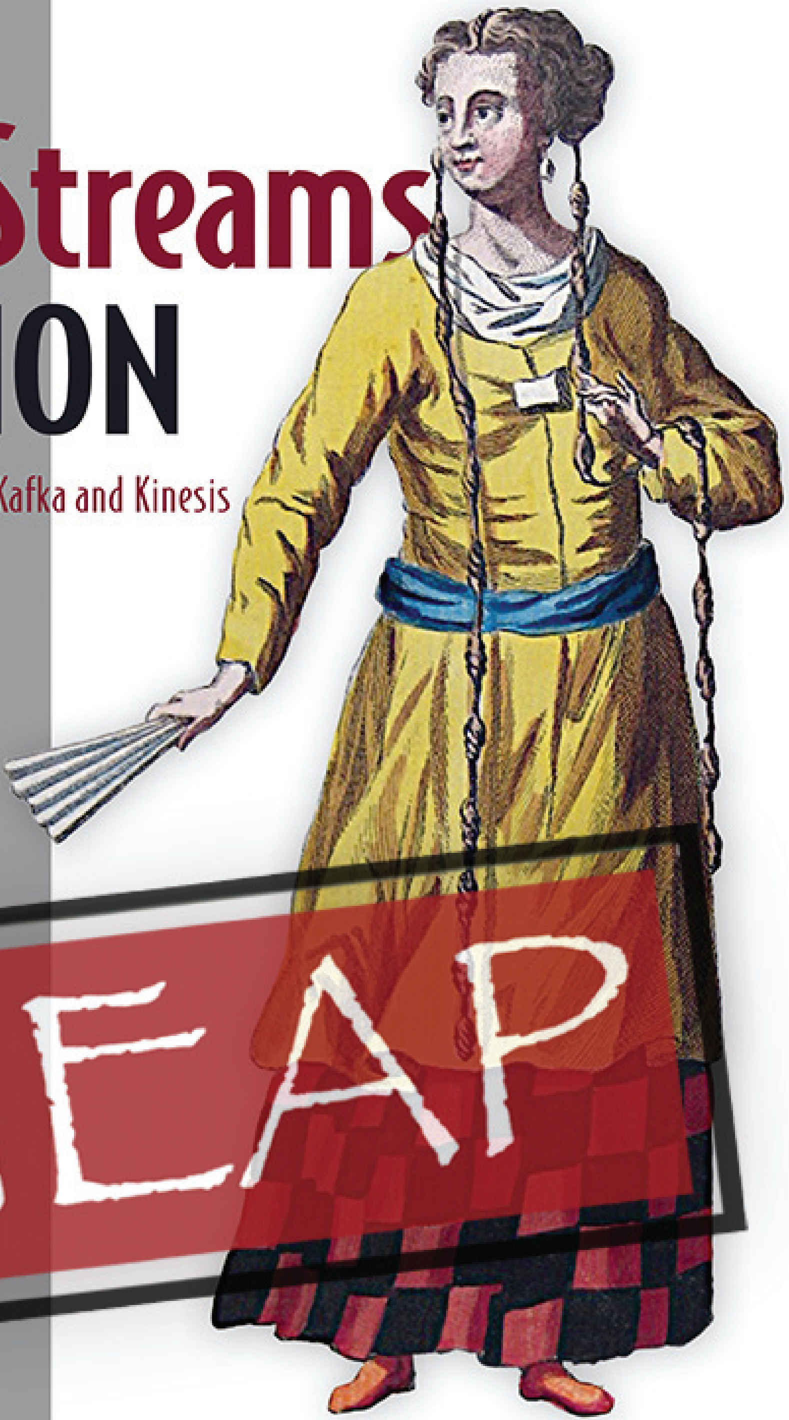
Event Streams IN ACTION

Unified log processing with Kafka and Kinesis

Alexander Dean



MANNING

MEAP Edition
Manning Early Access Program

Event Streams in Action

Unified log processing with Kafka and Kinesis

Version 14

welcome

Welcome to the MEAP for *Event Streams in Action: Unified log processing with Kafka and Kinesis*! *Event Streams in Action* is all about events: how to define events, how to send streams of events into unified log technologies like Apache Kafka and Amazon Kinesis, and how to write applications that process those event streams. We're going to cover a lot of ground in this book: Kafka and Kinesis, stream processing frameworks like Samza and Spark Streaming, event-friendly databases like Amazon Redshift and ElasticSearch, and more.

Mastering Event Streams will give you confidence to identify, model and process event streams wherever you find them - and I guarantee that by the end of this book, you will be seeing event streams everywhere! Above all, I hope that this MEAP and ultimately the book act as a springboard for a broader conversation about how as software engineers we should work with events; this a young but hugely important field and there is a lot still to discuss.

— Alexander Dean

brief contents

1 Introducing event streams

This chapter covers:

- Defining events and continuous event streams
- Some familiar event streams
- Unifying event streams with a unified log
- Use cases for a unified log

Believe it or not, a continuous stream of real-world and digital events already powers the company where you work. But it's unlikely that many of your co-workers think in those terms. Instead, they probably think about their work in terms of:

- The people or things that they interact with on a daily basis, e.g. customers, the Marketing team, code commits, or new product releases
- The software and hardware that they use to get stuff done
- Their own daily inbox of tasks to accomplish

People think and work in these terms because people are not computers. It is easy to get up in the morning and come to work because Sue in QA really needs those reports for her boss by lunchtime. If we stopped and started to think about our work as creating and responding to a continuous stream of events, we would probably go a little crazy – or at least call in to the office for a duvet day.

Computers don't have this problem – they would be comfortable with a definition of business that ran:

"A company is an organization which generates and responds to a continuous stream of events"

This definition is not going to win any awards from economists, but I believe that reframing your business in terms of a continuous stream of events offers huge benefits. Specifically, event streams enable:

- *Fresher insights* – a continuous stream of events represents the "pulse" of a business and makes a conventional batch-loaded data warehouse look stale in comparison
- *A single version of the truth* – ask several co-workers the same question and you may well get different answers, because they are working from different "pots" of data. Well-modeled event streams replace this confusion with a single version of the truth
- *Faster reactions* – automated near-real-time processing of continuous event streams allows a business to respond to those events within minutes or even seconds
- *Simpler architectures* – most businesses have built up a bird's nest of bespoke point-to-point connections between their various transactional systems. Event streams can help to unravel these messy architectures

Some of these benefits may not seem obvious now, but don't worry: in this chapter we will go back to first principles, starting with what we mean by events. I will introduce some simple examples of events, and then explain what a *continuous event stream* really is. There's a good chance you will find that you are pretty comfortable working with event streams already – you just hadn't thought of them in those terms.

Once we have looked at some familiar event streams, we will zoom out a level and look at how businesses' handling of events has evolved over the past 20 years. We will see that successive waves of technology have made things much more complex than they should be – but that a new architectural pattern called the *unified log* promises to simplify things again.

For these new approaches to reach the mainstream, they must be backed up with some compelling use cases. We will make the benefits of continuous event streams and the unified log significantly more real with a set of tangible real-world use cases, across a variety of industries.

1.1 Defining our terms

If you work in any kind of modern-day business, chances are that you have already worked with event streams in various forms, but have not been introduced to them as such. In this section, we will first present a simple definition for an event, and then explain how events combine into a continuous event stream.

1.1.1 Events

Before we can define a continuous event stream, we need to break out of Synonym City and concretely define a single *event*. Luckily the definition is simple: an event is anything that we can observe occurring at a particular point in time. That's it, *fin*. In Figure 1.1 we set out four example events from four different business sectors.

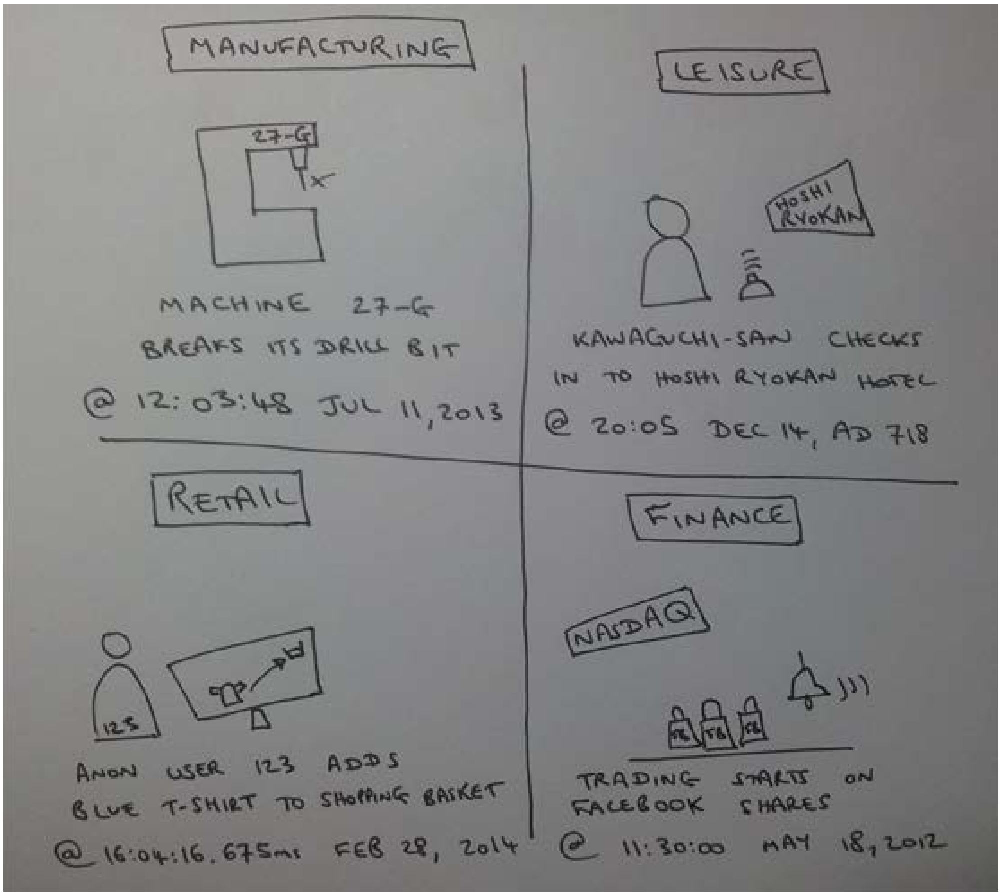

MANUFACTURING
27-G
MACHINE 27-G BREAKS ITS DRILL BIT
@ 12:03:48 JUL 11, 2013
LEISURE
HOSHI RYOKAN
KAWAGUCHI-SAN CHECKS IN TO HOSHI RYOKAN HOTEL
@ 20:05 DEC 14, AD 718
RETAIL
123
ANON USER 123 ADDS BLUE T-SHIRT TO SHOPPING BASKET
@ 16:04:16.675ms FEB 28, 2014
FINANCE
NASDAQ
FB
TRADING STARTS ON FACEBOOK SHARES
@ 11:30:00 MAY 18, 2012

Figure 1.1. The precision on the timestamps varies a little, but we can see that all four of these events are discrete, recordable occurrences that take place in the physical or digital worlds (or both).

It is easy to get carried away with the simplicity of this definition of an event – so before we go any further, let's clarify what is *not* an event. This is by no means an exhaustive list, but these are some of the more common mistakes to avoid. An event is not:

- A description of the ongoing state of something – the day was warm; the car was black; the API client was broken. However, the API client broke at noon on Tuesday is an event
- A recurring occurrence – *the NASDAQ opened at 09:30 every day in 2013*. However, each individual opening of the NASDAQ in 2013 *is* an event
- A collection of individual events – the Franco-Prussian war involved the Battle of Spicheren, the Siege of Metz and the Battle of Sedan. However, war was declared between France and Prussia on 19 July *1870* is an event

- A happening which spans a timeframe – *the 2014 Black Friday sale ran from 00:00:00 to 23:59:59 on November 28, 2014*. However, the beginning of the sale and the end of the sale *are* events

A general rule of thumb is: if the thing you are describing can be tied to a specific point in time, chances are that you are describing an event of some kind – even if it needs some verbal gymnastics to represent.

1.1.2 Continuous event streams

Now that we have defined what an event is, what is a *continuous event stream*? Simply put, a continuous event stream is an un-terminated succession of individual events, ordered by the point in time at which each event occurred. In Figure 1.1 we sketch out what a continuous event stream looks like at a high-level: we can see a succession of individual events, stepping forwards in time.

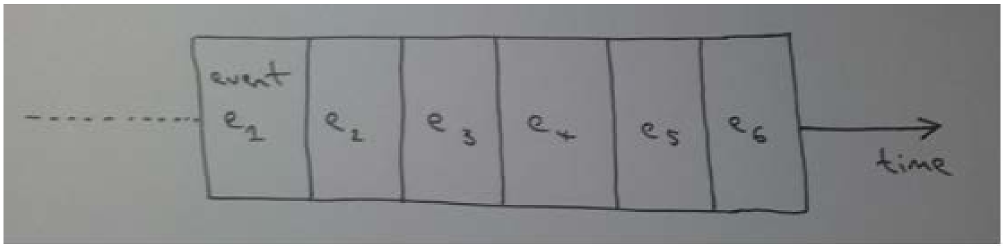


Figure 1.2. Anatomy of a continuous event stream: time is progressing left-to-right, and individual events are ordered within this timeframe. Note that the event stream is un-terminated – it can extend in both directions beyond our ability to process it.

We say that the succession of events is *un-terminated*, because:

- The start of the stream may pre-date our observing of the stream
- The end of the stream is at some unknown point in the future

To illustrate this, let's consider guests checking into Hoshi Ryokan hotel in Japan: Hoshi Ryokan is one of the oldest businesses in the world, having been founded in AD 718. Whatever stream of guest check-in events we could analyze for Hoshi Ryokan, we would know that the oldest guest check-ins are lost in the mists of time, and that future check-in events will continue to occur long after we have retired.

1.2 Some familiar event streams

If you read the previous section and thought that events and continuous event streams seemed familiar, then chances are that you have already worked with event streams, although they were likely not labeled as such. A huge number of software systems are heavily influenced by the idea of generating and responding to a continuous stream of events, including:

- *Transactional systems* – many of which respond to external events such as customers placing orders or suppliers delivering parts

- *Data warehouses* – which collect the event histories of other systems for later analysis, storing them in so-called fact tables
- *Systems monitoring* – which continually checks system- and application-level events coming from software or hardware systems to detect issues
- *Web analytics packages* – through which analysts can explore website visitors' on-site event stream to generate insights

In this section, we will take a brief tour through three very common areas of programming where the "event stream" concept is close to the surface. Hopefully this will make you think about part of your existing toolkit in a more event-centric way – but if all of these examples are unfamiliar to you, don't worry: there will be plenty of opportunities to master event streams from first principles later.

1.2.1 Application-level logging

Let's start with the event stream that almost all back-end (and many front-end) developers will be familiar with: application-level logging. If you have worked with Java, chances are that you have worked with Apache log4j at one time or another, but don't worry if not: its approach to logging is pretty similar to lots of other tools. Assuming the log4j.properties file is correctly configured and a static Logger is initialized, logging with log4j is pretty simple. In Listing 1.1 we set out some example simple examples of log messages a Java developer might add to their application.

Listing 1.1. Application logging with log4j

```
doSomethingInteresting();
log.info("Did something interesting");
doSomethingLessInteresting();
log.debug("Did something less interesting");

// Log output: #a
// INFO  2014-03-14 10:50:14,125 [Log4jExample_main]
"org.alexanderdean.Log4jExample": Did something interesting
// INFO  2014-03-14 10:55:34,345 [Log4jExample_main]
"org.alexanderdean.Log4jExample": Did something less interesting
```

#a The log output format assumes that we configured our log4j.properties file like so: log4j.appender.stdout.layout.ConversionPattern=%-5p %d [%t] %c: %m%n

You can see that application-level logging is generally used to record specific events at a point in time. The log events expressed in the code are deliberately quite primitive: consisting of just a "log level" which indicates the severity of the event, and a message string to describe the event. But log4j does add some additional metadata behind the scenes: in this case, the time of the event and the reporting thread and class name.

What happens to the log events after they are generated by your application? Best practice says that you write the log events to disk as log files, and then use some kind of log collection technology, such as Flume, fluentd or Scribe, to collect the log files from the individual servers and ingest them into some kind of systems monitoring or log file analysis tool. This event stream is illustrated in Figure 1.3.

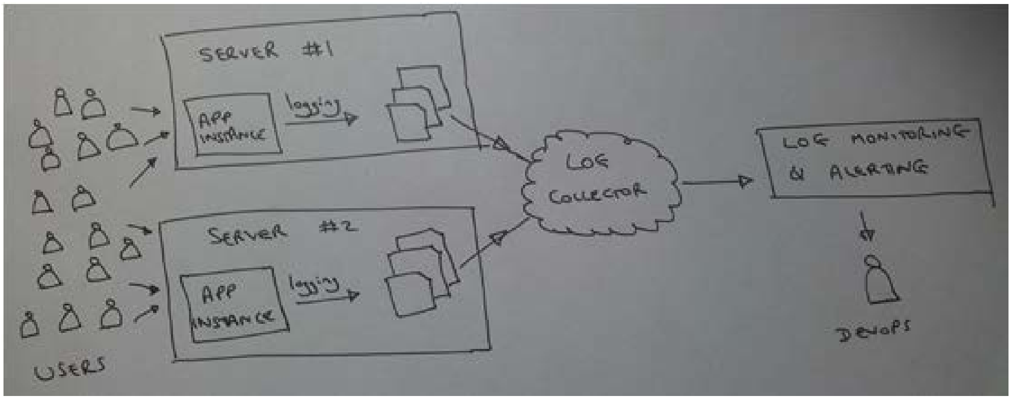


Figure 1.3. An application is running on two servers, with each application instance generating log messages. The log messages are written ("rotated") to disk before being collected and forwarded to a systems monitoring or log file analysis tool.

So application-level logging is clearly a continuous event stream, albeit one that leans heavily on schema-less messages which are often only human-readable. As the log4j example hints at, application-level logging is highly configurable, and also not well standardized across different languages and frameworks; working on a polyglot project, it can be very painful to standardize on a common log format across all software.

1.2.2 Web analytics

Let's move onto another example. If you are a front-end web developer, there's a good chance that you have embedded JavaScript tags in a website or web-app to provide some kind of web or event analytics. The most popular software in this category is Google Analytics, a Software-as-a-Service web analytics platform from Google; in 2012 Google released a new iteration of their analytics offering called Universal Analytics.

In Listing 1.2 we show some example JavaScript code used to instrument Universal Analytics. This code would either be embedded directly in the source code of the website, or invoked through a JavaScript tag manager. Either way, this code will run for each visitor to the website, generating a continuous stream of events representing each visitor's set of interactions with the website. These events flow back to Google, where they are stored, processed and displayed in a variety of different reports. The overall event flow is demonstrated in Figure 1.4.

Listing 1.2. Web tracking with Universal Analytics

```
<script>
(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){
  (i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new Date();a=s.createElement(o),
  m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)
  })(window,document,'script','//www.google-analytics.com/analytics.js','ga'); #a

  ga('create', 'UA-34290195-2', 'test.com'); #b
  ga('send', 'pageview'); #c
  ga('send', 'event', 'video', 'play', 'doge-video-01');

</script>
```

#a Initialization code for the Universal Analytics tracking tag
#b Create an event tracker for the given account, for the test.com website
#c Track the website visitor viewing this web page
#d Track the website visitor watching a video on this web page

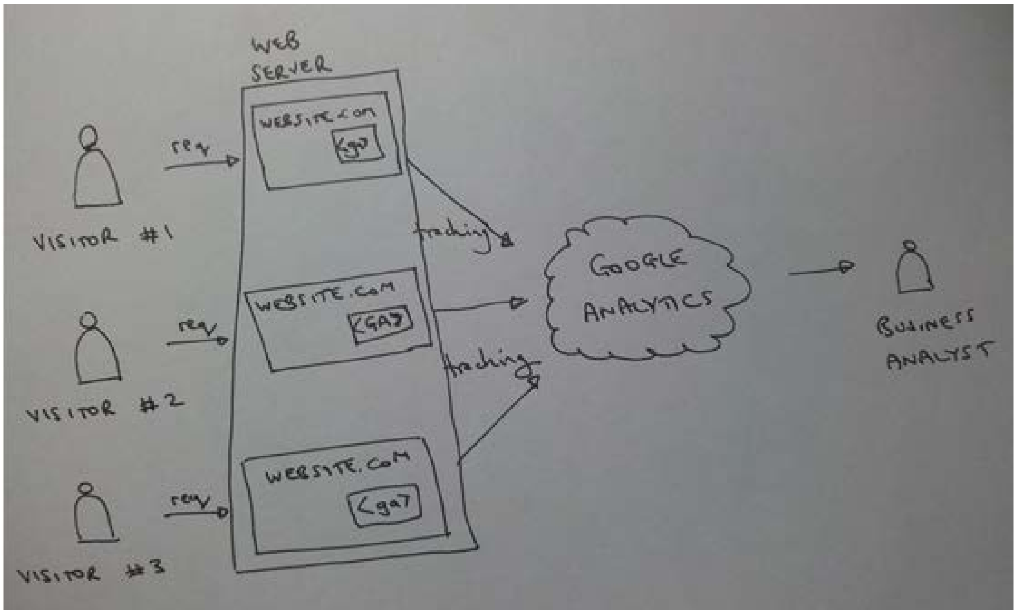
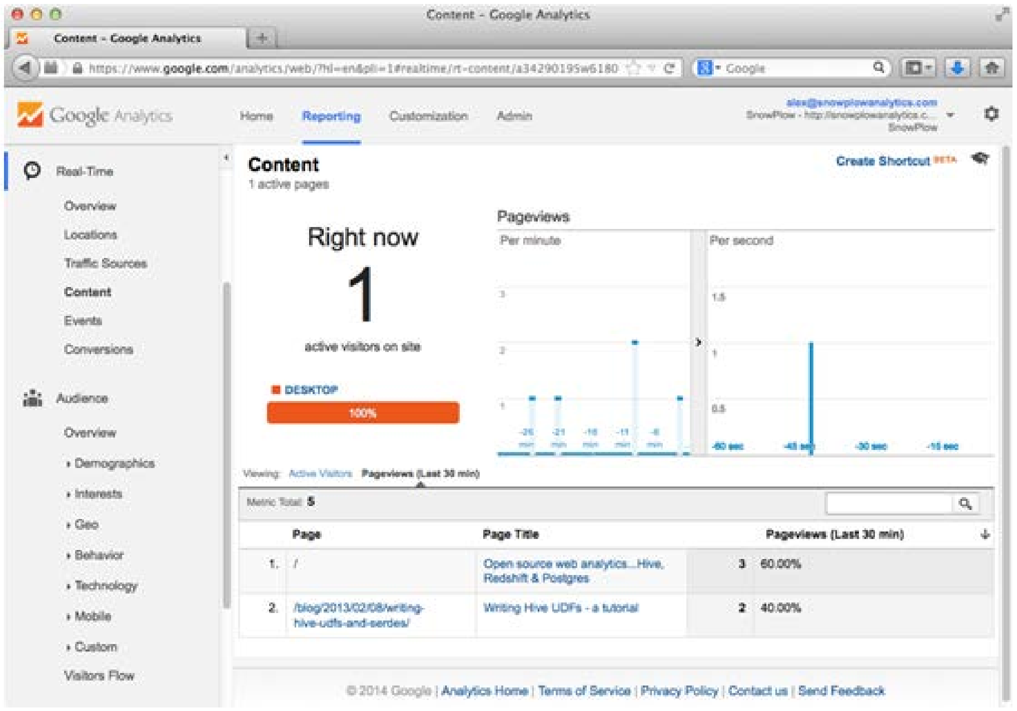


Figure 1.4. A JavaScript tracking tag sends visitors’ interactions with a website to Universal Analytics. This event stream is made available for analysis from within the Google Analytics user interface.

With Google Analytics deployed like this, a business analyst can log in to the Google Analytics web interface and start to make sense of the website’s event stream across all of its visitors. Figure 1.5 is a screenshot taken from Universal Analytics’ real-time dashboard, showing the previous 30 minutes’ worth of events occurring on the Snowplow Analytics website.



Viewing: Active Visitors Pageviews (Last 30 min)

Metric Total: 5

	Page	Page Title	Pageviews (Last 30 min)	
1.	/	Open source web analytics...Hive, Redshift & Postgres	3	60.00%
2.	/blog/2013/02/08/writing-hive-udfs-and-serdes/	Writing Hive UDFs - a tutorial	2	40.00%

© 2014 Google | Analytics Home | Terms of Service | Privacy Policy | Contact us | Send Feedback

Figure 1.5. Google Analytics is recording a real-time stream of events generated by website visitors. Bottom-right you can see the counts of views of individual webpages in the last 30 minutes.

1.2.3 Publish/subscribe messaging

Let's take a slightly lower-level example, but hopefully still one that many readers will be familiar with: application messaging, specifically in the publish/subscribe pattern. Publish/subscribe, sometimes shortened to pub/sub, is a simple way of communicating messages, whereby:

- Message senders publish messages which are associable with one or more topics
- Message receivers subscribe to specific topics, and then receive all messages associated with that topic

If you have worked with pub/sub messaging, there's a good chances that the messages you were sending were *events* of some form or another.

For a hands-on example let's try out NSQ, a popular distributed pub/sub messaging platform originally created by Bitly. NSQ brokering events between a single *publishing* app and two *subscribing* apps is illustrated in Figure 1.6.

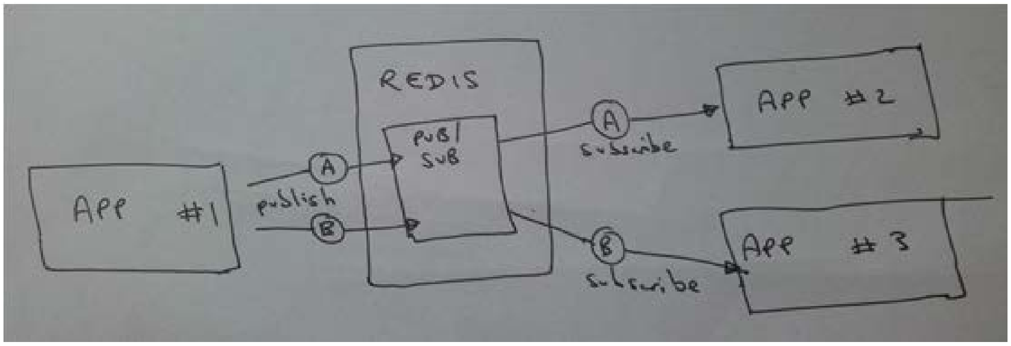


Figure 1.6. NSQ pub/sub is facilitating communication between App 1, which is publishing messages into a single Topic 1, and Apps 2 and 3 which are each subscribing to receive messages from Topic 1.

The nice thing about NSQ for demonstration purposes is that it is super-simple to install and setup. In OS X I opened up a new terminal, installed NSQ using Homebrew and then started up the nsqlookupd daemon:

```
$ brew install nsq
...
$ nsqlookupd
...
```

And then in a second terminal window I started the main NSQ daemon, nsqd:

```
$ nsqd --lookupd-tcp-address=127.0.0.1:4160
...
```

I left those two daemons running and then opened a third terminal. I used the nsqd daemon's HTTP API to create a new topic:

```
$ curl -X POST http://127.0.0.1:4151/topic/create\?topic\=Topic1
```

Next I was ready to create the two subscribers, Apps 2 and 3. In two further terminals, I started the `nsq_tail` app to simulate Apps 2 and 3 subscribing to Topic 1:

```
$ nsq_tail --lookupd-http-address=127.0.0.1:4161 \
  --topic=Topic1 --channel=App2
2015/10/15 20:53:10 INF    1 [Topic1/App2] querying nsqlookupd
      http://127.0.0.1:4161/lookup?topic=Topic1
2015/10/15 20:53:10 INF    1 [Topic1/App2] (Alexanders-MacBook-Pro.local:4150)
      connecting to nsqd
```

And my fifth and final terminal:

```
$ nsq_tail --lookupd-http-address=127.0.0.1:4161 \
  --topic=Topic1 --channel=App2
2015/10/15 20:57:55 INF    1 [Topic1/App3] querying nsqlookupd
      http://127.0.0.1:4161/lookup?topic=Topic1
2015/10/15 20:57:55 INF    1 [Topic1/App3] (Alexanders-MacBook-Pro.local:4150)
```

```
      connecting to nsqd
```

Returning to my third terminal (the only one not running a daemon) I sent in some events, again using the HTTP API:

```
$  curl -d 'checkout' 'http://127.0.0.1:4151/put?topic=Topic1'
OK%
$ curl -d 'ad_click' 'http://127.0.0.1:4151/put?topic=Topic1'
OK%
$  curl -d 'save_game' 'http://127.0.0.1:4151/put?topic=Topic1'
OK%
```

Check back in our tailing terminals to see the events arriving:

```
2015/10/15 20:59:06 INF    1 [Topic1/App2] querying nsqlookupd
      http://127.0.0.1:4161/lookup?topic=Topic1
checkout
ad_click
save_game
```

And the same for App 3:

```
2015/10/15 20:59:08 INF    1 [Topic1/App3] querying nsqlookupd
      http://127.0.0.1:4161/lookup?topic=Topic1
checkout
ad_click
save_game
```

So in this pub/sub architecture we have events being published by one application and being subscribed to by two other applications; simply add more events and again you have a continuous event stream being processed.

Hopefully the examples in this section have shown you how the concept of the "event stream" is a familiar one, underpinning disparate systems and approaches include application logging, web analytics and publish/subscribe messaging. The terminology may be different, but in all three examples we see the same building blocks: some structure or schema of event (even if extremely minimal); a way of generating these events; and a way of collecting and subsequently processing these events.

1.3 Unifying continuous event streams

So far in this chapter I have introduced the idea of event streams, defined our terms and also highlighted some familiar technologies that make use of event streams in one form or another. This usage is a good start, but hopefully you can see that these technologies are highly fragmented: their evented nature is poorly understood, their event schemas are unstandardized, and their use cases are trapped in separate silos. In this section I will introduce a much more radical – and powerful – approach to using continuous event streams for your business.

Simply put, the argument of this book is that every digital business should be re-structured around a process which:

- accretes events from disparate source systems,
- stores them in what we call a “unified log”, and
- enables data processing applications to operate on these event streams

This is a bold statement – and one that sounds like a lot of work to implement! What evidence do we have that this is a practical and useful course of action for a business?

In this section we will map out the historical and ongoing evolution of business data processing, extending up to continuous event streams and this unified log. We have split this evolution into two distinct eras that we have both lived through and experienced first-hand, plus a third era that is soon approaching:

1. The classic era - the pre-big data, pre-SaaS era of operational systems and batch-loaded data warehouses
2. The hybrid era - today’s hotchpotch of different systems and approaches
3. The unified era – an emerging architecture, enabled by processing continuous event streams in a unified log

1.3.1 The classic era

In the classic era, businesses primarily operated a disparate set of on-premise transactional systems, feeding into a data warehouse; this architecture is illustrated in Figure 1.7. Each of the transactional systems would feature:

- An internal “local loop” for near-real-time data processing
- Its own data silo
- Where necessary, point-to-point connections to peer systems (e.g. via APIs or feed import/exports)

A data warehouse would be added to give the Management team a much-needed view across these transactional systems. This data warehouse would typically be fed from the transactional systems overnight by a set of batch ETL (extract, transform, load) processes. This data warehouse provided the business with a single version of the truth, with full data history and wide data coverage. Internally, it was often constructed following the star schema style of fact and dimension tables, as popularized by Ralph Kimball.

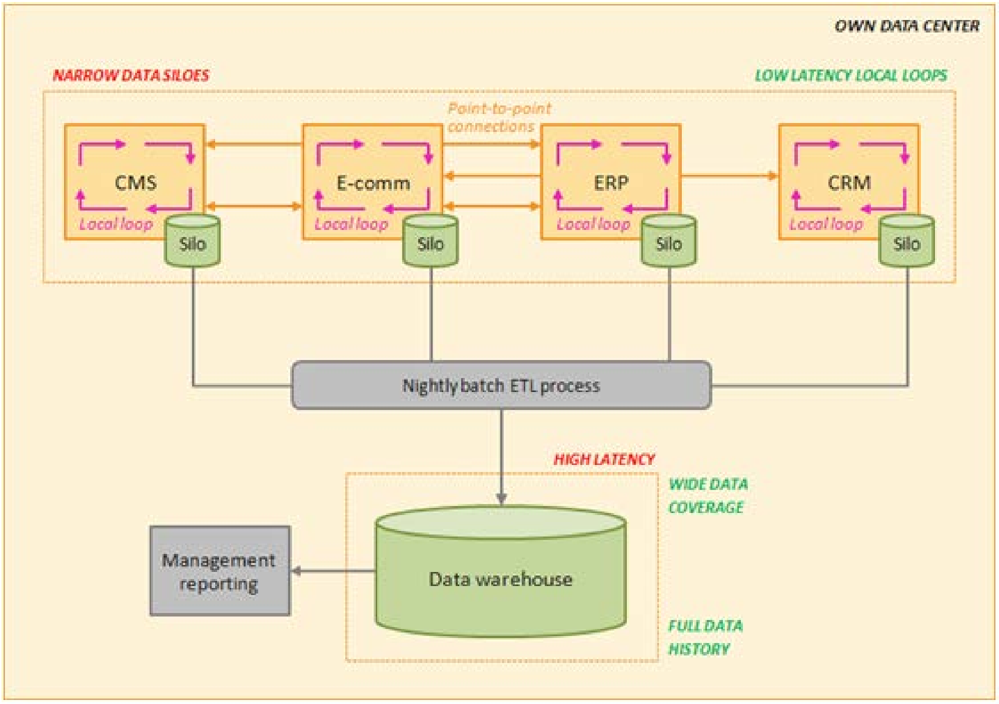


Figure 1.7. This retailer has four transactional systems, each with their own data silo. These systems are connected to each other as necessary with point-to-point connections. A nightly batch ETL process *extracts* data out of the data siloes, *transforms* it for reporting purposes and then *loads* it into a data warehouse. Management reports are then based on the contents of the data warehouse.

Although we call this the classic era, in truth many businesses still run on a close descendant of this approach, albeit with more SaaS platforms mixed in. This is a tried and tested architecture, although one with serious pain points:

- *High latency for reporting* – the time taken from an event occurring to the event appearing in management reporting is counted in the hours (potentially even days), not seconds
- *Point-to-point spaghetti* – extra transactional systems mean even more point-to-point connections, as illustrated in Figure 1.8. This point-to-point spaghetti is expensive to build and maintain and increases the overall fragility of the system
- *Schema woes* – classic data warehousing assumes that each business has an intrinsic data model that can be mined from the state stored in its transactional systems. This is a highly flawed assumption, as we explore in chapter X

Faced with these issues, businesses have made the leap to a new model – particularly businesses in fast-moving sectors like retail, technology and media. We might call this new model the hybrid era.

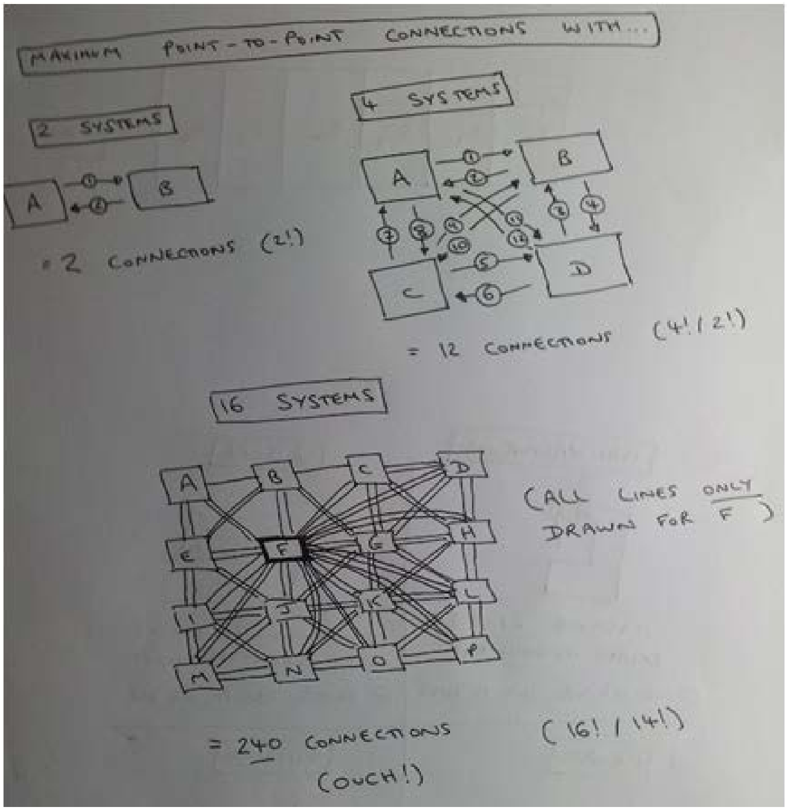

MAXIMUM POINT-TO-POINT CONNECTIONS WITH...
2 SYSTEMS
A
B
= 2 CONNECTIONS (2!)
4 SYSTEMS
A
B
C
D
= 12 CONNECTIONS (4! / 2!)
16 SYSTEMS
(ALL LINES ONLY DRAWN FOR F)
= 240 CONNECTIONS (16! / 14!)
(OUCH!)

Figure 1.8. The maximum number of point-to-point connections possibly required between 2, 4 and 16 software systems is 2, 12 and 240 connections respectively. Adding systems grows the number of point-to-point connections quadratically.

1.3.2 The hybrid era

The hybrid era is characterized by companies operating a hotchpotch of different transactional and analytics systems - some on-premise packages, some from SaaS vendors, plus some home-grown systems. See Figure 1.9 for an example of a hybrid-era architecture.

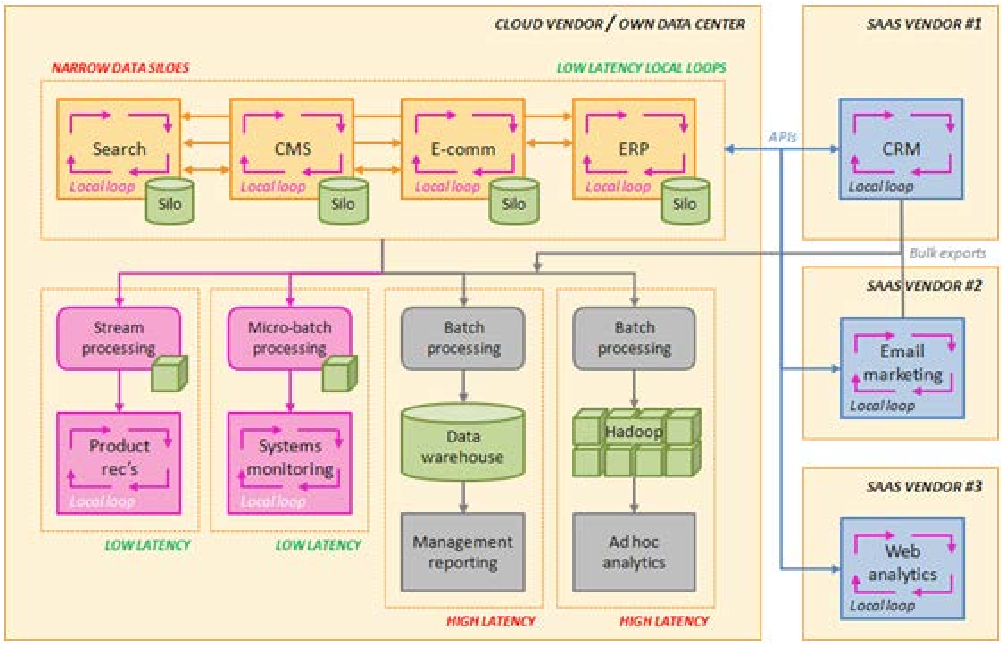


Figure 1.9. Compared to the classic era, our retailer has now added external SaaS dependencies, Hadoop as a new high-latency "log everything" platform and new low-latency data pipelines for use cases like systems monitoring and product recommendations.

It is hard to generalize what these hybrid architectures look like - again they have strong local loops and data silos, but there are also attempts at "log everything" approaches with Hadoop and/or systems monitoring. There tends to be a mix of near-real-time processing for narrow analytics use cases like product recommendations, plus separate batch processing efforts into Hadoop as well as a classic data warehouse. Hybrid architectures also feature attempts to bulk export data from external SaaS vendors for warehousing, and efforts to feed these external systems with proprietary data through these systems' own APIs.

Although it is obvious that this hybrid approach delivers capabilities sorely lacking from the classic approach, it brings its own problems:

- *No single version of the truth* - data is now warehoused in multiple places, depending on the data volumes and the analytics latency required. There is no system which has 100% visibility
- *Decisioning has become fragmented* - the number of local systems loops, each operating on silo'ed data, has grown since the classic era. These loops represent a highly fragmented approach to making near-real-time decisions from data
- *Point-to-point connections have proliferated* - as the number of systems has grown, the number of point-to-point connections has exploded. Many of these connections are

fragile or incomplete; getting sufficiently-granular and timely data out of external SaaS systems is particularly challenging

- *Analytics can have low latency or wide data coverage, but not both* - where stream processing is selected for low latency, it becomes effectively another local processing loop. The warehouses aim for much wider data coverage, but at the cost of duplication of data and high latency

1.3.3 The unified era

These two eras bring us up to the present day, and the emerging unified era of data processing. The key innovation in business terms is putting a unified log at the heart of all of our data collection and processing. A unified log is an append-only log to which we write all events generated by our applications. Going further, the unified log:

- can be read from at low latency
- is readable by multiple applications simultaneously, with different applications able to consume from the log at their own pace
- only holds a rolling window of events – probably a week or a month's worth. But we can archive the historic log data in HDFS or Amazon S3

For now, don't worry about the mechanics of the unified log – we will discuss this in much more detail in Chapter Two. For now, it is more important to understand how the unified log can re-shape how data flows through a business. We have updated our retailer's architecture to the unified era in Figure 1.10. The new architecture is guided by two simple rules:

1. All software systems can and should write their individual continuous event streams to the unified log – even third-party SaaS vendors can emit events via webhooks and streaming APIs
2. Unless very low latency or transactional guarantees are required, software systems should communicate with each other in an un-coupled way through the unified log, not via point-to-point connections

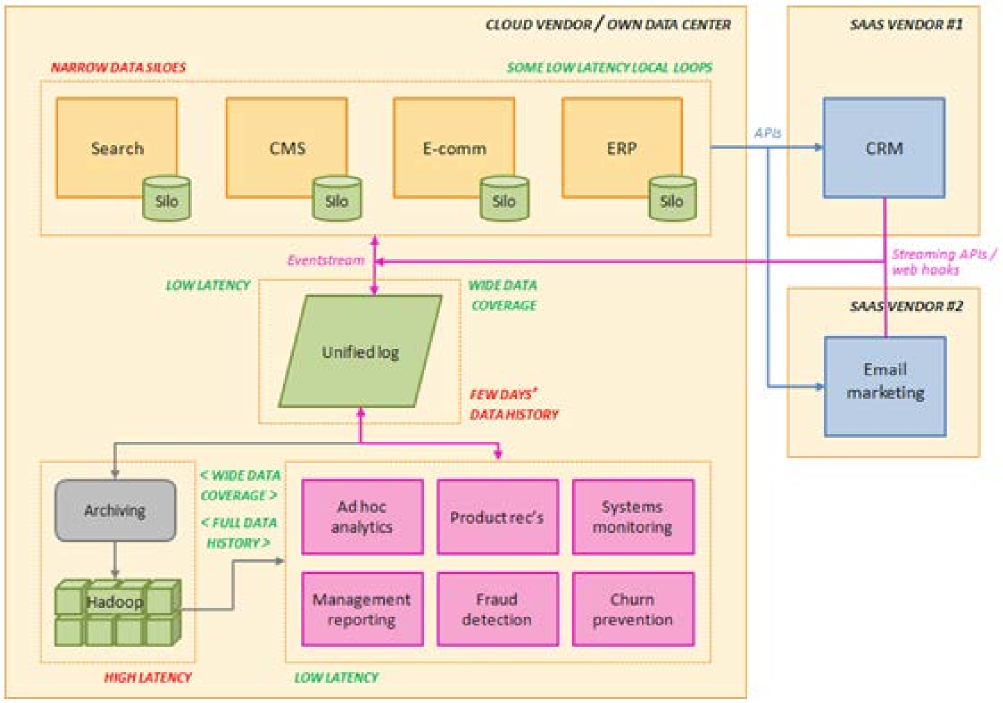


Figure 1.10. Our retailer has now re-architected around a unified log, plus a longer-term archive of events in Hadoop. The data architecture is now much simpler, with far fewer point-to-point connections, and all of our analytics and decision-making systems now working off a single version of the truth.

A few advantages should be clear compared to one or both of the previous architectures:

- *We have a single version of the truth* – together, the unified log plus Hadoop archive represent our single version of the truth. They contain exactly the same data - our event stream - they just have different time windows of data
- *The single version of the truth is upstream from the data warehouse* – in the classic era, the data warehouse provided the single version of the truth, making all reports generated from it consistent. In the unified era, the log provides the single version of the truth: as a result, operational systems (e.g. recommendation and ad targeting systems) compute on the same truth as analysts producing management reports
- *Point-to-point connections have largely been unravelled* - in their place, applications can append to the unified log and other applications can read their writes. This is illustrated in Figure 1.11.
- *Local loops have been unbundled* - in place of local silos, applications can collaborate on near-real-time decision-making via the unified log

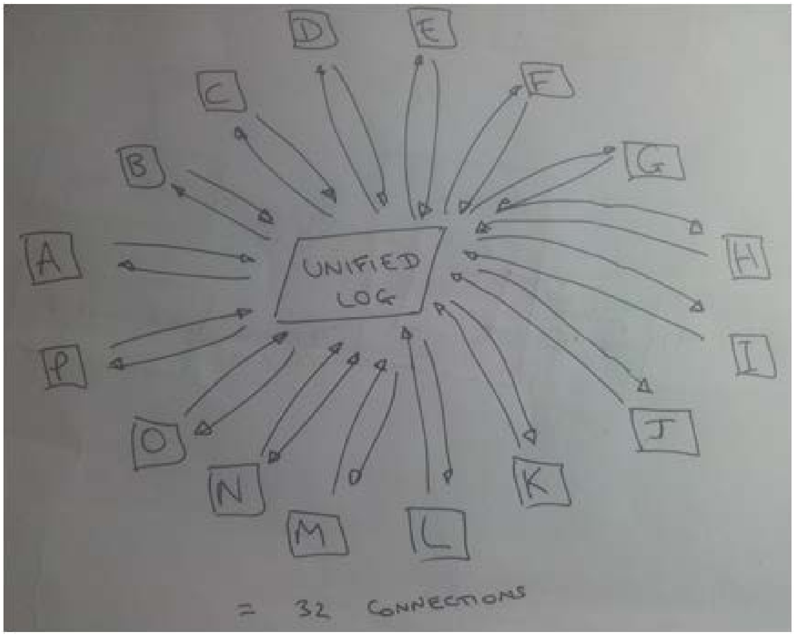


Figure 1.11. Our unified log acts as the "glue" between all of our software systems. In place of the proliferation of point-to-point connections seen prior, we now have systems reading and writing to the unified log. Conceptually we now have a maximum of 32 unidirectional connections, compared to 240 for a point-to-point approach.

1.4 Use cases for the unified log

You may have read through the last section and thought: continuous event streams and the unified log look all well and good, but they seem like an architectural optimization, not something that enables wholly new applications. In fact this is both a significant architectural improvement on previous approaches, and an enabler for powerful new use cases. In this section, we will whet your appetite with three of these use cases.

1.4.1 Customer feedback loops

One of the most exciting use cases of continuous data processing is the ability to respond to an individual's customer behavior while that customer is *still engaged with your service*. These real-time feedback loops will look a little different depending on what industry you are in – here are just a few examples:

- *Retail* – whenever the customer looks like they are about to abandon their shopping cart, pop-up a coupon in their web browser to coax them into checking out. See Figure 1.12 for an example
- *TV* – adjust the electronic program guide in real-time based on the viewer's current behavior and historical watching patterns, to maximize their viewing hours

- *Automotive* – detect abnormal driving patterns and notify the owner that the car may have been stolen
- *Gaming* – if a player is finding a four-player co-operative game too challenging, adjust the difficulty level to prevent them quitting and spoiling the game for the other players

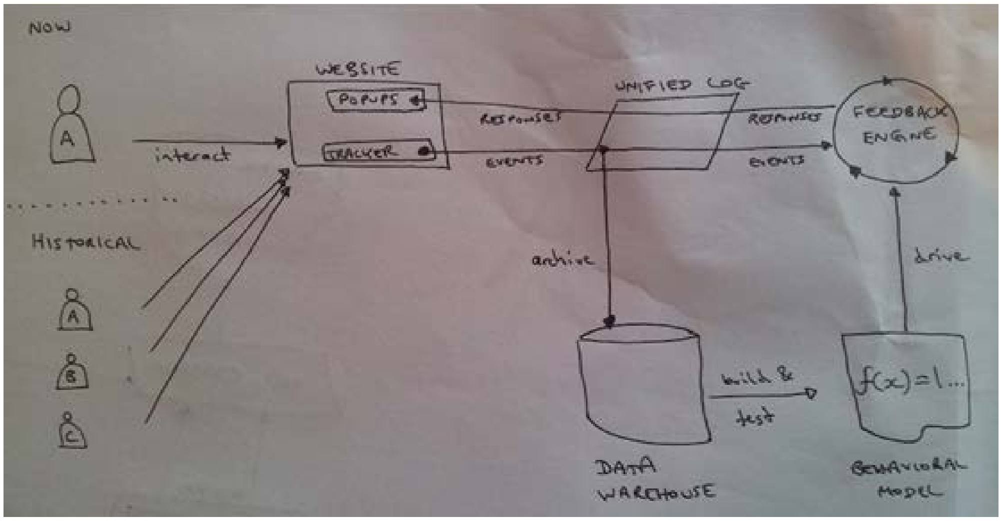


Figure 1.12. We can use a unified log to respond to customers in an e-commerce environment. The event stream of customer behavior is fed to the unified log and onwards to the data warehouse for analysis. A behavioral model is constructed using the historic data, and then this is used to drive a feedback engine. The feedback engine communicates back to the site again via the unified log.

Customer feedback loops are not new – even in the classic era, you could capture user behavior and use that to inform some kind of personalized mail-out or email drip marketing. And there are startups today that will put a JavaScript tag in your website, track a user in real-time and attempt to alter their behavior with banners, flash messages and popups. But the feedback loops enabled by a unified log are much more powerful:

- They are fully under the control of the service provider, not a third party such as a SaaS analytics provider, meaning that these loops can be programmed to whatever algorithm makes most sense to the business itself
- They are driven off models tested across the full archive of the exact same event stream
- Customers' reactions to the feedback loops can be added into event stream as well, making machine learning approaches possible

1.4.2 Holistic systems monitoring

Robust monitoring of software and services is painful because the signals available to detect (or even pre-empt) issues are so fragmented:

- Server monitoring is fed into a third-party service or a self-hosted time-series database. This data is often pre-aggregated or pre-sampled because of storage and network concerns
- Application log messages are written to the application servers as log files and hopefully collected before the server is killed or shutdown
- Customer events are sent to a third-party service and frequently not available for granular customer- or instance-level inspection

With a unified log, any systems issue can be investigated by exploring *any* of the event stream data held in the unified log. The data does not have to be stored in the unified log specifically for systems monitoring purposes – the systems administrator can explore any of that data to identify correlations, perform root cause analysis and so on. See Figure 1.13 for an example of holistic systems monitoring for a mobile app business.

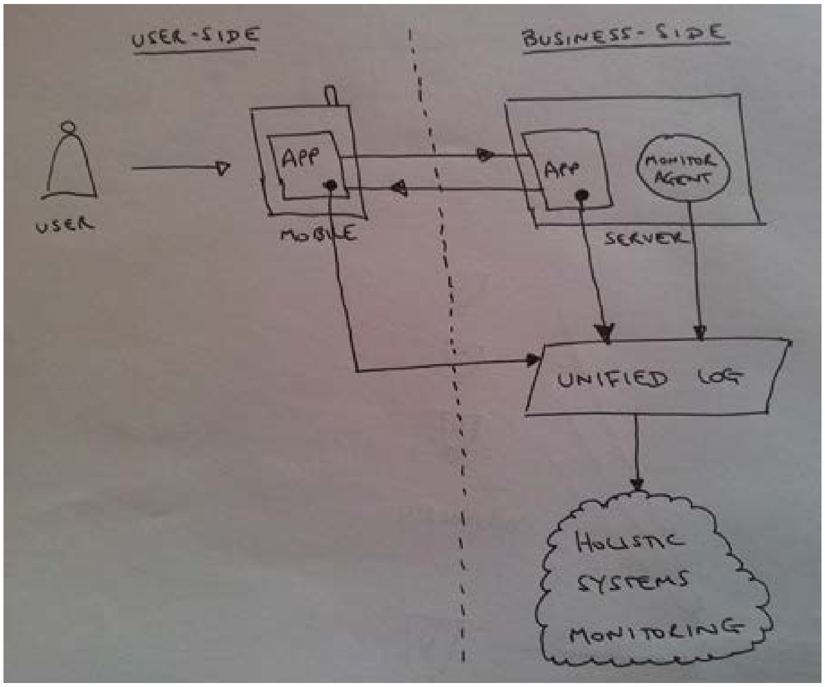


Figure 1.13. The unified log can power holistic systems monitoring for a business operating a mobile app with a client-server architecture. Events are sent to the unified log from the mobile client, the server application and the server's monitoring agent; if any problem occurs, the systems administrator can work with the application developers to identify the issue looking at the whole event stream.

1.4.3 Hot swapping data application versions

We mentioned earlier that a unified log is readable by multiple applications simultaneously, and that each application can read events at its own pace. In other words, each application using the unified log can independently keep track of which events it has already processed, and which are next to process.

If we can have multiple applications reading from the unified log, then it follows that we can also have multiple versions of the *same* application processing events from the unified log. This is hugely useful as it allows us to allow us to “hot swap” our data processing applications – in other words, to upgrade our applications without taking them offline. While the current version of our application is still running, we can:

1. Kick off the new version of the application from the start of the unified log
2. Let it catch up to the same cursor position as the old version
3. Switch our users over to the new version of the application
4. Shut down the old version

This hot swapping of our old application version with our new version is illustrated in Figure 1.14.

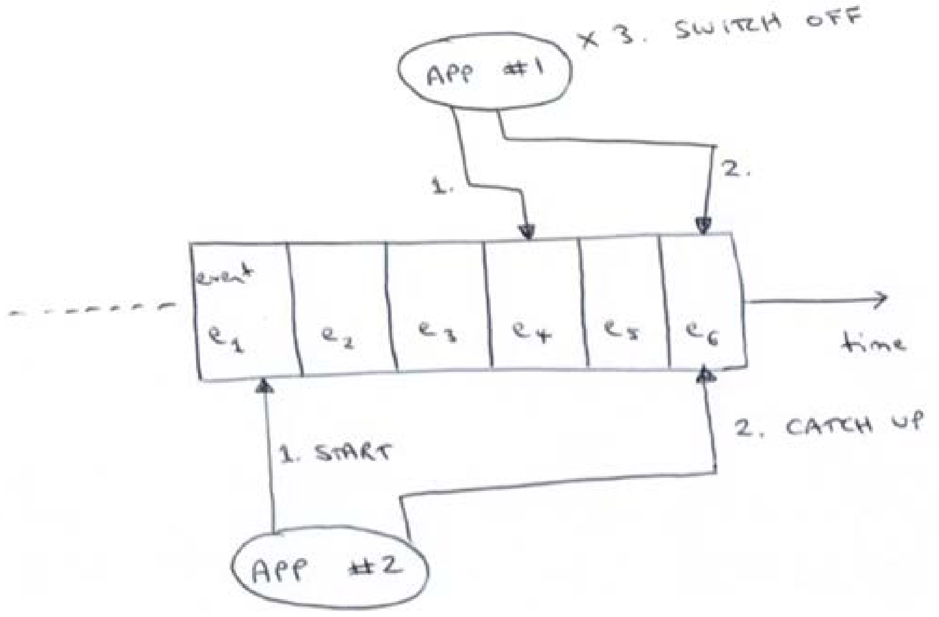


Figure 1.14. The unified log allows us to “hot swap” two different versions of the same application. First we kick off the new application version processing from the start of our unified log, then we let the new version catch up to the old version’s cursor position, then finally we switch off the old version.

The ability for each application (or application version) to maintain its own cursor position is incredibly powerful. As well as the ability to upgrade live data processing applications without service interruptions, we can also use this capability to:

- Test new versions of our applications against the live event stream
- Compare the results of different algorithms against the same event stream
- Have different users consume different versions of our applications

1.5 Conclusion

- An event is anything that we can observe occurring at a particular point in time
- A continuous event stream is an un-terminated succession of individual events, ordered by the point in time at which each event occurred
- Many software systems are heavily influenced by the idea of event streams, including application-level logging, web analytics and publish/subscribe messaging
- In the classic era of data processing, businesses operated a disparate set of on-premise systems, feeding into a data warehouse. These systems featured high data latency, heavily silo'ed data and many point-to-point connections between systems
- In the hybrid era of data processing, businesses operate a hotchpotch of different transactional and analytics systems. There are disparate data silos, but also attempts at "log everything" approaches with Hadoop and/or systems monitoring
- The unified log era proposes that businesses restructure around an append-only log to which we write all events generated by our applications; software systems should communicate with each other in an un-coupled way through the unified log
- "Hero" use cases for this unified architecture include customer feedback loops, holistic systems monitoring and hot swapping data application versions