Building LinkedIn's Real-time Activity Data Pipeline

Ken Goodhope, Joel Koshy, Jay Kreps, Neha Narkhede,
Richard Park, Jun Rao, Victor Yang Ye
LinkedIn

Abstract

One trend in the implementation of modern web systems is the use of activity data in the form of log or event messages that capture user and server activity. This data is at the heart of many internet systems in the domains of advertising, relevance, search, recommendation systems, and security, as well as continuing to fulfill its traditional role in analytics and reporting. Many of these uses place real-time demands on data feeds. Activity data is extremely high volume and real-time pipelines present new design challenges. This paper discusses the design and engineering problems we encountered in moving LinkedIn's data pipeline from a batch-oriented file aggregation mechanism to a real-time publish-subscribe system called Kafka. This pipeline currently runs in production at LinkedIn and handles more than 10 billion message writes each day with a sustained peak of over 172,000 messages per second. Kafka supports dozens of subscribing systems and delivers more than 55 billion messages to these consumer processing each day. We discuss the origins of this systems, missteps on the path to real-time, and the design and engineering problems we encountered along the way.

1 Introduction

Activity data is one of the newer ingredients in internet systems. At LinkedIn this kind of data feeds into virtually all parts of our product. We show our users information about who has viewed their profile and searching for them; we train machine learning models against activity data to predict connections, match jobs, and optimize ad display; we show similar profiles, jobs, and companies based on click co-occurance to help aid content discovery; we also populate an activity-driven newsfeed of relevant occurrences in a user's social network. Likewise, this data helps keep the web site running smoothly: streams of activity data form the basis of many of our security measures to prevent fraud and abuse as well as being the data source for various real-time site monitoring tools. These uses are in addition to the more traditional role of this kind of data in our data warehouse and Hadoop environments for batch processing jobs, reporting, and ad hoc analysis. There are two critical differences in the use of this data at web companies in comparison to traditional enterprises. The first is that activity data is often needed in real-time for monitoring, security, and user-facing product features. The second is that these data feeds are not simply used for reporting, they are a dependency for many parts of the website itself and as such require more robust infrastructure than might be needed to support simple log analysis. We refer to this real-time feed of messages as our activity data pipeline. We describe the motivations and design of a data pipeline built around Apache Kafka [11], a piece of infrastructure we designed for this usage and subsequently released as an open source project with the Apache Software Foundation.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

1.1 Previous Systems

We will describe some details of our previous generation of infrastructure for these domains to help motivate our design. We believe they are fairly representative of industry practices. We began, several years ago, with two separate systems: a batch-oriented system that handled *user* activity data, designed purely to serve the needs of loading data into a data warehouse, and a second system that handled *server* metrics and logging but provided this feed only to our monitoring system. All of these systems were effectively point-to-point data pipelines that delivered data to a single destination with no integration between.

Our user activity tracking system consisted of a simple HTTP logging service to which applications directly published small batches of xml messages; these messages were written to aggregate files and then copied to ETL servers where they were parsed, and loaded into our relational data warehouse and Hadoop [6] clusters. The problems with this service included the lack of real-time data access, and the limitation that only a single destination for data was supported (though many other uses for the data had since arisen). In addition, because there was a large diversity of activity data types captured, and the xml schemas were determined by the application emitting the data, schemas would often change unexpectedly, breaking downstream processing. Reasoning about or testing for compatibility was difficult because of the very asynchronous nature of this data consumption, making it hard to assess what code might break due to any given change. The use of XML was itself problematic and dated from the early days of the company. Custom parsing work had to be done for each xml type to make the data available and this parsing proved very computationally expensive and difficult to maintain. The fragility and complexity of this pipeline lead to inevitable delays in adding new types of activity data, which resulted in shoe-horning new activities into inappropriate existing types to avoid manual work, or worse, not capturing activities at all.

Monitoring data was not handled by this system due to its hourly batch-oriented nature, and was instead scraped using JMX hooks in applications and other system tools and made available in Zenoss, an open source monitoring tool. At LinkedIn, monitoring data consists of application counters that track various system statistics as well as structured logs. This includes fairly sophisticated request tracing capabilities in the service layer modeled after Google's Dapper system [14] to allow correlating requests across a graph of service calls. Unfortunately adding and maintaining metrics in this system was a cumbersome manual process and the data was not available elsewhere.

We had several goals in replacing these systems. The first was to be able to share data easily between these different types of feeds. For a large-scale consumer website, operational system metrics about server and service performance *are* business metrics. They are of concern to everyone from the individual engineers and operations staff to our executive team. Artificially separating these from more traditional business metrics lead to difficulties correlating problems that appeared at the system level with impacts in business metrics. Many types of system problems and bugs go undetected in system metrics but have negative business impact (e.g. a decrease in page views, sign-up rate, or other activity). Since our business metrics were not available in real-time we were dependent on a data warehouse ETL pipeline that ran with several hours of delay to detect many types of problems. This lag lead to long detection times for certain problems as well as ever-increasing pressure on the latency of the data warehouse ETL processes. Likewise, because our operational metrics were only captured in a real-time monitoring system they were not easily accessible for deeper longitudinal queries for capacity analysis or system debugging. Both systems suffered due to the sheer number of feeds that were maintained by hand, and both had similar backlogs of missing data that needed to be added by the central teams responsible for the systems. One goal was to ensure that any system—Hadoop, monitoring tools, etc—could integrate with the data pipeline and produce or consume any of these data sources with minimal additional integration work.

We had a broader motivation based in our conception of the role of the data warehouse for an internet company such as LinkedIn. The traditional role of the data warehouse is to curate and maintain a cleansed, structured copy of all data. Access to clean, complete data is of the utmost importance to a data-centric company, however having the data warehouse be the sole location of the cleansed and structured data is problematic. Our

data warehouse infrastructure consists of both a traditional relational system as well as a set of large Hadoop clusters. These are inherently batch-oriented systems that process data on an hourly or daily cycle. Making this the only location where clean data is uniformly available is problematic because it limits access to clean data to batch-oriented infrastructure. Modern websites are in large part real-time or near-real-time systems and have significant need to populate data in systems with low latency. We wanted to move as much of the structure and cleanliness "up-stream" into the real-time feed and allow batch systems to inherit this structure as one among many consumers. This helps to scale the organizational problem of integrating all data by moving many problems off the central team that owns Hadoop or the data warehouse infrastructure and bringing it to the owner of the relevant data sources directly.

1.2 Struggles with real-time infrastructure

We made several exploratory attempts to support real-time consumption with the existing pipeline infrastructure. We began by experimenting with off-the-shelf messaging products. Our tests with these systems showed poor performance when any significant backlog of persistent data accumulated, making them unsuitable for slower batch consumption. However, we envisioned them running as a real time feed in front of our central logging service as a method of providing real-time access to that data stream, with the existing batch file consumption remaining untouched. We selected ActiveMQ as a potential system for the message queue. It did well in our real-time performance tests, being capable of processing several thousand messages per second, and was already being used successfully in other LinkedIn systems for more traditional queuing workloads. We released a quick prototype built on ActiveMQ to test the system under full production load without any products or systems depending on it other than a test consumer and the tracking system that produced log file output. In our production tests we ran into several significant problems, some due to our approach to distributing load, and some due to problems in ActiveMQ itself. We understood that if the queue backed up beyond what could be kept in memory, performance would severely degrade due to heavy amounts of random I/O. We hoped we could keep the pipeline caught up to avoid this, but we found that this was not easy. Consumer processes balanced themselves at random over ActiveMQ instances so there were short periods where a given queue had no consumer processes due to this randomization. We did not want to maintain static assignment of consumer to brokers as this would be unmanageable, and downtime in a consumer would then cause backup. We also encountered a number of bugs that came out under production load that caused brokers to hang, leak connections, or run out of memory. Worse, the back-pressure mechanism would cause a hung broker to effectively block the client, which in some cases could block the web server from serving the pages for which the data was sent. The "virtual topic" feature we used to support clustered consumers required duplicating the data for each consumer in a separate queue. This leads to some inefficiencies since all topics would have at least two copies of the data, one for the real-time consumer and one for the batch log aggregation service. Further we had difficulties with ActiveMQ's built in persistence mechanism that lead to very long restart times. We believe some of these issues have since been addressed in subsequent ActiveMQ releases, but the fundamental limitations remain. We have included more detailed comparative benchmarks in a prior publication [11].

These problems motivated us to change direction. Messaging systems seem to target low-latency settings rather than the high-volume scale-out deployment that we required. It would be possible to provide enough buffer to keep the ActiveMQ brokers above water but according to our analysis this would have required several hundred servers to process a subset of activity data even if we omitted all operational data. In addition this did not help improve the batch data pipeline, adding to the overall complexity of the systems we would need to run.

As a result of this we decided to replace both systems with a piece of custom infrastructure meant to provide efficient persistence, handle long consumer backlogs and batch consumers, support multiple consumers with low overhead, and explicitly support distributed consumption while retaining the clean real-time messaging abstraction of ActiveMQ and other messaging systems. We put in place a simple in-memory relay as a stop-gap solution to support real-time product use-cases and began the development of the first version of Kafka.

2 Pipeline Infrastructure: An Overview of Apache Kafka

In this section we give a brief overview of Apache Kafka. Kafka acts as a kind of write-ahead log that records messages to a persistent store and allows subscribers to read and apply these changes to their own stores in a system appropriate time-frame. Common subscribers include live services that do message aggregation or other processing of these streams, as well as Hadoop and data warehousing pipelines which load virtually all feeds for batch-oriented processing.

2.1 Conceptual Overview of Kafka

Kafka models messages as opaque arrays of bytes. Producers send messages to a Kafka *topic* that holds a feed of all messages of that type. For example, an event that captures the occurrence of a search on the website might record details about the query, the results and ranking, and the time of the search. This message would be sent to the "searches" topic. Each topic is spread over a cluster of Kafka brokers, with each broker hosting zero or more partitions for each topic. In our usage we always have a uniform number of partitions per broker and all topics on each machine. Each partition is an ordered write-ahead log of messages. There are only two operations the system performs: one to append to the end of the log, and one to fetch messages from a given partition beginning from a particular message id. We provide access to these operations with APIs in several different programming languages. Our system is designed to handle high throughput (billions of messages) on a moderate number of topics (less than a thousand). All messages are persistent, and several tricks are used to make sure this can be done with low overhead. All topics are available for reading by any number of subscribers, and additional subscribers have very low overhead.

Kafka is a publish/subscribe system, but our notion of a subscriber is generalized to be a group of co-operating processes running as a cluster. We deliver each message in the topic to one machine in each of these subscriber groups. Consumer clusters vary in size: real-time services that consume data feeds are often made up of a few dozen servers, whereas our Hadoop clusters typically range between 100 and 500 nodes.

Both broker and consumer group membership is fully dynamic, which allows us to dynamically add nodes to the groups and have load automatically rebalanced without any static cluster configuration. We use Zookeeper [9] to manage this group membership and assign partition consumers. Each consumer process announces itself as part of a named group (e.g. "hadoop-etl") and the appearance or disappearance of such a process triggers a re-assignment of partitions to rebalance load across the new set of consumer processes. From the time of this assignment to the next reassignment, each process is the only member of the group allowed to consume from its set of partitions. As a result the partition is the unit of parallelism of the topic and controls the maximum parallelism of the subscriber. This arrangement allows the metadata concerning what has been consumed from a particular partition by a given consumer group to be a simple eight-byte monotonically increasing message id recording the last message in that partition that has been acknowledged.

Because it is the sole consumer of the partition within that group, the consuming process can lazily record its own position, rather than marking each message immediately. If the process crashes before the position is recorded it will just reprocess a small number of messages, giving "at-least-once" delivery semantics. This ability to lazily record which messages have been consumed is essential for performance. Systems which cannot elect owners for partitions in this fashion must either statically assign consumers with only a single consumer per broker or else randomly balance consumers. Random assignment means multiple consumers consume the same partition at the same time, and this requires a mechanism to provide mutual exclusion to ensure each messages goes to only one consumer. Most messaging systems implement this by maintaining per-message state to record message acknowledgement. This incurs a great deal of random I/O in the systems that we benchmarked to update the per-message state. Furthermore this model generally eliminates any ordering as messages are distributed at random over the consumers which process in a non-deterministic order.

Message producers balance load over brokers and sub-partitions either at random or using some application-

supplied key to hash messages over broker partitions. This key-based partitioning has two uses. First the delivery of data within a Kafka partition is ordered but no guarantee of order is given between partitions. Consequently, to avoid requiring a global order over messages, feeds that have a requirement for ordering need to be partitioned by some key within which the ordering will be maintained. The second use is to allow in-process aggregation for distributed consumers. For example, consumers can partition a data stream by user id, and perform simple in-memory session analysis distributed across multiple processes relying on the assumption that all activity for a particular user will be sticky to one of those consumer process. Without this guarantee distributed message processors would be forced to materialize all aggregate state into a shared storage system, likely incurring an expensive look-up round-trip per message.

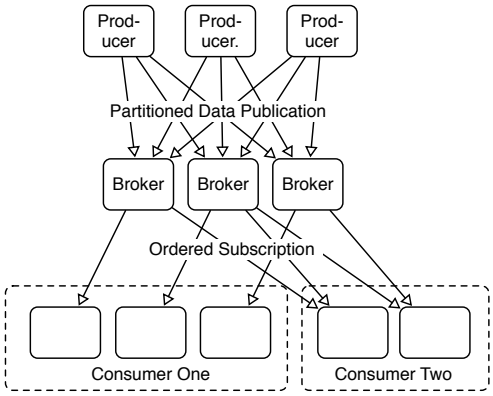

Figure 1: The physical layout of a Kafka cluster. This figure depicts several producer processes sending data to a single multi-broker Kafka cluster with two clustered subscribers.

In our current deployment Kafka doesn't attempt to provide guarantees to prevent all possibility of data loss. A small number of messages can be lost in the event of a hard server failure before messages are transfered to the Kafka brokers or flushed to disk. Instead of attempting to provide hard guarantees, we measure data at each logical tier (producer, broker, and consumers) to measure loss or delay. We discuss this approach and the results we achieve for our production system in a subsequent section.

2.2 Kafka Usage at LinkedIn

Kafka handles over 10 billion message writes per day with a sustained load over the highest traffic hour that averages 172,000 messages per second. We see very heavy use of our multi-subscriber support, both for our own inter-data-center replication and various external applications that make use of the data. There are a total of 40 real-time data consumers that consume one or more topics, eight of which are our own replication and pipeline monitoring tools, and 32 of which are applications written by other users for other product features or tools. In total, we see a ratio of roughly 5.5 messages consumed for each message produced in our live data centers. This results in a daily total in excess of 55 billion messages delivered to real-time consumers as well as a complete copy of all data delivered to Hadoop clusters with a short delay fashion.

We support 367 topics that cover both user activity topics (such as page views, searches, and other activities) and operational data (such as log, trace, and metrics for most of LinkedIn's services). The largest of these topics adds an average of 92GB per day of batch-compressed messages, and the smallest adds only a few hundred kilobytes. In addition to these outbound topics that produce feeds from live applications, we have "derived" data feeds that come out of asynchronous processing services or out of our Hadoop clusters to deliver data back to the live serving systems.

Message data is maintained for 7 days. Log segments are garbage collected after this time period whether or not they have been consumed. Consumers can override this policy on a per-topic basis but few have seen a need to do so. In total we maintain 9.5 TB of compressed messages across all topics.

Due to careful attention to efficiency, the footprint for the Kafka pipeline is only 8 servers per data center, making it significantly more efficient than the activity logging system it replaced. Each Kafka server maintains 6TB of storage on a RAID 10 array of commodity SATA drives. Each of these servers has a fairly high connection fan-in and fan-out, with over 10,000 connections into the cluster for data production or consumption. In addition to these live consumers we have numerous large Hadoop clusters which consume infrequent, high-throughput, parallel bursts as part of the offline data load.

We handle multiple data centers as follows: producers always send data to a cluster in their local data center, and these clusters replicate to a central cluster that provides a unified view of all messages. This replication works through the normal consumer API, with the central destination cluster acting as a consumer on the source clusters in the online data centers. We see an average end-to-end latency through this pipeline of 10 seconds and a worst case of 32 seconds from producer event generation to a live consumer in the offline data center. This delay comes largely from throughput-enhancing batching optimizations that will be described in the next section. Disabling these in performance experiments gives end-to-end round trips of only a few milliseconds but incurs lots of small random I/O operations.

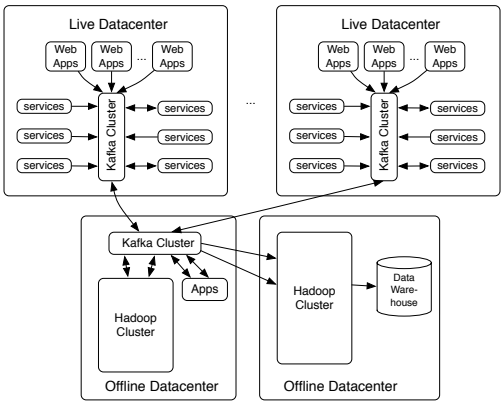

Figure 2: A simplified view of how the data pipeline fits into LinkedIn's data center topology. Our configuration always sends live traffic to a local Kafka cluster and these live clusters are replicated to an aggregate cluster which contains a full view of all data for processing or loading into Hadoop. Data is also sent back from these applications and from Hadoop processing as new Kafka streams. The "live data centers" are where the real-time application serving happens and the "offline data centers" house Hadoop and other offline analytical infrastructure.

3 Engineering for High Throughput

The primary design criteria for a pipeline such as this is maximizing the end-to-end throughput while providing low-latency and reasonable semantics to the system users. It is also important that throughput and latency remain constant with respect to the volume of unconsumed data as we often have slow or batch-oriented consumers. We will describe some of the optimizations that are essential for achieving high and consistent throughput.

High-throughput is one advantage of the design of traditional log aggregation systems over most messaging systems. Data is written to a set of text logs with no immediate flush to disk, allowing very efficient I/O patterns. Unfortunately there are a number of drawbacks. Simple log files provide very poor semantics with respect

to message loss and make low-latency incremental consumption challenging. Partial messages can occur in the event of a crash and can not automatically be corrected if the system doesn't understand the internal log format, resulting in corrupt data in the feed. The unit of consumption is usually the file as there is no logical identifier at the message level. This introduces the log file (which should be an implementation detail) as a first-class concept; clients need to remember which files have been processed and restart correctly in the event of a failure. Likewise, the file must be complete before it can be aggregated for processing, which introduces artificial latency. This latency can be mitigated by rolling over files more often (for example, some companies use "minute files") but this comes at a cost of exploding the number of files maintained. One of our design goals was to maintain the simple I/O model of logging systems while providing the cleaner, more granular model of messaging systems.

Kafka uses three primary techniques to improve the effective throughput. The first is partitioning up data so that production, brokering, and consumption of data are all handled by clusters of machines that can be scaled incrementally as load increases. The second is to batch messages together to send larger chunks at once. The third is shrinking the data to make the total data sent smaller. These techniques are well known, but not always used in messaging systems we tested. We will show how these techniques can be implemented end-to-end throughout the system.

3.1 Batching

Kafka bridges the gap between the online request-driven world, which supplies small messages one at a time as events occur, and the near-real-time asynchronous systems or the batch processing systems that feed off this data. Directly processing these small messages as they occur in a request-driven fashion leads to small network and disk operations which in turn severely limit throughput. This trade-off between latency and throughput is well known [12] and exists in some form in many disk and network I/O systems. The fundamental technique we use is to incur a small amount of latency to group small messages together and improve throughput. We implement this in a way that is tunable at the application level. We performed a series of simple experiments on a configuration that matches our production hardware to demonstrate the effect of individual batching optimizations over unbatched operation.

The Kafka producer client can be configured to send messages in either a synchronous or asynchronous fashion. The async mode allows the client to batch small random messages into larger data chunks before sending it over the network. TCP implementations generally support a similar trick for batching using Nagle's algorithm, but we do this at the application level. This allows us to implement a precise timeout and message count threshold that triggers the send, guaranteeing that we never batch more than a given number of messages and that no message is held for more than a configurable number of milliseconds. This precise control is important since messages sent asynchronously are at risk of loss if the application crashes. Originally we performed this batching only on a per-topic basis, however performance analysis showed it to be very effective and there are many low-volume topics, so we reworked our APIs to support a "multisend" operation. This allows us to group messages for low-volume topics into batches with the high-volume topics and further reduce the number of small requests. In our tests allowing the producer to batch messages into groups of 200 improves throughput by a factor of 3.2.

The broker does not do further batching of filesystem writes as we found the cost of a buffered write to be extremely low, especially after the client side batching has taken place. Instead we simply rely on the filesystem pagecache to buffer writes and delay the flush to disk to batch the expensive physical disk writes. This ability to delay physical write activity allows the operating system I/O scheduler to coalesce small random appends to hundreds of topics into a larger linear write pattern. Putting all data immediately into the filesystem has the benefit that no data is lost during application crashes that leave the operating system intact. This flush batching is also governed by a configurable policy that is triggered by an elapsed time or the number of accumulated messages in that partition. A similar delayed flush policy is implemented in Linux's "pdflush" daemon [10]; but

again doing this in the application allows us to explicitly respect message boundaries and have different policy configurations for different topics including an option for synchronous flushing of writes for important topics. Without this technique, our strategy of always persisting messages would be extremely inefficient: we see an improvement of 51.7x over immediately flushing each write due to the large number of topics and partitions and resulting random I/O incurred.

Consumer data fetch requests are also effectively batched. Data consumption occurs with the client requesting messages from the broker by specifying a given starting message id and some user-specified maximum buffer size (say one megabyte) to fill with message data. This batching improves performance in our tests by a factor of 20.1. As with the send API, this fetch API is a "multifetch" allowing the user to pull data from many topics and partitions in a single request to avoid small requests on low-volume topics as messages trickle in.

In combination these batching techniques allow us to give end-to-end throughput that matches the I/O capacity of the underlying disk sub-system, 50-100 MB/sec for an array of commodity disks. There is little or no negative impact from small messages or large backlogs of unconsumed data. We give a more detailed comparison to other messaging systems in previous work [11].

Batching Type	Improvement	Default Threshold
Producer	3.2x	200 messages or 30 seconds
Broker	51.7x	50000 messages or 30 seconds
Consumer	20.1x	1 megabyte

Table 5: Improvements due to batching

3.2 Shrinking Data

Shrinking data is the most important performance technique as one of our most fundamental bottlenecks is the bandwidth between data center facilities. This bandwidth is more expensive per-byte to scale than disk I/O, CPU, or network bandwidth capacity within a facility. There are two basic approaches to shrinking data: exploiting explicit structure in message serialization and exploiting implicit structure using compression.

We extract repeated known structure from the messages by associating schemas with the topics that capture the repeated field names and type information used for efficient storage. Schemas have other benefits in ensuring compatibility and allowing integration which we will discuss in a later section. Though Kafka itself is serialization-agnostic, LinkedIn has standardized on the use of Apache Avro [4] as the schema and serialization language for all of its activity data messages as well as data in Hadoop and most other data systems. In uncompressed form our Avro data was roughly 7 times smaller than XML messages in the activity logging system we replaced.

One advantage of traditional file-logging is that it naturally allows compressing an entire log file as a single stream, whereas no messaging system we are aware of provides this kind of batch compression of messages. Most messages are small and once they are encoded in a compact serialization format have little redundancy on their own. Kafka allows the batch compression of multiple messages into a single composite message set to allow effective compression. We find that this batch compression of a groups of a few hundred messages is comparable to full file compression in compression ratio and allows us to use compression end-to-end from the message producer to the destination consumer. Compression is done as part of the batching in the producer API and compressed message sets containing a few hundred messages are sent to Kafka where they are retained and served in compressed form. Kafka supports a variety of compression algorithms, but at LinkedIn we use standard GZIP as it provides tight compression. We see a compression ratio of roughly 3x on our Avro data sets. Despite the CPU intensive nature of GZIP compression we actually see a 30% improvement in throughput in our tests, presumably due to the reduced network and filesystem I/O.

3.3 Reliance on Pagecache

Kafka does not attempt to make any use of in-process caching and instead relies on persistent data structures and OS page cache. We have found this to be an effective strategy. In Linux (which we use for our production environment) page cache effectively uses all free memory, which means the Kafka servers have roughly 20GB of cache available without any danger of swapping or garbage collection problems. Using this cache directly instead of an in-process cache avoids double-buffering data in both the OS and in process cache. The operating system's read-ahead strategy is very effective for optimizing the linear read pattern of our consumers which sequentially consume chunks of log files. The buffering of writes naturally populates this cache when a message is added to the log, and this in combination with the fact that most consumers are no more than 10 minutes behind, means that we see a 99% pagecache hit ratio on our production servers for read activity, making reads nearly free in terms of disk I/O.

The log file itself is written in a persistent, checksummed, and versioned format that can be directly included when responding to client fetch requests. As a result we are able to take advantage of Linux's sendfile system call [10]. This allows the transmission of file chunks directly from pagecache with no buffering or copying back and forth between user and kernel space to perform the network transfer.

These three features—the high cache hit ratio of reads, the sendfile optimization, and the low bookkeeping overhead of the consumers—make adding consumers exceptionally low impact.

4 Handling Diverse Data

Some of the most difficult problems we have encountered have to do with the variety of data, use cases, users, and environments that a centralized data pipeline must support. These problems tend to be the least glamorous and hardest to design for ahead of time because they can't be predicted by extrapolating forward current usage. They are also among the most important as efficiency problems can often be resolved by simply adding machines, while "integration" problems may fundamentally limit the usefulness of the system. Our pipeline is used in a direct way by several hundred engineers in all LinkedIn product areas and we handle topics that cover both business metrics, application monitoring and "logging", as well as derived feeds of data that come out of the Hadoop environment or from other asynchronous processing applications.

One of the biggest problems we faced in the previous system was managing hundreds of evolving XML formats for data feeds without breaking compatibility with consumers depending on the data feeds. The root cause of the problem was that the format of the data generated by applications was controlled by the application generating the data. However testing or even being aware of all processing by downstream applications for a given data feed was a difficult undertaking. Since XML did not map well to the data model of either our Hadoop processing or live consumers, this meant custom parsing and mapping logic for each data consumer. As a result loading new data in systems was time consuming and error prone. These load jobs often failed and were difficult to test with upcoming changes prior to production release.

This problem was fixed by moving to Avro and maintaining a uniform schema tied to each topic that can be carried with the data throughout the pipeline and into the offline world. Those responsible for the application, those responsible for various data processing tasks, and those responsible for data analysis, could all discuss the exact form of the data prior to release. This review process was mandatory and built into the automatic code review process schema addition and changes. This also helped to ensure that the meaning of the thousands of fields in these events were precisely documented in documentation fields included with the schema. We have a service which maintains a history of all schema versions ever associated with a given topic and each message carries an id that refers to exact schema version with which it was written. This means it is effectively impossible for schema changes to break the deserialization of messages, as messages are always read with precisely the schema they were written.

This system was an improvement over XML, however during the process of rolling out the system we found that even these improvements were not sufficient in fully transitioning ownership of data format off a centralized team or in preventing unexpected incompatibilities with data consumers. Although the messages were now individually understandable, it was possible to make backwards incompatible changes that broke consumer code that used the data (say, by removing needed fields or changing its type in an incompatible way). Many of these incompatible changes can easily be made in the application code that generates the data but have implications for the existing retained data. Historical data sets are often quite large (hundreds of TBs), and updating previous data to match a new schema across a number of Hadoop clusters can be a major undertaking. Hadoop tools expect a single schema that describes a data set—for example Hive [7], Pig [8], and other data query tools allow flexible data format but they assume a static set of typed columns so they do not cope well with an evolving data model. To address these issues we developed an exact compatibility model to allow us to programmatically check schema changes for backwards compatibility against existing production schemas. This model only allows changes that maintain compatibility with historical data. For example, new fields can be added, but must have a default value to be used for older data that does not contain the field. This check is done when the schema is registered with the schema registry service and results in a fatal error if it fails. We also do a proactive check at compile time and in various integration environments using the current production schema to detect incompatibilities as early as possible.

5 Hadoop Data Loads

Making data available in Hadoop is of particular importance to us. LinkedIn uses Hadoop for both batch data processing for user-facing features as well as for more traditional data analysis and reporting. Previous experience had lead us to believe that any custom data load work that had to be done on a per-topic basis would become a bottleneck in making data available, even if it were only simple configuration changes. Our strategy to address this was to push the schema management and definition out of Hadoop and into the upstream pipeline system and make the data load into Hadoop a fully automated projection of the contents of this pipeline with no customization on a per-event basis. This means that there is a single process that loads all data feeds into Hadoop, handles data partitioning, and creates and maintains Hive tables to match the most recent schema. The data pipeline and the Hadoop tooling were designed together to ensure this was possible. This ability is heavily dependent on Hadoop's flexible data model to allow us to carry through a data model from the pipeline to Hadoop and integrate with a rich collection of query tools without incurring non-trivial mapping work for each usage.

To support this we have built custom Avro plugins for Pig and Hive to allow them to work directly with Avro data and complement the built-in Avro support for Java MapReduce. We have contributed these back to the open source community. This allows all these common tools to share a single authoritative source of data and allow the output of a processing step built with one technology to seamlessly integrate with other tools.

The load job requires no customization or configuration on a per-topic basis whatsoever. When a new topic is created we automatically detect this, load the data, and use the Avro schema for that data to create an appropriate Hive table. Schema changes are detected and handled in a similar fully-automatic manner. This kind of automated processing is not uncommon in Hadoop, but it is usually accomplished by maintaining data in an unstructured text form that pushes all parsing and structure to query time. This simplifies the load process, but maintaining parsing that is spread across countless MapReduce jobs, pig scripts, and queries as data models evolve is a severe maintenance problem (and likely a performance problem as well). Our approach is different in that our data is fully structured and typed, but we inherit this structure from the upstream system automatically. This means that users never need to declare any data schemas or do any parsing to use our common tools on data in Hadoop: the mapping between Avro's data model and Pig and Hive's data model is handled transparently.

We run Hadoop data loads as a MapReduce job with no reduce phase. We have created a custom Kafka

InputFormat to allow a Kafka cluster to act as a stand in for HDFS, providing the input data for the job. This job is run on a ten minute interval and takes on average two minutes to complete the data load for that interval. At startup time the job reads its current offset for each partition from a file in HDFS and queries Kafka to discover any new topics and read the current log offset for each partition. It then loads all data from the last load offset to the current Kafka offset and writes it out to Hadoop, project all messages forward to the current latest schema.

Hadoop has various limitations that must be taken into consideration in the design of the job. Early versions of our load process used a single task per Kafka topic or partition. However we found that in practice we have a large number of low-volume topics and this lead to bottlenecks in the number of jobs or tasks many of which have only small amounts of processing to do. Instead, we found that we needed to intelligently balance Kafka topic partitions over a fixed pool of tasks to ensure an even distribution of work with limited start-up and tear-down costs.

This approach to data load has proven very effective. A single engineer was able to implement and maintain the process that does data loads for all topics; no incremental work or co-ordination is needed as teams add new topics or change schemas.

6 Operational Considerations

One of the most important considerations in bringing a system to production is the need to fully monitor its correctness. This kind of monitoring is at least as important as traditional QA procedures and goes well beyond tracking simple system-level stats. In practice, we have found that there are virtually no systems which cannot tolerate some data loss, transient error, or other misfortune, provided it is sufficiently rare, has bounded impact, and is fully measured.

One deficiency of previous generations of systems at LinkedIn was that there was no such measurement in place, and indeed it is inherently a hard thing to do. This meant that although certain errors were seen in the old system, there was no way to assert that the final number of messages received was close to the right number, and no way to know if the errors seen were the only errors or just the ones that were visible.

To address this we added deep monitoring capabilities, in addition to the normal application metrics we would capture for any system. We built a complete end-to-end audit trail that attempts to detect any data loss. This is implemented as a special Kafka topic that holds audit messages recording the number of messages for a given topic in a given time period. Each logical tier—a producer or consumer in some data center—send a periodic report of the number of messages it has processed for each topic in a given ten minute time window. The sum of the counts for each topic should be the same for all tiers for a ten-minute window if no data is lost. The count is always with respect to the time stamp in the message not the time on the machine at the time of processing to ensure the consumption of delayed messages count towards the completeness of the time bucket in which they originated not the current time bucket. This allows exact reconciliation even in the face of delays. We also use this system to measure the time it takes to achieve completeness in each tier as a percentage of the data the producer sent. We phrase our SLA as the time to reach 99.9% completeness at each tier below the producer. This audit monitoring measures each step the data takes throughout the pipeline—across data centers, to each consumer, and into Hadoop—to ensure data is properly delivered in a complete and timely fashion everywhere. We have a standalone monitoring application that consumes the audit data topic and performs an exact reconciliation between all tiers to compute loss or duplication rates as well as to perform graphing and alerting on top of this data.

This detailed audit helped discover numerous errors including bugs, misconfigurations, and dozens of corner cases that would otherwise have gone undetected. This general methodology of building a self-proving verification system into the system (especially one that relies on the fewest possible assumptions about the correctness of the system code) is something we found invaluable and intend to replicate as part of future systems.

We are able to run this pipeline with end-to-end correctness of 99.9999%. The overwhelming majority of

data loss comes from the producer batching where hard kills or crashes of the application process can lead to dropping unsent messages.

7 Related Work

Logging systems are a major concern for web sites, and similar systems exist at other large web companies. Facebook has released Scribe as open source which provides similar structured logging capabilities [3] as well as described some of the other real-time infrastructure in their pipeline [2]. Other open source logging solutions such as Flume also exist [5]. Yahoo has some description of their data pipeline [1] as well as of a Hadoop specific log-aggregation system [13]. These pipelines do not seem to unify operational and business metrics, and Yahoo seems to describe separate infrastructure pieces for real-time and batch consumption. Flume and Scribe are more specifically focused on loading logs into Hadoop, though they are flexible enough to produce output to other systems. Both are based on a push rather than pull model which complicates certain real-time consumption scenarios.

8 Acknowledgements

We would like to acknowledge a few people who were helpful in bringing this system to production: David DeMaggd was responsible for the operation and rollout of LinkedIn's Kafka infrastructure, John Fung was responsible for performance and quality testing, Rajappa Iyer and others on the data warehouse team helped with data model definition and conversion of the warehouse to use this pipeline as well as helping define SLAs and monitoring requirements. In addition we are grateful for the code and suggestions from contributors in the open source community.

References

[1] Mona Ahuja, Cheng Che Chen, Ravi Gottapu, Jörg Hallmann, Waqar Hasan, Richard Johnson, Maciek Kozyrczak, Ramesh Pabbati, Neeta Pandit, Sreenivasulu Pokuri, and Krishna Uppala. Peta-scale data warehousing at yahoo! In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 855–862, New York, NY, USA, 2009. ACM.

[2] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1071–1080, New York, NY, USA, 2011. ACM.

[3] Facebook Engineering. Scribe. https://github.com/facebook/scribe.

[4] Apache Software Foundation. Avro. http://avro.apache.org.

[5] Apache Software Foundation. Flume. https://cwiki.apache.org/FLUME.

[6] Apache Software Foundation. Hadoop. http://hadoop.apache.org.

[7] Apache Software Foundation. Hive. http://hive.apache.org.

[8] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proc. VLDB Endow.*, 2(2):1414–1425, August 2009.

[9] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, 2010.

[10] Michael Kerrisk. *The Linux Programming Interface*. No Starch Press, 2010.

[11] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. *ACM SIGMOD Workshop on Networking Meets Databases*, page 6, 2011.

[12] David Patterson. Latency lags bandwidth. In *Proceedings of the 2005 International Conference on Computer Design*, ICCD '05, pages 3–6, Washington, DC, USA, 2005. IEEE Computer Society.

[13] Ariel Rabkin and Randy Katz. Chukwa: a system for reliable large-scale log collection. In *Proceedings of the 24th international conference on Large installation system administration*, LISA'10, pages 1–15, Berkeley, CA, USA, 2010. USENIX Association.

[14] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.