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).

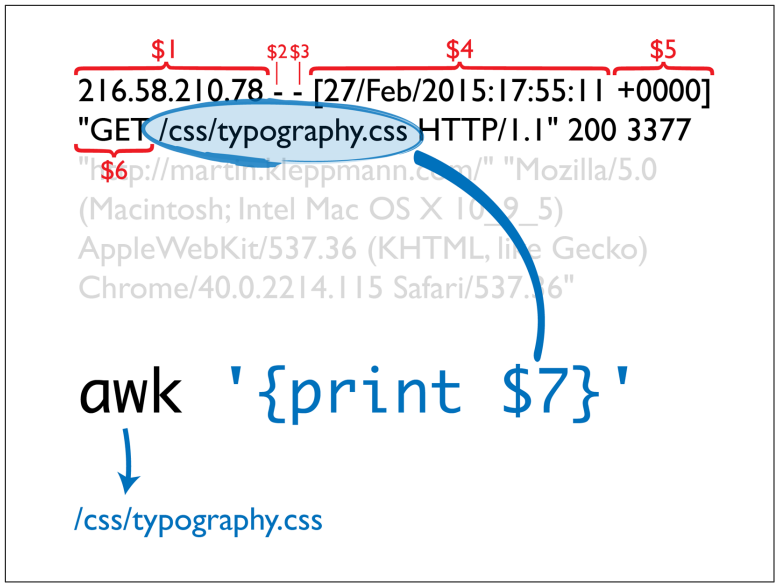


Figure 4-1. Extracting the requested URL path from a web server log by using awk.

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}'`.

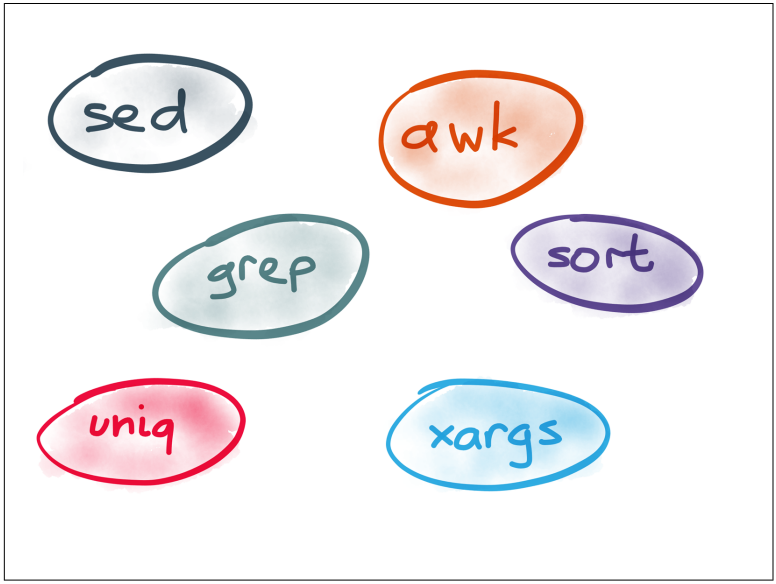


Figure 4-2. Unix: small, focused tools that combine well with one another.

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.

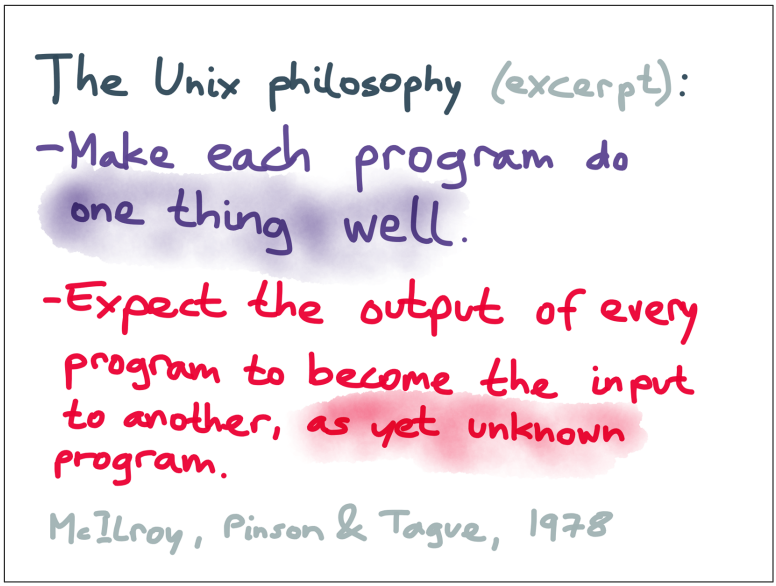


Figure 4-3. Two aspects of the Unix philosophy, as articulated by some of its designers in 1978.

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...

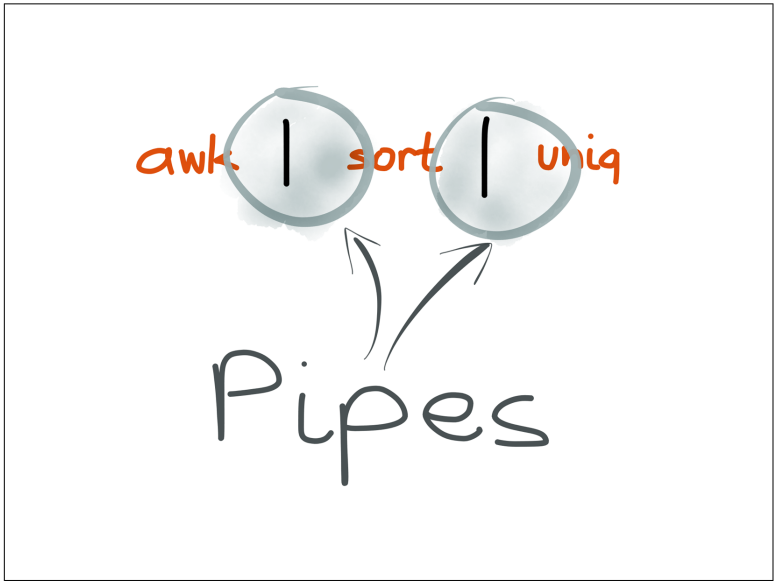


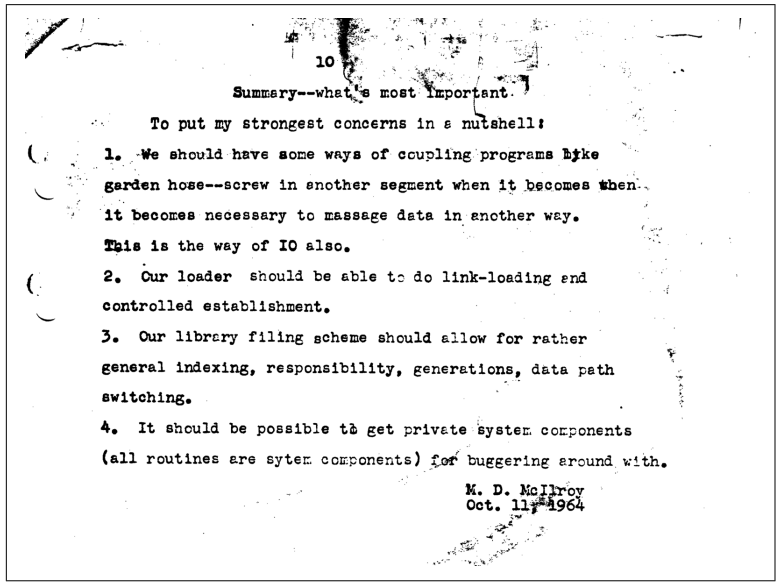
Figure 4-4. A Unix pipe joins the output of one process to the input of another.

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).

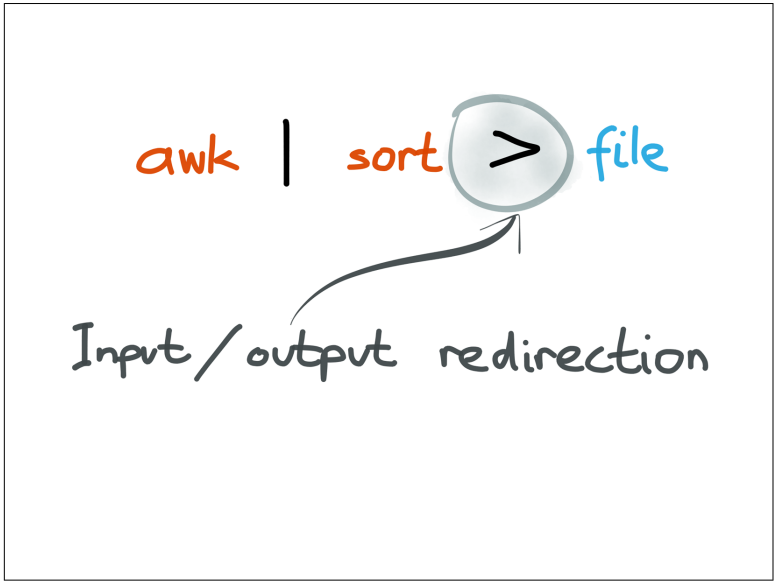


Figure 4-6. A process doesn't care whether its input and output are files on disk, or pipes to other processes.

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).

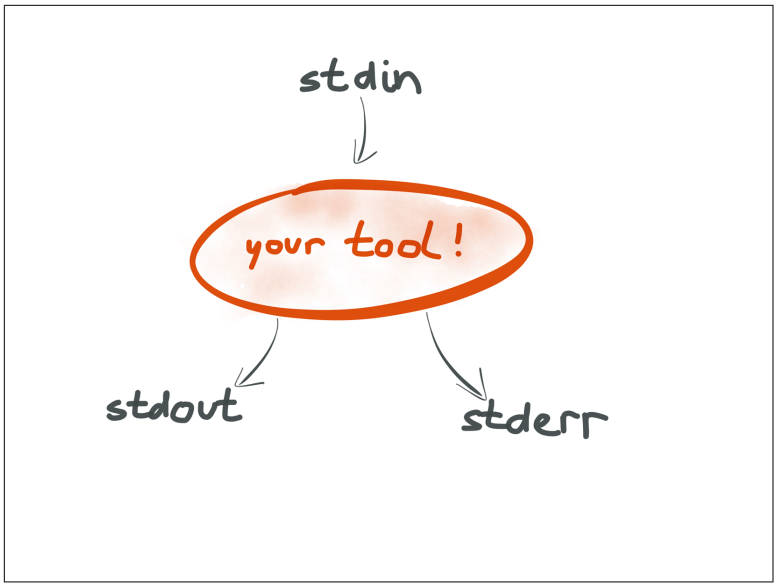


Figure 4-7. Unix tools all have the same interface of input and output streams. This standardization is crucial to enabling composability.

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.

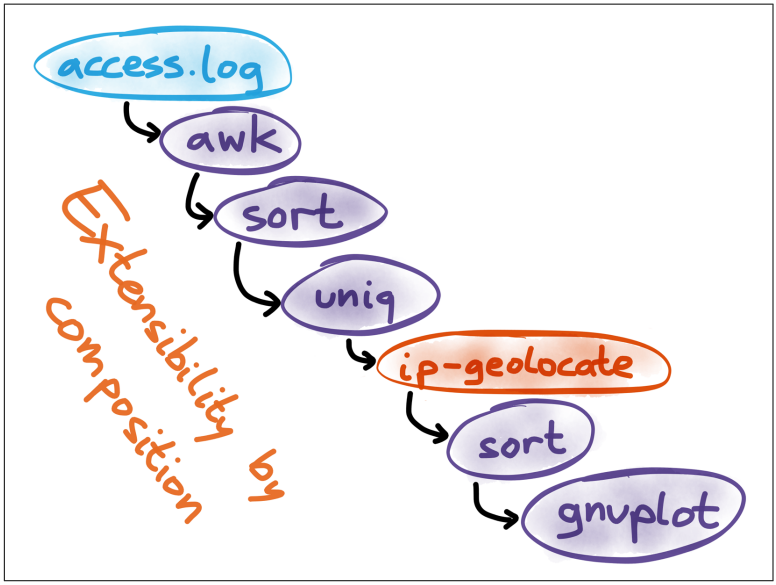


Figure 4-8. You can write your own tool that reads `stdin` and writes `stdout`, and it will work just fine with tools provided by the operating system.

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?

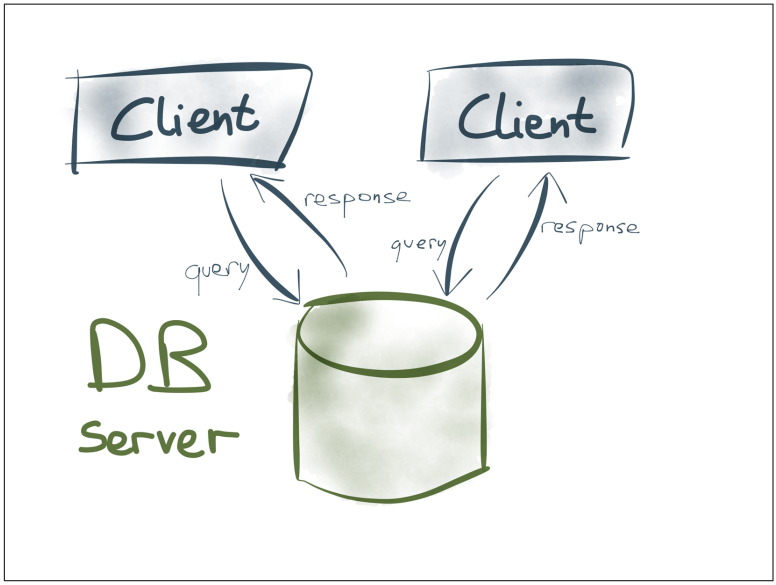


Figure 4-9. In database systems, servers and clients serve two very different roles.

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).

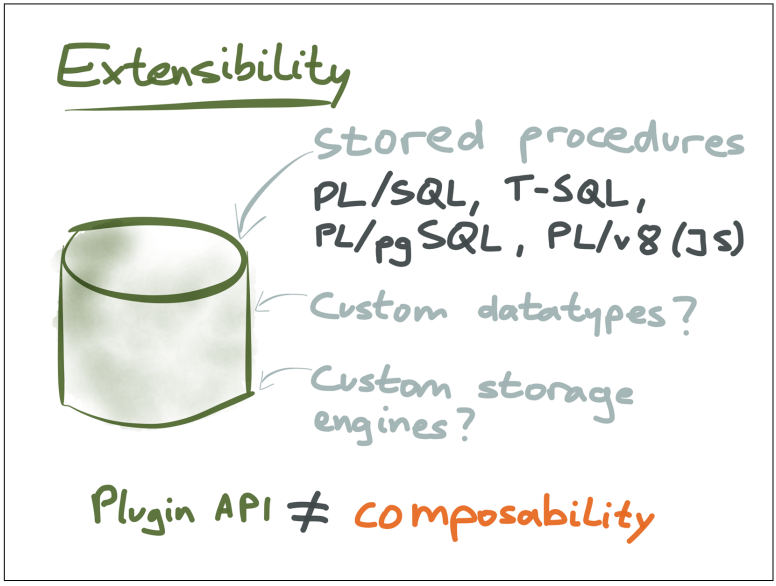


Figure 4-10. Databases have various extension points, but they generally don't have the same modularity and composability as Unix.

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.

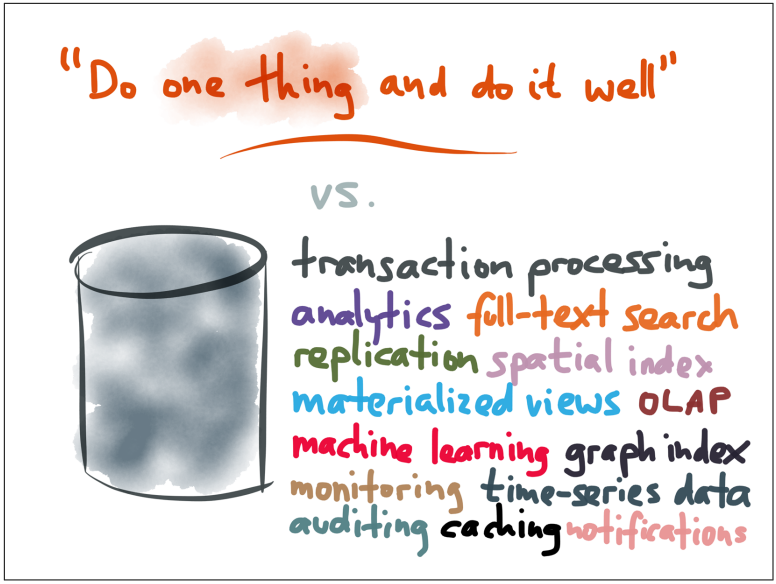


Figure 4-11. A general-purpose database with many features is convenient but philosophically very different from Unix.

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.

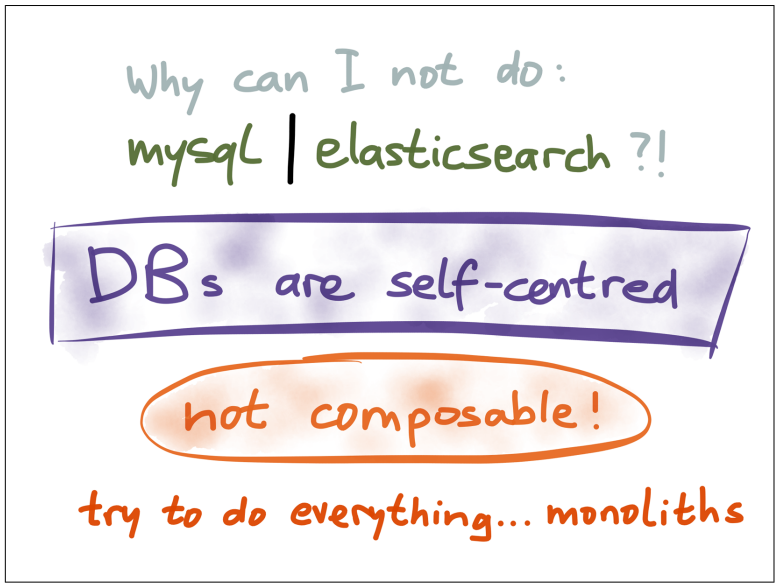


Figure 4-13. Sadly, most databases are not designed with composability in mind.

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.

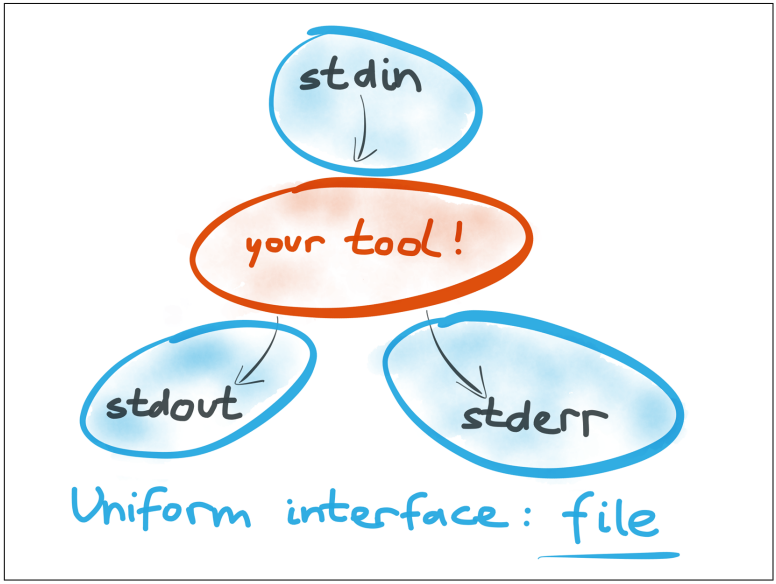


Figure 4-14. On Unix, `stdin`, `stdout`, and `stderr` are all the same kind of thing: a file descriptor (i.e,. a stream of bytes). This makes them compatible.

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.

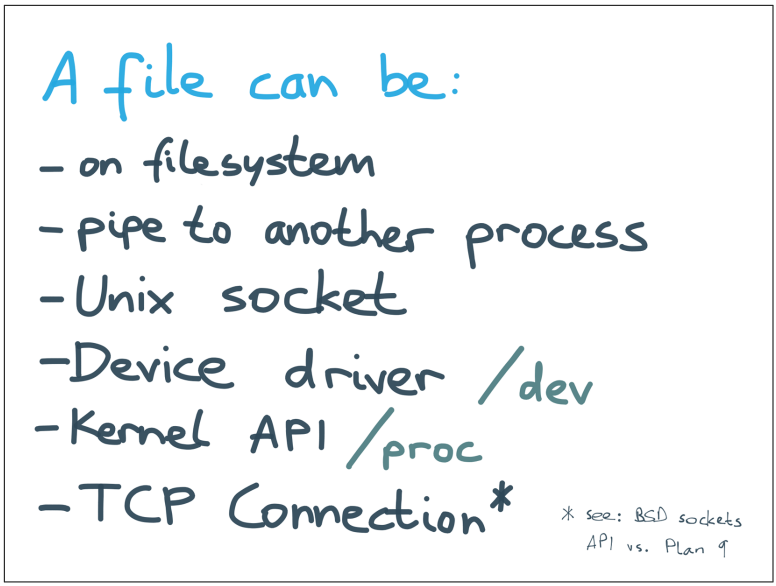


Figure 4-15. The file abstraction can be used to represent many different hardware and software concepts.

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.

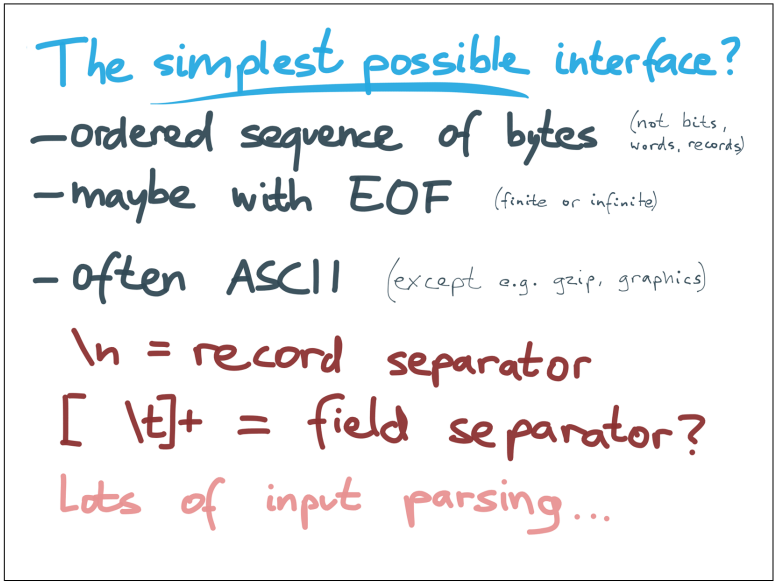


Figure 4-16. A file is just a stream of bytes, and most programs need to parse that stream before they can do anything useful with it.

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).

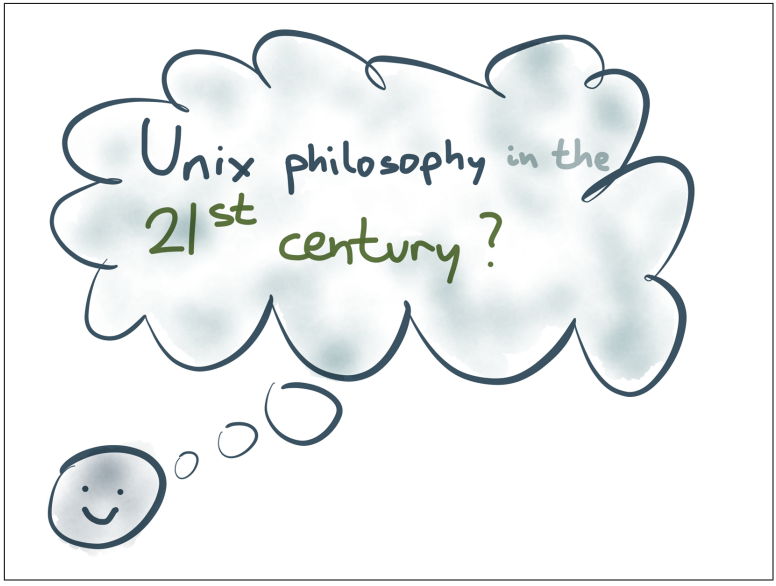


Figure 4-17. Can we improve contemporary data systems by borrowing the best ideas from Unix but avoiding its mistakes?

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).

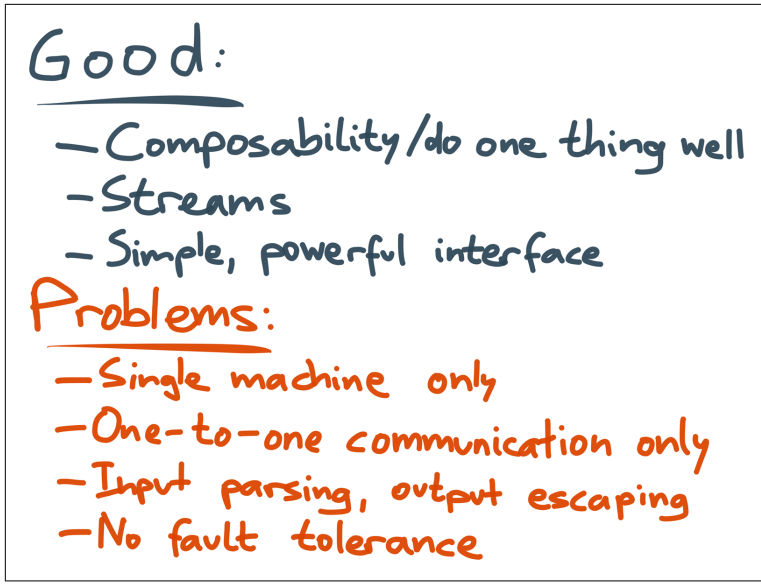


Figure 4-18. Pros and cons of Unix pipes.

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.

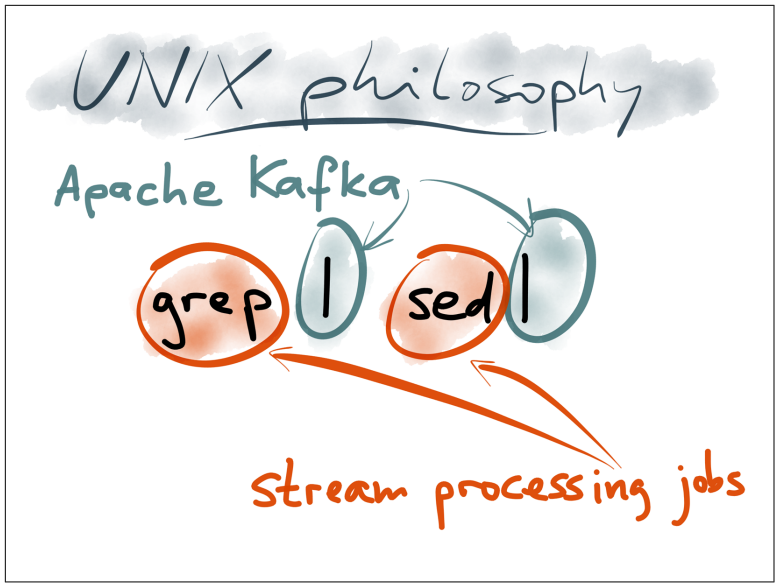


Figure 4-19. The data flow between stream processing jobs, using Kafka for message transport, resembles a pipeline of Unix tools.

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.

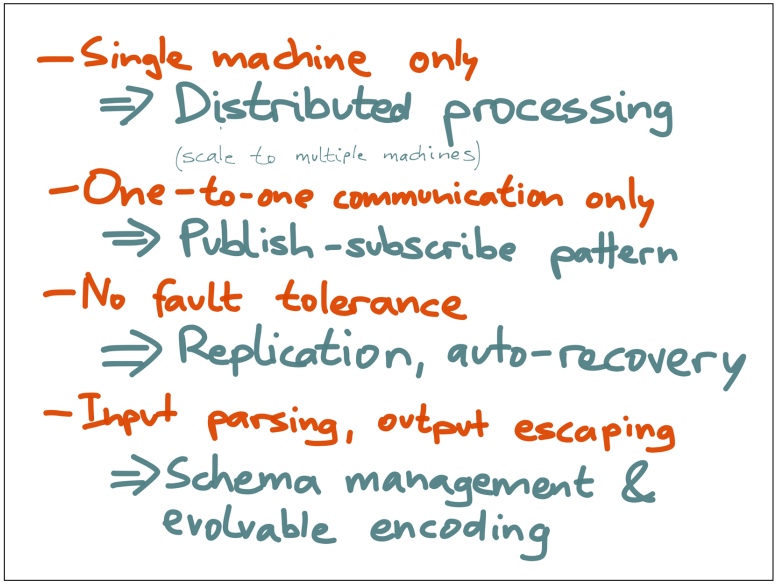


Figure 4-20. How Kafka addresses the problems with Unix pipes.

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.

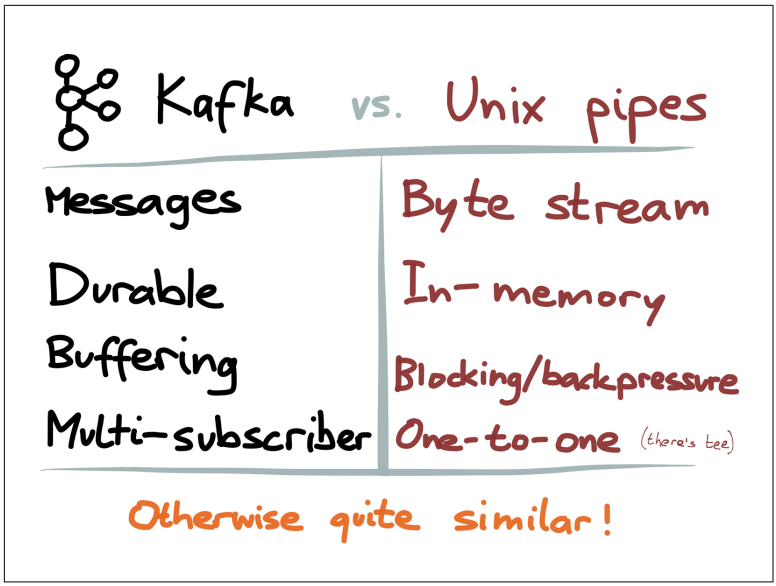


Figure 4-21. Side-by-side comparison of Apache Kafka and Unix pipes.

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.

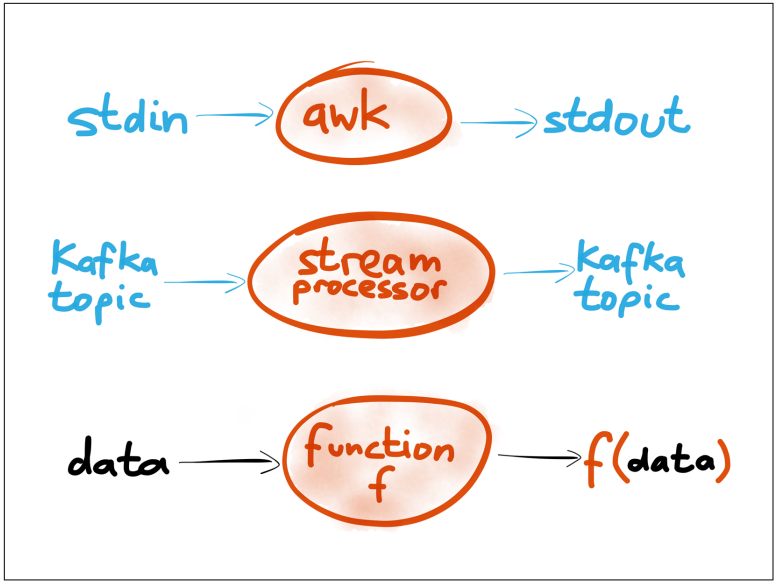


Figure 4-22. Unix tools, stream processors and functional programming share a common trait: inputs are immutable, processing has no global side-effects, and the output is explicit.

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.

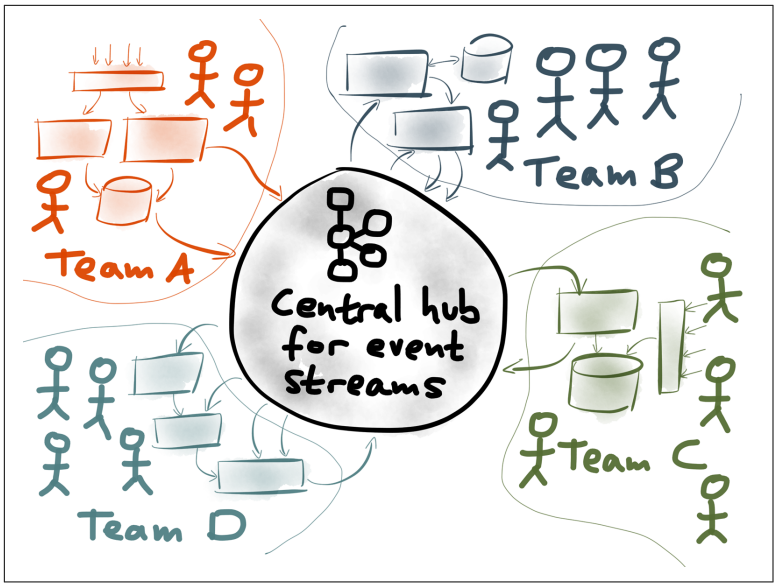


Figure 4-23. Loosely coupled stream processors are good for organizational scalability: Kafka topics can transport data from one team to another, and each team can maintain its own stream processing jobs.

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).

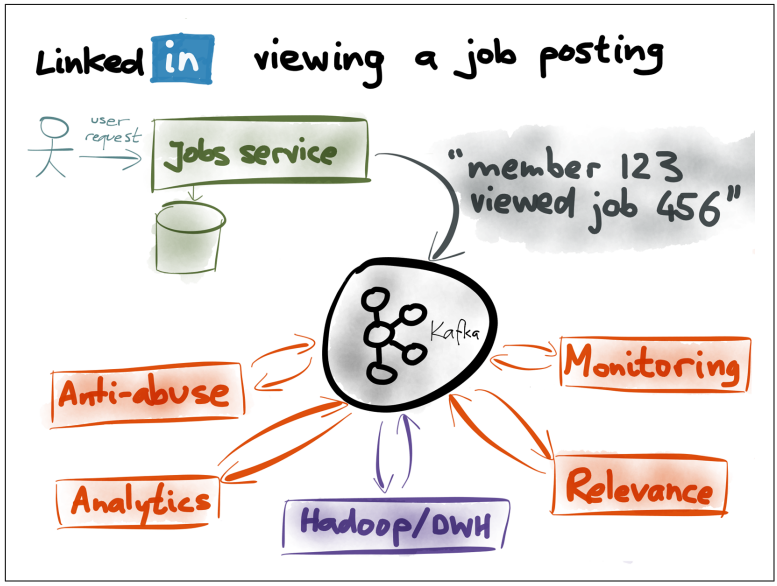


Figure 4-24. What happens when someone views a job posting on LinkedIn?

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.