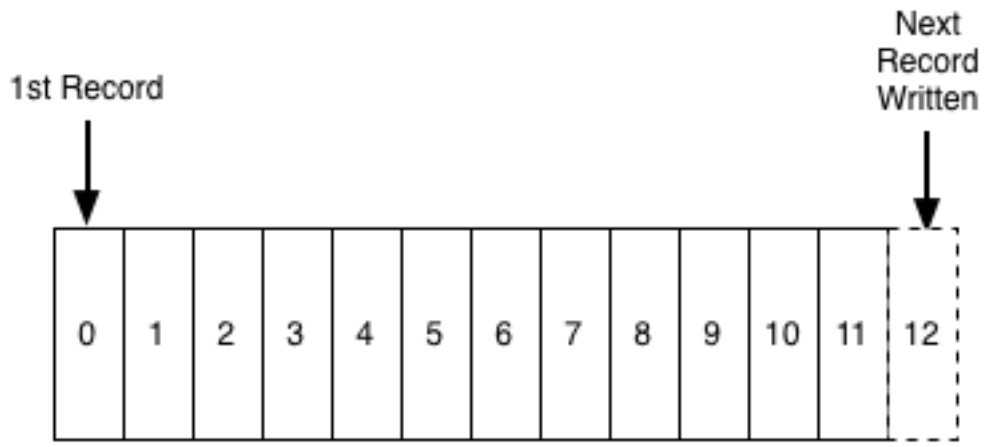
The Log: What every software engineer should know about real-time data's unifying abstraction

I joined LinkedIn about six years ago at a particularly interesting time. We were just beginning to run up against the limits of our monolithic, centralized database and needed to start the transition to a portfolio of specialized distributed systems. This has been an interesting experience: we built, deployed, and run to this day a distributed graph database, a distributed search backend, a Hadoop installation, and a first and second generation key-value store.

One of the most useful things I learned in all this was that many of the things we were building had a very simple concept at their heart: the log. Sometimes called write-ahead logs or commit logs or transaction logs, logs have been around almost as long as computers and are at the heart of many distributed data systems and real-time application architectures.
You can't fully understand databases, NoSQL stores, key value stores, replication, paxos, hadoop, version control, or almost any software system without understanding logs; and yet, most software engineers are not familiar with them. I'd like to change that. In this post, I'll walk you through everything you need to know about logs, including what is log and how to use logs for data integration, real time processing, and system building.

Part One: What Is a Log?

A log is perhaps the simplest possible storage abstraction. It is an append-only, totally-ordered sequence of records ordered by time. It looks like this:



Records are appended to the end of the log, and reads proceed left-to-right. Each entry is assigned a unique sequential log entry number.
The ordering of records defines a notion of "time" since entries to the left are defined to be older then entries to the right. The log entry number can be thought of as the "timestamp" of the entry. Describing this ordering as a notion of time seems a bit odd at first, but it has the convenient property that it is decoupled from any particular physical clock. This property will turn out to be essential as we get to distributed systems.
The contents and format of the records aren't important for the purposes of this discussion. Also, we can't just keep adding records to the log as we'll eventually run out of space. I'll come back to this in a bit.
So, a log is not all that different from a file or a table. A file is an array of bytes, a table is an array of records, and a log is really just a kind of table or file where the records are sorted by time.
At this point you might be wondering why it is worth talking about something so simple? How is a append-only sequence of records in any way related to data systems? The answer is that logs have a specific purpose: they record what happened and when. For distributed data systems this is, in many ways, the very heart of the problem.

But before we get too far let me clarify something that is a bit confusing. Every programmer is familiar with another definition of logging—the unstructured error messages or trace info an application might write out to a local file using syslog or log4j. For clarity I will call this "application logging". The application log is a degenerative form of the log concept I am describing. The biggest difference is that text logs are meant to be primarily for humans to read and the "journal" or "data logs" I'm describing are built for programmatic access.
(Actually, if you think about it, the idea of humans reading through logs on individual machines is something of an anachronism. This approach quickly becomes an unmanageable strategy when many services and servers are involved and the purpose of logs quickly becomes as an input to queries and graphs to understand behavior across many machines—something for which english text in files is not nearly as appropriate as the kind structured log described here.)
Logs in databases

I don't know where the log concept originated—probably it is one of those things like binary search that is too simple for the inventor to realize it was an invention. It is present as early as IBM's System R. The usage in databases has to do with keeping in sync the variety of data structures and indexes in the presence of crashes. To make this atomic and durable, a database uses a log to write out information about the records they will be modifying, before applying the changes to all the various data structures it maintains. The log is the record of what happened, and each table or index is a projection of this history into some useful data structure or index. Since the log is immediately persisted it is used as the authoritative source in restoring all other persistent structures in the event of a crash.
Over-time the usage of the log grew from an implementation detail of ACID to a method for replicating data between databases. It turns out that the sequence of changes that happened on the database is exactly what is needed to keep a remote replica database in sync. Oracle, MySQL, and PostgreSQL include log shipping protocols to transmit portions of log to replica databases which act as slaves. Oracle has productized the log as a general data subscription mechanism for non-oracle data subscribers with their XStreams and GoldenGate and similar facilities in MySQL and PostgreSQL are key components of many data architectures.
Because of this origin, the concept of a machine readable log has largely been confined to database internals. The use of logs as a mechanism for data subscription seems to have arisen almost by chance. But this very abstraction is ideal for supporting all kinds of messaging, data flow, and real-time data processing.
Logs in distributed systems

The two problems a log solves—ordering changes and distributing data—are even more important in distributed data systems. Agreeing upon an ordering for updates (or agreeing to disagree and coping with the side-effects) are among the core design problems for these systems.
The log-centric approach to distributed systems arises from a simple observation that I will call the State Machine Replication Principle:
If two identical, deterministic processes begin in the same state and get the same inputs in the same order, they will produce the same output and end in the same state.
This may seem a bit obtuse, so let's dive in and understand what it means.
Deterministic means that the processing isn't timing dependent and doesn't let any other "out of band" input influence its results. For example a program whose output is influenced by the particular order of execution of threads or by a call to gettimeofday or some other non-repeatable thing is generally best considered as non-deterministic.
The state of the process is whatever data remains on the machine, either in memory or on disk, at the end of the processing.
The bit about getting the same input in the same order should ring a bell—that is where the log comes in. This is a very intuitive notion: if you feed two deterministic pieces of code the same input log, they will produce the same output.
The application to distributed computing is pretty obvious. You can reduce the problem of making multiple machines all do the same thing to the problem of implementing a distributed consistent log to feed these processes input. The purpose of the log here is to squeeze all the non-determinism out of the input stream to ensure that each replica processing this input stays in sync.

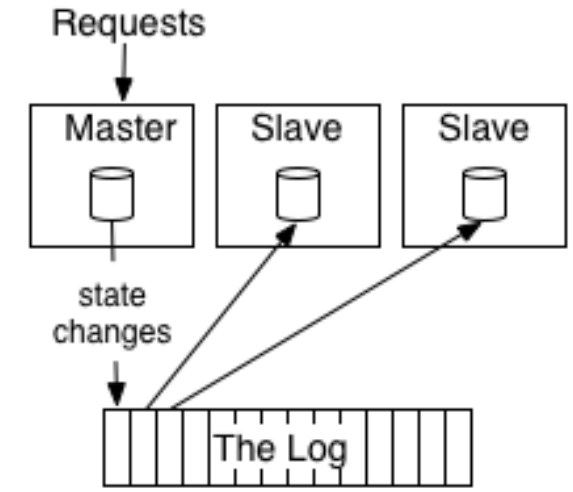
When you understand it, there is nothing complicated or deep about this principle: it more or less amounts to saying "deterministic processing is deterministic". Nonetheless, I think it is one of the more general tools for distributed systems design.

One of the beautiful things about this approach is that the time stamps that index the log now act as the clock for the state of the replicas—you can describe each replica by a single number, the timestamp for the maximum log entry it has processed. This timestamp combined with the log uniquely captures the entire state of the replica.

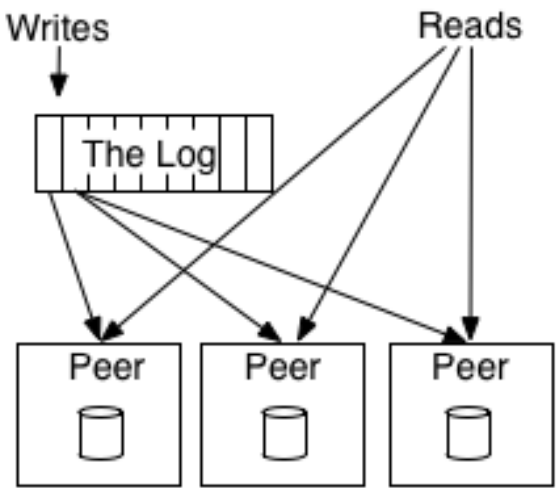
There are a multitude of ways of applying this principle in systems depending on what is put in the log. For example, we can log the incoming requests to a service, or the state changes the service undergoes in response to request, or the transformation commands it executes. Theoretically, we could even log a series of machine instructions for each replica to execute or the method name and arguments to invoke on each replica. As long as two processes process these inputs in the same way, the processes will remaining consistent across replicas.

Different groups of people seem to describe the uses of logs differently. Database people generally differentiate between physical and logical logging. Physical logging means logging the contents of each row that is changed. Logical logging means logging not the changed rows but the SQL commands that lead to the row changes (the insert, update, and delete statements).

The distributed systems literature commonly distinguishes two broad approaches to processing and replication. The "state machine model" usually refers to an active-active model where we keep a log of the incoming requests and each replica processes each request. A slight modification of this, called the "primary-backup model", is to elect one replica as the leader and allow this leader to process requests in the order they arrive and log out the changes to its state from processing the requests. The other replicas apply in order the state changes the leader makes so that they will be in sync and ready to take over as leader should the leader fail.



Active-Passive Processing
(Primary-Backup)



Active-Active Processing
(State-Machine Replication)

To understand the difference between these two approaches, let's look at a toy problem. Consider a replicated "arithmetic service" which maintains a single number as its state (initialized to zero) and applies additions and multiplications to this value. The active-active approach might log out the transformations to apply, say "+1", "*2", etc. Each replica would apply these transformations and hence go through the same set of values. The "active-passive" approach would have a single master execute the transformations and log out the result, say "1", "3", "6", etc. This example also

makes it clear why ordering is key for ensuring consistency between replicas: reordering an addition and multiplication will yield a different result.

The distributed log can be seen as the data structure which models the problem of consensus. A log, after all, represents a series of decisions on the "next" value to append. You have to squint a little to see a log in the Paxos family of algorithms, though log-building is their most common practical application. With Paxos, this is usually done using an extension of the protocol called "multi-paxos", which models the log as a series of consensus problems, one for each slot in the log. The log is much more prominent in other protocols such as ZAB, RAFT, and Viewstamped Replication, which directly model the problem of maintaining a distributed, consistent log.
My suspicion is that our view of this is a little bit biased by the path of history, perhaps due to the few decades in which the theory of distributed computing outpaced its practical application. In reality, the consensus problem is a bit too simple. Computer systems rarely need to decide a single value, they almost always handle a sequence of requests. So a log, rather than a simple single-value register, is the more natural abstraction.
Furthermore, the focus on the algorithms obscures the underlying log abstraction systems need. I suspect we will end up focusing more on the log as a commoditized building block irrespective of its implementation in the same way we often talk about a hash table without bothering to get in the details of whether we mean the murmur hash with linear probing or some other variant. The log will become something of a commoditized interface, with many algorithms and implementations competing to provide the best guarantees and optimal performance.
Changelog 101: Tables and Events are Dual

Let's come back to databases for a bit. There is a facinating duality between a log of changes and a table. The log is similar to the list of all credits and debits and bank processes; a table is all the current account balances. If you have a log of changes, you can apply these changes in order to create the table capturing the current state. This table will record the latest state for each key (as of a particular log time). There is a sense in which the log is the more fundamental data structure: in addition to creating the original table you can also transform it to create all kinds of derived tables. (And yes, table can mean keyed data store for the non-relational folks.)

This process works in reverse too: if you have a table taking updates, you can record these changes and publish a "changelog" of all the updates to the state of the table. This changelog is exactly what you need to support near-real-time replicas. So in this sense you can see tables and events as dual: tables support data at rest and logs capture change. The magic of the log is that if it is a complete log of changes, it holds not only the contents of the final version of the table, but also allows recreating all other versions that might have existed. It is, effectively, a sort of backup of every previous state of the table.
This might remind you of source code version control. There is a close relationship between source control and databases. Version control solves a very similar problem to what distributed data systems have to solve—managing distributed, concurrent changes in state. A version control system usually models the sequence of patches, which is in effect a log. You interact directly with a checked out "snapshot" of the current code which is analogous to the table. You will note that in version control systems, as in other distributed stateful systems, replication happens via the log: when you update, you pull down just the patches and apply them to your current snapshot.
Some people have seen some of these ideas recently from Datomic, a company selling a log-centric database. This presentation gives a great overview of how they have applied the idea in their system. These ideas are not unique to this system, of course, as they have been a part of the distributed systems and database literature for well over a decade.
This may all seem a little theoretical. Do not despair! We'll get to practical stuff pretty quickly.
What's next

In the remainder of this article I will try to give a flavor of what a log is good for that goes beyond the internals of distributed computing or abstract distributed computing models. This includes:
Data Integration—Making all of an organization's data easily available in all its storage and processing systems.

Real-time data processing—Computing derived data streams.
Distributed system design—How practical systems can by simplified with a log-centric design.
These uses all resolve around the idea of a log as a stand-alone service.
In each case, the usefulness of the log comes from simple function that the log provides: producing a persistent, re-playable record of history. Surprisingly, at the core of these problems is the ability to have many machines playback history at their own rate in a deterministic manner.
Part Two: Data Integration

Let me first say what I mean by "data integration" and why I think it's important, then we'll see how it relates back to logs.
Data integration is making all the data an organization has available in all its services and systems. This phrase "data integration" isn't all that common, but I don't know a better one. The more recognizable term ETL usually covers only a limited part of data integration—populating a relational data warehouse. But much of what I am describing can be thought of as ETL generalized to cover real-time systems and processing flows.

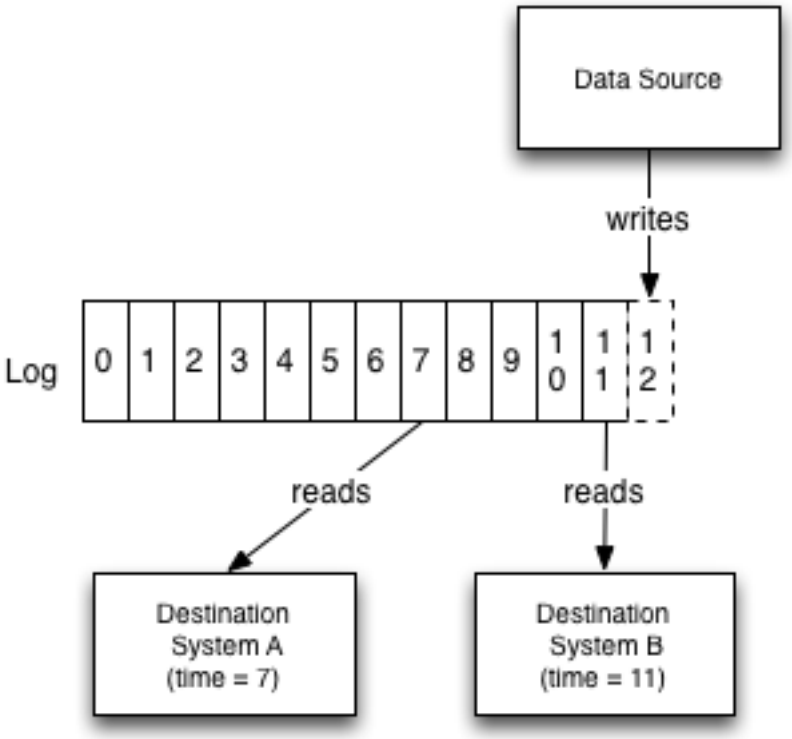
You don't hear much about data integration in all the breathless interest and hype around the idea of big data, but nonetheless, I believe this mundane problem of "making the data available" is one of the more valuable things an organization can focus on.
Effective use of data follows a kind of Maslow's hierarchy of needs. The base of the pyramid involves capturing all the relevant data, being able to put it together in an applicable processing environment (be that a fancy real-time query system or just text files and python scripts). This data needs to be modeled in a uniform way to make it easy to read and process. Once these basic needs of capturing data in a uniform way are taken care of it is reasonable to work on infrastructure to process this data in various ways—MapReduce, real-time query systems, etc.
It's worth noting the obvious: without a reliable and complete data flow, a Hadoop cluster is little more than a very expensive and difficult to assemble space heater. Once data and processing are available, one can move concern on to more refined problems of good data models and consistent well understood semantics. Finally, concentration can shift to more sophisticated processing—better visualization, reporting, and algorithmic processing and prediction.
In my experience, most organizations have huge holes in the base of this pyramid—they lack reliable complete data flow—but want to jump directly to advanced data modeling techniques. This is completely backwards.
So the question is, how can we build reliable data flow throughout all the data systems in an organization?
Data Integration: Two complications

Two trends make data integration harder.
The event data firehose
The first trend is the rise of event data. Event data records things that happen rather than things that are. In web systems, this means user activity logging, but also the machine-level events and statistics required to reliably operate and monitor a data center's worth of machines. People tend to call this "log data" since it is often written to application logs, but that confuses form with function. This data is at the heart of the modern web: Google's fortune, after all, is generated by a relevance pipeline built on clicks and impressions—that is, events.
And this stuff isn't limited to web companies, it's just that web companies are already fully digital, so they are easier to instrument. Financial data has long been event-centric. RFID adds this kind of tracking to physical objects. I think this trend will continue with the digitization of traditional businesses and activities.
This type of event data records what happened, and tends to be several orders of magnitude larger than traditional database uses. This presents significant challenges for processing.
The explosion of specialized data systems
The second trend comes from the explosion of specialized data systems that have become popular and often freely available in the last five years. Specialized systems exist for OLAP, search, simple online storage, batch processing, graph analysis, and so on.

The combination of more data of more varieties and a desire to get this data into more systems leads to a huge data integration problem.
Log-structured data flow

The log is the natural data structure for handling data flow between systems. The recipe is very simple:
Take all the organization's data and put it into a central log for real-time subscription.
Each logical data source can be modeled as its own log. A data source could be an application that logs out events (say clicks or page views), or a database table that accepts modifications. Each subscribing system reads from this log as quickly as it can, applies each new record to its own store, and advances its position in the log. Subscribers could be any kind of data system—a cache, Hadoop, another database in another site, a search system, etc.

For example, the log concept gives a logical clock for each change against which all subscribers can be measured. This makes reasoning about the state of the different subscriber systems with respect to one another far simpler, as each has a "point in time" they have read up to.
To make this more concrete, consider a simple case where there is a database and a collection of caching servers. The log provides a way to synchronize the updates to all these systems and reason about the point of time of each of these systems. Let's say we write a record with log entry X and then need to do a read from the cache. If we want to guarantee we don't see stale data, we just need to ensure we don't read from any cache which has not replicated up to X.



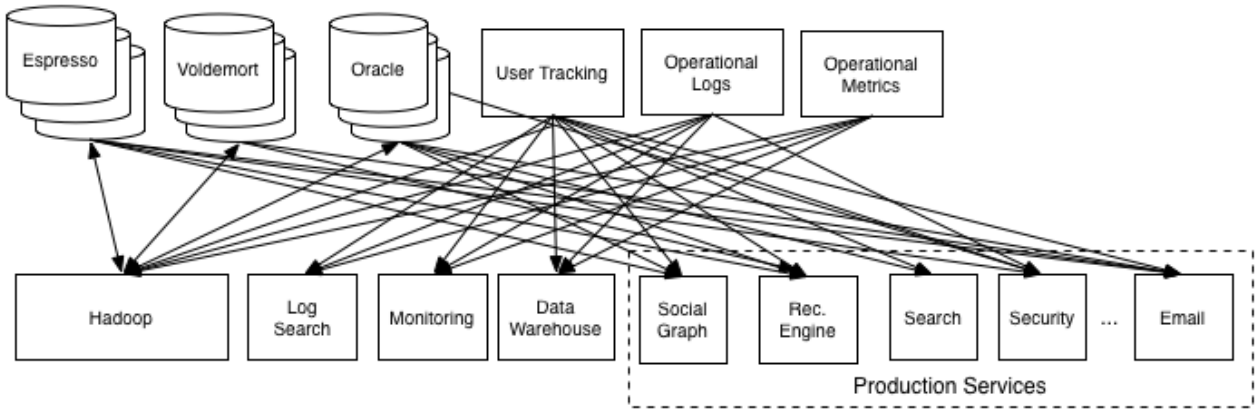
The log also acts as a buffer that makes data production asynchronous from data consumption. This is important for a lot of reasons, but particularly when there are multiple subscribers that may consume at different rates. This means a subscribing system can crash or go down for maintenance and catch up when it comes back: the subscriber consumes at a pace it controls. A batch system such as Hadoop or a data warehouse may consume only hourly or daily, whereas a real-time query system may need to be up-to-the-second. Neither the originating data source nor the log has knowledge of the various data destination systems, so consumer systems can be added and removed with no change in the pipeline.

Of particular importance: the destination system only knows about the log and not any details of the system of origin. The consumer system need not concern itself with whether the data came from an RDBMS, a new-fangled key-value store, or was generated without a real-time query system of any kind. This seems like a minor point, but is in fact critical.
I use the term "log" here instead of "messaging system" or "pub sub" because it is a lot more specific about semantics and a much closer description of what you need in a practical implementation to support data replication. I have found that "publish subscribe" doesn't imply much more than indirect addressing of messages—if you compare any two messaging systems promising publish-subscribe, you find that they guarantee very different things, and most models are not useful in this domain. You can think of the log as acting as a kind of messaging system with durability guarantees and strong ordering semantics. In distributed systems, this model of communication sometimes goes by the (somewhat terrible) name of atomic broadcast.
It's worth emphasizing that the log is still just the infrastructure. That isn't the end of the story of mastering data flow: the rest of the story is around metadata, schemas, compatibility, and all the details of handling data structure and evolution. But until there is a reliable, general way of handling the mechanics of data flow, the semantic details are secondary.
At LinkedIn

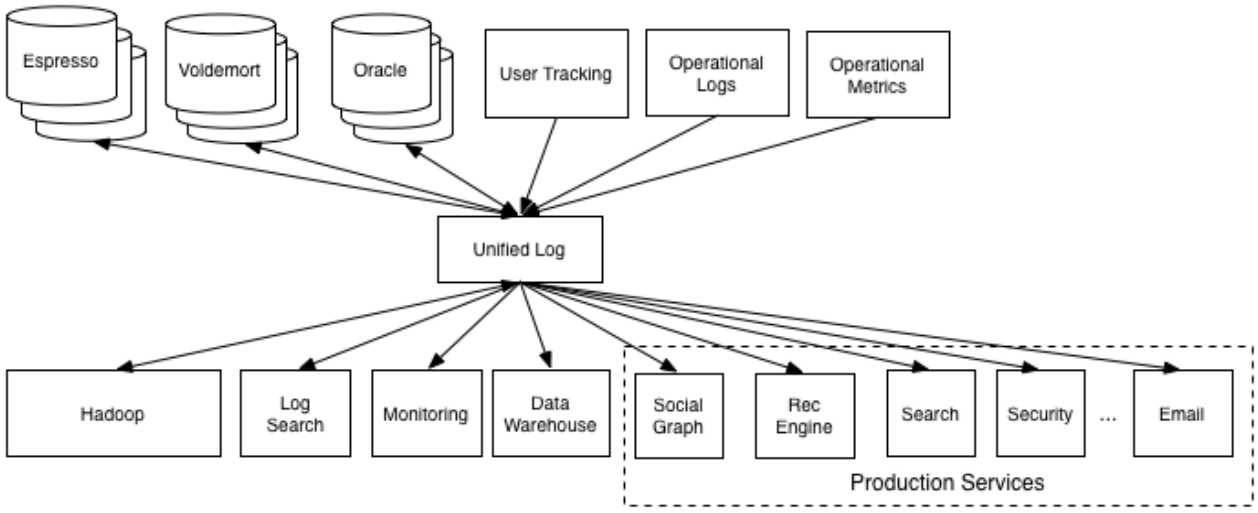
I got to watch this data integration problem emerge in fast-forward as LinkedIn moved from a centralized relational database to a collection of distributed systems.
These days our major data systems include:
Search
Social Graph
Voldemort (key-value store)
Espresso (document store)
Recommendation engine
OLAP query engine
Hadoop
Terradata
Ingraphs (monitoring graphs and metrics services)
Each of these is a specialized distributed system that provides advanced functionality in its area of specialty.

This idea of using logs for data flow has been floating around LinkedIn since even before I got here. One of the earliest pieces of infrastructure we developed was a service called databus that provided a log caching abstraction on top of our early Oracle tables to scale subscription to database changes so we could feed our social graph and search indexes.
I'll give a little bit of the history to provide context. My own involvement in this started around 2008 after we had shipped our key-value store. My next project was to try to get a working Hadoop setup going, and move some of our recommendation processes there. Having little experience in this area, we naturally budgeted a few weeks for getting data in and out, and the rest of our time for implementing fancy prediction algorithms. So began a long slog.
We originally planned to just scrape the data out of our existing Oracle data warehouse. The first discovery was that getting data out of Oracle quickly is something of a dark art. Worse, the data warehouse processing was not appropriate for the production batch processing we planned for Hadoop—much of the processing was non-reversable and specific to the reporting being done. We ended up avoiding the data warehouse and going directly to source databases and log files.
Finally, we implemented another pipeline to load data into our key-value store for serving results.
This mundane data copying ended up being one of the dominate items for the original development. Worse, any time there was a problem in any of the pipelines, the Hadoop system was largely useless—running fancy algorithms on bad data just produces more bad data.
Although we had built things in a fairly generic way, each new data source required custom configuration to set up. It also proved to be the source of a huge number of errors and failures. The site features we had implemented on Hadoop became popular and we found ourselves with a long list of interested engineers. Each user had a list of systems they wanted integration with and a long list of new data feeds they wanted.

A few things slowly became clear to me.
First, the pipelines we had built, though a bit of a mess, were actually extremely valuable. Just the process of making data available in a new processing system (Hadoop) unlocked a lot of possibilities. New computation was possible on the data that would have been hard to do before. Many new products and analysis just came from putting together multiple pieces of data that had previously been locked up in specialized systems.
Second, it was clear that reliable data loads would require deep support from the data pipeline. If we captured all the structure we needed, we could make Hadoop data loads fully automatic, so that no manual effort was expanded adding new data sources or handling schema changes—data would just magically appear in HDFS and Hive tables would automatically be generated for new data sources with the appropriate columns.
Third, we still had very low data coverage. That is, if you looked at the overall percentage of the data LinkedIn had that was available in Hadoop, it was still very incomplete. And getting to completion was not going to be easy given the amount of effort required to operationalize each new data source.
The way we had been proceeding, building out custom data loads for each data source and destination, was clearly infeasible. We had dozens of data systems and data repositories. Connecting all of these would have lead to building custom piping between each pair of systems something like this:



Note that data often flows in both directions, as many systems (databases, Hadoop) are both sources and destinations for data transfer. This meant we would end up building two pipelines per system: one to get data in and one to get data out.
This clearly would take an army of people to build and would never be operable. As we approached fully connectivity we would end up with something like O(N2) pipelines.
Instead, we needed something generic like this:



As much as possible, we needed to isolate each consumer from the source of the data. They should ideally integrate with just a single data repository that would give them access to everything.

The idea is that adding a new data system—be it a data source or a data destination—should create integration work only to connect it to a single pipeline instead of each consumer of data.

This experience lead me to focus on building Kafka to combine what we had seen in messaging systems with the log concept popular in databases and distributed system internals. We wanted something to act as a central pipeline first for all activity data, and eventually for many other uses, including data deployment out of Hadoop, monitoring data, etc.

For a long time, Kafka was a little unique (some would say odd) as an infrastructure product—neither a database nor a log file collection system nor a traditional messaging system. But recently Amazon has offered a service that is very very similar to Kafka called Kinesis. The similarity goes right down to the way partitioning is handled, data is retained, and the fairly odd split in the Kafka API between high- and low-level consumers. I was pretty happy about this. A sign you've created a good infrastructure abstraction is that AWS offers it as a service! Their vision for this seems to be exactly similar to what I am describing: it is the piping that connects all their distributed systems—DynamoDB, RedShift, S3, etc.—as well as the basis for distributed stream processing using EC2.

Relationship to ETL and the Data Warehouse

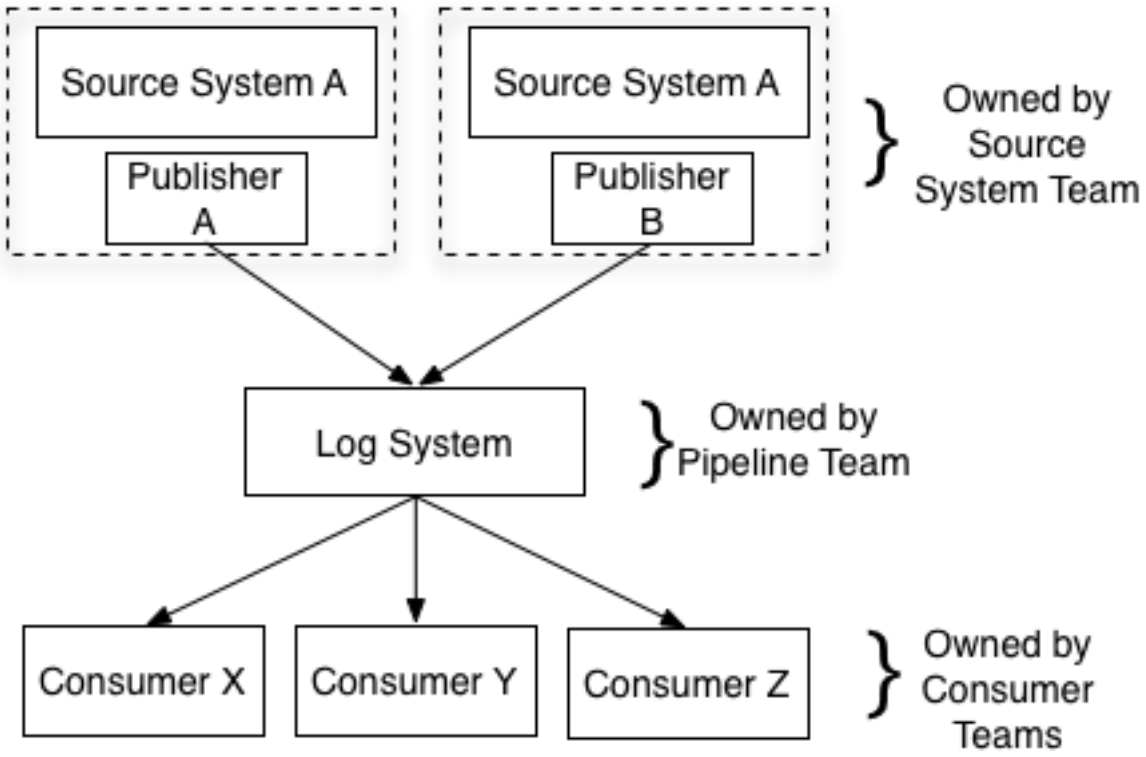
Let's talk data warehousing for a bit. The data warehouse is meant to be a repository of the clean, integrated data structured to support analysis. This is a great idea. For those not in the know, the data warehousing methodology involves periodically extracting data from source databases, munging it into some kind of understandable form, and loading it into a central data warehouse. Having this central location that contains a clean copy of all your data is a hugely valuable asset for data-intensive analysis and processing. At a high level, this methodology doesn't change too much whether you use a traditional data warehouse like Oracle or Teradata or Hadoop, though you might switch up the order of loading and munging.

A data warehouse containing clean, integrated data is a phenomenal asset, but the mechanics of getting this are a bit out of date.

The key problem for a data-centric organization is coupling the clean integrated data to the data warehouse. A data warehouse is a piece of batch query infrastructure which is well suited to many kinds of reporting and ad hoc analysis, particularly when the queries involve simple counting, aggregation, and filtering. But having a batch system be the only repository of clean complete data means the data is unavailable for systems requiring a real-time feed—real-time processing, search indexing, monitoring systems, etc.

In my view, ETL is really two things. First, it is an extraction and data cleanup process—essentially liberating data locked up in a variety of systems in the organization and removing an system-specific non-sense. Secondly, that data is restructured for data warehousing queries (i.e. made to

fit the type system of a relational DB, forced into a star or snowflake schema, perhaps broken up into a high performance column format, etc). Conflating these two things is a problem. The clean, integrated repository of data should be available in real-time as well for low-latency processing as well as indexing in other real-time storage systems.



I think this has the added benefit of making data warehousing ETL much more organizationally scalable. The classic problem of the data warehouse team is that they are responsible for collecting and cleaning all the data generated by every other team in the organization. The incentives are not aligned: data producers are often not very aware of the use of the data in the data warehouse and end up creating data that is hard to extract or requires heavy, hard to scale transformation to get into usable form. Of course, the central team never quite manages to scale to match the pace of the rest of the organization, so data coverage is always spotty, data flow is fragile, and changes are slow.
A better approach is to have a central pipeline, the log, with a well defined API for adding data. The responsibility of integrating with this pipeline and providing a clean, well-structured data feed lies with the producer of this data feed. This means that as part of their system design and implementation they must consider the problem of getting data out and into a well structured form for delivery to the central pipeline. The addition of new storage systems is of no consequence to the data warehouse team as they have a central point of integration. The data warehouse team handles only the simpler problem of loading structured feeds of data from the central log and carrying out transformation specific to their system.

This point about organizational scalability becomes particularly important when one considers adopting additional data systems beyond a traditional data warehouse. Say, for example, that one wishes to provide search capabilities over the complete data set of the organization. Or, say that one wants to provide sub-second monitoring of data streams with real-time trend graphs and alerting. In either of these cases, the infrastructure of the traditional data warehouse or even a

Hadoop cluster is going to be inappropriate. Worse, the ETL processing pipeline built to support database loads is likely of no use for feeding these other systems, making bootstrapping these pieces of infrastructure as large an undertaking as adopting a data warehouse. This likely isn't feasible and probably helps explain why most organizations do not have these capabilities easily available for all their data. By contrast, if the organization had built out feeds of uniform, well-structured data, getting any new system full access to all data requires only a single bit of integration plumbing to attach to the pipeline.

This architecture also raises a set of different options for where a particular cleanup or transformation can reside:

It can be done by the data producer prior to adding the data to the company wide log.

It can be done as a real-time transformation on the log (which in turn produces a new, transformed log)

It can be done as part of the load process into some destination data system

The best model is to have cleanup done prior to publishing the data to the log by the publisher of the data. This means ensuring the data is in a canonical form and doesn't retain any hold-overs from the particular code that produced it or the storage system in which it may have been maintained. These details are best handled by the team that creates the data since they know the most about their own data. Any logic applied in this stage should be lossless and reversible.

Any kind of value-added transformation that can be done in real-time should be done as post-processing on the raw log feed produced. This would include things like sessionization of event data, or the addition of other derived fields that are of general interest. The original log is still available, but this real-time processing produces a derived log containing augmented data.

Finally, only aggregation that is specific to the destination system should be performed as part of the loading process. This might include transforming data into a particular star or snowflake schema for analysis and reporting in a data warehouse. Because this stage, which most naturally maps to the traditional ETL process, is now done on a far cleaner and more uniform set of streams, it should be much simplified.

Log Files and Events

Let's talk a little bit about a side benefit of this architecture: it enables decoupled, event-driven systems.

The typical approach to activity data in the web industry is to log it out to text files where it can be scrapped into a data warehouse or into Hadoop for aggregation and querying. The problem with this is the same as the problem with all batch ETL: it couples the data flow to the data warehouse's capabilities and processing schedule.

At LinkedIn, we have built our event data handling in a log-centric fashion. We are using Kafka as the central, multi-subscriber event log. We have defined several hundred event types, each capturing the unique attributes about a particular type of action. This covers everything from page views, ad impressions, and searches, to service invocations and application exceptions.

To understand the advantages of this, imagine a simple event—showing a job posting on the job page. The job page should contain only the logic required to display the job. However, in a fairly dynamic site, this could easily become larded up with additional logic unrelated to showing the job. For example let's say we need to integrate the following systems:

We need to send this data to Hadoop and data warehouse for offline processing purposes

We need to count the view to ensure that the viewer is not attempting some kind of content scraping

We need to aggregate this view for display in the Job poster's analytics page

We need to record the view to ensure we properly impression cap any job recommendations for that user (we don't want to show the same thing over and over)

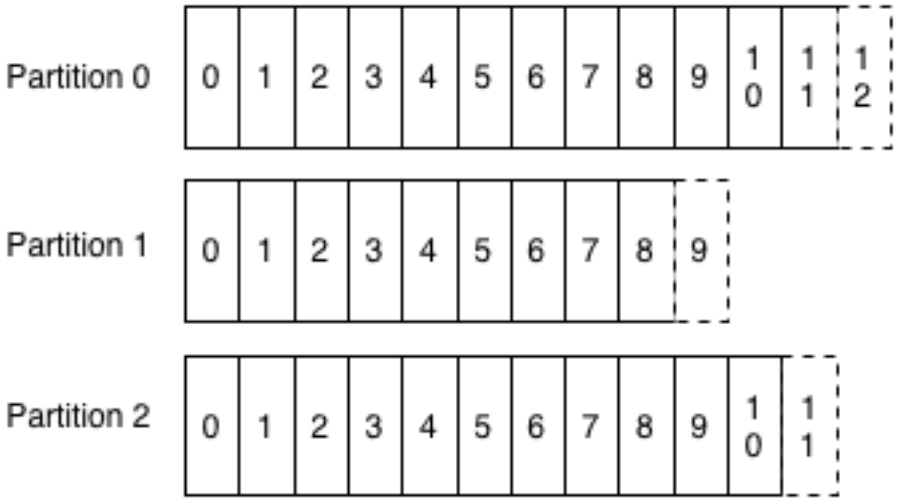
Our recommendation system may need to record the view to correctly track the popularity of that job

Etc

Pretty soon, the simple act of displaying a job has become quite complex. And as we add other places where jobs are displayed—mobile applications, and so on—this logic must be carried over and the complexity increases. Worse, the systems that we need to interface with are now somewhat intertwined—the person working on displaying jobs needs to know about many other

systems and features and make sure they are integrated properly. This is just a toy version of the problem, any real application would be more, not less, complex.
The "event-driven" style provides an approach to simplifying this. The job display page now just shows a job and records the fact that a job was shown along with the relevant attributes of the job, the viewer, and any other useful facts about the display of the job. Each of the other interested systems—the recommendation system, the security system, the job poster analytics system, and the data warehouse—all just subscribe to the feed and do their processing. The display code need not be aware of these other systems, and needn't be changed if a new data consumer is added.
Building a Scalable Log

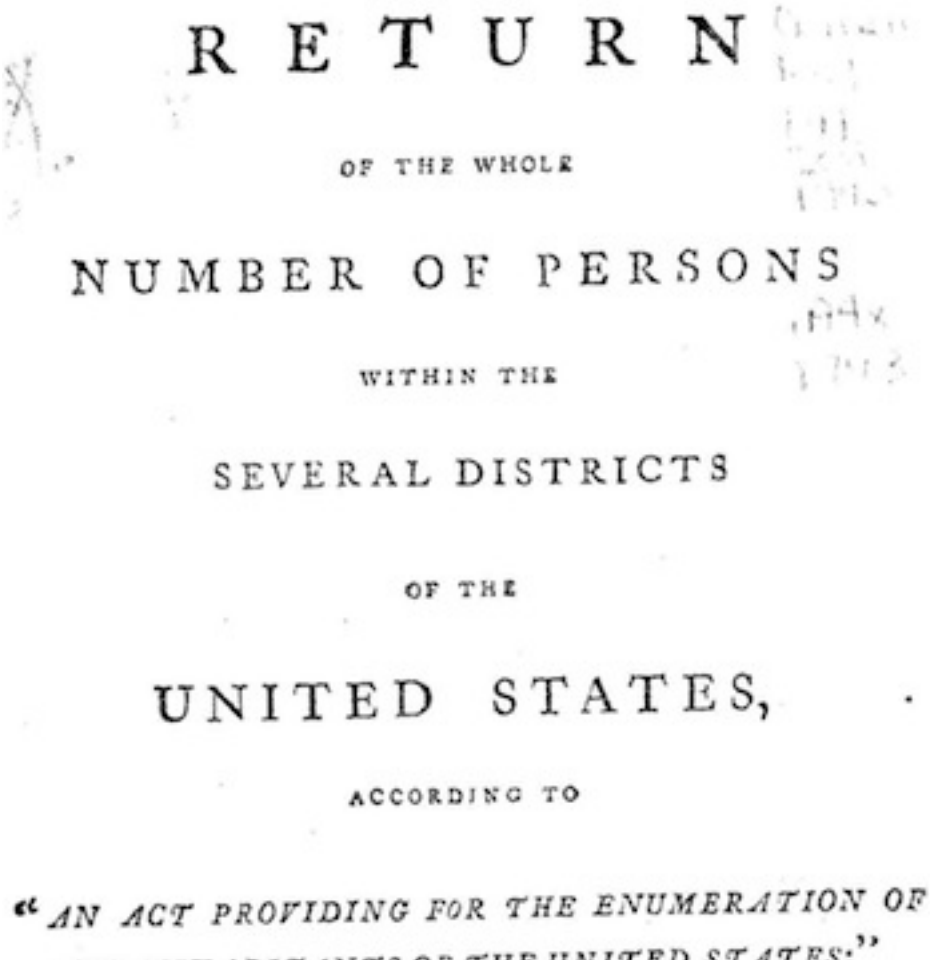
Of course, separating publishers from subscribers is nothing new. But if you want to keep a commit log that acts as a multi-subscriber real-time journal of everything happening on a consumer-scale website, scalability will be a primary challenge. Using a log as a universal integration mechanism is never going to be more than an elegant fantasy if we can't build a log that is fast, cheap, and scalable enough to make this practical at scale.
Systems people typically think of a distributed log as a slow, heavy-weight abstraction (and usually associate it only with the kind of "metadata" uses for which Zookeeper might be appropriate). But with a thoughtful implementation focused on journaling large data streams, this need not be true. At LinkedIn we are currently running over 60 billion unique message writes through Kafka per day (several hundred billion if you count the writes from mirroring between datacenters).
We used a few tricks in Kafka to support this kind of scale:
Partitioning the log
Optimizing throughput by batching reads and writes
Avoiding needless data copies
In order to allow horizontal scaling we chop up our log into partitions:



Each partition is a totally ordered log, but there is no global ordering between partitions (other than perhaps some wall-clock time you might include in your messages). The assignment of the messages to a particular partition is controllable by the writer, with most users choosing to partition by some kind of key (e.g. user id). Partitioning allows log appends to occur without co-ordination between shards and allows the throughput of the system to scale linearly with the Kafka cluster size.
Each partition is replicated across a configurable number of replicas, each of which has an identical copy of the partition's log. At any time, a single one of them will act as the leader; if the leader fails, one of the replicas will take over as leader.
Lack of a global order across partitions is a limitation, but we have not found it to be a major one. Indeed, interaction with the log typically comes from hundreds or thousands of distinct processes so it is not meaningful to talk about a total order over their behavior. Instead, the guarantees that

we provide are that each partition is order preserving, and Kafka guarantees that appends to a particular partition from a single sender will be delivered in the order they are sent.
A log, like a filesystem, is easy to optimize for linear read and write patterns. The log can group small reads and writes together into larger, high-throughput operations. Kafka pursues this optimization aggressively. Batching occurs from client to server when sending data, in writes to disk, in replication between servers, in data transfer to consumers, and in acknowledging committed data.
Finally, Kafka uses a simple binary format that is maintained between in-memory log, on-disk log, and in network data transfers. This allows us to make use of numerous optimizations including zero-copy data transfer.
The cumulative effect of these optimizations is that you can usually write and read data at the rate supported by the disk or network, even while maintaining data sets that vastly exceed memory.
This write-up isn't meant to be primarily about Kafka so I won't go into further details. You can read a more detailed overview of LinkedIn's approach here and a thorough overview of Kafka's design here.
Part Three: Logs & Real-time Stream Processing

So far, I have only described what amounts to a fancy method of copying data from place-to-place. But shlepping bytes between storage systems is not the end of the story. It turns out that "log" is another word for "stream" and logs are at the heart of stream processing.
But, wait, what exactly is stream processing?
If you are a fan of late 90s and early 2000s database literature or semi-successful data infrastructure products, you likely associate stream processing with efforts to build a SQL engine or "boxes and arrows" interface for event driven processing.
If you follow the explosion of open source data systems, you likely associate stream processing with some of the systems in this space—for example, Storm, Akka, S4, and Samza. But most people see these as a kind of asynchronous message processing system not that different from a cluster-aware RPC layer (and in fact some things in this space are exactly that).
Both these views are a little limited. Stream processing has nothing to do with SQL. Nor is it limited to real-time processing. There is no inherent reason you can't process the stream of data from yesterday or a month ago using a variety of different languages to express the computation.



RETURN

OF THE WHOLE

NUMBER OF PERSONS

WITHIN THE

SEVERAL DISTRICTS

OF THE

UNITED STATES,

ACCORDING TO

"AN ACT PROVIDING FOR THE ENUMERATION OF

I see stream processing as something much broader: infrastructure for continuous data processing. I think the computational model can be as general as MapReduce or other distributed processing frameworks, but with the ability to produce low-latency results.

The real driver for the processing model is the method of data collection. Data which is collected in batch is naturally processed in batch. When data is collected continuously, it is naturally processed continuously.

The US census provides a good example of batch data collection. The census periodically kicks off and does a brute force discovery and enumeration of US citizens by having people walking around door-to-door. This made a lot of sense in 1790 when the census was first begun. Data collection at the time was inherently batch oriented, it involved riding around on horseback and writing down records on paper, then transporting this batch of records to a central location where humans added up all the counts. These days, when you describe the census process one immediately wonders why we don't keep a journal of births and deaths and produce population counts either continuously or with whatever granularity is needed.

This is an extreme example, but many data transfer processes still depend on taking periodic dumps and bulk transfer and integration. The only natural way to process a bulk dump is with a batch process. But as these processes are replaced with continuous feeds, one naturally starts to move towards continuous processing to smooth out the processing resources needed and reduce latency.

LinkedIn, for example, has almost no batch data collection at all. The majority of our data is either activity data or database changes, both of which occur continuously. In fact, when you think about any business, the underlying mechanics are almost always a continuous process—events happen in real-time, as Jack Bauer would tell us. When data is collected in batches, it is almost always due to some manual step or lack of digitization or is a historical relic left over from the automation of some non-digital process. Transmitting and reacting to data used to be very slow when the mechanics were mail and humans did the processing. A first pass at automation always retains the form of the original process, so this often lingers for a long time.

Production "batch" processing jobs that run daily are often effectively mimicking a kind of continuous computation with a window size of one day. The underlying data is, of course, always changing. These were actually so common at LinkedIn (and the mechanics of making them work in Hadoop so tricky) that we implemented a whole framework for managing incremental Hadoop workflows.

Seen in this light, it is easy to have a different view of stream processing: it is just processing which includes a notion of time in the underlying data being processed and does not require a static snapshot of the data so it can produce output at a user-controlled frequency instead of waiting for the "end" of the data set to be reached. In this sense, stream processing is a generalization of batch processing, and, given the prevalence of real-time data, a very important generalization.

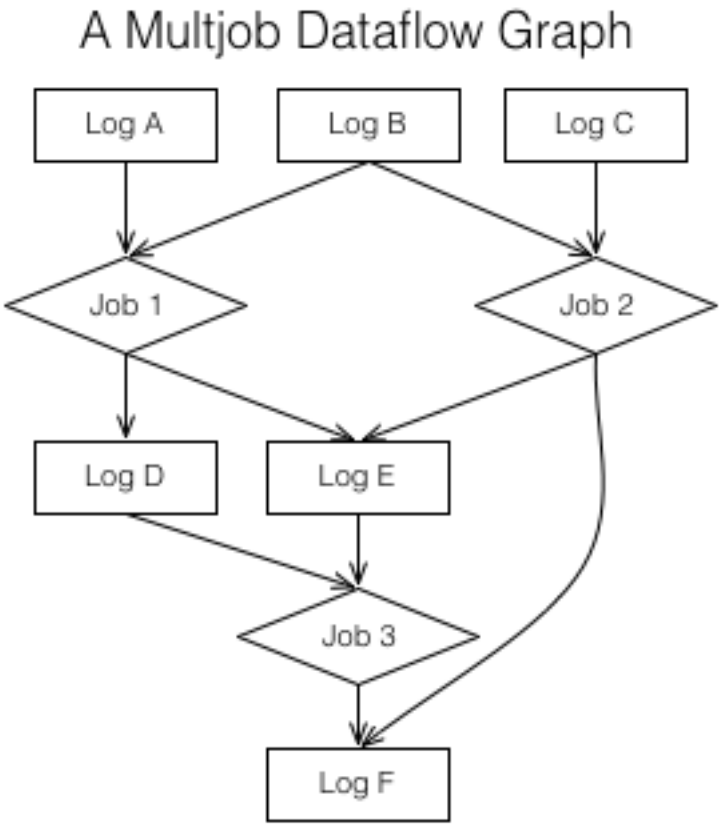
So why has the traditional view of stream processing been as a niche application? I think the biggest reason is that a lack of real-time data collection made continuous processing something of an academic concern.

I think the lack of real-time data collection is likely what doomed the commercial stream-processing systems. Their customers were still doing file-oriented, daily batch processing for ETL and data integration. Companies building stream processing systems focused on providing processing engines to attach to real-time data streams, but it turned out that at the time very few people actually had real-time data streams. Actually, very early at my career at LinkedIn, a company tried to sell us a very cool stream processing system, but since all our data was collected in hourly files at that time, the best application we could come up with was to pipe the hourly files into the stream system at the end of the hour! They noted that this was a fairly common problem. The exception actually proves the rule here: finance, the one domain where stream processing has met with some success, was exactly the area where real-time data streams were already the norm and processing had become the bottleneck.

Even in the presence of a healthy batch processing ecosystem, I think the actual applicability of stream processing as an infrastructure style is quite broad. I think it covers the gap in infrastructure between real-time request/response services and offline batch processing. For modern internet companies, I think around 25% of their code falls into this category.

It turns out that the log solves some of the most critical technical problems in stream processing, which I'll describe, but the biggest problem that it solves is just making data available in real-time multi-subscriber data feeds. For those interested in more details, we have open sourced Samza, a stream processing system explicitly built on many of these ideas. We describe a lot of these applications in more detail in the documentation here.

Data flow graphs



The most interesting aspect of stream processing has nothing to do with the internals of a stream processing system, but instead has to do with how it extends our idea of what a data feed is from the earlier data integration discussion. We discussed primarily feeds or logs of primary data—the events and rows of data produced in the execution of various applications. But stream processing allows us to also include feeds computed off other feeds. These derived feeds look no different to consumers then the feeds of primary data from which they are computed. These derived feeds can encapsulate arbitrary complexity.

Let's dive into this a bit. A stream processing job, for our purposes, will be anything that reads from logs and writes output to logs or other systems. The logs they use for input and output join these processes into a graph of processing stages. Indeed, using a centralized log in this fashion, you can view all the organization's data capture, transformation, and flow as just a series of logs and processes that write to them.

A stream processor need not have a fancy framework at all: it can be any process or set of processes that read and write from logs, but additional infrastructure and support can be provided for helping manage processing code.

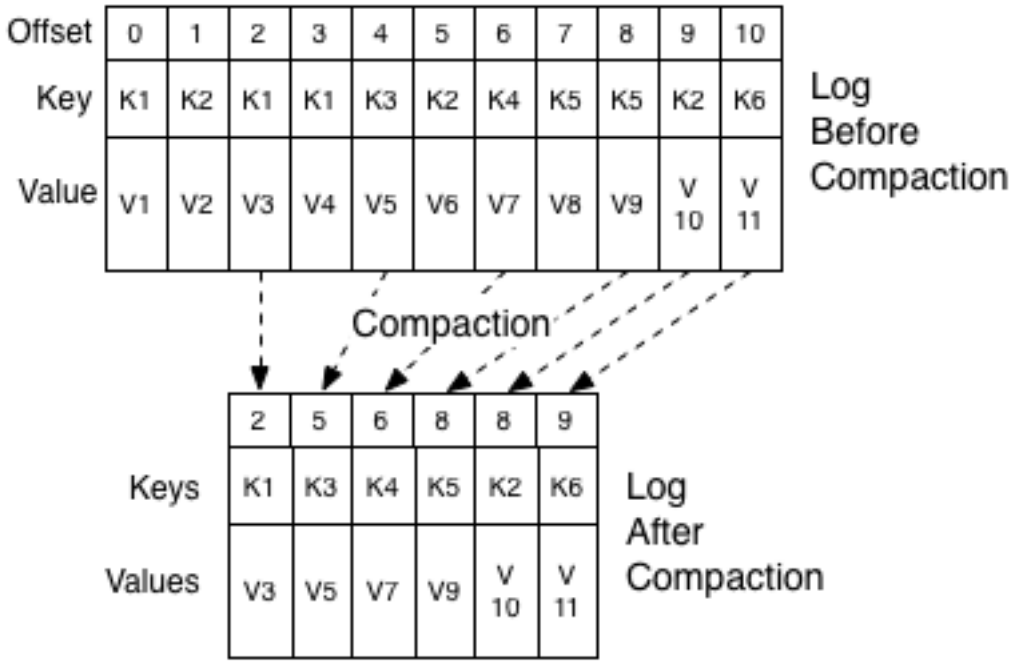
The purpose of the log in the integration is two-fold.

First, it makes each dataset multi-subscriber and ordered. Recall our "state replication" principle to remember the importance of order. To make this more concrete, consider a stream of updates from a database—if we re-order two updates to the same record in our processing we may produce the wrong final output. This order is more permanent than what is provided by something like TCP as it is not limited to a single point-to-point link and survives beyond process failures and reconnections.

Second, the log provides buffering to the processes. This is very fundamental. If processing proceeds in an unsynchronized fashion it is likely to happen that an upstream data producing job will produce data more quickly than another downstream job can consume it. When this occurs processing must block, buffer or drop data. Dropping data is likely not an option; blocking may cause the entire processing graph to grind to a halt. The log acts as a very, very large buffer that allows process to be restarted or fail without slowing down other parts of the processing graph. This isolation is particularly important when extending this data flow to a larger organization, where processing is happening by jobs made by many different teams. We cannot have one faulty job cause back-pressure that stops the entire processing flow.
Both Storm and Samza are built in this fashion and can use Kafka or other similar systems as their log.
Stateful Real-Time Processing

Some real-time stream processing is just stateless record-at-a-time transformation, but many of the uses are more sophisticated counts, aggregations, or joins over windows in the stream. One might, for example, want to enrich an event stream (say a stream of clicks) with information about the user doing the click—in effect joining the click stream to the user account database. Invariably, this kind of processing ends up requiring some kind of state to be maintained by the processor: for example, when computing a count, you have the count so far to maintain. How can this kind of state be maintained correctly if the processors themselves can fail?
The simplest alternative would be to keep state in memory. However if the process crashed it would lose its intermediate state. If state is only maintained over a window, the process could just fall back to the point in the log where the window began. However, if one is doing a count over an hour, this may not be feasible.
An alternative is to simply store all state in a remote storage system and join over the network to that store. The problem with this is that there is no locality of data and lots of network round-trips. How can we support something like a "table" that is partitioned up with our processing?
Well recall the discussion of the duality of tables and logs. This gives us exactly the tool to be able to convert streams to tables co-located with our processing, as well as a mechanism for handling fault tolerance for these tables.
A stream processor can keep it's state in a local "table" or "index"—a bdb, leveldb, or even something more unusual such as a Lucene or fastbit index. The contents of this store is fed from its input streams (after first perhaps applying arbitrary transformation). It can journal out a changelog for this local index it keeps to allow it to restore its state in the event of a crash and restart. This mechanism allows a generic mechanism for keeping co-partitioned state in arbitrary index types local with the incoming stream data.
When the process fails, it restores its index from the changelog. The log is the transformation of the local state into a sort of incremental record at a time backup.
This approach to state management has the elegant property that the state of the processors is also maintained as a log. We can think of this log just like we would the log of changes to a database table. In fact, the processors have something very like a co-partitioned table maintained along with them. Since this state is itself a log, other processors can subscribe to it. This can actually be quite useful in cases when the goal of the processing is to update a final state and this state is the natural output of the processing.
When combined with the logs coming out of databases for data integration purposes, the power of the log/table duality becomes clear. A change log may be extracted from a database and indexed in different forms by various stream processors to join against event streams.
We give more detail on this style of managing stateful processing in Samza and a lot more practical examples here.

Log Compaction



Of course, we can't hope to keep a complete log for all state changes for all time. Unless one wants to use infinite space, somehow the log must be cleaned up. I'll talk a little about the implementation of this in Kafka to make it more concrete. In Kafka, cleanup has two options depending on whether the data contains keyed updates or event data. For event data, Kafka supports just retaining a window of data. Usually, this is configured to a few days, but the window can be defined in terms of time or space. For keyed data, though, a nice property of the complete log is that you can replay it to recreate the state of the source system (potentially recreating it in another system).

However, retaining the complete log will use more and more space as time goes by, and the replay will take longer and longer. Hence, in Kafka, we support a different type of retention. Instead of simply throwing away the old log, we remove obsolete records—i.e. records whose primary key has a more recent update. By doing this, we still guarantee that the log contains a complete backup of the source system, but now we can no longer recreate all previous states of the source system, only the more recent ones. We call this feature log compaction.

Part Four: System Building

The final topic I want to discuss is the role of the log in data system design for online data systems. There is an analogy here between the role a log serves for data flow inside a distributed database and the role it serves for data integration in a larger organization. In both cases, it is responsible for data flow, consistency, and recovery. What, after all, is an organization, if not a very complicated distributed data system?

Unbundling?

So maybe if you squint a bit, you can see the whole of your organization's systems and data flows as a single distributed database. You can view all the individual query-oriented systems (Redis, SOLR, Hive tables, and so on) as just particular indexes on your data. You can view the stream processing systems like Storm or Samza as just a very well-developed trigger and view materialization mechanism. Classical database people, I have noticed, like this view very much because it finally explains to them what on earth people are doing with all these different data systems—they are just different index types!

There is undeniably now an explosion of types of data systems, but in reality, this complexity has always existed. Even in the heyday of the relational database, organizations had lots and lots of relational databases! So perhaps real integration hasn't existed since the mainframe when all the data really was in one place. There are many motivations for segregating data into multiple systems: scale, geography, security, and performance isolation are the most common. But these issues can be addressed by a good system: it is possible for an organization to have a single

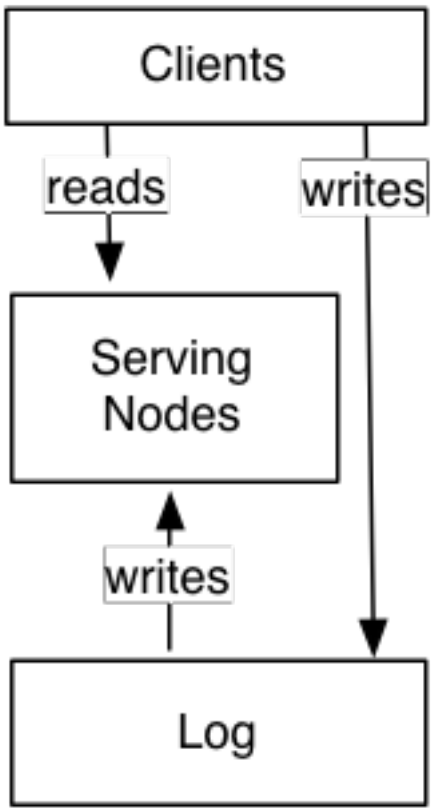
Hadoop cluster, for example, that contains all the data and serves a large and diverse constituency.
So there is already one possible simplification in the handling of data that has become possible in the move to distributed systems: coalescing lots of little instances of each system into a few big clusters. Many systems aren't good enough to allow this yet: they don't have security, or can't guarantee performance isolation, or just don't scale well enough. But each of these problems is solvable.
My take is that the explosion of different systems is caused by the difficulty of building distributed data systems. By cutting back to a single query type or use case each system is able to bring its scope down into the set of things that are feasible to build. But running all these systems yields too much complexity.
I see three possible directions this could follow in the future.
The first possibility is a continuation of the status quo: the separation of systems remains more or less as it is for a good deal longer. This could happen either because the difficulty of distribution is too hard to overcome or because this specialization allows new levels of convenience and power for each system. As long as this remains true, the data integration problem will remain one of the most centrally important things for the successful use of data. In this case, an external log that integrates data will be very important.
The second possibility is that there could be a re-consolidation in which a single system with enough generality starts to merge back in all the different functions into a single uber-system. This uber-system could be like the relational database superficially, but it's use in an organization would be far different as you would need only one big one instead of umpteen little ones. In this world, there is no real data integration problem except what is solved inside this system. I think the practical difficulties of building such a system make this unlikely.
There is another possible outcome, though, which I actually find appealing as an engineer. One interesting facet of the new generation of data systems is that they are virtually all open source. Open source allows another possibility: data infrastructure could be unbundled into a collection of services and application-facing system apis. You already see this happening to a certain extent in the Java stack:
Zookeeper handles much of the system co-ordination (perhaps with a bit of help from higher-level abstractions like Helix or Curator).
Mesos and YARN do process virtualization and resource management
Embedded libraries like Lucene and LevelDB do indexing
Netty, Jetty and higher-level wrappers like Finagle and rest.li handle remote communication
Avro, Protocol Buffers, Thrift, and umpteen zillion other libraries handle serialization
Kafka and BookKeeper provide a backing log.
If you stack these things in a pile and squint a bit, it starts to look a bit like a lego version of distributed data system engineering. You can piece these ingredients together to create a vast array of possible systems. This is clearly not a story relevant to end-users who presumably care primarily more about the API then how it is implemented, but it might be a path towards getting the simplicity of the single system in a more diverse and modular world that continues to evolve. If the implementation time for a distributed system goes from years to weeks because reliable, flexible building blocks emerge, then the pressure to coalesce into a single monolithic system disappears.
The place of the log in system architecture

A system that assumes an external log is present allows the individual systems to relinquish a lot of their own complexity and rely on the shared log. Here are the things I think a log can do:
Handle data consistency (whether eventual or immediate) by sequencing concurrent updates to nodes
Provide data replication between nodes
Provide "commit" semantics to the writer (i.e. acknowledging only when your write guaranteed not to be lost)
Provide the external data subscription feed from the system
Provide the capability to restore failed replicas that lost their data or bootstrap new replicas
Handle rebalancing of data between nodes.

This is actually a substantial portion of what a distributed data system does. In fact, the majority of what is left over is related to the final client-facing query API and indexing strategy. This is exactly the part that should vary from system to system: for example, a full-text search query may need to query all partitions whereas a query by primary key may only need to query a single node responsible for that key's data.

Here is how this works. The system is divided into two logical pieces: the log and the serving layer. The log captures the state changes in sequential order. The serving nodes store whatever index is required to serve queries (for example a key-value store might have something like a btree or sstable, a search system would have an inverted index). Writes may either go directly to the log, though they may be proxied by the serving layer. Writing to the log yields a logical timestamp (say the index in the log). If the system is partitioned, and I assume it is, then the log and the serving nodes will have the same number of partitions, though they may have very different numbers of machines.

The serving nodes subscribe to the log and apply writes as quickly as possible to its local index in the order the log has stored them.

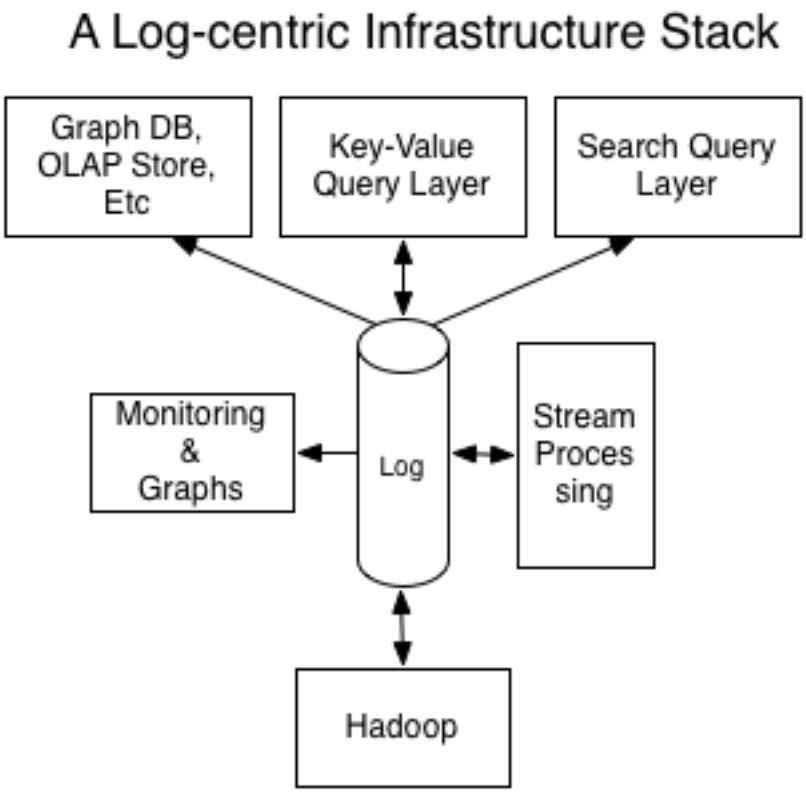


The client can get read-your-write semantics from any node by providing the timestamp of a write as part of its query—a serving node receiving such a query will compare the desired timestamp to its own index point and if necessary delay the request until it has indexed up to at least that time to avoid serving stale data.

The serving nodes may or may not need to have any notion of "mastership" or "leader election". For many simple use cases, the serving nodes can be completely without leaders, since the log is the source of truth.

One of the trickier things a distributed system must do is handle restoring failed nodes or moving partitions from node to node. A typical approach would have the log retain only a fixed window of data and combine this with a snapshot of the data stored in the partition. It is equally possible for the log to retain a complete copy of data and garbage collect the log itself. This moves a significant amount of complexity out of the serving layer, which is system-specific, and into the log, which can be general purpose.

By having this log system, you get a fully developed subscription API for the contents of the data store which feeds ETL into other systems. In fact, many systems can share the same log while providing different indexes, like this:



Note how such a log-centric system is itself immediately a provider of data streams for processing and loading in other systems. Likewise, a stream processor can consume multiple input streams and then serve them via another system that indexes that output.

I find this view of systems as factored into a log and query api very revealing, as it lets you separate the query characteristics from the availability and consistency aspects of the system. I actually think this is even a useful way to mentally factor a system that isn't built this way to better understand it.

It's worth noting that although Kafka and BookKeeper are consistent logs, this is not a requirement. You could just as easily factor a Dynamo-like database into an eventually consistent AP log and a key-value serving layer. Such a log is a bit tricky to work with, as it will redeliver old messages and depends on the subscriber to handle this (much like Dynamo itself).

The idea of having a separate copy of data in the log (especially if it is a complete copy) strikes many people as wasteful. In reality, though there are a few factors that make this less of an issue. First, the log can be a particularly efficient storage mechanism. We store up to 5TB on our production Kafka servers. Meanwhile many serving systems require much more memory to serve data efficiently (text search, for example, is often all in memory). The serving system may also use optimized hardware. For example, most our live data systems either serve out of memory or else use SSDs. In contrast, the log system does only linear reads and writes, so it is quite happy using large multi-TB hard drives. Finally, as in the picture above, in the case where the data is served by multiple systems, the cost of the log is amortized over multiple indexes. This combination makes the expense of an external log pretty minimal.

This is exactly the pattern that LinkedIn has used to build out many of its own real-time query systems. These systems feed off a database (using Databus as a log abstraction or off a dedicated log from Kafka) and provide a particular partitioning, indexing, and query capability on top of that data stream. This is the way we have implemented our search, social graph, and OLAP query systems. In fact, it is quite common to have a single data feed (whether a live feed or a derived

feed coming from Hadoop) replicated into multiple serving systems for live serving. This has proven to be an enormous simplifying assumption. None of these systems need to have an externally accessible write api at all, Kafka and databases are used as the system of record and changes flow to the appropriate query systems through that log. Writes are handled locally by the nodes hosting a particular partition. These nodes blindly transcribe the feed provided by the log to their own store. A failed node can be restored by replaying the upstream log.
The degree to which these systems rely on the log varies. A fully reliant system could make use of the log for data partitioning, node restore, rebalancing, and all aspects of consistency and data propagation. In this setup, the actual serving tier is actually nothing less than a sort of "cache" structured to enable a particular type of processing with writes going directly to the log.
The End

If you made it this far you know most of what I know about logs.
Here are a few interesting references you may want to check out.
Everyone seems to uses different terms for the same things so it is a bit of a puzzle to connect the database literature to the distributed systems stuff to the various enterprise software camps to the open source world. Nonetheless, here are a few pointers in the general direction.
Academic papers, systems, talks, and blogs:
A good overview of state machine and primary-backup replication
PacificA is a generic framework for implementing log-based distributed storage systems at Microsoft.
Spanner—Not everyone loves logical time for their logs. Google's new database tries to use physical time and models the uncertainty of clock drift directly by treating the timestamp as a range.
Datanomic: Deconstructing the database is a great presentation by Rich Hickey, the creator of Clojure, on his startup's database product.
A Survey of Rollback-Recovery Protocols in Message-Passing Systems. I found this to be a very helpful introduction to fault-tolerance and the practical application of logs to recovery outside databases.
Reactive Manifesto—I'm actually not quite sure what is meant by reactive programming, but I think it means the same thing as "event driven". This link doesn't have much info, but this class by Martin Odersky (of Scala fame) looks facinating.
Paxos!
Original paper is here. Leslie Lamport has an interesting history of how the algorithm was created in the 1980s but not published until 1998 because the reviewers didn't like the Greek parable in the paper and he didn't want to change it.
Even once the original paper was published it wasn't well understood. Lamport tries again and this time even includes a few of the "uninteresting details" of how to put it to use using these new-fangled automatic computers. It is still not widely understood.
Fred Schneider and Butler Lampson each give more detailed overview of applying Paxos in real systems.
A few Google engineers summarize their experience implementing Paxos in Chubby.
I actually found all the Paxos papers pretty painful to understand but dutifully struggled through. But you don't need to because this video by John Ousterhout (of log-structured filesystem fame!) will make it all very simple. Somehow these consensus algorithms are much better presented by drawing them as the communication rounds unfold, rather than in a static presentation in a paper. Ironically, this video was created in an attempt to show that Paxos was hard to understand.
Using Paxos to Build a Scalable Consistent Data Store: This is a cool paper on using a log to build a data store, by Jun, one of the co-authors is also one of the earliest engineers on Kafka.
Paxos has competitors! Actually each of these map a lot more closely to the implementation of a log and are probably more suitable for practical implementation:
Viewstamped Replication by Barbara Liskov is an early algorithm to directly model log replication.
Zab is the algorithm used by Zookeeper.
RAFT is an attempt at a more understandable consensus algorithm. The video presentation, also by John Ousterhout, is great too.
You can see the role of the log in action in different real distributed databases.

PNUTS is a system which attempts to apply to log-centric design of traditional distributed databases at large scale.
HBase and Bigtable both give another example of logs in modern databases.
LinkedIn's own distributed database Espresso, like PNUTs, uses a log for replication, but takes a slightly different approach using the underlying table itself as the source of the log.
If you find yourself comparison shopping for a replication algorithm, this paper may help you out.
Replication: Theory and Practice is a great book that collects a bunch of summary papers on replication in distributed systems. Many of the chapters are online (e.g. 1, 4, 5, 6, 7, 8).
Stream processing. This is a bit too broad to summarize, but here are a few things I liked.
Models and Issues in Data Stream Systems: probably the best overview of the early research in this area.
High-Availability Algorithms for Distributed Stream Processing
A couple of random systems papers:
TelegraphCQ
Aurora
NiagaraCQ
Discretized Streams: This paper discusses Spark's streaming system.
MillWheel is one of Google's stream processing systems.
Naiad: A Timely Dataflow System
Enterprise software has all the same problems but with different names, a smaller scale, and XML. Ha ha, just kidding. Kind of.
Event Sourcing—As far as I can tell this is basically the enterprise software engineer's way of saying "state machine replication". It's interesting that the same idea would be invented again in such a different context. Event sourcing seems to focus on smaller, in-memory use cases. This approach to application development seems to combine the "stream processing" that occurs on the log of events with the application. Since this becomes pretty non-trivial when the processing is large enough to require data partitioning for scale I focus on stream processing as a separate infrastructure primitive.
Change Data Capture—There is a small industry around getting data out of databases, and this is the most log-friendly style of data extraction.
Enterprise Application Integration seems to be about solving the data integration problem when what you have is a collection of off-the-shelf enterprise software like CRM or supply-chain management software.
Complex Event Processing (CEP): Fairly certain nobody knows what this means or how it actually differs from stream processing. The difference seems to be that the focus is on unordered streams and on event filtering and detection rather than aggregation, but this, in my opinion is a distinction without a difference. I think any system that is good at one should be good at another.
Enterprise Service Bus—I think the enterprise service bus concept is very similar to some of the ideas I have described around data integration. This idea seems to have been moderately successful in enterprise software communities and is mostly unknown among web folks or the distributed data infrastructure crowd.
Interesting open source stuff:
Kafka Is the "log as a service" project that is the basis for much of this post.
BookKeeper and Hedwig comprise another open source "log as a service". They seem to be more targeted at data system internals then at event data.
Databus is a system that provides a log-like overlay for database tables.
Akka is an actor framework for Scala. It has an add on, eventsourced, that provides persistence and journaling.
Samza is a stream processing framework we are working on at LinkedIn. It uses a lot of the ideas in this article as well as integrating with Kafka as the underlying log.
Storm is popular stream processing framework that integrates well with Kafka.
Spark Streaming is a stream processing framework that is part of Spark.
Summingbird is a layer on top of Storm or Hadoop that provides a convenient computing abstraction.
I try to keep up on this area so if you know of some things I've left out, let me know.
I leave you with this message: