Chapter 12. Stream Processing

> *A complex system that works is invariably found to have evolved from a simple system that works. The inverse proposition also appears to be true: A complex system designed from scratch never works and cannot be made to work.*
>
> —John Gall, *Systemantics* (1975)

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 12th chapter of the final book. The GitHub repo for this book is *https://github.com/ept/ddia2-feedback*.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out on GitHub.

In Chapter 11 we discussed batch processing—techniques that read a set of files as input and produce a new set of output files. The output is a form of *derived data*; that is, a dataset that can be recreated by running the batch process again if necessary. We saw how this simple but powerful idea can be used to create search indexes, recommendation systems, analytics, and more.

However, one big assumption remained throughout Chapter 11: namely, that the input is bounded—i.e., of a known and finite size—so the batch process knows when it has finished reading its input. For example, the sorting operation that is central to MapReduce must read its entire input before it can start producing output: it could happen that the very last input record is the one with the lowest key, and thus needs to be the very first output record, so starting the output early is not an option.

In reality, a lot of data is unbounded because it arrives gradually over time: your users produced data yesterday and today, and they will continue to produce more data tomorrow. Unless you go out of business, this process never ends, and so the dataset is never "complete" in any meaningful way [1]. Thus, batch processors must artificially divide the data into chunks of fixed duration: for example, processing a day's worth of data at the end of every day, or processing an hour's worth of data at the end of every hour.

The problem with daily batch processes is that changes in the input are only reflected in the output a day later, which is too slow for many impatient users. To reduce the delay, we can run the processing more frequently—say, processing a second's worth of data at the end of every second—or even continuously, abandoning the fixed time slices entirely and simply processing every event as it happens. That is the idea behind *stream processing*.

In general, a "stream" refers to data that is incrementally made available over time. The concept appears in many places: in the `stdin` and `stdout` of Unix, programming languages (lazy lists) [2], filesystem APIs (such as Java's `FileInputStream`), TCP connections, delivering audio and video over the internet, and so on.

In this chapter we will look at *event streams* as a data management mechanism: the unbounded, incrementally processed counterpart to the batch data we saw in the last chapter. We will first discuss how streams are represented, stored, and transmitted over a network. In "Databases and Streams" we will investigate the relationship between streams and databases. And finally, in "Processing Streams" we will explore approaches and tools for processing those streams continually, and ways that they can be used to build applications.

Transmitting Event Streams

In the batch processing world, the inputs and outputs of a job are files (perhaps on a distributed filesystem). What does the streaming equivalent look like?

When the input is a file (a sequence of bytes), the first processing step is usually to parse it into a sequence of records. In a stream processing context, a record is more commonly known as an *event*, but it is essentially the same thing: a small, self-contained, immutable object containing the details of something that happened at some point in time. An event usually contains a timestamp indicating when it happened according to a time-of-day clock (see "Monotonic Versus Time-of-Day Clocks").

For example, the thing that happened might be an action that a user took, such as viewing a page or making a purchase. It might also originate from a machine, such as a periodic measurement from a temperature sensor, or a CPU utilization metric. In the example of "Batch Processing with Unix Tools", each line of the web server log is an event.

An event may be encoded as a text string, or JSON, or perhaps in some binary form, as discussed in Chapter 5. This encoding allows you to store an event, for example by appending it to a file, inserting it into a relational table, or writing it to a document database. It also allows you to send the event over the network to another node in order to process it.

In batch processing, a file is written once and then potentially read by multiple jobs. Analogously, in streaming terminology, an event is generated once by a *producer* (also known as a *publisher* or *sender*), and then potentially processed by multiple *consumers* (*subscribers* or *recipients*) [3]. In a filesystem, a filename identifies a set of related records; in a streaming system, related events are usually grouped together into a *topic* or *stream*.

In principle, a file or database is sufficient to connect producers and consumers: a producer writes every event that it generates to the datastore, and each consumer periodically polls the datastore to check for events that have appeared since it last ran. This is essentially what a batch process does when it processes a day's worth of data at the end of every day.

However, when moving toward continual processing with low delays, polling becomes expensive if the datastore is not designed for this kind of usage. The more often you poll, the lower the percentage of requests that return new events, and thus the higher the overheads become. Instead, it is better for consumers to be notified when new events appear.

Databases have traditionally not supported this kind of notification mechanism very well: relational databases commonly have *triggers*, which can react to a change (e.g., a row being inserted into a table), but they are very limited in what they can do and have been somewhat of an afterthought in database design [4]. Instead, specialized tools have been developed for the purpose of delivering event notifications.

Messaging Systems

A common approach for notifying consumers about new events is to use a *messaging system*: a producer sends a message containing the event, which is then pushed to consumers. We touched on these systems previously in "Event-Driven Architectures", but we will now go into more detail.

A direct communication channel like a Unix pipe or TCP connection between producer and consumer would be a simple way of implementing a messaging system. However, most messaging systems expand on this basic model. In particular, Unix pipes and TCP connect exactly one sender with one recipient, whereas a messaging system allows multiple producer nodes to send messages to the same topic and allows multiple consumer nodes to receive messages in a topic.

Within this *publish/subscribe* model, different systems take a wide range of approaches, and there is no one right answer for all purposes. To differentiate the systems, it is particularly helpful to ask the following two questions:

1. *What happens if the producers send messages faster than the consumers can process them?* Broadly speaking, there are three options: the system can drop messages, buffer messages in a queue, or apply *backpressure* (also known as *flow control*; i.e., blocking the producer from sending more messages). For example, Unix pipes and TCP use backpressure: they have a small fixed-size buffer, and if it fills up, the sender is blocked until the recipient takes data out of the buffer (see "Network congestion and queueing").
   If messages are buffered in a queue, it is important to understand what happens as that queue grows. Does the system crash if the queue no longer fits in memory, or does it write messages to disk? In the latter case, how does the disk access affect the performance of the messaging system [5], and what happens when the disk fills up [6]?
2. *What happens if nodes crash or temporarily go offline—are any messages lost?* As with databases, durability may require some combination of writing to disk and/or replication (see the sidebar "Replication and Durability"), which has a cost. If you can afford to sometimes lose

messages, you can probably get higher throughput and lower latency on the same hardware.

Whether message loss is acceptable depends very much on the application. For example, with sensor readings and metrics that are transmitted periodically, an occasional missing data point is perhaps not important, since an updated value will be sent a short time later anyway. However, beware that if a large number of messages are dropped, it may not be immediately apparent that the metrics are incorrect [7]. If you are counting events, it is more important that they are delivered reliably, since every lost message means incorrect counters.

A nice property of the batch processing systems we explored in Chapter 11 is that they provide a strong reliability guarantee: failed tasks are automatically retried, and partial output from failed tasks is automatically discarded. This means the output is the same as if no failures had occurred, which helps simplify the programming model. Later in this chapter we will examine how we can provide similar guarantees in a streaming context.

Direct messaging from producers to consumers

A number of messaging systems use direct network communication between producers and consumers without going via intermediary nodes:

- UDP multicast is widely used in the financial industry for streams such as stock market feeds, where low latency is important [8]. Although UDP itself is unreliable, application-level protocols can recover lost packets (the producer must remember packets it has sent so that it can retransmit them on demand).
- Brokerless messaging libraries such as ZeroMQ and nanomsg take a similar approach, implementing publish/subscribe messaging over TCP or IP multicast.
- Some metrics collection agents, such as StatsD [9] use unreliable UDP messaging to collect metrics from all machines on the network and monitor them. (In the StatsD protocol, counter metrics are only correct if all messages are received; using UDP makes the metrics at best approximate [10]. See also "TCP Versus UDP".)
- If the consumer exposes a service on the network, producers can make a direct HTTP or RPC request (see "Dataflow Through Services: REST and RPC") to push messages to the consumer. This is the idea behind

webhooks [11], a pattern in which a callback URL of one service is registered with another service, and it makes a request to that URL whenever an event occurs.

Although these direct messaging systems work well in the situations for which they are designed, they generally require the application code to be aware of the possibility of message loss. The faults they can tolerate are quite limited: even if the protocols detect and retransmit packets that are lost in the network, they generally assume that producers and consumers are constantly online.

If a consumer is offline, it may miss messages that were sent while it is unreachable. Some protocols allow the producer to retry failed message deliveries, but this approach may break down if the producer crashes, losing the buffer of messages that it was supposed to retry.

Message brokers

A widely used alternative is to send messages via a *message broker* (also known as a *message queue*), which is essentially a kind of database that is optimized for handling message streams [12]. It runs as a server, with producers and consumers connecting to it as clients. Producers write messages to the broker, and consumers receive them by reading them from the broker.

By centralizing the data in the broker, these systems can more easily tolerate clients that come and go (connect, disconnect, and crash), and the question of durability is moved to the broker instead. Some message brokers only keep messages in memory, while others (depending on configuration) write them to disk so that they are not lost in case of a broker crash. Faced with slow consumers, they generally allow unbounded queueing (as opposed to dropping messages or backpressure), although this choice may also depend on the configuration.

A consequence of queueing is also that consumers are generally *asynchronous*: when a producer sends a message, it normally only waits for the broker to confirm that it has buffered the message and does not wait for the message to be processed by consumers. The delivery to consumers will happen at some undetermined future point in time—often within a fraction of a second, but sometimes significantly later if there is a queue backlog.

Message brokers compared to databases

Some message brokers can even participate in two-phase commit protocols using XA or JTA (see "Distributed Transactions Across Different Systems"). This feature makes them quite similar in nature to databases, although there are still important practical differences between message brokers and databases:

- Databases usually keep data until it is explicitly deleted, whereas some message brokers automatically delete a message when it has been successfully delivered to its consumers. Such message brokers are not suitable for long-term data storage.
- Since they quickly delete messages, most message brokers assume that their working set is fairly small—i.e., the queues are short. If the broker needs to buffer a lot of messages because the consumers are slow (perhaps spilling messages to disk if they no longer fit in memory), each individual message takes longer to process, and the overall throughput may degrade [5].
- Databases often support secondary indexes and various ways of searching for data using a query language, while message brokers often support some way of subscribing to a subset of topics matching some pattern. Both are essentially ways for a client to select the portion of the data that it wants to know about, but databases typically offer much more advanced query functionality.
- When querying a database, the result is typically based on a point-in-time snapshot of the data; if another client subsequently writes something to the database that changes the query result, the first client does not find out that its prior result is now outdated (unless it repeats the query, or polls for changes). By contrast, message brokers do not support arbitrary queries and don't allow message updates once they're sent, but they do notify clients when data changes (i.e., when new messages become available).

This is the traditional view of message brokers, which is encapsulated in standards like JMS [13] and AMQP [14] and implemented in software like RabbitMQ, ActiveMQ, HornetQ, Qpid, TIBCO Enterprise Message Service, IBM MQ, Azure Service Bus, and Google Cloud Pub/Sub [15]. Although it is possible to use databases as queues, tuning them to get good performance is not straightforward [16].

Multiple consumers

When multiple consumers read messages in the same topic, two main patterns of messaging are used, as illustrated in Figure 12-1:

Load balancing

Each message is delivered to *one* of the consumers, so the consumers can share the work of processing the messages in the topic. The broker may assign messages to consumers arbitrarily. This pattern is useful when the messages are expensive to process, and so you want to be able to add consumers to parallelize the processing. (In AMQP, you can implement load balancing by having multiple clients consuming from the same queue, and in JMS it is called a *shared subscription*.)

Fan-out

Each message is delivered to *all* of the consumers. Fan-out allows several independent consumers to each "tune in" to the same broadcast of messages, without affecting each other—the streaming equivalent of having several different batch jobs that read the same input file. (This feature is provided by topic subscriptions in JMS, and exchange bindings in AMQP.)

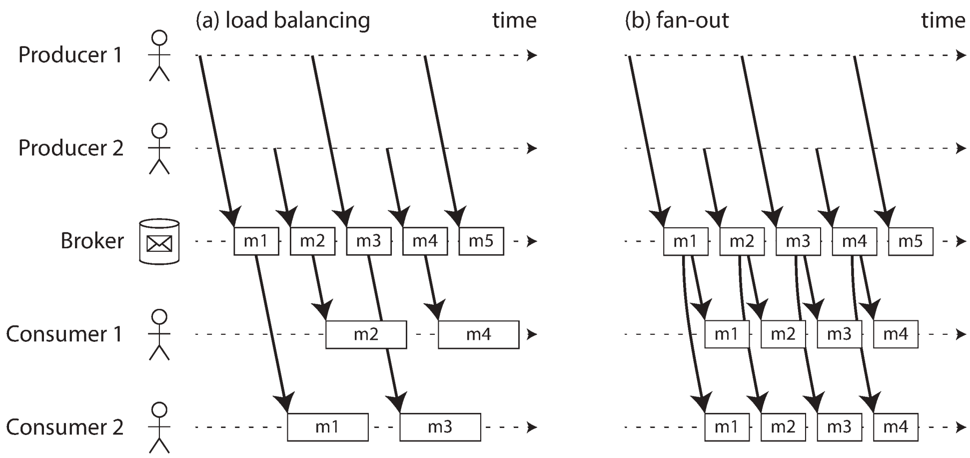

Figure 12-1. (a) Load balancing: sharing the work of consuming a topic among consumers; (b) fan-out: delivering each message to multiple consumers.

The two patterns can be combined, for example using Kafka's *consumer groups* feature. When a consumer group subscribes to a topic, each message in the topic is sent to one of the consumers in the group (load-balancing across the consumers in the group). If two separate consumer groups subscribe to the same topic, each message is sent to one consumer in each group (providing fan-out across consumer groups).

Acknowledgments and redelivery

Consumers may crash at any time, so it could happen that a broker delivers a message to a consumer but the consumer never processes it, or only partially processes it before crashing. In order to ensure that the message is not lost, message brokers use *acknowledgments*: a client must explicitly tell the broker when it has finished processing a message so that the broker can remove it from the queue.

If the connection to a client is closed or times out without the broker receiving an acknowledgment, it assumes that the message was not processed, and therefore it delivers the message again to another consumer. (Note that it could happen that the message actually *was* fully processed, but the acknowledgment was lost in the network. Handling this case requires an atomic commit protocol, as discussed in "Exactly-once message processing", unless the operation was idempotent or exactly-once semantics are not required.)

When combined with load balancing, this redelivery behavior has an interesting effect on the ordering of messages. In Figure 12-2, the consumers generally process messages in the order they were sent by producers. However, consumer 2 crashes while processing message *m3*, at the same time as consumer 1 is processing message *m4*. The unacknowledged message *m3* is subsequently redelivered to consumer 1, with the result that consumer 1 processes messages in the order *m4*, *m3*, *m5*. Thus, *m3* and *m4* are not delivered in the same order as they were sent by producer 1.

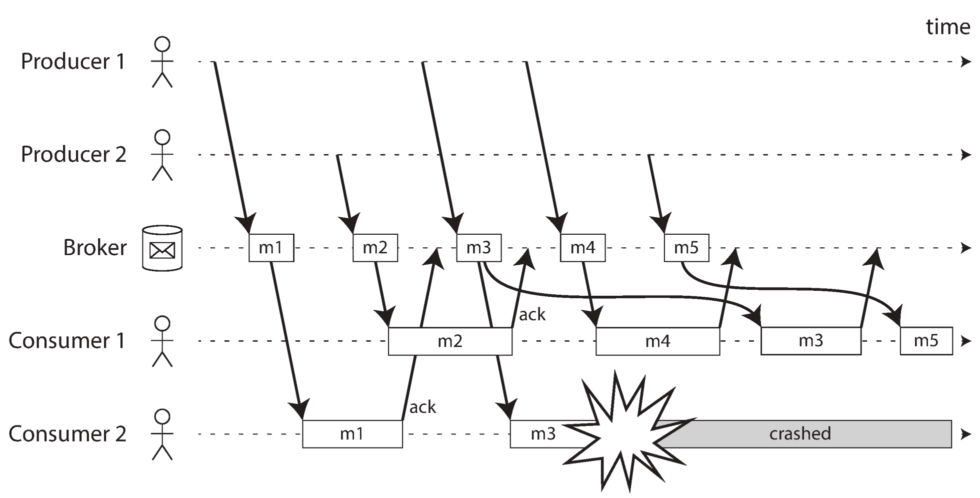

Figure 12-2. Consumer 2 crashes while processing m3, so it is redelivered to consumer 1 at a later time.

Even if the message broker otherwise tries to preserve the order of messages (as required by both the JMS and AMQP standards), the combination of load balancing with redelivery inevitably leads to messages being reordered. To avoid this issue, you can use a separate queue per consumer (i.e., not use the load balancing feature). Message reordering is not a problem if messages are completely independent of each other, but it can be important if there are causal dependencies between messages, as we shall see later in the chapter.

Redelivery can also result in wasted resources, resource starvation, or permanent blockages in a stream. A common scenario is a producer that improperly serializes a message; for example, by leaving out a required key in a JSON-encoded object. Any consumer that reads the message will expect the key, and fail if it's missing. No acknowledgement is sent, so the broker will re-send the message, which will cause another consumer to fail. This loop repeats itself indefinitely. If the broker guarantees strong ordering, no further progress can be made. Brokers that allow message reordering can continue to make progress, but will waste resources on messages that will never be acknowledged.

Dead letter queues (DLQs) are used to handle this problem. Rather than keeping the message in the current queue and retrying forever, the message is moved to a different queue to unblock consumers [17, 18]. Monitoring is usually set up on dead letter queues—any message in the queue is an error. Once a new message is detected, an operator can decide to permanently drop it, manually modify and re-produce the message, or fix consumer code to handle the message appropriately. DLQs are common in most queuing systems, but log-based messaging systems such as Apache Pulsar and stream processing systems such as Kafka Streams now support them as well [19].

Log-based Message Brokers

Sending a packet over a network or making a request to a network service is normally a transient operation that leaves no permanent trace. Although it is possible to record it permanently (using packet capture and logging), we normally don't think of it that way. AMQP/JMS-style message brokers inherited this transient messaging mindset: even though they may write messages to disk, they quickly delete the messages again after they have been delivered to consumers.

Databases and filesystems take the opposite approach: everything that is written to a database or file is normally expected to be permanently recorded, at least until someone explicitly chooses to delete it again.

This difference in mindset has a big impact on how derived data is created. A key feature of batch processes, as discussed in Chapter 11, is that you can run them repeatedly, experimenting with the processing steps, without risk of damaging the input (since the input is read-only). This is not the case with AMQP/JMS-style messaging: receiving a message is destructive if the acknowledgment causes it to be deleted from the broker, so you cannot run the same consumer again and expect to get the same result.

If you add a new consumer to a messaging system, it typically only starts receiving messages sent after the time it was registered; any prior messages are already gone and cannot be recovered. Contrast this with files and databases, where you can add a new client at any time, and it can read data written arbitrarily far in the past (as long as it has not been explicitly overwritten or deleted by the application).

Why can we not have a hybrid, combining the durable storage approach of databases with the low-latency notification facilities of messaging? This is the idea behind *log-based message brokers*, which have become very popular in recent years.

Using logs for message storage

A log is simply an append-only sequence of records on disk. We previously discussed logs in the context of log-structured storage engines and write-ahead logs in Chapter 4, in the context of replication in Chapter 6, and as a form of consensus in Chapter 10.

The same structure can be used to implement a message broker: a producer sends a message by appending it to the end of the log, and a consumer receives messages by reading the log sequentially. If a consumer reaches the end of the log, it waits for a notification that a new message has been appended. The Unix tool `tail -f`, which watches a file for data being appended, essentially works like this.

In order to scale to higher throughput than a single disk can offer, the log can be *sharded* (in the sense of Chapter 7). Different shards can then be hosted on different machines, making each shard a separate log that can be read and written independently from other shards. A topic can then be defined as a group of shards that all carry messages of the same type. This approach is illustrated in Figure 12-3.

Within each shard, which Kafka calls a *partition*, the broker assigns a monotonically increasing sequence number, or *offset*, to every message (in Figure 12-3, the numbers in boxes are message offsets). Such a sequence number makes sense because a partition (shard) is append-only, so the messages within a partition are totally ordered. There is no ordering guarantee across different partitions.

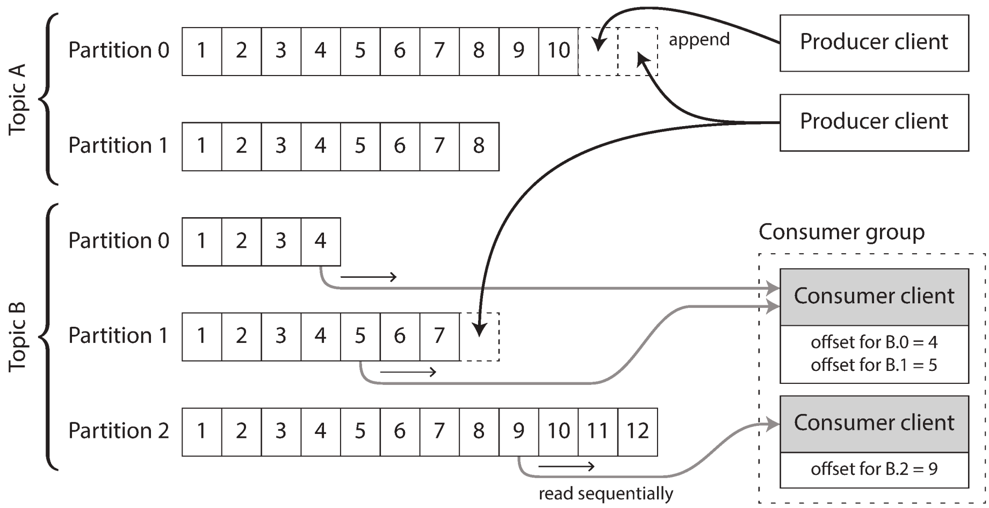

Figure 12-3. Producers send messages by appending them to a topic-partition file, and consumers read these files sequentially.

Apache Kafka [20] and Amazon Kinesis Streams are log-based message brokers that work like this. Google Cloud Pub/Sub is architecturally similar but exposes a JMS-style API rather than a log abstraction [15]. Even though these message brokers write all messages to disk, they are able to achieve throughput of millions of messages per second by sharding across multiple machines, and fault tolerance by replicating messages [21, 22].

Logs compared to traditional messaging

The log-based approach trivially supports fan-out messaging, because several consumers can independently read the log without affecting each other—reading a message does not delete it from the log. To achieve load balancing across a group of consumers, instead of assigning individual messages to consumer clients, the broker can assign entire shards to nodes in the consumer group.

Each client then consumes *all* the messages in the shards it has been assigned. Typically, when a consumer has been assigned a log shard, it reads the messages in the shard sequentially, in a straightforward single-threaded manner. This coarse-grained load balancing approach has some downsides:

- The number of nodes sharing the work of consuming a topic can be at most the number of log shards in that topic, because messages within the same shard are delivered to the same node. (It's possible to create a load balancing scheme in which two consumers share the work of processing a shard by having both read the full set of messages, but one of them only considers messages with even-numbered offsets while the other deals with the odd-numbered offsets. Alternatively, you could spread message processing over a thread pool, but that approach complicates consumer offset management. In general, single-threaded processing of a shard is preferable, and parallelism can be increased by using more shards.)
- If a single message is slow to process, it holds up the processing of subsequent messages in that shard (a form of head-of-line blocking; see "Describing Performance").

Thus, in situations where messages may be expensive to process and you want to parallelize processing on a message-by-message basis, and where message ordering is not so important, the JMS/AMQP style of message broker is preferable. On the other hand, in situations with high message throughput, where each message is fast to process and where message ordering is important, the log-based approach works very well [23, 24]. However, the distinction between the two architectures is being blurred as log-based messaging systems such as Kafka now support JMS/AMQP style consumer groups, which allow multiple consumers to receive messages from the same partition [25, 26].

Since sharded logs typically preserve message ordering only within a single shard, all messages that need to be ordered consistently need to be routed to the same shard. For example, an application may require that the events relating to one particular user appear in a fixed order. This can be achieved by choosing the shard for an event based on the user ID of that event (in other words, making the user ID the *partition key*).

Consumer offsets

Consuming a shard sequentially makes it easy to tell which messages have been processed: all messages with an offset less than a consumer's current offset have already been processed, and all messages with a greater offset have not yet been seen. Thus, the broker does not need to track acknowledgments for every single message—it only needs to periodically record the consumer offsets. The reduced bookkeeping overhead and the opportunities for batching and pipelining in this approach help increase the throughput of log-based systems. If a consumer fails, however, it will resume from the last recorded offset rather than the more recent last offset it saw. This can causes the consumer to see some messages twice.

This offset is in fact very similar to the *log sequence number* that is commonly found in single-leader database replication, and which we discussed in "Setting Up New Followers". In database replication, the log sequence number allows a follower to reconnect to a leader after it has become disconnected, and resume replication without skipping any writes. Exactly the same principle is used here: the message broker behaves like a leader database, and the consumer like a follower.

If a consumer node fails, another node in the consumer group is assigned the failed consumer's shards, and it starts consuming messages at the last recorded offset. If the consumer had processed subsequent messages but not yet recorded their offset, those messages will be processed a second time upon restart. We will discuss ways of dealing with this issue later in the chapter.

Disk space usage

If you only ever append to the log, you will eventually run out of disk space. To reclaim disk space, the log is actually divided into segments, and from time to time old segments are deleted or moved to archive storage. (We'll discuss a more sophisticated way of freeing disk space in "Log compaction".)

This means that if a slow consumer cannot keep up with the rate of messages, and it falls so far behind that its consumer offset points to a deleted segment, it will miss some of the messages. Effectively, the log implements a bounded-size buffer that discards old messages when it gets full, also known as a *circular buffer* or *ring buffer*. However, since that buffer is on disk, it can be quite large.

Let's do a back-of-the-envelope calculation. At the time of writing, a typical large hard drive has a capacity of 20 TB and a sequential write throughput of 250 MB/s. If you are writing messages at the fastest possible rate, it takes about 22 hours until the drive is full and you need to start deleting the oldest messages. That means a disk-based log can always buffer at least 22 hours worth of messages, even if you have many disks with many machines (having more disks increases both the available space and the total write bandwidth).

In practice, deployments rarely use the full write bandwidth of the disk, so the log can typically keep a buffer of several days' or even weeks' worth of messages.

Many log-based message brokers now store messages in object storage to increase their storage capacity, similarly to databases as we saw in "Databases Backed by Object Storage". Message brokers such as Apache Kafka and Redpanda serve older messages from object storage as part of their tiered storage. Others, such as WarpStream, Confluent Freight, and Bufstream store all of their data in the object store. In addition to cost-efficiency, this architecture also makes data integration easier: messages in object storage are stored as Iceberg tables, which enable batch and data warehouse job execution directly on the data without having to copy it into another system.

When consumers cannot keep up with producers

At the beginning of "Messaging Systems" we discussed three choices of what to do if a consumer cannot keep up with the rate at which producers are sending messages: dropping messages, buffering, or applying backpressure. In this taxonomy, the log-based approach is a form of buffering with a large but fixed-size buffer (limited by the available disk space).

If a consumer falls so far behind that the messages it requires are older than what is retained on disk, it will not be able to read those messages—so the broker effectively drops old messages that go back further than the size of the buffer can accommodate. You can monitor how far a consumer is behind the head of the log, and raise an alert if it falls behind significantly. As the buffer is large, there is enough time for a human operator to fix the slow consumer and allow it to catch up before it starts missing messages.

Even if a consumer does fall too far behind and starts missing messages, only that consumer is affected; it does not disrupt the service for other consumers. This fact is a big operational advantage: you can experimentally consume a production log for development, testing, or debugging purposes, without having to worry much about disrupting production services. When a consumer is shut down or crashes, it stops consuming resources—the only thing that remains is its consumer offset.

This behavior also contrasts with traditional message brokers, where you need to be careful to delete any queues whose consumers have been shut down—

otherwise they continue unnecessarily accumulating messages and taking away memory from consumers that are still active.

Replaying old messages

We noted previously that with AMQP- and JMS-style message brokers, processing and acknowledging messages is a destructive operation, since it causes the messages to be deleted on the broker. On the other hand, in a log-based message broker, consuming messages is more like reading from a file: it is a read-only operation that does not change the log.

The only side effect of processing, besides any output of the consumer, is that the consumer offset moves forward. But the offset is under the consumer's control, so it can easily be manipulated if necessary: for example, you can start a copy of a consumer with yesterday's offsets and write the output to a different location, in order to reprocess the last day's worth of messages. You can repeat this any number of times, varying the processing code.

This aspect makes log-based messaging more like the batch processes of the last chapter, where derived data is clearly separated from input data through a repeatable transformation process. It allows more experimentation and easier recovery from errors and bugs, making it a good tool for integrating dataflows within an organization [27].

Databases and Streams

We have drawn some comparisons between message brokers and databases. Even though they have traditionally been considered separate categories of tools, we saw that log-based message brokers have been successful in taking ideas from databases and applying them to messaging. We can also go in reverse: take ideas from messaging and streams, and apply them to databases.

One approach is to use an *event stream as the system of record* for storing data (see "Systems of Record and Derived Data"). This is what happens in *event sourcing*, which we discussed in "Event Sourcing and CQRS": instead of storing data in a data model that is mutated by updating and deleting, you can model every state change as an immutable event, and write it to an append-only log. Any read-optimized materialized views are derived from these events. Log-based message brokers (configured to never delete old events) are well suited for event sourcing since they use append-only storage, and they can notify consumers about new events with low latency.

But you don't have to go as far as adopting event sourcing; even with mutable data models, event streams are useful for databases. In fact, every write to a database is an event that can be captured, stored, and processed. The connection between databases and streams runs deeper than just the physical storage of logs on disk—it is quite fundamental.

For example, a replication log (see "Implementation of Replication Logs") is a stream of database write events, produced by the leader as it processes transactions. The followers apply that stream of writes to their own copy of the database and thus end up with an accurate copy of the same data. The events in the replication log describe the data changes that occurred.

We also came across the *state machine replication* principle in "Using shared logs", which states: if every event represents a write to the database, and every replica processes the same events in the same order, then the replicas will all end up in the same final state. (Processing an event is assumed to be a deterministic operation.) It's just another case of event streams!

In this section we will first look at a problem that arises in heterogeneous data systems, and then explore how we can solve it by bringing ideas from event streams to databases.

Keeping Systems in Sync

As we have seen throughout this book, there is no single system that can satisfy all data storage, querying, and processing needs. In practice, most nontrivial applications need to combine several different technologies in order to satisfy their requirements: for example, using an OLTP database to serve user requests, a cache to speed up common requests, a full-text index to handle search queries, and a data warehouse for analytics. Each of these has its own copy of the data, stored in its own representation that is optimized for its own purposes.

As the same or related data appears in several different places, they need to be kept in sync with one another: if an item is updated in the database, it also needs to be updated in the cache, search indexes, and data warehouse. With data warehouses this synchronization is usually performed by ETL processes (see “Data Warehousing”), often by taking a full copy of a database, transforming it, and bulk-loading it into the data warehouse—in other words, a batch process. Similarly, we saw in “Batch Use Cases” how search indexes, recommendation systems, and other derived data systems might be created using batch processes.

If periodic full database dumps are too slow, an alternative that is sometimes used is *dual writes*, in which the application code explicitly writes to each of the systems when data changes: for example, first writing to the database, then updating the search index, then invalidating the cache entries (or even performing those writes concurrently).

However, dual writes have some serious problems, one of which is a race condition illustrated in Figure 12-4. In this example, two clients concurrently want to update an item X: client 1 wants to set the value to A, and client 2 wants to set it to B. Both clients first write the new value to the database, then write it to the search index. Due to unlucky timing, the requests are interleaved: the database first sees the write from client 1 setting the value to A, then the write from client 2 setting the value to B, so the final value in the database is B. The search index first sees the write from client 2, then client 1, so the final value in the search index is A. The two systems are now permanently inconsistent with each other, even though no error occurred.

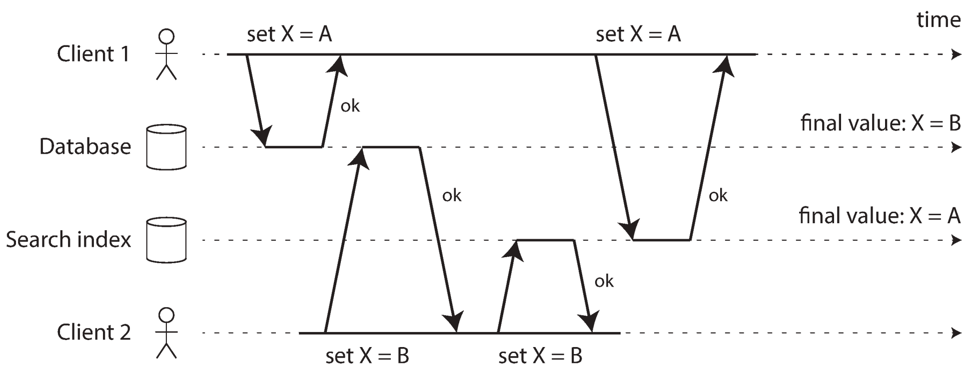

Figure 12-4. In the database, X is first set to A and then to B, while at the search index the writes arrive in the opposite order.

Unless you have some additional concurrency detection mechanism, such as the version vectors we discussed in “Detecting Concurrent Writes”, you will not even notice that concurrent writes occurred—one value will simply silently overwrite another value.

Another problem with dual writes is that one of the writes may fail while the other succeeds. This is a fault-tolerance problem rather than a concurrency problem, but it also has the effect of the two systems becoming inconsistent with each other. Ensuring that they either both succeed or both fail is a case of the atomic commit problem, which is expensive to solve (see "Two-Phase Commit (2PC)").

If you only have one replicated database with a single leader, then that leader determines the order of writes, so the state machine replication approach works among replicas of the database. However, in Figure 12-4 there isn't a single leader: the database may have a leader and the search index may have a leader, but neither follows the other, and so conflicts can occur (see "Multi-Leader Replication").

The situation would be better if there really was only one leader—for example, the database—and if we could make the search index a follower of the database. But is this possible in practice?

Change Data Capture

The problem with most databases' replication logs is that they have long been considered to be an internal implementation detail of the database, not a public API. Clients are supposed to query the database through its data model and query language, not parse the replication logs and try to extract data from them.

For decades, many databases simply did not have a documented way of getting the log of changes written to them. For this reason it was difficult to take all the changes made in a database and replicate them to a different storage technology such as a search index, cache, or data warehouse.

More recently, there has been growing interest in *change data capture* (CDC), which is the process of observing all data changes written to a database and extracting them in a form in which they can be replicated to other systems [28]. CDC is especially interesting if changes are made available as a stream, immediately as they are written.

For example, you can capture the changes in a database and continually apply the same changes to a search index. If the log of changes is applied in the same order, you can expect the data in the search index to match the data in the database. The search index and any other derived data systems are just consumers of the change stream.

Figure 12-5 shows how the concurrency problem of Figure 12-4 is solved with CDC. Even though the two requests to set X to A and B respectively arrive concurrently at the database, the database decides on some order in which to execute them, and writes them to its replication log in that order. The search index picks them up and applies them in the same order. If you need the data in another system, such as a data warehouse, you can simply add it as another consumer of the CDC event stream.

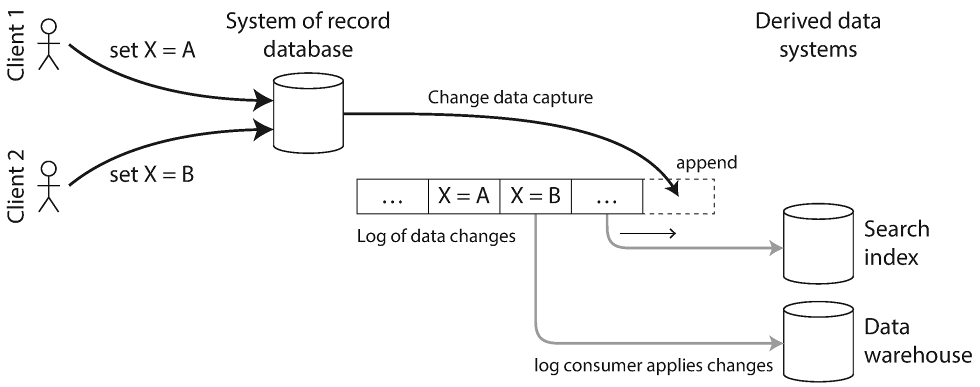

Figure 12-5. Taking data in the order it was written to one database, and applying the changes to other systems in the same order.

Implementing change data capture

We can call the log consumers *derived data systems*, as discussed in "Systems of Record and Derived Data": the data stored in the search index and the data warehouse is just another view onto the data in the system of record. Change data capture is a mechanism for ensuring that all changes made to the system of record are also reflected in the derived data systems so that the derived systems have an accurate copy of the data.

Essentially, change data capture makes one database the leader (the one from which the changes are captured), and turns the others into followers. A log-based message broker is well suited for transporting the change events from the source database to the derived systems, since it preserves the ordering of messages (avoiding the reordering issue of Figure 12-2).

Logical replication logs can be used to implement change data capture (see "Logical (row-based) log replication"), although it comes with challenges, such as handling schema changes and properly modeling updates. The Debezium open source project addresses these challenges. The project contains *source connectors* for MySQL, PostgreSQL, Oracle, SQL Server, Db2, Cassandra, and many other databases. These connectors attach to database replication logs and surface the changes in a standard event schema.

Messages can then be transformed and written to downstream databases. The Kafka Connect framework offers further CDC connectors for various databases, as well. Maxwell does something similar for MySQL by parsing the binlog [29], GoldenGate provides similar facilities for Oracle, and pgcapture does the same for PostgreSQL.

Like message brokers, change data capture is usually asynchronous: the system of record database does not wait for the change to be applied to consumers before committing it. This design has the operational advantage that adding a slow consumer does not affect the system of record too much, but it has the downside that all the issues of replication lag apply (see "Problems with Replication Lag").

Initial snapshot

If you have the log of all changes that were ever made to a database, you can reconstruct the entire state of the database by replaying the log. However, in many cases, keeping all changes forever would require too much disk space, and replaying it would take too long, so the log needs to be truncated.

Building a new full-text index, for example, requires a full copy of the entire database—it is not sufficient to only apply a log of recent changes, since it would be missing items that were not recently updated. Thus, if you don't have the entire log history, you need to start with a consistent snapshot, as previously discussed in "Setting Up New Followers".

The snapshot of the database must correspond to a known position or offset in the change log, so that you know at which point to start applying changes after the snapshot has been processed. Some CDC tools integrate this snapshot facility, while others leave it as a manual operation. Debezium uses Netflix's DBLog watermarking algorithm to provide incremental snapshots [30, 31].

Log compaction

If you can only keep a limited amount of log history, you need to go through the snapshot process every time you want to add a new derived data system. However, *log compaction* provides a good alternative.

We discussed log compaction previously in "Log-Structured Storage", in the context of log-structured storage engines (see Figure 4-3 for an example). The principle is simple: the storage engine periodically looks for log records with the same key, throws away any duplicates, and keeps only the most recent update for each key. This might make log segments much smaller, so segments may also be merged as part of the compaction process, as shown in Figure 12-6. This process runs in the background.

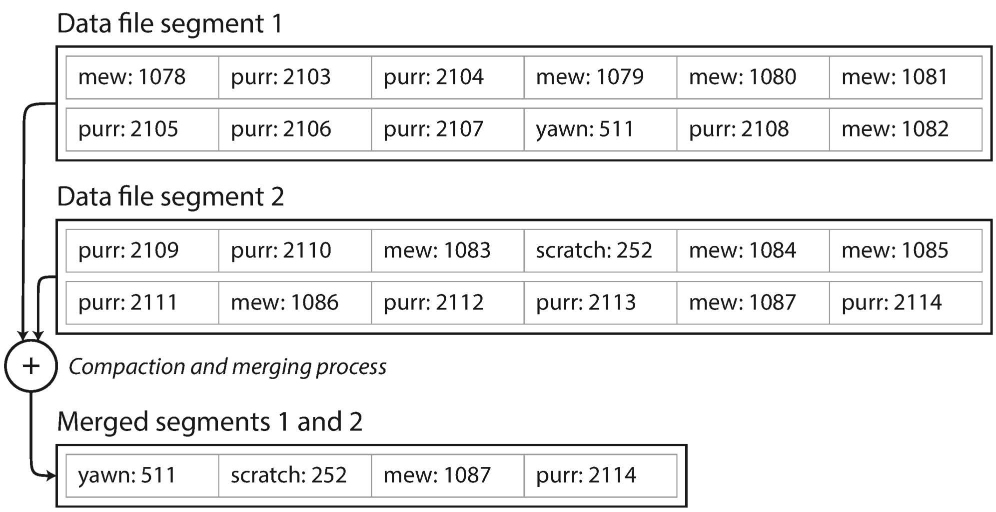

Figure 12-6. A log of key-value pairs, where the key is the ID of a cat video (mew, purr, scratch, or yawn), and the value is the number of times it has been played. Log compaction retains only the most value for each key.

In a log-structured storage engine, an update with a special null value (a *tombstone*) indicates that a key was deleted, and causes it to be removed during log compaction. But as long as a key is not overwritten or deleted, it stays in the log forever. The disk space required for such a compacted log depends only on the current contents of the database, not the number of writes that have ever occurred in the database. If the same key is frequently overwritten, previous values will eventually be garbage-collected, and only the latest value will be retained.

The same idea works in the context of log-based message brokers and change data capture. If the CDC system is set up such that every change has a primary key, and every update for a key replaces the previous value for that key, then it's sufficient to keep just the most recent write for a particular key.

Now, whenever you want to rebuild a derived data system such as a search index, you can start a new consumer from offset 0 of the log-compacted topic, and sequentially scan over all messages in the log. The log is guaranteed to contain the most recent value for every key in the database (and maybe some older values)—in other words, you can use it to obtain a full copy of the database contents without having to take another snapshot of the CDC source database.

This log compaction feature is supported by Apache Kafka. As we shall see later in this chapter, it allows the message broker to be used for durable storage, not just for transient messaging.

API support for change streams

Most popular databases now expose change streams as a first-class interface, rather than the retrofitted and reverse-engineered CDC efforts of the past. Relational databases such as MySQL and PostgreSQL typically send changes through the same replication log they use for their own replicas. Most cloud vendors offer CDC solutions for their products as well: for example, Datastream offers streaming data access for Google Cloud's relational databases and data warehouses.

Even evenutally consistent, quorum-based databases such as Cassandra now support change data capture. As we saw in "Linearizability and quorums", clients must persist writes to a majority of nodes before they're considered visible. CDC support for quorum writes is challenging because there's no single source of truth to subscribe to. Whether the data is visible or not depends on each reader's consistency preferences. Cassandra sidesteps this issue by exposing raw log segments for each node rather than providing a single stream of mutations. Systems that wish to consume the data must read the raw log segments for each node and decide how best to merge them into a single stream (much like a quorum reader does) [32].

Kafka Connect [33] integrates change data capture tools for a wide range of database systems with Kafka. Once the stream of change events is in Kafka, it can be used to update derived data systems such as search indexes, and also feed into stream processing systems as discussed later in this chapter.

Change data capture versus event sourcing

Let's compare change data capture to event sourcing. Similarly to change data capture, event sourcing involves storing all changes to the application state as a log of change events. The biggest difference is that event sourcing applies the idea at a different level of abstraction:

- In change data capture, the application uses the database in a mutable way, updating and deleting records at will. The log of changes is extracted from the database at a low level (e.g., by parsing the replication log), which ensures that the order of writes extracted from the database matches the order in which they were actually written, avoiding the race condition in Figure 12-4.

- In event sourcing, the application logic is explicitly built on the basis of immutable events that are written to an event log. In this case, the event store is append-only, and updates or deletes of events are discouraged or prohibited. Events are designed to reflect things that happened at the application level, rather than low-level state changes.

Which one is better depends on your situation. Adopting event sourcing is a big change for an application that is not already doing it; it has a number of pros and cons, which we discussed in "Event Sourcing and CQRS". In contrast, CDC can be added to an existing database with minimal changes—the application writing to the database might not even know that CDC is occurring.

CHANGE DATA CAPTURE AND DATABASE SCHEMAS

Though change data capture appears easier to adopt than event sourcing, it comes with its own set of challenges.

In a microservices architecture, a database is typically only accessed from one service. Other services interact with it through that service's public API, but they don't normally access the database directly. This makes the database an internal implementation detail of the service, allowing the developers to change its database schema without affecting the public API.

However, CDC systems typically use the upstream database's schema when replicating its data, which turns these schemas into public APIs that must be managed much like the public API of the service. A developer who removes a table column in their database table will break downstream consumers that depend on this field. Such challenges have always existed with data pipelines, but they typically only impacted data warehouse ETL. Since CDC is often implemented as a data stream, other production services might be consumers. Breaking such consumers can cause a customer-facing outage [34]. Data contracts are often used to prevent these breakages.

A common way to decouple internal from external schemas is to use the *outbox pattern*. Outboxes are tables with their own schemas, which are exposed to the CDC system rather than the internal domain model in the database [35, 36]. Developers can then modify their internal schemas as they see fit while leaving their outbox tables untouched. This might look like a dual write—it is. However, outboxes avoid the challenges we discussed in "Keeping Systems in Sync" by keeping both writes in the same system (the database). This design allows both writes to appear in a single transaction.

Outboxes present a few tradeoffs, though. Developers must still maintain the transformation between their internal and outbox schemas, which can be challenging. An outbox also increases the amount of data that the database has to write to its underlying storage, which might trigger performance problems.

Like with change data capture, replaying the event log allows you to reconstruct the current state of the system. However, log compaction needs to be handled differently:

- A CDC event for the update of a record typically contains the entire new version of the record, so the current value for a primary key is entirely determined by the most recent event for that primary key, and log compaction can discard previous events for the same key.
- On the other hand, with event sourcing, events are modeled at a higher level: an event typically expresses the intent of a user action, not the mechanics of the state update that occurred as a result of the action. In this case, later events typically do not override prior events, and so you need the full history of events to reconstruct the final state. Log compaction is not possible in the same way.

Applications that use event sourcing typically have some mechanism for storing snapshots of the current state that is derived from the log of events, so they don't need to repeatedly reprocess the full log. However, this is only a performance optimization to speed up reads and recovery from crashes; the intention is that the system is able to store all raw events forever and reprocess the full event log whenever required. We discuss this assumption in "Limitations of immutability".

State, Streams, and Immutability

We saw in Chapter 11 that batch processing benefits from the immutability of its input files, so you can run experimental processing jobs on existing input files without fear of damaging them. This principle of immutability is also what makes event sourcing and change data capture so powerful.

We normally think of databases as storing the current state of the application—this representation is optimized for reads, and it is usually the most convenient for serving queries. The nature of state is that it changes, so databases support updating and deleting data as well as inserting it. How does this fit with immutability?

Whenever you have state that changes, that state is the result of the events that mutated it over time. For example, your list of currently available seats is the result of the reservations you have processed, the current account balance is the result of the credits and debits on the account, and the response time graph for your web server is an aggregation of the individual response times of all web requests that have occurred.

No matter how the state changes, there was always a sequence of events that caused those changes. Even as things are done and undone, the fact remains true that those events occurred. The key idea is that mutable state and an append-only log of immutable events do not contradict each other: they are two sides of the same coin. The log of all changes, the *changelog*, represents the evolution of state over time.

If you are mathematically inclined, you might say that the application state is what you get when you integrate an event stream over time, and a change stream is what you get when you differentiate the state by time, as shown in Figure 12-7 [37, 38]. The analogy has limitations (for example, the second derivative of state does not seem to be meaningful), but it's a useful starting point for thinking about data.

$$state(now) = \int_{t=0}^{now} stream(t)\ \mathrm{d}t \qquad\qquad stream(t) = \frac{\mathrm{d}\ state(t)}{\mathrm{d}t}$$

Figure 12-7. The relationship between the current application state and an event stream.

If you store the changelog durably, that simply has the effect of making the state reproducible. If you consider the log of events to be your system of record, and any mutable state as being derived from it, it becomes easier to reason about the flow of data through a system. As Jim Gray and Andreas Reuter put it in 1992 [39]:

> *[T]here is no fundamental need to keep a database at all; the log contains all the information there is. The only reason for storing the database (i.e., the current end-of-the-log) is performance of retrieval operations.*

Log compaction is one way of bridging the distinction between log and database state: it retains only the latest version of each record, and discards overwritten versions.

Advantages of immutable events

Immutability in databases is an old idea. For example, accountants have been using immutability for centuries in financial bookkeeping. When a transaction occurs, it is recorded in an append-only *ledger*, which is essentially a log of events describing money, goods, or services that have changed hands. The accounts, such as profit and loss or the balance sheet, are derived from the transactions in the ledger by adding them up [40].

If a mistake is made, accountants don't erase or change the incorrect transaction in the ledger—instead, they add another transaction that compensates for the mistake, for example refunding an incorrect charge. The incorrect transaction still remains in the ledger forever, because it might be important for auditing reasons. If incorrect figures, derived from the incorrect ledger, have already been published, then the figures for the next accounting period include a correction. This process is entirely normal in accounting [41].

Although such auditability is particularly important in financial systems, it is also beneficial for many other systems that are not subject to such strict regulation. If you accidentally deploy buggy code that writes bad data to a database, recovery is much harder if the code is able to destructively overwrite data. With an append-only log of immutable events, it is much easier to diagnose what happened and recover from the problem. Similarly, customer service can use an audit log to diagnose customer requests and complaints.

Immutable events also capture more information than just the current state. For example, on a shopping website, a customer may add an item to their cart and then remove it again. Although the second event cancels out the first event from the point of view of order fulfillment, it may be useful to know for analytics purposes that the customer was considering a particular item but then decided against it. Perhaps they will choose to buy it in the future, or perhaps they found a substitute. This information is recorded in an event log, but would be lost in a database that deletes items when they are removed from the cart.

Deriving several views from the same event log

Moreover, by separating mutable state from the immutable event log, you can derive several different read-oriented representations from the same log of events. This works just like having multiple consumers of a stream (Figure 12-5): for example, the analytic database Druid ingests directly from Kafka using this approach, and Kafka Connect sinks can export data from Kafka to various different databases and indexes [33].

Having an explicit translation step from an event log to a database makes it easier to evolve your application over time: if you want to introduce a new feature that presents your existing data in some new way, you can use the event log to build a separate read-optimized view for the new feature, and run it alongside the existing systems without having to modify them. Running old and new systems side by side is often easier than performing a complicated schema migration in an existing system. Once readers have switched to the new system and the old system is no longer needed, you can simply shut it down and reclaim its resources [42, 43].

This idea of writing data in one write-optimized form, and then translating it into different read-optimized representations as needed, is the *command query responsibility segregation* (CQRS) pattern that we already encountered in "Event Sourcing and CQRS". It doesn't necessarily require event sourcing: you can just as well build multiple materialized views from a stream of CDC events [44].

The traditional approach to database and schema design is based on the fallacy that data must be written in the same form as it will be queried. Debates about normalization and denormalization (see "Normalization, Denormalization, and Joins") become largely irrelevant if you can translate data from a write-optimized event log to read-optimized application state: it is entirely reasonable to denormalize data in the read-optimized views, as the translation process gives you a mechanism for keeping it consistent with the event log.

In "Case Study: Social Network Home Timelines" we discussed a social network's home timelines, a cache of recent posts by the people a particular user is following (like a mailbox). This is another example of read-optimized state: home timelines are highly denormalized, since your posts are duplicated in all of the timelines of the people following you. However, the fan-out service keeps this duplicated state in sync with new posts and new following relationships, which keeps the duplication manageable.

Concurrency control

The biggest downside of CQRS is that the consumers of the event log are usually asynchronous, so there is a possibility that a user may make a write to the log, then read from a derived view and find that their write has not yet been reflected in the view. We discussed this problem and potential solutions previously in "Reading Your Own Writes".

One solution would be to perform the updates of the read view synchronously with appending the event to the log. This either requires a distributed transaction across the event log and the derived view, or some way of waiting until an event has been reflected in the view. Both approaches are usually impractical, so views are normally updated asynchronously.

On the other hand, deriving the current state from an event log also simplifies some aspects of concurrency control. Much of the need for multi-object transactions (see "Single-Object and Multi-Object Operations") stems from a single user action requiring data to be changed in several different places. With event sourcing, you can design an event such that it is a self-contained description of a user action. The user action then requires only a single write in one place—namely appending the event to the log—which is easy to make atomic.

If the event log and the application state are sharded in the same way (for example, processing an event for a customer in shard 3 only requires updating shard 3 of the application state), then a straightforward single-threaded log consumer needs no concurrency control for writes—by construction, it only processes a single event at a time (see also "Actual Serial Execution"). The log removes the nondeterminism of concurrency by defining a serial order of events in a shard [27]. If an event touches multiple state shards, a bit more work is required, which we will discuss in Chapter 13.

Many systems that don't use an event-sourced model nevertheless rely on immutability for concurrency control: various databases internally use immutable data structures or multi-version data to support point-in-time snapshots (see "Indexes and snapshot isolation"). Version control systems such as Git, Mercurial, and Fossil also rely on immutable data to preserve version history of files.

Limitations of immutability

To what extent is it feasible to keep an immutable history of all changes forever? The answer depends on the amount of churn in the dataset. Some workloads mostly add data and rarely update or delete; they are easy to make immutable. Other workloads have a high rate of updates and deletes on a comparatively small dataset; in these cases, the immutable history may grow prohibitively large, fragmentation may become an issue, and the performance of compaction and garbage collection becomes crucial for operational robustness [45, 46].

Besides the performance reasons, there may also be circumstances in which you need data to be deleted for administrative or legal reasons, in spite of all immutability. For example, privacy regulations such as the European General Data Protection Regulation (GDPR) require that a user's personal information be deleted and erroneous information be removed on demand, or an accidental leak of sensitive information may need to be contained.

In these circumstances, it's not sufficient to just append another event to the log to indicate that the prior data should be considered deleted—you actually want to rewrite history and pretend that the data was never written in the first place. For example, Datomic calls this feature *excision* [47], and the Fossil version control system has a similar concept called *shunning* [48].

Truly deleting data is surprisingly hard [49], since copies can live in many places: for example, storage engines, filesystems, and SSDs often write to a new location rather than overwriting in place [41], and backups are often deliberately immutable to prevent accidental deletion or corruption.

One way of enabling deletion of immutable data is *crypto-shredding* [50]: data that you may want to delete in the future is stored encrypted, and when you want to get rid of it, you forget the encryption key. The encrypted data is then still there, but nobody can use it. In some sense this only moves the problem around: the actual data is now immutable, but your key storage is mutable.

Moreover, you have to decide up front which data is going to be encrypted with the same key, and when you are going to use different keys—an important decision, since you can later crypto-shred either all or none of the data encrypted with a particular key, but not some of it. Storing a separate key for every single data item would get too unwieldy, as the key storage would get as big as the primary data storage. More sophisticated schemes such as puncturable encryption [51] make it possible to selectively revoke a key's decryption abilities, but they are not widely used.

Overall, deletion is more a matter of "making it harder to retrieve the data" than actually "making it impossible to retrieve the data." Nevertheless, you sometimes have to try, as we shall see in "Legislation and Self-Regulation".

Processing Streams

So far in this chapter we have talked about where streams come from (user activity events, sensors, and writes to databases), and we have talked about how streams are transported (through direct messaging, via message brokers, and in event logs).

What remains is to discuss what you can do with the stream once you have it—namely, you can process it. Broadly, there are three options:

1. You can take the data in the events and write it to a database, cache, search index, or similar storage system, from where it can then be queried by other clients. As shown in Figure 12-5, this is a good way of keeping a database in sync with changes happening in other parts of the system—especially if the stream consumer is the only client writing to the database. Writing to a storage system is the streaming equivalent of what we discussed in "Batch Use Cases".
2. You can push the events to users in some way, for example by sending email alerts or push notifications, or by streaming the events to a real-time dashboard where they are visualized. In this case, a human is the ultimate consumer of the stream.
3. You can process one or more input streams to produce one or more output streams. Streams may go through a pipeline consisting of several such processing stages before they eventually end up at an output (option 1 or 2).

In the rest of this chapter, we will discuss option 3: processing streams to produce other, derived streams. A piece of code that processes streams like this is known as an *operator* or a *job*. It is closely related to the Unix processes and MapReduce jobs we discussed in Chapter 11, and the pattern of dataflow is similar: a stream processor consumes input streams in a read-only fashion and writes its output to a different location in an append-only fashion.

The patterns for sharding and parallelization in stream processors are also very similar to those in MapReduce and the dataflow engines we saw in Chapter 11, so we won't repeat those topics here. Basic mapping operations such as transforming and filtering records also work the same.

The one crucial difference from batch jobs is that a stream never ends. This difference has many implications: as discussed at the start of this chapter, sorting does not make sense with an unbounded dataset, and so sort-merge joins (see "JOIN and GROUP BY") cannot be used. Fault-tolerance mechanisms must also change: with a batch job that has been running for a few minutes, a failed task can simply be restarted from the beginning, but with a stream job that has been running for several years, restarting from the beginning after a crash may not be a viable option.

Uses of Stream Processing

Stream processing has long been used for monitoring purposes, where an organization wants to be alerted if certain things happen. For example:

- Fraud detection systems need to determine if the usage patterns of a credit card have unexpectedly changed, and block the card if it is likely to have been stolen.
- Trading systems need to examine price changes in a financial market and execute trades according to specified rules.
- Manufacturing systems need to monitor the status of machines in a factory, and quickly identify the problem if there is a malfunction.
- Military and intelligence systems need to track the activities of a potential aggressor, and raise the alarm if there are signs of an attack.

These kinds of applications require quite sophisticated pattern matching and correlations. However, other uses of stream processing have also emerged over time. In this section we will briefly compare and contrast some of these applications.

Complex event processing

Complex event processing (CEP) is an approach developed in the 1990s for analyzing event streams, especially geared toward the kind of application that requires searching for certain event patterns [52]. Similarly to the way that a regular expression allows you to search for certain patterns of characters in a string, CEP allows you to specify rules to search for certain patterns of events in a stream.

CEP systems often use a high-level declarative query language like SQL, or a graphical user interface, to describe the patterns of events that should be detected. These queries are submitted to a processing engine that consumes the input streams and internally maintains a state machine that performs the required matching. When a match is found, the engine emits a *complex event* (hence the name) with the details of the event pattern that was detected [53].

In these systems, the relationship between queries and data is reversed compared to normal databases. Usually, a database stores data persistently and treats queries as transient: when a query comes in, the database searches for data matching the query, and then forgets about the query when it has finished. CEP engines reverse these roles: queries are stored long-term; as each event arrives, the engine checks whether it has now seen an event pattern that matches any of its standing queries [54].

Implementations of CEP include Esper, Apama, and TIBCO StreamBase. Distributed stream processors like Flink and Spark Streaming also have SQL support for declarative queries on streams.

Stream analytics

Another area in which stream processing is used is for *analytics* on streams. The boundary between CEP and stream analytics is blurry, but as a general rule, analytics tends to be less interested in finding specific event sequences and is more oriented toward aggregations and statistical metrics over a large number of events—for example:

- Measuring the rate of some type of event (how often it occurs per time interval)
- Calculating the rolling average of a value over some time period
- Comparing current statistics to previous time intervals (e.g., to detect trends or to alert on metrics that are unusually high or low compared to the same time last week)

Such statistics are usually computed over fixed time intervals—for example, you might want to know the average number of queries per second to a service over the last 5 minutes, and their 99th percentile response time during that period. Averaging over a few minutes smoothes out irrelevant fluctuations from one second to the next, while still giving you a timely picture of any changes in traffic pattern. The time interval over which you aggregate is known as a *window*, and we will look into windowing in more detail in "Reasoning About Time".

Stream analytics systems sometimes use probabilistic algorithms, such as Bloom filters (which we encountered in "Bloom filters") for set membership, HyperLogLog [55] for cardinality estimation, and various percentile estimation algorithms (see "Computing Percentiles"). Probabilistic algorithms produce approximate results, but have the advantage of requiring significantly less memory in the stream processor than exact algorithms. This use of approximation algorithms sometimes leads people to believe that stream processing systems are always lossy and inexact, but that is wrong: there is nothing inherently approximate about stream processing, and using probabilistic algorithms is merely an optimization [56].

Many open source distributed stream processing frameworks are designed with analytics in mind: for example, Apache Storm, Spark Streaming, Flink, Samza, Apache Beam, and Kafka Streams [57]. Hosted services include Google Cloud Dataflow and Azure Stream Analytics.

Maintaining materialized views

We saw that a stream of changes to a database can be used to keep derived data systems, such as caches, search indexes, and data warehouses, up to date with a source database. These are examples of maintaining materialized views: deriving an alternative view onto some dataset so that you can query it efficiently, and updating that view whenever the underlying data changes [37].

Similarly, in event sourcing, application state is maintained by applying a log of events; here the application state is also a kind of materialized view. Unlike stream analytics scenarios, it is usually not sufficient to consider only events within some time window: building the materialized view potentially requires *all* events over an arbitrary time period, apart from any obsolete events that may be discarded by log compaction. In effect, you need a window that stretches all the way back to the beginning of time.

In principle, any stream processor could be used for materialized view maintenance, although the need to maintain events forever runs counter to the assumptions of some analytics-oriented frameworks that mostly operate on windows of a limited duration. Kafka Streams and Confluent's ksqlDB support this kind of usage, building upon Kafka's support for log compaction [58].

INCREMENTAL VIEW MAINTENANCE

Databases might seem well suited for materialized view maintenance; they are designed to keep full copies of a dataset, after all. Many also support materialized views. We saw in "Materialized Views and Data Cubes" that analytical queries typical of a data warehouse can be materialized into OLAP cubes.

Unfortunately, databases often refresh materialized view tables using batch jobs or on-demand requests such as PostgreSQL's `REFRESH MATERIALIZED VIEW`. Views are recalculated periodically rather than as updates to souce data occurs. This approach has two significant drawbacks that make it inappropriate for stream processing view maintenance:

1. Poor efficiency: All data is reprocessed every time the view is updated, though it's likely that most of the data remains unchanged.
2. Data freshness: changes in source data are not reflected in a materialized view until its query is re-run during its next scheduled update.

It is possible to write database triggers that update materialized views efficiently in scenarios where the data is easily partitioned and the computation is naturally incremental. For example, if a materialized view maintains total sales revenue per-day, the row for the appropriate day can be updated every time a new sale occurs. Bespoke solutions work in a few cases, but many SQL queries can't be easily or efficiently converted to incremental computation.

Incremental view maintenance (IVM) is a more general solution to the problems listed above. IVM techniques convert relational grammars such as SQL into operators capable of incremental computations. Rather than processing entire datasets, IVM algorithms recompute and update only data that has changed [38, 59, 60]. View computation becomes far more efficient. Updates can then be run much more frequently, which dramatically increases data freshness.

Databases such as Materialize [61], RisingWave, ClickHouse, and Feldera all use IVM techniques to provide efficient incremental materialized views. These databases ingest streams of events to expose materialized views in realtime. Recent events are buffered in-memory and periodically used to update on-disk materialized views. Reads combine the recent events and the materialized data to provide a single realtime view. Since reads are often expressed in SQL and materialized views are often stored in OLAP-style formats, these systems also support large-scale data warehouse-style queries such as those disucssed in Chapter 11.

Search on streams

Besides CEP, which allows searching for patterns consisting of multiple events, there is also sometimes a need to search for individual events based on complex criteria, such as full-text search queries.

For example, media monitoring services subscribe to feeds of news articles and broadcasts from media outlets, and search for any news mentioning companies, products, or topics of interest. This is done by formulating a search query in advance, and then continually matching the stream of news items against this query. Similar features exist on some websites: for example, users of real estate websites can ask to be notified when a new property matching their search criteria appears on the market. The percolator feature of Elasticsearch [62] is one option for implementing this kind of stream search.

Conventional search engines first index the documents and then run queries over the index. By contrast, searching a stream turns the processing on its head: the queries are stored, and the documents run past the queries, like in CEP. In the simplest case, you can test every document against every query, although this can get slow if you have a large number of queries. To optimize the process, it is possible to index the queries as well as the documents, and thus narrow down the set of queries that may match [63].

Event-Driven Architectures and RPC

In "Event-Driven Architectures" we discussed message-passing systems as an alternative to RPC—i.e., as a mechanism for services to communicate, as used for example in the actor model. Although these systems are also based on messages and events, we normally don't think of them as stream processors:

- Actor frameworks are primarily a mechanism for managing concurrency and distributed execution of communicating modules, whereas stream processing is primarily a data management technique.
- Communication between actors is often ephemeral and one-to-one, whereas event logs are durable and multi-subscriber.

- Actors can communicate in arbitrary ways (including cyclic request/response patterns), but stream processors are usually set up in acyclic pipelines where every stream is the output of one particular job, and derived from a well-defined set of input streams.

That said, there is some crossover area between RPC-like systems and stream processing. For example, Apache Storm has a feature called *distributed RPC*, which allows user queries to be farmed out to a set of nodes that also process event streams; these queries are then interleaved with events from the input streams, and results can be aggregated and sent back to the user. (See also "Multi-shard data processing".)

It is also possible to process streams using actor frameworks. However, many such frameworks do not guarantee message delivery in the case of crashes, so the processing is not fault-tolerant unless you implement additional retry logic.

Reasoning About Time

Stream processors often need to deal with time, especially when used for analytics purposes, which frequently use time windows such as "the average over the last five minutes." It might seem that the meaning of "the last five minutes" should be unambiguous and clear, but unfortunately the notion is surprisingly tricky.

In a batch process, the processing tasks rapidly crunch through a large collection of historical events. If some kind of breakdown by time needs to happen, the batch process needs to look at the timestamp embedded in each event. There is no point in looking at the system clock of the machine running the batch process, because the time at which the process is run has nothing to do with the time at which the events actually occurred.

A batch process may read a year's worth of historical events within a few minutes; in most cases, the timeline of interest is the year of history, not the few minutes of processing. Moreover, using the timestamps in the events allows the processing to be deterministic: running the same process again on the same input yields the same result.

On the other hand, many stream processing frameworks use the local system clock on the processing machine (the *processing time*) to determine windowing [64]. This approach has the advantage of being simple, and it is reasonable if the delay between event creation and event processing is negligibly short. However, it breaks down if there is any significant processing lag—i.e., if the processing may happen noticeably later than the time at which the event actually occurred.

Event time versus processing time

There are many reasons why processing may be delayed: queueing, network faults, a performance issue leading to contention in the message broker or processor, a restart of the stream consumer, or reprocessing of past events while recovering from a fault or after fixing a bug in the code.

Moreover, message delays can also lead to unpredictable ordering of messages. For example, say a user first makes one web request (which is handled by web server A), and then a second request (which is handled by server B). A and B emit events describing the requests they handled, but B's event reaches the message broker before A's event does. Now stream processors will first see the B event and then the A event, even though they actually occurred in the opposite order.

If it helps to have an analogy, consider the *Star Wars* movies: Episode IV was released in 1977, Episode V in 1980, and Episode VI in 1983, followed by Episodes I, II, and III in 1999, 2002, and 2005, respectively, and Episodes VII, VIII, and IX in 2015, 2017, and 2019 [65]. If you watched the movies in the order they came out, the order in which you processed the movies is inconsistent with the order of their narrative. (The episode number is like the event timestamp, and the date when you watched the movie is the processing time.) As humans, we are able to cope with such discontinuities, but stream processing algorithms need to be specifically written to accommodate such timing and ordering issues.

Confusing event time and processing time leads to bad data. For example, say you have a stream processor that measures the rate of requests (counting the number of requests per second). If you redeploy the stream processor, it may be shut down for a minute and process the backlog of events when it comes back up. If you measure the rate based on the processing time, it will look as if there was a sudden anomalous spike of requests while processing the backlog, when in fact the real rate of requests was steady (Figure 12-8).

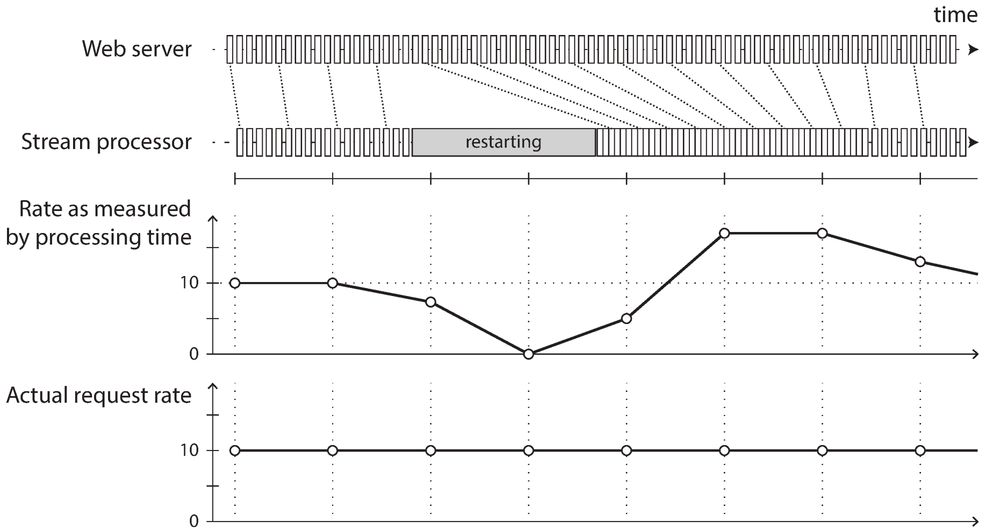

Figure 12-8. Windowing by processing time introduces artifacts due to variations in processing rate.

Handling straggler events

A tricky problem when defining windows in terms of event time is that you can never be sure when you have received all of the events for a particular window, or whether there are some events still to come.

For example, say you're grouping events into one-minute windows so that you can count the number of requests per minute. You have counted some number of events with timestamps that fall in the 37th minute of the hour, and time has moved on; now most of the incoming events fall within the 38th and 39th minutes of the hour. When do you declare that you have finished the window for the 37th minute, and output its counter value?

You can time out and declare a window ready after you have not seen any new events for that window in a while. However, it could still happen that some events were buffered on another machine somewhere, delayed due to a network interruption. You need to be able to handle such *straggler* events that arrive after the window has already been declared complete. Broadly, you have two options [1]:

1. Ignore the straggler events, as they are probably a small percentage of events in normal circumstances. You can track the number of dropped events as a metric, and alert if you start dropping a significant amount of data.
2. Publish a *correction*, an updated value for the window with stragglers included. You may also need to retract the previous output.

In some cases it is possible to use a special message to indicate, “From now on there will be no more messages with a timestamp earlier than *t*,” which can be used by consumers to trigger windows [66]. However, if several producers on different machines are generating events, each with their own minimum timestamp thresholds, the consumers need to keep track of each producer individually. Adding and removing producers is trickier in this case.

Whose clock are you using, anyway?

Assigning timestamps to events is even more difficult when events can be buffered at several points in the system. For example, consider a mobile app that reports events for usage metrics to a server. The app may be used while the device is offline, in which case it will buffer events locally on the device and send them to a server when an internet connection is next available (which may be hours or even days later). To any consumers of this stream, the events will appear as extremely delayed stragglers.

In this context, the timestamp on the events should really be the time at which the user interaction occurred, according to the mobile device’s local clock. However, the clock on a user-controlled device often cannot be trusted, as it may be accidentally or deliberately set to the wrong time (see “Clock Synchronization and Accuracy”). The time at which the event was received by the server (according to the server’s clock) is more likely to be accurate, since the server is under your control, but less meaningful in terms of describing the user interaction.

To adjust for incorrect device clocks, one approach is to log three timestamps [67]:

- The time at which the event occurred, according to the device clock
- The time at which the event was sent to the server, according to the device clock
- The time at which the event was received by the server, according to the server clock

By subtracting the second timestamp from the third, you can estimate the offset between the device clock and the server clock (assuming the network delay is negligible compared to the required timestamp accuracy). You can then apply that offset to the event timestamp, and thus estimate the true time at which the event actually occurred (assuming the device clock offset did not change between the time the event occurred and the time it was sent to the server).

This problem is not unique to stream processing—batch processing suffers from exactly the same issues of reasoning about time. It is just more noticeable in a streaming context, where we are more aware of the passage of time.

Types of windows

Once you know how the timestamp of an event should be determined, the next step is to decide how windows over time periods should be defined. The window can then be used for aggregations, for example to count events, or to calculate the average of values within the window. Several types of windows are in common use [64, 68]:

Tumbling window

> A tumbling window has a fixed length, and every event belongs to exactly one window. For example, if you have a 1-minute tumbling window, all the events with timestamps between 10:03:00 and 10:03:59 are grouped into one window, events between 10:04:00 and 10:04:59 into the next window, and so on. You could implement a 1-minute tumbling window by taking each event timestamp and rounding it down to the nearest minute to determine the window that it belongs to.

Hopping window

> A hopping window also has a fixed length, but allows windows to overlap in order to provide some smoothing. For example, a 5-minute window with a hop size of 1 minute would contain the events between 10:03:00 and 10:07:59, then the next window would cover events between 10:04:00 and 10:08:59, and so on. You can implement this hopping window by first calculating 1-minute tumbling windows, and then aggregating over several adjacent windows.

Sliding window

> A sliding window contains all the events that occur within some interval of each other. For example, a 5-minute sliding window would cover events at 10:03:39 and 10:08:12, because they are less than 5

minutes apart (note that tumbling and hopping 5-minute windows would not have put these two events in the same window, as they use fixed boundaries). A sliding window can be implemented by keeping a buffer of events sorted by time and removing old events when they expire from the window.

Session window

Unlike the other window types, a session window has no fixed duration. Instead, it is defined by grouping together all events for the same user that occur closely together in time, and the window ends when the user has been inactive for some time (for example, if there have been no events for 30 minutes). Sessionization is a common requirement for website analytics.

Window operations usually maintain temporary state. In some cases, the state is of a fixed size, no matter how large the window or how many events occur: for example, a counting operation will only have one counter regardless of the window size or event count. On the other hand, sliding windows or stream joins, which we discuss in the next section, require that events be buffered until the window finishes. Therefore, large window sizes or high-throughput streams can cause stream processors to keep a lot of temporary state. You must then take care to ensure the machines running stream processing tasks have enough capacity to maintain this state, whether in-memory or on-disk.

Stream Joins

In "JOIN and GROUP BY" we discussed how batch jobs can join datasets by key, and how such joins form an important part of data pipelines. Since stream processing generalizes data pipelines to incremental processing of unbounded datasets, there is exactly the same need for joins on streams.

However, the fact that new events can appear anytime on a stream makes joins on streams more challenging than in batch jobs. To understand the situation better, let's distinguish three different types of joins: *stream-stream* joins, *stream-table* joins, and *table-table* joins. In the following sections we'll illustrate each by example.

Stream-stream join (window join)

Say you have a search feature on your website, and you want to detect recent trends in searched-for URLs. Every time someone types a search query, you log an event containing the query and the results returned. Every time someone clicks one of the search results, you log another event recording the click. In order to calculate the click-through rate for each URL in the search results, you need to bring together the events for the search action and the click action, which are connected by having the same session ID. Similar analyses are needed in advertising systems [69].

The click may never come if the user abandons their search, and even if it comes, the time between the search and the click may be highly variable: in many cases it might be a few seconds, but it could be as long as days or weeks (if a user runs a search, forgets about that browser tab, and then returns to the tab and clicks a result sometime later). Due to variable network delays, the click event may even arrive before the search event. You can choose a suitable window for the join—for example, you may choose to join a click with a search if they occur at most one hour apart.

Note that embedding the details of the search in the click event is not equivalent to joining the events: doing so would only tell you about the cases where the user clicked a search result, not about the searches where the user did not click any of the results. In order to measure search quality, you need accurate click-through rates, for which you need both the search events and the click events.

To implement this type of join, a stream processor needs to maintain *state*: for example, all the events that occurred in the last hour, indexed by session ID. Whenever a search event or click event occurs, it is added to the appropriate index, and the stream processor also checks the other index to see if another event for the same session ID has already arrived. If there is a matching event, you emit an event saying which search result was clicked. If the search event expires without you seeing a matching click event, you emit an event saying which search results were not clicked.

Stream-table join (stream enrichment)

In "JOIN and GROUP BY" (Figure 11-2) we saw an example of a batch job joining two datasets: a set of user activity events and a database of user profiles. It is natural to think of the user activity events as a stream, and to perform the same join on a continuous basis in a stream processor: the input is a stream of activity events containing a user ID, and the output is a stream of activity events in which the user ID has been augmented with profile information about the user. This process is sometimes known as *enriching* the activity events with information from the database.

To perform this join, the stream process needs to look at one activity event at a time, look up the event's user ID in the database, and add the profile information to the activity event. The database lookup could be implemented by querying a remote database; however, as discussed in "JOIN and GROUP BY", such remote queries are likely to be slow and risk overloading the database [58].

Another approach is to load a copy of the database into the stream processor so that it can be queried locally without a network round-trip. This technique is called a *hash join* since the local copy of the database might be an in-memory hash table if it is small enough, or an index on the local disk.

The difference from batch jobs is that a batch job uses a point-in-time snapshot of the database as input, whereas a stream processor is long-running, and the contents of the database are likely to change over time, so the stream processor's local copy of the database needs to be kept up to date. This issue can be solved by change data capture: the stream processor can subscribe to a changelog of the user profile database as well as the stream of activity events. When a profile is created or modified, the stream processor updates its local copy. Thus, we obtain a join between two streams: the activity events and the profile updates.

A stream-table join is actually very similar to a stream-stream join; the biggest difference is that for the table changelog stream, the join uses a window that reaches back to the "beginning of time" (a conceptually infinite window), with newer versions of records overwriting older ones. For the stream input, the join might not maintain a window at all.

Table-table join (materialized view maintenance)

Consider the social network timeline example that we discussed in "Case Study: Social Network Home Timelines". We said that when a user wants to view their home timeline, it is too expensive to iterate over all the people the user is following, find their recent posts, and merge them.

Instead, we want a timeline cache: a kind of per-user “inbox” to which posts are written as they are sent, so that reading the timeline is a single lookup. Materializing and maintaining this cache requires the following event processing:

- When user u sends a new post, it is added to the timeline of every user who is following u.
- When a user deletes a post, or deletes their entire account, it is removed from all users’ timelines.
- When user u_1 starts following user u_2, recent posts by u_2 are added to u_1’s timeline.
- When user u_1 unfollows user u_2, posts by u_2 are removed from u_1’s timeline.

To implement this cache maintenance in a stream processor, you need streams of events for posts (sending and deleting) and for follow relationships (following and unfollowing). The stream process needs to maintain a database containing the set of followers for each user so that it knows which timelines need to be updated when a new post arrives.

Another way of looking at this stream process is that it maintains a materialized view for a query that joins two tables (posts and follows), something like the following:

```
SELECT follows.follower_id AS timeline_id,
  array_agg(posts.* ORDER BY posts.timestamp DESC)
FROM posts
JOIN follows ON follows.followee_id = posts.sender_id
GROUP BY follows.follower_id
```

The join of the streams corresponds directly to the join of the tables in that query. The timelines are effectively a cache of the result of this query, updated every time the underlying tables change.

NOTE

If you regard a stream as the derivative of a table, as in Figure 12-7, and regard a join as a product of two tables $u \cdot v$, something interesting happens: the stream of changes to the materialized join follows the product rule $(u \cdot v)' = u'v + uv'$. In words: any change of posts is joined with the current followers, and any change of followers is joined with the current posts [37].

Time-dependence of joins

The three types of joins described here (stream-stream, stream-table, and table-table) have a lot in common: they all require the stream processor to maintain some state (search and click events, user profiles, or follower list) based on one join input, and query that state on messages from the other join input.

The order of the events that maintain the state is important (it matters whether you first follow and then unfollow, or the other way round). In a sharded event log like Kafka, the ordering of events within a single shard (partition) is preserved, but there is typically no ordering guarantee across different streams or shards.

This raises a question: if events on different streams happen around a similar time, in which order are they processed? In the stream-table join example, if a user updates their profile, which activity events are joined with the old profile (processed before the profile update), and which are joined with the new profile (processed after the profile update)? Put another way: if state changes over time, and you join with some state, what point in time do you use for the join?

Such time dependence can occur in many places. For example, if you sell things, you need to apply the right tax rate to invoices, which depends on the country or state, the type of product, and the date of sale (since tax rates change from time to time). When joining sales to a table of tax rates, you probably want to join with the tax rate at the time of the sale, which may be different from the current tax rate if you are reprocessing historical data.

If the ordering of events across streams is undetermined, the join becomes nondeterministic [70], which means you cannot rerun the same job on the same input and necessarily get the same result: the events on the input streams may be interleaved in a different way when you run the job again.

In data warehouses, this issue is known as a *slowly changing dimension* (SCD), and it is often addressed by using a unique identifier for a particular version of the joined record: for example, every time the tax rate changes, it is given a new identifier, and the invoice includes the identifier for the tax rate at the time of sale [71, 72]. This change makes the join deterministic, but has the consequence that log compaction is not possible, since all versions of the records in the table need to be retained. Alternatively, you can denormalize the data and include the applicable tax rate directly in every sale event.

Fault Tolerance

In the final section of this chapter, let's consider how stream processors can tolerate faults. We saw in Chapter 11 that batch processing frameworks can tolerate faults fairly easily: if a task fails, it can simply be started again on another machine, and the output of the failed task is discarded. This transparent retry is possible because input files are immutable, each task writes its output to a separate file, and output is only made visible when a task completes successfully.

In particular, the batch approach to fault tolerance ensures that the output of the batch job is the same as if nothing had gone wrong, even if in fact some tasks did fail. It appears as though every input record was processed exactly once—no records are skipped, and none are processed twice. Although restarting tasks means that records may in fact be processed multiple times, the visible effect in the output is as if they had only been processed once. This principle is known as *exactly-once semantics*, although *effectively-once* would be a more descriptive term [73].

The same issue of fault tolerance arises in stream processing, but it is less straightforward to handle: waiting until a task is finished before making its output visible is not an option, because a stream is infinite and so you can never finish processing it.

Microbatching and checkpointing

One solution is to break the stream into small blocks, and treat each block like a miniature batch process. This approach is called *microbatching*, and it is used in Spark Streaming [74]. The batch size is typically around one second, which is the result of a performance compromise: smaller batches incur greater scheduling and coordination overhead, while larger batches mean a longer delay before results of the stream processor become visible.

Microbatching also implicitly provides a tumbling window equal to the batch size (windowed by processing time, not event timestamps); any jobs that require larger windows need to explicitly carry over state from one microbatch to the next.

A variant approach, used in Apache Flink, is to periodically generate rolling checkpoints of state and write them to durable storage [75, 76]. If a stream operator crashes, it can restart from its most recent checkpoint and discard any output generated between the last checkpoint and the crash. The checkpoints are triggered by barriers in the message stream, similar to the boundaries between microbatches, but without forcing a particular window size.

Within the confines of the stream processing framework, the microbatching and checkpointing approaches provide the same exactly-once semantics as batch processing. However, as soon as output leaves the stream processor (for example, by writing to a database, sending messages to an external message broker, or sending emails), the framework is no longer able to discard the output of a failed microbatch. In this case, restarting a failed task causes the external side effect to happen twice, and microbatching or checkpointing alone is not sufficient to prevent this problem.

Atomic commit revisited

In order to give the appearance of exactly-once processing in the presence of faults, we need to ensure that all outputs and side effects of processing an event take effect *if and only if* the processing is successful. Those effects include any messages sent to downstream operators or external messaging systems (including email or push notifications), any database writes, any changes to operator state, and any acknowledgment of input messages (including moving the consumer offset forward in a log-based message broker).

Those things either all need to happen atomically, or none of them must happen, but they should not go out of sync with each other. If this approach sounds familiar, it is because we discussed it in "Exactly-once message processing" in the context of distributed transactions and two-phase commit.

In Chapter 10 we discussed the problems in the traditional implementations of distributed transactions, such as XA. However, in more restricted environments it is possible to implement such an atomic commit facility efficiently. This approach is used in Google Cloud Dataflow [66, 75], VoltDB [77], and Apache Kafka [78, 79]. Unlike XA, these implementations do not attempt to provide transactions across heterogeneous technologies, but instead keep the transactions internal by managing both state changes and messaging within the stream processing framework. The overhead of the transaction protocol can be amortized by processing several input messages within a single transaction.

Idempotence

Our goal is to discard the partial output of any failed tasks so that they can be safely retried without taking effect twice. Distributed transactions are one way of achieving that goal, but another way is to rely on *idempotence*, as we saw in "Durable Execution and Workflows" [80].

An idempotent operation is one that you can perform multiple times, and it has the same effect as if you performed it only once. For example, deleting a key in a key-value store is idempotent (deleting the value again has no further effect), whereas incrementing a counter is not idempotent (performing the increment again means the value is incremented twice).

Even if an operation is not naturally idempotent, it can often be made idempotent with a bit of extra metadata. For example, when consuming messages from Kafka, every message has a persistent, monotonically increasing offset. When writing a value to an external database, you can include the offset of the message that triggered the last write with the value. Thus, you can tell whether an update has already been applied, and avoid performing the same update again.

The state handling in Storm's Trident is based on a similar idea. Relying on idempotence implies several assumptions: restarting a failed task must replay the same messages in the same order (a log-based message broker does this), the processing must be deterministic, and no other node may concurrently update the same value [81, 82].

When failing over from one processing node to another, fencing may be required (see "Distributed Locks and Leases") to prevent interference from a node that is thought to be dead but is actually alive. Despite all those caveats, idempotent operations can be an effective way of achieving exactly-once semantics with only a small overhead.

Rebuilding state after a failure

Any stream process that requires state—for example, any windowed aggregations (such as counters, averages, and histograms) and any tables and indexes used for joins—must ensure that this state can be recovered after a failure.

One option is to keep the state in a remote datastore and replicate it, although having to query a remote database for each individual message can be slow. An alternative is to keep state local to the stream processor, and replicate it periodically. Then, when the stream processor is recovering from a failure, the new task can read the replicated state and resume processing without data loss.

For example, Flink periodically captures snapshots of operator state and writes them to durable storage such as a distributed filesystem [75, 76], and Kafka Streams replicates state changes by sending them to a dedicated Kafka topic with log compaction, similar to change data capture [83]. VoltDB replicates state by redundantly processing each input message on several nodes (see "Actual Serial Execution").

In some cases, it may not even be necessary to replicate the state, because it can be rebuilt from the input streams. For example, if the state consists of aggregations over a fairly short window, it may be fast enough to simply replay the input events corresponding to that window. If the state is a local replica of a database, maintained by change data capture, the database can also be rebuilt from the log-compacted change stream.

However, all of these trade-offs depend on the performance characteristics of the underlying infrastructure: in some systems, network delay may be lower than disk access latency, and network bandwidth may be comparable to disk bandwidth. There is no universally ideal trade-off for all situations, and the merits of local versus remote state may also shift as storage and networking technologies evolve.

Summary

In this chapter we have discussed event streams, what purposes they serve, and how to process them. In some ways, stream processing is very much like the batch processing we discussed in Chapter 11, but done continuously on unbounded (never-ending) streams rather than on a fixed-size input [84]. From this perspective, message brokers and event logs serve as the streaming equivalent of a filesystem.

We spent some time comparing two types of message brokers:

AMQP/JMS-style message broker

The broker assigns individual messages to consumers, and consumers acknowledge individual messages when they have been successfully processed. Messages are deleted from the broker once they have been acknowledged. This approach is appropriate as an asynchronous form of RPC (see also "Event-Driven Architectures"), for example in a task queue, where the exact order of message processing is not important and where there is no need to go back and read old messages again after they have been processed.

Log-based message broker

The broker assigns all messages in a shard to the same consumer node, and always delivers messages in the same order. Parallelism is achieved through sharding, and consumers track their progress by checkpointing the offset of the last message they have processed. The broker retains messages on disk, so it is possible to jump back and reread old messages if necessary.

The log-based approach has similarities to the replication logs found in databases (see Chapter 6) and log-structured storage engines (see Chapter 4). It is also a form of consensus, as we saw in Chapter 10. We saw that this approach is especially appropriate for stream processing systems that consume input streams and generate derived state or derived output streams.

In terms of where streams come from, we discussed several possibilities: user activity events, sensors providing periodic readings, and data feeds (e.g., market data in finance) are naturally represented as streams. We saw that it can also be useful to think of the writes to a database as a stream: we can capture the changelog—i.e., the history of all changes made to a database—either implicitly through change data capture or explicitly through event sourcing. Log compaction allows the stream to retain a full copy of the contents of a database.

Representing databases as streams opens up powerful opportunities for integrating systems. You can keep derived data systems such as search indexes, caches, and analytics systems continually up to date by consuming the log of changes and applying them to the derived system. You can even build fresh views onto existing data by starting from scratch and consuming the log of changes from the beginning all the way to the present.

The facilities for maintaining state as streams and replaying messages are also the basis for the techniques that enable stream joins and fault tolerance in various stream processing frameworks. We discussed several purposes of stream processing, including searching for event patterns (complex event processing), computing windowed aggregations (stream analytics), and keeping derived data systems up to date (materialized views).

We then discussed the difficulties of reasoning about time in a stream processor, including the distinction between processing time and event timestamps, and the problem of dealing with straggler events that arrive after you thought your window was complete.

We distinguished three types of joins that may appear in stream processes:

Stream-stream joins

> Both input streams consist of activity events, and the join operator searches for related events that occur within some window of time. For example, it may match two actions taken by the same user within 30 minutes of each other. The two join inputs may in fact be the same stream (a *self-join*) if you want to find related events within that one stream.

Stream-table joins

> One input stream consists of activity events, while the other is a database changelog. The changelog keeps a local copy of the database up to date. For each activity event, the join operator queries the database and outputs an enriched activity event.

Table-table joins

> Both input streams are database changelogs. In this case, every change on one side is joined with the latest state of the other side. The result is a stream of changes to the materialized view of the join between the two tables.

Finally, we discussed techniques for achieving fault tolerance and exactly-once semantics in a stream processor. As with batch processing, we need to discard the partial output of any failed tasks. However, since a stream process is long-running and produces output continuously, we can't simply discard all output. Instead, a finer-grained recovery mechanism can be used, based on microbatching, checkpointing, transactions, or idempotent writes.

FOOTNOTES

REFERENCES

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment*, volume 8, issue 12, pages 1792–1803, August 2015. doi:10.14778/2824032.2824076

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*, 2nd edition. MIT Press, 1996. ISBN: 978-0-262-51087-5, archived at archive.org/details/sicp_20211010

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

[4] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring Streams – A New Class of Data Management Applications. At *28th International Conference on Very Large Data Bases* (VLDB), August 2002. doi:10.1016/B978-155860869-6/50027-5

[5] Matthew Sackman. Pushing Back. *wellquite.org*, May 2016. Archived at perma.cc/3KCZ-RUFY

[6] Thomas Figg (tef). how (not) to write a pipeline. *cohost.org*, June 2023. Archived at perma.cc/A3V8-NYCM

[7] Vicent Martí. Brubeck, a statsd-Compatible Metrics Aggregator. *github.blog*, June 2015. Archived at perma.cc/TP3Q-DJYM

[8] Seth Lowenberger. MoldUDP64 Protocol Specification V 1.00. *nasdaqtrader.com*, July 2009. Archived at https://perma.cc/7CRQ-QBD7

[9] Ian Malpass. Measure Anything, Measure Everything. *codeascraft.com*, February 2011. Archived at archive.org

[10] Dieter Plaetinck. 25 Graphite, Grafana and statsd Gotchas. *grafana.com*, March 2016. Archived at perma.cc/3NP3-67U7

[11] Jeff Lindsay. Web Hooks to Revolutionize the Web. *progrium.com*, May 2007. Archived at perma.cc/BF9U-XNX4

[12] Jim N. Gray. Queues Are Databases. Microsoft Research Technical Report MSR-TR-95-56, December 1995. Archived at arxiv.org

[13] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, Kate Stout, and Nigel Deakin. JSR-343 Java Message Service (JMS) 2.0 Specification. *jms-spec.java.net*, March 2013. Archived at perma.cc/E4YG-46TA

[14] Sanjay Aiyagari, Matthew Arrott, Mark Atwell, Jason Brome, Alan Conway, Robert Godfrey, Robert Greig, Pieter Hintjens, John O'Hara, Matthias Radestock, Alexis Richardson, Martin Ritchie, Shahrokh Sadjadi, Rafael Schloming, Steven Shaw, Martin Sustrik, Carl Trieloff, Kim van der Riet, and Steve Vinoski. AMQP: Advanced Message Queuing Protocol Specification. Version 0-9-1, November 2008. Archived at perma.cc/6YJJ-GM9X

[15] Architectural overview of Pub/Sub. *cloud.google.com*, 2025. Archived at perma.cc/VWF5-ABP4

[16] Aris Tzoumas. Lessons from scaling PostgreSQL queues to 100k events per second. *rudderstack.com*, July 2025. Archived at perma.cc/QD8C-VA4Y

[17] Robin Moffatt. Kafka Connect Deep Dive – Error Handling and Dead Letter Queues. *confluent.io*, March 2019. Archived at perma.cc/KQ5A-AB28

[18] Dunith Danushka. Message reprocessing: How to implement the dead letter queue. *redpanda.com*. Archived at perma.cc/R7UB-WEWF

[19] Damien Gasparina, Loic Greffier, and Sebastien Viale. KIP-1034: Dead letter queue in Kafka Streams. *cwiki.apache.org*, April 2024. Archived at perma.cc/3VXV-QXAN

[20] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: A Distributed Messaging System for Log Processing. At *6th International Workshop on Networking Meets Databases* (NetDB), June 2011. Archived at perma.cc/CSW7-TCQ5

[21] Jay Kreps. Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines). *engineering.linkedin.com*, April 2014. Archived at archive.org

[22] Kartik Paramasivam. How We're Improving and Advancing Kafka at LinkedIn. *engineering.linkedin.com*, September 2015. Archived at perma.cc/3S3V-JCYJ

[23] Philippe Dobbelaere and Kyumars Sheykh Esmaili. Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations. At *11th ACM International Conference on Distributed and Event-based Systems* (DEBS), June 2017. doi:10.1145/3093742.3093908

[24] Kate Holterhoff. Why Message Queues Endure: A History. *redmonk.com*, December 2024. Archived at perma.cc/6DX8-XK4W

[25] Andrew Schofield. KIP-932: Queues for Kafka. *cwiki.apache.org*, May 2023. Archived at perma.cc/LBE4-BEMK

[26] Jack Vanlightly. The advantages of queues on logs. *jack-vanlightly.com*, October 2023. Archived at perma.cc/WJ7V-287K

[27] 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

[28] Andy Hattemer. Change Data Capture is having a moment. Why? *materialize.com*, September 2021. Archived at perma.cc/AL37-P53C

[29] Prem Santosh Udaya Shankar. Streaming MySQL Tables in Real-Time to Kafka. *engineeringblog.yelp.com*, August 2016. Archived at perma.cc/5ZR3-2GVV

[30] Andreas Andreakis, Ioannis Papapanagiotou. DBLog: A Watermark Based Change-Data-Capture Framework. October 2020. Archived at arxiv.org

[31] Jiri Pechanec. Percolator. *debezium.io*, October 2021. Archived at perma.cc/EQ8E-W6KQ

[32] Debezium maintainers. Debezium Connector for Cassandra. *debezium.io*. Archived at perma.cc/WR6K-EKMD

[33] Neha Narkhede. Announcing Kafka Connect: Building Large-Scale Low-Latency Data Pipelines. *confluent.io*, February 2016. Archived at perma.cc/8WXJ-L6GF

[34] Chris Riccomini. Kafka change data capture breaks database encapsulation. *cnr.sh*, November 2018. Archived at perma.cc/P572-9MKF

[35] Gunnar Morling. "Change Data Capture Breaks Encapsulation". Does it, though? *decodable.co*, November 2023. Archived at perma.cc/YX2P-WNWR

[36] Gunnar Morling. Revisiting the Outbox Pattern. *decodable.co*, October 2024. Archived at perma.cc/M5ZL-RPS9

[37] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, volume 18, issue 2, pages 3–18, June 1995. Archived at archive.org

[38] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, Val Tannen. DBSP: Incremental Computation on Streams and Its Applications to Databases. *SIGMOD Record*, volume 53, issue 1, pages 87–95, March 2024. doi:10.1145/3665252.3665271

[39] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992. ISBN: 9781558601901

[40] Martin Kleppmann. Accounting for Computer Scientists. *martin.kleppmann.com*, March 2011. Archived at perma.cc/9EGX-P38N

[41] Pat Helland. Immutability Changes Everything. At *7th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2015.

[42] Martin Kleppmann. *Making Sense of Stream Processing*. Report, O'Reilly Media, May 2016. Archived at perma.cc/RAY4-JDVX

[43] Kartik Paramasivam. Stream Processing Hard Problems – Part 1: Killing Lambda. *engineering.linkedin.com*, June 2016. Archived at archive.org

[44] Stéphane Derosiaux. CQRS: What? Why? How? *sderosiaux.medium.com*, September 2019. Archived at perma.cc/FZ3U-HVJ4

[45] Baron Schwartz. Immutability, MVCC, and Garbage Collection. *xaprb.com*, December 2013. Archived at archive.org

[46] Daniel Eloff, Slava Akhmechet, Jay Kreps, et al. Re: Turning the Database Inside-out with Apache Samza. Hacker News discussion, *news.ycombinator.com*, March 2015. Archived at perma.cc/ML9E-JC83

[47] Datomic Documentation: Excision. Cognitect, Inc., *docs.datomic.com*. Archived at perma.cc/J5QQ-SH32

[48] Fossil Documentation: Deleting Content from Fossil. *fossil-scm.org*, 2025. Archived at perma.cc/DS23-GTNG

[49] Jay Kreps. The irony of distributed systems is that data loss is really easy but deleting data is surprisingly hard. *x.com*, March 2015. Archived at perma.cc/7RRZ-V7B7

[50] Brent Robinson. Crypto shredding: How it can solve modern data retention challenges. *medium.com*, January 2019. Archived at https://perma.cc/4LFK-S6XE

[51] Matthew D. Green and Ian Miers. Forward Secure Asynchronous Messaging from Puncturable Encryption. At *IEEE Symposium on Security and Privacy*, May 2015. doi:10.1109/SP.2015.26

[52] David C. Luckham. What's the Difference Between ESP and CEP? *complexevents.com*, June 2019. Archived at perma.cc/E7PZ-FDEF

[53] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, volume 15, issue 2, pages 121–142, June 2006. doi:10.1007/s00778-004-0147-z

[54] Julian Hyde. Data in Flight: How Streaming SQL Technology Can Help Solve the Web 2.0 Data Crunch. *ACM Queue*, volume 7, issue 11, December 2009. doi:10.1145/1661785.1667562

[55] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. HyperLogLog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm. At *Conference on Analysis of Algorithms* (AofA), June 2007. doi:10.46298/dmtcs.3545

[56] Jay Kreps. Questioning the Lambda Architecture. *oreilly.com*, July 2014. Archived at perma.cc/2WY5-HC8Y

[57] Ian Reppel. An Overview of Apache Streaming Technologies. *ianreppel.org*, March 2016. Archived at perma.cc/BB3E-QJLW

[58] Jay Kreps. Why Local State is a Fundamental Primitive in Stream Processing. *oreilly.com*, July 2014. Archived at perma.cc/P8HU-R5LA

[59] RisingWave Labs. Deep Dive Into the RisingWave Stream Processing Engine - Part 2: Computational Model. *risingwave.com*, November 2023. Archived at perma.cc/LM74-XDEL

[60] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. At *6th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2013.

[61] Andy Hattemer. Incremental Computation in the Database. *materialize.com*, March 2020. Archived at perma.cc/AL94-YVRN

[62] Shay Banon. Percolator. *elastic.co*, February 2011. Archived at perma.cc/LS5R-4FQX

[63] Alan Woodward and Martin Kleppmann. Real-Time Full-Text Search with Luwak and Samza. *martin.kleppmann.com*, April 2015. Archived at perma.cc/2U92-Q7R4

[64] Tyler Akidau. The World Beyond Batch: Streaming 102. *oreilly.com*, January 2016. Archived at perma.cc/4XF9-8M2K

[65] Stephan Ewen. Streaming Analytics with Apache Flink. At *Kafka Summit*, April 2016. Archived at perma.cc/QBQ4-F9MR

[66] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proceedings of the VLDB Endowment*, volume 6, issue 11, pages 1033–1044, August 2013. doi:10.14778/2536222.2536229

[67] Alex Dean. Improving Snowplow's Understanding of Time. *snowplow.io*, September 2015. Archived at perma.cc/6CT9-Z3Q2

[68] Azure Stream Analytics: Windowing functions. Microsoft Azure Reference, *learn.microsoft.com*, July 2025. Archived at archive.org

[69] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, and Shivakumar Venkataraman. Photon: Fault-Tolerant and Scalable Joining of Continuous Data Streams. At *ACM International Conference on Management of Data* (SIGMOD), June 2013. doi:10.1145/2463676.2465272

[70] Ben Kirwin. Doing the Impossible: Exactly-Once Messaging Patterns in Kafka. *ben.kirw.in*, November 2014. Archived at perma.cc/A5QL-QRX7

[71] Pat Helland. Data on the Outside Versus Data on the Inside. At *2nd Biennial Conference on Innovative Data Systems Research* (CIDR), January 2005.

[72] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, 3rd edition. John Wiley & Sons, 2013. ISBN: 978-1-118-53080-1

[73] Viktor Klang. I'm coining the phrase 'effectively-once' for message processing with at-least-once + idempotent operations. *x.com*, October 2016. Archived at perma.cc/7DT9-TDG2

[74] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. At *4th USENIX Conference in Hot Topics in Cloud Computing* (HotCloud), June 2012.

[75] Kostas Tzoumas, Stephan Ewen, and Robert Metzger. High-Throughput, Low-Latency, and Exactly-Once Stream Processing with Apache Flink. *ververica.com*, August 2015. Archived at archive.org

[76] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight Asynchronous Snapshots for Distributed Dataflows. arXiv:1506.08603 [cs.DC], June 2015.

[77] Ryan Betts and John Hugg. *Fast Data: Smart and at Scale*. Report, O'Reilly Media, October 2015. Archived at perma.cc/VQ6S-XQQY

[78] Neha Narkhede and Guozhang Wang. Exactly-Once Semantics Are Possible: Here's How Kafka Does It. *confluent.io*, June 2019. Archived at perma.cc/Q2AU-Q2ED

[79] Jason Gustafson, Flavio Junqueira, Apurva Mehta, Sriram Subramanian, and Guozhang Wang. KIP-98 – Exactly Once Delivery and Transactional Messaging. *cwiki.apache.org*, November 2016. Archived at perma.cc/95PT-RCTG

[80] Pat Helland. Idempotence Is Not a Medical Condition. *Communications of the ACM*, volume 55, issue 5, page 56, May 2012. doi:10.1145/2160718.2160734

[81] Jay Kreps. Re: Trying to Achieve Deterministic Behavior on Recovery/Rewind. Email to *samza-dev* mailing list, September 2014. Archived at perma.cc/7DPD-GJNL

[82] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, volume 34, issue 3, pages 375–408, September 2002. doi:10.1145/568522.568525

[83] Adam Warski. Kafka Streams – How Does It Fit the Stream Processing Landscape? *softwaremill.com*, June 2016. Archived at perma.cc/WQ5Q-H2J2

[84] Stephan Ewen, Fabian Hueske, and Xiaowei Jiang. Batch as a Special Case of Streaming and Alibaba's contribution of Blink. *flink.apache.org*, February 2019. Archived at perma.cc/A529-SKA9