Chapter 13. A Philosophy of Streaming 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)

In Chapter 2 we discussed the goal of creating 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, sharding 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 the streaming/event-driven architecture ideas from Chapter 12 in particular to develop a philosophy of application development that meets those goals. This chapter is more opinionated than previous chapters, presenting a deep-dive into one particular philosophy rather than comparing multiple approaches.

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 4, we saw log-structured storage, B-trees, and column-oriented storage. When discussing replication in Chapter 6, 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". As the number of different representations of the data increases, the integration 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.

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") 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 12-4, 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 “Consensus in Practice”. 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”), 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 “Two-Phase Commit (2PC)”. 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, 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 guarantee that after a value is written, you can immediately read the up-to-date value (see “Reading Your Own Writes”). On the other hand, derived data systems are often updated asynchronously, and so they do not by default guarantee that reads are up-to-date.

Within limited environments that are willing to pay the cost of distributed transactions, they have been used successfully. However, XA has poor fault tolerance and performance characteristics (see “Distributed Transactions Across Different Systems”), which severely limit its usefulness. 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 is unlikely to happen soon.

In the absence of widespread support for a good distributed transaction protocol, 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 it is not 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).

Later in this chapter 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 shard the log across multiple machines. The order of events in two different shards is then ambiguous.
- If the servers are spread across multiple *geographically distributed* regions, 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. This implies an undefined ordering of events that originate in two different datacenters.
- When applications are deployed as *microservices*, 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. 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 "The Many Faces of Consensus"). 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.

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 shard. However, causal dependencies sometimes arise in more subtle ways.

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"). 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 "ID Generators and Logical Clocks"), 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].
- Conflict resolution algorithms (see "Automatic conflict resolution") 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, patterns for application development will emerge in the future 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

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.

As we saw in Chapter 11 and Chapter 12, 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.

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.

The principle of deterministic functions with well-defined inputs and outputs is not only good for fault tolerance, but also simplifies reasoning about the dataflows in an organization [5]. 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.

We saw in "Sharding and Secondary Indexes" that secondary indexes often cross shard boundaries. A sharded system with secondary indexes either needs to send writes to multiple shards (if the index is term-partitioned) or send reads to all shards (if the index is document-partitioned). Such cross-shard communication is also most reliable and scalable if the index is maintained asynchronously [6].

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. 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. 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 [7].

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 [8, 9].

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 [10].

Unifying batch and stream processing

An early proposal for unifying batch and stream processing was the *lambda architecture* [11], which had a number of problems [12] and has fallen out of use. More recent systems allow batch computations (reprocessing historical data) and stream computations (processing events as they arrive) to be implemented in the same system [13], an approach that is sometimes known as the *kappa architecture* [12].

Unifying batch and stream processing in one system requires the following features:

- 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, and some stream processors can read input from a distributed filesystem or object storage.
- 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. 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. 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, batch/stream processors, and operating systems all perform the same functions: they store some data, and they allow you to process and query that data [14, 15]. 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 [16]. As we saw in Chapter 11, batch processors are 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, the NoSQL movement could be interpreted as wanting to apply a Unix-esque approach of low-level abstractions to the domain of distributed OLTP data storage.

This section attempts 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;
- Materialized views, which are a kind of precomputed cache of query results;
- Replication logs, which keep copies of the data on other nodes up to date; and

- Full-text search indexes, which allow keyword search in text and which are built into some relational databases [1].

In Chapters 11 and 12, similar themes emerged. We talked about building full-text search indexes, about materialized view maintenance, and about replicating changes from a database to derived data systems using change data capture.

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"), and also very similar to bootstrapping change data capture in a streaming system (see "Initial snapshot").

Whenever you run `CREATE INDEX`, the database essentially reprocesses the existing dataset 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.

The meta-database of everything

In this light, the dataflow across an entire organization starts looking like one huge database [5]. 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 algorithms. 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, 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, 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* [17, 18]. For example, PostgreSQL's *foreign data wrapper* feature fits this pattern, as do federated query engines such as Trino, Hoptimator, and Xorq. 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 [5, 19].

The unbundled approach follows the Unix tradition of small tools that do one thing well [20], that communicate through a uniform low-level API (pipes), and that can be composed using a higher-level language (the shell) [14].

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. Keeping the writes to several storage systems in sync is the harder engineering problem, and so we will focus on it here.

The traditional approach to synchronizing writes requires distributed transactions across heterogeneous storage systems [17], which are problematic, as discussed previously. Transactions within a single storage or stream processing system are feasible, but when data crosses the boundary between different technologies, an asynchronous event log with idempotent writes is a much more robust and practicable approach.

For example, distributed transactions are used within some stream processors to achieve exactly-once semantics, 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 is a much simpler abstraction, and thus much more feasible to implement across heterogeneous systems [5].

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, 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.

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. Specialized query engines will continue to be important for particular workloads: for example, query engines in data warehouses are optimized for exploratory analytic queries and handle this kind of workload very well.

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 [21]. 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.

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.

The tools for composing data systems are getting better: Debezium can extract change streams from many databases, Kafka's protocol is becoming a de-facto standard for event streams, and incremental view maintenance engines (see "Incremental View Maintenance") make it possible to precompute and update caches of complex queries.

Designing Applications Around Dataflow

The general idea of updating derived data when its underlying data changes is nothing new. For example, spreadsheets have powerful dataflow programming capabilities [22]: 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, most data systems still have something to learn from the features that VisiCalc already had in 1979 [23]. 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 we 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 SSTable or B-tree index, which are sorted by key).
- 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 [24].

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.

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 Kubernetes, Docker, Mesos, YARN, 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.

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 [25]. As people in the functional programming community like to joke, “We believe in the separation of Church and state” [26].

NOTE

Explaining a joke usually ruins it, but here is an explanation anyway so that nobody feels left out. *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.

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.

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 have already seen this idea in change data capture, in the actor model, in triggers, and incremental view maintenance. 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.

Maintaining derived data requires the following properties, which log-based message brokers can provide:

- 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).
- Fault tolerance is essential: 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.

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 dominant style of application development involves breaking down functionality into a set of *services* that communicate via synchronous network requests such as REST APIs. 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 [27, 28]. However, the underlying communication mechanism is very different: one-directional, asynchronous message streams rather than synchronous request/response interactions.

Besides the advantages listed in "Event-Driven Architectures", such as better fault tolerance, dataflow systems can also achieve better performance than traditional REST APIs or RPC. 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 [27, 29]:

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). 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.

Not only is the dataflow approach faster, but it is also more robust to 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.

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").

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 building applications around dataflow ideas is a very promising direction to explore.

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 13-1 shows an example of updating a search index.

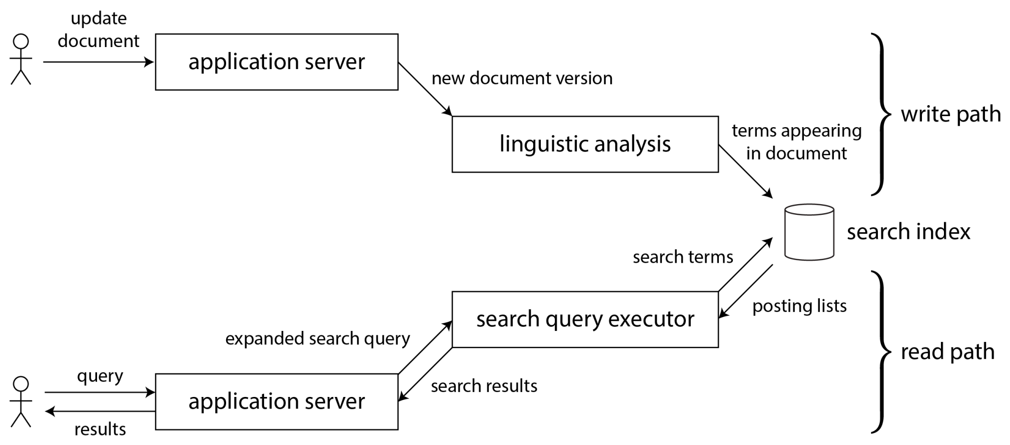

Figure 13-1. In a search index, writes (document updates) meet reads (queries).

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 13-1. 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 (or at least exponential in the number of terms in the corpus), and thus precomputing all possible search results would not be possible.

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 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 social networking example in "Case Study: Social Network Home Timelines". 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

The idea of a boundary between write and read paths is 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.

In the past, web browsers were 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, single-page JavaScript web apps now have 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.

In “Sync Engines and Local-First Software” we saw how persistent local state enables a class of applications in which users can work offline, without an internet connection, and sync with remote servers in the background when a network connection is available [30]. Since mobile devices sometimes 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.

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 [31].

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 in the browser 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 [32].

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" 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

Tools for developing stateful clients and user interfaces, such as React and Elm [33], already have the ability to update the rendered user interface in response to changes in the underlying state. 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). 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.

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 [31]. This would require effort, but it would have the advantage of making user interfaces more responsive and providing better offline support.

Reads are events too

We discussed that when a stream processor writes derived data to a store (database, cache, or index), and that store is queried, 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. This state is normally hidden inside the stream processor, but some frameworks allow it to also be queried by outside clients [34], turning the stream processor itself into a kind of simple database.

Let's 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 [35].

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 shard holding the data, just like batch and stream processors need to copartition inputs on the same key when joining.

This correspondence between serving requests and performing joins is quite fundamental [36]. A one-off read request passes through the join operator, which then immediately forgets the request; 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 requests to durable storage thus enables better tracking of causal dependencies, 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-shard data processing

For queries that only touch a single shard, 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 shards, taking advantage of the infrastructure for message routing, sharding, and joining that is already provided by stream processors.

Storm's distributed RPC feature supports this usage pattern. For example, it has been used to compute the number of people who have seen a URL on a social network—i.e., the union of the follower sets of everyone who has posted that URL [37]. As the set of users is sharded, this computation requires combining results from many shards.

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 sharded, and so collecting the scores for a particular purchase event requires a sequence of joins with differently sharded datasets [38].

The internal query execution graphs of data warehouse query engines have similar characteristics. If you need to perform this kind of multi-shard 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 [39].

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 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").

In some areas, transactions have been abandoned entirely and replaced with models that offer better performance and scalability, but much messier semantics. *Consistency* is often talked about, but poorly defined. 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 [40, 41]. 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 [42] 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, it is not the last word in making applications correct and resilient to faults. In this section we will explore 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 is an argument 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” we encountered *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*; 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 “Distributed Locks and Leases”).

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 13-1. 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 9-1).

Example 13-1. A nonidempotent transfer of money from one account to another

```
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance + 11.00 WHERE acc
UPDATE accounts SET balance = balance - 11.00 WHERE acc
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 13-1 is not idempotent, it could happen that $22 is transferred instead of the desired $11. Thus, even though Example 13-1 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 "Two-Phase Commit (2PC)") 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 network 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 [43] 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.

Uniquely identifying requests

To make the request 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 a request (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 request ID [3]. If the web browser submits the POST request twice, the two requests will have the same request ID. You can then pass that request ID all the way through to the database and check that you only ever execute one request with a given ID, as shown in Example 13-2.

Example 13-2. 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,

UPDATE accounts SET balance = balance + 11.00 WHERE acc
UPDATE accounts SET balance = balance - 11.00 WHERE acc

COMMIT;
```

Example 13-2 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”).

Besides suppressing duplicate requests, the `requests` table in Example 13-2 acts as a kind of event log, which can be useful for event sourcing or change data capture. 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 [44]:

> *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 [44]: 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 us back to the 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 it seems that we have not yet found the right abstraction.

Transactions have long been seen as a useful abstraction. As discussed in Chapter 8, 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 it is not enough.

Transactions are expensive, especially when they involve heterogeneous storage technologies (see "Distributed Transactions Across Different Systems"). 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 most application-level mechanisms do not work correctly. The consequence is lost or corrupted data.

For these reasons, 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. 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 13-2. In "Constraints and uniqueness guarantees" 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 10 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. Consensus algorithms like Raft tackle the problem of safely electing a new leader if the current leader has failed (or is believed to have failed due to a network problem), and preventing split brain.

Uniqueness checking can be scaled out by sharding based on the value that needs to be unique. For example, if you need to ensure uniqueness by request ID, as in Example 13-2, you can ensure all requests with the same request ID are routed to the same shard. If you need usernames to be unique, you can shard by hash of username.

However, asynchronous multi-leader replication is ruled out, because it could happen that different leaders concurrently accept conflicting writes, and thus the values are no longer unique. If you want to be able to immediately reject any writes that would violate the constraint, synchronous coordination is unavoidable [45].

Uniqueness in log-based messaging

A shared 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 "The Many Faces of Consensus"). 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 shard sequentially on a single thread. Thus, if the log is sharded 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 in the log. For example, in the case of several users trying to claim the same username [46]:

1. Every request for a username is encoded as a message, and appended to a shard 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 the same as the construction for achieving consensus using a shared log, which we saw in Chapter 10. It scales easily to a large request throughput by increasing the number of shards, as each shard 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 shard and processed sequentially. The definition of a conflict may depend on the application, but the stream processor can use arbitrary logic to validate a request.

Multi-shard request processing

Ensuring that an operation is executed atomically, while satisfying constraints, becomes more interesting when several shards are involved. In Example 13-2, there are potentially three shards: the one containing the request ID, the one containing the payee account, and the one containing the payer account. There is no reason why those three things should be in the same shard, 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 shards, which essentially forces it into a total order with respect to all other transactions on any of those shards. Since there is now cross-shard coordination, different shards can no longer be processed independently, so throughput is likely to suffer.

However, equivalent correctness can be achieved without cross-shard transactions using sharded logs and stream processors. Figure 13-2 shows an example of a payment transaction that needs to check whether there is sufficient money in the source account, and if so, atomically transfers some amount to a destination account while deducting fees. It works as follows [47]:

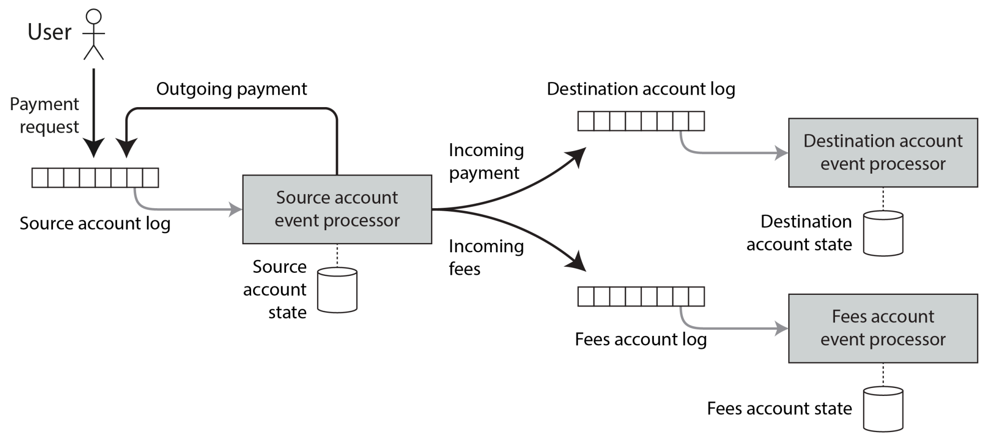

Figure 13-2. Checking whether a source account has enough money, and atomically transferring money to a destination account and a fees account, using event logs and stream processors.

1. The request to transfer money from the source account to the destination account is given a unique request ID by the user's client, and appended to a log shard based on the source account ID.
2. A stream processor reads the log of requests and maintains a database containing the state of the source account and the IDs of requests it has already processed. The contents of this database are entirely derived from the log. When the stream processor encounters a request with an ID that it has not seen before, it checks in its local database whether the source account has enough money to perform the transfer.
   If yes, it updates its local database to reserve the payment amount on the source account, and emits events to several other logs: an outgoing payment event to the log shard for the source account (its own input log), an incoming payment event to the log shard for the destination account, and an incoming payment event to the log shard for the fees account. The original request ID is included in those emitted events.
3. Eventually the outgoing payment event is delivered back to the source account processor (possibly after having received unrelated events in the meantime). The stream processor recognises based on the request ID that this is a payment it previously reserved, and it now executes the payment, again updating its local state of the source account. It ignores duplicates based on request ID.
4. The log shards for the destination and fees accounts are consumed by independent stream processing tasks. When they receive an incoming payment event, they update their local state to reflect the payment, and they deduplicate events based on request ID.

Figure 13-2 shows the three accounts as being in three separate shards, but they could just as well be in the same shards—it doesn't matter. All we need is that the events for any given account are processed strictly in log order with at-least-once semantics, and that the stream processors are deterministic.

For example, consider what happens if the source account processor crashes while processing a payment request. The output messages may or may not have been emitted before the crash occurred. When it recovers from the crash, it will process the same request again (due to at-least-once semantics), and it will make the same decision on whether to allow the payment (since it's deterministic). It will therefore emit the same output messages with the same request ID to the outgoing, incoming, and fees account shards. If they are duplicates, the downstream consumers will ignore them based on the request ID.

Atomicity in this system comes not from any transactions, but from the fact that writing the initial request event to the source account log is an atomic action. Once that one event in the log, all the downstream events will eventually be written as well—possibly after stream processors have recovered from crashes, and possibly with duplicates, but they will appear eventually.

With exactly-once semantics this example becomes easier to implement, since it ensures that the stream processor's local state is consistent with the set of messages it has processed. Thus, if it crashes and re-processes some messages, its local state is also reset to what it was before those messages were processed.

If the user in Figure 13-2 wants to find out whether their transfer was approved or not, they can subscribe to the source account log shard and wait for the outgoing payment event. In order to explicitly notify the user if the balance is insufficient, the stream processor can emit a "declined payment" event to that log shard.

By breaking down the multi-shard transaction into several differently sharded 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.

Timeliness and Integrity

A convenient property of many transactional systems is that as soon as one transaction commits, its writes are immediately visible to other transactions. This property is formalized as *strict serializability* (see “Linearizability Versus Serializability”).

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, like the user waiting for an outgoing payment or payment declined event in Figure 13-2, which depends on whether there was enough money in the source account.

In this example, the correctness of the source account balance check does not depend on whether the user making the request waits for the outcome. The waiting only has the purpose of synchronously informing the user whether or not the payment succeeded, but this notification is decoupled from the effects of processing the request.

More generally, 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”). However, that inconsistency is temporary, and will eventually be resolved simply by waiting and trying again.
>
> The CAP theorem uses consistency in the sense of linearizability, which is a strong way of achieving timeliness. Weaker timeliness properties like *read-after-write* consistency 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, 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, "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."

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. For example, a user could request a payment and then read the state of their account before the stream processor has executed the request; the user will not see the payment they just requested.

However, integrity is in fact central to streaming systems. *Exactly-once* or *effectively-once* semantics 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
- Deriving all other state updates from that single message using deterministic derivation functions, similarly to stored procedures
- 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

Loosely interpreted constraints

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

However, another thing to realize is that in many real applications there is actually a business requirement to allow violations of what you might think of as hard constraints:

- 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 [3]. Thus, the apology workflow already needs to be part of your business processes anyway in order to deal with forklift incidents, and a hard constraint on the number of items in stock might be unnecessary.

- 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 deliberately 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 systems that integrate data between different organizations, inconsistencies will inevitably arise, and correction mechanisms are necessary to handle them. As noted in "Batch Use Cases", settlement of payments between banks is an example of this.

In many business contexts, it is therefore acceptable to temporarily violate a constraint and fix it up later by apologizing. This kind of change to correct a mistake is called a *compensating transaction* [48, 49]. 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. It may well be reasonable 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 "Dealing with Conflicting Writes".

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-shard 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 [45].

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 [6]. Heterogeneous distributed transactions such as XA transactions 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 [32].

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 "System Model and Reality"): 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 in memory (see "Hardware and Software Faults"), on disk (see "Replication and Durability"), and on the network (see "Weak forms of lying"). Maybe this is something we should be paying more attention to? If you are operating at large enough scale, even very unlikely things do happen.

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: for example, past versions of MySQL have failed to correctly maintain uniqueness constraints [50] and PostgreSQL's serializable isolation level has exhibited write skew anomalies in the past [51], 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 [25].

Consistency in the sense of ACID 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", auditing is not just for financial applications. However, auditability is very 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 [52, 53].

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.

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. In the future we may see more *self-validating* or *self-auditing* systems that continually check their own integrity, rather than relying on blind trust [54].

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, 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 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, 55]. 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: 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 [56]. 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.

Blockchains such as Bitcoin or Ethereum are shared append-only logs with cryptographic consistency checks; the transactions they store are events, and smart contracts are basically stream processors. The consensus protocols they use ensure that all nodes agree on the same sequence of events. The difference to the consensus protocols of Chapter 10 is that blockchains are Byzantine fault tolerant, i.e. they still work if some of the participating nodes have corrupted data because the replicas continually check each other's integrity.

For most applications, blockchains have too high overhead to be useful. However, some of their cryptographic tools can also be used in a lighterweight context. For example, *Merkle trees* [57], are trees of hashes that can be used to efficiently prove that a record appears in some dataset (and a few other things). *Certificate transparency* uses cryptographically verified append-only logs and Merkle trees to check the validity of TLS/SSL certificates [58, 59]; it avoids needing a consensus protocol by having a single leader per log.

Integrity-checking and auditing algorithms, like those of certificate transparency and distributed ledgers, might becoming more widely used in data systems in general in the future. 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 they are nevertheless interesting.

Summary

In this chapter we discussed new approaches to designing data systems based on ideas from stream processing. 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 request 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, and observed that the techniques used by blockchains also have a similarity to event-based systems.

FOOTNOTES

REFERENCES

[1] Rachid Belaid. Postgres Full-Text Search is Good Enough! *rachbelaid.com*, July 2015. Archived at perma.cc/ZVP9-YDCB

[2] Philippe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. 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. *jessitron.com*, September 2016. Archived at perma.cc/DTD2-F8ZM

[5] Jay Kreps. The Log: What Every Software Engineer Should Know About Real-Time Data's Unifying Abstraction. *engineering.linkedin.com*, December 2013. Archived at perma.cc/2JHR-FR64

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

[7] Lionel A. Smith. The Broad Gauge Story. *Journal of the Monmouthshire Railway Society*, Summer 1985. Archived at perma.cc/DDK9-JA6X

[8] Jacqueline Xu. Online Migrations at Scale. *stripe.com*, February 2017. Archived at perma.cc/ZQY2-EAU2

[9] Flavio Santos and Robert Stephenson. Changing the Wheels on a Moving Bus — Spotify's Event Delivery Migration. *engineering.atspotify.com*, October 2021. Archived at perma.cc/5C4V-G8EV

[10] Molly Bartlett Dishman and Martin Fowler. Agile Architecture. At *O'Reilly Software Architecture Conference*, March 2015.

[11] 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

[12] Jay Kreps. Questioning the Lambda Architecture. *oreilly.com*, July 2014. Archived at perma.cc/PGH6-XUCH

[13] Raul Castro Fernandez, Peter Pietzuch, Jay Kreps, Neha Narkhede, Jun Rao, Joel Koshy, Dong Lin, Chris Riccomini, and Guozhang Wang. Liquid: Unifying Nearline and Offline Big Data Integration. At *7th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2015.

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

[15] Wes McKinney. The Road to Composable Data Systems: Thoughts on the Last 15 Years and the Future. *wesmckinney.com*, September 2023. Archived at perma.cc/J9SJ-886N

[16] Eric A. Brewer and Joseph M. Hellerstein. CS262a: Advanced Topics in Computer Systems. Lecture notes, University of California, Berkeley, *cs.berkeley.edu*, August 2011. Archived at perma.cc/TE79-LGWU

[17] Michael Stonebraker. The Case for Polystores. *wp.sigmod.org*, July 2015. Archived at perma.cc/G7J2-KR45

[18] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. The BigDAWG Polystore System. *ACM SIGMOD Record*, volume 44, issue 2, pages 11–16, June 2015. doi:10.1145/2814710.2814713

[19] 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.

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

[21] John Hugg. Winning Now and in the Future: Where Volt Active Data Shines. *voltactivedata.com*, March 2016. Archived at perma.cc/44MP-3MWM

[22] Felienne Hermans. Spreadsheets Are Code. At *Code Mesh*, November 2015.

[23] Dan Bricklin and Bob Frankston. VisiCalc: Information from Its Creators. *danbricklin.com*. Archived at archive.org

[24] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine Learning: The High-Interest Credit Card of Technical Debt. At *NIPS Workshop on Software Engineering for Machine Learning* (SE4ML), December 2014. Archived at https://perma.cc/M3MD-U7WL

[25] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 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

[26] Guy Steele. Re: Need for Macros (Was Re: Icon). email to *ll1-discuss* mailing list, *people.csail.mit.edu*, December 2001. Archived at perma.cc/K9X8-CJ65

[27] Ben Stopford. Microservices in a Streaming World. At *QCon London*, March 2016.

[28] Adam Bellemare. *Building Event-Driven Microservices, 2nd Edition*. O'Reilly Media, 2025.

[29] Christian Posta. Why Microservices Should Be Event Driven: Autonomy vs Authority. *blog.christianposta.com*, May 2016. Archived at perma.cc/E6N9-3X92

[30] Alex Feyerke. Designing Offline-First Web Apps. *alistapart.com*, December 2013. Archived at perma.cc/WH7R-S2DS

[31] Martin Kleppmann. Turning the Database Inside-out with Apache Samza. at *Strange Loop*, September 2014. Archived at perma.cc/U6E8-A9MT

[32] 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

[33] 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

[34] Eno Thereska, Damian Guy, Michael Noll, and Neha Narkhede. Unifying Stream Processing and Interactive Queries in Apache Kafka. *confluent.io*, October 2016. Archived at perma.cc/W8JG-EAZF

[35] Frank McSherry. Dataflow as Database. *github.com*, July 2016. Archived at perma.cc/384D-DUFH

[36] Peter Alvaro. I See What You Mean. At *Strange Loop*, September 2015.

[37] Nathan Marz. Trident: A High-Level Abstraction for Realtime Computation. *blog.x.com*, August 2012. Archived at archive.org

[38] Edi Bice. Low Latency Web Scale Fraud Prevention with Apache Samza, Kafka and Friends. At *Merchant Risk Council MRC Vegas Conference*, March 2016. Archived at perma.cc/T3H5-QN3R

[39] Charity Majors. The Accidental DBA. *charity.wtf*, October 2016. Archived at perma.cc/6ANP-ARB6

[40] 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

[41] 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.

[42] Kyle Kingsbury. Jespen: Distributed Systems Safety Research. *jepsen.io*.

[43] Michael Jouravlev. Redirect After Post. *theserverside.com*, August 2004. Archived at archive.org

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

[45] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination Avoidance in Database Systems. *Proceedings of the VLDB Endowment*, volume 8, issue 3, pages 185–196, November 2014. doi:10.14778/2735508.2735509

[46] Alex Yarmula. Strong Consistency in Manhattan. *blog.x.com*, March 2016. Archived at archive.org

[47] Martin Kleppmann, Alastair R. Beresford, and Boerge Svingen. Online Event Processing: Achieving consistency where distributed transactions have failed. *Communications of the ACM*, volume 62, issue 5, pages 43-49, May 2019. doi:10.1145/3312527

[48] Jim Gray. The Transaction Concept: Virtues and Limitations. At *7th International Conference on Very Large Data Bases* (VLDB), September 1981. Archived at perma.cc/8VPT-N5H6

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

[50] Annamalai Gurusami and Daniel Price. Bug #73170: Duplicates in Unique Secondary Index Because of Fix of Bug#68021. *bugs.mysql.com*, July 2014. Archived at perma.cc/P6BV-W7JJ

[51] Gary Fredericks. Postgres Serializability Bug. *github.com*, September 2015. Archived at perma.cc/N8UP-2822

[52] Xiao Chen. HDFS DataNode Scanners and Disk Checker Explained. *blog.cloudera.com*, December 2016. Archived at perma.cc/6S36-X98L

[53] Daniel Persson. How does Ceph scrubbing work? *youtube.com*, March 2022.

[54] Jay Kreps. Getting Real About Distributed System Reliability. *blog.empathybox.com*, March 2012. Archived at perma.cc/9B5Q-AEBW

[55] Martin Fowler. The LMAX Architecture. *martinfowler.com*, July 2011. Archived at perma.cc/5AV4-N6RJ

[56] Sam Stokes. Move Fast with Confidence. *five-eights.com*, July 2016. Archived at perma.cc/J8C6-DHXB

[57] 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

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

[59] 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