O'REILLY®

Compliments of confluent

Making Sense of Stream Processing

The Philosophy Behind Apache Kafka and Scalable Stream Data Platforms

Martin Kleppmann

Compliments of

DOWNLOAD ▸▸
Apache Kafka and **Confluent Platform**

confluent

Making Sense of Stream Processing

The Philosophy Behind Apache Kafka and Scalable Stream Data Platforms

Martin Kleppmann

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Making Sense of Stream Processing
by Martin Kleppmann

Copyright © 2016 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Shannon Cutt
Production Editor: Melanie Yarbrough
Copyeditor: Octal Publishing
Proofreader: Christina Edwards

Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

March 2016: First Edition

Revision History for the First Edition
2016-03-04: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. Making Sense of Stream Processing, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-94010-5

[LSI]

Table of Contents

Foreword v

Preface vii

1. Events and Stream Processing 1
Implementing Google Analytics: A Case Study 3
Event Sourcing: From the DDD Community 9
Bringing Together Event Sourcing and Stream Processing 14
Using Append-Only Streams of Immutable Events 27
Tools: Putting Ideas into Practice 31
CEP, Actors, Reactive, and More 34

2. Using Logs to Build a Solid Data Infrastructure 39
Case Study: Web Application Developers Driven to Insanity 40
Making Sure Data Ends Up in the Right Places 52
The Ubiquitous Log 53
How Logs Are Used in Practice 54
Solving the Data Integration Problem 72
Transactions and Integrity Constraints 74
Conclusion: Use Logs to Make Your Infrastructure Solid 76
Further Reading 79

3. Integrating Databases and Kafka with Change Data Capture 81
Introducing Change Data Capture 81
Database = Log of Changes 83
Implementing the Snapshot and the Change Stream 85

Bottled Water: Change Data Capture with PostgreSQL and Kafka 86
The Logical Decoding Output Plug-In 96
Status of Bottled Water 100

4. The Unix Philosophy of Distributed Data. 101
Simple Log Analysis with Unix Tools 101
Pipes and Composability 106
Unix Architecture versus Database Architecture 110
Composability Requires a Uniform Interface 117
Bringing the Unix Philosophy to the Twenty-First Century 120

5. Turning the Database Inside Out. 133
How Databases Are Used 134
Materialized Views: Self-Updating Caches 153
Streaming All the Way to the User Interface 165
Conclusion 170

Foreword

Whenever people are excited about an idea or technology, they come up with buzzwords to describe it. Perhaps you have come across some of the following terms, and wondered what they are about: "stream processing", "event sourcing", "CQRS", "reactive", and "complex event processing".

Sometimes, such self-important buzzwords are just smoke and mirrors, invented by companies that want to sell you their solutions. But sometimes, they contain a kernel of wisdom that can really help us design better systems.

In this report, Martin goes in search of the wisdom behind these buzzwords. He discusses how event streams can help make your applications more scalable, more reliable, and more maintainable. People are excited about these ideas because they point to a future of simpler code, better robustness, lower latency, and more flexibility for doing interesting things with data. After reading this report, you'll see the architecture of your own applications in a completely new light.

This report focuses on the architecture and design decisions behind stream processing systems. We will take several different perspectives to get a rounded overview of systems that are based on event streams, and draw comparisons to the architecture of databases, Unix, and distributed systems. Confluent, a company founded by the creators of Apache Kafka, is pioneering work in the stream processing area and is building an open source stream data platform to put these ideas into practice.

For a deep dive into the architecture of databases and scalable data systems in general, see Martin Kleppmann's book "Designing Data-Intensive Applications," available from O'Reilly.

—*Neha Narkhede*, Cofounder and CTO, Confluent Inc.

Preface

This report is based on a series of conference talks I gave in 2014/15:

- "Turning the database inside out with Apache Samza," at Strange Loop, St. Louis, Missouri, US, 18 September 2014.
- "Making sense of stream processing," at /dev/winter, Cambridge, UK, 24 January 2015.
- "Using logs to build a solid data infrastructure," at Craft Conference, Budapest, Hungary, 24 April 2015.
- "Systems that enable data agility: Lessons from LinkedIn," at Strata + Hadoop World, London, UK, 6 May 2015.
- "Change data capture: The magic wand we forgot," at Berlin Buzzwords, Berlin, Germany, 2 June 2015.
- "Samza and the Unix philosophy of distributed data," at UK Hadoop Users Group, London, UK, 5 August 2015

Transcripts of those talks were previously published on the Confluent blog, and video recordings of some of the talks are available online. For this report, we have edited the content and brought it up to date. The images were drawn on an iPad, using the app "Paper" by FiftyThree, Inc.

Many people have provided valuable feedback on the original blog posts and on drafts of this report. In particular, I would like to thank Johan Allansson, Ewen Cheslack-Postava, Jason Gustafson, Peter van Hardenberg, Jeff Hartley, Pat Helland, Joe Hellerstein, Flavio Junqueira, Jay Kreps, Dmitry Minkovsky, Neha Narkhede, Michael Noll, James Nugent, Assaf Pinhasi, Gwen Shapira, and Greg Young for their feedback.

Thank you to LinkedIn for funding large portions of the open source development of Kafka and Samza, to Confluent for sponsoring this report and for moving the Kafka ecosystem forward, and to Ben Lorica and Shannon Cutt at O'Reilly for their support in creating this report.

—Martin Kleppmann, January 2016

CHAPTER 1
Events and Stream Processing

The idea of structuring data as a stream of events is nothing new, and it is used in many different fields. Even though the underlying principles are often similar, the terminology is frequently inconsistent across different fields, which can be quite confusing. Although the jargon can be intimidating when you first encounter it, don't let that put you off; many of the ideas are quite simple when you get down to the core.

We will begin in this chapter by clarifying some of the terminology and foundational ideas. In the following chapters, we will go into more detail of particular technologies such as Apache Kafka[1] and explain the reasoning behind their design. This will help you make effective use of those technologies in your applications.

Figure 1-1 lists some of the technologies using the idea of event streams. Part of the confusion seems to arise because similar techniques originated in different communities, and people often seem to stick within their own community rather than looking at what their neighbors are doing.

1 "Apache Kafka," Apache Software Foundation, kafka.apache.org.

Figure 1-1. Buzzwords related to event-stream processing.

The current tools for distributed stream processing have come out of Internet companies such as LinkedIn, with philosophical roots in database research of the early 2000s. On the other hand, *complex event processing* (CEP) originated in event simulation research in the 1990s[2] and is now used for operational purposes in enterprises. Event sourcing has its roots in the *domain-driven design* (DDD) community, which deals with enterprise software development—people who have to work with very complex data models but often smaller datasets than Internet companies.

My background is in Internet companies, but here we'll explore the jargon of the other communities and figure out the commonalities and differences. To make our discussion concrete, I'll begin by giving an example from the field of *stream processing*, specifically analytics. I'll then draw parallels with other areas.

2 David C Luckham: "Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events," Stanford University, Computer Systems Laboratory, Technical Report CSL-TR-96-705, September 1996.

Implementing Google Analytics: A Case Study

As you probably know, Google Analytics is a bit of JavaScript that you can put on your website, and that keeps track of which pages have been viewed by which visitors. An administrator can then explore this data, breaking it down by time period, by URL, and so on, as shown in Figure 1-2.

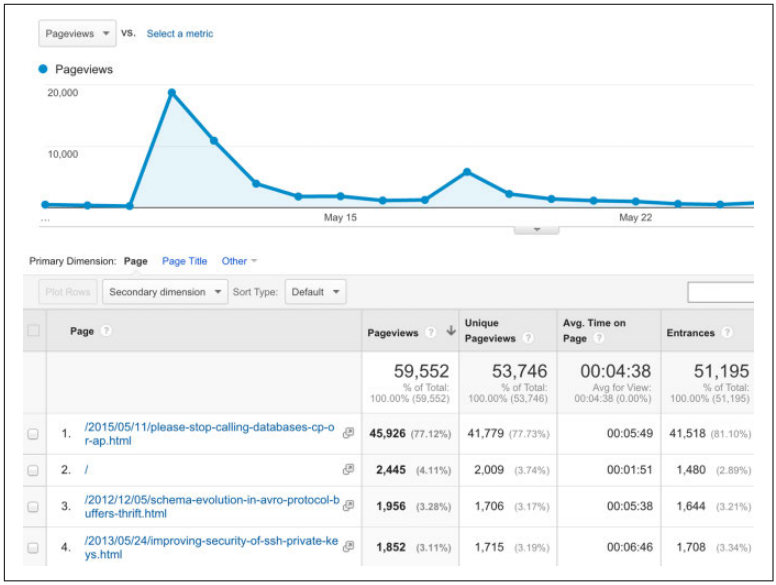


Figure 1-2. Google Analytics collects events (page views on a website) and helps you to analyze them.

How would you implement something like Google Analytics? First take the input to the system. Every time a user views a page, we need to log an event to record that fact. A page view event might look something like the example in Figure 1-3 (using a kind of pseudo-JSON).

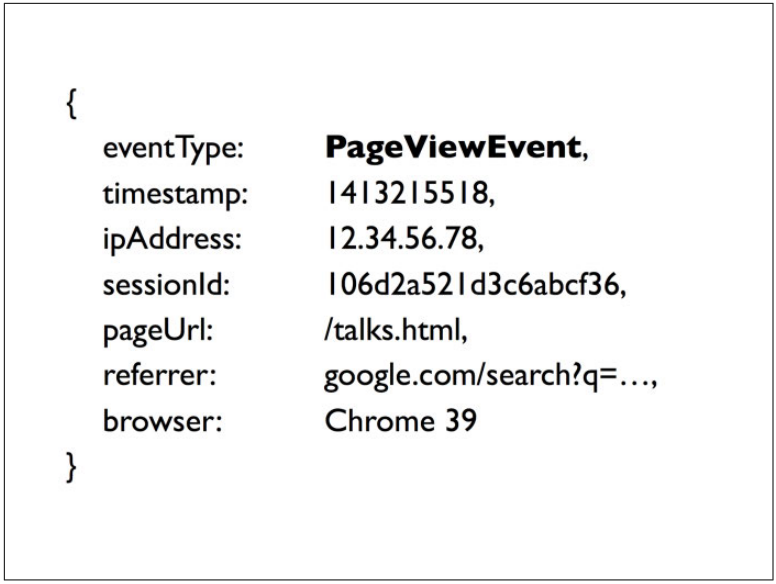


Figure 1-3. An event that records the fact that a particular user viewed a particular page.

A page view has an event type (PageViewEvent), a Unix timestamp that indicates when the event happened, the IP address of the client, the session ID (this may be a unique identifier from a cookie that allows you to figure out which series of page views is from the same person), the URL of the page that was viewed, how the user got to that page (for example, from a search engine, or by clicking a link from another site), the user's browser and language settings, and so on.

Note that each page view event is a simple, immutable fact—it simply records that something happened.

Now, how do you go from these page view events to the nice graphical dashboard on which you can explore how people are using your website?

Broadly speaking, you have two options, as shown in Figure 1-4.

Figure 1-4. Two options for turning page view events into aggregate statistics.

Option (a)

You can simply store every single event as it comes in, and then dump them all into a big database, a data warehouse, or a Hadoop cluster. Now, whenever you want to analyze this data in some way, you run a big SELECT query against this dataset. For example, you might group by URL and by time period, or you might filter by some condition and then COUNT(*) to get the number of page views for each URL over time. This will scan essentially all of the events, or at least some large subset, and do the aggregation on the fly.

Option (b)

If storing every single event is too much for you, you can instead store an aggregated summary of the events. For example, if you're counting things, you can increment a few counters every time an event comes in, and then you throw away the

actual event. You might keep several counters in an *OLAP cube*:[3] imagine a multidimensional cube for which one dimension is the URL, another dimension is the time of the event, another dimension is the browser, and so on. For each event, you just need to increment the counters for that particular URL, that particular time, and so on.

With an OLAP cube, when you want to find the number of page views for a particular URL on a particular day, you just need to read the counter for that combination of URL and date. You don't need to scan over a long list of events—it's just a matter of reading a single value.

Now, option (a) in Figure 1-5 might sound a bit crazy, but it actually works surprisingly well. I believe Google Analytics actually does store the raw events—or at least a large sample of events—and performs a big scan over those events when you look at the data. Modern analytic databases have become really good at scanning quickly over large amounts of data.

3 Jim N Gray, Surajit Chaudhuri, Adam Bosworth, et al.: "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals," *Data Mining and Knowledge Discovery*, volume 1, number 1, pages 29–53, March 2007. doi: 10.1023/A:1009726021843

Figure 1-5. Storing raw event data versus aggregating immediately.

The big advantage of storing raw event data is that you have maximum flexibility for analysis. For example, you can trace the sequence of pages that one person visited over the course of their session. You can't do that if you've squashed all the events into counters. That sort of analysis is really important for some offline processing tasks such as training a recommender system (e.g., "people who bought X also bought Y"). For such use cases, it's best to simply keep all the raw events so that you can later feed them all into your shiny new machine-learning system.

However, option (b) in Figure 1-5 also has its uses, especially when you need to make decisions or react to things in real time. For example, if you want to prevent people from scraping your website, you can introduce a rate limit so that you only allow 100 requests per hour from any particular IP address; if a client exceeds the limit, you block it. Implementing that with raw event storage would be incredibly inefficient because you'd be continually rescanning your history of events to determine whether someone has exceeded the limit. It's much more efficient to just keep a counter of number of page views per IP address per time window, and then you can check on every request whether that number has crossed your threshold.

Similarly, for alerting purposes, you need to respond quickly to what the events are telling you. For stock market trading, you also need to be quick.

The bottom line here is that raw event storage and aggregated summaries of events are both very useful—they just have different use cases.

Aggregated Summaries

Let's focus on aggregated summaries for now—how do you implement them?

Well, in the simplest case, you simply have the web server update the aggregates directly, as illustrated in Figure 1-6. Suppose that you want to count page views per IP address per hour, for rate limiting purposes. You can keep those counters in something like memcached or Redis, which have an atomic increment operation. Every time a web server processes a request, it directly sends an increment command to the store, with a key that is constructed from the client IP address and the current time (truncated to the nearest hour).

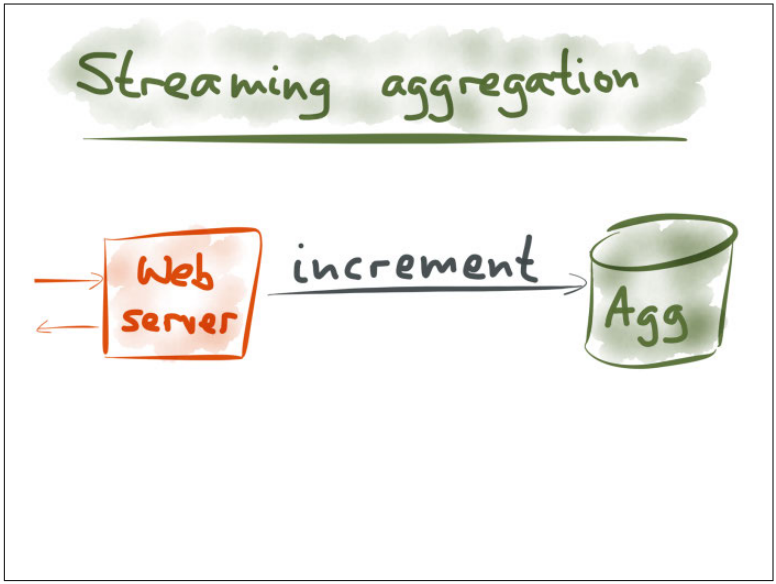


Figure 1-6. The simplest implementation of streaming aggregation.

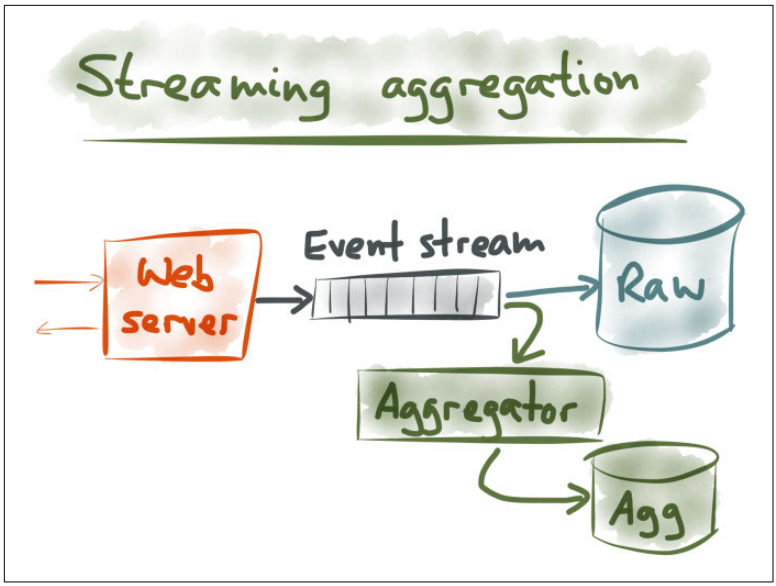


Figure 1-7. Implementing streaming aggregation with an event stream.

If you want to get a bit more sophisticated, you can introduce an event stream, or a message queue, or an event log (or whatever you want to call it), as illustrated in Figure 1-7. The messages on that stream are the PageViewEvent records that we saw earlier: one message contains the content of one particular page view.

The advantage of this architecture is that you can now have multiple consumers for the same event data. You can have one consumer that simply archives the raw events to some big storage; even if you don't yet have the capability to process the raw events, you might as well store them, since storage is cheap and you can figure out how to use them in future. Then, you can have another consumer that does some aggregation (for example, incrementing counters), and another consumer that does monitoring or something else—those can all feed off of the same event stream.

Event Sourcing: From the DDD Community

Now let's change the topic for a moment, and look at similar ideas from a different field. Event sourcing is an idea that has come out of

the DDD community[4]—it seems to be fairly well known among enterprise software developers, but it's totally unknown in Internet companies. It comes with a large amount of jargon that I find confusing, but it also contains some very good ideas.

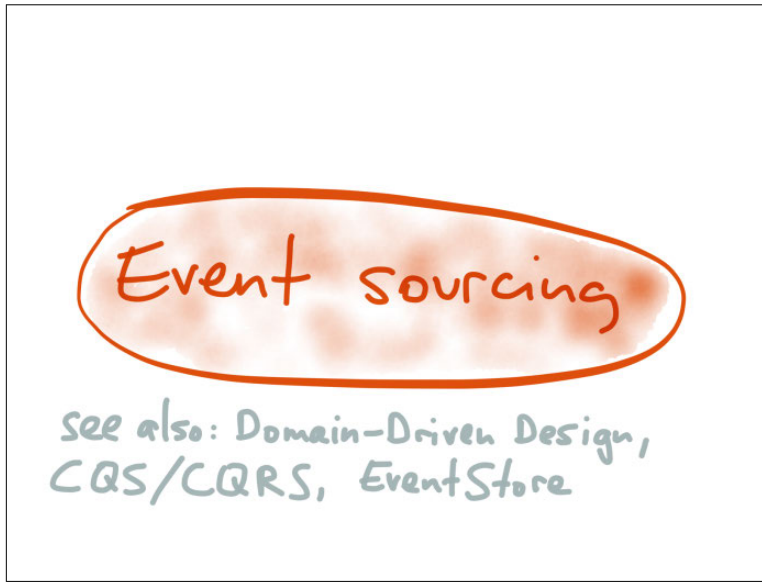


Figure 1-8. Event sourcing is an idea from the DDD community.

Let's try to extract those good ideas without going into all of the jargon, and we'll see that there are some surprising parallels with the last example from the field of stream processing analytics.

Event sourcing is concerned with *how we structure data in databases*. A sample database I'm going to use is a shopping cart from an e-commerce website (Figure 1-9). Each customer may have some number of different products in their cart at one time, and for each item in the cart there is a quantity.

4 Vaughn Vernon: *Implementing Domain-Driven Design*. Addison-Wesley Professional, February 2013. ISBN: 0321834577

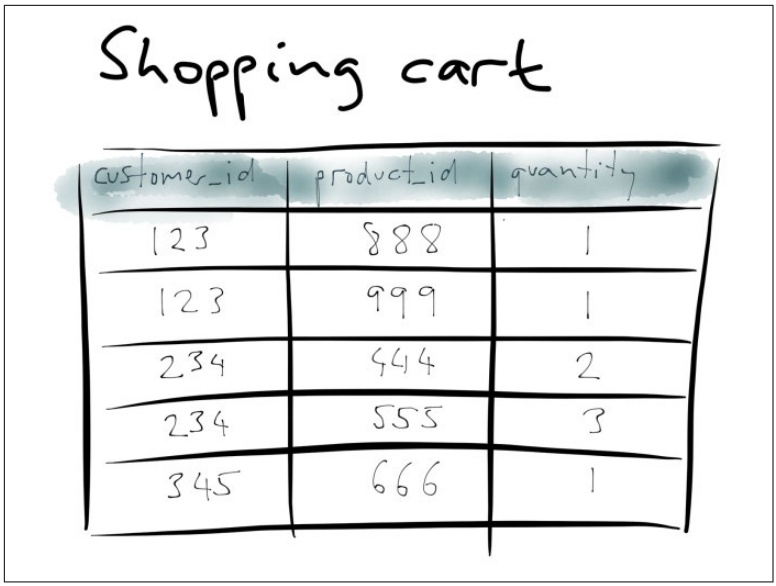


Figure 1-9. Example database: a shopping cart in a traditional relational schema.

Now, suppose that customer 123 updates their cart: instead of quantity 1 of product 999, they now want quantity 3 of that product. You can imagine this being recorded in the database using an UPDATE query, which matches the row for customer 123 and product 999, and modifies that row, changing the quantity from 1 to 3 (Figure 1-10).

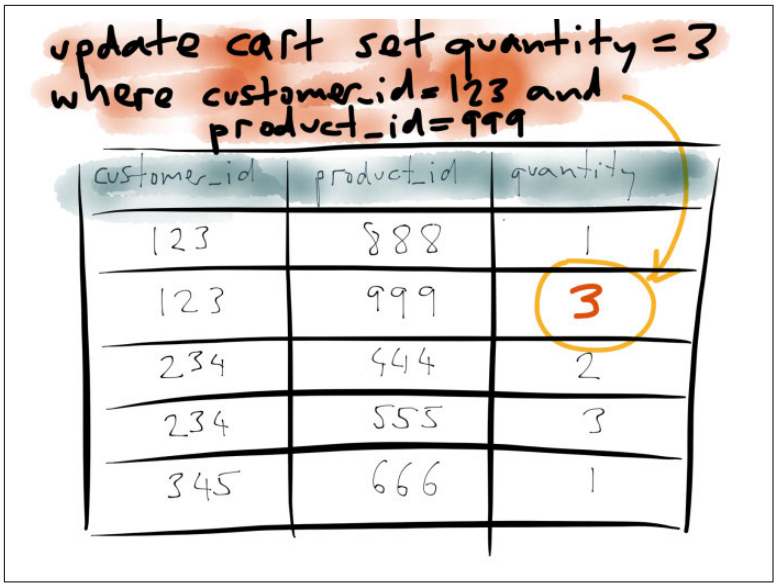


Figure 1-10. Changing a customer's shopping cart, as an UPDATE query.

This example uses a relational data model, but that doesn't really matter. With most non-relational databases you'd do more or less the same thing: overwrite the old value with the new value when it changes.

However, event sourcing says that this isn't a good way to design databases. Instead, we should individually record every change that happens to the database.

For example, Figure 1-11 shows an example of the events logged during a user session. We recorded an AddedToCart event when customer 123 first added product 888 to their cart, with quantity 1. We then recorded a separate UpdatedCartQuantity event when they changed the quantity to 3. Later, the customer changed their mind again, and reduced the quantity to 2, and, finally, they went to the checkout.

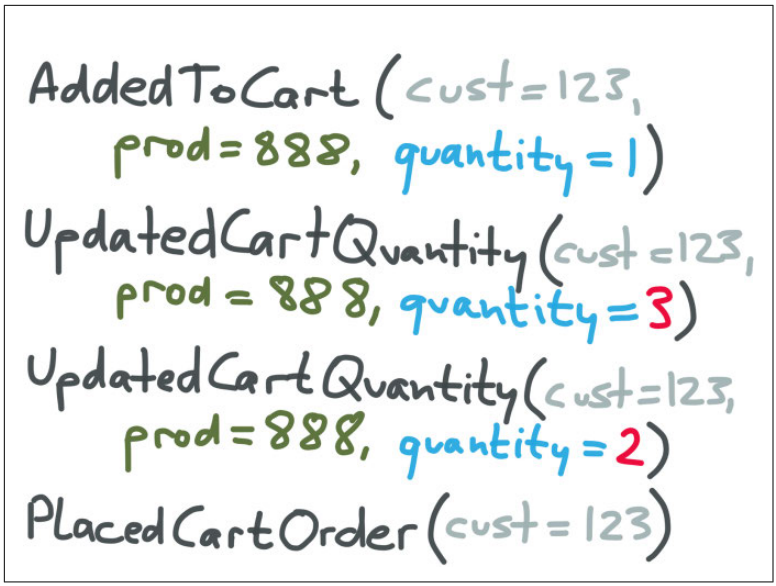


Figure 1-11. Recording every change that was made to a shopping cart.

Each of these actions is recorded as a separate event and appended to the database. You can imagine having a timestamp on every event, too.

When you structure the data like this, every change to the shopping cart is an immutable event—a fact (Figure 1-12). Even if the customer did change the quantity to 2, it is still true that at a previous point in time, the selected quantity was 3. If you overwrite data in your database, you lose this historic information. Keeping the list of all changes as a log of immutable events thus gives you strictly richer information than if you overwrite things in the database.

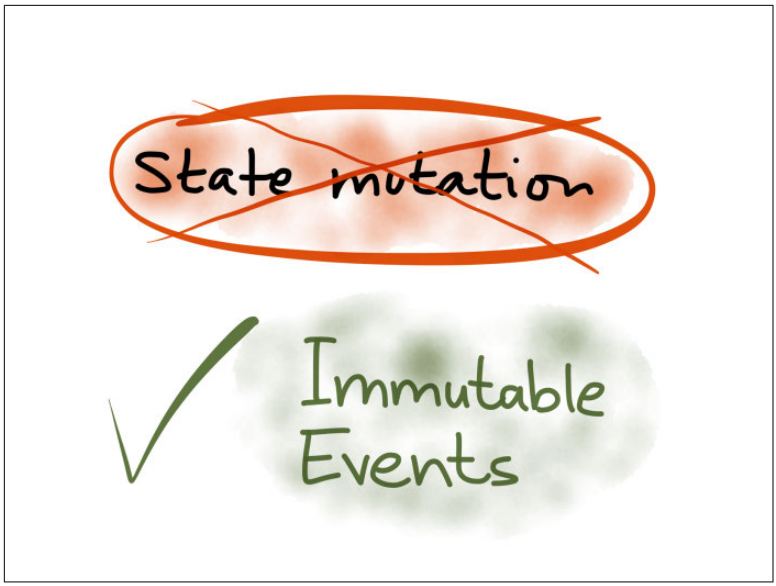


Figure 1-12. Record every write as an immutable event rather than just updating a database in place.

And this is really the essence of event sourcing: rather than performing destructive state mutation on a database when writing to it, we should record every write as an immutable event.

Bringing Together Event Sourcing and Stream Processing

This brings us back to our stream-processing example (Google Analytics). Remember we discussed two options for storing data: (a) raw events, or (b) aggregated summaries (Figure 1-13).

Figure 1-13. Storing raw events versus aggregated data.

Put like this, stream processing for analytics and event sourcing are beginning to look quite similar. Both PageViewEvent (Figure 1-3) and an event-sourced database (AddedToCart, UpdatedCartQuantity) comprise the history of what happened over time. But, when you're looking at the contents of your shopping cart, or the count of page views, you see the current state of the system—the end result, which is what you get when you have applied the entire history of events and squashed them together into one thing.

So the current state of the cart might say quantity 2. The history of raw events will tell you that at some previous point in time the quantity was 3, but that the customer later changed their mind and updated it to 2. The aggregated end result only tells you that the current quantity is 2.

Thinking about it further, you can observe that the raw events are the form in which it's ideal to write the data: all the information in the database write is contained in a single blob. You don't need to go and update five different tables if you're storing raw events—you only need to append the event to the end of a log. That's the simplest and fastest possible way of writing to a database (Figure 1-14).

Figure 1-14. Events are optimized for writes; aggregated values are optimized for reads.

On the other hand, the aggregated data is the form in which it's ideal to read data from the database. If a customer is looking at the contents of their shopping cart, they are not interested in the entire history of modifications that led to the current state: they only want to know what's in the cart right now. An analytics application normally doesn't need to show the user the full list of all page views—only the aggregated summary in the form of a chart.

Thus, when you're reading, you can get the best performance if the history of changes has already been squashed together into a single object representing the current state. In general, the form of data that's best optimized for writing is not the same as the form that is best optimized for reading. It can thus make sense to separate the way you write to your system from the way you read from it (this idea is sometimes known as *command-query responsibility segregation*, or CQRS[5])—more on this later.

5 Greg Young: "CQRS and Event Sourcing," codebetter.com, 13 February 2010.

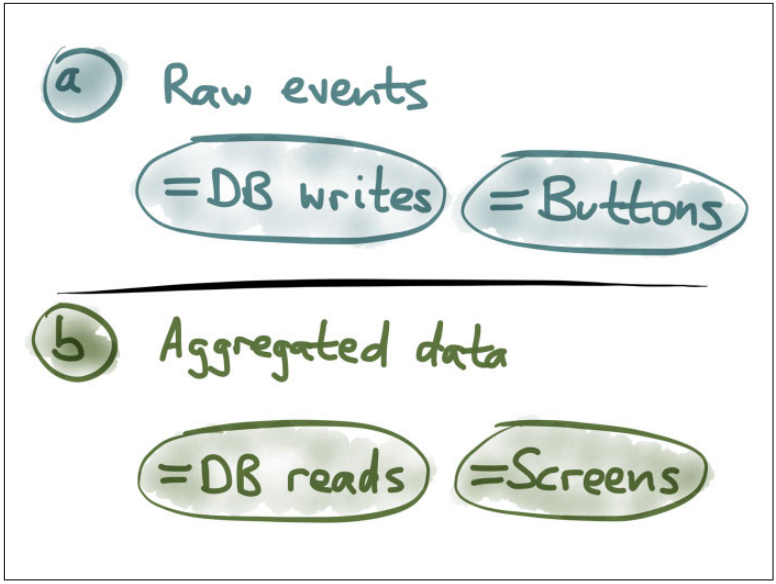


Figure 1-15. As a rule of thumb, clicking a button causes an event to be written, and what a user sees on their screen corresponds to aggregated data that is read.

Going even further, think about the user interfaces that lead to database writes and database reads. A database write typically happens because the user clicks some button; for example, they edit some data and then click the save button. So, buttons in the user interface correspond to raw events in the event sourcing history (Figure 1-15).

On the other hand, a database read typically happens because the user views some screen; they click on some link or open some document, and now they need to read the contents. These reads typically want to know the current state of the database. Thus, screens in the user interface correspond to aggregated state.

This is quite an abstract idea, so let me go through a few examples.

Twitter

For our first example, let's take a look at Twitter (Figure 1-16). The most common way of writing to Twitter's database—that is, to provide input into the Twitter system—is to tweet something. A tweet is very simple: it consists of some text, a timestamp, and the ID of the

user who tweeted (perhaps also optionally a location or a photo). The user then clicks that "Tweet" button, which causes a database write to happen—an event is generated.

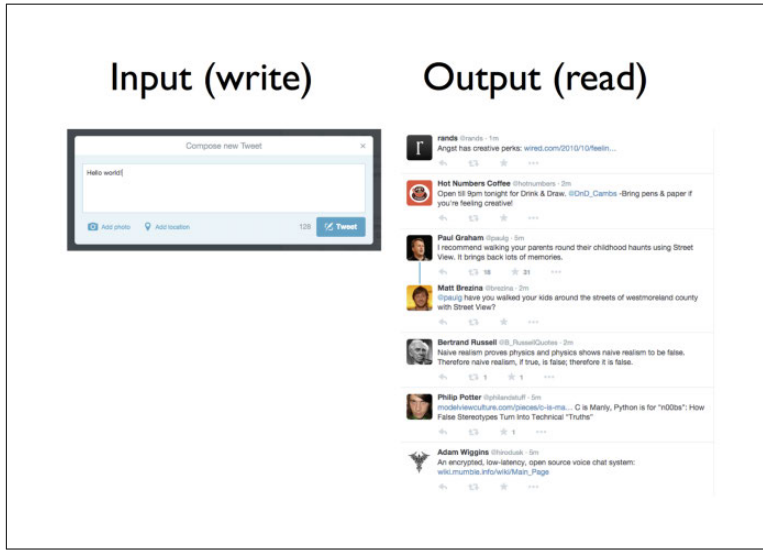


Figure 1-16. Twitter's input: a tweet button. Twitter's output: a timeline.

On the output side, how you read from Twitter's database is by viewing your timeline. It shows all the stuff that was written by people you follow. It's a vastly more complicated structure (Figure 1-17).

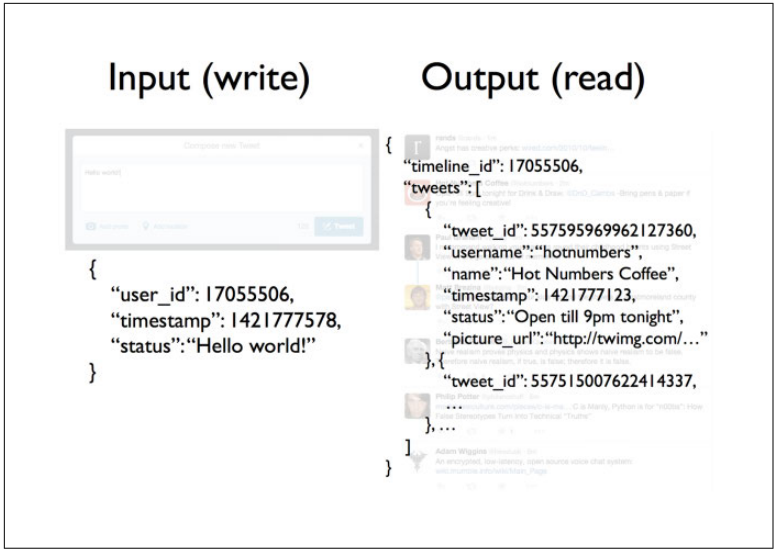


Figure 1-17. Data is written in a simple form; it is read in a much more complex form.

For each tweet, you now have not just the text, timestamp, and user ID, but also the name of the user, their profile photo, and other information that has been joined with the tweet. Also, the list of tweets has been selected based on the people you follow, which may itself change.

How would you go from the simple input to the more complex output? Well, you could try expressing it in SQL, as shown in Figure 1-18.

```
SELECT tweets.*, users.*
  FROM tweets

 JOIN users
   ON users.id = tweets.sender_id

 JOIN follows
   ON follows.followee_id = users.id

 WHERE follows.follower_id = $user

 ORDER BY tweets.time DESC
 LIMIT 100;
```

Figure 1-18. Generating a timeline of tweets by using SQL.

That is, find all of the users who `$user` is following, find all the tweets that they have written, order them by time and pick the 100 most recent. It turns out this query really doesn't scale very well. Do you remember in the early days of Twitter, when it kept having the fail whale all the time? Essentially, that was because they were using something like the query above[6].

When a user views their timeline, it's too expensive to iterate over all the people they are following to get those users' tweets. Instead, Twitter must compute a user's timeline ahead of time, and cache it so that it's fast to read when a user looks at it. To do that, the system needs a process that translates from the write-optimized event (a single tweet) to the read-optimized aggregate (a timeline). Twitter has such a process, and calls it the fanout service. We will discuss it in more detail in Chapter 5.

Facebook

For another example, let's look at Facebook. It has many buttons that enable you to write something to Facebook's database, but a classic one is the "Like" button. When you click it, you generate an event, a

6 Raffi Krikorian: "Timelines at Scale," at *QCon San Francisco*, November 2012.

fact with a very simple structure: *you* (identified by your user ID) *like* (an action verb) *some item* (identified by its ID) (Figure 1-19).

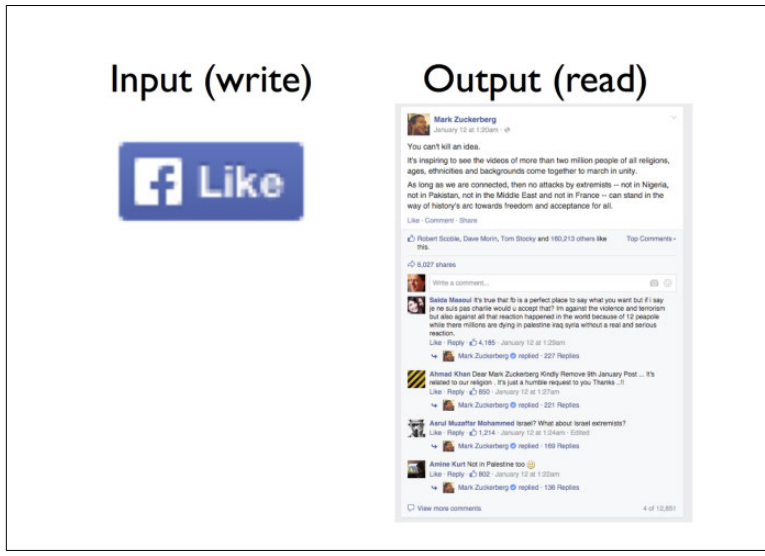


Figure 1-19. Facebook's input: a "like" button. Facebook's output: a timeline post, liked by lots of people.

However, if you look at the output side—reading something on Facebook—it's incredibly complicated. In this example, we have a Facebook post which is not just some text, but also the name of the author and his profile photo; and it's telling me that 160,216 people like this update, of which three have been especially highlighted (presumably because Facebook thinks that among those who liked this update, these are the ones I am most likely to know); it's telling me that there are 6,027 shares and 12,851 comments, of which the top 4 comments are shown (clearly some kind of comment ranking is happening here); and so on.

There must be some translation process happening here, which takes the very simple events as input and then produces a massively complex and personalized output structure (Figure 1-20).

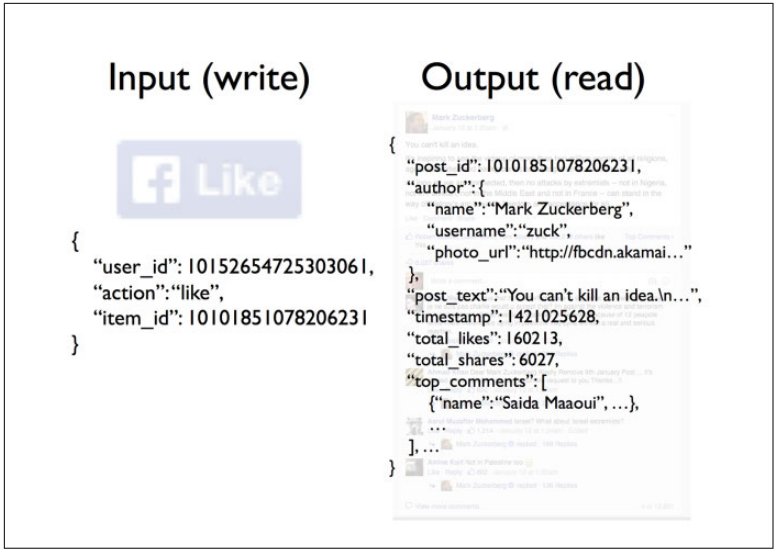


Figure 1-20. When you view a Facebook post, hundreds of thousands of events may have been aggregated in its making.

One can't even conceive what the database query would look like to fetch all of the information in that one Facebook update. It is unlikely that Facebook could efficiently query all of this on the fly—not with over 100,000 likes. Clever caching is absolutely essential if you want to build something like this.

Immutable Facts and the Source of Truth

From the Twitter and Facebook examples we can see a certain pattern: the input events, corresponding to the buttons in the user interface, are quite simple. They are immutable facts, we can simply store them all, and we can treat them as the *source of truth* (Figure 1-21).

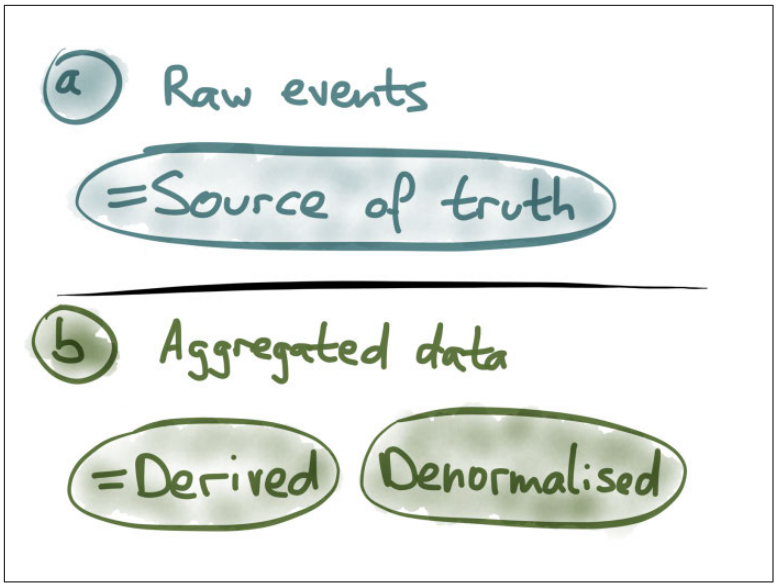


Figure 1-21. Input events that correspond to buttons in a user interface are quite simple.

You can derive everything that you can see on a website—that is, everything that you read from the database—from those raw events. There is a process that derives those aggregates from the raw events, and which updates the caches when new events come in, and that process is entirely deterministic. You could, if necessary, re-run it from scratch: if you feed in the entire history of everything that ever happened on the site, you can reconstruct every cache entry to be exactly as it was before. The database you read from is just a cached view of the event log.[7]

The beautiful thing about this separation between source of truth and caches is that in your caches, you can denormalize data to your heart's content. In regular databases, it is often considered best practice to normalize data, because if something changes, you then only need to change it one place. Normalization makes writes fast and simple, but it means you must do more work (joins) at read time.

To speed up reads, you can denormalize data; that is, duplicate information in various places so that it can be read faster. The prob-

7 Pat Helland: "Accountants Don't Use Erasers," blogs.msdn.com, 14 June 2007.

lem now is that if the original data changes, all the places where you copied it to also need to change. In a typical database, that's a nightmare because you might not know all the places where something has been copied. But, if your caches are built from your raw events using a repeatable process, you have much more freedom to denormalize because you know what data is flowing where.

Wikipedia

Another example is Wikipedia. This is almost a counter-example to Twitter and Facebook, because on Wikipedia the input and the output are almost the same (Figure 1-22).

Figure 1-22. Wikipedia's input: an edit form. Wikipedia's output: an article.

When you edit a page on Wikipedia, you get a big text field containing the entire page content (using wiki markup), and when you click the save button, it sends that entire page content back to the server. The server replaces the entire page with whatever you posted to it. When someone views the page, it returns that same content back to the user (formatted into HTML), as illustrated in Figure 1-23.

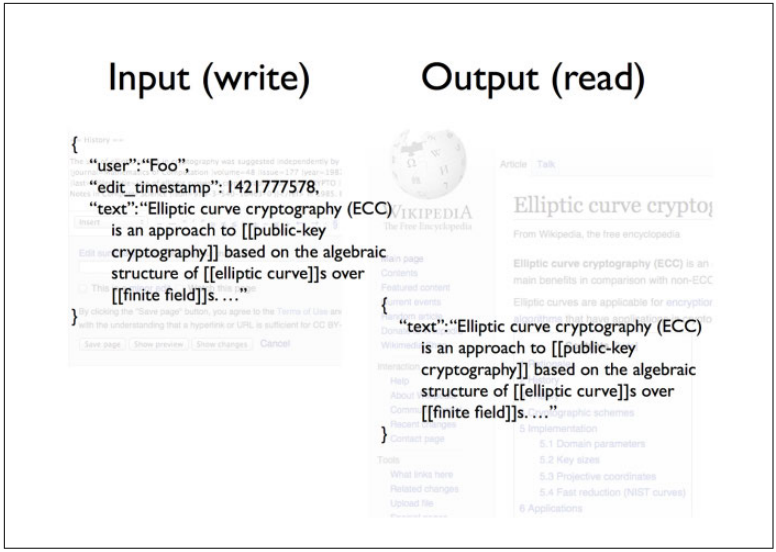


Figure 1-23. On Wikipedia, the input and the output are almost the same.

So, in this case, the input and the output are essentially the same.

What would *event sourcing* mean in this case? Would it perhaps make sense to represent a write event as a diff, like a patch file, rather than a copy of the entire page? It's an interesting case to think about. (Google Docs works by continually applying diffs at the granularity of individual characters—effectively an event per keystroke.[8])

LinkedIn

For our final example, let's consider LinkedIn. Suppose that you update your LinkedIn profile, and add your current job, which consists of a job title, a company, and some text. Again, the edit event for writing to the database is very simple (Figure 1-24).

8 John Day-Richter: "What's different about the new Google Docs: Making collaboration fast," googledrive.blogspot.com, 23 September 2010.

Figure 1-24. LinkedIn's input: your profile edits. LinkedIn's output: a search engine over everybody's profiles.

There are various ways how you can read this data, but in this example, let's look at the search feature. One way that you can read LinkedIn's database is by typing some keywords (and maybe a company name) into a search box and finding all the people who match those criteria.

How is that implemented? Well, to search, you need a full-text index, which is essentially a big dictionary—for every keyword, it tells you the IDs of all the profiles that contain the keyword (Figure 1-25).

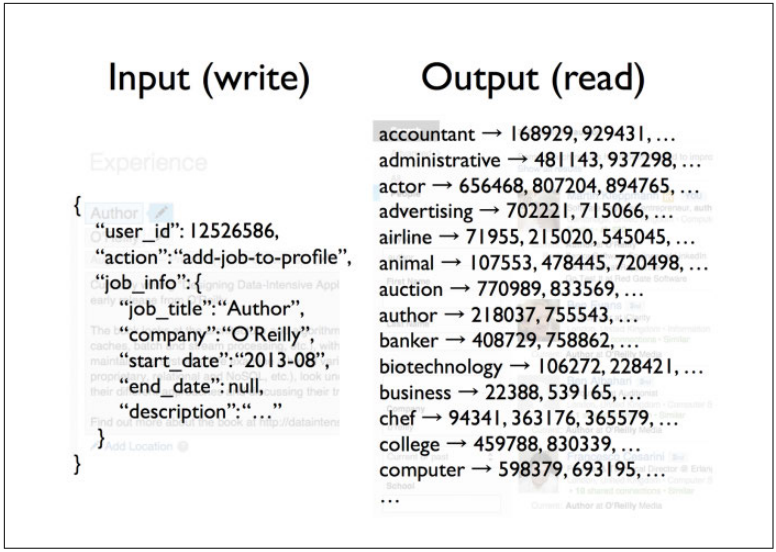


Figure 1-25. A full-text index summarizes which profiles contain which keywords; when a profile is updated, the index needs to be updated accordingly.

This search index is another aggregate structure, and whenever some data is written to the database, this structure needs to be updated with the new data.

So, for example, if I add my job "Author at O'Reilly" to my profile, the search index must now be updated to include my profile ID under the entries for "author" and "o'reilly." The search index is just another kind of cache. It also needs to be built from the source of truth (all the profile edits that have ever occurred), and it needs to be updated whenever a new event occurs (someone edits their profile).

Using Append-Only Streams of Immutable Events

Now, let's return to stream processing.

I first described how you might build something like Google Analytics, compared storing raw page view events versus aggregated counters, and discussed how you can maintain those aggregates by consuming a stream of events (Figure 1-7). I then explained event

sourcing, which applies a similar approach to databases: treat all the database writes as a stream of events, and build aggregates (views, caches, search indexes) from that stream.

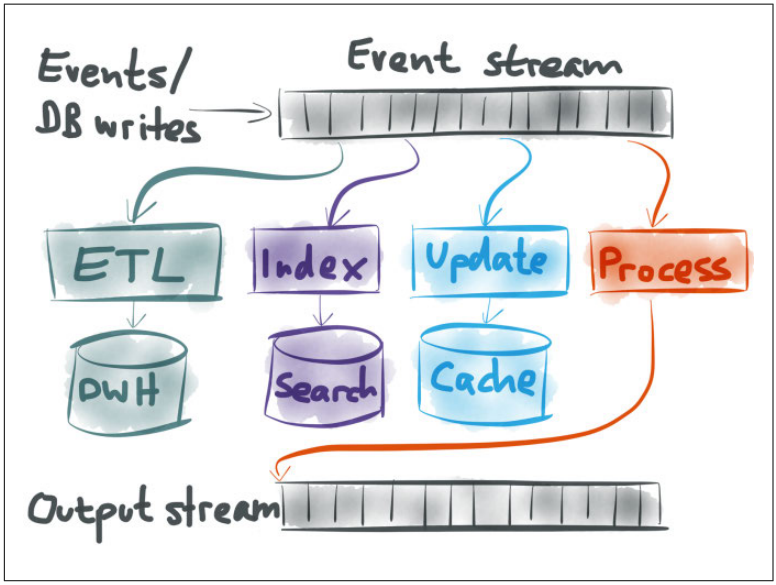


Figure 1-26. Several possibilities for using an event stream.

When you have that event stream, you can do many great things with it (Figure 1-26):

- You can take all the raw events, perhaps transform them a bit, and load them into Hadoop or a big data warehouse where analysts can query the data to their heart's content.
- You can update full-text search indexes so that when a user hits the search box, they are searching an up-to-date version of the data. We will discuss this in more detail in Chapter 2.
- You can invalidate or refill any caches so that reads can be served from fast caches while also ensuring that the data in the cache remains fresh.
- And finally, you can even take one event stream, and process it in some way (perhaps joining a few streams together) to create a new output stream. This way, you can plug the output of one system into the input of another system. This is a very powerful

way of building complex applications cleanly, which we will discuss in Chapter 4.

Moving to an event-sourcing-like approach for databases is a big change from the way that databases have traditionally been used (in which you can update and delete data at will). Why would you want to go to all that effort of changing the way you do things? What's the benefit of using append-only streams of immutable events?

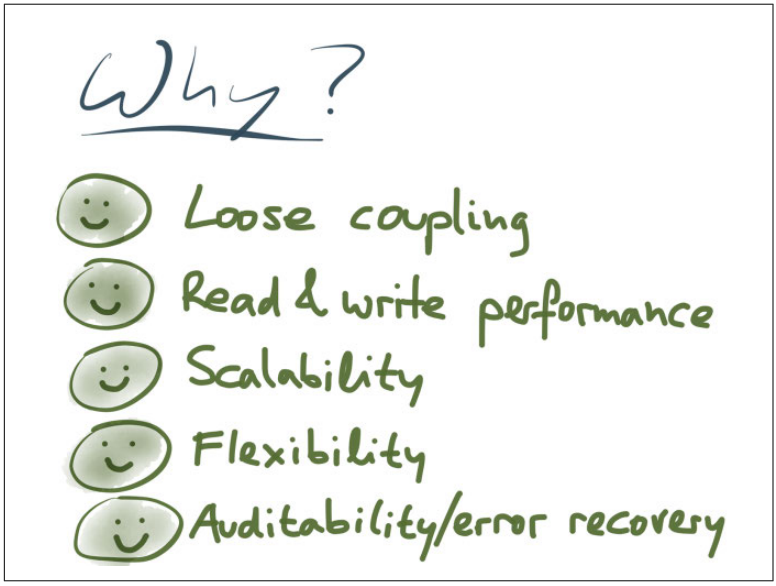


Figure 1-27. Several reasons why you might benefit from an event-sourced approach.

There are several reasons (Figure 1-27):

Loose coupling
: If you write data to the database in the same schema as you use for reading, you have tight coupling between the part of the application doing the writing (the "button") and the part doing the reading (the "screen"). We know that loose coupling is a good design principle for software. By separating the form in which you write and read data, and by explicitly translating from one to the other, you get much looser coupling between different parts of your application.

Read and write performance
: The decades-old debate over normalization (faster writes) versus denormalization (faster reads) exists only because of the assumption that writes and reads use the same schema. If you separate the two, you can have fast writes *and* fast reads.

Scalability
: Event streams are great for scalability because they are a simple abstraction (comparatively easy to parallelize and scale across multiple machines), and because they allow you to decompose your application into producers and consumers of streams (which can operate independently and can take advantage of more parallelism in hardware).

Flexibility and agility
: Raw events are so simple and obvious that a "schema migration" doesn't really make sense (you might just add a new field from time to time, but you don't usually need to rewrite historic data into a new format). On the other hand, the ways in which you want to present data to users are much more complex, and can be continually changing. If you have an explicit translation process between the source of truth and the caches that you read from, you can experiment with new user interfaces by just building new caches using new logic, running the new system in parallel with the old one, gradually moving people over from the old system, and then discarding the old system (or reverting to the old system if the new one didn't work). Such flexibility is incredibly liberating.

Error scenarios
: Error scenarios are much easier to reason about if data is immutable. If something goes wrong in your system, you can always replay events in the same order and reconstruct exactly what happened[9] (especially important in finance, for which auditability is crucial). If you deploy buggy code that writes bad data to a database, you can just re-run it after you fixed the bug and thus correct the outputs. Those things are not possible if your database writes are destructive.

9 Martin Fowler: "The LMAX Architecture," martinfowler.com, 12 July 2011.

Tools: Putting Ideas into Practice

Let's talk about how you might put these ideas into practice. How do you build applications using this idea of event streams?

Some databases such as Event Store[10] have oriented themselves specifically at the event sourcing model, and some people have implemented event sourcing on top of relational databases.

The systems I have worked with most—and that we discuss most in this report—are Apache Kafka[11] and Apache Samza.[12] Both are open source projects that originated at LinkedIn and now have a big community around them. Kafka provides a publish-subscribe message queuing service, supporting event streams with many millions of messages per second, durably stored on disk and replicated across multiple machines.[13,14]

For consuming input streams and producing output streams, Kafka comes with a client library called Kafka Streams (Figure 1-28): it lets you write code to process messages, and it handles stuff like state management and recovering from failures.[15]

10 "Event Store," Event Store LLP, geteventstore.com.

11 "Apache Kafka," Apache Software Foundation, kafka.apache.org.

12 "Apache Samza," Apache Software Foundation, samza.apache.org.

13 Jay Kreps: "Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines)," engineering.linkedin.com, 27 April 2014.

14 Todd Palino: "Running Kafka At Scale," engineering.linkedin.com, 20 March 2015.

15 Guozhang Wang: "KIP-28 – Add a processor client," cwiki.apache.org, 24 July 2015.

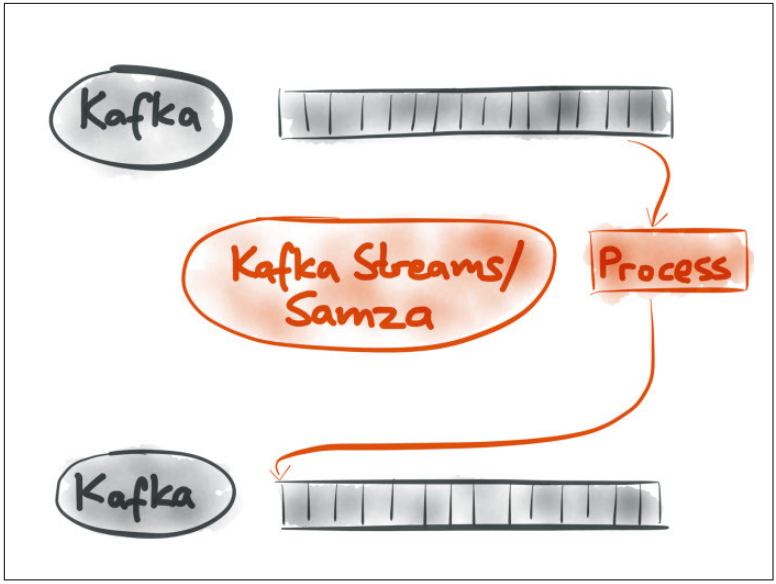


Figure 1-28. Apache Kafka is a good implementation of event streams, and tools like Kafka Streams or Apache Samza can be used to process those streams.

I would definitely recommend Kafka as a system for high-throughput reliable event streams. When you want to write code to process those events, you can either use Kafka's client libraries directly, or you can use one of several frameworks (Figure 1-29): Samza,[16] Storm,[17] Spark Streaming[18] and Flink[19] are the most popular. Besides message processing, these frameworks also include tools for deploying a processing job to a cluster of machines and scheduling its tasks.

16 "Apache Samza," Apache Software Foundation, samza.apache.org.

17 "Apache Storm," Apache Software Foundation, storm.apache.org.

18 "Apache Spark Streaming," Apache Software Foundation, spark.apache.org.

19 "Apache Flink," Apache Software Foundation, flink.apache.org.

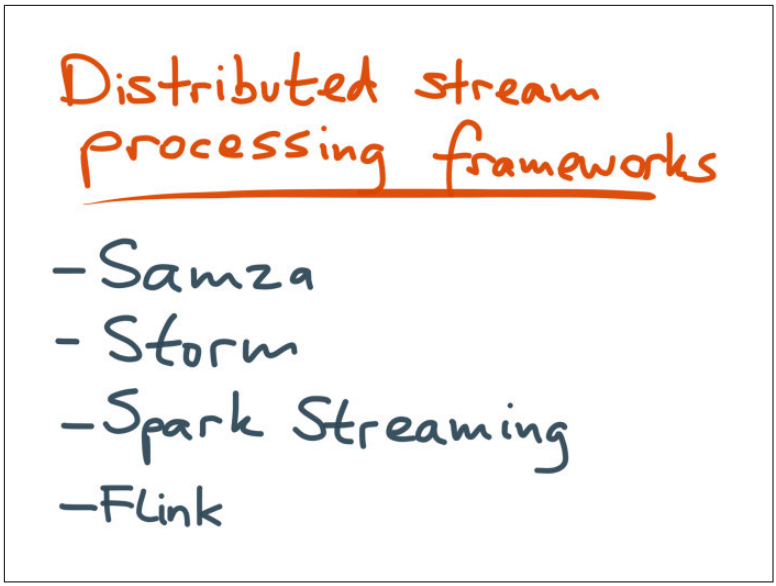


Figure 1-29. List of distributed stream processing frameworks.

There are interesting design differences (pros and cons) between these tools. In this report we will not go into the details of stream processing frameworks and their APIs—you can find a detailed comparison in the Samza documentation.[20] Instead, in this report we focus on the conceptual foundations that underpin all stream processing systems.

Today's distributed stream processing systems have their roots in stream processing research from the early 2000s (TelegraphCQ,[21] Borealis,[22] and so on), which originated from a relational database background. Just as NoSQL datastores stripped databases down to a minimal feature set, modern stream processing systems look quite stripped-down compared to the earlier research.

20 "Comparison Introduction," Apache Samza 0.8 Documentation, samza.apache.org, 3 April 2015.

21 Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, et al.: "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World," at *1st Biennial Conference on Innovative Data Systems Research* (CIDR), January 2003.

22 Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, et al.: "The Design of the Borealis Stream Processing Engine," at *2nd Biennial Conference on Innovative Data Systems Research* (CIDR), November 2004.

CEP, Actors, Reactive, and More

Contemporary distributed stream processing frameworks (Kafka Streams, Samza, Storm, Spark Streaming, Flink) are mostly concerned with low-level matters: how to scale processing across multiple machines, how to deploy a job to a cluster, how to handle faults (crashes, machine failures, network outages), and how to achieve reliable performance in a multitenant environment.[23] The APIs they provide are often quite low-level (e.g., a callback that is invoked for every message). They look much more like MapReduce and less like a database, although there is work in progress to provide high-level query languages such as streaming SQL.

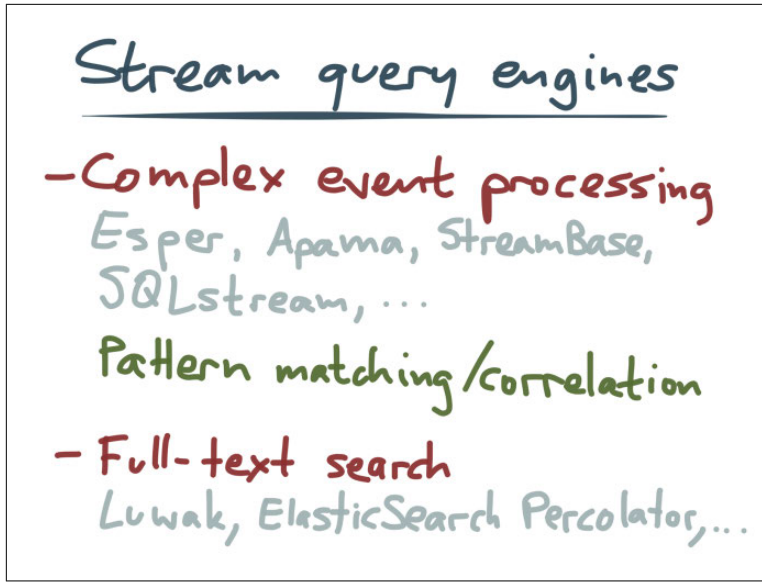


Figure 1-30. Stream query engines provide higher-level abstractions than stream processing frameworks.

There is also some existing work on high-level query languages for stream processing, and CEP is especially worth mentioning (Figure 1-30). It originated in 1990s research on event-driven simu-

23 Jay Kreps: "But the multi-tenancy thing is actually really really hard," tweetstorm, twitter.com, 31 October 2014.

lation.[24] Many CEP products are commercial, expensive enterprise software, although Esper[25] has an open source version. (Esper is a library that you can run inside a distributed stream processing framework, but it does not provide distributed query execution.)

With CEP, you write queries or rules that match certain patterns in the events. They are comparable to SQL queries (which describe what results you want to return from a database), except that the CEP engine continually searches the stream for sets of events that match the query and notifies you (generates a "complex event") whenever a match is found. This is useful for fraud detection or monitoring business processes, for example.

For use cases that can be easily described in terms of a CEP query language, such a high-level language is much more convenient than a low-level event processing API. On the other hand, a low-level API gives you more freedom, allowing you to do a wider range of things than a query language would let you do. Also, by focusing their efforts on scalability and fault tolerance, stream processing frameworks provide a solid foundation upon which query languages can be built.

Another idea for high-level querying is doing full-text search on streams, whereby you register a search query in advance and then are notified whenever an event matches your query. For example, Elasticsearch Percolator[26] provides this as a service, and Luwak[27] implements full-text search on streams as an embeddable library.

24 David C Luckham: "What's the Difference Between ESP and CEP?," complexevents.com, 1 August 2006.

25 "Esper: Event Processing for Java," EsperTech Inc., espertech.com.

26 "Elasticsearch 1.7 Reference: Percolator," Elasticsearch Global BV, elastic.co.

27 "Luwak – stored query engine from Flax," Flax, github.com.

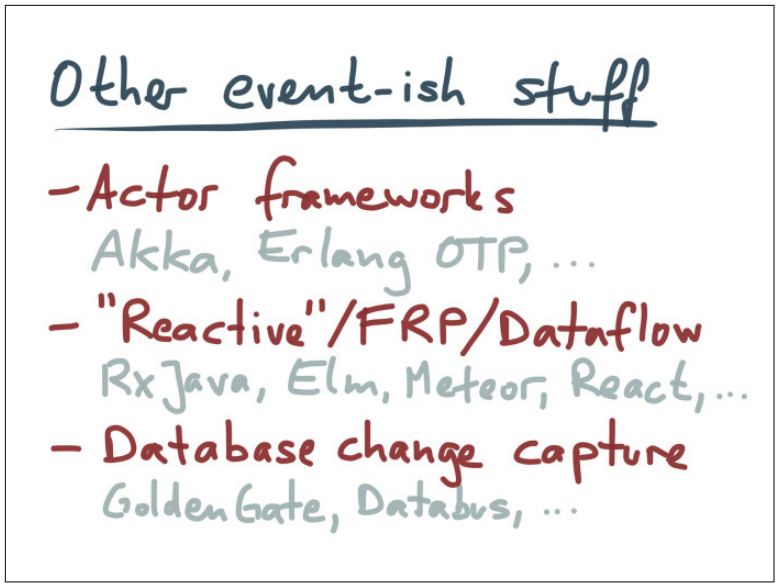


Figure 1-31. Lots of other people also seem to think that events are a good idea.

Finally, there are a lot of other ideas that are somehow related to event streams (Figure 1-31). Here is a brief summary:

- *Distributed actor frameworks* such as Akka,[28] Orleans,[29] and Erlang OTP[30] are also based on streams of immutable events/messages. However, they are primarily a mechanism for programming concurrent systems, less a mechanism for data management. In principle, you could build a distributed stream processing framework on top of actors, but it's worth looking carefully at the fault-tolerance guarantees and failure modes of these systems: many don't provide durability, for example. SEDA architectures[31] have some similarities to actors.

28 "Akka," Typesafe Inc., akka.io.

29 "Microsoft Project Orleans," Microsoft Research, dotnet.github.io.

30 "Erlang/OTP 18 Documentation," Ericsson AB, erlang.org.

31 Matt Welsh: "A Retrospective on SEDA," matt-welsh.blogspot.co.uk, 26 July 2010.

- There's a lot of buzz around "*reactive*", which seems to encompass a quite loosely defined set of ideas.[32] My impression is that there is some good work happening in dataflow languages, ReactiveX and functional reactive programming (FRP), which I see as mostly about bringing event streams to the user interface (i.e., updating the user interface when some underlying data changes).[33] This is a natural counterpart to event streams in the data backend (we touch on it in Chapter 5).
- Finally, *change data capture* (CDC) means using an existing database in the familiar way, but extracting any inserts, updates, and deletes into a stream of data change events that other applications can consume. We discuss this in detail in Chapter 3.

I hope this chapter helped you make some sense of the many facets of stream processing. In Chapter 2, we dig deep into the idea of a "log," which is a particularly good way of implementing streams.

32 Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson: "The Reactive Manifesto v2.0," reactivemanifesto.org, 16 September 2014.

33 "ReactiveX," reactivex.io.

CHAPTER 2

Using Logs to Build a Solid Data Infrastructure

In Chapter 1, we explored the idea of representing data as a series of events. This idea applies not only if you want to keep track of things that happened (e.g., page views in an analytics application), but we also saw that events work well for describing changes to a database (event sourcing).

However, so far we have been a bit vague about what the *stream* should look like. In this chapter, we will explore the answer in detail: a stream should be implemented as a *log*; that is, an append-only sequence of events in a fixed order. (This is what Apache Kafka does.)

It turns out that the ordering of events is really important, and many systems (such as AMQP or JMS message queues) do not provide a fixed ordering. In this chapter, we will go on a few digressions outside of stream processing, to look at logs appearing in other places: in database storage engines, in database replication, and even in distributed consensus systems.

Then, we will take what we have learned from those other areas of computing and apply it to stream processing. Those lessons will help us build applications that are operationally robust, reliable, and that perform well.

But before we get into logs, we will begin this chapter with a motivating example: the sprawling complexity of data integration in a

large application. If you work on a non-trivial application—something with more than just one database—you'll probably find these ideas very useful. (Spoiler: the solution involves a log.)

Case Study: Web Application Developers Driven to Insanity

To begin, let's assume that you're working on a web application. In the simplest case, it probably has the stereotypical three-tier architecture (Figure 2-1): you have some clients (which may be web browsers, or mobile apps, or both), which make requests to a web application running on your servers. The web application is where your application code or *business logic* lives.

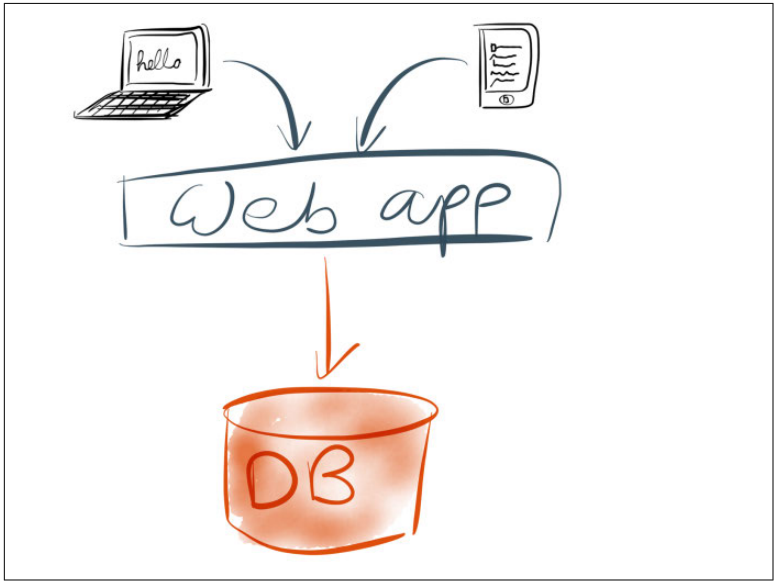


Figure 2-1. One web app, one database: life is simple.

Whenever the application wants to remember something for the future, it stores it in a database. Accordingly, whenever the application wants to look up something that it stored previously, it queries the database. This approach is simple to understand and works pretty well.

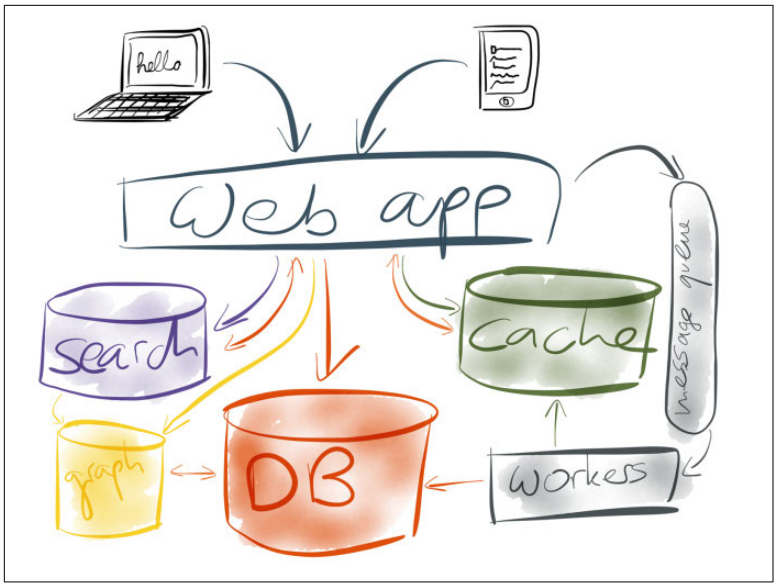


Figure 2-2. Things usually don't stay so simple for long.

However, things usually don't stay so simple for long (Figure 2-2). Perhaps you get more users, making more requests, your database becomes too slow, and you add a cache to speed it up—perhaps memcached or Redis, for example. Perhaps you need to add full-text search to your application, and the basic search facility built into your database is not good enough, so you set up a separate indexing service such as Elasticsearch or Solr.

Perhaps you need to do some graph operations that are not efficient on a relational or document database—for example for social features or recommendations—so you add a separate graph index to your system. Perhaps you need to move some expensive operations out of the web request flow and into an asynchronous background process, so you add a message queue that lets you send jobs to your background workers.

And it gets worse... (Figure 2-3)

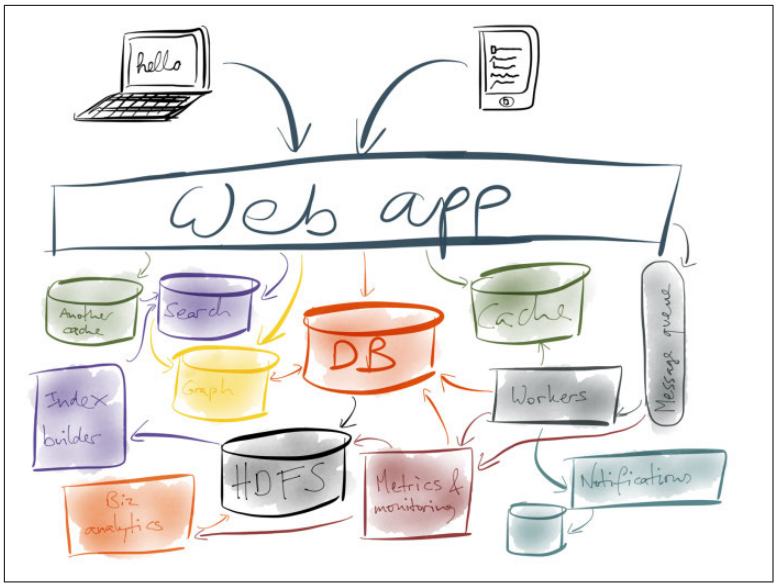


Figure 2-3. As the features and the business requirements of an application grow, we see a proliferation of different tools being used in combination with one another.

By now, other parts of the system are becoming slow again, so you add another cache. More caches always make things faster, right? But now you have a lot of systems and services, so you need to add metrics and monitoring so that you can see whether they are actually working. Of course, the metrics system is another system in its own right.

Next, you want to send notifications, such as email or push notifications to your users, so you chain a notification system off the side of the job queue for background workers, and perhaps it needs some kind of database of its own to keep track of stuff. However, now you're generating a lot of data that needs to be analyzed, and you can't have your business analysts running big expensive queries on your main database, so you add Hadoop or a data warehouse and load the data from the database into it.

Now that your business analytics are working, you find that your search system is no longer keeping up... but you realize that because you have all the data in HDFS anyway, you could actually build your search indexes in Hadoop and push them out to the search servers.

All the while, the system just keeps growing more and more complicated.

The result is complete and utter insanity (Figure 2-4).

Figure 2-4. A system with many interdependent components becomes very complex and difficult to manage and understand.

How did we get to that state? How did we end up with such complexity, where everything is calling everything else and nobody understands what is going on?

It's not that any particular decision we made along the way was bad. There is no one database or tool that can do everything that our application requires.[1] We use the best tool for the job, and for an application with a variety of features that implies using a variety of tools.

Also, as a system grows, you need a way of decomposing it into smaller components in order to keep it manageable. That's what microservices are all about (see Chapter 4). But, if your system

1 Michael Stonebraker and Uğur Çetintemel: "'One Size Fits All': An Idea Whose Time Has Come and Gone," at *21st International Conference on Data Engineering* (ICDE), April 2005.

becomes a tangled mess of interdependent components, that's not manageable either.

Simply having many different storage systems is not a problem in and of itself: if they were all independent from one another, it wouldn't be a big deal. The real trouble here is that many of them end up containing the same data, or related data, but in different form (Figure 2-5).

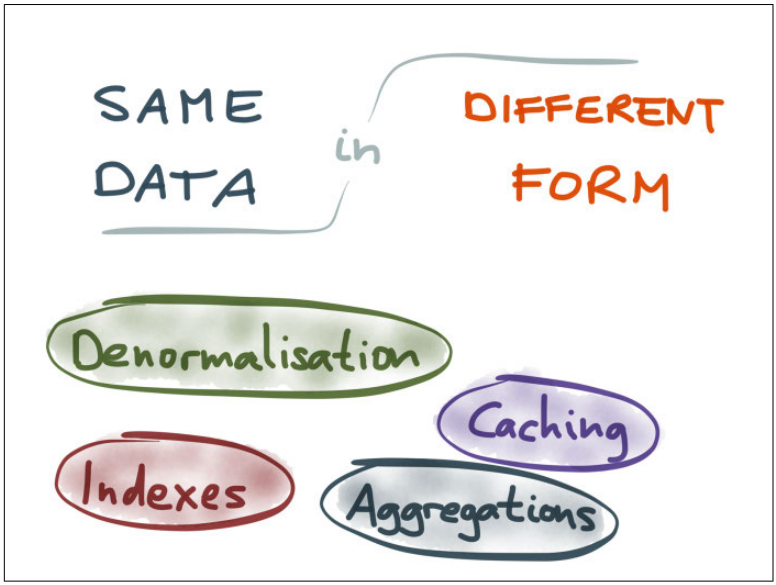


Figure 2-5. Denormalization, caching, indexes, and aggregations are various kinds of redundant data: keeping the same data in a different representation in order to speed up reads.

For example, the documents in your full-text indexes are typically also stored in a database because search indexes are not intended to be used as systems of record. The data in your caches is a duplicate of data in some database (perhaps joined with other data, or rendered into HTML, or something)—that's the definition of a cache.

Also, denormalization is just another form of duplicating data, similar to caching—if some value is too expensive to recompute on reads, you can store that value somewhere, but now you need to also keep it up-to-date when the underlying data changes. Materialized aggregates, such as those in the analytics example in Chapter 1, are again a form of redundant data.

I'm not saying that this duplication of data is bad—far from it. Caching, indexing, and other forms of redundant data are often essential for achieving good performance on reads. However, keeping the data synchronized between all these various different representations and storage systems becomes a real challenge (Figure 2-6).

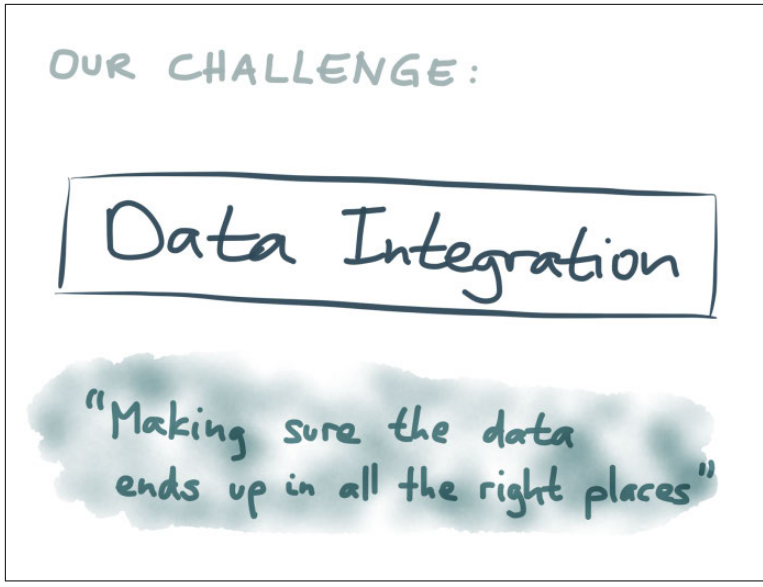


Figure 2-6. The problem of data integration: keeping data systems synchronized.

For lack of a better term, I'm going to call this the problem of "data integration." With that I really just mean *making sure that the data ends up in all the right places*. Whenever a piece of data changes in one place, it needs to change correspondingly in all the other places where there is a copy or derivative of that data.

So, how do we keep these different data systems synchronized? There are a few different techniques.

Dual Writes

A popular approach is called *dual writes* (Figure 2-7). The dual-writes technique is simple: it's the responsibility of your application code to update data in all the appropriate places. For example, if a user submits some data to your web app, there's some code in the web app that first writes the data to your database, then invalidates

or refreshes the appropriate cache entries, then re-indexes the document in your full-text search index, and so on. (Or, maybe it does those things in parallel—that doesn't matter for our purposes.)

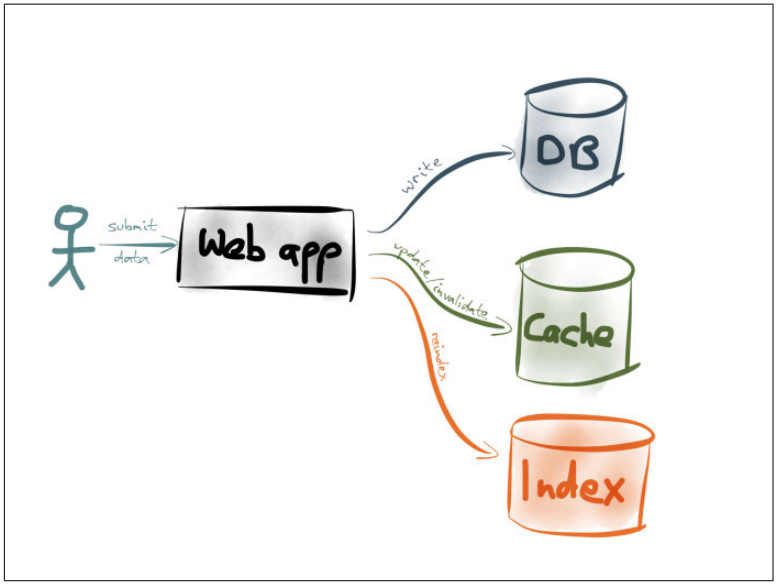


Figure 2-7. With dual writes, your application code is responsible for writing data to all the appropriate places.

The dual-writes approach is popular because it's easy to build, and it more or less works at first. But I'd like to argue that it's a really bad idea, because it has some fundamental problems. The first problem is race conditions.

Figure 2-8 shows two clients making dual writes to two datastores. Time flows from left to right, following the black arrows.

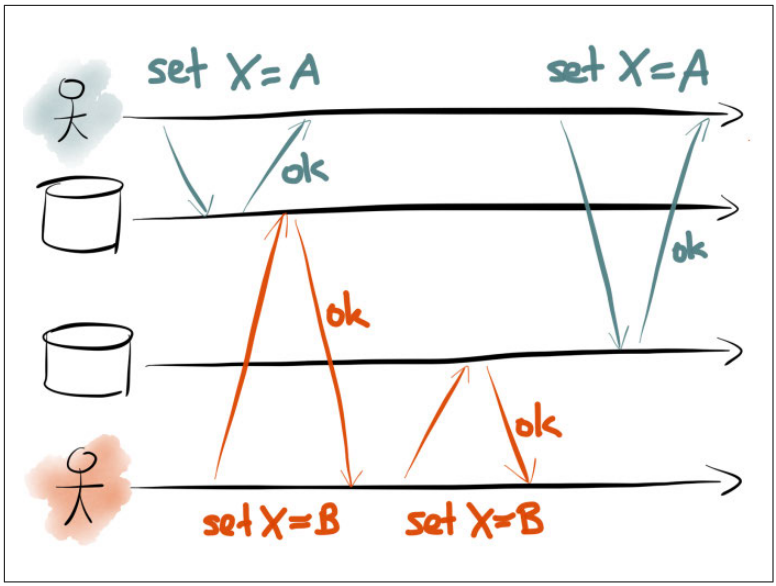


Figure 2-8. Timing diagram showing two different clients concurrently writing to the same key, using dual writes.

Here, the first client (teal) is setting the key X to be some value A. They first make a request to the first datastore—perhaps that's the database, for example—and set X=A. The datastore responds by saying the write was successful. Then, the client makes a request to the second datastore—perhaps that's the search index—and also sets X=A.

Simultaneously, another client (red) is also active. It wants to write to the same key X, but it wants to set the key to a different value B. The client proceeds in the same way: it first sends a request, X=B, to the first datastore and then sends a request, X=B, to the second datastore.

All of these writes are successful. However, look at what value is stored in each database over time (Figure 2-9).

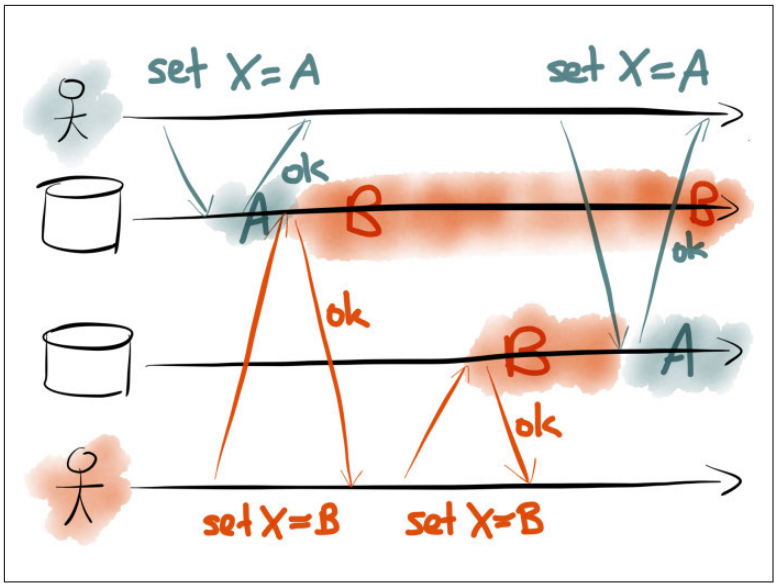


Figure 2-9. A race condition with dual writes leads to perpetual inconsistency between two datastores.

In the first datastore, the value is first set to A by the teal client, and then set to B by the red client, so the final value is B.

In the second datastore, the requests arrive in a different order: the value is first set to B and then set to A, so the final value is A. Now, the two datastores are inconsistent with each other, and they will permanently remain inconsistent until sometime later when someone comes and overwrites X again.

The worst thing is this: you probably won't even notice that your database and your search indexes have become inconsistent because no errors occurred. You'll probably only realize it six months later, while you're doing something completely different, that your database and your indexes don't match up, and you'll have no idea how that could have happened. This is not a problem of eventual consistency—it's *perpetual inconsistency*.

That alone should be enough to put anyone off dual writes.

But wait, there's more...

Denormalized data

Let's look at denormalized data. Suppose, for example, that you have an application with which users can send each other messages or emails, and you have an inbox for each user. When a new message is sent, you want to do two things: add the message to the list of messages in the user's inbox, and also increment the user's count of unread messages (Figure 2-10).

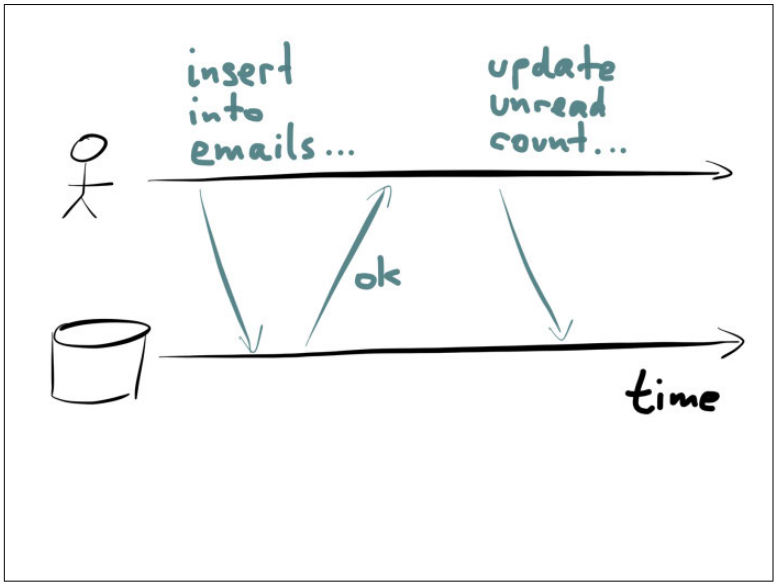


Figure 2-10. A counter of unread messages, which needs to be kept up-to-date when a new message comes in.

You keep a separate counter because you display it in the user interface all the time, and it would be too slow to query the number of unread messages by scanning over the list of messages every time you need to display the number. However, this counter is denormalized information: it's derived from the actual messages in the inbox, and whenever the messages change, you also need to update the counter accordingly.

Let's keep this one simple: one client, one database. Think about what happens over time: first, the client inserts the new message into the recipient's inbox. Then, the client makes a request to increment the unread counter.

However, just in that moment, something goes wrong—perhaps the database goes down, or a process crashes, or the network is interrupted, or someone unplugs the wrong network cable (Figure 2-11). Whatever the reason, the update to the unread counter fails.

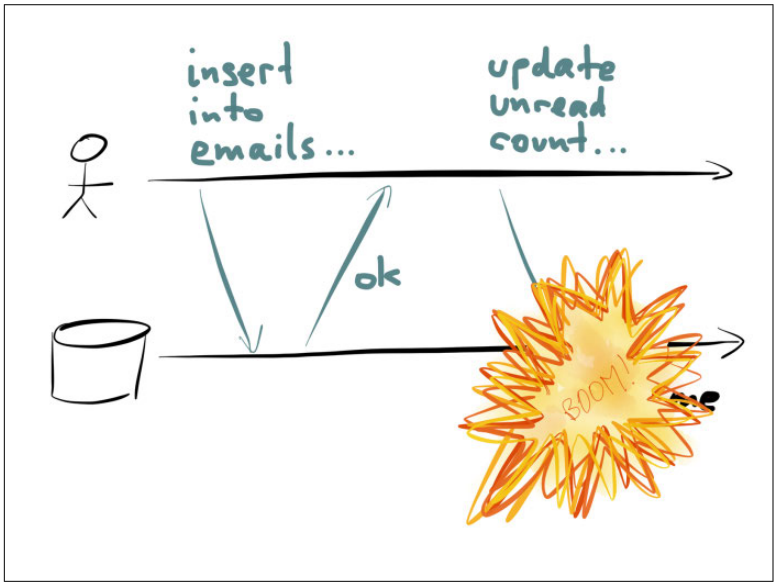


Figure 2-11. One write succeeds; the other write fails. What now?

Now, your database is inconsistent: the message has been added to the inbox, but the counter hasn't been updated. And unless you periodically recompute all your counter values from scratch, or undo the insertion of the message, it will forever remain inconsistent. Such problems are not hypothetical—they do occur in practice.[2]

Of course, you could argue that this problem was solved decades ago by *transactions*: atomicity, the "A" in "ACID," means that if you make several changes within one transaction, they either all happen or none happen (Figure 2-12).

2 Martin Kleppmann: "Eventual consistency? More like perpetual inconsistency," twitter.com, 17 November 2014.

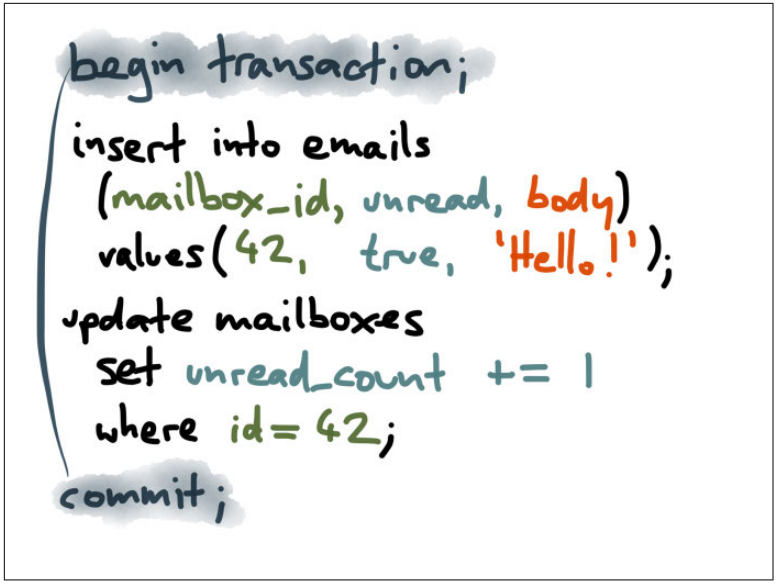


Figure 2-12. Transaction atomicity means that if you make several changes, they either all happen or none happen.

The purpose of atomicity is to solve precisely this issue—if something goes wrong during your writes, you don't need to worry about a half-finished set of changes making your data inconsistent.

The traditional approach of wrapping the two writes in a transaction works fine in databases that support it, but many of the new generation of databases ("NoSQL") don't, so you're on your own.

Also, if the denormalized information is stored in a different database—for example, if you keep your emails in a database but your unread counters in Redis—you lose the ability to tie the writes together into a single transaction. If one write succeeds and the other fails, you're going to have a difficult time clearing up the inconsistency.

Some systems support distributed transactions, based on 2-phase commit, for example.[3] However, many datastores nowadays don't support it, and even if they did, it's not clear whether distributed

3 Henry Robinson: "Consensus Protocols: Two-Phase Commit," the-paper-trail.org, 27 November 2008.

transactions are a good idea in the first place.[4] So, we must assume that with dual writes the application needs to deal with partial failure, which is difficult.

Making Sure Data Ends Up in the Right Places

So, back to our original question: how do we make sure that all the data ends up in all the right places (Figure 2-6)? How do we get a copy of the same data to appear in several different storage systems, and keep them all consistently synchronized as the data changes?

As we saw, dual writes isn't the solution, because it can introduce inconsistencies due to race conditions and partial failures. Then, how can we do better?

I'm a fan of stupidly simple solutions. The great thing about simple solutions is that you have a chance of understanding them and convincing yourself that they're correct. In this case, the simplest solution I can see is to store all your writes in a fixed order, and apply them in that fixed order to the various places they need to go (Figure 2-13).

4 Pat Helland: "Life beyond Distributed Transactions: an Apostate's Opinion," at *3rd Biennial Conference on Innovative Data Systems Research* (CIDR), pages 132–141, January 2007.

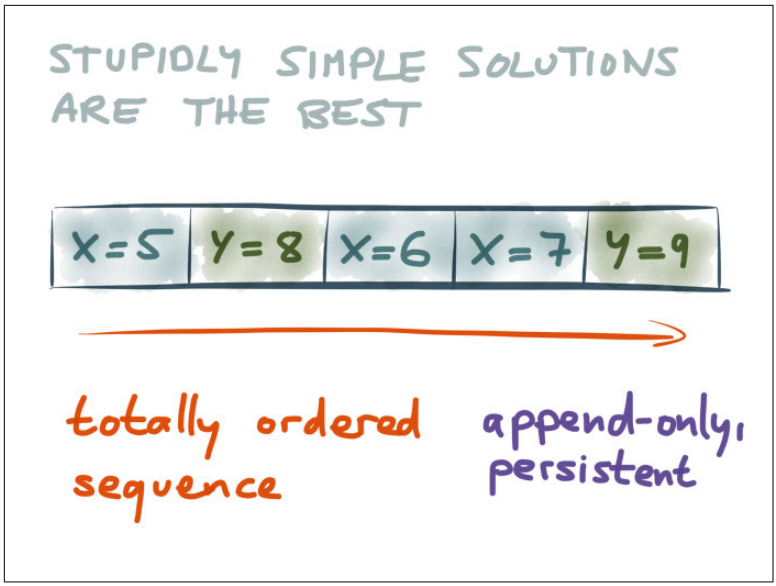


Figure 2-13. A totally ordered, persistently stored sequence of events, also known as a log.

If you do all your writes sequentially, without any concurrency, you have removed the potential for race conditions. Moreover, if you write down the order in which you make your writes, it becomes much easier to recover from partial failures, as I will show later.

So, the stupidly simple solution that I propose looks like this: whenever anyone wants to write some data, we append that write to the end of a sequence of records. That sequence is totally ordered, it's append-only (we never modify existing records, only ever add new records at the end), and it's persistent (we store it durably on disk).

Figure 2-13 shows an example of such a data structure: moving left to right, it records that we first wrote X=5, then we wrote Y=8, then we wrote X=6, and so on. That data structure has a name: we call it a *log*.

The Ubiquitous Log

The interesting thing about logs is that they pop up in many different areas of computing. Although it might seem like a stupidly simple idea that can't possibly work, it actually turns out to be incredibly powerful.

When I say "logs", the first thing you probably think of is textual application logs of the style you might get from Log4j or Syslog. Sure, that's one kind of log, but when I talk about logs here I mean something more general. I mean any kind of data structure of totally ordered records that is append-only and persistent—any kind of *append-only file*.

How Logs Are Used in Practice

Throughout the rest of this chapter, I'll run through a few examples of how logs are used in practice (Figure 2-14). It turns out that logs are already present in the databases and systems you likely use every day. When we understand how logs are used in various different systems, we'll be in a better position to understand how they can help us solve the problem of data integration.

Figure 2-14. Four areas of computing that use logs; we will look at each of them in turn.

The first area we'll discuss is the internals of database storage engines.

1) Database Storage Engines

Do you remember B-Trees[5] from your algorithms classes (Figure 2-15)? They are a very widely used data structure for storage engines—almost all relational databases, and many non-relational databases, use them.

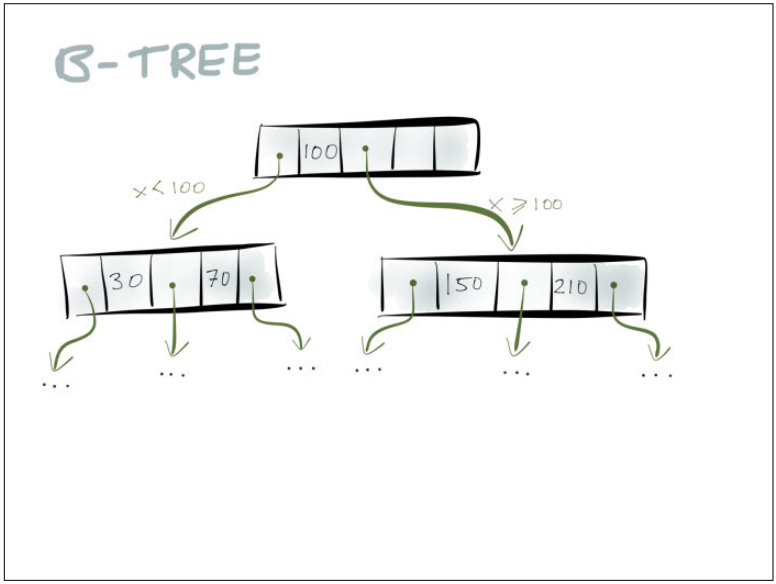


Figure 2-15. The upper levels of a B-Tree.

To summarize briefly: a B-Tree consists of *pages*, which are fixed-size blocks on disk, typically 4 or 8 KB in size. When you want to look up a particular key, you start with one page, which is at the root of the tree. The page contains pointers to other pages, and each pointer is tagged with a range of keys. For example, if your key is between 0 and 100, you follow the first pointer; if your key is between 100 and 300, you follow the second pointer; and so on.

The pointer takes you to another page, which further breaks down the key range into sub-ranges. Eventually you end up at the page containing the particular key for which you're looking.

5 Goetz Graefe: "Modern B-Tree Techniques," *Foundations and Trends in Databases*, volume 3, number 4, pages 203–402, August 2011. doi:10.1561/1900000028

Now what happens if you need to insert a new key/value pair into a B-Tree? You have to insert it into the page whose key range contains the key you're inserting. If there is enough spare space in that page, no problem. But, if the page is full, it needs to be split into two separate pages (Figure 2-16).

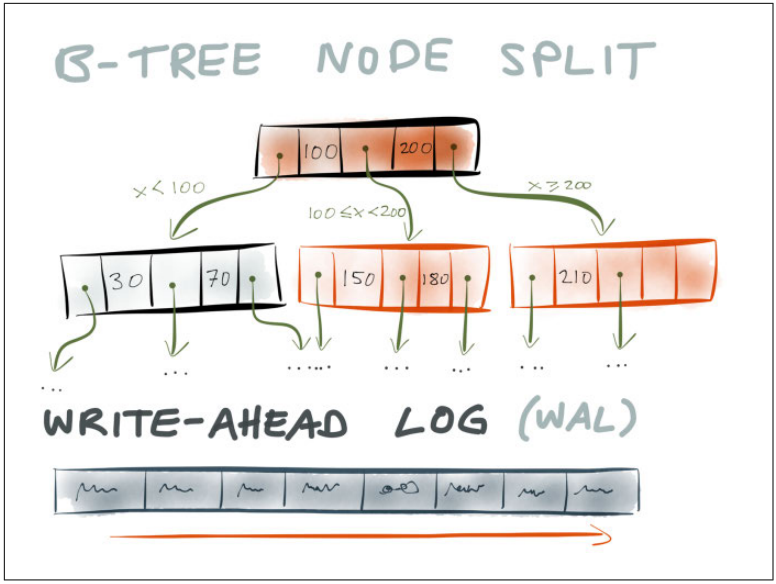


Figure 2-16. Splitting a full B-Tree page into two sibling pages (red outline). Page pointers in the parent (black outline, red fill) need to be updated, too.

When you split a page, you need to write at least three pages to disk: the two pages that are the result of the split, and the parent page (to update the pointers to the split pages). However, these pages might be stored at various different locations on disk.

This raises the question: what happens if the database crashes (or the power goes out, or something else goes wrong) halfway through the operation, after only some of those pages have been written to disk? In that case, you have the old (pre-split) data in some pages, and the new (post-split) data in other pages, and that's bad news. You're most likely going to end up with dangling pointers or pages to which nobody is pointing. In other words, you've got a corrupted index.

Now, storage engines have been doing this for decades, so how do they make B-Trees reliable? The answer is that they use a *write-ahead log*.[6]

Write-ahead log

A write-ahead log (WAL) is a particular kind of log. Whenever the storage engine wants to make any kind of change to the B-Tree, it must *first* write the change that it intends to make to the WAL, which is an append-only file on disk. Only after the change has been written to the WAL, and durably written to disk, is the storage engine allowed to modify the actual B-Tree pages on disk.

This makes the B-Tree reliable: if the database crashes while data was being appended to the WAL, no problem, because the B-Tree hasn't been touched yet. And if it crashes while the B-Tree is being modified, no problem, because the WAL contains the information about what changes were about to happen. When the database comes back up after the crash, it can use the WAL to repair the B-Tree and get it back into a consistent state.

This has been our first example to show that logs are a really neat idea.

Log-structured storage

Storage engines didn't stop with B-Trees. Some clever folks realized that if we're writing everything to a log anyway, we might as well use the log as the primary storage medium. This is known as *log-structured storage*,[7] which is used in HBase[8] and Cassandra,[9] and a variant appears in Riak.[10]

---

6 C Mohan, Don Haderle, Bruce G Lindsay, Hamid Pirahesh, and Peter Schwarz: "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," *ACM Transactions on Database Systems* (TODS), volume 17, number 1, pages 94–162, March 1992. doi:10.1145/128765.128770

7 Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil: "The Log-Structured Merge-Tree (LSM-Tree)," *Acta Informatica*, volume 33, number 4, pages 351–385, June 1996. doi:10.1007/s002360050048

8 Matteo Bertozzi: "Apache HBase I/O – HFile," blog.cloudera.com, 29 June 2012.

9 Jonathan Hui: "How Cassandra Read, Persists Data and Maintain Consistency," jonathanhui.com.

10 Justin Sheehy and David Smith: "Bitcask: A Log-Structured Hash Table for Fast Key/Value Data," Basho Technologies, April 2010.

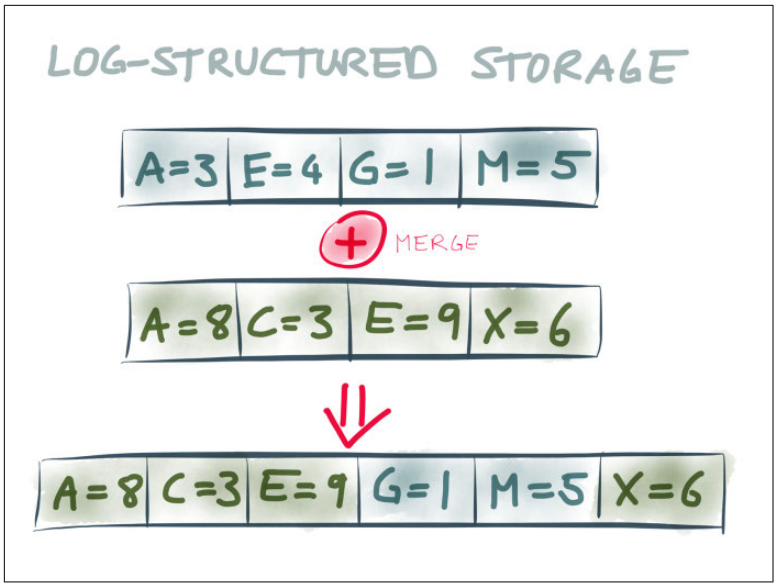


Figure 2-17. In log-structured storage, writes are appended to log segments, and periodically merged/compacted in the background.

In log-structured storage we don't always keep appending to the same file, because it would become too large and it would be too difficult to find the key we're looking for. Instead, the log is broken into *segments*, and from time to time the storage engine merges segments and discards duplicate keys, as illustrated in Figure 2-17. Segments can also be internally sorted by key, which can make it easier to find the key you're looking for and also simplifies merging. However, these segments are still logs: they are only written sequentially, and they are immutable after they have been written.

As you can see, logs play an important role in storage engines.

2) Database Replication

Let's move on to the second example where logs are used: *database replication*.

Replication is a feature that you find in many databases: it allows you to keep a copy of the same data on several different nodes. That can be useful for spreading the load, and it also means that if one node dies, you can failover to another one.

There are a few different ways of implementing replication, but a common choice is to designate one node as the *leader* (also known as *primary* or *master*), and the other replicas as *followers* (also known as *standby* or *slave*) (Figure 2-18). I don't like the master/slave terminology, so I'm going to stick with leader/follower.

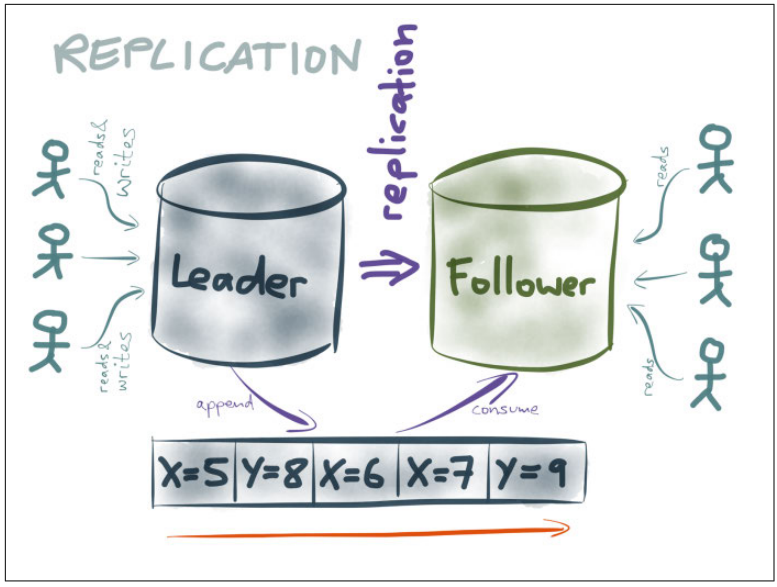


Figure 2-18. In leader-based replication, the leader processes writes, and uses a replication log to tell followers about writes.

Whenever a client wants to write something to the database, it needs to talk to the leader. Read-only clients can use either the leader or the follower (although the follower is typically asynchronous, so it might have slightly out-of-date information if the latest writes haven't yet been applied).

When clients write data to the leader, how does that data get to the followers? Big surprise: they use a log! They use a *replication log*, which may in fact be the same as the write-ahead log (this is what Postgres does, for example), or it may be a separate replication log (MySQL does this).

The replication log works as follows: whenever some data is written to the leader, it is also appended to the replication log. The followers read that log in the order in which it was written, and apply each of the writes to their own copy of the data. As a result, each follower

processes the same writes in the same order as the leader, and thus it ends up with a copy of the same data (Figure 2-19).

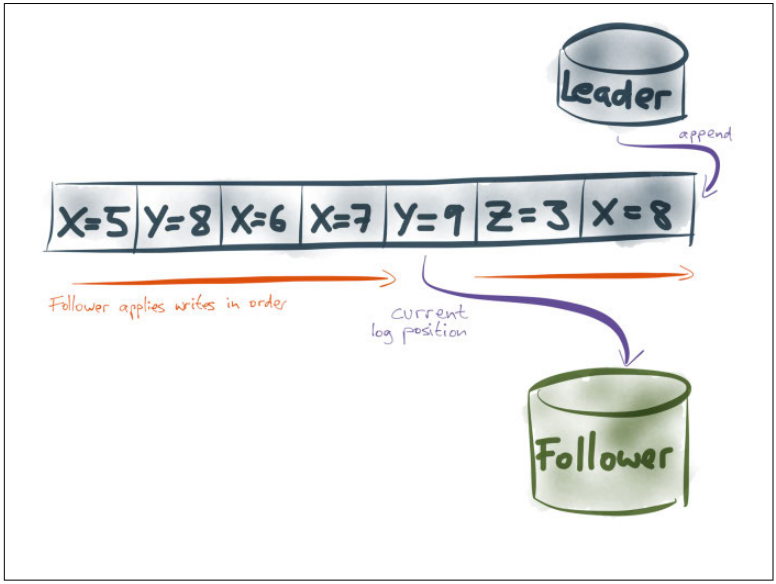


Figure 2-19. The follower applies writes in the order in which they appear in the replication log.

Even if the writes happen concurrently on the leader, the log still contains the writes in a total order. Thus, the log actually *removes* the concurrency from the writes—it "squeezes all the non-determinism out of the stream of writes,"[11] and on the follower there's no doubt about the order in which the writes happened.

So, what about the dual-writes race condition we discussed earlier (Figure 2-9)?

This race condition cannot happen with leader-based replication, because clients don't write directly to the followers. The only writes processed by followers are the ones they receive from the replication log. And because the log fixes the order of those writes, there is no ambiguity regarding which one happened first.

11 Jay Kreps: "The Log: What every software engineer should know about real-time data's unifying abstraction," engineering.linkedin.com, 16 December 2013.

Moreover, all followers are guaranteed to see the log in the same order, so if two overwrites occur in quick succession, that's no problem: all followers apply writes in that same order, and as a result they all end up in the same final state.

But, what about the second problem with dual writes that we discussed earlier, namely that one write could succeed and another could fail (Figure 2-11)? This could still happen: a follower could successfully process the first write from a transaction, but fail to process the second write from the transaction (perhaps because the disk is full or the network is interrupted, as illustrated in Figure 2-20).

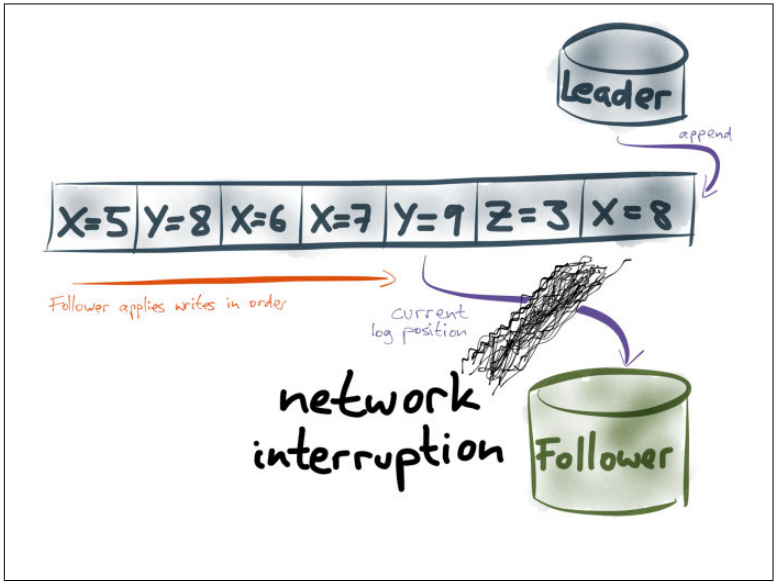


Figure 2-20. A network interruption causes the follower to stop applying writes from the log, but it can easily resume replication when the network is repaired.

If the network between the leader and the follower is interrupted, the replication log cannot flow from the leader to the follower. This could lead to an inconsistent replica, as we discussed previously. How does database replication recover from such errors and avoid becoming inconsistent?

Notice that the log has a very nice property: because the leader only ever appends to it, we can give each record in the log a sequential

number that is always increasing (which we might call *log position* or *offset*). Furthermore, followers only process it in sequential order (from left to right; i.e., in order of increasing log position), so we can describe a follower's current state with a single number: the position of the latest record it has processed.

When you know a follower's current position in the log, you can be sure that all the prior records in the log have already been processed, and none of the subsequent records have been processed.

This is great, because it makes error recovery quite simple. If a follower becomes disconnected from the leader, or it crashes, the follower just needs to store the log position up to which it has processed the replication log. When the follower recovers, it reconnects to the leader, and asks for the replication log beginning from the last offset that it previously processed. Thus, the follower can catch up on all the writes that it missed while it was disconnected, without losing any data or receiving duplicates.

The fact that the log is totally ordered makes this recovery much simpler than if you had to keep track of every write individually.

3) Distributed Consensus

The third example of logs in practice is in a different area: distributed consensus.

Achieving consensus is one of the well-known and often-discussed problems in distributed systems. It is important, but it is also surprisingly difficult to solve.

An example of consensus in the real world would be trying to get a group of friends to agree on where to go for lunch (Figure 2-21). This is a distinctive feature of a sophisticated civilization[12] and can be a surprisingly difficult problem, especially if some of your friends are easily distractible (so they don't always respond to your questions) or if they are fussy eaters.

12 Douglas Adams: *The Restaurant at the End of the Universe*. Pan Books, 1980. ISBN: 9780330262132

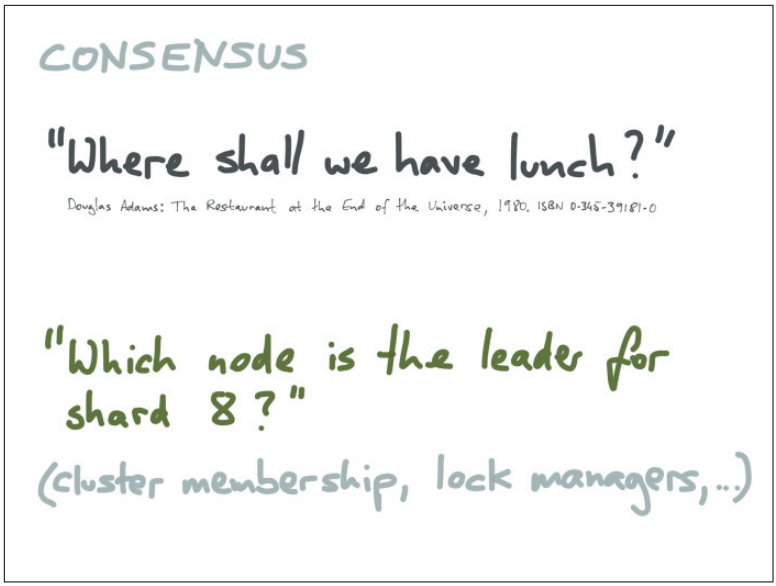


Figure 2-21. Consensus is useful if you don't want to stay hungry, and don't want to lose data.

Closer to our usual domain of computers, an example of where you might want consensus is in a distributed database system: for instance, you might require all your database nodes to agree on which node is the leader for a particular partition (shard) of the database.

It's pretty important that they all agree on whom the leader is: if two different nodes both think they are leader, they might both accept writes from clients. Later, when one of them finds out that it was wrong and it wasn't leader after all, the writes that it accepted might be lost. This situation is known as *split brain*, and it can cause nasty data loss.[13]

13 Kyle Kingsbury: "Call me maybe: MongoDB," aphyr.com, 18 May 2013.

There are a few different algorithms for implementing consensus. Paxos[14] is perhaps the most well-known, but there are also Zab[15] (used by ZooKeeper[16]), Raft,[17] and others.[18] These algorithms are quite tricky and have some non-obvious subtleties.[19] In this report, I will very briefly sketch one part of the Raft algorithm (Figure 2-22).

14 Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone: "Paxos Made Live - An Engineering Perspective," at *26th ACM Symposium on Principles of Distributed Computing* (PODC), June 2007.

15 Flavio P Junqueira, Benjamin C Reed, and Marco Serafini: "Zab: High-performance broadcast for primary-backup systems," at *41st IEEE International Conference on Dependable Systems and Networks* (DSN), pages 245–256, June 2011. doi:10.1109/DSN.2011.5958223

16 "Apache ZooKeeper," Apache Software Foundation, zookeeper.apache.org.

17 Diego Ongaro and John K Ousterhout: "In Search of an Understandable Consensus Algorithm (Extended Version)," at *USENIX Annual Technical Conference* (USENIX ATC), June 2014.

18 Robbert van Renesse, Nicolas Schiper, and Fred B Schneider: "Vive La Différence: Paxos vs. Viewstamped Replication vs. Zab," *IEEE Transactions on Dependable and Secure Computing*, volume 12, number 4, pages 472–484, September 2014. doi:10.1109/TDSC.2014.2355848

19 Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft: "Raft Refloated: Do We Have Consensus?," *ACM SIGOPS Operating Systems Review*, volume 49, number 1, pages 12–21, January 2015. doi:10.1145/2723872.2723876

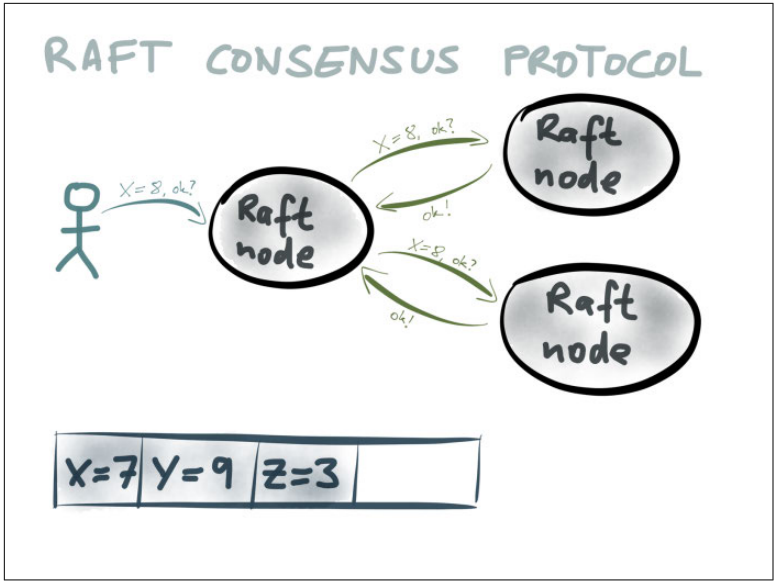


Figure 2-22. Raft consensus protocol: a value X=8 is proposed, and nodes vote on it.

In a consensus system, there are a number of nodes (three in Figure 2-22) which are in charge of agreeing what the value of a particular variable should be. A client proposes a value, for example X=8 (which might mean that node X is the leader for partition 8), by sending it to one of the Raft nodes. That node collects votes from the other nodes. If a majority of nodes agree that the value should be X=8, the first node is allowed to commit the value.

When that value is committed, what happens? In Raft, that value is appended to the end of a log. Thus, what Raft is doing is not just getting the nodes to agree on one particular value, it's actually building up a log of values that have been agreed over time. All Raft nodes are guaranteed to have exactly the same sequence of committed values in their log, and clients can consume this log (Figure 2-23).

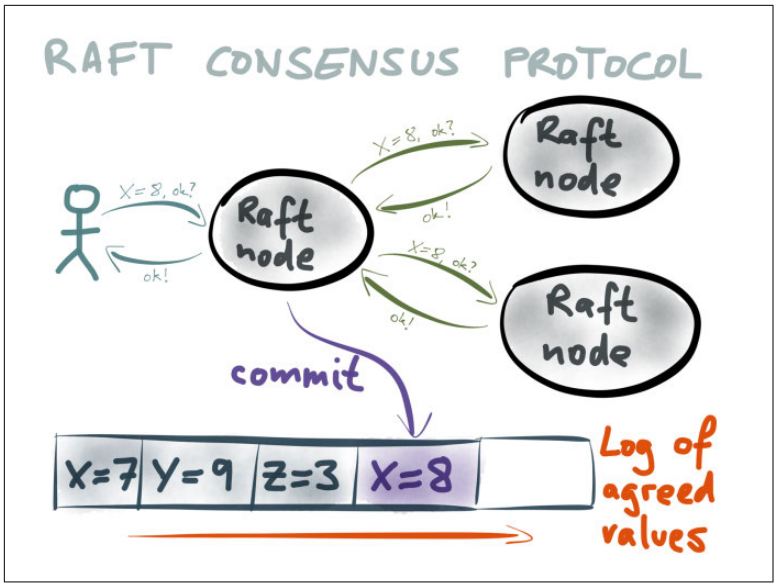


Figure 2-23. The Raft protocol provides consensus not just for a single value, but a log of agreed values.

After the newly agreed value has been committed, appended to the log, and replicated to the other nodes, the client that originally proposed the value X=8 is sent a response saying that the system succeeded in reaching consensus, and that the proposed value is now part of the Raft log.

(As a theoretical aside, the problems of consensus and *atomic broadcast*—that is, creating a log with exactly-once delivery—are reducible to each other.[20] This means Raft's use of a log is not just a convenient implementation detail, but also reflects a fundamental property of the consensus problem it is solving.)

4) Kafka

We've seen that logs are a recurring theme in surprisingly many areas of computing: storage engines, database replication, and consensus. As the fourth and final example, we'll cover Apache Kafka—

20 Tushar Deepak Chandra and Sam Toueg: "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, volume 43, number 2, pages 225–267, March 1996. doi:10.1145/226643.226647

another system that is built around the idea of logs. The interesting thing about Kafka is that it doesn't hide the log from you. Rather than treating the log as an implementation detail, Kafka exposes it to you so that you can build applications around it.

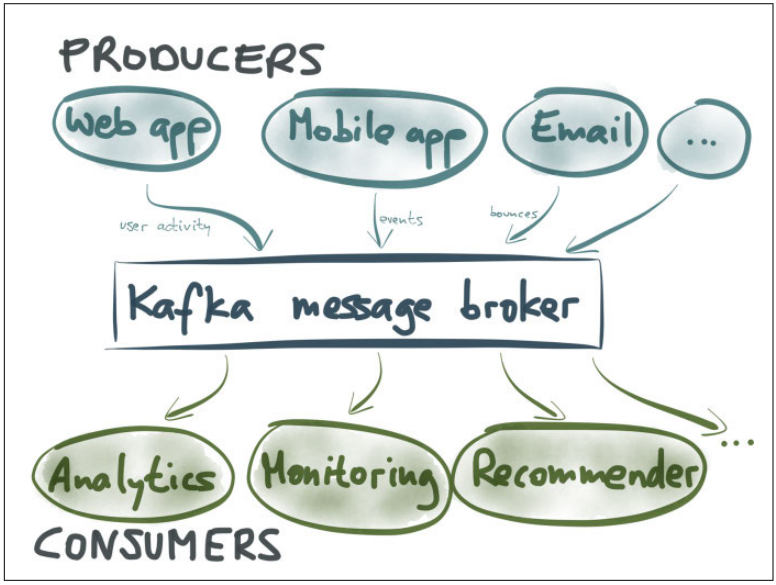


Figure 2-24. Kafka is typically used as a message broker for publish-subscribe event streams.

The typical use of Kafka is as a message broker (message queue), as illustrated in Figure 2-24—so it is somewhat comparable to AMQP (e.g. RabbitMQ), JMS (e.g. ActiveMQ or HornetQ), and other messaging systems. Kafka has two types of clients: *producers* or *publishers* (which send messages to Kafka) and *consumers* or *subscribers* (which read the streams of messages in Kafka).

For example, producers can be your web servers or mobile apps, and the types of messages they send to Kafka might be logging information—that is, events that indicate which user clicked which link at which point in time. The consumers are various processes that need to find out about stuff that is happening; for example, to generate analytics, to monitor for unusual activity, to generate personalized recommendations for users, and so on.

The thing that makes Kafka interestingly different from other message brokers is that it is structured as a log. In fact, it somewhat

resembles a log file in the sense of Log4j or Syslog: when a producer sends a message to Kafka, it is literally appended to the end of a file on disk. Thus, Kafka's internal data files are just a sequence of log messages, as illustrated in Figure 2-25. (While application log files typically use a newline character to delimit records, Kafka uses a binary format with checksums and a bit of useful metadata. But the principle is very similar.)

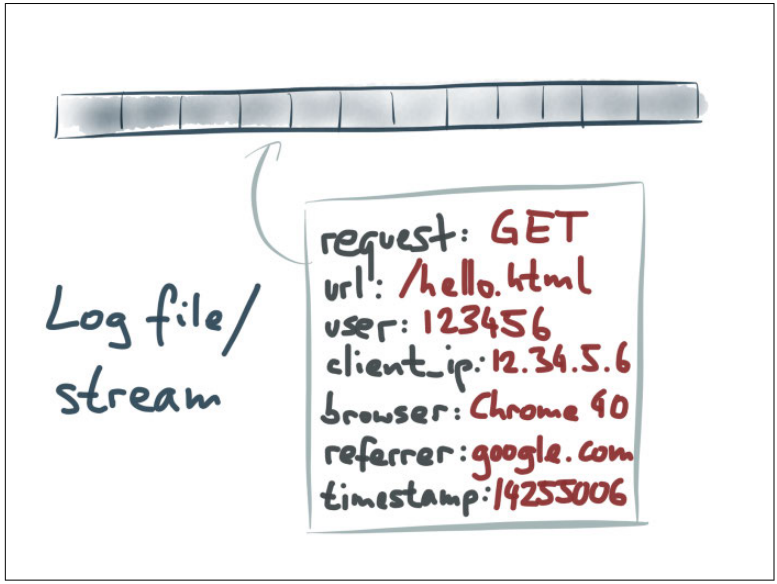


Figure 2-25. A message in Kafka is appended as a log record to the end of a file.

If Kafka wrote everything sequentially to a single file, its throughput would be limited to the sequential write throughput of a disk—which is perhaps tens of megabytes per second, but that's not enough. In order to make Kafka scalable, a stream of messages—a *topic*—is split into *partitions* (Figure 2-26). Each partition is a log, that is, a totally ordered sequence of messages. However, different partitions are completely independent from one another, so there is no ordering guarantee across different partitions. This allows different partitions to be handled on different servers, and so Kafka can scale horizontally.

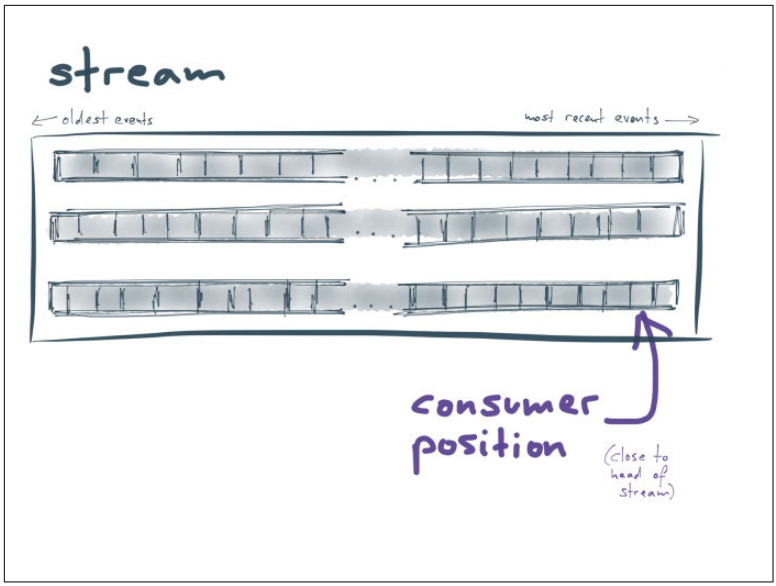


Figure 2-26. Data streams in Kafka are split into partitions.

Each partition is stored on disk and replicated across several machines, so it is durable and can tolerate machine failure without data loss. Producing and consuming logs is very similar to what we saw previously in the context of database replication:

- Every message that is sent to Kafka is appended to the end of a partition. That is the only write operation supported by Kafka: appending to the end of a log. It's not possible to modify past messages.
- Within each partition, messages have a monotonically increasing *offset* (log position). To consume messages from Kafka, a client reads messages sequentially, beginning from a particular offset, as indicated by the violet arrow in Figure 2-26. That offset is managed by the consumer.

We said previously that Kafka is a message broker somewhat like AMQP or JMS messaging systems. However, the similarity is superficial—although they all allow messages to be relayed from producers to consumers, the implementation under the hood is very different.

The biggest difference is in how the system ensures that consumers process every message, without dropping messages in case of failure. With AMQP and JMS-based queues, the consumer acknowledges every individual message after it has been successfully processed. The broker keeps track of the acknowledgement status of every message; if a consumer dies without acknowledging a message, the broker retries delivery, as shown in Figure 2-27.

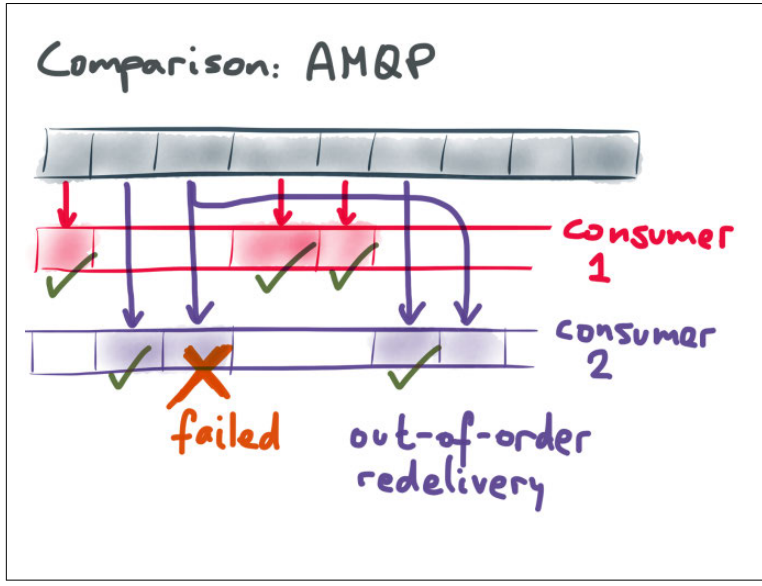


Figure 2-27. AMQP and JMS message brokers use per-message acknowledgements to keep track of which messages were successfully consumed, and redeliver any messages on which the consumer failed.

A consequence of this redelivery behavior is that messages can be delivered *out-of-order*: a consumer does not necessarily see messages in exactly the same order as the producer sent the messages. AMQP and JMS are designed for situations in which the exact ordering of messages is not important, and so this redelivery behavior is desirable.

However, in situations like database replication, the ordering of messages is critical. For example, in Figure 2-13 it matters that X is first set to 6 and then to 7, so the final value is 7. If the replication system were allowed to reorder messages, they would no longer mean the same thing.

Kafka maintains a fixed ordering of messages within one partition, and always delivers those messages in the same order. For that reason, Kafka doesn't need to keep track of acknowledgements for every single message: instead, it is sufficient to keep track of the latest message offset that a consumer has processed in each partition. Because the order of messages is fixed, we know that all messages prior to the current offset have been processed, and all messages after the current offset have not yet been processed.

Kafka's model has the advantage that it can be used for database-like applications where the order of messages is important. On the other hand, the consumer offset tracking means that a consumer must process messages sequentially on a single thread. Thus, we can distinguish two different families of messaging systems (Figure 2-28).

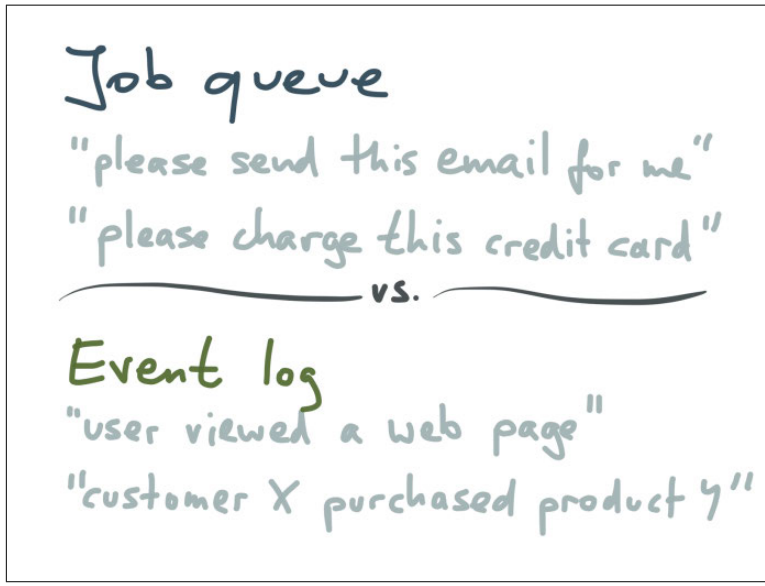


Figure 2-28. AMQP and JMS are good for job queues; Kafka is good for event logs.

On the one hand, message brokers that keep track of acknowledgements for every individual message are well suited for job queues, where one service needs to ask another service to perform some task (e.g. sending an email, charging a credit card) on its behalf. For these situations, the ordering of messages is not important, but it *is* important to be able to easily use a pool of threads to process jobs in parallel and retry any failed jobs.

On the other hand, Kafka shines when it comes to logging events (e.g. the fact that a user viewed a web page, or that a customer purchased some product). When subscribers process these events, it is normally a very lightweight operation (such as storing the event in a database, or incrementing some counters), so it is feasible to process all of the events in one Kafka partition on a single thread. For parallelism—using multiple threads on multiple machines—Kafka consumers can simply spread the data across multiple partitions.

Different tools are good for different purposes, and so it is perfectly reasonable to use both Kafka and a JMS or AMQP messaging system in the same application.

Solving the Data Integration Problem

Let's return to the data integration problem from the beginning of this chapter. Suppose that you have a tangle of different datastores, caches, and indexes that need to be synchronized with each other (Figure 2-3).

Now that we have seen a bunch of examples of practical applications of logs, can we use what we've learned to figure out how to solve data integration in a better way?

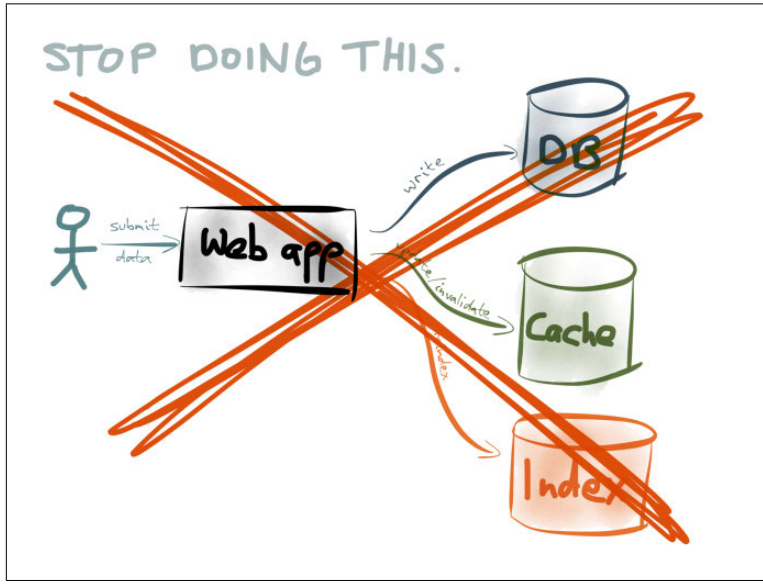


Figure 2-29. Stop doing dual writes—it leads to inconsistent data.

First, we need to stop doing dual writes (Figure 2-29). As discussed, it's probably going to make your data inconsistent, unless you have very carefully thought about the potential race conditions and partial failures that can occur in your application.

Note this inconsistency isn't just a kind of "eventual consistency" that is often quoted in asynchronous systems. What I'm talking about here is permanent inconsistency—if you've written two different values to two different datastores, due to a race condition or partial failure, that difference won't simply resolve itself. You'd have to take explicit actions to search for data mismatches and resolve them (which is difficult because the data is constantly changing).

What I propose is this: rather than having the application write directly to the various datastores, *the application only appends the data to a log* (such as Kafka). All the different representations of this data—your databases, your caches,[21] your indexes—are constructed by consuming the log in sequential order (Figure 2-30).

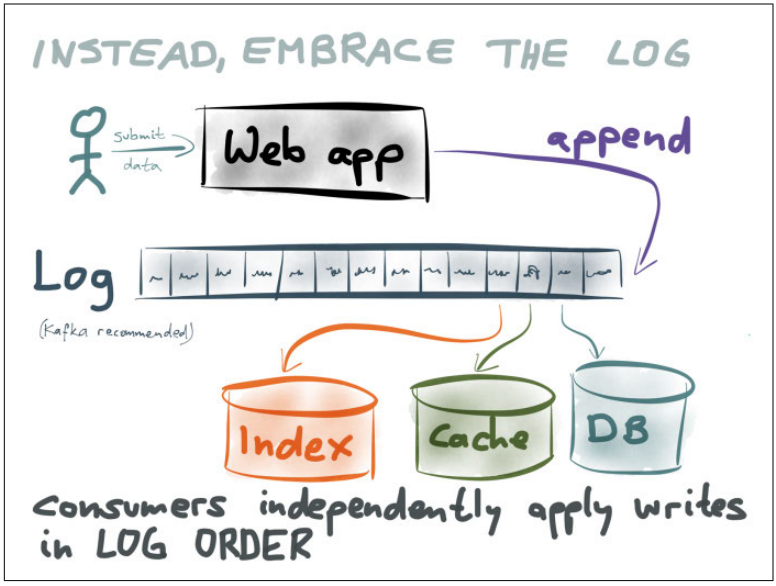


Figure 2-30. Have your application only append data to a log, and all databases, indexes, and caches constructed by reading sequentially from the log.

21 Jason Sobel: "Scaling Out," facebook.com, 20 August 2008.

Each datastore that needs to be kept in sync is an independent consumer of the log. Every consumer takes the data in the log, one record at a time, and writes it to its own datastore. The log guarantees that the consumers all see the records in the same order; by applying the writes in the same order, the problem of race conditions is gone. This looks very much like the database replication we saw earlier!

However, what about the problem of partial failure (Figure 2-11)? What if one of your stores has a problem and can't accept writes for a while?

That problem is also solved by the log: each consumer keeps track of the log position up to which it has processed the log. When the error in the datastore-writing consumer is resolved, it can resume processing records in the log from its last position, and catch up on everything that happened. That way, a datastore won't lose any updates, even if it's offline for a while. This is great for decoupling parts of your system: even if there is a problem in one datastore, the rest of the system remains unaffected.

You can even use the log to bootstrap a completely new cache or index when required. We discuss how this works in Chapter 3.

A log is such a stupidly simple idea: put your writes in a total order and show them to all consumers in the same order. As we saw, this simple idea turns out to be very powerful.

Transactions and Integrity Constraints

Just one problem remains: the consumers of the log all update their datastores asynchronously, so they are eventually consistent. This is not sufficient if you want to guarantee that your data meets certain constraints, for example that each username in your database must be unique, or that a user cannot spend more money than their account balance.

There are a few approaches for solving this issue. One is called change data capture, and we will discuss it in Chapter 3. Another, fairly simple approach is illustrated in Figure 2-31.

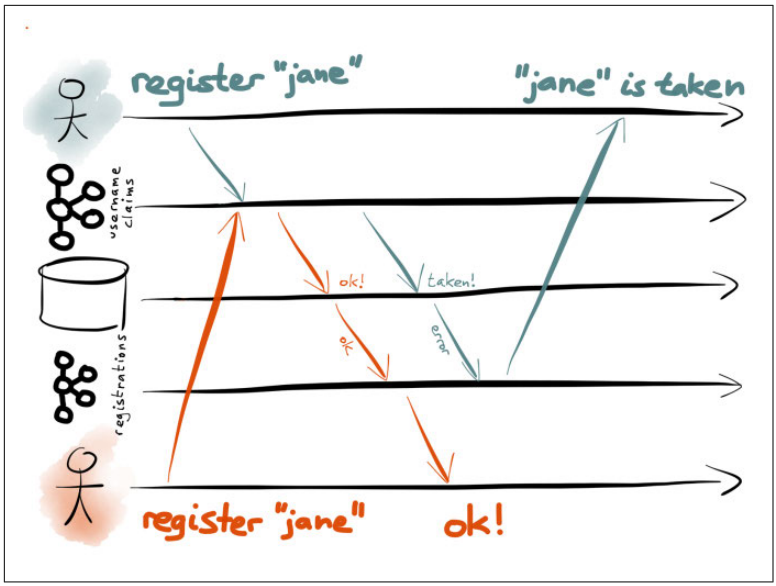


Figure 2-31. Validating that usernames are unique, while still making all writes through a log.

Suppose that you want to ensure that usernames are unique. You can check whether a username is already taken when a user tries to register, but that still allows the race condition of two people trying to claim the same username at just the same time. Traditionally, in a relational database, you'd use transactions and a unique constraint on the username column to prevent this.

When using an architecture in which you can only append to a log, we can solve this problem as a two-step process. First, when a user wants to claim a username, you send an event to a "username claims" stream. This event doesn't yet guarantee uniqueness; it merely establishes an ordering of claims. (If you're using a partitioned stream like a Kafka topic, you need to ensure that all claims to the same username go to the same partition. You can do this by using the username as the Kafka partitioning key.)

A stream processor consumes this stream, checks a database for uniqueness, writes the new username to the database, and then writes the outcome ("successfully registered" or "username already taken") to a separate "registrations" event stream. This validation processor can handle one event at a time, in a single-threaded fashion. To get more parallelism, use more Kafka partitions, each of

which is processed independently—this approach scales to millions of events per second. As the messages in each partition are processed serially, there are no concurrency problems, and conflicting registrations are sure to be found.

How does the user find out whether their username registration was successful? One option is that the server that submitted the claim can consume the "registrations" stream, and wait for the outcome of the uniqueness check to be reported. With a fast stream processor like Samza, this should only take a few milliseconds.

If conflicts are sufficiently rare, it might even be acceptable to tell the user "ok" as soon as the claim has been submitted. In the rare case that their registration failed, you can assign them a temporary random username, send them an email notification to apologize, and ask them to choose a new username.

The same approach can be used to make sure that an account balance does not go negative. For more complex situations, you can layer a transaction protocol on top of Kafka, such as the Tango project from Microsoft Research.[22]

Conclusion: Use Logs to Make Your Infrastructure Solid

To close this chapter, I'd like to leave you with a thought experiment (Figure 2-32).

22 Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, et al.: "Tango: Distributed Data Structures over a Shared Log," at *24th ACM Symposium on Operating Systems Principles* (SOSP), pages 325–340, November 2013. doi:10.1145/2517349.2522732

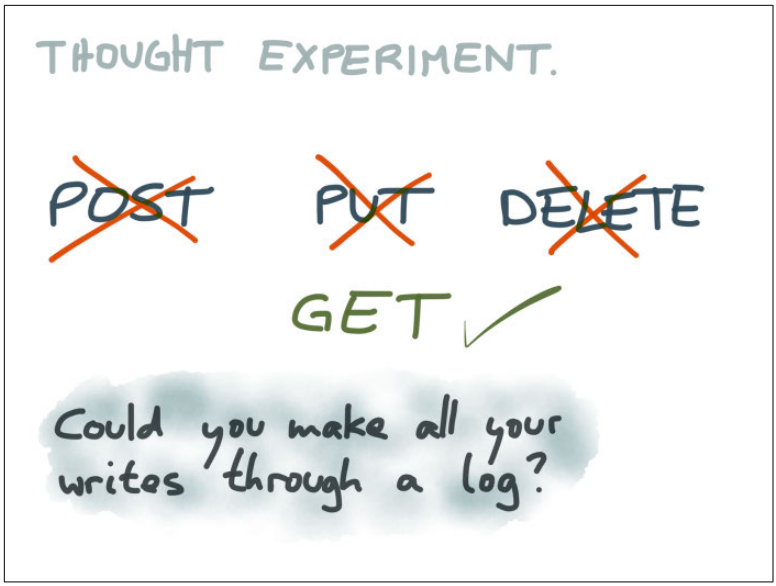


Figure 2-32. What if the only way to modify data in your service was to append an event to a log?

Most APIs we work with have endpoints for both reading and writing. In RESTful terms, GET is for reading (i.e., side-effect-free operations) and POST, PUT, and DELETE are for writing. These endpoints for writing are ok if you only have one system you're writing to, but if you have more than one such system, you quickly end up with dual writes and all their aforementioned problems.

Imagine a system with an API in which you eliminate all the endpoints for writing. Imagine that you keep all the GET requests but prohibit any POST, PUT, or DELETE. Instead, the only way you can send writes into the system is by appending them to a log, and having the system consume that log. (The log must be outside of the system to accommodate several consumers for the same log.)

For example, imagine a variant of Elasticsearch in which you cannot write documents through the REST API, but only write documents by sending them to Kafka. Elasticsearch would internally include a Kafka consumer that takes documents and adds them to the index. This would actually simplify some of the internals of Elasticsearch because it would no longer need to worry about concurrency control, and replication would be simpler to implement. And it would

sit neatly alongside other tools that might be consuming the same log.

In this world view, the log is the authoritative source of what has happened, and consumers of the log present that information in various different ways (Figure 2-33). Similar ideas appear at many different levels of the stack: from wear leveling on SSDs to database storage engines and file systems.[23] We expand on this idea in Chapter 5.

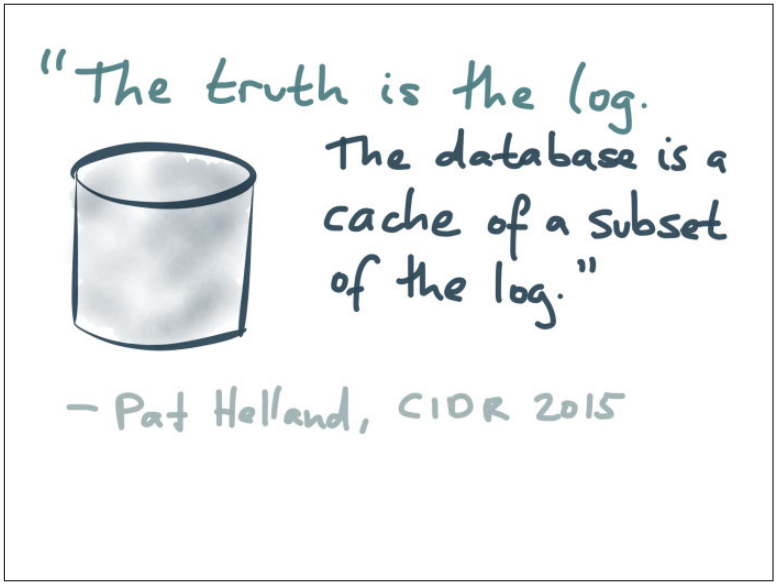


Figure 2-33. The idea of using the log as source of truth appears in various different places.

This is in fact very similar to the Event Sourcing approach we saw in Chapter 1, presented slightly differently. The lesson from this chapter is simple: to make an event-sourced approach work, you need to fix the ordering of the events using a log, because reordering events might lead to a different outcome (e.g., a different person getting the desired username).

In this chapter, we saw that logs are a good way of solving the data integration problem: ensuring that the same data ends up in several

23 Pat Helland: "Immutability Changes Everything," at *7th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2015.

different places, without introducing inconsistencies. Kafka is a good implementation of a log. In the next chapter we will look into the issue of integrating Kafka with your existing databases, so that you can begin integrating them in a log-centric architecture.

Further Reading

Many of the ideas in this chapter were previously laid out by Jay Kreps in his blog post "The Log."[24] An edited version was published as an ebook by O'Reilly Media.[25]

Confluent's vision of a Kafka-based *stream data platform* for data integration closely matches the approach we discussed in this chapter, as described in two blog posts by Jay Kreps.[26,27]

24 Jay Kreps: "The Log: What every software engineer should know about real-time data's unifying abstraction," engineering.linkedin.com, 16 December 2013.

25 Jay Kreps: *I Heart Logs*. O'Reilly Media, September 2014. ISBN: 978-1-4919-0932-4

26 Jay Kreps: "Putting Apache Kafka to use: A practical guide to building a stream data platform (Part 1)," confluent.io, 24 February 2015.

27 Jay Kreps: "Putting Apache Kafka to use: A practical guide to building a stream data platform (Part 2)," confluent.io, 24 February 2015.

CHAPTER 3

Integrating Databases and Kafka with Change Data Capture

The approach we've discussed in the last two chapters has been a radical departure from the way databases are traditionally used: away from transactions that query and update a database in place, and toward an ordered log of immutable events. We saw that this new approach offers many benefits, such as better integration of heterogeneous data systems, better scalability, reliability, and performance.

However, fundamentally changing the way we store and process data is a big, scary step. In reality, most of us have existing systems that we need to keep running and for which a rewrite is not an option. In this chapter, we will discuss a solution for those situations where you already have an existing database as system of record.

Introducing Change Data Capture

As discussed in Chapter 2, if you have data in a database, it's likely that you also need a copy of that data in other places: perhaps in a full-text index (for keyword search), in Hadoop or a data warehouse (for business analytics and offline processing such as recommendation systems), and perhaps in various other caches or indexes (to make reads faster and to take load off the database).

A log is still a great way of implementing this data integration. And if the data source is an existing database, we can simply extract that

log from your database. This idea is called *change data capture* (CDC), illustrated in Figure 3-1.

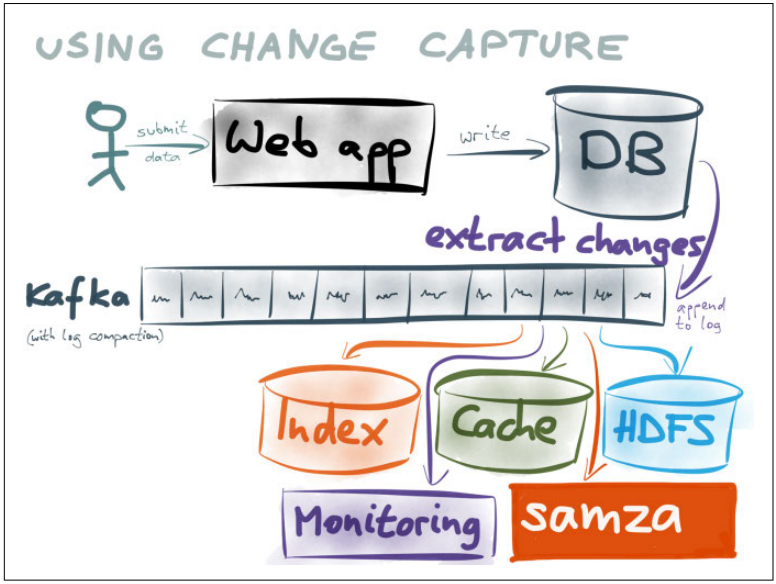


Figure 3-1. Capturing all data changes that are written to a database, and exporting them to a log.

Whereas in Figure 2-30 the application appended events directly to the log, the web app in Figure 3-1 uses the database for reading and writing. If it's a relational database, the application may insert, update, and delete rows arbitrarily, as usual.

The question is: how do we get the data in the database into a log without forcing the web app to change its behavior?

To begin, observe this: most databases have the ability to export a consistent snapshot of the entire database contents (e.g., for backup purposes). For example, MySQL has `mysqldump`, and PostgreSQL has `pg_dump`. If you want a copy of your database in a search index, you could take such a snapshot and then import it into your search server.

However, most databases never stand still: there is always someone writing to them. This means the snapshot is already outdated before you've even finished copying the data. But, maybe you can cope with slightly stale data; in that case you could take snapshots periodically

(e.g., once a day) and update the search index with each new snapshot.

To get more up-to-date information in the search index, you could take snapshots more frequently, although this quickly becomes inefficient: on a large database, it can take hours to make a copy of the entire database and re-index it.

Typically, only a small part of the database changes between one snapshot and the next. What if you could process only a "diff" of what changed in the database since the last snapshot? That would also be a smaller amount of data, so you could take such diffs more frequently. What if you could take such a "diff" every minute? Every second? 100 times a second?

Database = Log of Changes

When you take it to the extreme, the changes to a database become a stream of events. Every time someone writes to the database, that is an event in the stream. If you apply those events to a database in exactly the same order as the original database committed them, you end up with an exact copy of the database.

If you think about it, this is exactly how database replication works (see Chapter 2, Figure 2-19). The leader database produces a replication log—that is, a stream of events that tells the followers what changes they need to make to their copy of the data in order to stay up-to-date with the leader. By continually applying this stream, they maintain a copy of the leader's data.

We want to do the same, except that the follower isn't another instance of the same database software, but a different technology (a search index, cache, data warehouse, etc). Although replication is a common feature in databases, most databases unfortunately consider the replication log to be an implementation detail, not a public API. This means it is often difficult to get access to the replication events in a format that an application can use.

Oracle GoldenGate,[1] the MySQL binlog,[2] the MongoDB oplog,[3] or the CouchDB changes feed[4] do something like this, but they're not exactly easy to use correctly. More recently, a few databases such as RethinkDB[5] or Firebase[6] have oriented themselves toward real-time change streams.

Change Data Capture (CDC) effectively means replicating data from one storage technology to another. To make it work, we need to extract two things from the source database, in an application-readable data format:

- A *consistent snapshot* of the entire database contents at one point in time
- A *real-time stream of changes* from that point onward—every insert, update, or delete needs to be represented in a way that we can apply it to a copy of the data and ensure a consistent outcome.

At some companies, CDC has become a key building block for applications—for example, LinkedIn built Databus[7] and Facebook built Wormhole[8] for this purpose. Kafka 0.9 includes an API called *Kafka Connect*,[9] designed to connect Kafka to other systems, such as databases. A Kafka connector can use CDC to bring a snapshot and stream of changes from a database into Kafka, from where it can be

1 "Oracle GoldenGate 12c: Real-time access to real-time information." Oracle White Paper, oracle.com, March 2015.

2 "5.2.4 The Binary Log," MySQL 5.7 Reference Manual, dev.mysql.com.

3 Manuel Schoebel: "Meteor.js and MongoDB Replica Set for Oplog Tailing," manuel-schoebel.com, 28 January 2014.

4 J Chris Anderson, Jan Lehnardt, and Noah Slater: *CouchDB: The Definitive Guide*. O'Reilly Media, January 2010. ISBN: 978-0-596-15589-6, available online at guide.couchdb.org.

5 Slava Akhmechet: "Advancing the realtime web," rethinkdb.com, 27 January 2015.

6 "Firebase," Google Inc., firebase.com.

7 Shirshanka Das, Chavdar Botev, Kapil Surlaker, et al.: "All Aboard the Databus!," at *ACM Symposium on Cloud Computing* (SoCC), October 2012.

8 Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, et al.: "Wormhole: Reliable Pub-Sub to Support Geo-replicated Internet Services," at *12th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), May 2015.

9 "Kafka Connect," Confluent Platform documentation, docs.confluent.io, December 2015.

used for various applications. Kafka Connect draws from the lessons learnt from Databus and similar systems.

If you have a change stream that goes all the way back to the first ever write to the database, you don't need the snapshot, because the change stream contains the entire contents of the database already. However, most databases delete transaction logs after a while, to avoid running out of disk space. In this case, you need both a one-time snapshot (at the time when you start consuming the change stream) and the change stream (from that point onward) in order to reconstruct the database contents.

Implementing the Snapshot and the Change Stream

Figure 3-2 shows one good approach for getting both the snapshot and the change stream. Users are continually reading and writing to the database, and we want to allow the change capture process to begin without interfering with this (i.e., without downtime).

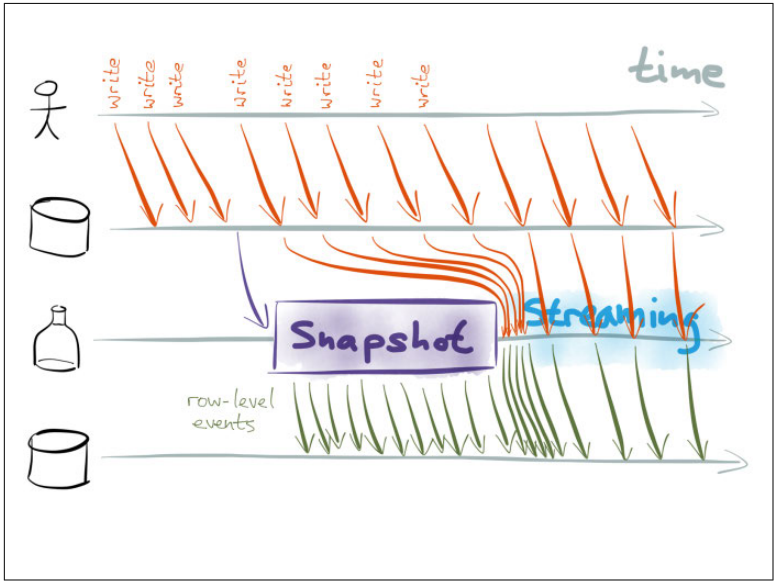


Figure 3-2. Change capture without stopping writes to a database.

Many databases can take a point-in-time snapshot of the database without locking the database for writes (this is implemented by

using the MVCC mechanism in PostgreSQL, MySQL/InnoDB, and Oracle). That is, the snapshot sees the entire database in a consistent state, as it existed at one point in time, even though parts of it may be modified by other transactions while the snapshot is running. This is a great feature, because it would be very difficult to reason about a copy of the database in which some parts are older and other parts are newer.

The change stream needs to be coordinated with this snapshot so that it contains exactly those data changes that occurred since the snapshot was taken, no more and no less. Achieving this is more difficult, and depends on the particular database system you're using. In the next section, we will discuss a particular implementation for PostgreSQL which does this.

With Kafka and Kafka Connect, we can actually unify the snapshot and the change stream into a single event log. The snapshot is translated into a log by generating an "insert row" event for every row in the database snapshot. This is then followed by the change stream, which consists of "insert row," "update row," or "delete row" events. Later in this chapter we will discuss how and why this works.

While the snapshot is being captured (which can take hours on a large database, as previously noted), clients continue writing to the database, as illustrated in Figure 3-2. The change events from these writes must be queued up, and sent to the log when the snapshot is complete. Finally, when the backlog is cleared, the change capture system can just pick up data change events, as and when they happen, and send them to the change log.

The resulting change log has all the good properties that we discussed in Chapter 2, without changing the way the application uses the database. We just need to figure out how to make the change data capture work. That's what the rest of this chapter is about.

Bottled Water: Change Data Capture with PostgreSQL and Kafka

There are many databases to choose from, and the right choice of database depends on the situation. In this section, we'll talk specifi-

cally about *PostgreSQL*[10] (or *Postgres* for short), an open source relational database that is surprisingly full-featured.[11] However, you can draw lessons from the general approach described here and apply them to any other database.

Until recently, if you wanted to get a stream of changes from Postgres, you had to use triggers. This is possible, but it is fiddly, requires schema changes, and doesn't perform very well. However, Postgres 9.4 (released in December 2014) introduced a new feature that changes everything: *logical decoding*.[12]

With logical decoding, change data capture for Postgres suddenly becomes much more feasible. So, when this feature was released, I set out to build a change data capture tool for Postgres that would take advantage of the new facilities. Confluent sponsored me to work on it (thank you, Confluent!), and we have released an alpha version of this tool as open source. It is called *Bottled Water*[13] (Figure 3-3).

At the time of writing, Bottled Water is a standalone tool that copies a consistent snapshot and a stream of changes from Postgres to Kafka. There are plans to integrate it with the Kafka Connect framework for easier deployment.

10 "PostgreSQL," The PostgreSQL Global Development Group, postgresql.org.

11 Peter van Hardenberg: "Postgres: The Bits You Haven't Found," at *Heroku Waza Conference*, 28 February 2013. Recording at vimeo.com.

12 "Chapter 46. Logical Decoding," PostgreSQL 9.4.4 Documentation, postgresql.org.

13 Martin Kleppmann: "Bottled Water for PostgreSQL," Confluent, Inc., github.com, April 2015.

Figure 3-3. Bottled Water is what you get if you take a stream and package it up in a form that's easy to transport and consume.

The name "logical decoding" comes from the fact that this feature decodes the database's write-ahead log (WAL). We encountered the WAL previously in Chapter 2 (Figure 2-16), in the context of making B-Trees robust against crashes. Besides crash recovery, Postgres also uses the WAL for replication. Follower nodes continuously apply the changes from the WAL to their own copy of the database, as if they were constantly recovering from a crash.

This is a good way of implementing replication, but it has a downside: the log records in the WAL are very low-level, describing byte modifications to Postgres' internal data structures. It's not feasible for an application to decode the WAL by itself.

Enter logical decoding: this feature parses the WAL, and gives us access to row-level change events. Every time a row in a table is inserted, updated, or deleted, that's an event. Those events are grouped by transaction, and appear in the order in which they were committed to the database. Aborted/rolled-back transactions do not appear in the stream. Thus, if you apply the change events in the same order, you end up with an exact, transactionally consistent copy of the database—precisely what we want for change capture.

The Postgres logical decoding is well-designed: it even creates a consistent snapshot that is coordinated with the change stream. You can use this snapshot to make a point-in-time copy of the entire database (without locking—you can continue writing to the database while the copy is being made) and then use the change stream to get all writes that happened since the snapshot.

Bottled Water uses these features to extract the entire contents of a database, and encode it using Avro,[14] an efficient binary data format. The encoded data is sent to Kafka, where you can use it in many ways: index it in Elasticsearch, use it to populate a cache, process it with Kafka Streams or a stream processing framework, load it into HDFS with the Kafka HDFS connector,[15] and so on. The nice thing is that you only need to get the data into Kafka once, and then you can have arbitrarily many subscribers, without putting any additional load on Postgres.

Why Kafka?

Kafka is best known for transporting high-volume activity events, such as web server logs, and user click events. Such events are typically retained for a certain period of time (e.g., a few days) and then discarded or archived to long-term storage. Is Kafka really a good fit for database change events? We don't want database data to be discarded!

In fact, Kafka is a perfect fit—the key is Kafka's log compaction feature, which was designed precisely for this purpose (Figure 3-4).

14 "Apache Avro," Apache Software Foundation, avro.apache.org.

15 "HDFS Connector," Confluent Platform 2.0.0 documentation, docs.confluent.io, December 2015.

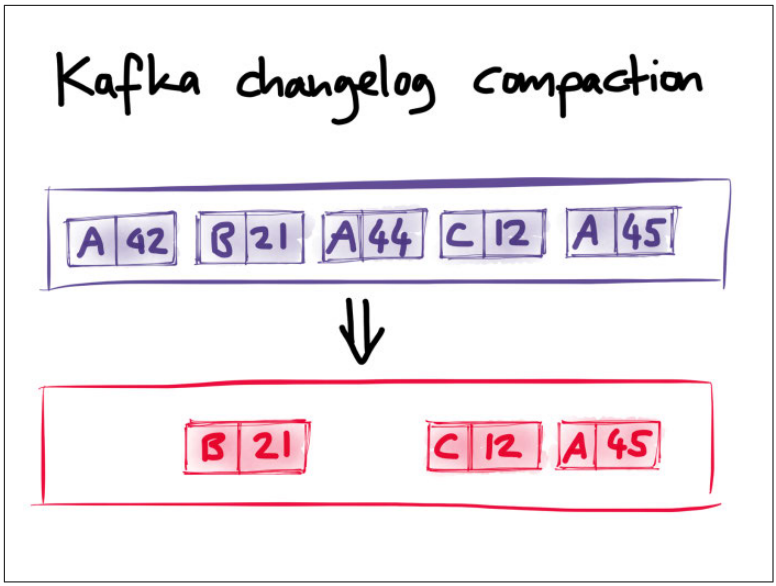


Figure 3-4. Kafka's log compaction rewrites a stream in the background: if there are several messages with the same key, only the most recent is retained, and older messages are discarded.

If you enable log compaction, there is no time-based expiry of data. Instead, every message has a key, and Kafka retains the latest message for a given key indefinitely. Earlier messages for a given key are eventually garbage-collected. This is quite similar to new values overwriting old values in a key-value store and is essentially the same technique as log-structured storage engines use (Figure 2-17).

Bottled Water identifies the primary key (or replica identity[16]) of each table in Postgres and uses that as the key of the messages sent to Kafka. The value of the message depends on the kind of event (Figure 3-5):

- For inserts and updates, the message value contains all of the row's fields, encoded as Avro.

16 Michael Paquier: "Postgres 9.4 feature highlight: REPLICA IDENTITY and logical replication," michael.otacoo.com, 24 April 2014.

- For deletes, the message value is set to null. This causes Kafka to remove the message during log compaction, so its disk space is freed up.

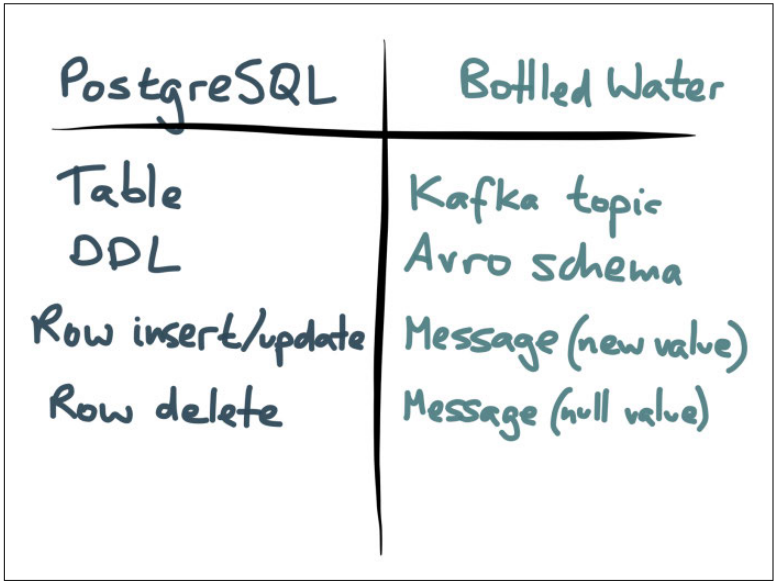


Figure 3-5. Postgres concepts and the way Bottled Water represents them in Kafka.

Each table in Postgres is sent to a separate topic in Kafka. It wouldn't necessarily have to be that way, but this approach makes log compaction work best: in SQL, a primary key uniquely identifies a row in a table, and in Kafka, a message key defines the unit of log compaction in a topic. (Tables with no primary key or replica identity are currently not well supported by logical decoding; this will hopefully be addressed in future versions of Postgres.)

The great thing about log compaction is that it blurs the distinction between the initial snapshot of the database and the ongoing change stream. Bottled Water writes the initial snapshot to Kafka by turning every single row in the database into a message, keyed by primary key, and sending them all to the Kafka brokers. When the snapshot is done, every row that is inserted, updated, or deleted similarly turns into a message.

If a row is frequently updated, there will be many messages with the same key (because each update turns into a message). Fortunately,

Kafka's log compaction will sort this out and garbage-collect the old values so that we don't waste disk space. On the other hand, if a row is never updated or deleted, it just stays unchanged in Kafka forever—it is never garbage-collected.

This means that with log compaction, every row that exists in the database also exists in Kafka—it is only removed from Kafka after it is overwritten or deleted in the database. In other words, the Kafka topic contains a *complete copy of the entire database* (Figure 3-6).

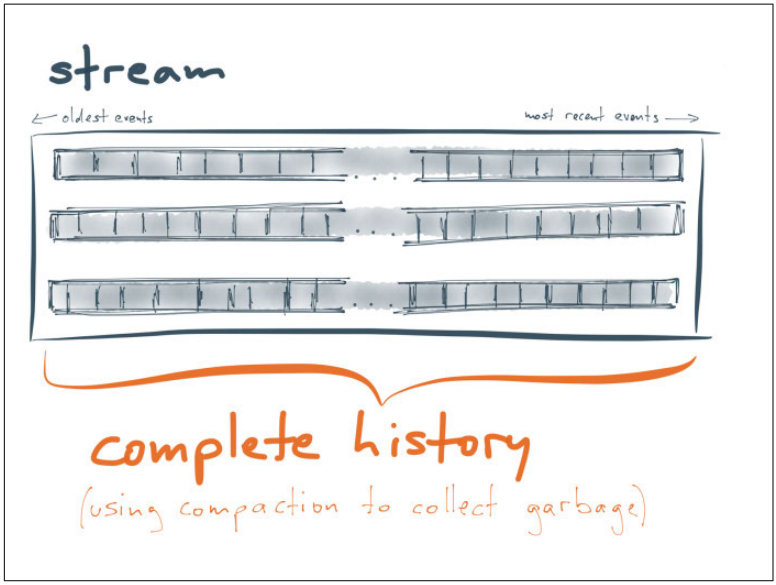


Figure 3-6. When log compaction is enabled, Kafka only removes a message if it is overwritten by another message with the same key; otherwise, it is retained indefinitely.

Having the full database dump and the real-time stream in the same system (Kafka) is tremendously powerful because it allows you to bootstrap new consumers by loading their contents from the log in Kafka.

For example, suppose that you're feeding a database into Kafka by using Bottled Water and you currently have a search index that you're maintaining by consuming that Kafka topic. Now suppose that you're working on a new application feature for which you need to support searching on a new field that you are currently not indexing.

In a traditional setup, you would need to somehow go through all of your documents and re-index them with the new field. Doing this at the same time as processing live updates is dangerous, because you might end up overwriting new data with older data.

If you have a full database dump in a log-compacted Kafka topic, this is no problem. You just create a new, completely empty index, and start your Kafka consumer from the *beginning* of the topic (also known as "offset 0"), as shown in Figure 3-7.

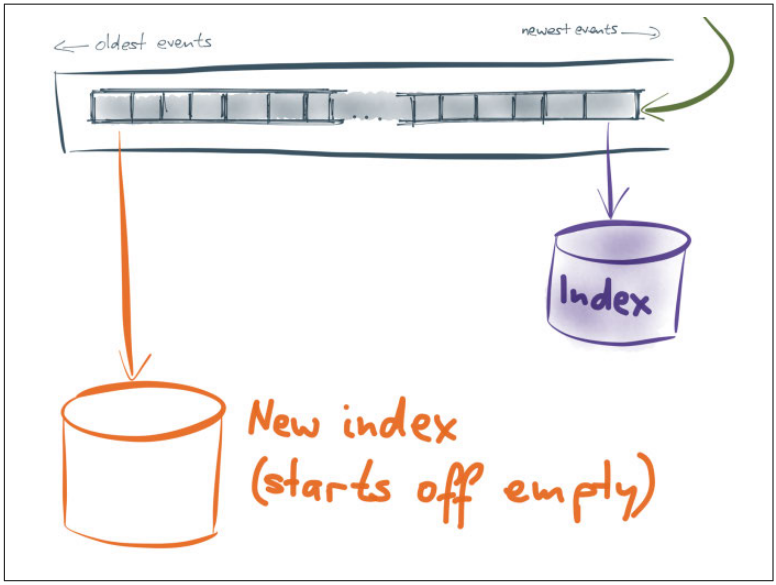


Figure 3-7. To build a new index or view of the data in a Kafka topic, consume the topic from the beginning.

Your consumer then gradually works its way forward through the topic, sequentially processing each message in order and writing it to the new index (including the new field). While this is going on, the old index is still being maintained as usual—it is completely unaffected by the new index being built at the same time. Users' reads are being handled by the old index.

Finally, after some time, the new index reaches the latest message in the topic (Figure 3-8). At this point, nothing special happens—it just continues consuming messages as they appear in Kafka, the same as it was doing before. However, we have done a great thing: we have

created a new index that contains all the data in the topic, and thus all the data in the database!

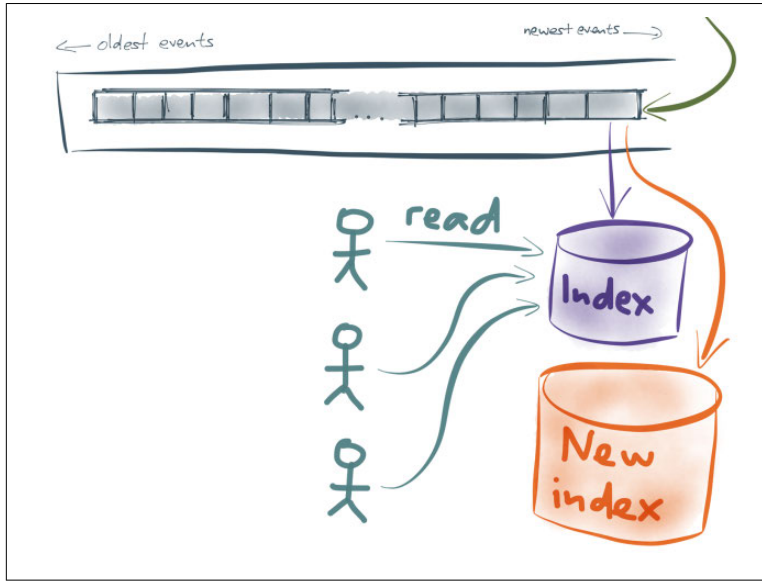


Figure 3-8. While building the new index, users can continue reading from the old index. When the new index is ready, you can switch over users at your leisure.

You now have two full indexes of the data, side by side: the old one and the new one, both being kept current with real-time updates from Kafka. Users are still reading from the old index, but as soon as you have tested the new index, you can switch users from the old index to the new one. Even this switch can be gradual, and you can always go back in case something goes wrong; the old index is still there, still being maintained.

After all users have been moved to the new index and you have assured yourself that everything is fine, you can stop updating the old index, shut it down and reclaim its resources.

This approach avoids a large, dangerous data migration, and replaces it with a gradual process that takes small steps. At each step you can always go back if something went wrong, which can give you much greater confidence about proceeding. This approach "minimi-

zes irreversibility" (as Martin Fowler puts it[17]), which allows you to move faster and be more agile without breaking things.

Moreover, you can use this technique to recover from bugs. Suppose that you deploy a bad version of your application that writes incorrect data to a database. In a traditional setup, where the application writes directly to the database, it is difficult to recover (restoring from a backup would most likely incur data loss). However, if you're going via a log and the bug is downstream from the log, you can recover by using the same technique as just described: process all the data in the log again using a bug-fixed version of the code. Being able to recover from incorrectly written data by re-processing is sometimes known as *human fault-tolerance*.[18]

The idea of maintaining a copy of your database in Kafka surprises people who are more familiar with traditional enterprise messaging and its limitations. Actually, this use case is exactly why Kafka is built around a replicated log: it makes this kind of large-scale data retention and distribution possible. Downstream systems can reload and re-process data at will without impacting the performance of the upstream database that is serving low-latency queries.

Why Avro?

When Bottled Water extracts data from Postgres, it could be encoded as JSON, or Protocol Buffers, or Thrift, or any number of formats. However, I believe Avro is the best choice. Gwen Shapira has written about the advantages of Avro for schema management,[19] and I've written a blog post comparing it to Protobuf and Thrift.[20] The Confluent stream data platform guide[21] gives some more reasons why Avro is good for data integration.

17 Daniel Bryant: "Agile Architecture: Reversibility, Communication and Collaboration," infoq.com, 4 May 2015.

18 Nathan Marz: "How to beat the CAP theorem," nathanmarz.com, 13 October 2011.

19 Gwen Shapira: "The problem of managing schemas," radar.oreilly.com, 4 November 2014.

20 Martin Kleppmann: "Schema evolution in Avro, Protocol Buffers and Thrift," martin.kleppmann.com, 5 December 2012.

21 Jay Kreps: "Putting Apache Kafka to use: A practical guide to building a stream data platform (Part 2)," confluent.io, 24 February 2015.

Bottled Water inspects the schema of your database tables and automatically generates an Avro schema for each table. The schemas are automatically registered with Confluent's schema registry,[22] and the schema version is embedded in the messages sent to Kafka. This means it "just works" with the stream data platform's serializers: you can work with the data from Postgres as meaningful application objects and rich datatypes, without writing a lot of tedious parsing code.

The Logical Decoding Output Plug-In

Now that we've examined Bottled Water's use of Kafka log compaction and Avro data encoding, let's have a little peek into the internals of its integration with Postgres, and see how it uses the logical decoding feature.

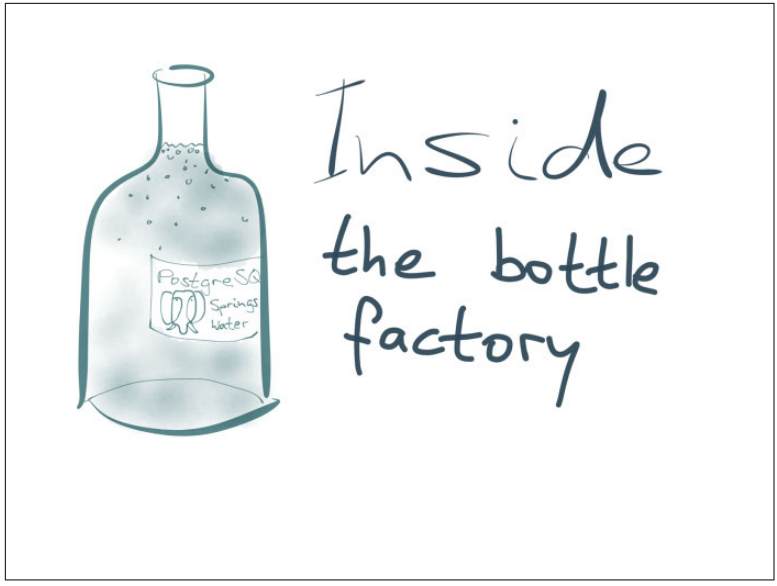


Figure 3-9. How the sausage is made—or rather, the water is bottled.

An interesting property of Postgres' logical decoding feature is that it does not define a wire format in which change data is sent over the

22 "Schema Registry," Confluent Platform Documentation, docs.confluent.io.

network to a consumer. Instead, it defines an output plug-in API[23] that receives a function call for every inserted, updated, or deleted row. Bottled Water uses this API to read data in the database's internal format, and serializes it to Avro.

The output plug-in must be written in C, using the Postgres extension mechanism, and then loaded into the database server as a shared library (Figure 3-10). This requires superuser privileges and filesystem access on the database server, so it's not something to be undertaken lightly. I understand that many a database administrator will be scared by the prospect of running custom code inside the database server. Unfortunately, this is the only way logical decoding can currently be used.

At the moment, the logical decoding plug-in must be installed on the leader database. In principle, it would be possible to have it run on a separate follower so that it cannot impact other clients, but the current implementation in Postgres does not allow this. This limitation will hopefully be lifted in future versions of Postgres.

23 "Chapter 46. Logical Decoding," PostgreSQL 9.4.4 Documentation, postgresql.org.

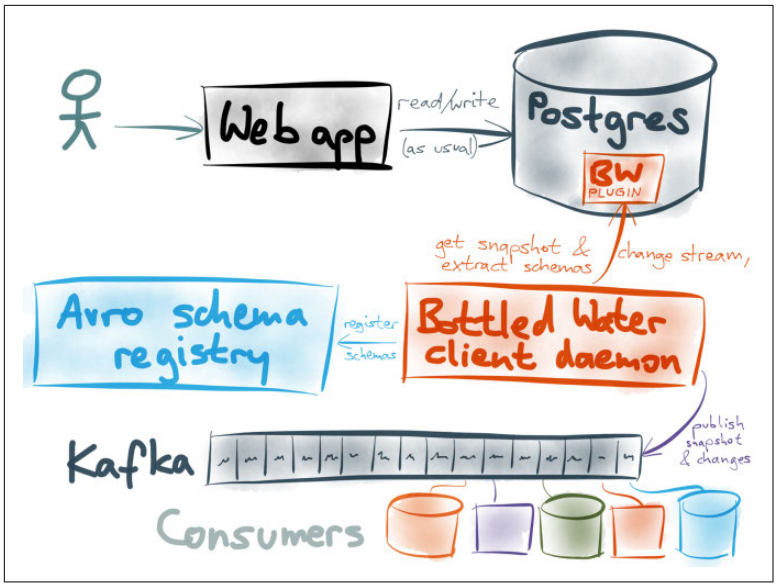


Figure 3-10. The Bottled Water plug-in runs inside the database server. The client daemon connects to it, sends schemas to the registry, and sends data to Kafka.

The Client Daemon

Besides the plug-in (which runs inside the database server), Bottled Water consists of a client program which you can run anywhere. It connects to the Postgres server and to the Kafka brokers, receives the Avro-encoded data from the database, and forwards it to Kafka.

The client is also written in C because it's easiest to use the Postgres client libraries that way, and because some code is shared between the plug-in and the client. It's fairly lightweight and doesn't need to write to disk. At the time of writing, work is underway to integrate the Bottled Water client with the Kafka Connect framework.

What happens if the client crashes or is disconnected from either Postgres or Kafka? No problem: it keeps track of which messages have been published and acknowledged by the Kafka brokers. When the client restarts after an error, it replays all messages that haven't been acknowledged. Thus, some messages could appear twice in Kafka, but no data should be lost. Log compaction will eventually remove the duplicated messages.

Concurrency

One more question remains: what happens if several clients are concurrently writing to the database (Figure 3-11)? How is the result of those writes reflected in the change stream that is sent to Kafka? What happens if a transaction writes some data and then aborts before committing?

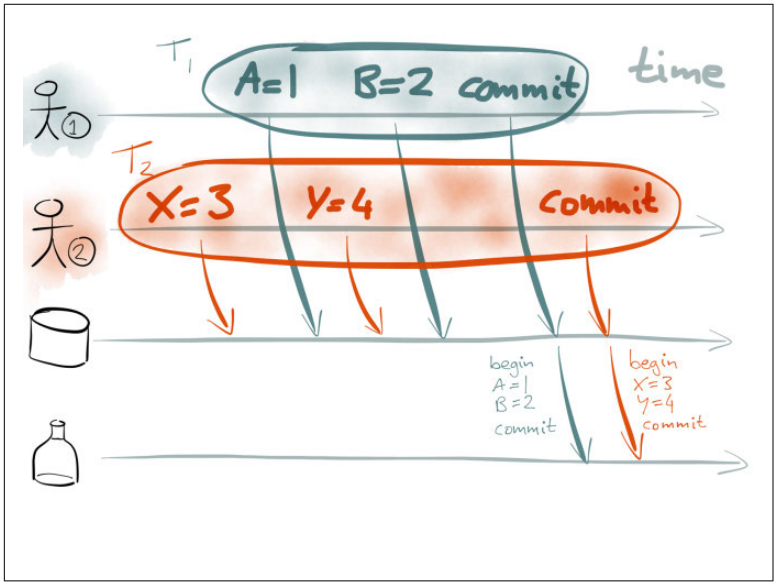


Figure 3-11. Two transactions concurrently write to the database, but Bottled Water only sees the changes when they are committed, in the order in which they are committed.

Fortunately, in the case of Bottled Water, PostgreSQL's logical decoding API offers a simple answer: all of the writes made during a transaction are exposed to the logical decoding API at the same time, at the time the transaction commits. This means Bottled Water doesn't need to worry about aborted transactions (it won't even see any writes made by a transaction that subsequently aborts) or about ordering of writes.

PostgreSQL's transaction isolation semantics ensure that if you apply writes in the order in which they were committed, you get the right result. However, the WAL may actually contain interleaved writes from several different transactions. Thus, while decoding the WAL,

the logical decoding feature needs to reorder those writes so that they appear in the order of transaction commit.

Postgres makes this particular aspect of change data capture easy. If you are implementing change data capture with another database, you may need to deal with these concurrency issues yourself.

Status of Bottled Water

At present, Bottled Water is alpha-quality software. Quite a bit of care has gone into its design and implementation, but it hasn't yet been run in any production environment. However, with some testing and tweaking it will hopefully become production-ready in future. We released it as open source early, in the hope of getting feedback from the community; the response and the number of contributions from the community has been encouraging. When integrated with Kafka Connect, it will hopefully become a fully supported part of the Kafka ecosystem.

I'm excited about change capture because it allows you to unlock the value in the data you already have and makes your architecture more agile by reducing irreversibility. Getting data out of databases and into a stream data platform[24] allows you to combine it with event streams and data from other databases in real time.

In the next chapter, we will see how this approach of building systems resembles the design of Unix, which has been successful for approximately 40 years and is still going strong.

24 Jay Kreps: "Putting Apache Kafka to use: A practical guide to building a stream data platform (Part 2)," confluent.io, 24 February 2015.

CHAPTER 4

The Unix Philosophy of Distributed Data

Contemporary software engineering still has a lot to learn from the 1970s. As we're in such a fast-moving field, we often have a tendency of dismissing older ideas as irrelevant—and consequently, we end up having to learn the same lessons over and over again, the hard way. Although computers have become faster, data has grown bigger, and requirements have become more complex, many old ideas are actually still highly relevant today.

In this chapter, I'd like to highlight one particular set of old ideas that I think deserves more attention today: the Unix philosophy. I'll show how this philosophy is very different from the design approach of mainstream databases.

In fact, you can consider Kafka and stream processing to be a twenty-first-century reincarnation of Unix pipes, drawing lessons from the design of Unix and correcting some historical mistakes. Lessons learned from the design of Unix can help us to create better application architectures that are easier to maintain in the long run.

Let's begin by examining the foundations of the Unix philosophy.

Simple Log Analysis with Unix Tools

You've probably seen the power of Unix tools before—but to get started, let me give you a concrete example that we can talk about. Suppose that you have a web server that writes an entry to a log file

every time it serves a request. For example, using the nginx default access log format, one line of the log might look like the following (this is actually one line; it's only broken up into multiple lines here for readability):

```
216.58.210.78 - - [27/Feb/2015:17:55:11 +0000] "GET
/css/typography.css HTTP/1.1" 200 3377
"http://martin.kleppmann.com/" "Mozilla/5.0 (Macintosh;
Intel Mac OS X 10_9_5) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/40.0.2214.115 Safari/537.36"
```

This line of the log indicates that on 27 February, 2015 at 17:55:11 UTC, the server received a request for the file `/css/typography.css` from the client IP address `216.58.210.78`. It then goes on to note various other details, including the browser's user-agent string.

Various tools can take these log files and produce pretty reports about your website traffic, but for the sake of the exercise, let's build our own, using basic Unix tools. Let's determine the five most popular URLs on our website. To begin, we need to extract the path of the URL that was requested, for which we can use `awk`.

`awk` doesn't know about the format of nginx logs—it just treats the log file as text. By default, `awk` takes one line of input at a time, splits it by whitespace, and makes the whitespace-separated components available as variables `$1`, `$2`, and so on. In the nginx log example, the requested URL path is the seventh whitespace-separated component (Figure 4-1).

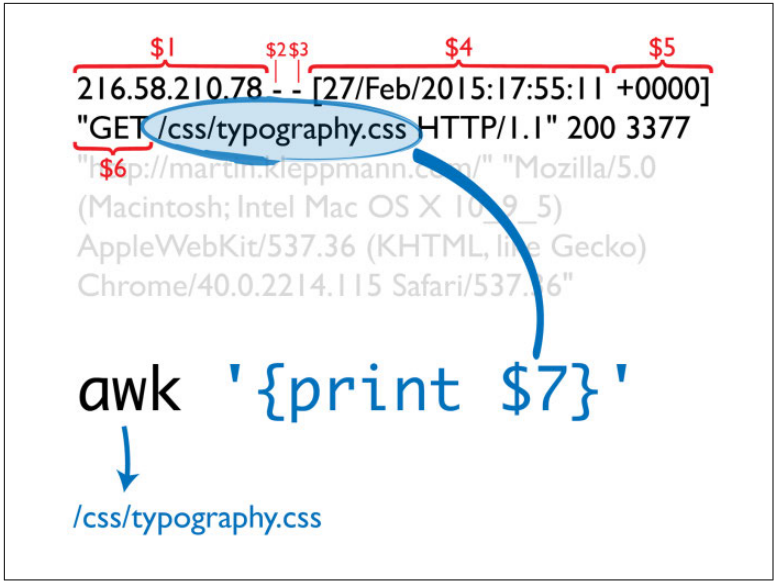


Figure 4-1. Extracting the requested URL path from a web server log by using awk.

Now that you've extracted the path, you can determine the five most popular pages on your website as follows:

```
awk '{print $7}' access.log | ❶
    sort      | ❷
    uniq -c   | ❸
    sort -rn  | ❹
    head -n 5   ❺
```

❶ Split by whitespace, 7th field is request path

❷ Make occurrences of the same URL appear consecutively in file

❸ Replace consecutive occurrences of the same URL with a count

❹ Sort by number of occurrences, descending

❺ Output top 5 URLs

The output of that series of commands looks something like this:

```
4189 /favicon.ico
3631 /2013/05/24/improving-security-of-ssh-private-keys.html
2124 /2012/12/05/schema-evolution-in-avro-protocol-buffers-
thrift.html
```

```
1369 /
 915 /css/typography.css
```

Although the chain of commands looks a bit obscure if you're unfamiliar with Unix tools, it is incredibly powerful. It will process gigabytes of log files in a matter of seconds, and you can easily modify the analysis to suit your needs. For example, if you want to count top client IP addresses instead of top pages, change the `awk` argument to `'{print $1}'`.

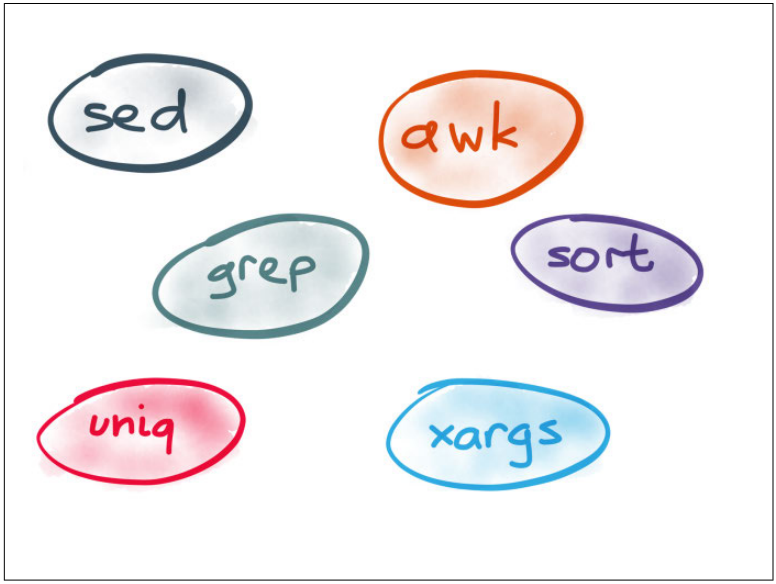


Figure 4-2. Unix: small, focused tools that combine well with one another.

Many data analyses can be done in a few minutes using some combination of `awk`, `sed`, `grep`, `sort`, `uniq`, and `xargs`, and they perform surprisingly well.[1] This is no coincidence: it is a direct result of the design philosophy of Unix (Figure 4-3).

1 Adam Drake: "Command-line tools can be 235x faster than your Hadoop cluster," aadrake.com, 25 January 2014.

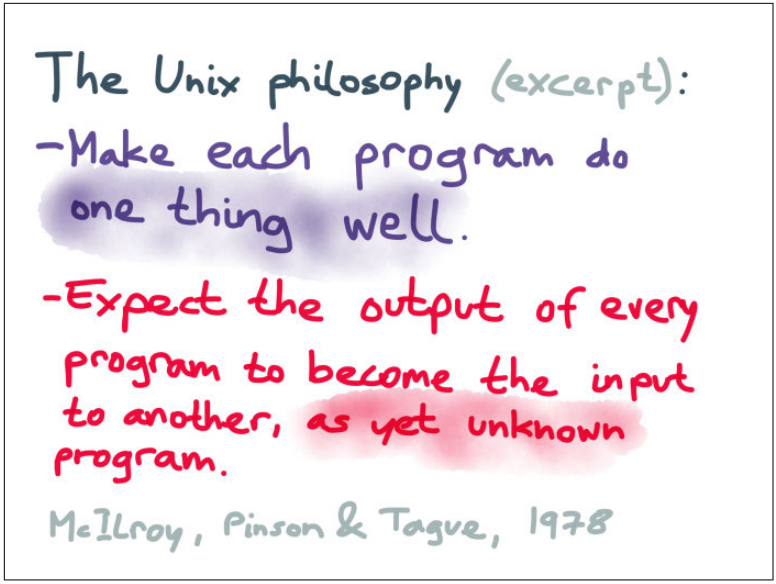


Figure 4-3. Two aspects of the Unix philosophy, as articulated by some of its designers in 1978.

The Unix philosophy is a set of principles that emerged gradually during the design and implementation of Unix systems during the late 1960s and 1970s. There are various interpretations of the Unix philosophy, but in the 1978 description by Doug McIlroy, Elliot Pinson, and Berk Tague,[2] two points particularly stand out:

- Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features."[3]
- Expect the output of every program to become the input to another, as yet unknown, program.

These principles are the foundation for chaining together programs into pipelines that can accomplish complex processing tasks. The

2 M D McIlroy, E N Pinson, and B A Tague: "UNIX Time-Sharing System: Foreword," *The Bell System Technical Journal*, volume 57, number 6, pages 1899–1904, July 1978.

3 Rob Pike and Brian W Kernighan: "Program design in the UNIX environment," AT&T Bell Laboratories Technical Journal, volume 63, number 8, pages 1595–1605, October 1984. doi:10.1002/j.1538-7305.1984.tb00055.x

key idea here is that a program does not know or care where its input is coming from, or where its output is going: it may be a file, or another program that's part of the operating system, or another program written by someone else entirely.

Pipes and Composability

The tools that come with the operating system are generic, but they are designed such that they can be *composed* together into larger programs that can perform application-specific tasks.

The benefits that the designers of Unix derived from this design approach sound quite like the ideas of the Agile and DevOps movements that appeared decades later: scripting and automation, rapid prototyping, incremental iteration, being friendly to experimentation, and breaking down large projects into manageable chunks. Plus ça change...

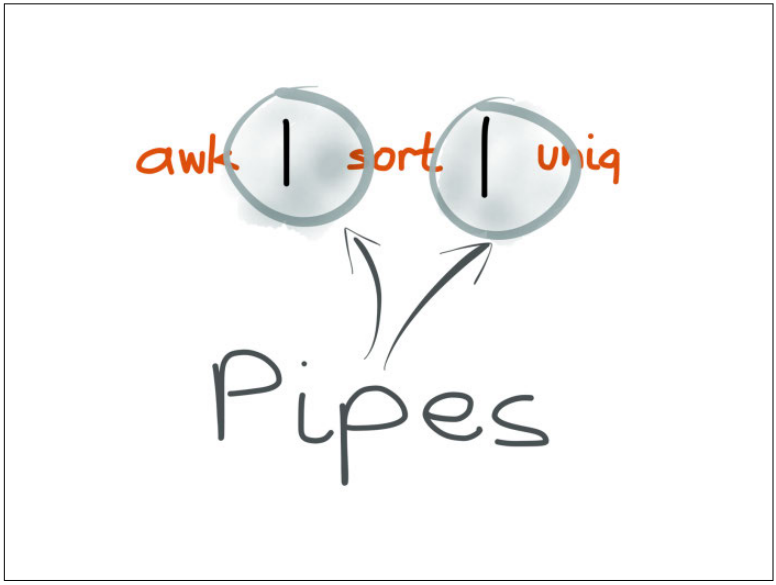


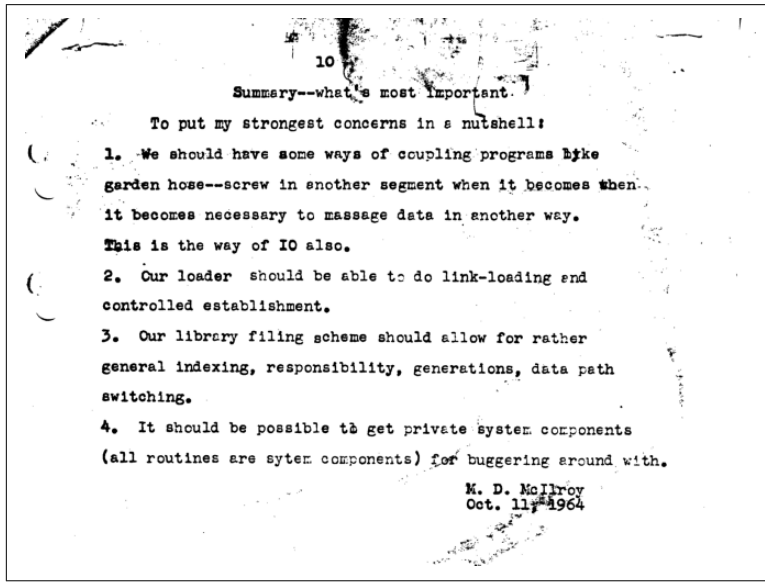
Figure 4-4. A Unix pipe joins the output of one process to the input of another.

When you join two commands by using the pipe character in your shell, the shell starts both programs at the same time, and attaches the output of the first process to the input of the second process.

This attachment mechanism uses the `pipe` syscall provided by the operating system.[4]

Note that this wiring is not done by the programs themselves; it's done by the shell—this allows the programs to be loosely coupled, and not worry about where their input is coming from or where their output is going.

The pipe had been invented in 1964 by Doug McIlroy (Figure 4-5), who described it like this in an internal Bell Labs memo:[5] "We should have some ways of coupling programs like [a] garden hose—screw in another segment when it becomes necessary to massage data in another way."



10

Summary--what's most important.

To put my strongest concerns in a nutshell:

1. We should have some ways of coupling programs like garden hose--screw in another segment when it becomes then it becomes necessary to massage data in another way. This is the way of IO also.

2. Our loader should be able to do link-loading and controlled establishment.

3. Our library filing scheme should allow for rather general indexing, responsibility, generations, data path switching.

4. It should be possible to get private system components (all routines are sytem components) for buggering around with.

M. D. McIlroy
Oct. 11, 1964

Figure 4-5. Doug McIlroy describes "coupling programs like [a] garden hose," the idea that was later known as pipes.

The Unix team also realized early that the interprocess communication mechanism (pipes) can look very similar to the I/O mechanism for reading and writing files. We now call this input redirection

4 Dennis M Ritchie and Ken Thompson: "The UNIX Time-Sharing System," *Communications of the ACM*, volume 17, number 7, July 1974. doi:10.1145/361011.361061

5 Dennis M Richie: "Advice from Doug McIlroy," cm.bell-labs.com.

(using the contents of a file as input to a process) and output redirection (writing the output of a process to a file, Figure 4-6).

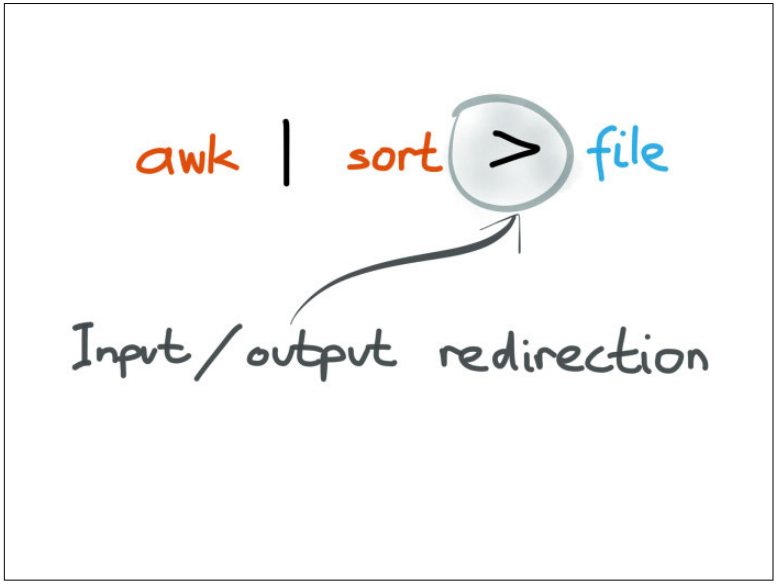


Figure 4-6. A process doesn't care whether its input and output are files on disk, or pipes to other processes.

The reason that Unix programs can be composed so flexibly is that they all conform to the same interface (Figure 4-7): most programs have one stream for input data (`stdin`) and two output streams (`stdout` for regular output data, and `stderr` for errors and diagnostic messages to the user).

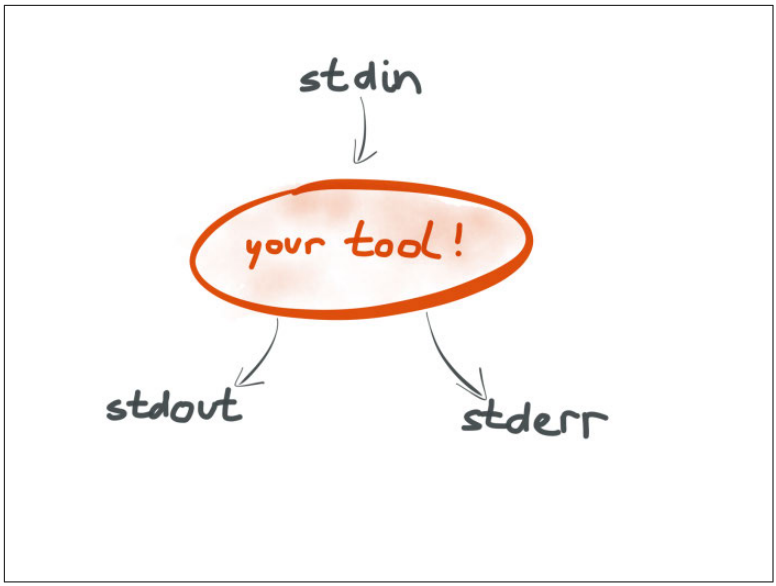


Figure 4-7. Unix tools all have the same interface of input and output streams. This standardization is crucial to enabling composability.

Programs can also do other things besides reading `stdin` and writing `stdout`, such as reading and writing files, communicating over the network, or drawing a graphical user interface. However, the `stdin`/`stdout` communication is considered to be the main means for data to flow from one Unix tool to another.

The great thing about the `stdin`/`stdout` interface is that anyone can implement it easily, in any programming language. You can develop your own tool that conforms to this interface, and it will play nicely with all the standard tools that ship as part of the operating system.

For example, when analyzing a web server log file, perhaps you want to find out how many visitors you have from each country. The log doesn't tell you the country, but it does tell you the IP address, which you can translate into a country by using an IP geolocation database. Such a database probably isn't included with your operating system by default, but you can write your own tool that takes IP addresses on `stdin`, and outputs country codes on `stdout`.

After you've written that tool, you can include it in the data processing pipeline we discussed previously, and it will work just fine (Figure 4-8). This might seem painfully obvious if you've been

working with Unix for a while, but I'd like to emphasize how remarkable this is: your own code runs on equal terms with the tools provided by the operating system.

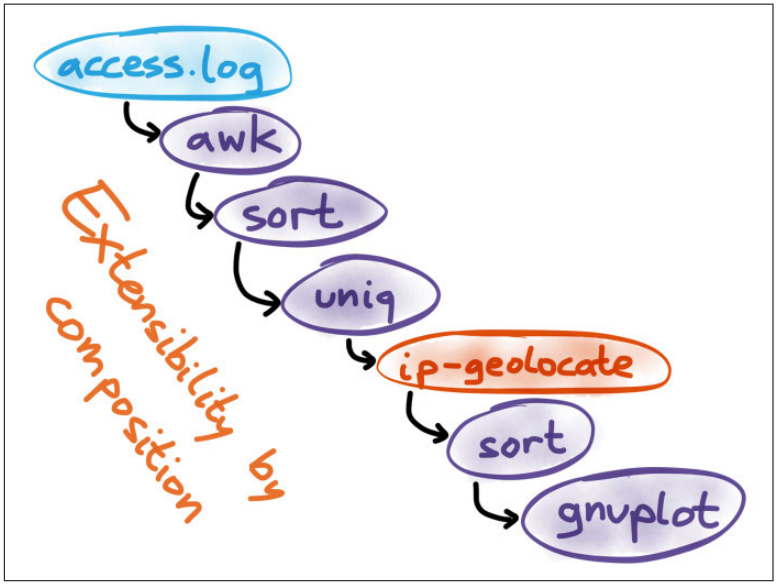


Figure 4-8. You can write your own tool that reads `stdin` and writes `stdout`, and it will work just fine with tools provided by the operating system.

Apps with graphical user interfaces or web apps cannot simply be extended and wired together like this. You can't just pipe Gmail into a separate search engine app, and post results to a wiki. Today it's an exception, not the norm, to have programs that work together as smoothly as Unix tools do.

Unix Architecture versus Database Architecture

Change of scene. Around the same time as Unix was being developed, the relational data model was proposed,[6] which in time

6 Edgar F Codd: "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, volume 13, number 6, pages 377–387, June 1970. doi: 10.1145/362384.362685

became SQL and subsequently took over the world. Many databases actually run on Unix systems. Does that mean they also follow the Unix philosophy?

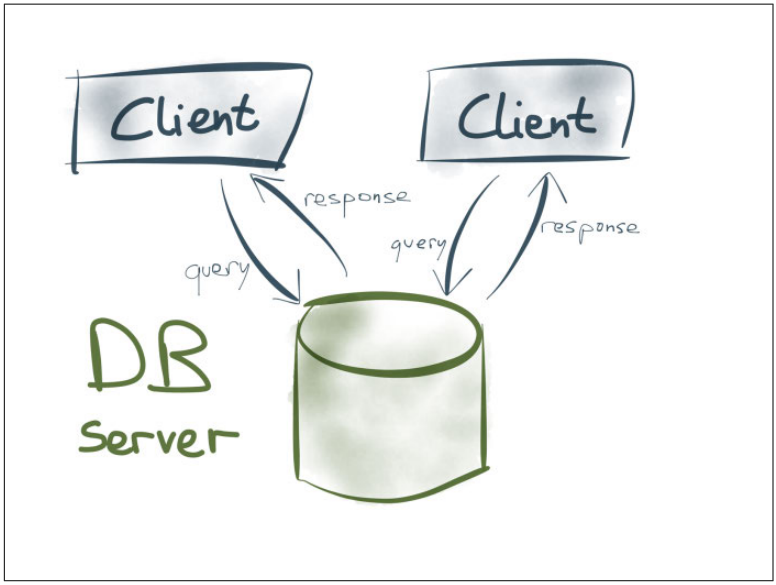


Figure 4-9. In database systems, servers and clients serve two very different roles.

The dataflow in most database systems is very different from Unix tools. Rather than using `stdin` and `stdout` as communication channels, there is a *database server*, and several *clients* (Figure 4-9). The clients send queries to read or write data on the server, the server handles the queries and sends responses to the clients. This relationship is fundamentally asymmetric: clients and servers are distinct roles.

The design philosophy of relational databases is also very different from Unix.[7] The relational model (and SQL, which was derived from it) defines clean high-level semantics that hides implementation details of the system—for example, applications don't need to care how the database represents data internally on disk. The fact

7 Eric A Brewer and Joseph M Hellerstein: "CS262a: Advanced Topics in Computer Systems," Course Notes, *University of California, Berkeley*, cs.berkeley.edu, August 2011.

that relational databases have been so wildly successful over decades indicates that this is a successful strategy.

On the other hand, Unix has very thin abstractions: it just tries to present hardware resources to programs in a consistent way, and that's it. Composition of small tools is elegant, but it's a much more low-level programming model than something like SQL.

This difference has follow-on effects on the extensibility of systems. We saw previously (Figure 4-8) that with Unix, you can add arbitrary code to a processing pipeline. In databases, clients can usually do anything they like (because they are application code), but the extensibility of database servers is much more limited (Figure 4-10).

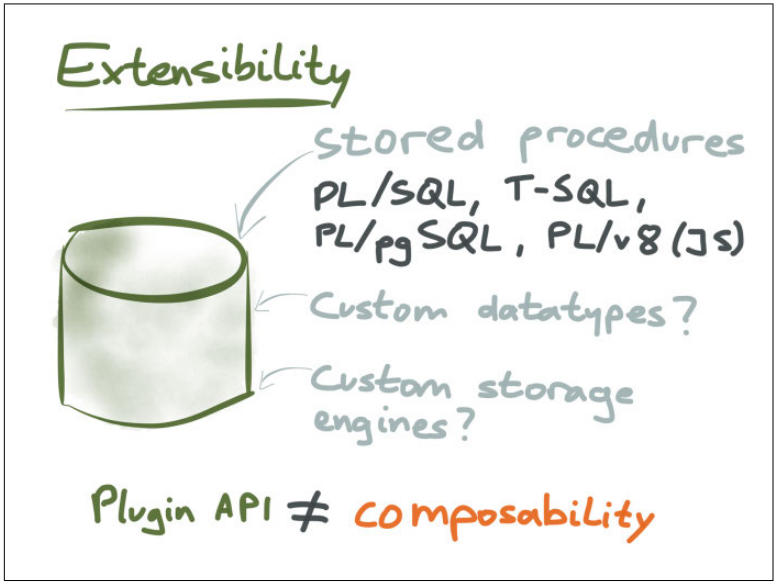


Figure 4-10. Databases have various extension points, but they generally don't have the same modularity and composability as Unix.

Many databases provide some ways of extending the database server with your own code. For example, many relational databases let you write stored procedures in their own procedural language such as PL/SQL (some let you use a general-purpose programming language such as JavaScript[8]). However, the things you can do in stored proce-

8 Craig Kerstiens: "JavaScript in your Postgres," postgres.heroku.com, 5 June 2013.

dures are limited. This prevents you from circumventing the database's transactional guarantees.

Other extension points in some databases are support for custom data types (this was one of the early design goals of Postgres[9]), foreign data wrappers and pluggable storage engines. Essentially, these are plug-in APIs: you can run your code in the database server, provided that your module adheres to a plug-in API exposed by the database server for a particular purpose.

This kind of extensibility is not the same as the arbitrary composability we saw with Unix tools. The plug-in API is provided for a particular purpose, and can't safely be used for other purposes. If you want to extend the database in some way that is not foreseen by a plug-in API or stored procedure, you'll probably need to change the code of the database server, which is a big undertaking.

Stored procedures also have a reputation of being hard to maintain and operate. Compared with normal application code, it is much more difficult to deal with monitoring, versioning, deployments, debugging, measuring performance impact, multitenant resource isolation, and so on.

There's no fundamental reason why a database couldn't be more like an operating system, allowing many users to run arbitrary code and access data in a shared environment, with good operational tooling and with appropriate security and access control. However, databases have not developed in this direction in practice over the past decades. Database servers are seen as mostly in the business of storing and retrieving your data, and letting you run arbitrary code is not their top priority.

But, why would you want arbitrary extensibility in a database at all? Isn't that just a recipe for shooting yourself in the foot? Well, as we saw in Chapter 2, many applications need to do a great variety of things with their data, and a single database with a SQL interface is simply not sufficient.

9 Michael Stonebraker and Lawrence A Rowe: "The design of Postgres," Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Technical Report UCB/ERL M85/95, 1985.

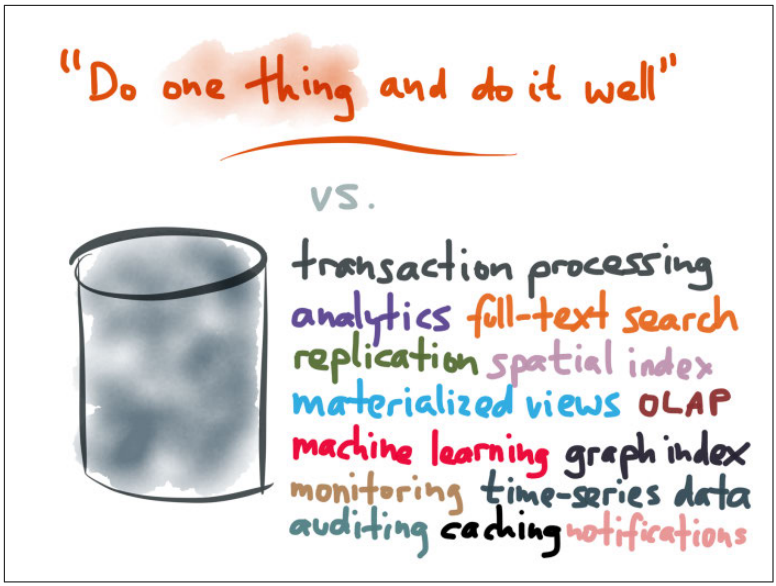


Figure 4-11. A general-purpose database with many features is convenient but philosophically very different from Unix.

A general-purpose database might try to provide many features in one product (Figure 4-11, the "one size fits all" approach), but in all likelihood it will not perform as well as a tool that is specialized for one particular purpose.[10] In practice, you can often get the best results by combining various different data storage and indexing systems: for example, you might take the same data and store it in a relational database for random access, in Elasticsearch for full-text search, in a columnar format in Hadoop for analytics, and cached in a denormalized form in memcached (Figure 4-12).

10 Michael Stonebraker and Uğur Çetintemel: "'One Size Fits All': An Idea Whose Time Has Come and Gone," at *21st International Conference on Data Engineering* (ICDE), April 2005.

Figure 4-12. Rather than trying to satisfy all use cases with one tool, it is better to support a diverse ecosystem of tools with different areas of speciality.

Moreover, there are some things that require custom code and can't just be done with an off-the-shelf database. For example:

- A machine-learning system (feature extraction, recommendation engines, trained classifiers, etc.) usually needs to be customized and adapted to a particular application;
- A notification system needs to be integrated with various external providers (email delivery, SMS, push notifications to mobile devices, webhooks, etc.);
- A cache might need to contain data that has been filtered, aggregated, or rendered according to application-specific business logic (which can become quite complicated).

Thus, although SQL and query planners are a great accomplishment, they can't satisfy all use cases. Integration with other storage systems and extensibility with arbitrary code is also necessary. Unix shows us that simple, composable tools give us an elegant way of making systems extensible and flexible—but databases are not like Unix. They are tremendously complicated, monolithic beasts that

try to implement all the features you might need in a single program.

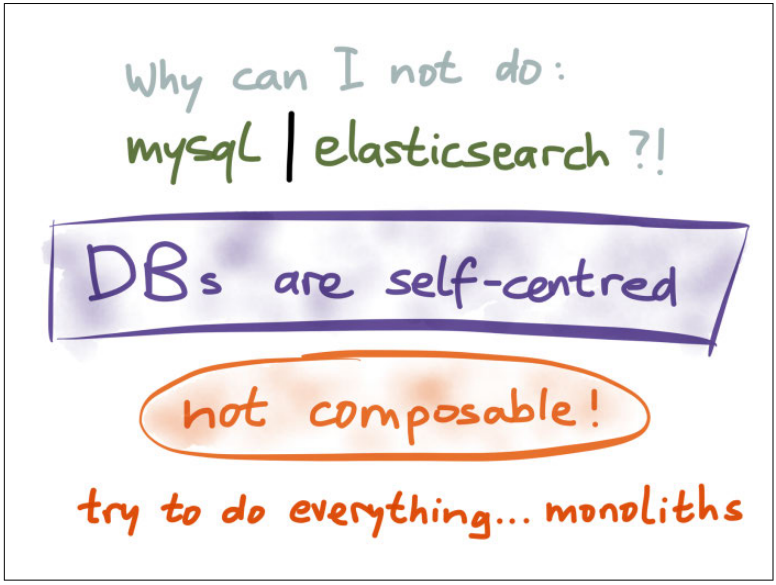


Figure 4-13. Sadly, most databases are not designed with composability in mind.

By default, you can't just pipe one database into another, even if they have compatible data models (Figure 4-13). You can use bulk loading and bulk dumping (backup/snapshot), but those are one-off operations, not designed to be used continuously. Change data capture (Chapter 3) allows us to build these pipe-like integrations, but CDC is somewhat of a fringe feature in many databases. I don't know of any mainstream database that uses change streams as its primary input/output mechanism.

Nor can you insert your own code into the database's internal processing pipelines, unless the server has specifically provided an extension point for you, such as triggers.

I feel the design of databases is very self-centered. A database seems to assume that it's the center of your universe: the only place where you might want to store and query your data, the source of truth, and the destination for all queries. They don't have the same kind of composability and extensibility that we find on Unix. As long as you only need the features provided by the database, this integrated/

monolithic model works very well, but it breaks down when you need more than what a single database can provide.

Composability Requires a Uniform Interface

We said that Unix tools are composable because they all implement the same interface of `stdin`, `stdout`, and `stderr`, and each of these is a *file descriptor*; that is, a stream of bytes that you can read or write like a file (Figure 4-14). This interface is simple enough that anyone can easily implement it, but it is also powerful enough that you can use it for anything.

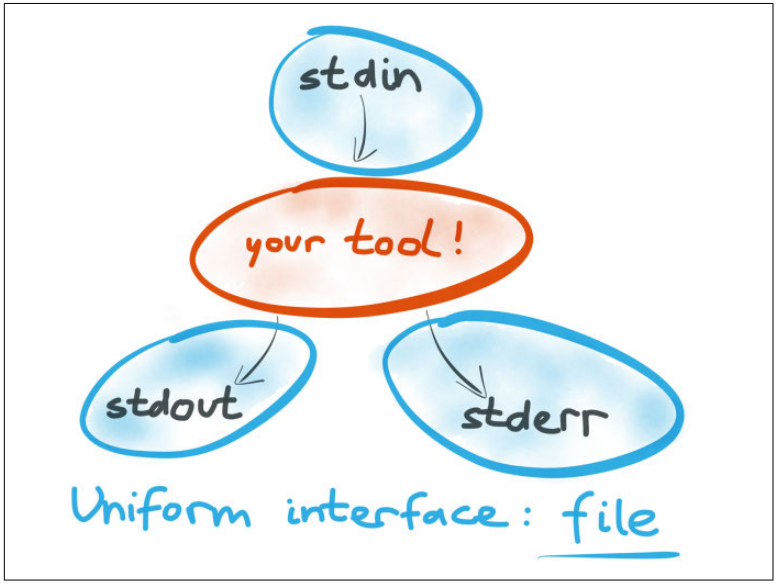


Figure 4-14. On Unix, `stdin`, `stdout`, and `stderr` are all the same kind of thing: a file descriptor (i.e,. a stream of bytes). This makes them compatible.

Because all Unix tools implement the same interface, we call it a *uniform interface*. That's why you can pipe the output of `gunzip` to `wc` without a second thought, even though those two tools appear to have nothing in common. It's like lego bricks, which all implement the same pattern of knobbly bits and grooves, allowing you to stack any lego brick on any other, regardless of their shape, size, or color.

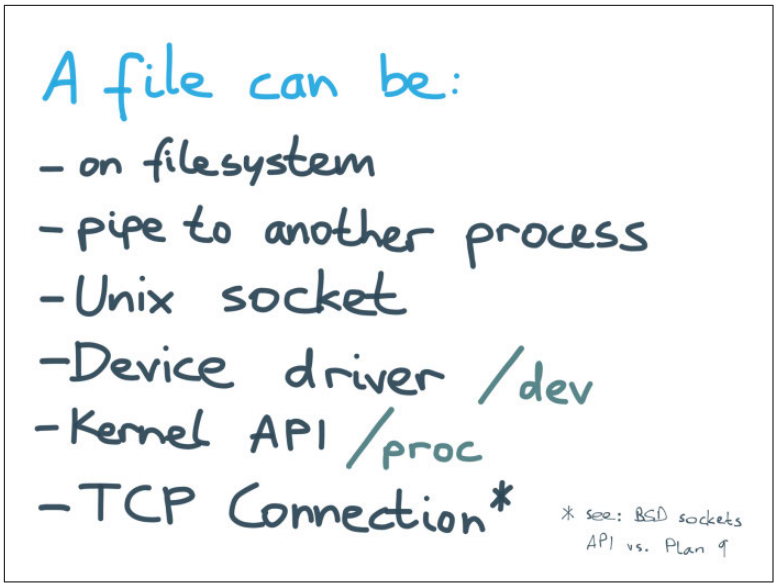


Figure 4-15. The file abstraction can be used to represent many different hardware and software concepts.

The uniform interface of file descriptors in Unix doesn't just apply to the input and output of processes; rather, it's a very broadly applied pattern (Figure 4-15). If you open a file on the filesystem, you get a file descriptor. Pipes and Unix sockets provide file descriptors that are a communication channel to another process on the same machine. On Linux, the virtual files in `/dev` are the interfaces of device drivers, so you might be talking to a USB port or even a GPU. The virtual files in `/proc` are an API for the kernel, but because they're exposed as files, you can access them with the same tools as regular files.

Even a TCP connection to a process on another machine is a file descriptor, although the BSD sockets API (which is most commonly used to establish TCP connections) is arguably not as "Unixy" as it could be. Plan 9 shows that even the network could have been cleanly integrated into the same uniform interface.[11]

11 Eric S Raymond: "Plan 9: The Way the Future Was," in *The Art of Unix Programming*, Addison-Wesley Professional, 2003. ISBN: 0131429019, available online at catb.org.

To a first approximation, everything on Unix is a file. This uniformity means the logic of Unix tools can be separated from the wiring, making them more composable. `sed` doesn't need to care whether it's talking to a pipe to another process, or a socket, or a device driver, or a real file on the filesystem — they all look the same.

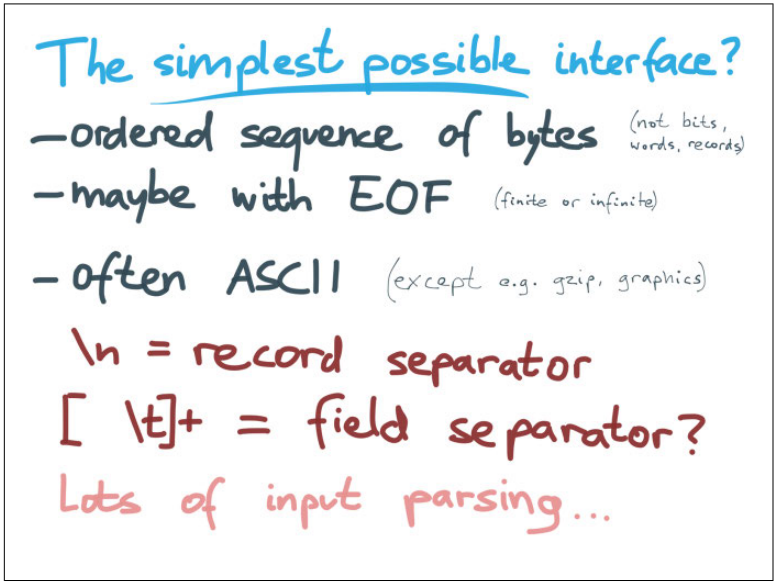


Figure 4-16. A file is just a stream of bytes, and most programs need to parse that stream before they can do anything useful with it.

A file is a *stream of bytes*, perhaps with an end-of-file (EOF) marker at some point, indicating that the stream has ended (a stream can be of arbitrary length, and a process might not know in advance how long its input is going to be).

A few tools (e.g., `gzip`) operate purely on byte streams and don't care about the structure of the data. But most tools need to parse their input in order to do anything useful with it (Figure 4-16). For this, most Unix tools use ASCII, with each record on one line, and fields separated by tabs or spaces, or maybe commas.

Files are totally obvious to us today, which shows that a byte stream turned out to be a good uniform interface. However, the implementors of Unix could have decided to do it very differently. For example, it could have been a function callback interface, using a schema to pass strongly typed records from process to process. Or, it could

have been shared memory (like System V IPC or mmap, which came along later). Or, it could have been a *bit* stream rather than a byte stream.

In a sense, a byte stream is a lowest common denominator—the simplest possible interface. Everything can be expressed in terms of a stream of bytes, and it's fairly agnostic to the transport medium (pipe from another process, file on disk, TCP connection, tape, etc). But this is also a disadvantage, as we shall discuss in the next section.

Bringing the Unix Philosophy to the Twenty-First Century

We've seen that both Unix and databases have developed good design principles for software development, but they have taken very different routes. I would love to see a future in which we can learn from both paths of development, and combine the best ideas and implementations from each (Figure 4-17).

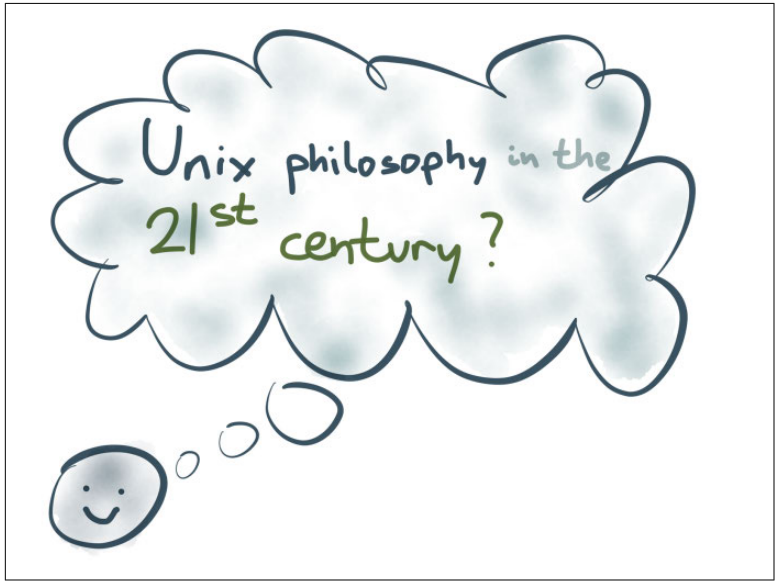


Figure 4-17. Can we improve contemporary data systems by borrowing the best ideas from Unix but avoiding its mistakes?

How can we make twenty-first-century data systems better by learning from the Unix philosophy? In the rest of this chapter, I'd like to explore what it might look like if we bring the Unix philosophy to the world of databases.

First, let's acknowledge that Unix is not perfect (Figure 4-18).

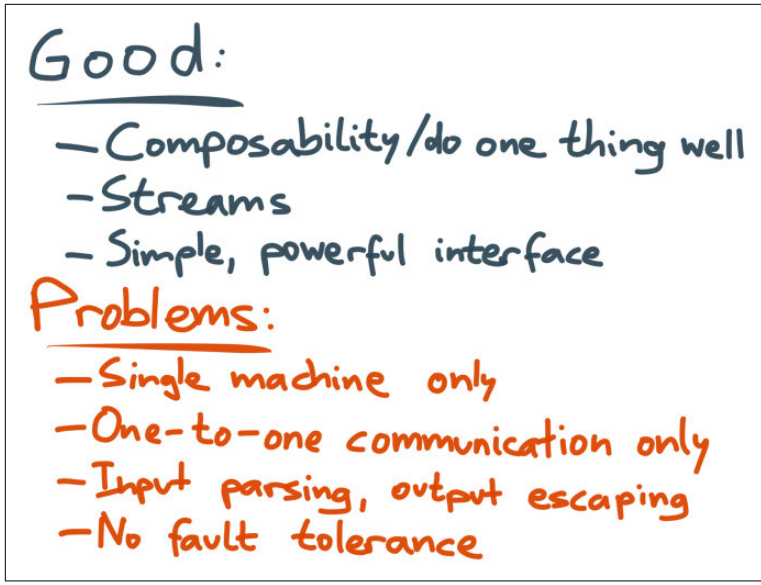


Figure 4-18. Pros and cons of Unix pipes.

Although I think the simple, uniform interface of byte streams was very successful at enabling an ecosystem of flexible, composable, powerful tools, Unix has some limitations:

- It's designed for use on a single machine. As our applications need to cope with ever-increasing data and traffic, and have higher uptime requirements, moving to distributed systems is becoming increasingly inevitable.[12] Although a TCP connection can be made to look somewhat like a file, I don't think that's the right answer: it only works if both sides of the connection are

12 Mark Cavage: "There's Just No Getting around It: You're Building a Distributed System," *ACM Queue*, volume 11, number 4, April 2013. doi:10.1145/2466486.2482856

up, and it has somewhat messy edge case semantics.[13] TCP is good, but by itself it's too low-level to serve as a distributed pipe implementation.

- A Unix pipe is designed to have a single sender process and a single recipient. You can't use pipes to send output to several processes, or to collect input from several processes. (You can branch a pipeline by using `tee`, but a pipe itself is always one-to-one.)
- ASCII text (or rather, UTF-8) is great for making data easily explorable, but it quickly becomes messy. Every process needs to be set up with its own input parsing: first breaking the byte stream into records (usually separated by newline, though some advocate `0x1e`, the ASCII record separator).[14] Then, a record needs to be broken up into fields, like the `$7` in the `awk` example (Figure 4-1). Separator characters that appear in the data need to be escaped somehow. Even a fairly simple tool like `xargs` has about half a dozen command-line options to specify how its input should be parsed. Text-based interfaces work tolerably well, but in retrospect, I am pretty sure that a richer data model with explicit schemas would have worked better.[15]
- Unix processes are generally assumed to be fairly short-running. For example, if a process in the middle of a pipeline crashes, there is no way for it to resume processing from its input pipe—the entire pipeline fails and must be re-run from scratch. That's no problem if the commands run only for a few seconds, but if an application is expected to run continuously for years, you need better fault tolerance.

I believe we already have an approach that overcomes these downsides while retaining the Unix philosophy's benefits: Kafka and stream processing.

13 Bert Hubert: "The ultimate SO_LINGER page, or: why is my tcp not reliable," blog.netherlabs.nl, 18 January 2009.

14 Ronald Duncan: "Text File formats – ASCII Delimited Text – Not CSV or TAB delimited text," ronalddunc an.wordpress.com, 31 October 2009.

15 Gwen Shapira: "The problem of managing schemas," radar.oreilly.com, 4 November 2014.

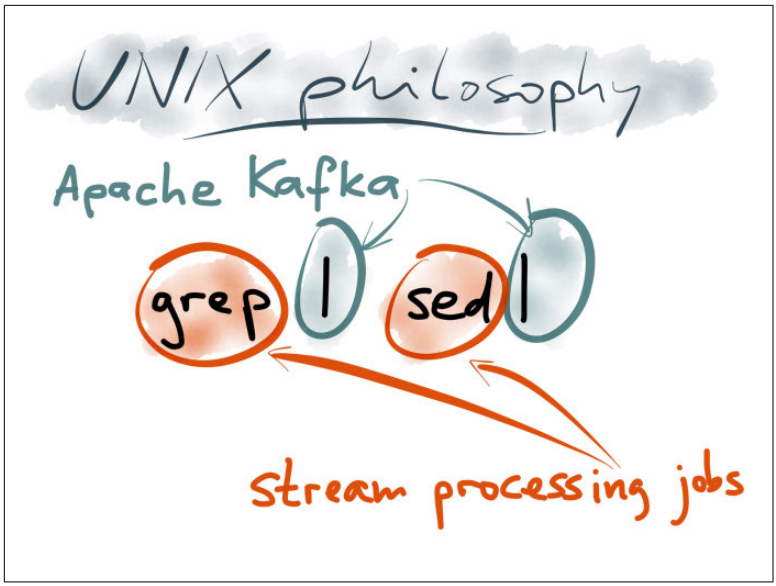


Figure 4-19. The data flow between stream processing jobs, using Kafka for message transport, resembles a pipeline of Unix tools.

When you look at it through the Unix lens, Kafka looks quite like the pipe that connects the output of one process to the input of another (Figure 4-19). And a stream processing framework like Samza looks quite like a standard library that helps you read `stdin` and write `stdout` (along with a few helpful additions, such as a deployment mechanism, state management,[16] metrics, and monitoring).

The Kafka Streams library and Samza apply this composable design more consistently than other stream processing frameworks. In Storm, Spark Streaming, and Flink, you create a *topology* (processing graph) of stream operators (bolts), which are connected through the framework's own mechanism for message transport. In Kafka Streams and Samza, there is no separate message transport protocol: the communication from one operator to the next always goes via Kafka, just like Unix tools always go via `stdout` and `stdin`. The core

16 Jay Kreps: "Why local state is a fundamental primitive in stream processing," radar.oreilly.com, 31 July 2014.

advantage is that they can leverage the guarantees provided by Kafka for reliable, large-scale, messaging.

Kafka Streams offers both a low-level processor API as well as a DSL for defining stream processing operations. Both Kafka Streams and Samza have a fairly low-level programming model that is very flexible: each operator can be deployed independently (perhaps by different teams), the processing graph can be gradually extended as new applications emerge, and you can add new consumers (e.g., for monitoring purposes) at any point in the processing graph.

However, as mentioned previously, Unix pipes have some problems. They are good for building quick, hacky data exploration pipelines, but they are not a good model for large applications that need to be maintained for many years. If we are going to build new systems using the Unix philosophy, we will need to address those problems.

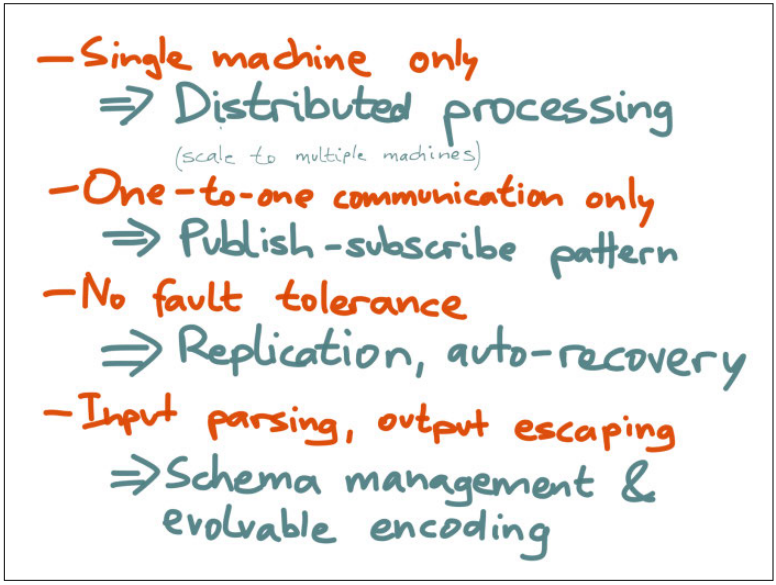


Figure 4-20. How Kafka addresses the problems with Unix pipes.

Kafka addresses the downsides of Unix pipes as follows (Figure 4-20):

- The single-machine limitation is lifted: Kafka itself is distributed by default, and any stream processors that use it can also be distributed across multiple machines.

- A Unix pipe connects the output of exactly one process with the input of exactly one process, whereas a stream in Kafka can have many producers and many consumers. Having many inputs is important for services that are distributed across multiple machines, and many outputs makes Kafka more like a broadcast channel. This is very useful because it allows the same data stream to be consumed independently for several different purposes (including monitoring and audit purposes, which are often outside of the application itself). Kafka consumers can come and go without affecting other consumers.
- Kafka also provides good fault tolerance: data is replicated across multiple Kafka nodes, so if one node fails, another node can automatically take over. If a stream processor node fails and is restarted, it can resume processing at its last checkpoint, so it does not miss any input.
- Rather than a stream of bytes, Kafka provides a stream of messages, which saves the first step of input parsing (breaking the stream of bytes into a sequence of records). Each message is just an array of bytes, so you can use your favorite serialization format for individual messages: JSON, Avro, Thrift, or Protocol Buffers are all reasonable choices.[17] It's well worth standardizing on one encoding,[18] and Confluent provides particularly good schema management support for Avro.[19] This allows applications to work with objects that have meaningful field names, and not have to worry about input parsing or output escaping. It also provides good support for schema evolution without breaking compatibility.

17 Martin Kleppmann: "Schema evolution in Avro, Protocol Buffers and Thrift," martin.kleppmann.com, 5 December 2012.

18 Jay Kreps: "Putting Apache Kafka to use: A practical guide to building a stream data platform (Part 2)," confluent.io, 24 February 2015.

19 "Schema Registry," Confluent Platform Documentation, docs.confluent.io.

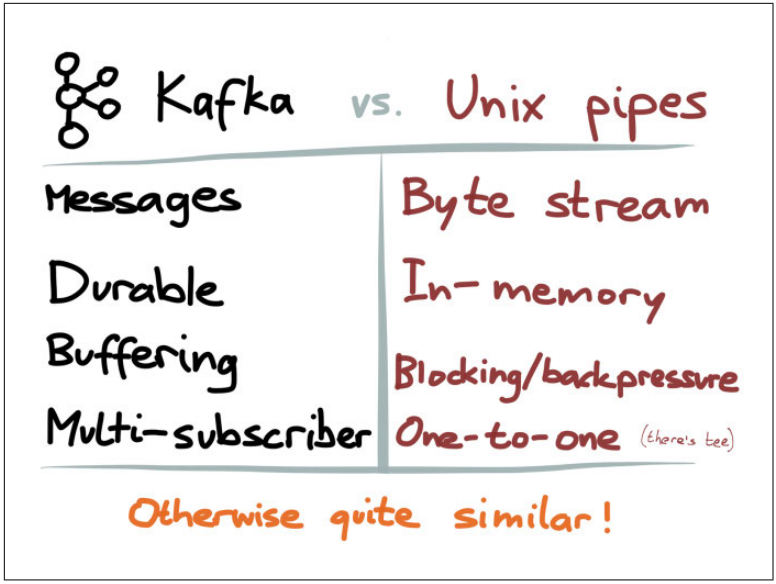


Figure 4-21. Side-by-side comparison of Apache Kafka and Unix pipes.

There are a few more things that Kafka does differently from Unix pipes, which are worth calling out briefly (Figure 4-21):

- As mentioned, Unix pipes provide a byte stream, whereas Kafka provides a stream of messages. This is especially important if several processes concurrently write to the same stream: in a byte stream, the bytes from different writers can be interleaved, leading to an unparseable mess. Because messages are coarser-grained and self-contained, they can be safely interleaved, making it safe for multiple processes to concurrently write to the same stream.
- Unix pipes are just a small in-memory buffer, whereas Kafka durably writes all messages to disk. In this regard, Kafka is less like a pipe and more like one process writing to a temporary file, while several other processes continuously read that file using `tail -f` (each consumer tails the file independently). Kafka's approach provides better fault tolerance because it allows a consumer to fail and restart without skipping messages. Kafka automatically splits those "temporary" files into segments and garbage-collects old segments on a configurable schedule.

- In Unix, if the consuming process of a pipe is slow to read the data, the buffer fills up and the sending process is blocked from writing to the pipe. This is a kind of backpressure. In Kafka, the producer and consumer are more decoupled: a slow consumer has its input buffered, so it doesn't slow down the producer or other consumers. As long as the buffer fits within Kafka's available disk space, the slow consumer can catch up later. This makes the system less sensitive to individual slow components and more robust overall.
- A data stream in Kafka is called a *topic*, and you can refer to it by name (which makes it more like a Unix named pipe[20]). A pipeline of Unix programs is usually started all at once, so the pipes normally don't need explicit names. On the other hand, a long-running application usually has bits added, removed, or replaced gradually over time, so you need names in order to tell the system what you want to connect to. Naming also helps with discovery and management.

Despite those differences, I still think it makes sense to think of Kafka as Unix pipes for distributed data. For example, one thing they have in common is that Kafka keeps messages in a fixed order (like Unix pipes, which keep the byte stream in a fixed order). As discussed in Chapter 2, this is a very useful property for event log data: the order in which things happened is often meaningful and needs to be preserved. Other types of message brokers, like AMQP and JMS, do not have this ordering property.

20 Vince Buffalo: "Using Named Pipes and Process Substitution," vincebuffalo.org, 8 August 2013.

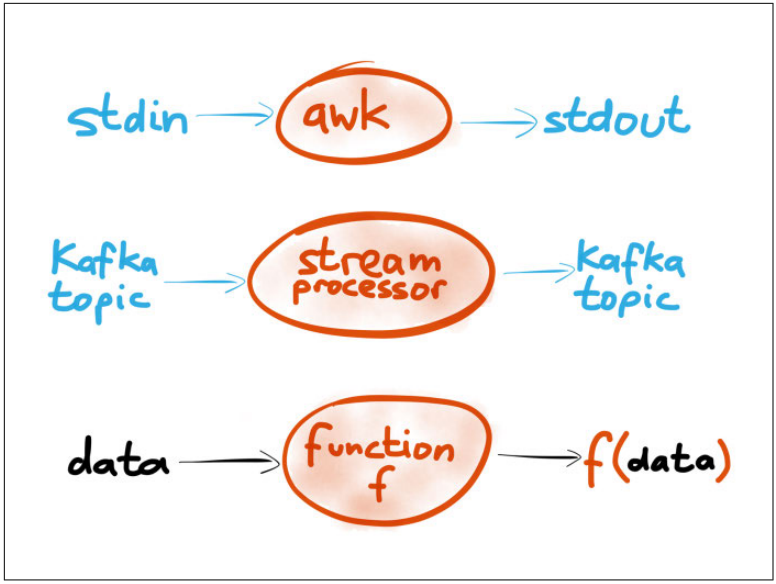


Figure 4-22. Unix tools, stream processors and functional programming share a common trait: inputs are immutable, processing has no global side-effects, and the output is explicit.

So we've got Unix tools and stream processors that look quite similar. Both read some input stream, modify or transform it in some way, and produce an output stream that is somehow derived from the input (Figure 4-22).

Importantly, the processing does not modify the input itself: it remains immutable. If you run `sed` or `awk` on some file, the input file remains unmodified (unless you explicitly choose to overwrite it), and the output is sent somewhere else. Also, most Unix tools are *deterministic*; that is, if you give them the same input, they always produce the same output. This means that you can re-run the same command as many times as you want and gradually iterate your way toward a working program. It's great for experimentation, because you can always go back to your original data if you mess up the processing.

This deterministic and side-effect-free processing looks a lot like functional programming. That doesn't mean you must use a functional programming language like Haskell (although you're welcome to do so if you want), but you still get many of the benefits of functional code.

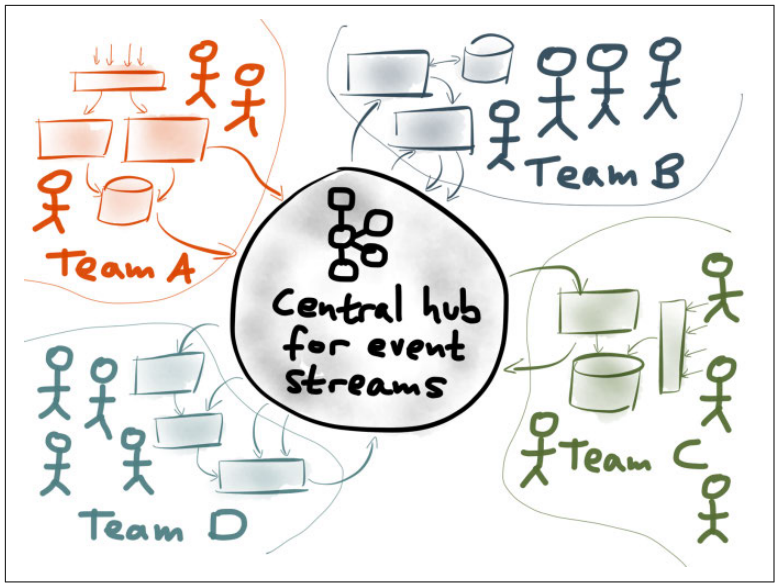


Figure 4-23. Loosely coupled stream processors are good for organizational scalability: Kafka topics can transport data from one team to another, and each team can maintain its own stream processing jobs.

The Unix-like design principles of Kafka enable building composable systems at a large scale (Figure 4-23). In a large organization, different teams can each publish their data to Kafka. Each team can independently develop and maintain stream processing jobs that consume streams and produce new streams. Because a stream can have any number of independent consumers, no coordination is required to set up a new consumer.

We've been calling this idea a stream data platform.[21] In this kind of architecture, the data streams in Kafka act as the communication channel between different teams' systems. Each team focuses on making their particular part of the system do one thing well. Whereas Unix tools can be composed to accomplish a data processing task, distributed streaming systems can be composed to comprise the entire operation of a large organization.

21 Jay Kreps: "Putting Apache Kafka to use: A practical guide to building a stream data platform (Part 1)," confluent.io, 24 February 2015.

A Unix-like approach manages the complexity of a large system by encouraging loose coupling: thanks to the uniform interface of streams, different components can be developed and deployed independently. Thanks to the fault tolerance and buffering of the pipe (Kafka), when a problem occurs in one part of the system, it remains localized. And schema management[22] allows changes to data structures to be made safely so that each team can move fast without breaking things for other teams.

To wrap up this chapter, let's consider a real-life example of how this works at LinkedIn (Figure 4-24).

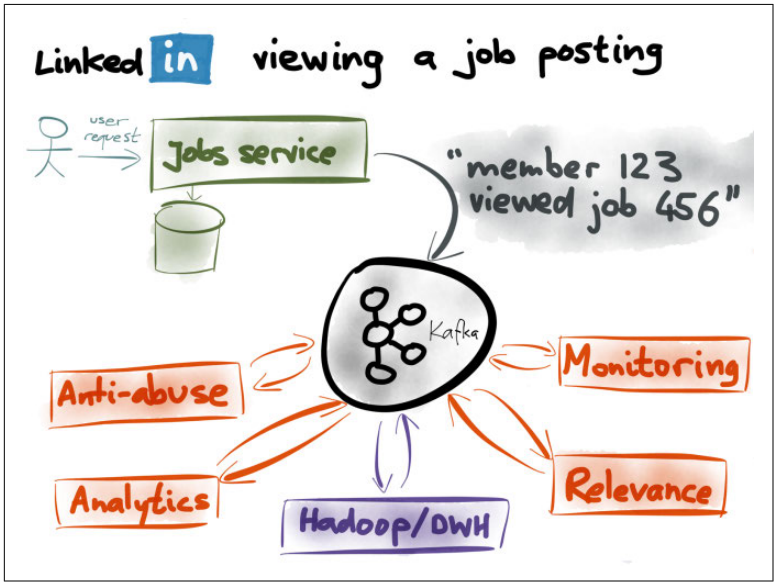


Figure 4-24. What happens when someone views a job posting on LinkedIn?

As you may know, companies can post their job openings on LinkedIn, and jobseekers can browse and apply for those jobs. What happens if a LinkedIn member (user) views one of those job postings?

The service that handles job views publishes an event to Kafka, saying something like "member 123 viewed job 456 at time 789." Now

22 Jay Kreps: "Putting Apache Kafka to use: A practical guide to building a stream data platform (Part 2)," confluent.io, 24 February 2015.

that this information is in Kafka, it can be used for many good purposes:[23]

Monitoring systems
: Companies pay LinkedIn to post their job openings, so it's important that the site is working correctly. If the rate of job views drops unexpectedly, alarms should go off because it indicates a problem that needs to be investigated.

Relevance and recommendations
: It's annoying for users to see the same thing over and over again, so it's good to track how many times the users have seen a job posting and feed that into the scoring process. Keeping track of who viewed what also allows for collaborative filtering recommendations (people who viewed X also viewed Y).

Preventing abuse
: LinkedIn doesn't want people to be able to scrape all the jobs, submit spam, or otherwise violate the terms of service. Knowing who is doing what is the first step toward detecting and blocking abuse.

Job poster analytics
: The companies who post their job openings want to see stats (in the style of Google Analytics) about who is viewing their postings,[24] so that they can test which wording attracts the best candidates.

Import into Hadoop and Data Warehouse
: For LinkedIn's internal business analytics, for senior management's dashboards, for crunching numbers that are reported to Wall Street, for evaluating A/B tests, and so on.

All of those systems are complex in their own right and are maintained by different teams. Kafka provides a fault-tolerant, scalable implementation of a pipe. A stream data platform based on Kafka allows all of these various systems to be developed independently, and to be connected and composed in a robust way.

---

23 Ken Goodhope, Joel Koshy, Jay Kreps, et al.: "Building LinkedIn's Real-time Activity Data Pipeline," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, volume 35, number 2, pages 33–45, June 2012.

24 Praveen Neppalli Naga: "Real-time Analytics at Massive Scale with Pinot," engineering.linkedin.com, 29 September 2014.

CHAPTER 5
Turning the Database Inside Out

In the previous four chapters, we have covered a lot of ground:

- In Chapter 1, we discussed the idea of event sourcing; that is, representing the changes to a database as a log of immutable events. We explored the distinction between raw events (which are optimized for writing) and aggregated summaries of events (which are optimized for reading).
- In Chapter 2, we saw how a log (an ordered sequence of events) can help integrate different storage systems by ensuring that data is written to all stores in the same order.
- In Chapter 3, we discussed change data capture (CDC), a technique for taking the writes to a traditional database and turning them into a log. We saw how log compaction makes it possible for us to build new indexes onto existing data from scratch without affecting other parts of the system.
- In Chapter 4, we explored the Unix philosophy for building composable systems and compared it to the traditional database philosophy. We saw how a Kafka-based stream data platform can scale to encompass the data flows in a large organization.

In this final chapter, we will pull all of those ideas together and use them to speculate about the future of databases and data-intensive applications. By extrapolating some current trends (such as the growing variety of SQL and NoSQL datastores being used, the growing mainstream use of functional programming, the increasing interactivity of user interfaces, and the proliferation of mobile devi-

ces) we can illuminate some of the path ahead: how will we be developing applications in a few years' time?

To figure out an answer, we will examine some aspects of traditional database-backed applications (replication, secondary indexes, caching, and materialized views) and compare them to the event log approach discussed in the last few chapters. We will see that many of the internal architectural patterns of databases are being repeated at a larger scale on the infrastructure level.

What is happening here is very interesting: software engineers are taking the components of databases that have been traditionally fused together into a monolithic program, unbundling them into separate components, independently making each of those components highly scalable and robust, and then putting them back together again as a large-scale system. The final result looks somewhat like a database, except that it is flexibly composed around the structure of your application and operates at much larger scale. We are taking the database architecture we know and turning it inside out.

How Databases Are Used

To gain clarity, let's take a few steps back and talk about databases. What I mean is not any particular brand of database—I don't mind whether you're using relational, or NoSQL, or something else. I'm really talking about the general concept of a database, as we use it when building applications.

Take, for example, the stereotypical web application architecture shown in Figure 5-1.

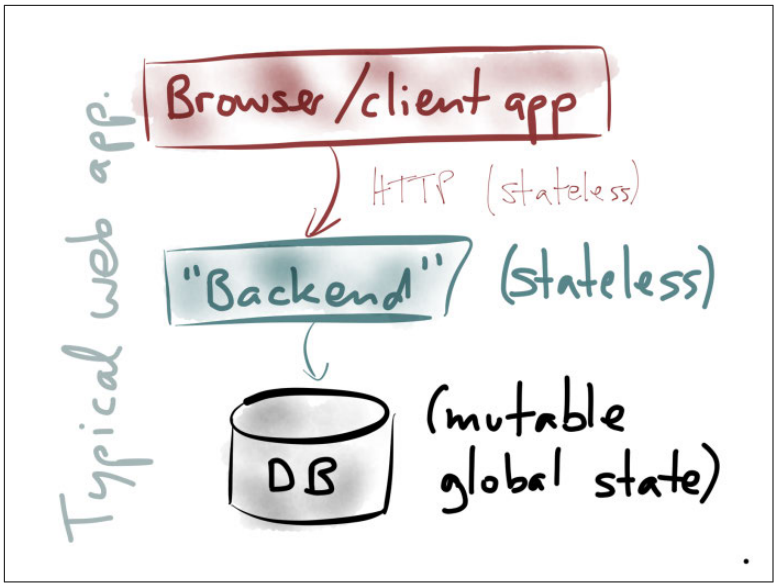


Figure 5-1. Simplest-case web application architecture.

You have a client, which may be a web browser or a mobile app, and that client talks to some kind of server-side system (a "backend"). The backend typically implements some kind of business logic, performs access control, accepts input, and produces output. When the backend needs to remember something for the future, it stores that data in a database, and when it needs to look something up, it queries a database. That's all very familiar stuff.

The way we typically build these sorts of applications is that we make the backend layer *stateless*: it processes every request independently, and doesn't remember anything from one request to the next. That has a lot of advantages: you can scale-out the backend by just running more processes in parallel, and you can route any request to any backend instance (they are all equally well qualified to handle the request), so it's easy to spread the load across multiple machines. Any state that is required to handle a request will be looked-up from the database on each request. That also works nicely with HTTP because HTTP is a stateless protocol.

However, the state must go *somewhere*, and so we put it in the database. We are now using the database as a kind of gigantic, global, shared, mutable state. It's like a persistent global variable that's shared between all your application servers.

This approach for building database-backed applications has worked for decades, so it can't be all that bad. However, from time to time it's worth looking beyond the familiar and explore potentially better ways of building software. For example, people who use functional programming languages say that the lack of mutable global variables is helpful for building better software, reducing bugs, making code easier to reason about, and so on. Perhaps something similar is true in database-backed applications?

The event sourcing approach we discussed in Chapter 1 is a way of moving from the imperative world of mutable state to the functional world of immutable values. In Chapter 4 we also noticed that pipelines of Unix tools have a functional flavor. However, so far we have not been very clear about how to actually build systems that use these ideas.

To try to figure out a way forward, I'd like to review four different examples of things that databases currently do, and things that we do with databases. These four examples will help us structure the ideas around event streams and pave the way to applying them in practice.

1. Replication

We previously discussed replication in Chapter 2, and observed that leader-based replication uses a replication log to send data changes to followers (Figure 2-18). We came across the idea again in Chapter 3: change data capture is similar to replication, except that the follower is not another instance of the same database software, but a different storage technology.

What does such a replication log actually look like? For example, take the shopping cart example of Figure 1-10, in which customer 123 changes their cart to contain quantity 3 of product 999. The update is executed on the leader, and replicated to followers. There are several different ways by which the followers might apply this write. One option is to send the same update query to the follower, and it executes the same statement on its own copy of the database. Another option is to ship the write-ahead log from the leader to the follower.

A third option for replication, which I'll focus on here, is called a *logical log*, which you can see illustrated in Figure 5-2. In this case,

the leader writes out the effect that the query had—that is, which rows were inserted, updated, or deleted—like a kind of diff.

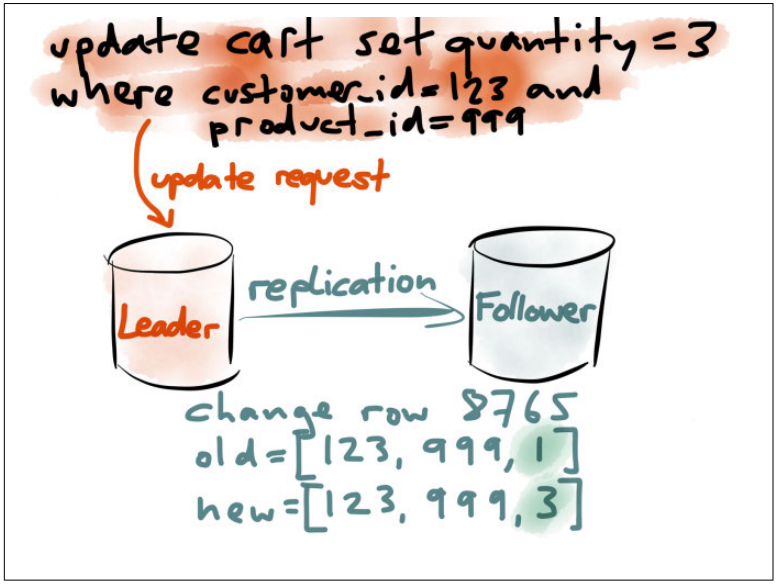


Figure 5-2. A logical change event in a replication log indicates which row changed and what its new value needs to be.

For an update, like in this example, the logical log identifies the row that was changed (using a primary key or some kind of internal tuple identifier), gives the new value of that row, and perhaps also the old value.

This might seem like nothing special, but notice that something interesting has happened (Figure 5-3).

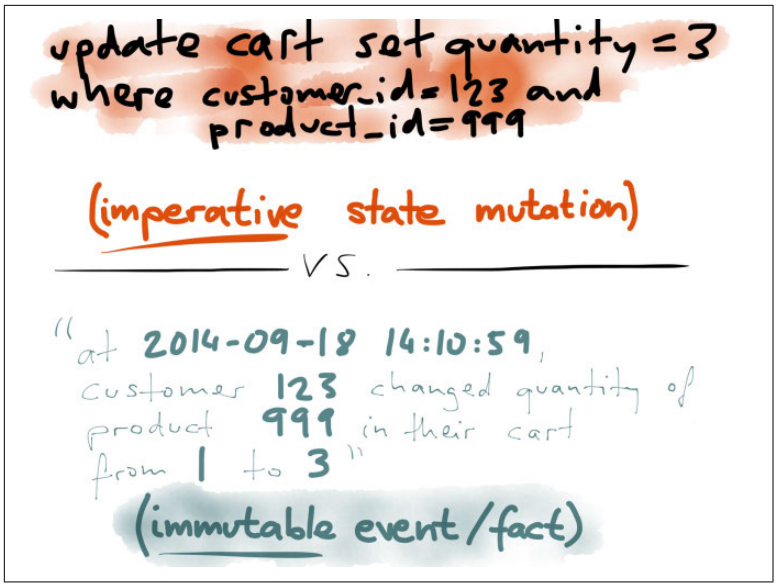


Figure 5-3. In a logical replication log, imperative commands are transformed into immutable change events.

At the top of Figure 5-3, we have the update statement, an imperative statement describing the state mutation. It is an instruction to the database, telling it to modify certain rows in the database that match certain conditions.

On the other hand, when the write is replicated from the leader to the follower as part of the logical log, it takes a different form: it becomes an event, stating that at a particular point in time, a particular customer changed the quantity of a particular product in their cart from 1 to 3. This is a *fact*—even if the customer later removes the item from their cart, or changes the quantity again, or goes away and never comes back, that doesn't change the fact that this state change occurred. The fact always remains true.

We can see that a change event in the replication log actually looks quite similar to an event in the sense of event sourcing (Chapter 1). Thus, even if you use your database in the traditional way—overwriting old state with new state—the database's internal replication mechanism may still be translating those imperative statements into a stream of immutable events.

Hold that thought for now; I'm going to talk about some completely different things and return to this idea later.

2. Secondary Indexes

Our second example of things that databases do is *secondary indexing*. You're probably familiar with secondary indexes; they are the bread and butter of relational databases.

Let's use the shopping cart example again (Figure 5-4): to efficiently find all the items that a particular customer has in their cart, you need an index on `customer_id`. If you also create an index on `product_id`, you can efficiently find all the carts that contain a particular product.

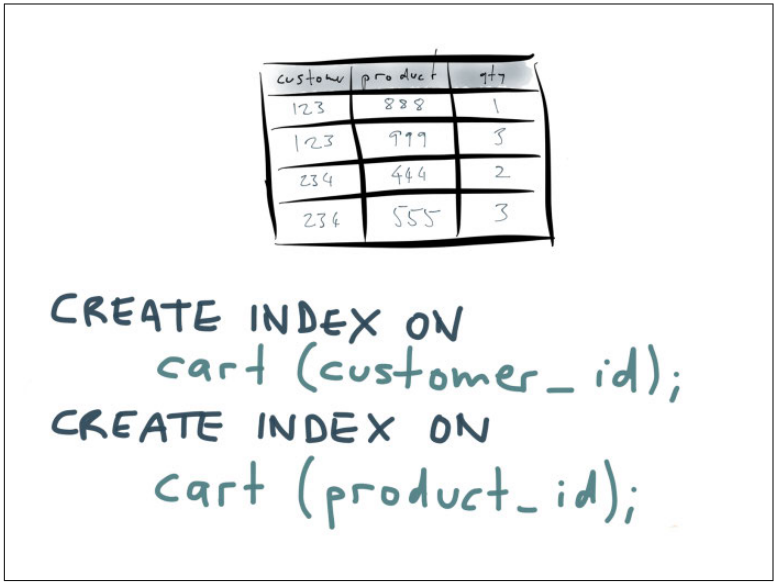


Figure 5-4. Secondary indexes allow you to efficiently look up rows by their value in a particular column.

What does the database do when you run one of these `CREATE INDEX` queries? It scans over the entire table, and it creates an auxiliary data structure for each index (Figure 5-5).

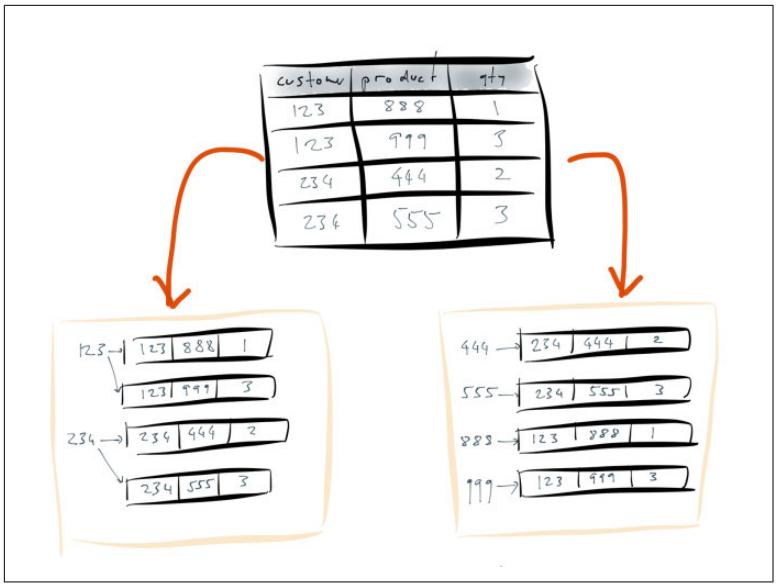


Figure 5-5. Each index is a separate data structure that is derived from the rows in the table.

An index is a data structure that represents the information in the base table in some different way. In this case, the index is a key-value-like structure: the keys are the contents of the column that you're indexing, and the values are the rows that contain this particular key.

Put another way: to build the index for the `customer_id` column, the database takes all the values that appear in that column, and uses them as keys in a dictionary. A value points to all occurrences of that value—for example, the index entry 123 points to all of the rows that have a `customer_id` of 123. This index construction is illustrated in Figure 5-6.

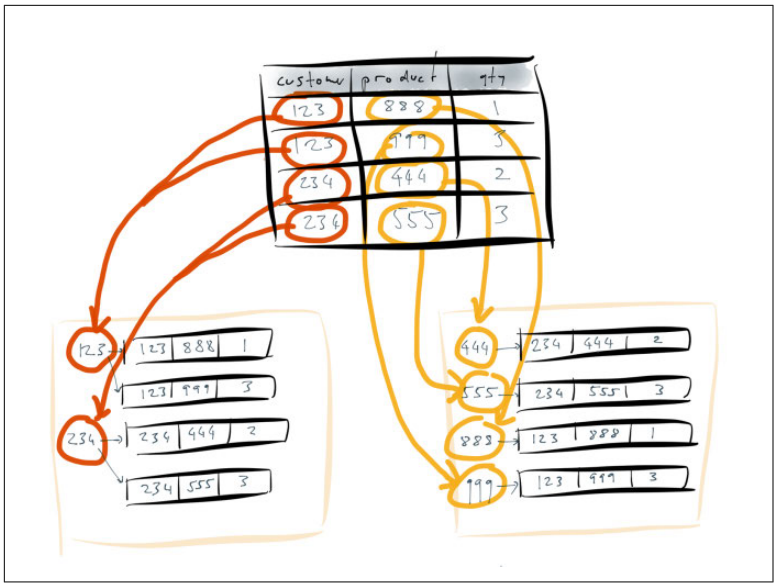


Figure 5-6. Values in the table become keys in the index.

The important point here is that the process of going from the base table to the indexes is *completely mechanical*. You simply tell the database that you want a particular index to exist, and it goes away and builds that index for you.

The index doesn't add any new information to the database—it just represents the same data in a different form. (Put another way, if you drop the index, that doesn't delete any data from your database; see also Figure 2-5.) An index is a redundant data structure that only exists to make certain queries faster, and that can be entirely *derived* from the original table (Figure 5-7).

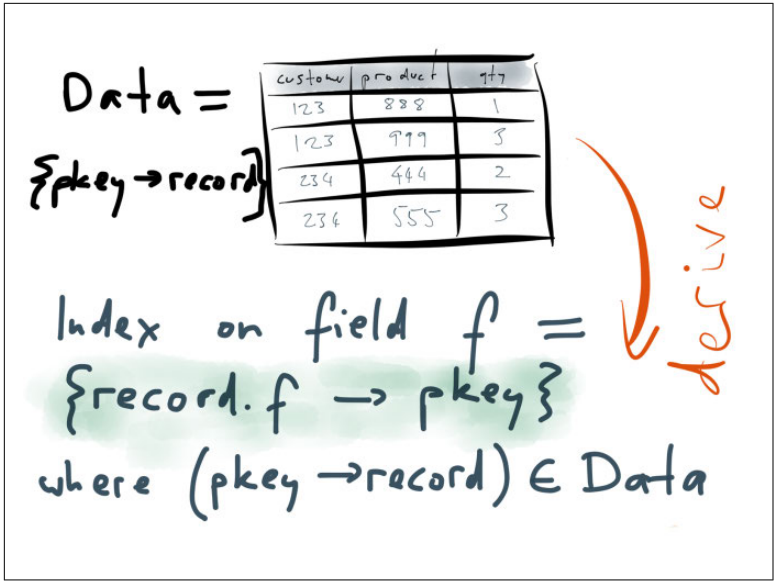


Figure 5-7. An index is derived from the data in the table by using a deterministic transformation.

Creating an index is essentially a transformation which takes a database table as input and produces an index as output. The transformation consists of going through all the rows in the table, picking out the field that you want to index, and restructuring the data so that you can look up by that field. That transformation process is built into the database, so you don't need to implement it yourself. You just tell the database that you want an index on a particular field to exist, and it does all the work of building it.

Here's another great thing about indexes: whenever the data in the underlying table changes, the database automatically updates the indexes to be consistent with the new data in the table. In other words, this transformation function which derives the index from the original table is not just applied once when you create the index: it's applied continuously.

With many databases, these index updates are even done in a *transactionally consistent way*. This means that any later transactions will see the data in the index in the same state as it is in the underlying table. If a transaction aborts and rolls back, the index modifications are also rolled back. This is a really great feature that we often don't appreciate!

Figure 5-8. The `CONCURRENTLY` option in PostgreSQL allows an index to be built without locking the base table for writes.

Moreover, some databases let you build an index at the same time as continuing to process write queries. In PostgreSQL, for example, you can say `CREATE INDEX CONCURRENTLY` (Figure 5-8). On a large table, creating an index could take several hours, and on a production database, you wouldn't want to have to stop writing to the table while the index is being built. The index builder needs to be a background process that can run while your application is simultaneously reading and writing to the database as usual.

The fact that databases can do this is quite impressive. After all, to build an index, the database must scan the entire table contents, but those contents are changing at the same time as the scan is happening. The index builder is tracking a moving target. At the end, the database ends up with a transactionally consistent index, despite the fact that the data was changing concurrently.

To do this, the database needs to build the index from a consistent snapshot at one point in time. It also needs to keep track of all the changes that occurred since that snapshot while the index build was in progress. The procedure is remarkably similar to what we saw in Chapter 3 in the context of change capture (Figure 3-2). Creating a

new index outside of the database (Figure 3-7) is not that different from creating a new index inside of the database.

So far, we've discussed two aspects of databases: replication and secondary indexing. Let's move on to the third: caching.

3. Caching

What I'm talking about here is caching that is explicitly done by the application. (Caching also happens automatically at various levels, such as the operating system's page cache and the CPU's cache, but that's not what I'm referring to here.)

Suppose that you have a website that becomes popular, and it becomes too expensive or too slow to hit the database for every web request, so you introduce a caching layer—often implemented by using memcached or Redis or something of that sort. Often this cache is managed in application code, which typically looks something like Figure 5-9.

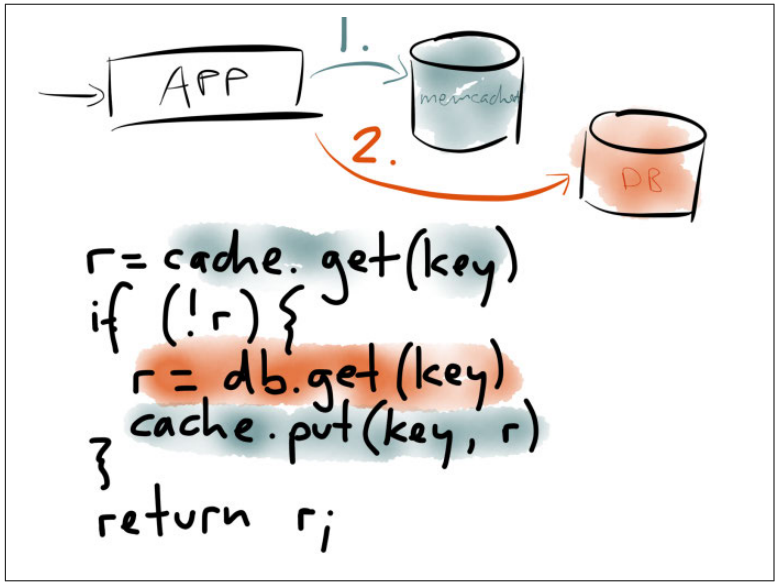


Figure 5-9. A read-through cache managed in application code.

When a request arrives at the application, you first look in a cache to see whether the data you want is already there. The cache lookup is typically by some key that describes the data you want. If the data is in the cache, you can return it straight to the client.

If the data you want isn't in the cache, that's a cache miss. You then go to the underlying database and query the data that you want. On the way out, the application also writes that data to the cache so that it's there for the next request that needs it. The thing it writes to the cache is whatever the application would have wanted to see there in the first place. Then, the application returns the data to the client.

This is a very common pattern, but there are several big problems with it (Figure 5-10).

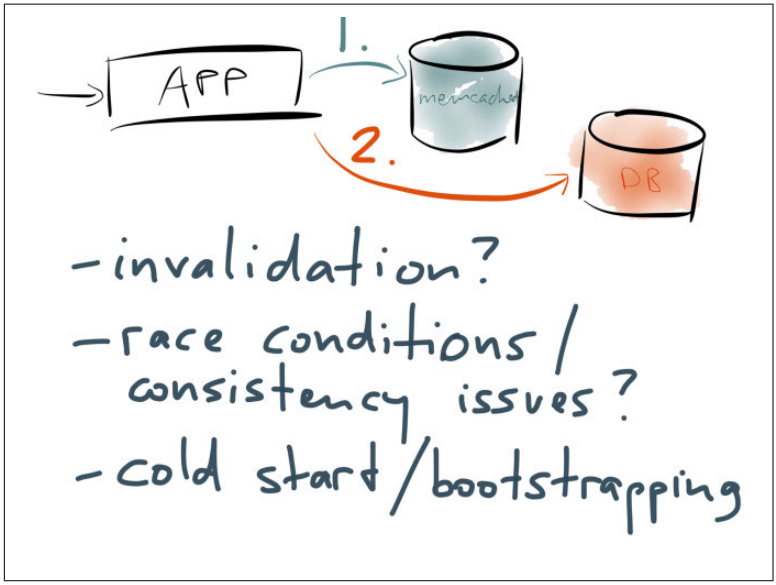


Figure 5-10. Problems with application-managed read-through caches.

Cache invalidation is considered by some to be a difficult problem to the point of cliché.[1] When data in the underlying database changes, how do you know which entries in the cache to expire or update? Figuring out which database change affects which cache entries is tractable for simple data models, and algorithms such as generational caching and russian-doll caching[2] are used. For more complex data dependencies, invalidation algorithms become com-

1 Phil Karlton: "There are only two hard things in Computer Science: cache invalidation and naming things." Quoted on martinfowler.com.

2 David Heinemeier Hansson: "How Basecamp Next got to be so damn fast without using much client-side UI," signalvnoise.com, 18 February 2012.

plex, brittle, and error-prone. Some applications side-step the problem by using only a time-to-live (expiry time) and accepting that they sometimes read stale data from the cache.

Another problem is that this architecture is very prone to race conditions. In fact, it is an example of the *dual-writes* problem that we saw in Chapter 2 (Figure 2-9): several clients concurrently accessing the same data can cause the cache to become inconsistent with the database.

A third problem is cold start. If you reboot your memcached servers and they lose all their cached contents, suddenly every request is a cache miss, the database is overloaded because of the sudden surge in requests, and you're in a world of pain. If you want to create a new cache, you need some way of bootstrapping its contents without overloading other parts of the system.

So, here we have a contrast (Figure 5-11). On the one hand, creating a secondary index in a database is beautifully simple, one line of SQL—the database handles it automatically, keeping everything up-to-date and even making the index transactionally consistent. On the other hand, application-level cache maintenance is a complete mess of complicated invalidation logic, race conditions, and operational problems.

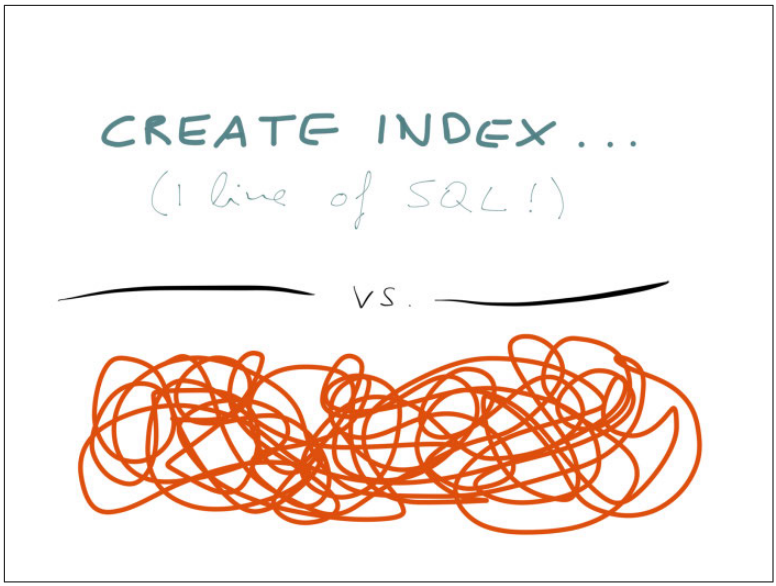


Figure 5-11. Databases hide the complexity of creating a secondary index behind a simple interface, but application-level cache maintenance is a complete mess.

Why should it be that way? Secondary indexes and caches are not fundamentally different. We said earlier that a secondary index is just a redundant data structure on the side, which structures the same data in a different way, in order to speed up read queries. If you think about it, a cache is also the result of taking your data in one form (the form in which it's stored in the database) and transforming it into a different form for faster reads. In other words, the contents of the cache are derived from the contents of the database (Figure 5-12) — very similar to an index.

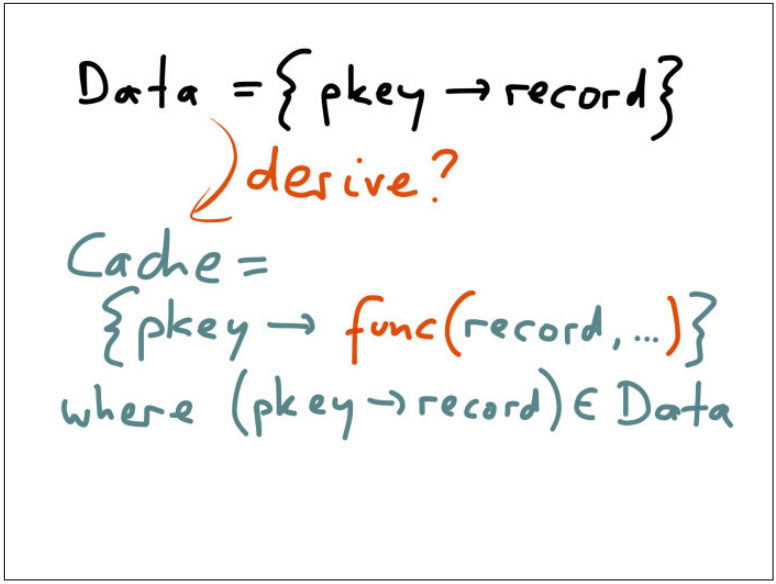


Figure 5-12. Similarly to an index, the contents of a cache are derived from the contents of the database.

We said that a secondary index is built by picking out one field from every record and using that as the key in a dictionary (Figure 5-7). In the case of a cache, we may apply an arbitrary function to the data (Figure 5-12): the data from the database may have gone through some kind of business logic or rendering before it's put in the cache, and it may be the result of joining several records from different tables. But, the end result is similar: if you lose your cache, you can rebuild it from the underlying database; thus, the contents of the cache are derived from the database.

In a read-through cache, this transformation happens on the fly, when there is a cache miss. However, we could perhaps imagine making the process of building and updating a cache more systematic, and more similar to secondary indexes. Let's return to that idea later.

Now, let's move on to the fourth idea about databases: *materialized views*.

4. Materialized Views

You might already know what materialized views are, but let me explain them briefly in case you've not previously come across them. You might be more familiar with "normal" views—non-materialized views, or virtual views, or whatever you want to call them.

They work like this: in a relational database, where views are common, you would create a view by saying "`CREATE VIEW view name...`" followed by a `SELECT` query (Figure 5-13).

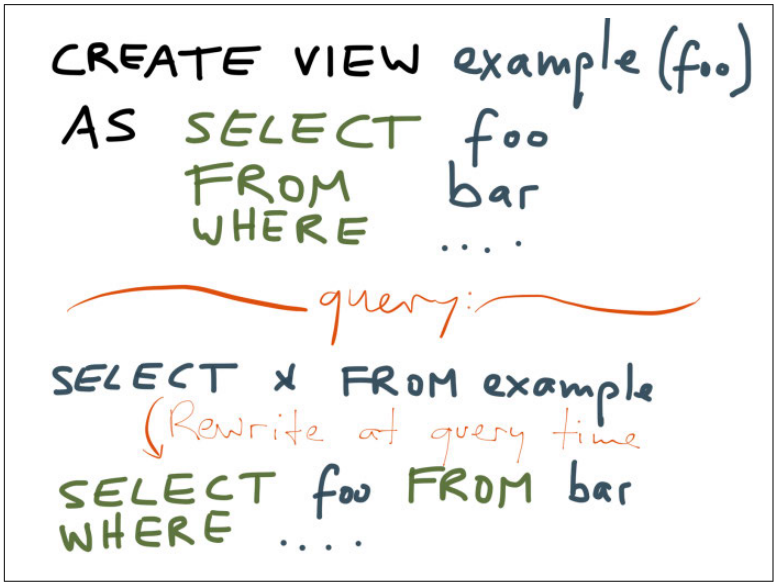


Figure 5-13. A non-materialized (virtual) view is just an alias for a query; when you read from the view, the database translates it into the underlying query.

When you look at this view in the database, it looks somewhat like a table—you can use it in read queries like any other table. And when you do this, say you `SELECT *` from that view, the database's query planner actually rewrites the query into the underlying query that you used in the definition of the view.

So, you can think of a view as a kind of convenient alias, a wrapper that allows you to create an abstraction, hiding a complicated query behind a simpler interface—but it has no consequences for performance or data storage.

Contrast that with a *materialized* view, which is defined using almost identical syntax (see Figure 5-14).

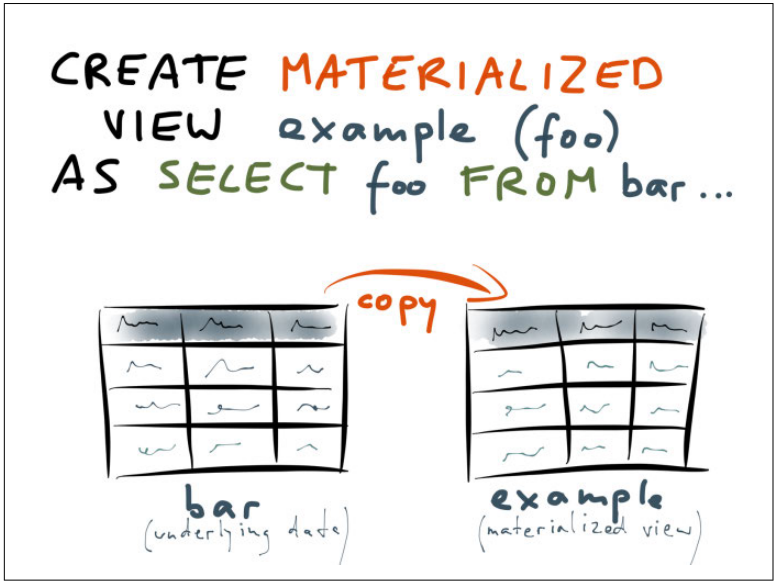


Figure 5-14. Materialized view: very similar syntax, very different implementation.

You also define a materialized view in terms of a `SELECT` query; the only syntactic difference is that you say `CREATE MATERIALIZED VIEW` instead of `CREATE VIEW`. However, the implementation is totally different.

When you create a materialized view, the database starts with the underlying tables—that is, the tables you're querying in the `SELECT` statement of the view ("`bar`" in the example). The database scans over the entire contents of those tables, executes that `SELECT` query on all of the data, and copies the results of that query into something like a temporary table.

The results of this query are actually written to disk, in a form that's very similar to a normal table. And that's really what "materialized" means in this context: the view's query has been executed, and the results written to disk.

Remember that with the non-materialized view, the database would expand the view into the underlying query at query time. On the other hand, when you query a materialized view, the database can

read its contents directly from the materialized query result because the view's underlying query has already been executed ahead of time. This is especially useful if the underlying query is expensive.

If you're thinking, "this seems like a cache of query results," you would be right—that's exactly what it is. However, the big difference between a materialized view and application-managed caches is the responsibility for keeping it up to date.

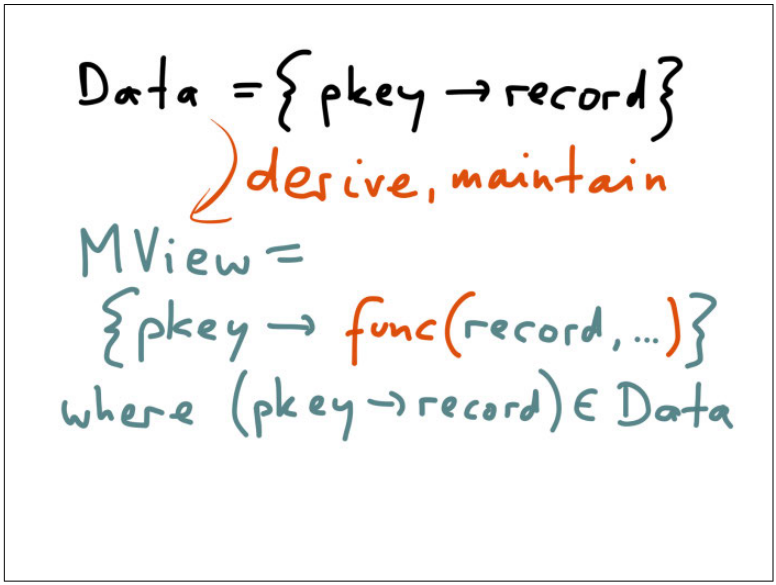


Figure 5-15. Like caches and secondary indexes, materialized views are also redundant data that is derived from the underlying tables.

With a materialized view, you declare once how you want the materialized view to be defined, and the database takes care of building that view from a consistent snapshot of the underlying tables (Figure 5-15, much like building a secondary index). Moreover, when the data in the underlying tables changes, the database takes responsibility for maintaining the materialized view, keeping it up-to-date. Some databases do this materialized view maintenance on an ongoing basis, and some require you to periodically refresh the view so that changes take effect, but you certainly don't have to do cache invalidation in your application code.

An advantage of application-managed caches is that you can apply arbitrary business logic to the data before storing it in the cache so

that you can do less work at query time or reduce the amount of data you need to cache. Doing the same in a materialized view would require that you run your application code in the database as a stored procedure (Figure 4-10). As discussed in Chapter 4, this is possible in principle, but often operationally problematic in practice. However, materialized views address the concurrency control and bootstrapping problems of caches (Figure 5-10).

Summary: Four Database-Related Ideas

Let's recap the four ideas we discussed: replication, secondary indexing, caching, and materialized views (Figure 5-16). What they all have in common is that they are dealing with *derived data* in some way: some secondary data structure is derived from an underlying, primary dataset, via a transformation process.

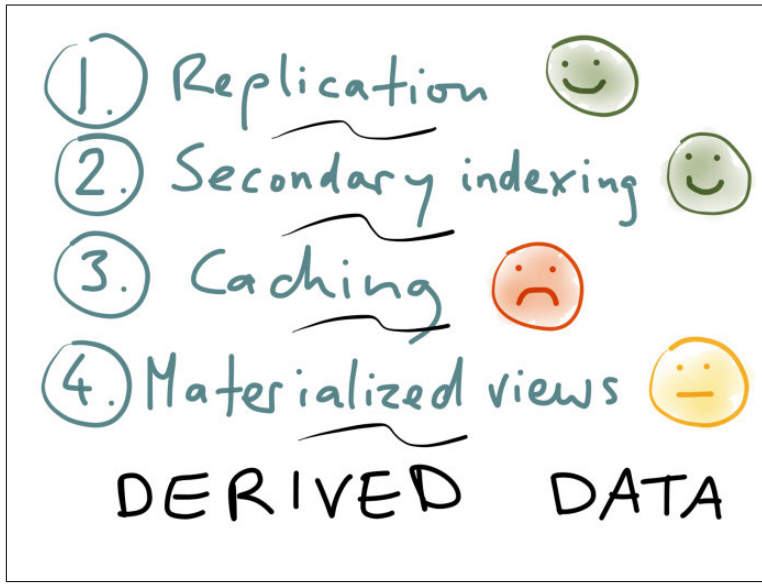


Figure 5-16. All four aspects of a database deal with derived data.

In Figure 5-16, I've given each point a rating (smile, neutral, frown) to indicate how well it works. Here's a quick recap:

Replication
: We first discussed replication; that is, keeping a copy of the same data on multiple machines. It generally works very well. There are some operational quirks with some databases, and

some of the tooling is a bit weird. But on the whole, it's mature, well understood, and well supported.

Secondary indexing
: Similarly, secondary indexing works very well. You can build a secondary index concurrently with processing write queries, and the database somehow manages to do this in a transactionally consistent way.

Caching
: Application-level read-through caching is a complete mess of complexity, race conditions, and operational problems.

Materialized views
: Materialized views are so-so: the idea is good, but the way they're implemented is not what you'd want from a modern application development platform. Maintaining the materialized view puts additional load on the database, whereas the entire point of a cache is to *reduce* load on the database!

Materialized Views: Self-Updating Caches

There's something really compelling about the idea of materialized views. I see a materialized view almost as a kind of cache that magically keeps itself up to date. Instead of putting all of the complexity of cache invalidation in the application (risking race conditions and all of the problems we have discussed), materialized views say that cache maintenance should be the responsibility of the *data infrastructure*.

So, let's think about this: can we reinvent materialized views, implement them in a modern and scalable way, and use them as a general mechanism for cache maintenance? If we started with a clean slate, without the historical baggage of existing databases, what would the ideal architecture for applications look like (Figure 5-17)?

Figure 5-17. What would materialized views look like if we started with a clean slate?

In Chapter 3, we discussed building a completely new index using the events in a log-compacted Kafka topic and then keeping it up-to-date by continuously consuming events from the log and applying them to the index. Whether we call this an index, or a cache, or a materialized view does not make a big difference: they are all derived representations of the data in the log (Figure 5-18).

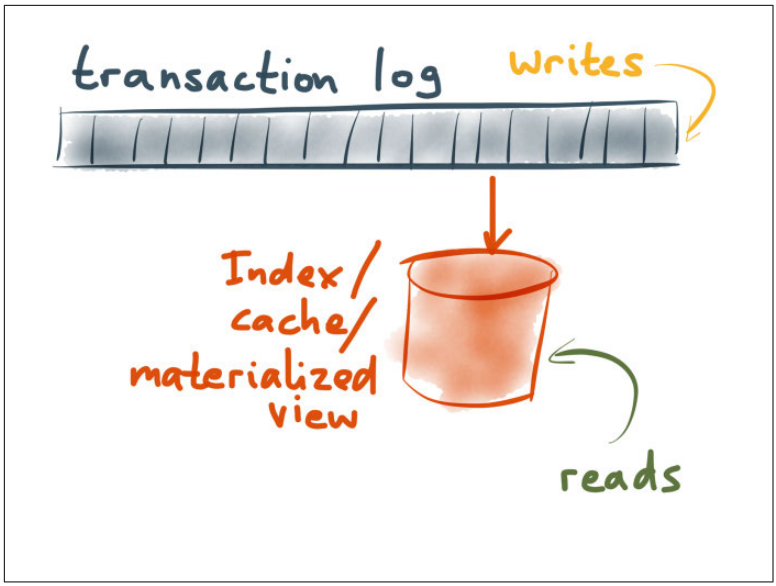


Figure 5-18. An index, a cache and a materialized view are all just projections of the log into a read-optimized structure.

The difference is that an index is typically built by extracting one field from an event, and using it as lookup key (Figure 5-6), whereas constructing a cache or a materialized view might require more complex transformations:

- In a materialized view, you might want data from several sources to be joined together into a denormalized object, to save having to perform the join at read time. For example, in Figure 1-17, each tweet contains only the user_id of the author, but when reading tweets, you want the tweet to be joined with the user profile information (the username, profile photo, etc.).
- The materialized view can contain aggregate functions such as sum or count (e.g., the number of likes in Figure 1-20, or the count of unread messages in Figure 2-10).
- You might need some arbitrary business logic to be applied (e.g., to honor the user's privacy settings).

Stream processing frameworks allow you to implement such joins, aggregations, and arbitrary business logic—we will look at an example shortly.

Let's also be clear about how a materialized view is different from a cache (Figure 5-19).

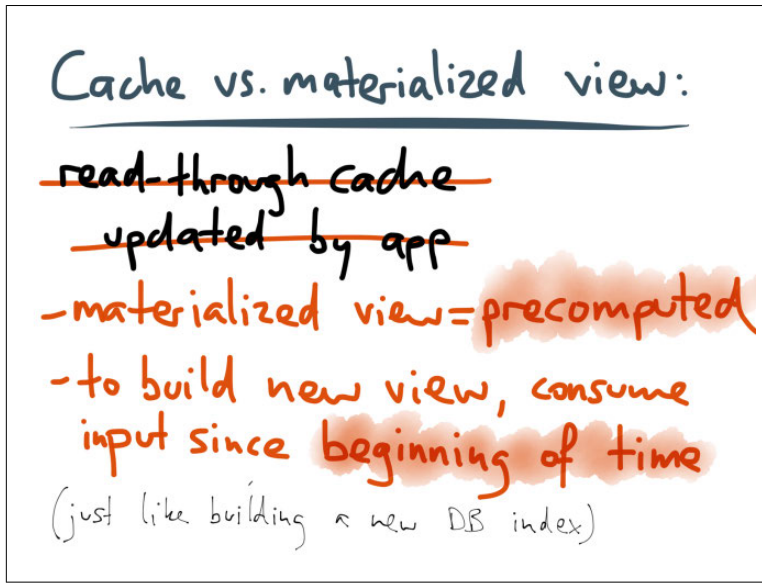


Figure 5-19. Advantages of a materialized view over an application-managed read-through cache.

As discussed, an application-managed read-through cache is invalidated or updated directly by application code, whereas a materialized view is maintained by consuming a log. This has some important advantages:

- A cache is filled on demand when there is a cache miss (so the first request for a given object is always slow, and you have the cold-start problem mentioned in Figure 5-10). By contrast, a materialized view is *precomputed*; that is, its entire contents are computed before anyone asks for it—just like an index. This means there is no such thing as a cache miss: if an item doesn't exist in the materialized view, it doesn't exist in the database. There is no need to fall back to some other underlying database. (This doesn't mean the entire view has to be in memory: just like an index, it can be written to disk, and the hot parts will automatically be kept in memory in the operating system's page cache.)

- With a materialized view there is a well-defined *translation process* that takes the write-optimized events in the log and transforms them into the read-optimized representation in the view. By contrast, in the typical read-through caching approach, the cache management logic is deeply interwoven with the rest of the application, making it prone to bugs and difficult to reason about.
- That translation process runs in a stream processor which you can test, deploy, monitor, debug, scale, and maintain independently from the rest of your application. The stream processor consumes events in log order, making it much less susceptible to race conditions. If it fails and is restarted, it just keeps going from where it left off. If you deploy bad code, you can re-run the stream processor on historical data to fix up its mistakes.
- With log compaction, you can build a brand new index by processing a stream from the beginning (Figure 3-7); the same is true of materialized views. If you want to present your existing data in some new way, you can simply create a new stream processing job, consume the input log from the beginning, and thus build a completely new view onto all the existing data. You can then maintain both views in parallel, gradually move clients to the new view, run A/B tests across the two views, and eventually discard the old view. No more scary stop-the-world schema migrations.

Example: Implementing Twitter

Let's make materialized views more concrete by looking at an example. In Chapter 1, we looked at how you might implement a Twitter-like messaging service. The most common read operation on that service is requesting the "home timeline"; that is, you want to see all recent tweets by users you follow (including username and profile picture for the sender of each tweet, see Figure 1-17).

In Figure 1-18, we saw a SQL query for a home timeline, but we noted that it is too slow to execute that query on every read. Instead, we need to precompute each user's home timeline ahead of time so that it's already there when the user asks for it. Sounds a bit like a materialized view, doesn't it?

No existing database is able to provide materialized views at Twitter's scale, but such materialized timelines can be implemented using stream processing tools.[3] Figure 5-20 shows a sketch of how you might do this.[4]

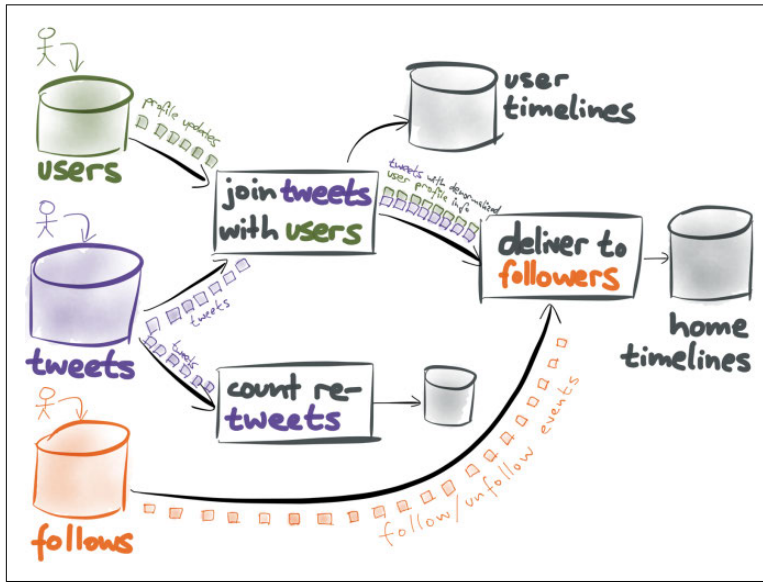


Figure 5-20. Implementing Twitter timelines by using a stream processing system.

To start with, you need to make all data sources available as event streams, either by using CDC (Chapter 3) or by writing events directly to a log (Chapter 2). In this example, we have event streams from three data sources:

Tweets
: Whenever a tweet is sent or retweeted, that is an event. It is quite natural to think of these as a stream.

User profiles
: Every time a user changes their username or profile picture, that is a profile update event. This stream needs to be log-

3 Raffi Krikorian: "Timelines at Scale," at *QCon San Francisco*, November 2012.

4 Martin Kleppmann: "Samza newsfeed demo," github.com, September 2014.

compacted, so that you can reconstruct the latest state of all user profiles from the stream.

Follow graph
: Every time someone follows or unfollows another user, that's an event. The full history of these events determines who is following whom.

If you put all of these streams in Kafka, you can create materialized views by writing stream processing jobs using Kafka Streams or Samza. For example, you can write a simple job that counts how many times a tweet has been retweeted, generating a "retweet count" materialized view.

You can also join streams together. For example, you can join tweets with user profile information, so the result is a stream of tweets in which each tweet carries a bit of denormalized profile information (e.g., username and profile photo of the sender). When someone updates their profile, you can decide whether the change should take effect only for their future tweets, or also for their most recent 100 tweets, or for every tweet they ever sent—any of these can be implemented in the stream processor. (It may be inefficient to rewrite thousands of cached historical tweets with a new username, but this is something you can easily adjust, as appropriate.)

Next, you can join tweets with followers. By collecting follow/unfollow events, you can build up a list of all users who currently follow user X. When user X tweets something, you can scan over that list, and deliver the new tweet to the home timeline of each of X's followers (Twitter calls this *fan-out*[5]).

Thus, the home timelines are like a mailbox, containing all the tweets that the user should see when they next log in. That mailbox is continually updated as people send tweets, update their profiles, and follow and unfollow one another. We have effectively created a materialized view for the SQL query in Figure 1-18. Note that the two joins in that query correspond to the two stream joins in Figure 5-20: the stream processing system is like a continuously running query execution graph!

5 Raffi Krikorian: "Timelines at Scale," at *QCon San Francisco*, November 2012.

The Unbundled Database

What we see here is an interesting pattern: derived data structures (indexes, materialized views) have traditionally been implemented internally within a monolithic database, but now we are seeing similar structures increasingly being implemented at the application level, using stream processing tools.

This trend is driven by need: nobody would want to re-implement these features in a production system if existing databases already did the job well enough. Building database-like features is difficult: it's easy to introduce bugs, and many storage systems have high reliability requirements. Our discussion of read-through caching shows that data management at the application level can get very messy.

However, for better or for worse, this trend is happening. We are not going to judge it; we're going to try only to understand it and learn some lessons from the last few decades of work on databases and operating systems.

Earlier in this chapter (Figure 5-2) we observed that a database's replication log can look quite similar to an event log that you might use for event sourcing. The big difference is that an event log is an application-level construct, whereas a replication log is traditionally considered to be an implementation detail of a database (Figure 5-21).

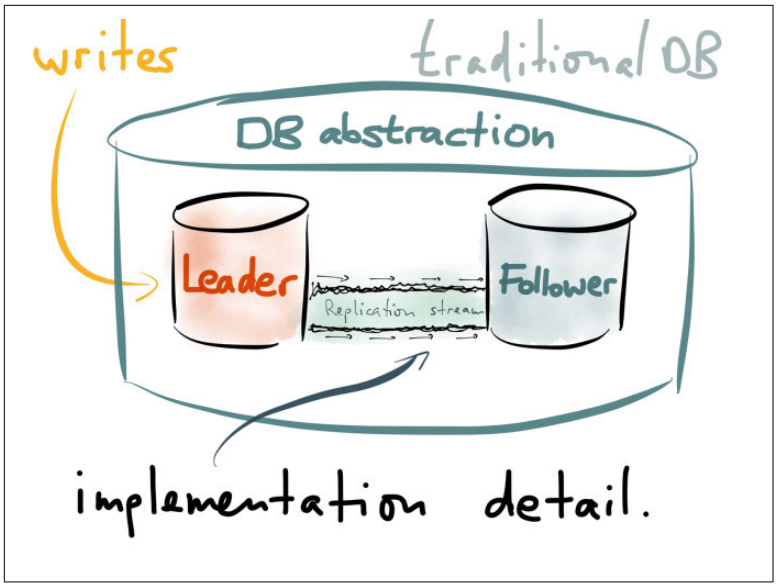


Figure 5-21. In traditional database architecture, the replication log is considered an implementation detail, not part of the database's public API.

SQL queries and responses are traditionally the database's public interface—and the replication log is an aspect that is hidden by that abstraction. (Change data capture is often retrofitted and not really part of the public interface.)

One way of interpreting stream processing is that it *turns the database inside out*: the commit log or replication log is no longer relegated to being an implementation detail; rather, it is made a first-class citizen of the application's architecture. We could call this a *log-centric architecture*, and interestingly, it begins to look somewhat like a giant distributed database:[6]

- You can think of various NoSQL databases, graph databases, time series databases, and full-text search servers as just being different index types. Just like a relational database might let you choose between a B-Tree, an R-Tree and a hash index (for

6 Jay Kreps: "The Log: What every software engineer should know about real-time data's unifying abstraction," engineering.linkedin.com, 16 December 2013.

example), your data system might write data to several different data stores in order to efficiently serve different access patterns.

- The same data can easily be loaded into Hadoop, a data warehouse, or analytic database (without complicated ETL processes, because event streams are already analytics friendly) to provide business intelligence.
- The Kafka Streams library and stream processing frameworks such as Samza are scalable implementations of triggers, stored procedures and materialized view maintenance routines.
- Datacenter resource managers such as Mesos or YARN provide scheduling, resource allocation, and recovery from physical machine failures.
- Serialization libraries such as Avro, Protocol Buffers, or Thrift handle the encoding of data on the network and on disk. They also handle schema evolution (allowing the schema to be changed over time without breaking compatibility).
- A log service such as Apache Kafka or Apache BookKeeper[7] is like the database's commit log and replication log. It provides durability, ordering of writes, and recovery from consumer failures. (In fact, people have already built databases that use Kafka as transaction/replication log.[8])

In a traditional database, all of those features are implemented in a single monolithic application. In a log-centric architecture, each feature is provided by a different piece of software. The result looks somewhat like a database, but with its individual components "unbundled" (Figure 5-22).

7 "Apache BookKeeper," Apache Software Foundation, bookkeeper.apache.org.

8 Gavin Li, Jianqiu Lv, and Hang Qi: "Pistachio: co-locate the data and compute for fastest cloud compute," yahooeng.tumblr.com, 13 April 2015.

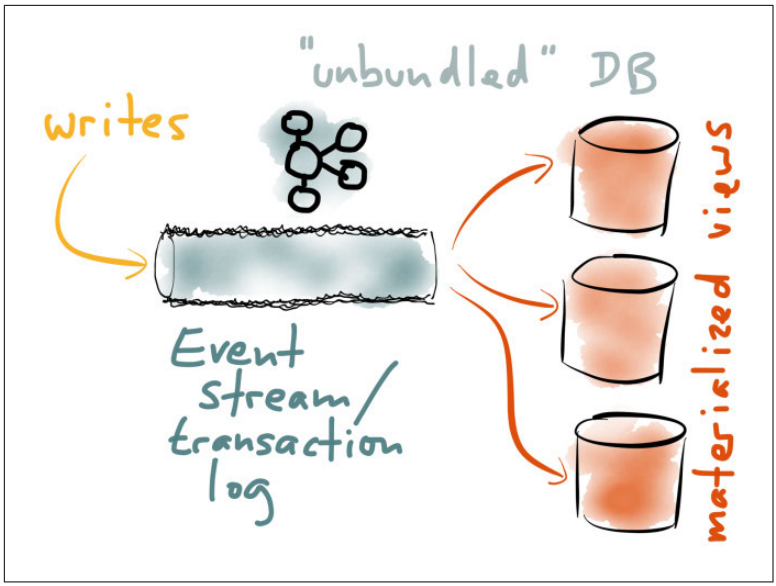


Figure 5-22. Updating indexes and materialized views based on writes in a log: more or less what a traditional database already does internally, at smaller scale.

In the unbundled approach, each component is a separately developed project, and many of them are open source. Each component is specialized: the log implementation does not try to provide indexes for random-access reads and writes—that service is provided by other components. The log can therefore focus its effort on being a really good log: it does *one thing well* (cf. Figure 4-3). A similar argument holds for other parts of the system.

The advantage of this approach is that each component can be developed and scaled independently, providing great flexibility and scalability on commodity hardware.[9] It essentially brings the Unix philosophy to databases: specialized tools are composed into an application that provides a complex service.

The downside is that there now many different pieces to learn about, deploy, and operate. Many practical details need to be figured out: how do we deploy and monitor these various components, how do

9 Jun Rao: "The value of Apache Kafka in Big Data ecosystem," odbms.org, 16 June 2015.

we make the system robust to various kinds of fault, how do we productively write software in this kind of environment (Figure 5-23)?

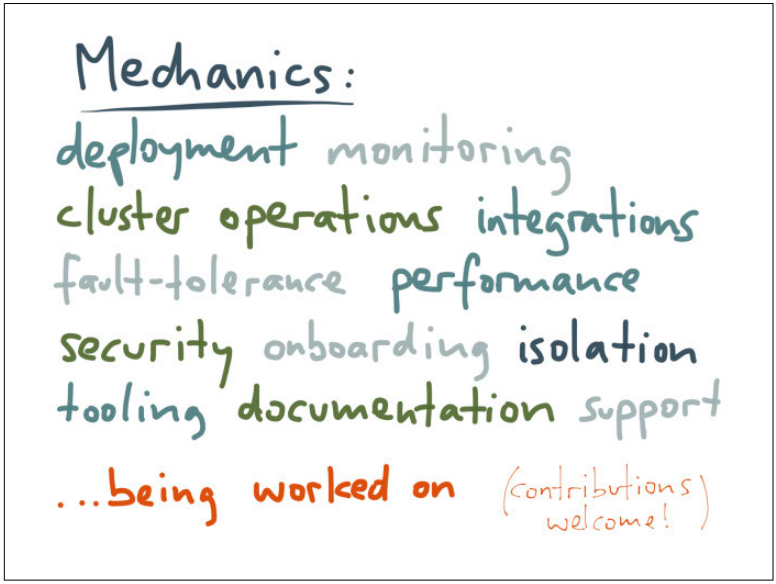


Figure 5-23. These ideas are new, and many challenges lie ahead on the path toward maturity.

Because many of the components were designed independently, without composability in mind, the integrations are not as smooth as one would hope (see change data capture, for example). And there is not yet a convincing equivalent of SQL or the Unix shell—that is, a high-level language for concisely describing data flows—for log-centric systems and materialized views. All in all, these systems are not nearly as elegantly integrated as a monolithic database from a single vendor.

Yet, there is hope. Linux distributions and Hadoop distributions are also assembled from many small parts written by many different groups of people, and they nevertheless feel like reasonably coherent products. We can expect the same will be the case with a Stream Data Platform.[10]

10 Neha Narkhede: "Announcing the Confluent Platform 2.0," confluent.io, 8 December, 2015.

This log-centric architecture for applications is definitely not going to replace databases, because databases are still needed to serve the materialized views. Also, data warehouses and analytic databases will continue to be important for answering ad hoc, exploratory queries.

I draw the comparison between stream processing and database architecture only because it helps clarify what is going on here: at scale, no single tool is able to satisfy all use cases, so we need to find good patterns for integrating a diverse set of tools into a single system. The architecture of databases provides a good set of patterns.

Streaming All the Way to the User Interface

Before we wrap up, there is one more thing we should talk about in the context of event streams and materialized views. (I saved the best for last!)

Imagine what happens when a user of your application views some data. In a traditional database architecture, the data is loaded from a database, perhaps transformed with some business logic, and perhaps written to a cache. Data in the cache is rendered into a user interface in some way—for example, by rendering it to HTML on the server, or by transferring it to the client as JSON and rendering it on the client.

The result of template rendering is some kind of structure describing the user interface layout: in a web browser, this would be the HTML DOM, and in a native application this would be using the operating system's UI components. Either way, a rendering engine eventually turns this description of UI components into pixels in video memory, and this is what the graphics device actually displays on the screen.

When you look at it like this, it looks very much like a data transformation pipeline (Figure 5-24). You can think of each lower layer as a materialized view onto the upper layer: the cache is a materialized view of the database (the cache contents are derived from the database contents); the HTML DOM is a materialized view of the cache (the HTML is derived from the JSON stored in the cache); and the pixels in video memory are a materialized view of the HTML DOM (the browser rendering engine derives the pixels from the UI layout).

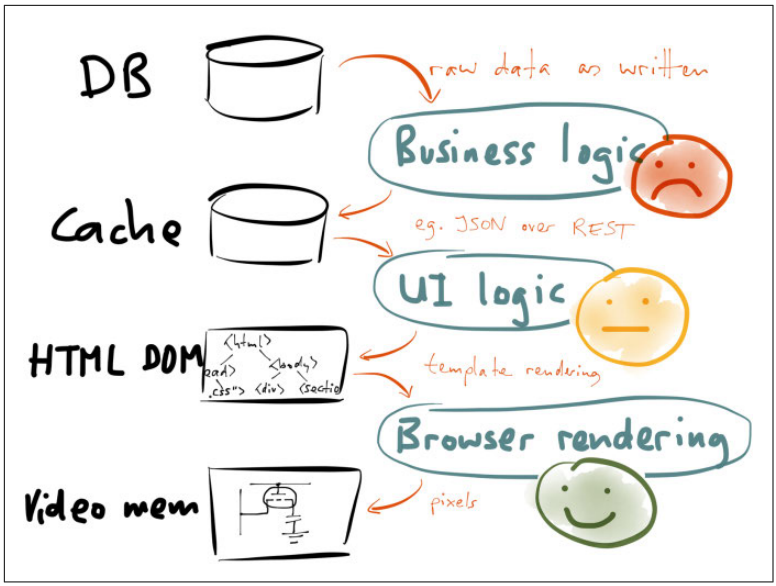


Figure 5-24. Rendering data on screen requires a sequence of transformation steps, not unlike materialized views.

Now, how well does each of these transformation steps work? I would argue that web browser rendering engines are brilliant feats of engineering. You can use JavaScript to change some CSS class, or have some CSS rules conditional on mouse-over, and the rendering engine automatically figures out which rectangle of the page needs to be redrawn as a result of the changes. It does hardware-accelerated animations and even 3D transformations. The pixels in video memory are automatically kept up to date with the underlying DOM state, and this very complex transformation process works remarkably well.

What about the transformation from data objects to user interface components? For now, I consider it "so-so," because the techniques for updating user interface based on data changes are still quite new. However, they are rapidly maturing: on the web, frameworks such as Facebook's React,[11] Angular,[12] and Ember[13] are enabling user inter-

11 "React," Facebook Inc., facebook.github.io.

12 "AngularJS," Google, Inc., angularjs.org.

13 "Ember," Tilde Inc., emberjs.com.

faces that can be updated from a stream, and Functional Reactive Programming (FRP) languages such as Elm[14] are in the same area. There is a lot of activity in this field, and it is heading in a good direction.

The transformation from database contents to cache entries is now the weakest link in this entire data-transformation pipeline. The problem is that a cache is request-oriented: a client can read from it, but if the data subsequently changes, the client doesn't find out about the change (it can poll periodically, but that soon becomes inefficient).

We are now in the bizarre situation in which the UI logic and the browser rendering engine can dynamically update the pixels on the screen in response to changes in the underlying data, but the database-driven backend services don't have a way of notifying clients about data changes. To build applications that quickly respond to user input (such as real-time collaborative apps), we need to make this pipeline work smoothly, end to end.

Fortunately, if we build materialized views that are maintained by using stream processors, as discussed in this chapter, we have the missing piece of the pipeline (Figure 5-25).

14 Evan Czaplicki: "Elm," elm-lang.org.

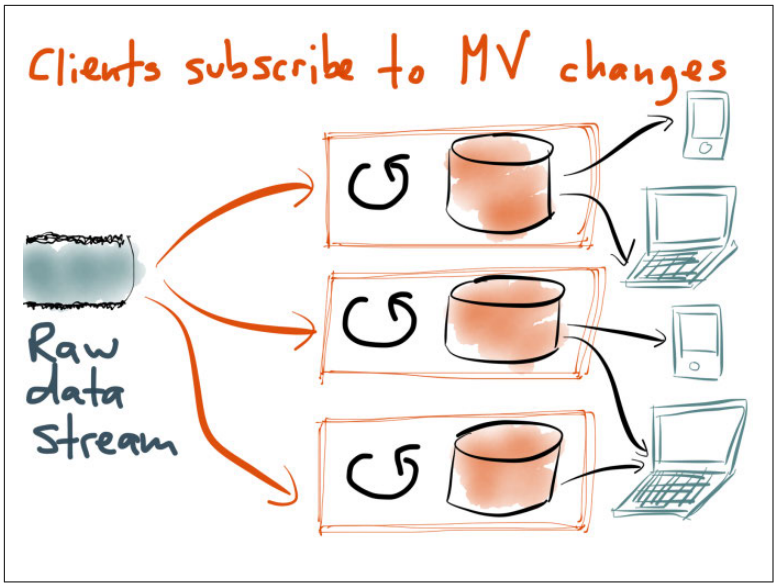


Figure 5-25. If you update materialized views by using an event stream, you can also push changes to those views to clients.

When a client reads from a materialized view, it can keep the network connection open. If that view is later updated, due to some event that appeared in the stream, the server can use this connection to notify the client about the change (for example, using a WebSocket[15] or Server-Sent Events[16]). The client can then update its user interface accordingly.

This means that the client is not just reading the view at one point in time, but actually subscribing to the stream of changes that may subsequently happen. Provided that the client's Internet connection remains active, the server can push any changes to the client, and the client can immediately render it. After all, why would you ever want outdated information on your screen if more recent information is available? The notion of static web pages, which are requested once and then never change, is looking increasingly anachronistic.

However, allowing clients to subscribe to changes in data requires a big rethink of the way we write applications. The request-response

15 "WebSockets," Mozilla Developer Network, developer.mozilla.org.

16 "Server-sent events," Mozilla Developer Network, developer.mozilla.org.

model is very deeply engrained in our thinking, in our network protocols and in our programming languages: whether it's a request to a RESTful service, or a method call on an object, the assumption is generally that you're going to make one request, and get one response. In most APIs there is no provision for an ongoing stream of responses.

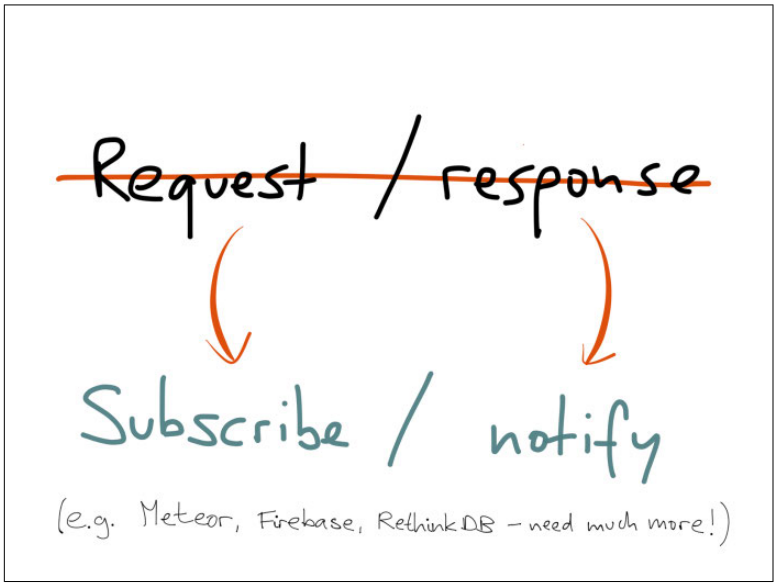


Figure 5-26. To support dynamically updated views we need to move away from request/response RPC models and use push-based publish-subscribe dataflow everywhere.

This will need to change. Instead of thinking of requests and responses, we need to begin thinking of subscribing to streams and notifying subscribers of new events (Figure 5-26). This needs to happen through all the layers of the stack—the databases, the client libraries, the application servers, the business logic, the frontends, and so on. If you want the user interface to dynamically update in response to data changes, that will only be possible if we systematically apply stream thinking everywhere so that data changes can propagate through all the layers.

Most RESTful APIs, database drivers, and web application frameworks today are based on a request/response assumption, and they will struggle to support streaming dataflow. In the future, I think we're going to see a lot more people using stream-friendly program-

ming models. We came across some of these in Chapter 1 (Figure 1-31): frameworks based on actors and channels, or *reactive* frameworks (ReactiveX, functional reactive programming), are a natural fit for applications that make heavy use of event streams.

I'm glad to see that some people are already working on better end-to-end support for event streams. For example, RethinkDB supports queries that notify the client if query results change.[17] Meteor[18] and Firebase[19] are frameworks that integrate the database backend and user interface layers so as to be able to push changes into the user interface. These are excellent efforts. We need many more like them (Figure 5-27).

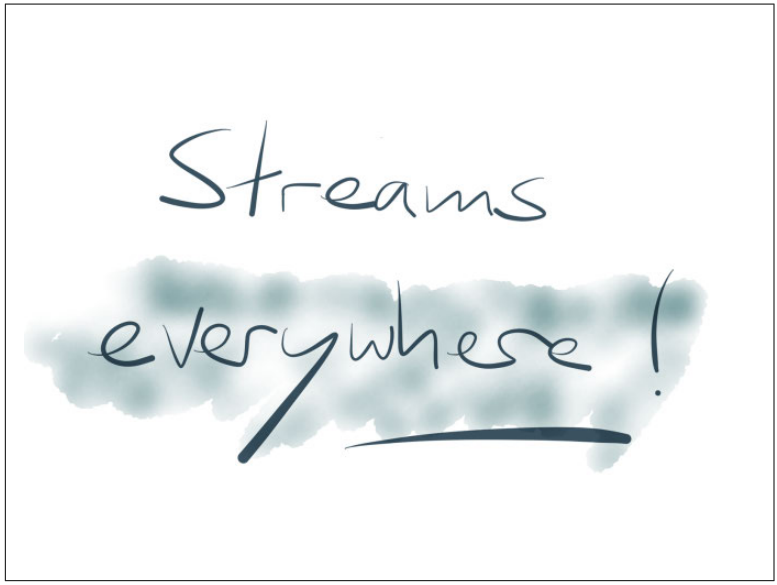


Figure 5-27. Event streams are a splendid idea. We should put them everywhere.

Conclusion

Application development is fairly easy if a single monolithic database can satisfy all of your requirements for data storage, access, and

17 Slava Akhmechet: "Advancing the realtime web," rethinkdb.com, 27 January 2015.

18 "Meteor," Meteor Development Group, meteor.com.

19 "Firebase," Google Inc., firebase.com.

processing. As soon as that is no longer the case—perhaps due to scale, or complexity of data access patterns, or other reasons—there is a lack of guidance and patterns to help application developers build reliable, scalable and maintainable applications.

In this report, we explored a particular architectural style for building large-scale applications, based on streams of immutable events (event logs). Stream processing is already widely used for analytics and monitoring purposes (e.g., finding certain patterns of events for fraud detection purposes, or alerting about anomalies in time series data), but in this report we saw that stream processing is also good for situations that are traditionally considered to be in the realm of OLTP databases: maintaining indexes and materialized views.

In this world view, the event log is regarded as the system of record (source of truth), and other datastores are derived from it through stream transformations (mapping, joining, and aggregating events). Incoming data is written to the log, and read requests are served from a datastore containing some projection of the data.

The following are some of the most important observations we made about log-centric systems:

- An event log such as Apache Kafka *scales very well*. Because it is such a simple data structure, it can easily be partitioned and replicated across multiple machines, and is comparatively easy to make reliable. It can achieve very high throughput on disks because its I/O is mostly sequential.
- If all your data is available in the form of a log, it becomes much easier to *integrate and synchronize data* across different systems. You can easily avoid race conditions and recover from failures if all consumers see events in the same order. You can rewind the stream and re-process events to build new indexes and recover from corruption.
- *Materialized views*, maintained through stream processors, are a good alternative to read-through caches. A view is fully precomputed (avoiding the cold-start problem, and allowing new views to be created easily) and kept up to date through streams of change events (avoiding race conditions and partial failures).
- Writing data as an event log produces *better-quality data* than if you update a database directly. For example, if someone adds an item to their shopping cart and then removes it again, your ana-

lytics, audit, and recommendation systems might want to know. This is the motivation behind event sourcing.

- Traditional database systems are based on the fallacy that data must be written in the same form as it is read. As we saw in Chapter 1, an application's *inputs often look very different from its outputs*. Materialized views allow us to write input data as simple, self-contained, immutable events, and then transform it into several different (denormalized or aggregated) representations for reading.
- Asynchronous stream processors usually don't have transactions in the traditional sense, but you can still guarantee *integrity constraints* (e.g., unique username, positive account balance) by using the ordering of the event log (Figure 2-31).
- *Change data capture* is a good way of bringing existing databases into a log-centric architecture. In order to be fully useful, it must capture both a consistent snapshot of the entire database, and also the ongoing stream of writes in transaction commit order.
- To support applications that dynamically update their user interface when underlying data changes, programming models need to *move away from a request/response assumption* and become friendlier to streaming dataflow.

We are still figuring out how to build large-scale applications well—what techniques we can use to make our systems scalable, reliable, and maintainable. However, to me, this approach of immutable events, stream processing, and materialized views seems like a very promising route forward. I am optimistic that this kind of application architecture will help us to build better software faster.

Fortunately, this is not science fiction—it's happening now. People are working on various parts of the problem and finding good solutions. The tools at our disposal are rapidly becoming better. It's an exciting time to be building software.

About the Author

Martin Kleppmann is a researcher and engineer in the area of distributed systems, databases and security at the University of Cambridge, UK. He previously co-founded two startups, including Rapportive, which was acquired by LinkedIn in 2012. Through working on large-scale production data infrastructure, experimental research systems, and various open source projects, he learned a few things the hard way.

Martin enjoys figuring out complex problems and breaking them down, making them clear and accessible. He does this in his conference talks, on his blog and in his book *Designing Data-Intensive Applications* (O'Reilly). You can find him as @martinkl on Twitter.