint.

Stream data warehouses build queriable materialized views from continuous data streams. Systems such as DataDepot [11] and Active Data Warehouse [27] use a stream processing model to update a data warehouse in real time. Instead of running costly ETL jobs that process and load data in batches, updating the warehouse on-the-fly leads to lower end-to-end latency. These approaches are orthogonal to our solution because our focus is to provide an architecture in which new back-end applications that have varied processing and data requirements, including stream warehouses, can be implemented.

Enterprise information integration (EII) systems [41, 31] promise a unified view of data across an organization. These solutions avoid a single data warehouse and its associated costly ETL pipelines for data collection. However, EII systems are typically used to propagate data within an organization, and not to ingest high-volume data generated externally. Furthermore, they suffer from a number of problems that have prevented their wide adoption, such as their reliance on structured data [12]. In contrast, Liquid is designed to operate on unstructured data and transform it as required by back-end applications.

Stream processing systems such as Apache Flume [9] and Puma/Ptail [2] are used to collect and aggregate large amounts of log data. They are similar in spirit to Liquid's messaging layer, but they do not focus on high availability, which make them unsuitable for modern organizations.

Systems such as Storm [36] are general stream processing platforms that execute distributed dataflow graphs, but they do not support stateful processing. In contrast, Liquid's processing layer implements a stateful stream processing model, as realized by systems such as SEEP [3, 4], in order to support richer computation over continuous data.

Incremental processing systems. The incremental processing of continuously-changing data has received attention in both industry [14, 26] and academia [28, 1]. Approaches are typically based on either memoization techniques from programming languages or on maintaining state implicitly in the system. Liquid is not restricted to a particular approach, but rather focuses on the infrastructure needs for incremental processing, in particular support for explicit state.

7. CONCLUSIONS

Current data integration stacks based on distributed file systems fall short of the needs of modern data-intensive organizations. They cannot satisfy the diverse latency and re-processing requirements of different back-end data systems. As the advantages of analyzing high volume data become evident, the need for fresh results has grown. The default solution is to fit new nearline systems into the current infrastructures of organizations—yet still relying on offline data integration stacks, resulting in ad-hoc solutions.

In this paper, we have presented Liquid, a nearline big data integration stack that permits organizations to support new nearline back-end data processing applications. We showed the benefits of a stateful stream processing model on top of a highly-available messaging layer, and reported our experience of using Liquid at LinkedIn, where it transfers terabytes of data daily across different back-end systems with low latency.

8. REFERENCES

[1] P. Bhatotia, A. Wieder, et al. Incoop: MapReduce for Incremental Computations. In *SOCC*, 2011.

[2] D. Borthakur and J. S. Sarma. Apache Hadoop Goes Realtime at Facebook. In *SIGMOD*, 2011.

[3] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management. In *SIGMOD*, 2013.

[4] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Making State Explicit for Imperative Big Data Processing. In *USENIX ATC*, 2014.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *CACM*, 2008.

[6] J. DeBrabant, A. Pavlo, et al. Anti-Caching: A New Approach to Database Management System Architectures. In *VLDB*, 2013.

[7] K. Elmeleegy, C. Olston, et al. SpongeFiles: Mitigating Data Skew in Mapreduce using Distributed Memory. In *SIGMOD*, 2014.

[8] P. T. E. P. A. Felber et al. The Many Faces of Publish/Subscribe. In *ACM Surveys*, 2003.

[9] Apache Flume. `http://flume.apache.org`, 2014.

[10] S. Ghemawat, H. Gobioff, et al. The Google File System. In *SOSP*, 2003.

[11] L. Golab, T. Johnson, et al. Stream Warehousing with DataDepot. In *SIGMOD*, 2009.

[12] A. Halevy, N. Ashish, et al. Enterprise Information Integration: Successes, Challenges and Controversies. In *SIGMOD*, 2005.

[13] T. Harter, D. Borthakur, et al. Analysis of HDFS under HBase: a Facebook Messages Case Study. In *FAST*, 2014.

[14] M. Hayes and S. Shah. Hourglass: a Library for Incremental Processing on Hadoop. In *IEEE BigData*, 2013.

[15] Apache HBase. `http://hbase.apache.org`, 2014.

[16] B. Hindman, A. Konwinski, et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[17] P. Hunt, M. Konar, et al. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX ATC*, 2010.

[18] Impala. `http://impala.io`, 2014.

[19] Apache Kafka. `http://kafka.apache.org`, 2014.

[20] Kappa architecture. `http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html`.

[21] Amazon Web Services Lambda. `http://aws.amazon.com/lambda/`, 2014.

[22] J. Lin and D. Ryaboy. Scaling Big Data Mining Infrastructure: the Twitter Experience. In *SIGKDD*, 2013.

[23] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2014.

[24] D. Ongaro, S. M. Rumble, et al. Fast Crash Recovery in RAMcloud. In *SOSP*, 2011.

[25] F. Özcan, D. Hoa, et al. Emerging Trends in the Enterprise Data Analytics: Connecting Hadoop and DB2 Warehouse. In *SIGMOD*, 2011.

[26] D. Peng and F. Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *OSDI*, 2010.

[27] N. Polyzotis, S. Skiadopoulos, et al. Supporting Streaming Updates in an Active Data Warehouse. In *ICDE*, 2007.

[28] L. Popa, M. Budiu, et al. DryadInc: Reusing Work in Large-scale Computations. In *HotCloud*, 2009.

[29] RocksDB. `http://rocksdb.org`, 2014.

[30] Apache Samza. `http://samza.incubator.apache.org`, 2014.

[31] L. Seligman, P. Mork, et al. OpenII: an Open Source Information Integration Toolkit. In *SIGMOD*, 2010.

[32] Shvachko K and Hairon Kuang and others. The hadoop distributed file system. In *MSST*, 2010.

[33] B. H. Sigelman, L. A. Barroso, et al. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. In *Google Technical Report*, 2010.

[34] A. Thusoo, J. S. Sarma, et al. Hive - A Warehousing Solution Over a Map-Reduce Framework. In *VLDB*, 2009.

[35] A. Thusoo, Z. Shao, et al. Data Warehousing and Analytics Infrastructure at Facebook. In *SIGMOD*, 2010.

[36] A. Toshniwal, S. Taneja, et al. Storm@Twitter. In *SIGMOD*, 2014.

[37] Zipkin: A Distributed Tracing System. `http://twitter.github.io/zipkin/`, 2014.

[38] V. K. Vavilapalli, A. C. Murthy, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SOCC*, 2013.

[39] K. V. Vishwanath and N. Nagappan. Characterizing Cloud Computing Hardware Reliability. In *SOCC*, 2010.

[40] R. S. Xin, J. Rosen, et al. Shark: SQL and Rich Analytics at Scale. In *SIGMOD*, 2013.

[41] L. D. Xu. Enterprise Systems: State-of-the-Art and Future Trends. In *IEEE Trans. on Industrial Informatics*, 2011.

[42] M. Zaharia, M. Chowdhury, et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.

[43] M. Zaharia, A. Konwinski, et al. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, 2008.