CHAPTER 12
The Future of Data Systems

> *If a thing be ordained to another as to its end, its last end cannot consist in the preservation of its being. Hence a captain does not intend as a last end, the preservation of the ship entrusted to him, since a ship is ordained to something else as its end, viz. to navigation.*
>
> *(Often quoted as: If the highest aim of a captain was the preserve his ship, he would keep it in port forever.)*
>
> —St. Thomas Aquinas, *Summa Theologica* (1265–1274)

So far, this book has been mostly about describing things as they *are* at present. In this final chapter, we will shift our perspective toward the future and discuss how things *should be*: I will propose some ideas and approaches that, I believe, may fundamentally improve the ways we design and build applications.

Opinions and speculation about the future are of course subjective, and so I will use the first person in this chapter when writing about my personal opinions. You are welcome to disagree with them and form your own opinions, but I hope that the ideas in this chapter will at least be a starting point for a productive discussion and bring some clarity to concepts that are often confused.

The goal of this book was outlined in Chapter 1: to explore how to create applications and systems that are *reliable*, *scalable*, and *maintainable*. These themes have run through all of the chapters: for example, we discussed many fault-tolerance algorithms that help improve reliability, partitioning to improve scalability, and mechanisms for evolution and abstraction that improve maintainability. In this chapter we will bring all of these ideas together, and build on them to envisage the future. Our goal is to discover how to design applications that are better than the ones of today—robust, correct, evolvable, and ultimately beneficial to humanity.

Data Integration

A recurring theme in this book has been that for any given problem, there are several solutions, all of which have different pros, cons, and trade-offs. For example, when discussing storage engines in Chapter 3, we saw log-structured storage, B-trees, and column-oriented storage. When discussing replication in Chapter 5, we saw single-leader, multi-leader, and leaderless approaches.

If you have a problem such as "I want to store some data and look it up again later," there is no one right solution, but many different approaches that are each appropriate in different circumstances. A software implementation typically has to pick one particular approach. It's hard enough to get one code path robust and performing well—trying to do everything in one piece of software almost guarantees that the implementation will be poor.

Thus, the most appropriate choice of software tool also depends on the circumstances. Every piece of software, even a so-called "general-purpose" database, is designed for a particular usage pattern.

Faced with this profusion of alternatives, the first challenge is then to figure out the mapping between the software products and the circumstances in which they are a good fit. Vendors are understandably reluctant to tell you about the kinds of workloads for which their software is poorly suited, but hopefully the previous chapters have equipped you with some questions to ask in order to read between the lines and better understand the trade-offs.

However, even if you perfectly understand the mapping between tools and circumstances for their use, there is another challenge: in complex applications, data is often used in several different ways. There is unlikely to be one piece of software that is suitable for *all* the different circumstances in which the data is used, so you inevitably end up having to cobble together several different pieces of software in order to provide your application's functionality.

Combining Specialized Tools by Deriving Data

For example, it is common to need to integrate an OLTP database with a full-text search index in order to handle queries for arbitrary keywords. Although some databases (such as PostgreSQL) include a full-text indexing feature, which can be sufficient for simple applications [1], more sophisticated search facilities require specialist information retrieval tools. Conversely, search indexes are generally not very suitable as a durable system of record, and so many applications need to combine two different tools in order to satisfy all of the requirements.

We touched on the issue of integrating data systems in "Keeping Systems in Sync" on page 452. As the number of different representations of the data increases, the inte-

gration problem becomes harder. Besides the database and the search index, perhaps you need to keep copies of the data in analytics systems (data warehouses, or batch and stream processing systems); maintain caches or denormalized versions of objects that were derived from the original data; pass the data through machine learning, classification, ranking, or recommendation systems; or send notifications based on changes to the data.

Surprisingly often I see software engineers make statements like, "In my experience, 99% of people only need X" or "…don't need X" (for various values of X). I think that such statements say more about the experience of the speaker than about the actual usefulness of a technology. The range of different things you might want to do with data is dizzyingly wide. What one person considers to be an obscure and pointless feature may well be a central requirement for someone else. The need for data integration often only becomes apparent if you zoom out and consider the dataflows across an entire organization.

Reasoning about dataflows

When copies of the same data need to be maintained in several storage systems in order to satisfy different access patterns, you need to be very clear about the inputs and outputs: where is data written first, and which representations are derived from which sources? How do you get data into all the right places, in the right formats?

For example, you might arrange for data to first be written to a system of record database, capturing the changes made to that database (see "Change Data Capture" on page 454) and then applying the changes to the search index in the same order. If change data capture (CDC) is the only way of updating the index, you can be confident that the index is entirely derived from the system of record, and therefore consistent with it (barring bugs in the software). Writing to the database is the only way of supplying new input into this system.

Allowing the application to directly write to both the search index and the database introduces the problem shown in Figure 11-4, in which two clients concurrently send conflicting writes, and the two storage systems process them in a different order. In this case, neither the database nor the search index is "in charge" of determining the order of writes, and so they may make contradictory decisions and become permanently inconsistent with each other.

If it is possible for you to funnel all user input through a single system that decides on an ordering for all writes, it becomes much easier to derive other representations of the data by processing the writes in the same order. This is an application of the state machine replication approach that we saw in "Total Order Broadcast" on page 348. Whether you use change data capture or an event sourcing log is less important than simply the principle of deciding on a total order.

Updating a derived data system based on an event log can often be made deterministic and idempotent (see "Idempotence" on page 478), making it quite easy to recover from faults.

Derived data versus distributed transactions

The classic approach for keeping different data systems consistent with each other involves distributed transactions, as discussed in "Atomic Commit and Two-Phase Commit (2PC)" on page 354. How does the approach of using derived data systems fare in comparison to distributed transactions?

At an abstract level, they achieve a similar goal by different means. Distributed transactions decide on an ordering of writes by using locks for mutual exclusion (see "Two-Phase Locking (2PL)" on page 257), while CDC and event sourcing use a log for ordering. Distributed transactions use atomic commit to ensure that changes take effect exactly once, while log-based systems are often based on deterministic retry and idempotence.

The biggest difference is that transaction systems usually provide linearizability (see "Linearizability" on page 324), which implies useful guarantees such as reading your own writes (see "Reading Your Own Writes" on page 162). On the other hand, derived data systems are often updated asynchronously, and so they do not by default offer the same timing guarantees.

Within limited environments that are willing to pay the cost of distributed transactions, they have been used successfully. However, I think that XA has poor fault tolerance and performance characteristics (see "Distributed Transactions in Practice" on page 360), which severely limit its usefulness. I believe that it might be possible to create a better protocol for distributed transactions, but getting such a protocol widely adopted and integrated with existing tools would be challenging, and unlikely to happen soon.

In the absence of widespread support for a good distributed transaction protocol, I believe that log-based derived data is the most promising approach for integrating different data systems. However, guarantees such as reading your own writes are useful, and I don't think that it is productive to tell everyone "eventual consistency is inevitable—suck it up and learn to deal with it" (at least not without good guidance on *how* to deal with it).

In "Aiming for Correctness" on page 515 we will discuss some approaches for implementing stronger guarantees on top of asynchronously derived systems, and work toward a middle ground between distributed transactions and asynchronous log-based systems.

The limits of total ordering

With systems that are small enough, constructing a totally ordered event log is entirely feasible (as demonstrated by the popularity of databases with single-leader replication, which construct precisely such a log). However, as systems are scaled toward bigger and more complex workloads, limitations begin to emerge:

- In most cases, constructing a totally ordered log requires all events to pass through a *single leader node* that decides on the ordering. If the throughput of events is greater than a single machine can handle, you need to partition it across multiple machines (see "Partitioned Logs" on page 446). The order of events in two different partitions is then ambiguous.
- If the servers are spread across multiple *geographically distributed* datacenters, for example in order to tolerate an entire datacenter going offline, you typically have a separate leader in each datacenter, because network delays make synchronous cross-datacenter coordination inefficient (see "Multi-Leader Replication" on page 168). This implies an undefined ordering of events that originate in two different datacenters.
- When applications are deployed as *microservices* (see "Dataflow Through Services: REST and RPC" on page 131), a common design choice is to deploy each service and its durable state as an independent unit, with no durable state shared between services. When two events originate in different services, there is no defined order for those events.
- Some applications maintain client-side state that is updated immediately on user input (without waiting for confirmation from a server), and even continue to work offline (see "Clients with offline operation" on page 170). With such applications, clients and servers are very likely to see events in different orders.

In formal terms, deciding on a total order of events is known as *total order broadcast*, which is equivalent to consensus (see "Consensus algorithms and total order broadcast" on page 366). Most consensus algorithms are designed for situations in which the throughput of a single node is sufficient to process the entire stream of events, and these algorithms do not provide a mechanism for multiple nodes to share the work of ordering the events. It is still an open research problem to design consensus algorithms that can scale beyond the throughput of a single node and that work well in a geographically distributed setting.

Ordering events to capture causality

In cases where there is no causal link between events, the lack of a total order is not a big problem, since concurrent events can be ordered arbitrarily. Some other cases are easy to handle: for example, when there are multiple updates of the same object, they can be totally ordered by routing all updates for a particular object ID to the same log

partition. However, causal dependencies sometimes arise in more subtle ways (see also "Ordering and Causality" on page 339).

For example, consider a social networking service, and two users who were in a relationship but have just broken up. One of the users removes the other as a friend, and then sends a message to their remaining friends complaining about their ex-partner. The user's intention is that their ex-partner should not see the rude message, since the message was sent after the friend status was revoked.

However, in a system that stores friendship status in one place and messages in another place, that ordering dependency between the *unfriend* event and the *message-send* event may be lost. If the causal dependency is not captured, a service that sends notifications about new messages may process the *message-send* event before the *unfriend* event, and thus incorrectly send a notification to the ex-partner.

In this example, the notifications are effectively a join between the messages and the friend list, making it related to the timing issues of joins that we discussed previously (see "Time-dependence of joins" on page 475). Unfortunately, there does not seem to be a simple answer to this problem [2, 3]. Starting points include:

- Logical timestamps can provide total ordering without coordination (see "Sequence Number Ordering" on page 343), so they may help in cases where total order broadcast is not feasible. However, they still require recipients to handle events that are delivered out of order, and they require additional metadata to be passed around.
- If you can log an event to record the state of the system that the user saw before making a decision, and give that event a unique identifier, then any later events can reference that event identifier in order to record the causal dependency [4]. We will return to this idea in "Reads are events too" on page 513.
- Conflict resolution algorithms (see "Automatic Conflict Resolution" on page 174) help with processing events that are delivered in an unexpected order. They are useful for maintaining state, but they do not help if actions have external side effects (such as sending a notification to a user).

Perhaps, over time, patterns for application development will emerge that allow causal dependencies to be captured efficiently, and derived state to be maintained correctly, without forcing all events to go through the bottleneck of total order broadcast.

Batch and Stream Processing

I would say that the goal of data integration is to make sure that data ends up in the right form in all the right places. Doing so requires consuming inputs, transforming, joining, filtering, aggregating, training models, evaluating, and eventually writing to

the appropriate outputs. Batch and stream processors are the tools for achieving this goal.

The outputs of batch and stream processes are derived datasets such as search indexes, materialized views, recommendations to show to users, aggregate metrics, and so on (see "The Output of Batch Workflows" on page 411 and "Uses of Stream Processing" on page 465).

As we saw in Chapter 10 and Chapter 11, batch and stream processing have a lot of principles in common, and the main fundamental difference is that stream processors operate on unbounded datasets whereas batch process inputs are of a known, finite size. There are also many detailed differences in the ways the processing engines are implemented, but these distinctions are beginning to blur.

Spark performs stream processing on top of a batch processing engine by breaking the stream into *microbatches*, whereas Apache Flink performs batch processing on top of a stream processing engine [5]. In principle, one type of processing can be emulated on top of the other, although the performance characteristics vary: for example, microbatching may perform poorly on hopping or sliding windows [6].

Maintaining derived state

Batch processing has a quite strong functional flavor (even if the code is not written in a functional programming language): it encourages deterministic, pure functions whose output depends only on the input and which have no side effects other than the explicit outputs, treating inputs as immutable and outputs as append-only. Stream processing is similar, but it extends operators to allow managed, fault-tolerant state (see "Rebuilding state after a failure" on page 478).

The principle of deterministic functions with well-defined inputs and outputs is not only good for fault tolerance (see "Idempotence" on page 478), but also simplifies reasoning about the dataflows in an organization [7]. No matter whether the derived data is a search index, a statistical model, or a cache, it is helpful to think in terms of data pipelines that derive one thing from another, pushing state changes in one system through functional application code and applying the effects to derived systems.

In principle, derived data systems could be maintained synchronously, just like a relational database updates secondary indexes synchronously within the same transaction as writes to the table being indexed. However, asynchrony is what makes systems based on event logs robust: it allows a fault in one part of the system to be contained locally, whereas distributed transactions abort if any one participant fails, so they tend to amplify failures by spreading them to the rest of the system (see "Limitations of distributed transactions" on page 363).

We saw in "Partitioning and Secondary Indexes" on page 206 that secondary indexes often cross partition boundaries. A partitioned system with secondary indexes either

needs to send writes to multiple partitions (if the index is term-partitioned) or send reads to all partitions (if the index is document-partitioned). Such cross-partition communication is also most reliable and scalable if the index is maintained asynchronously [8] (see also "Multi-partition data processing" on page 514).

Reprocessing data for application evolution

When maintaining derived data, batch and stream processing are both useful. Stream processing allows changes in the input to be reflected in derived views with low delay, whereas batch processing allows large amounts of accumulated historical data to be reprocessed in order to derive new views onto an existing dataset.

In particular, reprocessing existing data provides a good mechanism for maintaining a system, evolving it to support new features and changed requirements (see Chapter 4). Without reprocessing, schema evolution is limited to simple changes like adding a new optional field to a record, or adding a new type of record. This is the case both in a schema-on-write and in a schema-on-read context (see "Schema flexibility in the document model" on page 39). On the other hand, with reprocessing it is possible to restructure a dataset into a completely different model in order to better serve new requirements.

Schema Migrations on Railways

Large-scale "schema migrations" occur in noncomputer systems as well. For example, in the early days of railway building in 19th-century England there were various competing standards for the gauge (the distance between the two rails). Trains built for one gauge couldn't run on tracks of another gauge, which restricted the possible interconnections in the train network [9].

After a single standard gauge was finally decided upon in 1846, tracks with other gauges had to be converted—but how do you do this without shutting down the train line for months or years? The solution is to first convert the track to *dual gauge* or *mixed gauge* by adding a third rail. This conversion can be done gradually, and when it is done, trains of both gauges can run on the line, using two of the three rails. Eventually, once all trains have been converted to the standard gauge, the rail providing the nonstandard gauge can be removed.

"Reprocessing" the existing tracks in this way, and allowing the old and new versions to exist side by side, makes it possible to change the gauge gradually over the course of years. Nevertheless, it is an expensive undertaking, which is why nonstandard gauges still exist today. For example, the BART system in the San Francisco Bay Area uses a different gauge from the majority of the US.

Derived views allow *gradual* evolution. If you want to restructure a dataset, you do not need to perform the migration as a sudden switch. Instead, you can maintain the old schema and the new schema side by side as two independently derived views onto the same underlying data. You can then start shifting a small number of users to the new view in order to test its performance and find any bugs, while most users continue to be routed to the old view. Gradually, you can increase the proportion of users accessing the new view, and eventually you can drop the old view [10].

The beauty of such a gradual migration is that every stage of the process is easily reversible if something goes wrong: you always have a working system to go back to. By reducing the risk of irreversible damage, you can be more confident about going ahead, and thus move faster to improve your system [11].

The lambda architecture

If batch processing is used to reprocess historical data, and stream processing is used to process recent updates, then how do you combine the two? The *lambda architecture* [12] is a proposal in this area that has gained a lot of attention.

The core idea of the lambda architecture is that incoming data should be recorded by appending immutable events to an always-growing dataset, similarly to event sourcing (see "Event Sourcing" on page 457). From these events, read-optimized views are derived. The lambda architecture proposes running two different systems in parallel: a batch processing system such as Hadoop MapReduce, and a separate stream-processing system such as Storm.

In the lambda approach, the stream processor consumes the events and quickly produces an approximate update to the view; the batch processor later consumes the *same* set of events and produces a corrected version of the derived view. The reasoning behind this design is that batch processing is simpler and thus less prone to bugs, while stream processors are thought to be less reliable and harder to make fault-tolerant (see "Fault Tolerance" on page 476). Moreover, the stream process can use fast approximate algorithms while the batch process uses slower exact algorithms.

The lambda architecture was an influential idea that shaped the design of data systems for the better, particularly by popularizing the principle of deriving views onto streams of immutable events and reprocessing events when needed. However, I also think that it has a number of practical problems:

- Having to maintain the same logic to run both in a batch and in a stream processing framework is significant additional effort. Although libraries such as Summingbird [13] provide an abstraction for computations that can be run in either a batch or a streaming context, the operational complexity of debugging, tuning, and maintaining two different systems remains [14].

- Since the stream pipeline and the batch pipeline produce separate outputs, they need to be merged in order to respond to user requests. This merge is fairly easy if the computation is a simple aggregation over a tumbling window, but it becomes significantly harder if the view is derived using more complex operations such as joins and sessionization, or if the output is not a time series.
- Although it is great to have the ability to reprocess the entire historical dataset, doing so frequently is expensive on large datasets. Thus, the batch pipeline often needs to be set up to process incremental batches (e.g., an hour's worth of data at the end of every hour) rather than reprocessing everything. This raises the problems discussed in "Reasoning About Time" on page 468, such as handling stragglers and handling windows that cross boundaries between batches. Incrementalizing a batch computation adds complexity, making it more akin to the streaming layer, which runs counter to the goal of keeping the batch layer as simple as possible.

Unifying batch and stream processing

More recent work has enabled the benefits of the lambda architecture to be enjoyed without its downsides, by allowing both batch computations (reprocessing historical data) and stream computations (processing events as they arrive) to be implemented in the same system [15].

Unifying batch and stream processing in one system requires the following features, which are becoming increasingly widely available:

- The ability to replay historical events through the same processing engine that handles the stream of recent events. For example, log-based message brokers have the ability to replay messages (see "Replaying old messages" on page 451), and some stream processors can read input from a distributed filesystem like HDFS.
- Exactly-once semantics for stream processors—that is, ensuring that the output is the same as if no faults had occurred, even if faults did in fact occur (see "Fault Tolerance" on page 476). Like with batch processing, this requires discarding the partial output of any failed tasks.
- Tools for windowing by event time, not by processing time, since processing time is meaningless when reprocessing historical events (see "Reasoning About Time" on page 468). For example, Apache Beam provides an API for expressing such computations, which can then be run using Apache Flink or Google Cloud Dataflow.

Unbundling Databases

At a most abstract level, databases, Hadoop, and operating systems all perform the same functions: they store some data, and they allow you to process and query that data [16]. A database stores data in records of some data model (rows in tables, documents, vertices in a graph, etc.) while an operating system's filesystem stores data in files—but at their core, both are "information management" systems [17]. As we saw in Chapter 10, the Hadoop ecosystem is somewhat like a distributed version of Unix.

Of course, there are many practical differences. For example, many filesystems do not cope very well with a directory containing 10 million small files, whereas a database containing 10 million small records is completely normal and unremarkable. Nevertheless, the similarities and differences between operating systems and databases are worth exploring.

Unix and relational databases have approached the information management problem with very different philosophies. Unix viewed its purpose as presenting programmers with a logical but fairly low-level hardware abstraction, whereas relational databases wanted to give application programmers a high-level abstraction that would hide the complexities of data structures on disk, concurrency, crash recovery, and so on. Unix developed pipes and files that are just sequences of bytes, whereas databases developed SQL and transactions.

Which approach is better? Of course, it depends what you want. Unix is "simpler" in the sense that it is a fairly thin wrapper around hardware resources; relational databases are "simpler" in the sense that a short declarative query can draw on a lot of powerful infrastructure (query optimization, indexes, join methods, concurrency control, replication, etc.) without the author of the query needing to understand the implementation details.

The tension between these philosophies has lasted for decades (both Unix and the relational model emerged in the early 1970s) and still isn't resolved. For example, I would interpret the NoSQL movement as wanting to apply a Unix-esque approach of low-level abstractions to the domain of distributed OLTP data storage.

In this section I will attempt to reconcile the two philosophies, in the hope that we can combine the best of both worlds.

Composing Data Storage Technologies

Over the course of this book we have discussed various features provided by databases and how they work, including:

- Secondary indexes, which allow you to efficiently search for records based on the value of a field (see "Other Indexing Structures" on page 85)

- Materialized views, which are a kind of precomputed cache of query results (see "Aggregation: Data Cubes and Materialized Views" on page 101)
- Replication logs, which keep copies of the data on other nodes up to date (see "Implementation of Replication Logs" on page 158)
- Full-text search indexes, which allow keyword search in text (see "Full-text search and fuzzy indexes" on page 88) and which are built into some relational databases [1]

In Chapters 10 and 11, similar themes emerged. We talked about building full-text search indexes (see "The Output of Batch Workflows" on page 411), about materialized view maintenance (see "Maintaining materialized views" on page 467), and about replicating changes from a database to derived data systems (see "Change Data Capture" on page 454).

It seems that there are parallels between the features that are built into databases and the derived data systems that people are building with batch and stream processors.

Creating an index

Think about what happens when you run `CREATE INDEX` to create a new index in a relational database. The database has to scan over a consistent snapshot of a table, pick out all of the field values being indexed, sort them, and write out the index. Then it must process the backlog of writes that have been made since the consistent snapshot was taken (assuming the table was not locked while creating the index, so writes could continue). Once that is done, the database must continue to keep the index up to date whenever a transaction writes to the table.

This process is remarkably similar to setting up a new follower replica (see "Setting Up New Followers" on page 155), and also very similar to bootstrapping change data capture in a streaming system (see "Initial snapshot" on page 455).

Whenever you run `CREATE INDEX`, the database essentially reprocesses the existing dataset (as discussed in "Reprocessing data for application evolution" on page 496) and derives the index as a new view onto the existing data. The existing data may be a snapshot of the state rather than a log of all changes that ever happened, but the two are closely related (see "State, Streams, and Immutability" on page 459).

The meta-database of everything

In this light, I think that the dataflow across an entire organization starts looking like one huge database [7]. Whenever a batch, stream, or ETL process transports data from one place and form to another place and form, it is acting like the database subsystem that keeps indexes or materialized views up to date.

Viewed like this, batch and stream processors are like elaborate implementations of triggers, stored procedures, and materialized view maintenance routines. The derived data systems they maintain are like different index types. For example, a relational database may support B-tree indexes, hash indexes, spatial indexes (see "Multi-column indexes" on page 87), and other types of indexes. In the emerging architecture of derived data systems, instead of implementing those facilities as features of a single integrated database product, they are provided by various different pieces of software, running on different machines, administered by different teams.

Where will these developments take us in the future? If we start from the premise that there is no single data model or storage format that is suitable for all access patterns, I speculate that there are two avenues by which different storage and processing tools can nevertheless be composed into a cohesive system:

Federated databases: unifying reads
: It is possible to provide a unified query interface to a wide variety of underlying storage engines and processing methods—an approach known as a *federated database* or *polystore* [18, 19]. For example, PostgreSQL's *foreign data wrapper* feature fits this pattern [20]. Applications that need a specialized data model or query interface can still access the underlying storage engines directly, while users who want to combine data from disparate places can do so easily through the federated interface.

  A federated query interface follows the relational tradition of a single integrated system with a high-level query language and elegant semantics, but a complicated implementation.

Unbundled databases: unifying writes
: While federation addresses read-only querying across several different systems, it does not have a good answer to synchronizing writes across those systems. We said that within a single database, creating a consistent index is a built-in feature. When we compose several storage systems, we similarly need to ensure that all data changes end up in all the right places, even in the face of faults. Making it easier to reliably plug together storage systems (e.g., through change data capture and event logs) is like *unbundling* a database's index-maintenance features in a way that can synchronize writes across disparate technologies [7, 21].

  The unbundled approach follows the Unix tradition of small tools that do one thing well [22], that communicate through a uniform low-level API (pipes), and that can be composed using a higher-level language (the shell) [16].

Making unbundling work

Federation and unbundling are two sides of the same coin: composing a reliable, scalable, and maintainable system out of diverse components. Federated read-only

querying requires mapping one data model into another, which takes some thought but is ultimately quite a manageable problem. I think that keeping the writes to several storage systems in sync is the harder engineering problem, and so I will focus on it.

The traditional approach to synchronizing writes requires distributed transactions across heterogeneous storage systems [18], which I think is the wrong solution (see "Derived data versus distributed transactions" on page 492). Transactions within a single storage or stream processing system are feasible, but when data crosses the boundary between different technologies, I believe that an asynchronous event log with idempotent writes is a much more robust and practical approach.

For example, distributed transactions are used within some stream processors to achieve exactly-once semantics (see "Atomic commit revisited" on page 477), and this can work quite well. However, when a transaction would need to involve systems written by different groups of people (e.g., when data is written from a stream processor to a distributed key-value store or search index), the lack of a standardized transaction protocol makes integration much harder. An ordered log of events with idempotent consumers (see "Idempotence" on page 478) is a much simpler abstraction, and thus much more feasible to implement across heterogeneous systems [7].

The big advantage of log-based integration is *loose coupling* between the various components, which manifests itself in two ways:

1. At a system level, asynchronous event streams make the system as a whole more robust to outages or performance degradation of individual components. If a consumer runs slow or fails, the event log can buffer messages (see "Disk space usage" on page 450), allowing the producer and any other consumers to continue running unaffected. The faulty consumer can catch up when it is fixed, so it doesn't miss any data, and the fault is contained. By contrast, the synchronous interaction of distributed transactions tends to escalate local faults into large-scale failures (see "Limitations of distributed transactions" on page 363).
2. At a human level, unbundling data systems allows different software components and services to be developed, improved, and maintained independently from each other by different teams. Specialization allows each team to focus on doing one thing well, with well-defined interfaces to other teams' systems. Event logs provide an interface that is powerful enough to capture fairly strong consistency properties (due to durability and ordering of events), but also general enough to be applicable to almost any kind of data.

Unbundled versus integrated systems

If unbundling does indeed become the way of the future, it will not replace databases in their current form—they will still be needed as much as ever. Databases are still

required for maintaining state in stream processors, and in order to serve queries for the output of batch and stream processors (see "The Output of Batch Workflows" on page 411 and "Processing Streams" on page 464). Specialized query engines will continue to be important for particular workloads: for example, query engines in MPP data warehouses are optimized for exploratory analytic queries and handle this kind of workload very well (see "Comparing Hadoop to Distributed Databases" on page 414).

The complexity of running several different pieces of infrastructure can be a problem: each piece of software has a learning curve, configuration issues, and operational quirks, and so it is worth deploying as few moving parts as possible. A single integrated software product may also be able to achieve better and more predictable performance on the kinds of workloads for which it is designed, compared to a system consisting of several tools that you have composed with application code [23]. As I said in the Preface, building for scale that you don't need is wasted effort and may lock you into an inflexible design. In effect, it is a form of premature optimization.

The goal of unbundling is not to compete with individual databases on performance for particular workloads; the goal is to allow you to combine several different databases in order to achieve good performance for a much wider range of workloads than is possible with a single piece of software. It's about breadth, not depth—in the same vein as the diversity of storage and processing models that we discussed in "Comparing Hadoop to Distributed Databases" on page 414.

Thus, if there is a single technology that does everything you need, you're most likely best off simply using that product rather than trying to reimplement it yourself from lower-level components. The advantages of unbundling and composition only come into the picture when there is no single piece of software that satisfies all your requirements.

What's missing?

The tools for composing data systems are getting better, but I think one major part is missing: we don't yet have the unbundled-database equivalent of the Unix shell (i.e., a high-level language for composing storage and processing systems in a simple and declarative way).

For example, I would love it if we could simply declare `mysql | elasticsearch`, by analogy to Unix pipes [22], which would be the unbundled equivalent of `CREATE INDEX`: it would take all the documents in a MySQL database and index them in an Elasticsearch cluster. It would then continually capture all the changes made to the database and automatically apply them to the search index, without us having to write custom application code. This kind of integration should be possible with almost any kind of storage or indexing system.

Similarly, it would be great to be able to precompute and update caches more easily. Recall that a materialized view is essentially a precomputed cache, so you could imagine creating a cache by declaratively specifying materialized views for complex queries, including recursive queries on graphs (see "Graph-Like Data Models" on page 49) and application logic. There is interesting early-stage research in this area, such as *differential dataflow* [24, 25], and I hope that these ideas will find their way into production systems.

Designing Applications Around Dataflow

The approach of unbundling databases by composing specialized storage and processing systems with application code is also becoming known as the "database inside-out" approach [26], after the title of a conference talk I gave in 2014 [27]. However, calling it a "new architecture" is too grandiose. I see it more as a design pattern, a starting point for discussion, and we give it a name simply so that we can better talk about it.

These ideas are not mine; they are simply an amalgamation of other people's ideas from which I think we should learn. In particular, there is a lot of overlap with *dataflow* languages such as Oz [28] and Juttle [29], *functional reactive programming* (FRP) languages such as Elm [30, 31], and *logic programming* languages such as Bloom [32]. The term *unbundling* in this context was proposed by Jay Kreps [7].

Even spreadsheets have dataflow programming capabilities that are miles ahead of most mainstream programming languages [33]. In a spreadsheet, you can put a formula in one cell (for example, the sum of cells in another column), and whenever any input to the formula changes, the result of the formula is automatically recalculated. This is exactly what we want at a data system level: when a record in a database changes, we want any index for that record to be automatically updated, and any cached views or aggregations that depend on the record to be automatically refreshed. You should not have to worry about the technical details of how this refresh happens, but be able to simply trust that it works correctly.

Thus, I think that most data systems still have something to learn from the features that VisiCalc already had in 1979 [34]. The difference from spreadsheets is that today's data systems need to be fault-tolerant, scalable, and store data durably. They also need to be able to integrate disparate technologies written by different groups of people over time, and reuse existing libraries and services: it is unrealistic to expect all software to be developed using one particular language, framework, or tool.

In this section I will expand on these ideas and explore some ways of building applications around the ideas of unbundled databases and dataflow.

Application code as a derivation function

When one dataset is derived from another, it goes through some kind of transformation function. For example:

- A secondary index is a kind of derived dataset with a straightforward transformation function: for each row or document in the base table, it picks out the values in the columns or fields being indexed, and sorts by those values (assuming a B-tree or SSTable index, which are sorted by key, as discussed in Chapter 3).
- A full-text search index is created by applying various natural language processing functions such as language detection, word segmentation, stemming or lemmatization, spelling correction, and synonym identification, followed by building a data structure for efficient lookups (such as an inverted index).
- In a machine learning system, we can consider the model as being derived from the training data by applying various feature extraction and statistical analysis functions. When the model is applied to new input data, the output of the model is derived from the input and the model (and hence, indirectly, from the training data).
- A cache often contains an aggregation of data in the form in which it is going to be displayed in a user interface (UI). Populating the cache thus requires knowledge of what fields are referenced in the UI; changes in the UI may require updating the definition of how the cache is populated and rebuilding the cache.

The derivation function for a secondary index is so commonly required that it is built into many databases as a core feature, and you can invoke it by merely saying `CREATE INDEX`. For full-text indexing, basic linguistic features for common languages may be built into a database, but the more sophisticated features often require domain-specific tuning. In machine learning, feature engineering is notoriously application-specific, and often has to incorporate detailed knowledge about the user interaction and deployment of an application [35].

When the function that creates a derived dataset is not a standard cookie-cutter function like creating a secondary index, custom code is required to handle the application-specific aspects. And this custom code is where many databases struggle. Although relational databases commonly support triggers, stored procedures, and user-defined functions, which can be used to execute application code within the database, they have been somewhat of an afterthought in database design (see "Transmitting Event Streams" on page 440).

Separation of application code and state

In theory, databases could be deployment environments for arbitrary application code, like an operating system. However, in practice they have turned out to be

poorly suited for this purpose. They do not fit well with the requirements of modern application development, such as dependency and package management, version control, rolling upgrades, evolvability, monitoring, metrics, calls to network services, and integration with external systems.

On the other hand, deployment and cluster management tools such as Mesos, YARN, Docker, Kubernetes, and others are designed specifically for the purpose of running application code. By focusing on doing one thing well, they are able to do it much better than a database that provides execution of user-defined functions as one of its many features.

I think it makes sense to have some parts of a system that specialize in durable data storage, and other parts that specialize in running application code. The two can interact while still remaining independent.

Most web applications today are deployed as stateless services, in which any user request can be routed to any application server, and the server forgets everything about the request once it has sent the response. This style of deployment is convenient, as servers can be added or removed at will, but the state has to go somewhere: typically, a database. The trend has been to keep stateless application logic separate from state management (databases): not putting application logic in the database and not putting persistent state in the application [36]. As people in the functional programming community like to joke, "We believe in the separation of Church and state" [37].[i]

In this typical web application model, the database acts as a kind of mutable shared variable that can be accessed synchronously over the network. The application can read and update the variable, and the database takes care of making it durable, providing some concurrency control and fault tolerance.

However, in most programming languages you cannot subscribe to changes in a mutable variable—you can only read it periodically. Unlike in a spreadsheet, readers of the variable don't get notified if the value of the variable changes. (You can implement such notifications in your own code—this is known as the *observer pattern*—but most languages do not have this pattern as a built-in feature.)

Databases have inherited this passive approach to mutable data: if you want to find out whether the content of the database has changed, often your only option is to poll (i.e., to repeat your query periodically). Subscribing to changes is only just beginning to emerge as a feature (see "API support for change streams" on page 456).

i. Explaining a joke rarely improves it, but I don't want anyone to feel left out. Here, *Church* is a reference to the mathematician Alonzo Church, who created the lambda calculus, an early form of computation that is the basis for most functional programming languages. The lambda calculus has no mutable state (i.e., no variables that can be overwritten), so one could say that mutable state is separate from Church's work.

Dataflow: Interplay between state changes and application code

Thinking about applications in terms of dataflow implies renegotiating the relationship between application code and state management. Instead of treating a database as a passive variable that is manipulated by the application, we think much more about the interplay and collaboration between state, state changes, and code that processes them. Application code responds to state changes in one place by triggering state changes in another place.

We saw this line of thinking in "Databases and Streams" on page 451, where we discussed treating the log of changes to a database as a stream of events that we can subscribe to. Message-passing systems such as actors (see "Message-Passing Dataflow" on page 136) also have this concept of responding to events. Already in the 1980s, the *tuple spaces* model explored expressing distributed computations in terms of processes that observe state changes and react to them [38, 39].

As discussed, similar things happen inside a database when a trigger fires due to a data change, or when a secondary index is updated to reflect a change in the table being indexed. Unbundling the database means taking this idea and applying it to the creation of derived datasets outside of the primary database: caches, full-text search indexes, machine learning, or analytics systems. We can use stream processing and messaging systems for this purpose.

The important thing to keep in mind is that maintaining derived data is not the same as asynchronous job execution, for which messaging systems are traditionally designed (see "Logs compared to traditional messaging" on page 448):

- When maintaining derived data, the order of state changes is often important (if several views are derived from an event log, they need to process the events in the same order so that they remain consistent with each other). As discussed in "Acknowledgments and redelivery" on page 445, many message brokers do not have this property when redelivering unacknowledged messages. Dual writes are also ruled out (see "Keeping Systems in Sync" on page 452).
- Fault tolerance is key for derived data: losing just a single message causes the derived dataset to go permanently out of sync with its data source. Both message delivery and derived state updates must be reliable. For example, many actor systems by default maintain actor state and messages in memory, so they are lost if the machine running the actor crashes.

Stable message ordering and fault-tolerant message processing are quite stringent demands, but they are much less expensive and more operationally robust than distributed transactions. Modern stream processors can provide these ordering and reliability guarantees at scale, and they allow application code to be run as stream operators.

This application code can do the arbitrary processing that built-in derivation functions in databases generally don't provide. Like Unix tools chained by pipes, stream operators can be composed to build large systems around dataflow. Each operator takes streams of state changes as input, and produces other streams of state changes as output.

Stream processors and services

The currently trendy style of application development involves breaking down functionality into a set of *services* that communicate via synchronous network requests such as REST APIs (see "Dataflow Through Services: REST and RPC" on page 131). The advantage of such a service-oriented architecture over a single monolithic application is primarily organizational scalability through loose coupling: different teams can work on different services, which reduces coordination effort between teams (as long as the services can be deployed and updated independently).

Composing stream operators into dataflow systems has a lot of similar characteristics to the microservices approach [40]. However, the underlying communication mechanism is very different: one-directional, asynchronous message streams rather than synchronous request/response interactions.

Besides the advantages listed in "Message-Passing Dataflow" on page 136, such as better fault tolerance, dataflow systems can also achieve better performance. For example, say a customer is purchasing an item that is priced in one currency but paid for in another currency. In order to perform the currency conversion, you need to know the current exchange rate. This operation could be implemented in two ways [40, 41]:

1. In the microservices approach, the code that processes the purchase would probably query an exchange-rate service or database in order to obtain the current rate for a particular currency.
2. In the dataflow approach, the code that processes purchases would subscribe to a stream of exchange rate updates ahead of time, and record the current rate in a local database whenever it changes. When it comes to processing the purchase, it only needs to query the local database.

The second approach has replaced a synchronous network request to another service with a query to a local database (which may be on the same machine, even in the same process).[ii] Not only is the dataflow approach faster, but it is also more robust to

ii. In the microservices approach, you could avoid the synchronous network request by caching the exchange rate locally in the service that processes the purchase. However, in order to keep that cache fresh, you would need to periodically poll for updated exchange rates, or subscribe to a stream of changes—which is exactly what happens in the dataflow approach.

the failure of another service. The fastest and most reliable network request is no network request at all! Instead of RPC, we now have a stream join between purchase events and exchange rate update events (see "Stream-table join (stream enrichment)" on page 473).

The join is time-dependent: if the purchase events are reprocessed at a later point in time, the exchange rate will have changed. If you want to reconstruct the original output, you will need to obtain the historical exchange rate at the original time of purchase. No matter whether you query a service or subscribe to a stream of exchange rate updates, you will need to handle this time dependence (see "Time-dependence of joins" on page 475).

Subscribing to a stream of changes, rather than querying the current state when needed, brings us closer to a spreadsheet-like model of computation: when some piece of data changes, any derived data that depends on it can swiftly be updated. There are still many open questions, for example around issues like time-dependent joins, but I believe that building applications around dataflow ideas is a very promising direction to go in.

Observing Derived State

At an abstract level, the dataflow systems discussed in the last section give you a process for creating derived datasets (such as search indexes, materialized views, and predictive models) and keeping them up to date. Let's call that process the *write path*: whenever some piece of information is written to the system, it may go through multiple stages of batch and stream processing, and eventually every derived dataset is updated to incorporate the data that was written. Figure 12-1 shows an example of updating a search index.

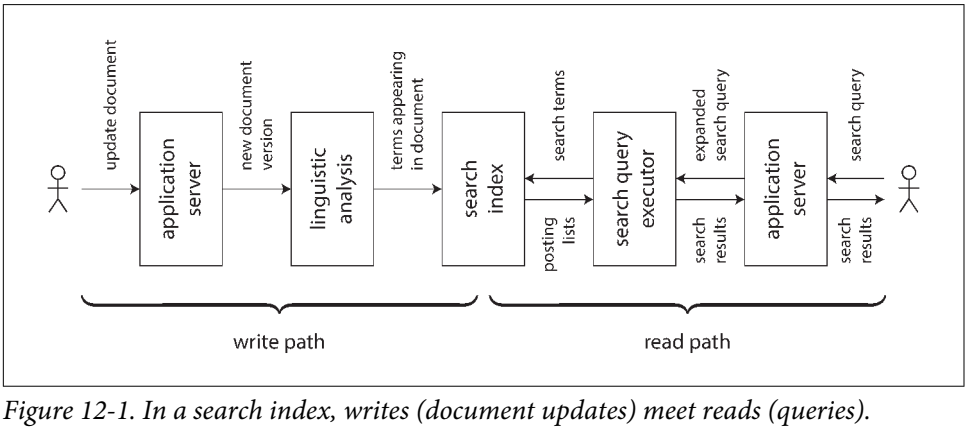

Figure 12-1. In a search index, writes (document updates) meet reads (queries).

But why do you create the derived dataset in the first place? Most likely because you want to query it again at a later time. This is the *read path*: when serving a user

request you read from the derived dataset, perhaps perform some more processing on the results, and construct the response to the user.

Taken together, the write path and the read path encompass the whole journey of the data, from the point where it is collected to the point where it is consumed (probably by another human). The write path is the portion of the journey that is precomputed —i.e., that is done eagerly as soon as the data comes in, regardless of whether anyone has asked to see it. The read path is the portion of the journey that only happens when someone asks for it. If you are familiar with functional programming languages, you might notice that the write path is similar to eager evaluation, and the read path is similar to lazy evaluation.

The derived dataset is the place where the write path and the read path meet, as illustrated in Figure 12-1. It represents a trade-off between the amount of work that needs to be done at write time and the amount that needs to be done at read time.

Materialized views and caching

A full-text search index is a good example: the write path updates the index, and the read path searches the index for keywords. Both reads and writes need to do some work. Writes need to update the index entries for all terms that appear in the document. Reads need to search for each of the words in the query, and apply Boolean logic to find documents that contain *all* of the words in the query (an `AND` operator), or *any* synonym of each of the words (an `OR` operator).

If you didn't have an index, a search query would have to scan over all documents (like `grep`), which would get very expensive if you had a large number of documents. No index means less work on the write path (no index to update), but a lot more work on the read path.

On the other hand, you could imagine precomputing the search results for all possible queries. In that case, you would have less work to do on the read path: no Boolean logic, just find the results for your query and return them. However, the write path would be a lot more expensive: the set of possible search queries that could be asked is infinite, and thus precomputing all possible search results would require infinite time and storage space. That wouldn't work so well.[iii]

Another option would be to precompute the search results for only a fixed set of the most common queries, so that they can be served quickly without having to go to the index. The uncommon queries can still be served from the index. This would generally be called a *cache* of common queries, although we could also call it a materialized

iii. Less facetiously, the set of distinct search queries with nonempty search results is finite, assuming a finite corpus. However, it would be exponential in the number of terms in the corpus, which is still pretty bad news.

view, as it would need to be updated when new documents appear that should be included in the results of one of the common queries.

From this example we can see that an index is not the only possible boundary between the write path and the read path. Caching of common search results is possible, and `grep`-like scanning without the index is also possible on a small number of documents. Viewed like this, the role of caches, indexes, and materialized views is simple: they shift the boundary between the read path and the write path. They allow us to do more work on the write path, by precomputing results, in order to save effort on the read path.

Shifting the boundary between work done on the write path and the read path was in fact the topic of the Twitter example at the beginning of this book, in "Describing Load" on page 11. In that example, we also saw how the boundary between write path and read path might be drawn differently for celebrities compared to ordinary users. After 500 pages we have come full circle!

Stateful, offline-capable clients

I find the idea of a boundary between write and read paths interesting because we can discuss shifting that boundary and explore what that shift means in practical terms. Let's look at the idea in a different context.

The huge popularity of web applications in the last two decades has led us to certain assumptions about application development that are easy to take for granted. In particular, the client/server model—in which clients are largely stateless and servers have the authority over data—is so common that we almost forget that anything else exists. However, technology keeps moving on, and I think it is important to question the status quo from time to time.

Traditionally, web browsers have been stateless clients that can only do useful things when you have an internet connection (just about the only thing you could do offline was to scroll up and down in a page that you had previously loaded while online). However, recent "single-page" JavaScript web apps have gained a lot of stateful capabilities, including client-side user interface interaction and persistent local storage in the web browser. Mobile apps can similarly store a lot of state on the device and don't require a round-trip to the server for most user interactions.

These changing capabilities have led to a renewed interest in *offline-first* applications that do as much as possible using a local database on the same device, without requiring an internet connection, and sync with remote servers in the background when a network connection is available [42]. Since mobile devices often have slow and unreliable cellular internet connections, it's a big advantage for users if their user interface does not have to wait for synchronous network requests, and if apps mostly work offline (see "Clients with offline operation" on page 170).

When we move away from the assumption of stateless clients talking to a central database and toward state that is maintained on end-user devices, a world of new opportunities opens up. In particular, we can think of the on-device state as a *cache of state on the server*. The pixels on the screen are a materialized view onto model objects in the client app; the model objects are a local replica of state in a remote datacenter [27].

Pushing state changes to clients

In a typical web page, if you load the page in a web browser and the data subsequently changes on the server, the browser does not find out about the change until you reload the page. The browser only reads the data at one point in time, assuming that it is static—it does not subscribe to updates from the server. Thus, the state on the device is a stale cache that is not updated unless you explicitly poll for changes. (HTTP-based feed subscription protocols like RSS are really just a basic form of polling.)

More recent protocols have moved beyond the basic request/response pattern of HTTP: server-sent events (the EventSource API) and WebSockets provide communication channels by which a web browser can keep an open TCP connection to a server, and the server can actively push messages to the browser as long as it remains connected. This provides an opportunity for the server to actively inform the end-user client about any changes to the state it has stored locally, reducing the staleness of the client-side state.

In terms of our model of write path and read path, actively pushing state changes all the way to client devices means extending the write path all the way to the end user. When a client is first initialized, it would still need to use a read path to get its initial state, but thereafter it could rely on a stream of state changes sent by the server. The ideas we discussed around stream processing and messaging are not restricted to running only in a datacenter: we can take the ideas further, and extend them all the way to end-user devices [43].

The devices will be offline some of the time, and unable to receive any notifications of state changes from the server during that time. But we already solved that problem: in "Consumer offsets" on page 449 we discussed how a consumer of a log-based message broker can reconnect after failing or becoming disconnected, and ensure that it doesn't miss any messages that arrived while it was disconnected. The same technique works for individual users, where each device is a small subscriber to a small stream of events.

End-to-end event streams

Recent tools for developing stateful clients and user interfaces, such as the Elm language [30] and Facebook's toolchain of React, Flux, and Redux [44], already manage

internal client-side state by subscribing to a stream of events representing user input or responses from a server, structured similarly to event sourcing (see "Event Sourcing" on page 457).

It would be very natural to extend this programming model to also allow a server to push state-change events into this client-side event pipeline. Thus, state changes could flow through an end-to-end write path: from the interaction on one device that triggers a state change, via event logs and through several derived data systems and stream processors, all the way to the user interface of a person observing the state on another device. These state changes could be propagated with fairly low delay—say, under one second end to end.

Some applications, such as instant messaging and online games, already have such a "real-time" architecture (in the sense of interactions with low delay, not in the sense of "Response time guarantees" on page 298). But why don't we build all applications this way?

The challenge is that the assumption of stateless clients and request/response interactions is very deeply ingrained in our databases, libraries, frameworks, and protocols. Many datastores support read and write operations where a request returns one response, but much fewer provide an ability to subscribe to changes—i.e., a request that returns a stream of responses over time (see "API support for change streams" on page 456).

In order to extend the write path all the way to the end user, we would need to fundamentally rethink the way we build many of these systems: moving away from request/response interaction and toward publish/subscribe dataflow [27]. I think that the advantages of more responsive user interfaces and better offline support would make it worth the effort. If you are designing data systems, I hope that you will keep in mind the option of subscribing to changes, not just querying the current state.

Reads are events too

We discussed that when a stream processor writes derived data to a store (database, cache, or index), and when user requests query that store, the store acts as the boundary between the write path and the read path. The store allows random-access read queries to the data that would otherwise require scanning the whole event log.

In many cases, the data storage is separate from the streaming system. But recall that stream processors also need to maintain state to perform aggregations and joins (see "Stream Joins" on page 472). This state is normally hidden inside the stream processor, but some frameworks allow it to also be queried by outside clients [45], turning the stream processor itself into a kind of simple database.

I would like to take that idea further. As discussed so far, the writes to the store go through an event log, while reads are transient network requests that go directly to

the nodes that store the data being queried. This is a reasonable design, but not the only possible one. It is also possible to represent read requests as streams of events, and send both the read events and the write events through a stream processor; the processor responds to read events by emitting the result of the read to an output stream [46].

When both the writes and the reads are represented as events, and routed to the same stream operator in order to be handled, we are in fact performing a stream-table join between the stream of read queries and the database. The read event needs to be sent to the database partition holding the data (see "Request Routing" on page 214), just like batch and stream processors need to copartition inputs on the same key when joining (see "Reduce-Side Joins and Grouping" on page 403).

This correspondence between serving requests and performing joins is quite fundamental [47]. A one-off read request just passes the request through the join operator and then immediately forgets it; a subscribe request is a persistent join with past and future events on the other side of the join.

Recording a log of read events potentially also has benefits with regard to tracking causal dependencies and data provenance across a system: it would allow you to reconstruct what the user saw before they made a particular decision. For example, in an online shop, it is likely that the predicted shipping date and the inventory status shown to a customer affect whether they choose to buy an item [4]. To analyze this connection, you need to record the result of the user's query of the shipping and inventory status.

Writing read events to durable storage thus enables better tracking of causal dependencies (see "Ordering events to capture causality" on page 493), but it incurs additional storage and I/O cost. Optimizing such systems to reduce the overhead is still an open research problem [2]. But if you already log read requests for operational purposes, as a side effect of request processing, it is not such a great change to make the log the source of the requests instead.

Multi-partition data processing

For queries that only touch a single partition, the effort of sending queries through a stream and collecting a stream of responses is perhaps overkill. However, this idea opens the possibility of distributed execution of complex queries that need to combine data from several partitions, taking advantage of the infrastructure for message routing, partitioning, and joining that is already provided by stream processors.

Storm's distributed RPC feature supports this usage pattern (see "Message passing and RPC" on page 468). For example, it has been used to compute the number of people who have seen a URL on Twitter—i.e., the union of the follower sets of everyone who has tweeted that URL [48]. As the set of Twitter users is partitioned, this computation requires combining results from many partitions.

Another example of this pattern occurs in fraud prevention: in order to assess the risk of whether a particular purchase event is fraudulent, you can examine the reputation scores of the user's IP address, email address, billing address, shipping address, and so on. Each of these reputation databases is itself partitioned, and so collecting the scores for a particular purchase event requires a sequence of joins with differently partitioned datasets [49].

The internal query execution graphs of MPP databases have similar characteristics (see "Comparing Hadoop to Distributed Databases" on page 414). If you need to perform this kind of multi-partition join, it is probably simpler to use a database that provides this feature than to implement it using a stream processor. However, treating queries as streams provides an option for implementing large-scale applications that run against the limits of conventional off-the-shelf solutions.

Aiming for Correctness

With stateless services that only read data, it is not a big deal if something goes wrong: you can fix the bug and restart the service, and everything returns to normal. Stateful systems such as databases are not so simple: they are designed to remember things forever (more or less), so if something goes wrong, the effects also potentially last forever—which means they require more careful thought [50].

We want to build applications that are reliable and *correct* (i.e., programs whose semantics are well defined and understood, even in the face of various faults). For approximately four decades, the transaction properties of atomicity, isolation, and durability (Chapter 7) have been the tools of choice for building correct applications. However, those foundations are weaker than they seem: witness for example the confusion of weak isolation levels (see "Weak Isolation Levels" on page 233).

In some areas, transactions are being abandoned entirely and replaced with models that offer better performance and scalability, but much messier semantics (see for example "Leaderless Replication" on page 177). *Consistency* is often talked about, but poorly defined (see "Consistency" on page 224 and Chapter 9). Some people assert that we should "embrace weak consistency" for the sake of better availability, while lacking a clear idea of what that actually means in practice.

For a topic that is so important, our understanding and our engineering methods are surprisingly flaky. For example, it is very difficult to determine whether it is safe to run a particular application at a particular transaction isolation level or replication configuration [51, 52]. Often simple solutions appear to work correctly when concurrency is low and there are no faults, but turn out to have many subtle bugs in more demanding circumstances.

For example, Kyle Kingsbury's Jepsen experiments [53] have highlighted the stark discrepancies between some products' claimed safety guarantees and their actual

behavior in the presence of network problems and crashes. Even if infrastructure products like databases were free from problems, application code would still need to correctly use the features they provide, which is error-prone if the configuration is hard to understand (which is the case with weak isolation levels, quorum configurations, and so on).

If your application can tolerate occasionally corrupting or losing data in unpredictable ways, life is a lot simpler, and you might be able to get away with simply crossing your fingers and hoping for the best. On the other hand, if you need stronger assurances of correctness, then serializability and atomic commit are established approaches, but they come at a cost: they typically only work in a single datacenter (ruling out geographically distributed architectures), and they limit the scale and fault-tolerance properties you can achieve.

While the traditional transaction approach is not going away, I also believe it is not the last word in making applications correct and resilient to faults. In this section I will suggest some ways of thinking about correctness in the context of dataflow architectures.

The End-to-End Argument for Databases

Just because an application uses a data system that provides comparatively strong safety properties, such as serializable transactions, that does not mean the application is guaranteed to be free from data loss or corruption. For example, if an application has a bug that causes it to write incorrect data, or delete data from a database, serializable transactions aren't going to save you.

This example may seem frivolous, but it is worth taking seriously: application bugs occur, and people make mistakes. I used this example in "State, Streams, and Immutability" on page 459 to argue in favor of immutable and append-only data, because it is easier to recover from such mistakes if you remove the ability of faulty code to destroy good data.

Although immutability is useful, it is not a cure-all by itself. Let's look at a more subtle example of data corruption that can occur.

Exactly-once execution of an operation

In "Fault Tolerance" on page 476 we encountered an idea called *exactly-once* (or *effectively-once*) semantics. If something goes wrong while processing a message, you can either give up (drop the message—i.e., incur data loss) or try again. If you try again, there is the risk that it actually succeeded the first time, but you just didn't find out about the success, and so the message ends up being processed twice.

Processing twice is a form of data corruption: it is undesirable to charge a customer twice for the same service (billing them too much) or increment a counter twice

(overstating some metric). In this context, *exactly-once* means arranging the computation such that the final effect is the same as if no faults had occurred, even if the operation actually was retried due to some fault. We previously discussed a few approaches for achieving this goal.

One of the most effective approaches is to make the operation *idempotent* (see "Idempotence" on page 478); that is, to ensure that it has the same effect, no matter whether it is executed once or multiple times. However, taking an operation that is not naturally idempotent and making it idempotent requires some effort and care: you may need to maintain some additional metadata (such as the set of operation IDs that have updated a value), and ensure fencing when failing over from one node to another (see "The leader and the lock" on page 301).

Duplicate suppression

The same pattern of needing to suppress duplicates occurs in many other places besides stream processing. For example, TCP uses sequence numbers on packets to put them in the correct order at the recipient, and to determine whether any packets were lost or duplicated on the network. Any lost packets are retransmitted and any duplicates are removed by the TCP stack before it hands the data to an application.

However, this duplicate suppression only works within the context of a single TCP connection. Imagine the TCP connection is a client's connection to a database, and it is currently executing the transaction in Example 12-1. In many databases, a transaction is tied to a client connection (if the client sends several queries, the database knows that they belong to the same transaction because they are sent on the same TCP connection). If the client suffers a network interruption and connection timeout after sending the `COMMIT`, but before hearing back from the database server, it does not know whether the transaction has been committed or aborted (Figure 8-1).

Example 12-1. A nonidempotent transfer of money from one account to another

```
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance + 11.00 WHERE account_id = 1234;
UPDATE accounts SET balance = balance - 11.00 WHERE account_id = 4321;
COMMIT;
```

The client can reconnect to the database and retry the transaction, but now it is outside of the scope of TCP duplicate suppression. Since the transaction in Example 12-1 is not idempotent, it could happen that $22 is transferred instead of the desired $11. Thus, even though Example 12-1 is a standard example for transaction atomicity, it is actually not correct, and real banks do not work like this [3].

Two-phase commit (see "Atomic Commit and Two-Phase Commit (2PC)" on page 354) protocols break the 1:1 mapping between a TCP connection and a transaction, since they must allow a transaction coordinator to reconnect to a database after a net-

work fault, and tell it whether to commit or abort an in-doubt transaction. Is this sufficient to ensure that the transaction will only be executed once? Unfortunately not.

Even if we can suppress duplicate transactions between the database client and server, we still need to worry about the network between the end-user device and the application server. For example, if the end-user client is a web browser, it probably uses an HTTP POST request to submit an instruction to the server. Perhaps the user is on a weak cellular data connection, and they succeed in sending the POST, but the signal becomes too weak before they are able to receive the response from the server.

In this case, the user will probably be shown an error message, and they may retry manually. Web browsers warn, "Are you sure you want to submit this form again?"—and the user says yes, because they wanted the operation to happen. (The Post/Redirect/Get pattern [54] avoids this warning message in normal operation, but it doesn't help if the POST request times out.) From the web server's point of view the retry is a separate request, and from the database's point of view it is a separate transaction. The usual deduplication mechanisms don't help.

Operation identifiers

To make the operation idempotent through several hops of network communication, it is not sufficient to rely just on a transaction mechanism provided by a database—you need to consider the *end-to-end* flow of the request.

For example, you could generate a unique identifier for an operation (such as a UUID) and include it as a hidden form field in the client application, or calculate a hash of all the relevant form fields to derive the operation ID [3]. If the web browser submits the POST request twice, the two requests will have the same operation ID. You can then pass that operation ID all the way through to the database and check that you only ever execute one operation with a given ID, as shown in Example 12-2.

Example 12-2. Suppressing duplicate requests using a unique ID

```
ALTER TABLE requests ADD UNIQUE (request_id);

BEGIN TRANSACTION;

INSERT INTO requests
  (request_id, from_account, to_account, amount)
  VALUES('0286FDB8-D7E1-423F-B40B-792B3608036C', 4321, 1234, 11.00);

UPDATE accounts SET balance = balance + 11.00 WHERE account_id = 1234;
UPDATE accounts SET balance = balance - 11.00 WHERE account_id = 4321;

COMMIT;
```

Example 12-2 relies on a uniqueness constraint on the `request_id` column. If a transaction attempts to insert an ID that already exists, the `INSERT` fails and the transaction is aborted, preventing it from taking effect twice. Relational databases can generally maintain a uniqueness constraint correctly, even at weak isolation levels (whereas an application-level check-then-insert may fail under nonserializable isolation, as discussed in "Write Skew and Phantoms" on page 246).

Besides suppressing duplicate requests, the `requests` table in Example 12-2 acts as a kind of event log, hinting in the direction of event sourcing (see "Event Sourcing" on page 457). The updates to the account balances don't actually have to happen in the same transaction as the insertion of the event, since they are redundant and could be derived from the request event in a downstream consumer—as long as the event is processed exactly once, which can again be enforced using the request ID.

The end-to-end argument

This scenario of suppressing duplicate transactions is just one example of a more general principle called the *end-to-end argument*, which was articulated by Saltzer, Reed, and Clark in 1984 [55]:

> The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the endpoints of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)

In our example, the *function in question* was duplicate suppression. We saw that TCP suppresses duplicate packets at the TCP connection level, and some stream processors provide so-called exactly-once semantics at the message processing level, but that is not enough to prevent a user from submitting a duplicate request if the first one times out. By themselves, TCP, database transactions, and stream processors cannot entirely rule out these duplicates. Solving the problem requires an end-to-end solution: a transaction identifier that is passed all the way from the end-user client to the database.

The end-to-end argument also applies to checking the integrity of data: checksums built into Ethernet, TCP, and TLS can detect corruption of packets in the network, but they cannot detect corruption due to bugs in the software at the sending and receiving ends of the network connection, or corruption on the disks where the data is stored. If you want to catch all possible sources of data corruption, you also need end-to-end checksums.

A similar argument applies with encryption [55]: the password on your home WiFi network protects against people snooping your WiFi traffic, but not against attackers elsewhere on the internet; TLS/SSL between your client and the server protects

against network attackers, but not against compromises of the server. Only end-to-end encryption and authentication can protect against all of these things.

Although the low-level features (TCP duplicate suppression, Ethernet checksums, WiFi encryption) cannot provide the desired end-to-end features by themselves, they are still useful, since they reduce the probability of problems at the higher levels. For example, HTTP requests would often get mangled if we didn't have TCP putting the packets back in the right order. We just need to remember that the low-level reliability features are not by themselves sufficient to ensure end-to-end correctness.

Applying end-to-end thinking in data systems

This brings me back to my original thesis: just because an application uses a data system that provides comparatively strong safety properties, such as serializable transactions, that does not mean the application is guaranteed to be free from data loss or corruption. The application itself needs to take end-to-end measures, such as duplicate suppression, as well.

That is a shame, because fault-tolerance mechanisms are hard to get right. Low-level reliability mechanisms, such as those in TCP, work quite well, and so the remaining higher-level faults occur fairly rarely. It would be really nice to wrap up the remaining high-level fault-tolerance machinery in an abstraction so that application code needn't worry about it—but I fear that we have not yet found the right abstraction.

Transactions have long been seen as a good abstraction, and I do believe that they are useful. As discussed in the introduction to Chapter 7, they take a wide range of possible issues (concurrent writes, constraint violations, crashes, network interruptions, disk failures) and collapse them down to two possible outcomes: commit or abort. That is a huge simplification of the programming model, but I fear that it is not enough.

Transactions are expensive, especially when they involve heterogeneous storage technologies (see "Distributed Transactions in Practice" on page 360). When we refuse to use distributed transactions because they are too expensive, we end up having to reimplement fault-tolerance mechanisms in application code. As numerous examples throughout this book have shown, reasoning about concurrency and partial failure is difficult and counterintuitive, and so I suspect that most application-level mechanisms do not work correctly. The consequence is lost or corrupted data.

For these reasons, I think it is worth exploring fault-tolerance abstractions that make it easy to provide application-specific end-to-end correctness properties, but also maintain good performance and good operational characteristics in a large-scale distributed environment.

Enforcing Constraints

Let's think about correctness in the context of the ideas around unbundling databases ("Unbundling Databases" on page 499). We saw that end-to-end duplicate suppression can be achieved with a request ID that is passed all the way from the client to the database that records the write. What about other kinds of constraints?

In particular, let's focus on uniqueness constraints—such as the one we relied on in Example 12-2. In "Constraints and uniqueness guarantees" on page 330 we saw several other examples of application features that need to enforce uniqueness: a username or email address must uniquely identify a user, a file storage service cannot have more than one file with the same name, and two people cannot book the same seat on a flight or in a theater.

Other kinds of constraints are very similar: for example, ensuring that an account balance never goes negative, that you don't sell more items than you have in stock in the warehouse, or that a meeting room does not have overlapping bookings. Techniques that enforce uniqueness can often be used for these kinds of constraints as well.

Uniqueness constraints require consensus

In Chapter 9 we saw that in a distributed setting, enforcing a uniqueness constraint requires consensus: if there are several concurrent requests with the same value, the system somehow needs to decide which one of the conflicting operations is accepted, and reject the others as violations of the constraint.

The most common way of achieving this consensus is to make a single node the leader, and put it in charge of making all the decisions. That works fine as long as you don't mind funneling all requests through a single node (even if the client is on the other side of the world), and as long as that node doesn't fail. If you need to tolerate the leader failing, you're back at the consensus problem again (see "Single-leader replication and consensus" on page 367).

Uniqueness checking can be scaled out by partitioning based on the value that needs to be unique. For example, if you need to ensure uniqueness by request ID, as in Example 12-2, you can ensure all requests with the same request ID are routed to the same partition (see Chapter 6). If you need usernames to be unique, you can partition by hash of username.

However, asynchronous multi-master replication is ruled out, because it could happen that different masters concurrently accept conflicting writes, and thus the values are no longer unique (see "Implementing Linearizable Systems" on page 332). If you want to be able to immediately reject any writes that would violate the constraint, synchronous coordination is unavoidable [56].

Uniqueness in log-based messaging

The log ensures that all consumers see messages in the same order—a guarantee that is formally known as *total order broadcast* and is equivalent to consensus (see "Total Order Broadcast" on page 348). In the unbundled database approach with log-based messaging, we can use a very similar approach to enforce uniqueness constraints.

A stream processor consumes all the messages in a log partition sequentially on a single thread (see "Logs compared to traditional messaging" on page 448). Thus, if the log is partitioned based on the value that needs to be unique, a stream processor can unambiguously and deterministically decide which one of several conflicting operations came first. For example, in the case of several users trying to claim the same username [57]:

1. Every request for a username is encoded as a message, and appended to a partition determined by the hash of the username.
2. A stream processor sequentially reads the requests in the log, using a local database to keep track of which usernames are taken. For every request for a username that is available, it records the name as taken and emits a success message to an output stream. For every request for a username that is already taken, it emits a rejection message to an output stream.
3. The client that requested the username watches the output stream and waits for a success or rejection message corresponding to its request.

This algorithm is basically the same as in "Implementing linearizable storage using total order broadcast" on page 350. It scales easily to a large request throughput by increasing the number of partitions, as each partition can be processed independently.

The approach works not only for uniqueness constraints, but also for many other kinds of constraints. Its fundamental principle is that any writes that may conflict are routed to the same partition and processed sequentially. As discussed in "What is a conflict?" on page 174 and "Write Skew and Phantoms" on page 246, the definition of a conflict may depend on the application, but the stream processor can use arbitrary logic to validate a request. This idea is similar to the approach pioneered by Bayou in the 1990s [58].

Multi-partition request processing

Ensuring that an operation is executed atomically, while satisfying constraints, becomes more interesting when several partitions are involved. In Example 12-2, there are potentially three partitions: the one containing the request ID, the one containing the payee account, and the one containing the payer account. There is no rea-

son why those three things should be in the same partition, since they are all independent from each other.

In the traditional approach to databases, executing this transaction would require an atomic commit across all three partitions, which essentially forces it into a total order with respect to all other transactions on any of those partitions. Since there is now cross-partition coordination, different partitions can no longer be processed independently, so throughput is likely to suffer.

However, it turns out that equivalent correctness can be achieved with partitioned logs, and without an atomic commit:

1. The request to transfer money from account A to account B is given a unique request ID by the client, and appended to a log partition based on the request ID.
2. A stream processor reads the log of requests. For each request message it emits two messages to output streams: a debit instruction to the payer account A (partitioned by A), and a credit instruction to the payee account B (partitioned by B). The original request ID is included in those emitted messages.
3. Further processors consume the streams of credit and debit instructions, deduplicate by request ID, and apply the changes to the account balances.

Steps 1 and 2 are necessary because if the client directly sent the credit and debit instructions, it would require an atomic commit across those two partitions to ensure that either both or neither happen. To avoid the need for a distributed transaction, we first durably log the request as a single message, and then derive the credit and debit instructions from that first message. Single-object writes are atomic in almost all data systems (see "Single-object writes" on page 230), and so the request either appears in the log or it doesn't, without any need for a multi-partition atomic commit.

If the stream processor in step 2 crashes, it resumes processing from its last checkpoint. In doing so, it does not skip any request messages, but it may process requests multiple times and produce duplicate credit and debit instructions. However, since it is deterministic, it will just produce the same instructions again, and the processors in step 3 can easily deduplicate them using the end-to-end request ID.

If you want to ensure that the payer account is not overdrawn by this transfer, you can additionally have a stream processor (partitioned by payer account number) that maintains account balances and validates transactions. Only valid transactions would then be placed in the request log in step 1.

By breaking down the multi-partition transaction into two differently partitioned stages and using the end-to-end request ID, we have achieved the same correctness property (every request is applied exactly once to both the payer and payee accounts), even in the presence of faults, and without using an atomic commit protocol. The

idea of using multiple differently partitioned stages is similar to what we discussed in "Multi-partition data processing" on page 514 (see also "Concurrency control" on page 462).

Timeliness and Integrity

A convenient property of transactions is that they are typically linearizable (see "Linearizability" on page 324): that is, a writer waits until a transaction is committed, and thereafter its writes are immediately visible to all readers.

This is not the case when unbundling an operation across multiple stages of stream processors: consumers of a log are asynchronous by design, so a sender does not wait until its message has been processed by consumers. However, it is possible for a client to wait for a message to appear on an output stream. This is what we did in "Uniqueness in log-based messaging" on page 522 when checking whether a uniqueness constraint was satisfied.

In this example, the correctness of the uniqueness check does not depend on whether the sender of the message waits for the outcome. The waiting only has the purpose of synchronously informing the sender whether or not the uniqueness check succeeded, but this notification can be decoupled from the effects of processing the message.

More generally, I think the term *consistency* conflates two different requirements that are worth considering separately:

Timeliness
: Timeliness means ensuring that users observe the system in an up-to-date state. We saw previously that if a user reads from a stale copy of the data, they may observe it in an inconsistent state (see "Problems with Replication Lag" on page 161). However, that inconsistency is temporary, and will eventually be resolved simply by waiting and trying again.

  The CAP theorem (see "The Cost of Linearizability" on page 335) uses consistency in the sense of linearizability, which is a strong way of achieving timeliness. Weaker timeliness properties like *read-after-write* consistency (see "Reading Your Own Writes" on page 162) can also be useful.

Integrity
: Integrity means absence of corruption; i.e., no data loss, and no contradictory or false data. In particular, if some derived dataset is maintained as a view onto some underlying data (see "Deriving current state from the event log" on page 458), the derivation must be correct. For example, a database index must correctly reflect the contents of the database—an index in which some records are missing is not very useful.

> If integrity is violated, the inconsistency is permanent: waiting and trying again is not going to fix database corruption in most cases. Instead, explicit checking and repair is needed. In the context of ACID transactions (see "The Meaning of ACID" on page 223), consistency is usually understood as some kind of application-specific notion of integrity. Atomicity and durability are important tools for preserving integrity.

In slogan form: violations of timeliness are "eventual consistency," whereas violations of integrity are "perpetual inconsistency."

I am going to assert that in most applications, integrity is much more important than timeliness. Violations of timeliness can be annoying and confusing, but violations of integrity can be catastrophic.

For example, on your credit card statement, it is not surprising if a transaction that you made within the last 24 hours does not yet appear—it is normal that these systems have a certain lag. We know that banks reconcile and settle transactions asynchronously, and timeliness is not very important here [3]. However, it would be very bad if the statement balance was not equal to the sum of the transactions plus the previous statement balance (an error in the sums), or if a transaction was charged to you but not paid to the merchant (disappearing money). Such problems would be violations of the integrity of the system.

Correctness of dataflow systems

ACID transactions usually provide both timeliness (e.g., linearizability) and integrity (e.g., atomic commit) guarantees. Thus, if you approach application correctness from the point of view of ACID transactions, the distinction between timeliness and integrity is fairly inconsequential.

On the other hand, an interesting property of the event-based dataflow systems that we have discussed in this chapter is that they decouple timeliness and integrity. When processing event streams asynchronously, there is no guarantee of timeliness, unless you explicitly build consumers that wait for a message to arrive before returning. But integrity is in fact central to streaming systems.

Exactly-once or *effectively-once* semantics (see "Fault Tolerance" on page 476) is a mechanism for preserving integrity. If an event is lost, or if an event takes effect twice, the integrity of a data system could be violated. Thus, fault-tolerant message delivery and duplicate suppression (e.g., idempotent operations) are important for maintaining the integrity of a data system in the face of faults.

As we saw in the last section, reliable stream processing systems can preserve integrity without requiring distributed transactions and an atomic commit protocol, which means they can potentially achieve comparable correctness with much better

performance and operational robustness. We achieved this integrity through a combination of mechanisms:

- Representing the content of the write operation as a single message, which can easily be written atomically—an approach that fits very well with event sourcing (see "Event Sourcing" on page 457)
- Deriving all other state updates from that single message using deterministic derivation functions, similarly to stored procedures (see "Actual Serial Execution" on page 252 and "Application code as a derivation function" on page 505)
- Passing a client-generated request ID through all these levels of processing, enabling end-to-end duplicate suppression and idempotence
- Making messages immutable and allowing derived data to be reprocessed from time to time, which makes it easier to recover from bugs (see "Advantages of immutable events" on page 460)

This combination of mechanisms seems to me a very promising direction for building fault-tolerant applications in the future.

Loosely interpreted constraints

As discussed previously, enforcing a uniqueness constraint requires consensus, typically implemented by funneling all events in a particular partition through a single node. This limitation is unavoidable if we want the traditional form of uniqueness constraint, and stream processing cannot avoid it.

However, another thing to realize is that many real applications can actually get away with much weaker notions of uniqueness:

- If two people concurrently register the same username or book the same seat, you can send one of them a message to apologize, and ask them to choose a different one. This kind of change to correct a mistake is called a *compensating transaction* [59, 60].
- If customers order more items than you have in your warehouse, you can order in more stock, apologize to customers for the delay, and offer them a discount. This is actually the same as what you'd have to do if, say, a forklift truck ran over some of the items in your warehouse, leaving you with fewer items in stock than you thought you had [61]. Thus, the apology workflow already needs to be part of your business processes anyway, and so it might be unnecessary to require a linearizable constraint on the number of items in stock.
- Similarly, many airlines overbook airplanes in the expectation that some passengers will miss their flight, and many hotels overbook rooms, expecting that some guests will cancel. In these cases, the constraint of "one person per seat" is delib-

erately violated for business reasons, and compensation processes (refunds, upgrades, providing a complimentary room at a neighboring hotel) are put in place to handle situations in which demand exceeds supply. Even if there was no overbooking, apology and compensation processes would be needed in order to deal with flights being cancelled due to bad weather or staff on strike—recovering from such issues is just a normal part of business [3].

- If someone withdraws more money than they have in their account, the bank can charge them an overdraft fee and ask them to pay back what they owe. By limiting the total withdrawals per day, the risk to the bank is bounded.

In many business contexts, it is actually acceptable to temporarily violate a constraint and fix it up later by apologizing. The cost of the apology (in terms of money or reputation) varies, but it is often quite low: you can't unsend an email, but you can send a follow-up email with a correction. If you accidentally charge a credit card twice, you can refund one of the charges, and the cost to you is just the processing fees and perhaps a customer complaint. Once money has been paid out of an ATM, you can't directly get it back, although in principle you can send debt collectors to recover the money if the account was overdrawn and the customer won't pay it back.

Whether the cost of the apology is acceptable is a business decision. If it is acceptable, the traditional model of checking all constraints before even writing the data is unnecessarily restrictive, and a linearizable constraint is not needed. It may well be a reasonable choice to go ahead with a write optimistically, and to check the constraint after the fact. You can still ensure that the validation occurs before doing things that would be expensive to recover from, but that doesn't imply you must do the validation before you even write the data.

These applications *do* require integrity: you would not want to lose a reservation, or have money disappear due to mismatched credits and debits. But they *don't* require timeliness on the enforcement of the constraint: if you have sold more items than you have in the warehouse, you can patch up the problem after the fact by apologizing. Doing so is similar to the conflict resolution approaches we discussed in "Handling Write Conflicts" on page 171.

Coordination-avoiding data systems

We have now made two interesting observations:

1. Dataflow systems can maintain integrity guarantees on derived data without atomic commit, linearizability, or synchronous cross-partition coordination.
2. Although strict uniqueness constraints require timeliness and coordination, many applications are actually fine with loose constraints that may be temporarily violated and fixed up later, as long as integrity is preserved throughout.

Taken together, these observations mean that dataflow systems can provide the data management services for many applications without requiring coordination, while still giving strong integrity guarantees. Such *coordination-avoiding* data systems have a lot of appeal: they can achieve better performance and fault tolerance than systems that need to perform synchronous coordination [56].

For example, such a system could operate distributed across multiple datacenters in a multi-leader configuration, asynchronously replicating between regions. Any one datacenter can continue operating independently from the others, because no synchronous cross-region coordination is required. Such a system would have weak timeliness guarantees—it could not be linearizable without introducing coordination—but it can still have strong integrity guarantees.

In this context, serializable transactions are still useful as part of maintaining derived state, but they can be run at a small scope where they work well [8]. Heterogeneous distributed transactions such as XA transactions (see "Distributed Transactions in Practice" on page 360) are not required. Synchronous coordination can still be introduced in places where it is needed (for example, to enforce strict constraints before an operation from which recovery is not possible), but there is no need for everything to pay the cost of coordination if only a small part of an application needs it [43].

Another way of looking at coordination and constraints: they reduce the number of apologies you have to make due to inconsistencies, but potentially also reduce the performance and availability of your system, and thus potentially increase the number of apologies you have to make due to outages. You cannot reduce the number of apologies to zero, but you can aim to find the best trade-off for your needs—the sweet spot where there are neither too many inconsistencies nor too many availability problems.

Trust, but Verify

All of our discussion of correctness, integrity, and fault-tolerance has been under the assumption that certain things might go wrong, but other things won't. We call these assumptions our *system model* (see "Mapping system models to the real world" on page 309): for example, we should assume that processes can crash, machines can suddenly lose power, and the network can arbitrarily delay or drop messages. But we might also assume that data written to disk is not lost after `fsync`, that data in memory is not corrupted, and that the multiplication instruction of our CPU always returns the correct result.

These assumptions are quite reasonable, as they are true most of the time, and it would be difficult to get anything done if we had to constantly worry about our computers making mistakes. Traditionally, system models take a binary approach toward faults: we assume that some things can happen, and other things can never happen. In reality, it is more a question of probabilities: some things are more likely, other

things less likely. The question is whether violations of our assumptions happen often enough that we may encounter them in practice.

We have seen that data can become corrupted while it is sitting untouched on disks (see "Replication and Durability" on page 227), and data corruption on the network can sometimes evade the TCP checksums (see "Weak forms of lying" on page 306). Maybe this is something we should be paying more attention to?

One application that I worked on in the past collected crash reports from clients, and some of the reports we received could only be explained by random bit-flips in the memory of those devices. It seems unlikely, but if you have enough devices running your software, even very unlikely things do happen. Besides random memory corruption due to hardware faults or radiation, certain pathological memory access patterns can flip bits even in memory that has no faults [62]—an effect that can be used to break security mechanisms in operating systems [63] (this technique is known as *rowhammer*). Once you look closely, hardware isn't quite the perfect abstraction that it may seem.

To be clear, random bit-flips are still very rare on modern hardware [64]. I just want to point out that they are not beyond the realm of possibility, and so they deserve some attention.

Maintaining integrity in the face of software bugs

Besides such hardware issues, there is always the risk of software bugs, which would not be caught by lower-level network, memory, or filesystem checksums. Even widely used database software has bugs: I have personally seen cases of MySQL failing to correctly maintain a uniqueness constraint [65] and PostgreSQL's serializable isolation level exhibiting write skew anomalies [66], even though MySQL and PostgreSQL are robust and well-regarded databases that have been battle-tested by many people for many years. In less mature software, the situation is likely to be much worse.

Despite considerable efforts in careful design, testing, and review, bugs still creep in. Although they are rare, and they eventually get found and fixed, there is still a period during which such bugs can corrupt data.

When it comes to application code, we have to assume many more bugs, since most applications don't receive anywhere near the amount of review and testing that database code does. Many applications don't even correctly use the features that databases offer for preserving integrity, such as foreign key or uniqueness constraints [36].

Consistency in the sense of ACID (see "Consistency" on page 224) is based on the idea that the database starts off in a consistent state, and a transaction transforms it from one consistent state to another consistent state. Thus, we expect the database to always be in a consistent state. However, this notion only makes sense if you assume that the transaction is free from bugs. If the application uses the database incorrectly

in some way, for example using a weak isolation level unsafely, the integrity of the database cannot be guaranteed.

Don't just blindly trust what they promise

With both hardware and software not always living up to the ideal that we would like them to be, it seems that data corruption is inevitable sooner or later. Thus, we should at least have a way of finding out if data has been corrupted so that we can fix it and try to track down the source of the error. Checking the integrity of data is known as *auditing*.

As discussed in "Advantages of immutable events" on page 460, auditing is not just for financial applications. However, auditability is highly important in finance precisely because everyone knows that mistakes happen, and we all recognize the need to be able to detect and fix problems.

Mature systems similarly tend to consider the possibility of unlikely things going wrong, and manage that risk. For example, large-scale storage systems such as HDFS and Amazon S3 do not fully trust disks: they run background processes that continually read back files, compare them to other replicas, and move files from one disk to another, in order to mitigate the risk of silent corruption [67].

If you want to be sure that your data is still there, you have to actually read it and check. Most of the time it will still be there, but if it isn't, you really want to find out sooner rather than later. By the same argument, it is important to try restoring from your backups from time to time—otherwise you may only find out that your backup is broken when it is too late and you have already lost data. Don't just blindly trust that it is all working.

A culture of verification

Systems like HDFS and S3 still have to assume that disks work correctly most of the time—which is a reasonable assumption, but not the same as assuming that they *always* work correctly. However, not many systems currently have this kind of "trust, but verify" approach of continually auditing themselves. Many assume that correctness guarantees are absolute and make no provision for the possibility of rare data corruption. I hope that in the future we will see more *self-validating* or *self-auditing* systems that continually check their own integrity, rather than relying on blind trust [68].

I fear that the culture of ACID databases has led us toward developing applications on the basis of blindly trusting technology (such as a transaction mechanism), and neglecting any sort of auditability in the process. Since the technology we trusted worked well enough most of the time, auditing mechanisms were not deemed worth the investment.

But then the database landscape changed: weaker consistency guarantees became the norm under the banner of NoSQL, and less mature storage technologies became widely used. Yet, because the audit mechanisms had not been developed, we continued building applications on the basis of blind trust, even though this approach had now become more dangerous. Let's think for a moment about designing for auditability.

Designing for auditability

If a transaction mutates several objects in a database, it is difficult to tell after the fact what that transaction means. Even if you capture the transaction logs (see "Change Data Capture" on page 454), the insertions, updates, and deletions in various tables do not necessarily give a clear picture of *why* those mutations were performed. The invocation of the application logic that decided on those mutations is transient and cannot be reproduced.

By contrast, event-based systems can provide better auditability. In the event sourcing approach, user input to the system is represented as a single immutable event, and any resulting state updates are derived from that event. The derivation can be made deterministic and repeatable, so that running the same log of events through the same version of the derivation code will result in the same state updates.

Being explicit about dataflow (see "Philosophy of batch process outputs" on page 413) makes the *provenance* of data much clearer, which makes integrity checking much more feasible. For the event log, we can use hashes to check that the event storage has not been corrupted. For any derived state, we can rerun the batch and stream processors that derived it from the event log in order to check whether we get the same result, or even run a redundant derivation in parallel.

A deterministic and well-defined dataflow also makes it easier to debug and trace the execution of a system in order to determine why it did something [4, 69]. If something unexpected occurred, it is valuable to have the diagnostic capability to reproduce the exact circumstances that led to the unexpected event—a kind of time-travel debugging capability.

The end-to-end argument again

If we cannot fully trust that every individual component of the system will be free from corruption—that every piece of hardware is fault-free and that every piece of software is bug-free—then we must at least periodically check the integrity of our data. If we don't check, we won't find out about corruption until it is too late and it has caused some downstream damage, at which point it will be much harder and more expensive to track down the problem.

Checking the integrity of data systems is best done in an end-to-end fashion (see "The End-to-End Argument for Databases" on page 516): the more systems we can

include in an integrity check, the fewer opportunities there are for corruption to go unnoticed at some stage of the process. If we can check that an entire derived data pipeline is correct end to end, then any disks, networks, services, and algorithms along the path are implicitly included in the check.

Having continuous end-to-end integrity checks gives you increased confidence about the correctness of your systems, which in turn allows you to move faster [70]. Like automated testing, auditing increases the chances that bugs will be found quickly, and thus reduces the risk that a change to the system or a new storage technology will cause damage. If you are not afraid of making changes, you can much better evolve an application to meet changing requirements.

Tools for auditable data systems

At present, not many data systems make auditability a top-level concern. Some applications implement their own audit mechanisms, for example by logging all changes to a separate audit table, but guaranteeing the integrity of the audit log and the database state is still difficult. A transaction log can be made tamper-proof by periodically signing it with a hardware security module, but that does not guarantee that the right transactions went into the log in the first place.

It would be interesting to use cryptographic tools to prove the integrity of a system in a way that is robust to a wide range of hardware and software issues, and even potentially malicious actions. Cryptocurrencies, blockchains, and distributed ledger technologies such as Bitcoin, Ethereum, Ripple, Stellar, and various others [71, 72, 73] have sprung up to explore this area.

I am not qualified to comment on the merits of these technologies as currencies or mechanisms for agreeing contracts. However, from a data systems point of view they contain some interesting ideas. Essentially, they are distributed databases, with a data model and transaction mechanism, in which different replicas can be hosted by mutually untrusting organizations. The replicas continually check each other's integrity and use a consensus protocol to agree on the transactions that should be executed.

I am somewhat skeptical about the Byzantine fault tolerance aspects of these technologies (see "Byzantine Faults" on page 304), and I find the technique of *proof of work* (e.g., Bitcoin mining) extraordinarily wasteful. The transaction throughput of Bitcoin is rather low, albeit for political and economic reasons more than for technical ones. However, the integrity checking aspects are interesting.

Cryptographic auditing and integrity checking often relies on *Merkle trees* [74], which are trees of hashes that can be used to efficiently prove that a record appears in some dataset (and a few other things). Outside of the hype of cryptocurrencies, *certificate transparency* is a security technology that relies on Merkle trees to check the validity of TLS/SSL certificates [75, 76].

I could imagine integrity-checking and auditing algorithms, like those of certificate transparency and distributed ledgers, becoming more widely used in data systems in general. Some work will be needed to make them equally scalable as systems without cryptographic auditing, and to keep the performance penalty as low as possible. But I think this is an interesting area to watch in the future.

Doing the Right Thing

In the final section of this book, I would like to take a step back. Throughout this book we have examined a wide range of different architectures for data systems, evaluated their pros and cons, and explored techniques for building reliable, scalable, and maintainable applications. However, we have left out an important and fundamental part of the discussion, which I would now like to fill in.

Every system is built for a purpose; every action we take has both intended and unintended consequences. The purpose may be as simple as making money, but the consequences for the world may reach far beyond that original purpose. We, the engineers building these systems, have a responsibility to carefully consider those consequences and to consciously decide what kind of world we want to live in.

We talk about data as an abstract thing, but remember that many datasets are about people: their behavior, their interests, their identity. We must treat such data with humanity and respect. Users are humans too, and human dignity is paramount.

Software development increasingly involves making important ethical choices. There are guidelines to help software engineers navigate these issues, such as the ACM's Software Engineering Code of Ethics and Professional Practice [77], but they are rarely discussed, applied, and enforced in practice. As a result, engineers and product managers sometimes take a very cavalier attitude to privacy and potential negative consequences of their products [78, 79, 80].

A technology is not good or bad in itself—what matters is how it is used and how it affects people. This is true for a software system like a search engine in much the same way as it is for a weapon like a gun. I think it is not sufficient for software engineers to focus exclusively on the technology and ignore its consequences: the ethical responsibility is ours to bear also. Reasoning about ethics is difficult, but it is too important to ignore.

Predictive Analytics

For example, predictive analytics is a major part of the "Big Data" hype. Using data analysis to predict the weather, or the spread of diseases, is one thing [81]; it is another matter to predict whether a convict is likely to reoffend, whether an applicant for a loan is likely to default, or whether an insurance customer is likely to make expensive claims. The latter have a direct effect on individual people's lives.

Naturally, payment networks want to prevent fraudulent transactions, banks want to avoid bad loans, airlines want to avoid hijackings, and companies want to avoid hiring ineffective or untrustworthy people. From their point of view, the cost of a missed business opportunity is low, but the cost of a bad loan or a problematic employee is much higher, so it is natural for organizations to want to be cautious. If in doubt, they are better off saying no.

However, as algorithmic decision-making becomes more widespread, someone who has (accurately or falsely) been labeled as risky by some algorithm may suffer a large number of those "no" decisions. Systematically being excluded from jobs, air travel, insurance coverage, property rental, financial services, and other key aspects of society is such a large constraint of the individual's freedom that it has been called "algorithmic prison" [82]. In countries that respect human rights, the criminal justice system presumes innocence until proven guilty; on the other hand, automated systems can systematically and arbitrarily exclude a person from participating in society without any proof of guilt, and with little chance of appeal.

Bias and discrimination

Decisions made by an algorithm are not necessarily any better or any worse than those made by a human. Every person is likely to have biases, even if they actively try to counteract them, and discriminatory practices can become culturally institutionalized. There is hope that basing decisions on data, rather than subjective and instinctive assessments by people, could be more fair and give a better chance to people who are often overlooked in the traditional system [83].

When we develop predictive analytics systems, we are not merely automating a human's decision by using software to specify the rules for when to say yes or no; we are even leaving the rules themselves to be inferred from data. However, the patterns learned by these systems are opaque: even if there is some correlation in the data, we may not know why. If there is a systematic bias in the input to an algorithm, the system will most likely learn and amplify that bias in its output [84].

In many countries, anti-discrimination laws prohibit treating people differently depending on protected traits such as ethnicity, age, gender, sexuality, disability, or beliefs. Other features of a person's data may be analyzed, but what happens if they are correlated with protected traits? For example, in racially segregated neighborhoods, a person's postal code or even their IP address is a strong predictor of race. Put like this, it seems ridiculous to believe that an algorithm could somehow take biased data as input and produce fair and impartial output from it [85]. Yet this belief often seems to be implied by proponents of data-driven decision making, an attitude that has been satirized as "machine learning is like money laundering for bias" [86].

Predictive analytics systems merely extrapolate from the past; if the past is discriminatory, they codify that discrimination. If we want the future to be better than the

past, moral imagination is required, and that's something only humans can provide [87]. Data and models should be our tools, not our masters.

Responsibility and accountability

Automated decision making opens the question of responsibility and accountability [87]. If a human makes a mistake, they can be held accountable, and the person affected by the decision can appeal. Algorithms make mistakes too, but who is accountable if they go wrong [88]? When a self-driving car causes an accident, who is responsible? If an automated credit scoring algorithm systematically discriminates against people of a particular race or religion, is there any recourse? If a decision by your machine learning system comes under judicial review, can you explain to the judge how the algorithm made its decision?

Credit rating agencies are an old example of collecting data to make decisions about people. A bad credit score makes life difficult, but at least a credit score is normally based on relevant facts about a person's actual borrowing history, and any errors in the record can be corrected (although the agencies normally do not make this easy). However, scoring algorithms based on machine learning typically use a much wider range of inputs and are much more opaque, making it harder to understand how a particular decision has come about and whether someone is being treated in an unfair or discriminatory way [89].

A credit score summarizes "How did you behave in the past?" whereas predictive analytics usually work on the basis of "Who is similar to you, and how did people like you behave in the past?" Drawing parallels to others' behavior implies stereotyping people, for example based on where they live (a close proxy for race and socioeconomic class). What about people who get put in the wrong bucket? Furthermore, if a decision is incorrect due to erroneous data, recourse is almost impossible [87].

Much data is statistical in nature, which means that even if the probability distribution on the whole is correct, individual cases may well be wrong. For example, if the average life expectancy in your country is 80 years, that doesn't mean you're expected to drop dead on your 80th birthday. From the average and the probability distribution, you can't say much about the age to which one particular person will live. Similarly, the output of a prediction system is probabilistic and may well be wrong in individual cases.

A blind belief in the supremacy of data for making decisions is not only delusional, it is positively dangerous. As data-driven decision making becomes more widespread, we will need to figure out how to make algorithms accountable and transparent, how to avoid reinforcing existing biases, and how to fix them when they inevitably make mistakes.

We will also need to figure out how to prevent data being used to harm people, and realize its positive potential instead. For example, analytics can reveal financial and

social characteristics of people's lives. On the one hand, this power could be used to focus aid and support to help those people who most need it. On the other hand, it is sometimes used by predatory business seeking to identify vulnerable people and sell them risky products such as high-cost loans and worthless college degrees [87, 90].

Feedback loops

Even with predictive applications that have less immediately far-reaching effects on people, such as recommendation systems, there are difficult issues that we must confront. When services become good at predicting what content users want to see, they may end up showing people only opinions they already agree with, leading to echo chambers in which stereotypes, misinformation, and polarization can breed. We are already seeing the impact of social media echo chambers on election campaigns [91].

When predictive analytics affect people's lives, particularly pernicious problems arise due to self-reinforcing feedback loops. For example, consider the case of employers using credit scores to evaluate potential hires. You may be a good worker with a good credit score, but suddenly find yourself in financial difficulties due to a misfortune outside of your control. As you miss payments on your bills, your credit score suffers, and you will be less likely to find work. Joblessness pushes you toward poverty, which further worsens your scores, making it even harder to find employment [87]. It's a downward spiral due to poisonous assumptions, hidden behind a camouflage of mathematical rigor and data.

We can't always predict when such feedback loops happen. However, many consequences can be predicted by thinking about the entire system (not just the computerized parts, but also the people interacting with it)—an approach known as *systems thinking* [92]. We can try to understand how a data analysis system responds to different behaviors, structures, or characteristics. Does the system reinforce and amplify existing differences between people (e.g., making the rich richer or the poor poorer), or does it try to combat injustice? And even with the best intentions, we must beware of unintended consequences.

Privacy and Tracking

Besides the problems of predictive analytics—i.e., using data to make automated decisions about people—there are ethical problems with data collection itself. What is the relationship between the organizations collecting data and the people whose data is being collected?

When a system only stores data that a user has explicitly entered, because they want the system to store and process it in a certain way, the system is performing a service for the user: the user is the customer. But when a user's activity is tracked and logged as a side effect of other things they are doing, the relationship is less clear. The service

no longer just does what the user tells it to do, but it takes on interests of its own, which may conflict with the user's interests.

Tracking behavioral data has become increasingly important for user-facing features of many online services: tracking which search results are clicked helps improve the ranking of search results; recommending "people who liked X also liked Y" helps users discover interesting and useful things; A/B tests and user flow analysis can help indicate how a user interface might be improved. Those features require some amount of tracking of user behavior, and users benefit from them.

However, depending on a company's business model, tracking often doesn't stop there. If the service is funded through advertising, the advertisers are the actual customers, and the users' interests take second place. Tracking data becomes more detailed, analyses become further-reaching, and data is retained for a long time in order to build up detailed profiles of each person for marketing purposes.

Now the relationship between the company and the user whose data is being collected starts looking quite different. The user is given a free service and is coaxed into engaging with it as much as possible. The tracking of the user serves not primarily that individual, but rather the needs of the advertisers who are funding the service. I think this relationship can be appropriately described with a word that has more sinister connotations: *surveillance*.

Surveillance

As a thought experiment, try replacing the word *data* with *surveillance*, and observe if common phrases still sound so good [93]. How about this: "In our surveillance-driven organization we collect real-time surveillance streams and store them in our surveillance warehouse. Our surveillance scientists use advanced analytics and surveillance processing in order to derive new insights."

This thought experiment is unusually polemic for this book, *Designing Surveillance-Intensive Applications*, but I think that strong words are needed to emphasize this point. In our attempts to make software "eat the world" [94], we have built the greatest mass surveillance infrastructure the world has ever seen. Rushing toward an Internet of Things, we are rapidly approaching a world in which every inhabited space contains at least one internet-connected microphone, in the form of smartphones, smart TVs, voice-controlled assistant devices, baby monitors, and even children's toys that use cloud-based speech recognition. Many of these devices have a terrible security record [95].

Even the most totalitarian and repressive regimes could only dream of putting a microphone in every room and forcing every person to constantly carry a device capable of tracking their location and movements. Yet we apparently voluntarily, even enthusiastically, throw ourselves into this world of total surveillance. The differ-

ence is just that the data is being collected by corporations rather than government agencies [96].

Not all data collection necessarily qualifies as surveillance, but examining it as such can help us understand our relationship with the data collector. Why are we seemingly happy to accept surveillance by corporations? Perhaps you feel you have nothing to hide—in other words, you are totally in line with existing power structures, you are not a marginalized minority, and you needn't fear persecution [97]. Not everyone is so fortunate. Or perhaps it's because the purpose seems benign—it's not overt coercion and conformance, but merely better recommendations and more personalized marketing. However, combined with the discussion of predictive analytics from the last section, that distinction seems less clear.

We are already seeing car insurance premiums linked to tracking devices in cars, and health insurance coverage that depends on people wearing a fitness tracking device. When surveillance is used to determine things that hold sway over important aspects of life, such as insurance coverage or employment, it starts to appear less benign. Moreover, data analysis can reveal surprisingly intrusive things: for example, the movement sensor in a smartwatch or fitness tracker can be used to work out what you are typing (for example, passwords) with fairly good accuracy [98]. And algorithms for analysis are only going to get better.

Consent and freedom of choice

We might assert that users voluntarily choose to use a service that tracks their activity, and they have agreed to the terms of service and privacy policy, so they consent to data collection. We might even claim that users are receiving a valuable service in return for the data they provide, and that the tracking is necessary in order to provide the service. Undoubtedly, social networks, search engines, and various other free online services are valuable to users—but there are problems with this argument.

Users have little knowledge of what data they are feeding into our databases, or how it is retained and processed—and most privacy policies do more to obscure than to illuminate. Without understanding what happens to their data, users cannot give any meaningful consent. Often, data from one user also says things about other people who are not users of the service and who have not agreed to any terms. The derived datasets that we discussed in this part of the book—in which data from the entire user base may have been combined with behavioral tracking and external data sources—are precisely the kinds of data of which users cannot have any meaningful understanding.

Moreover, data is extracted from users through a one-way process, not a relationship with true reciprocity, and not a fair value exchange. There is no dialog, no option for users to negotiate how much data they provide and what service they receive in

return: the relationship between the service and the user is very asymmetric and one-sided. The terms are set by the service, not by the user [99].

For a user who does not consent to surveillance, the only real alternative is simply not to use a service. But this choice is not free either: if a service is so popular that it is "regarded by most people as essential for basic social participation" [99], then it is not reasonable to expect people to opt out of this service—using it is *de facto* mandatory. For example, in most Western social communities, it has become the norm to carry a smartphone, to use Facebook for socializing, and to use Google for finding information. Especially when a service has network effects, there is a social cost to people choosing *not* to use it.

Declining to use a service due to its tracking of users is only an option for the small number of people who are privileged enough to have the time and knowledge to understand its privacy policy, and who can afford to potentially miss out on social participation or professional opportunities that may have arisen if they had participated in the service. For people in a less privileged position, there is no meaningful freedom of choice: surveillance becomes inescapable.

Privacy and use of data

Sometimes people claim that "privacy is dead" on the grounds that some users are willing to post all sorts of things about their lives to social media, sometimes mundane and sometimes deeply personal. However, this claim is false and rests on a misunderstanding of the word *privacy*.

Having privacy does not mean keeping everything secret; it means having the freedom to choose which things to reveal to whom, what to make public, and what to keep secret. The right to privacy is a decision right: it enables each person to decide where they want to be on the spectrum between secrecy and transparency in each situation [99]. It is an important aspect of a person's freedom and autonomy.

When data is extracted from people through surveillance infrastructure, privacy rights are not necessarily eroded, but rather transferred to the data collector. Companies that acquire data essentially say "trust us to do the right thing with your data," which means that the right to decide what to reveal and what to keep secret is transferred from the individual to the company.

The companies in turn choose to keep much of the outcome of this surveillance secret, because to reveal it would be perceived as creepy, and would harm their business model (which relies on knowing more about people than other companies do). Intimate information about users is only revealed indirectly, for example in the form of tools for targeting advertisements to specific groups of people (such as those suffering from a particular illness).

Even if particular users cannot be personally reidentified from the bucket of people targeted by a particular ad, they have lost their agency about the disclosure of some intimate information, such as whether they suffer from some illness. It is not the user who decides what is revealed to whom on the basis of their personal preferences—it is the company that exercises the privacy right with the goal of maximizing its profit.

Many companies have a goal of not being *perceived* as creepy—avoiding the question of how intrusive their data collection actually is, and instead focusing on managing user perceptions. And even these perceptions are often managed poorly: for example, something may be factually correct, but if it triggers painful memories, the user may not want to be reminded about it [100]. With any kind of data we should expect the possibility that it is wrong, undesirable, or inappropriate in some way, and we need to build mechanisms for handling those failures. Whether something is "undesirable" or "inappropriate" is of course down to human judgment; algorithms are oblivious to such notions unless we explicitly program them to respect human needs. As engineers of these systems we must be humble, accepting and planning for such failings.

Privacy settings that allow a user of an online service to control which aspects of their data other users can see are a starting point for handing back some control to users. However, regardless of the setting, the service itself still has unfettered access to the data, and is free to use it in any way permitted by the privacy policy. Even if the service promises not to sell the data to third parties, it usually grants itself unrestricted rights to process and analyze the data internally, often going much further than what is overtly visible to users.

This kind of large-scale transfer of privacy rights from individuals to corporations is historically unprecedented [99]. Surveillance has always existed, but it used to be expensive and manual, not scalable and automated. Trust relationships have always existed, for example between a patient and their doctor, or between a defendant and their attorney—but in these cases the use of data has been strictly governed by ethical, legal, and regulatory constraints. Internet services have made it much easier to amass huge amounts of sensitive information without meaningful consent, and to use it at massive scale without users understanding what is happening to their private data.

Data as assets and power

Since behavioral data is a byproduct of users interacting with a service, it is sometimes called "data exhaust"—suggesting that the data is worthless waste material. Viewed this way, behavioral and predictive analytics can be seen as a form of recycling that extracts value from data that would have otherwise been thrown away.

More correct would be to view it the other way round: from an economic point of view, if targeted advertising is what pays for a service, then behavioral data about people is the service's core asset. In this case, the application with which the user interacts is merely a means to lure users into feeding more and more personal infor-

mation into the surveillance infrastructure [99]. The delightful human creativity and social relationships that often find expression in online services are cynically exploited by the data extraction machine.

The assertion that personal data is a valuable asset is supported by the existence of data brokers, a shady industry operating in secrecy, purchasing, aggregating, analyzing, inferring, and reselling intrusive personal data about people, mostly for marketing purposes [90]. Startups are valued by their user numbers, by "eyeballs"—i.e., by their surveillance capabilities.

Because the data is valuable, many people want it. Of course companies want it—that's why they collect it in the first place. But governments want to obtain it too: by means of secret deals, coercion, legal compulsion, or simply stealing it [101]. When a company goes bankrupt, the personal data it has collected is one of the assets that get sold. Moreover, the data is difficult to secure, so breaches happen disconcertingly often [102].

These observations have led critics to saying that data is not just an asset, but a "toxic asset" [101], or at least "hazardous material" [103]. Even if we think that we are capable of preventing abuse of data, whenever we collect data, we need to balance the benefits with the risk of it falling into the wrong hands: computer systems may be compromised by criminals or hostile foreign intelligence services, data may be leaked by insiders, the company may fall into the hands of unscrupulous management that does not share our values, or the country may be taken over by a regime that has no qualms about compelling us to hand over the data.

When collecting data, we need to consider not just today's political environment, but all possible future governments. There is no guarantee that every government elected in future will respect human rights and civil liberties, so "it is poor civic hygiene to install technologies that could someday facilitate a police state" [104].

"Knowledge is power," as the old adage goes. And furthermore, "to scrutinize others while avoiding scrutiny oneself is one of the most important forms of power" [105]. This is why totalitarian governments want surveillance: it gives them the power to control the population. Although today's technology companies are not overtly seeking political power, the data and knowledge they have accumulated nevertheless gives them a lot of power over our lives, much of which is surreptitious, outside of public oversight [106].

Remembering the Industrial Revolution

Data is the defining feature of the information age. The internet, data storage, processing, and software-driven automation are having a major impact on the global economy and human society. As our daily lives and social organization have changed in the past decade, and will probably continue to radically change in the coming decades, comparisons to the Industrial Revolution come to mind [87, 96].

The Industrial Revolution came about through major technological and agricultural advances, and it brought sustained economic growth and significantly improved living standards in the long run. Yet it also came with major problems: pollution of the air (due to smoke and chemical processes) and the water (from industrial and human waste) was dreadful. Factory owners lived in splendor, while urban workers often lived in very poor housing and worked long hours in harsh conditions. Child labor was common, including dangerous and poorly paid work in mines.

It took a long time before safeguards were established, such as environmental protection regulations, safety protocols for workplaces, outlawing child labor, and health inspections for food. Undoubtedly the cost of doing business increased when factories could no longer dump their waste into rivers, sell tainted foods, or exploit workers. But society as a whole benefited hugely, and few of us would want to return to a time before those regulations [87].

Just as the Industrial Revolution had a dark side that needed to be managed, our transition to the information age has major problems that we need to confront and solve. I believe that the collection and use of data is one of those problems. In the words of Bruce Schneier [96]:

> Data is the pollution problem of the information age, and protecting privacy is the environmental challenge. Almost all computers produce information. It stays around, festering. How we deal with it—how we contain it and how we dispose of it—is central to the health of our information economy. Just as we look back today at the early decades of the industrial age and wonder how our ancestors could have ignored pollution in their rush to build an industrial world, our grandchildren will look back at us during these early decades of the information age and judge us on how we addressed the challenge of data collection and misuse.
>
> We should try to make them proud.

Legislation and self-regulation

Data protection laws might be able to help preserve individuals' rights. For example, the 1995 European Data Protection Directive states that personal data must be "collected for specified, explicit and legitimate purposes and not further processed in a way incompatible with those purposes," and furthermore that data must be "adequate, relevant and not excessive in relation to the purposes for which they are collected" [107].

However, it is doubtful whether this legislation is effective in today's internet context [108]. These rules run directly counter to the philosophy of Big Data, which is to maximize data collection, to combine it with other datasets, to experiment and to explore in order to generate new insights. Exploration means using data for unforeseen purposes, which is the opposite of the "specified and explicit" purposes for which the user gave their consent (if we can meaningfully speak of consent at all [109]). Updated regulations are now being developed [89].

Companies that collect lots of data about people oppose regulation as being a burden and a hindrance to innovation. To some extent that opposition is justified. For example, when sharing medical data, there are clear risks to privacy, but there are also potential opportunities: how many deaths could be prevented if data analysis was able to help us achieve better diagnostics or find better treatments [110]? Overregulation may prevent such breakthroughs. It is difficult to balance such potential opportunities with the risks [105].

Fundamentally, I think we need a culture shift in the tech industry with regard to personal data. We should stop regarding users as metrics to be optimized, and remember that they are humans who deserve respect, dignity, and agency. We should self-regulate our data collection and processing practices in order to establish and maintain the trust of the people who depend on our software [111]. And we should take it upon ourselves to educate end users about how their data is used, rather than keeping them in the dark.

We should allow each individual to maintain their privacy—i.e., their control over own data—and not steal that control from them through surveillance. Our individual right to control our data is like the natural environment of a national park: if we don't explicitly protect and care for it, it will be destroyed. It will be the tragedy of the commons, and we will all be worse off for it. Ubiquitous surveillance is not inevitable —we are still able to stop it.

How exactly we might achieve this is an open question. To begin with, we should not retain data forever, but purge it as soon as it is no longer needed [111, 112]. Purging data runs counter to the idea of immutability (see "Limitations of immutability" on page 463), but that issue can be solved. A promising approach I see is to enforce access control through cryptographic protocols, rather than merely by policy [113, 114]. Overall, culture and attitude changes will be necessary.

Summary

In this chapter we discussed new approaches to designing data systems, and I included my personal opinions and speculations about the future. We started with the observation that there is no one single tool that can efficiently serve all possible use cases, and so applications necessarily need to compose several different pieces of software to accomplish their goals. We discussed how to solve this *data integration* problem by using batch processing and event streams to let data changes flow between different systems.

In this approach, certain systems are designated as systems of record, and other data is derived from them through transformations. In this way we can maintain indexes, materialized views, machine learning models, statistical summaries, and more. By making these derivations and transformations asynchronous and loosely coupled, a

problem in one area is prevented from spreading to unrelated parts of the system, increasing the robustness and fault-tolerance of the system as a whole.

Expressing dataflows as transformations from one dataset to another also helps evolve applications: if you want to change one of the processing steps, for example to change the structure of an index or cache, you can just rerun the new transformation code on the whole input dataset in order to rederive the output. Similarly, if something goes wrong, you can fix the code and reprocess the data in order to recover.

These processes are quite similar to what databases already do internally, so we recast the idea of dataflow applications as *unbundling* the components of a database, and building an application by composing these loosely coupled components.

Derived state can be updated by observing changes in the underlying data. Moreover, the derived state itself can further be observed by downstream consumers. We can even take this dataflow all the way through to the end-user device that is displaying the data, and thus build user interfaces that dynamically update to reflect data changes and continue to work offline.

Next, we discussed how to ensure that all of this processing remains correct in the presence of faults. We saw that strong integrity guarantees can be implemented scalably with asynchronous event processing, by using end-to-end operation identifiers to make operations idempotent and by checking constraints asynchronously. Clients can either wait until the check has passed, or go ahead without waiting but risk having to apologize about a constraint violation. This approach is much more scalable and robust than the traditional approach of using distributed transactions, and fits with how many business processes work in practice.

By structuring applications around dataflow and checking constraints asynchronously, we can avoid most coordination and create systems that maintain integrity but still perform well, even in geographically distributed scenarios and in the presence of faults. We then talked a little about using audits to verify the integrity of data and detect corruption.

Finally, we took a step back and examined some ethical aspects of building data-intensive applications. We saw that although data can be used to do good, it can also do significant harm: making justifying decisions that seriously affect people's lives and are difficult to appeal against, leading to discrimination and exploitation, normalizing surveillance, and exposing intimate information. We also run the risk of data breaches, and we may find that a well-intentioned use of data has unintended consequences.

As software and data are having such a large impact on the world, we engineers must remember that we carry a responsibility to work toward the kind of world that we want to live in: a world that treats people with humanity and respect. I hope that we can work together toward that goal.

References

[1] Rachid Belaid: "Postgres Full-Text Search is Good Enough!," *rachbelaid.com*, July 13, 2015.

[2] Philippe Ajoux, Nathan Bronson, Sanjeev Kumar, et al.: "Challenges to Adopting Stronger Consistency at Scale," at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.

[3] Pat Helland and Dave Campbell: "Building on Quicksand," at *4th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2009.

[4] Jessica Kerr: "Provenance and Causality in Distributed Systems," *blog.jessitron.com*, September 25, 2016.

[5] Kostas Tzoumas: "Batch Is a Special Case of Streaming," *data-artisans.com*, September 15, 2015.

[6] Shinji Kim and Robert Blafford: "Stream Windowing Performance Analysis: Concord and Spark Streaming," *concord.io*, July 6, 2016.

[7] Jay Kreps: "The Log: What Every Software Engineer Should Know About Real-Time Data's Unifying Abstraction," *engineering.linkedin.com*, December 16, 2013.

[8] Pat Helland: "Life Beyond Distributed Transactions: An Apostate's Opinion," at *3rd Biennial Conference on Innovative Data Systems Research* (CIDR), January 2007.

[9] "Great Western Railway (1835–1948)," Network Rail Virtual Archive, *networkrail.co.uk*.

[10] Jacqueline Xu: "Online Migrations at Scale," *stripe.com*, February 2, 2017.

[11] Molly Bartlett Dishman and Martin Fowler: "Agile Architecture," at *O'Reilly Software Architecture Conference*, March 2015.

[12] Nathan Marz and James Warren: *Big Data: Principles and Best Practices of Scalable Real-Time Data Systems*. Manning, 2015. ISBN: 978-1-617-29034-3

[13] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin: "Summingbird: A Framework for Integrating Batch and Online MapReduce Computations," at *40th International Conference on Very Large Data Bases* (VLDB), September 2014.

[14] Jay Kreps: "Questioning the Lambda Architecture," *oreilly.com*, July 2, 2014.

[15] Raul Castro Fernandez, Peter Pietzuch, Jay Kreps, et al.: "Liquid: Unifying Nearline and Offline Big Data Integration," at *7th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2015.

[16] Dennis M. Ritchie and Ken Thompson: "The UNIX Time-Sharing System," *Communications of the ACM*, volume 17, number 7, pages 365–375, July 1974. doi:10.1145/361011.361061

[17] Eric A. Brewer and Joseph M. Hellerstein: "CS262a: Advanced Topics in Computer Systems," lecture notes, University of California, Berkeley, *cs.berkeley.edu*, August 2011.

[18] Michael Stonebraker: "The Case for Polystores," *wp.sigmod.org*, July 13, 2015.

[19] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, et al.: "The BigDAWG Polystore System," *ACM SIGMOD Record*, volume 44, number 2, pages 11–16, June 2015. doi:10.1145/2814710.2814713

[20] Patrycja Dybka: "Foreign Data Wrappers for PostgreSQL," *vertabelo.com*, March 24, 2015.

[21] David B. Lomet, Alan Fekete, Gerhard Weikum, and Mike Zwilling: "Unbundling Transaction Services in the Cloud," at *4th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2009.

[22] Martin Kleppmann and Jay Kreps: "Kafka, Samza and the Unix Philosophy of Distributed Data," *IEEE Data Engineering Bulletin*, volume 38, number 4, pages 4–14, December 2015.

[23] John Hugg: "Winning Now and in the Future: Where VoltDB Shines," *voltdb.com*, March 23, 2016.

[24] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard: "Differential Dataflow," at *6th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2013.

[25] Derek G Murray, Frank McSherry, Rebecca Isaacs, et al.: "Naiad: A Timely Dataflow System," at *24th ACM Symposium on Operating Systems Principles* (SOSP), pages 439–455, November 2013. doi:10.1145/2517349.2522738

[26] Gwen Shapira: "We have a bunch of customers who are implementing 'database inside-out' concept and they all ask 'is anyone else doing it? are we crazy?'" *twitter.com*, July 28, 2016.

[27] Martin Kleppmann: "Turning the Database Inside-out with Apache Samza," at *Strange Loop*, September 2014.

[28] Peter Van Roy and Seif Haridi: *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004. ISBN: 978-0-262-22069-9

[29] "Juttle Documentation," *juttle.github.io*, 2016.

[30] Evan Czaplicki and Stephen Chong: "Asynchronous Functional Reactive Programming for GUIs," at *34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), June 2013. doi:10.1145/2491956.2462161

[31] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter: "A Survey on Reactive Programming," *ACM Computing Surveys*, volume 45, number 4, pages 1–34, August 2013. doi:10.1145/2501654.2501666

[32] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak: "Consistency Analysis in Bloom: A CALM and Collected Approach," at *5th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2011.

[33] Felienne Hermans: "Spreadsheets Are Code," at *Code Mesh*, November 2015.

[34] Dan Bricklin and Bob Frankston: "VisiCalc: Information from Its Creators," *danbricklin.com*.

[35] D. Sculley, Gary Holt, Daniel Golovin, et al.: "Machine Learning: The High-Interest Credit Card of Technical Debt," at *NIPS Workshop on Software Engineering for Machine Learning* (SE4ML), December 2014.

[36] Peter Bailis, Alan Fekete, Michael J Franklin, et al.: "Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity," at *ACM International Conference on Management of Data* (SIGMOD), June 2015. doi:10.1145/2723372.2737784

[37] Guy Steele: "Re: Need for Macros (Was Re: Icon)," email to *ll1-discuss* mailing list, *people.csail.mit.edu*, December 24, 2001.

[38] David Gelernter: "Generative Communication in Linda," *ACM Transactions on Programming Languages and Systems* (TOPLAS), volume 7, number 1, pages 80–112, January 1985. doi:10.1145/2363.2433

[39] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec: "The Many Faces of Publish/Subscribe," *ACM Computing Surveys*, volume 35, number 2, pages 114–131, June 2003. doi:10.1145/857076.857078

[40] Ben Stopford: "Microservices in a Streaming World," at *QCon London*, March 2016.

[41] Christian Posta: "Why Microservices Should Be Event Driven: Autonomy vs Authority," *blog.christianposta.com*, May 27, 2016.

[42] Alex Feyerke: "Say Hello to Offline First," *hood.ie*, November 5, 2013.

[43] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich: "Global Sequence Protocol: A Robust Abstraction for Replicated Shared State," at

29th European Conference on Object-Oriented Programming (ECOOP), July 2015. doi:10.4230/LIPIcs.ECOOP.2015.568

[44] Mark Soper: "Clearing Up React Data Management Confusion with Flux, Redux, and Relay," *medium.com*, December 3, 2015.

[45] Eno Thereska, Damian Guy, Michael Noll, and Neha Narkhede: "Unifying Stream Processing and Interactive Queries in Apache Kafka," *confluent.io*, October 26, 2016.

[46] Frank McSherry: "Dataflow as Database," *github.com*, July 17, 2016.

[47] Peter Alvaro: "I See What You Mean," at *Strange Loop*, September 2015.

[48] Nathan Marz: "Trident: A High-Level Abstraction for Realtime Computation," *blog.twitter.com*, August 2, 2012.

[49] Edi Bice: "Low Latency Web Scale Fraud Prevention with Apache Samza, Kafka and Friends," at *Merchant Risk Council MRC Vegas Conference*, March 2016.

[50] Charity Majors: "The Accidental DBA," *charity.wtf*, October 2, 2016.

[51] Arthur J. Bernstein, Philip M. Lewis, and Shiyong Lu: "Semantic Conditions for Correctness at Different Isolation Levels," at *16th International Conference on Data Engineering* (ICDE), February 2000. doi:10.1109/ICDE.2000.839387

[52] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan: "Automating the Detection of Snapshot Isolation Anomalies," at *33rd International Conference on Very Large Data Bases* (VLDB), September 2007.

[53] Kyle Kingsbury: Jepsen blog post series, *aphyr.com*, 2013–2016.

[54] Michael Jouravlev: "Redirect After Post," *theserverside.com*, August 1, 2004.

[55] Jerome H. Saltzer, David P. Reed, and David D. Clark: "End-to-End Arguments in System Design," *ACM Transactions on Computer Systems*, volume 2, number 4, pages 277–288, November 1984. doi:10.1145/357401.357402

[56] Peter Bailis, Alan Fekete, Michael J. Franklin, et al.: "Coordination-Avoiding Database Systems," *Proceedings of the VLDB Endowment*, volume 8, number 3, pages 185–196, November 2014.

[57] Alex Yarmula: "Strong Consistency in Manhattan," *blog.twitter.com*, March 17, 2016.

[58] Douglas B Terry, Marvin M Theimer, Karin Petersen, et al.: "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System," at *15th ACM Symposium on Operating Systems Principles* (SOSP), pages 172–182, December 1995. doi:10.1145/224056.224070

[59] Jim Gray: "The Transaction Concept: Virtues and Limitations," at *7th International Conference on Very Large Data Bases* (VLDB), September 1981.

[60] Hector Garcia-Molina and Kenneth Salem: "Sagas," at *ACM International Conference on Management of Data* (SIGMOD), May 1987. doi:10.1145/38713.38742

[61] Pat Helland: "Memories, Guesses, and Apologies," *blogs.msdn.com*, May 15, 2007.

[62] Yoongu Kim, Ross Daly, Jeremie Kim, et al.: "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," at *41st Annual International Symposium on Computer Architecture* (ISCA), June 2014. doi:10.1145/2678373.2665726

[63] Mark Seaborn and Thomas Dullien: "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," *googleprojectzero.blogspot.co.uk*, March 9, 2015.

[64] Jim N. Gray and Catharine van Ingen: "Empirical Measurements of Disk Failure Rates and Error Rates," Microsoft Research, MSR-TR-2005-166, December 2005.

[65] Annamalai Gurusami and Daniel Price: "Bug #73170: Duplicates in Unique Secondary Index Because of Fix of Bug#68021," *bugs.mysql.com*, July 2014.

[66] Gary Fredericks: "Postgres Serializability Bug," *github.com*, September 2015.

[67] Xiao Chen: "HDFS DataNode Scanners and Disk Checker Explained," *blog.cloudera.com*, December 20, 2016.

[68] Jay Kreps: "Getting Real About Distributed System Reliability," *blog.empathybox.com*, March 19, 2012.

[69] Martin Fowler: "The LMAX Architecture," *martinfowler.com*, July 12, 2011.

[70] Sam Stokes: "Move Fast with Confidence," *blog.samstokes.co.uk*, July 11, 2016.

[71] "Sawtooth Lake Documentation," Intel Corporation, *intelledger.github.io*, 2016.

[72] Richard Gendal Brown: "Introducing R3 Corda™: A Distributed Ledger Designed for Financial Services," *gendal.me*, April 5, 2016.

[73] Trent McConaghy, Rodolphe Marques, Andreas Müller, et al.: "BigchainDB: A Scalable Blockchain Database," *bigchaindb.com*, June 8, 2016.

[74] Ralph C. Merkle: "A Digital Signature Based on a Conventional Encryption Function," at *CRYPTO '87*, August 1987. doi:10.1007/3-540-48184-2_32

[75] Ben Laurie: "Certificate Transparency," *ACM Queue*, volume 12, number 8, pages 10-19, August 2014. doi:10.1145/2668152.2668154

[76] Mark D. Ryan: "Enhanced Certificate Transparency and End-to-End Encrypted Mail," at *Network and Distributed System Security Symposium* (NDSS), February 2014. doi:10.14722/ndss.2014.23379

[77] "Software Engineering Code of Ethics and Professional Practice," Association for Computing Machinery, *acm.org*, 1999.

[78] François Chollet: "Software development is starting to involve important ethical choices," *twitter.com*, October 30, 2016.

[79] Igor Perisic: "Making Hard Choices: The Quest for Ethics in Machine Learning," *engineering.linkedin.com*, November 2016.

[80] John Naughton: "Algorithm Writers Need a Code of Conduct," *theguardian.com*, December 6, 2015.

[81] Logan Kugler: "What Happens When Big Data Blunders?," *Communications of the ACM*, volume 59, number 6, pages 15–16, June 2016. doi:10.1145/2911975

[82] Bill Davidow: "Welcome to Algorithmic Prison," *theatlantic.com*, February 20, 2014.

[83] Don Peck: "They're Watching You at Work," *theatlantic.com*, December 2013.

[84] Leigh Alexander: "Is an Algorithm Any Less Racist Than a Human?" *theguardian.com*, August 3, 2016.

[85] Jesse Emspak: "How a Machine Learns Prejudice," *scientificamerican.com*, December 29, 2016.

[86] Maciej Cegłowski: "The Moral Economy of Tech," *idlewords.com*, June 2016.

[87] Cathy O'Neil: *Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy*. Crown Publishing, 2016. ISBN: 978-0-553-41881-1

[88] Julia Angwin: "Make Algorithms Accountable," *nytimes.com*, August 1, 2016.

[89] Bryce Goodman and Seth Flaxman: "European Union Regulations on Algorithmic Decision-Making and a 'Right to Explanation'," *arXiv:1606.08813*, August 31, 2016.

[90] "A Review of the Data Broker Industry: Collection, Use, and Sale of Consumer Data for Marketing Purposes," Staff Report, *United States Senate Committee on Commerce, Science, and Transportation*, *commerce.senate.gov*, December 2013.

[91] Olivia Solon: "Facebook's Failure: Did Fake News and Polarized Politics Get Trump Elected?" *theguardian.com*, November 10, 2016.

[92] Donella H. Meadows and Diana Wright: *Thinking in Systems: A Primer*. Chelsea Green Publishing, 2008. ISBN: 978-1-603-58055-7

[93] Daniel J. Bernstein: "Listening to a 'big data'/'data science' talk," *twitter.com*, May 12, 2015.

[94] Marc Andreessen: "Why Software Is Eating the World," *The Wall Street Journal*, 20 August 2011.

[95] J. M. Porup: "'Internet of Things' Security Is Hilariously Broken and Getting Worse," *arstechnica.com*, January 23, 2016.

[96] Bruce Schneier: *Data and Goliath: The Hidden Battles to Collect Your Data and Control Your World*. W. W. Norton, 2015. ISBN: 978-0-393-35217-7

[97] The Grugq: "Nothing to Hide," *grugq.tumblr.com*, April 15, 2016.

[98] Tony Beltramelli: "Deep-Spying: Spying Using Smartwatch and Deep Learning," Masters Thesis, IT University of Copenhagen, December 2015. Available at *arxiv.org/abs/1512.05616*

[99] Shoshana Zuboff: "Big Other: Surveillance Capitalism and the Prospects of an Information Civilization," *Journal of Information Technology*, volume 30, number 1, pages 75–89, April 2015. doi:10.1057/jit.2015.5

[100] Carina C. Zona: "Consequences of an Insightful Algorithm," at *GOTO Berlin*, November 2016.

[101] Bruce Schneier: "Data Is a Toxic Asset, So Why Not Throw It Out?," *schneier.com*, March 1, 2016.

[102] John E. Dunn: "The UK's 15 Most Infamous Data Breaches," *techworld.com*, November 18, 2016.

[103] Cory Scott: "Data is not toxic - which implies no benefit - but rather hazardous material, where we must balance need vs. want," *twitter.com*, March 6, 2016.

[104] Bruce Schneier: "Mission Creep: When Everything Is Terrorism," *schneier.com*, July 16, 2013.

[105] Lena Ulbricht and Maximilian von Grafenstein: "Big Data: Big Power Shifts?," *Internet Policy Review*, volume 5, number 1, March 2016. doi:10.14763/2016.1.406

[106] Ellen P. Goodman and Julia Powles: "Facebook and Google: Most Powerful and Secretive Empires We've Ever Known," *theguardian.com*, September 28, 2016.

[107] Directive 95/46/EC on the protection of individuals with regard to the processing of personal data and on the free movement of such data, Official Journal of the European Communities No. L 281/31, *eur-lex.europa.eu*, November 1995.

[108] Brendan Van Alsenoy: "Regulating Data Protection: The Allocation of Responsibility and Risk Among Actors Involved in Personal Data Processing," Thesis, KU Leuven Centre for IT and IP Law, August 2016.

[109] Michiel Rhoen: "Beyond Consent: Improving Data Protection Through Consumer Protection Law," *Internet Policy Review*, volume 5, number 1, March 2016. doi:10.14763/2016.1.404

[110] Jessica Leber: "Your Data Footprint Is Affecting Your Life in Ways You Can't Even Imagine," *fastcoexist.com*, March 15, 2016.

[111] Maciej Cegłowski: "Haunted by Data," *idlewords.com*, October 2015.

[112] Sam Thielman: "You Are Not What You Read: Librarians Purge User Data to Protect Privacy," *theguardian.com*, January 13, 2016.

[113] Conor Friedersdorf: "Edward Snowden's Other Motive for Leaking," *theatlantic.com*, May 13, 2014.

[114] Phillip Rogaway: "The Moral Character of Cryptographic Work," Cryptology ePrint 2015/1162, December 2015.