Chapter 9. Building Data Pipelines

When people discuss building data pipelines using Apache Kafka, they are usually referring to a couple of use cases. The first is building a data pipeline where Apache Kafka is one of the two end points—for example, getting data from Kafka to S3 or getting data from MongoDB into Kafka. The second use case involves building a pipeline between two different systems but using Kafka as an intermediary. An example of this is getting data from Twitter to Elasticsearch by sending the data first from Twitter to Kafka and then from Kafka to Elasticsearch.

When we added Kafka Connect to Apache Kafka in version 0.9, it was after we saw Kafka used in both use cases at LinkedIn and other large organizations. We noticed that there were specific challenges in integrating Kafka into data pipelines that every organization had to solve, and decided to add APIs to Kafka that solve some of those challenges rather than force every organization to figure them out from scratch.

The main value Kafka provides to data pipelines is its ability to serve as a very large, reliable buffer between various stages in the pipeline. This effectively decouples producers and consumers of data within the pipeline and allows use of the same data from the source in multiple target applications and systems, all with different timeliness and availability requirements. This decoupling, combined with reliability, security, and efficiency, makes Kafka a good fit for most data pipelines.

PUTTING DATA INTEGRATION IN CONTEXT

Some organizations think of Kafka as an *end point* of a pipeline. They look at questions such as “How do I get data from Kafka to Elastic?” This is a valid question to ask—especially if there is data you need in Elastic and it is currently in Kafka—and we will look at ways to do exactly this. But we are going to start the discussion by looking at the use of Kafka within a larger context that includes at least two (and possibly many more) end points that are not Kafka itself. We encourage anyone faced with a data-integration problem to consider the bigger picture and not focus only on the immediate end points. Focusing on short-term integrations is how you end up with a complex and expensive-to-maintain data integration mess.

In this chapter, we’ll discuss some of the common issues that you need to take into account when building data pipelines. Those challenges are not specific to Kafka but are general data integration problems. Nonetheless, we will show why Kafka is a good fit for data integration use cases and how it addresses many of those challenges. We will discuss how the Kafka Connect API is different from the normal producer and consumer clients, and when each client type should be used. Then we’ll jump into some details of Kafka Connect. While a full discussion of Kafka Connect is outside the scope of this chapter, we will show examples of basic usage to get you started and give you pointers on where to learn more. Finally, we’ll discuss other data integration systems and how they integrate with Kafka.

Considerations When Building Data Pipelines

While we won’t get into all the details on building data pipelines here, we would like to highlight some of the most important things to take into account when designing

software architectures with the intent of integrating multiple systems.

Timeliness

Some systems expect their data to arrive in large bulks once a day; others expect the data to arrive a few milliseconds after it is generated. Most data pipelines fit somewhere in between these two extremes. Good data integration systems can support different timeliness requirements for different pipelines and also make the migration between different timetables easier as business requirements change. Kafka, being a streaming data platform with scalable and reliable storage, can be used to support anything from near-real-time pipelines to daily batches. Producers can write to Kafka as frequently and infrequently as needed, and consumers can also read and deliver the latest events as they arrive. Or consumers can work in batches: run every hour, connect to Kafka, and read the events that accumulated during the previous hour.

A useful way to look at Kafka in this context is that it acts as a giant buffer that decouples the time-sensitivity requirements between producers and consumers. Producers can write events in real time, while consumers process batches of events, or vice versa. This also makes it trivial to apply back pressure—Kafka itself applies back pressure on producers (by delaying acks when needed) since consumption rate is driven entirely by the consumers.

Reliability

We want to avoid single points of failure and allow for fast and automatic recovery from all sorts of failure events. Data pipelines are often the way data arrives to business-

critical systems; failure for more than a few seconds can be hugely disruptive, especially when the timeliness requirement is closer to the few milliseconds end of the spectrum. Another important consideration for reliability is delivery guarantees—some systems can afford to lose data, but most of the time there is a requirement for *at-least-once* delivery, which means every event from the source system will reach its destination, but sometimes retries will cause duplicates. Often, there is even a requirement for *exactly-once* delivery—every event from the source system will reach the destination with no possibility for loss or duplication.

We discussed Kafka's availability and reliability guarantees in depth in Chapter 7. As we discussed, Kafka can provide at-least-once on its own, and exactly-once when combined with an external data store that has a transactional model or unique keys. Since many of the end points are data stores that provide the right semantics for exactly-once delivery, a Kafka-based pipeline can often be implemented as exactly-once. It is worth highlighting that Kafka's Connect API makes it easier for connectors to build an end-to-end exactly-once pipeline by providing an API for integrating with the external systems when handling offsets. Indeed, many of the available open source connectors support exactly-once delivery.

High and Varying Throughput

The data pipelines we are building should be able to scale to very high throughputs, as is often required in modern data systems. Even more importantly, they should be able to adapt if throughput suddenly increases.

With Kafka acting as a buffer between producers and consumers, we no longer need to couple consumer

throughput to the producer throughput. We no longer need to implement a complex back-pressure mechanism because if producer throughput exceeds that of the consumer, data will accumulate in Kafka until the consumer can catch up. Kafka's ability to scale by adding consumers or producers independently allows us to scale either side of the pipeline dynamically and independently to match the changing requirements.

Kafka is a high-throughput distributed system—capable of processing hundreds of megabytes per second on even modest clusters—so there is no concern that our pipeline will not scale as demand grows. In addition, the Kafka Connect API focuses on parallelizing the work and can do this on a single node as well as by scaling out, depending on system requirements. We'll describe in the following sections how the platform allows data sources and sinks to split the work among multiple threads of execution and use the available CPU resources even when running on a single machine.

Kafka also supports several types of compression, allowing users and admins to control the use of network and storage resources as the throughput requirements increase.

Data Formats

One of the most important considerations in a data pipeline is reconciling different data formats and data types. The data types supported vary among different databases and other storage systems. You may be loading XMLs and relational data into Kafka, using Avro within Kafka, and then need to convert data to JSON when writing it to Elasticsearch, to Parquet when writing to HDFS, and to CSV when writing to S3.

Kafka itself and the Connect API are completely agnostic when it comes to data formats. As we've seen in previous chapters, producers and consumers can use any serializer to represent data in any format that works for you. Kafka Connect has its own in-memory objects that include data types and schemas, but as we'll soon discuss, it allows for pluggable converters to allow storing these records in any format. This means that no matter which data format you use for Kafka, it does not restrict your choice of connectors.

Many sources and sinks have a schema; we can read the schema from the source with the data, store it, and use it to validate compatibility or even update the schema in the sink database. A classic example is a data pipeline from MySQL to Snowflake. If someone added a column in MySQL, a great pipeline will make sure the column gets added to Snowflake too as we are loading new data into it.

In addition, when writing data from Kafka to external systems, sink connectors are responsible for the format in which the data is written to the external system. Some connectors choose to make this format pluggable. For example, the S3 connector allows a choice between Avro and Parquet formats.

It is not enough to support different types of data. A generic data integration framework should also handle differences in behavior between various sources and sinks. For example, Syslog is a source that pushes data, while relational databases require the framework to pull data out. HDFS is append-only and we can only write data to it, while most systems allow us to both append data and update existing records.

Transformations

Transformations are more controversial than other requirements. There are generally two approaches to building data pipelines: ETL and ELT. ETL, which stands for *Extract-Transform-Load*, means that the data pipeline is responsible for making modifications to the data as it passes through. It has the perceived benefit of saving time and storage because you don't need to store the data, modify it, and store it again. Depending on the transformations, this benefit is sometimes real, but sometimes it shifts the burden of computation and storage to the data pipeline itself, which may or may not be desirable. The main drawback of this approach is that the transformations that happen to the data in the pipeline may tie the hands of those who wish to process the data further down the pipe. If the person who built the pipeline between MongoDB and MySQL decided to filter certain events or remove fields from records, all the users and applications who access the data in MySQL will only have access to partial data. If they require access to the missing fields, the pipeline needs to be rebuilt, and historical data will require reprocessing (assuming it is available).

ELT stands for *Extract-Load-Transform* and means that the data pipeline does only minimal transformation (mostly around data type conversion), with the goal of making sure the data that arrives at the target is as similar as possible to the source data. In these systems, the target system collects "raw data" and all required processing is done at the target system. The benefit here is that the system provides maximum flexibility to users of the target system, since they have access to all the data. These systems also tend to be easier to troubleshoot since all data processing is limited to one system rather than split between the pipeline and additional applications. The drawback is that the transformations take CPU and storage resources at the

target system. In some cases, these systems are expensive and there is strong motivation to move computation off those systems when possible.

Kafka Connect includes the Single Message Transformation feature, which transforms records while they are being copied from a source to Kafka, or from Kafka to a target. This includes routing messages to different topics, filtering messages, changing data types, redacting specific fields, and more. More complex transformations that involve joins and aggregations are typically done using Kafka Streams, and we will explore those in detail in a separate chapter.

> **WARNING**
>
> When building an ETL system with Kafka, keep in mind that Kafka allows you to build one-to-many pipelines, where the source data is written to Kafka once and then consumed by multiple applications and written to multiple target systems. Some preprocessing and cleanup is expected, such as standardizing timestamps and data types, adding lineage, and perhaps removing personal information—transformations that will benefit all consumers of the data. But don't prematurely clean and optimize the data on ingest because it might be needed less refined elsewhere.

Security

Security should always be a concern. In terms of data pipelines, the main security concerns are usually:

- Who has access to the data that is ingested into Kafka?
- Can we make sure the data going through the pipe is encrypted? This is mainly a concern for data pipelines that cross datacenter boundaries.
- Who is allowed to make modifications to the pipelines?

- If the data pipeline needs to read or write from access-controlled locations, can it authenticate properly?
- Is our PII (Personally Identifiable Information) handling compliant with laws and regulations regarding its storage, access and use?

Kafka allows encrypting data on the wire, as it is piped from sources to Kafka and from Kafka to sinks. It also supports authentication (via SASL) and authorization—so you can be sure that if a topic contains sensitive information, it can't be piped into less secured systems by someone unauthorized. Kafka also provides an audit log to track access—unauthorized and authorized. With some extra coding, it is also possible to track where the events in each topic came from and who modified them, so you can provide the entire lineage for each record.

Kafka security is discussed in detail in Chapter 11. However, Kafka Connect and its connectors need to be able to connect to, and authenticate with, external data systems, and configuration of connectors will include credentials for authenticating with external data systems.

These days it is not recommended to store credentials in configuration files, since this means that the configuration files have to be handled with extra care and have restricted access. A common solution is to use an external secret management system such as HashiCorp Vault. Kafka Connect includes support for external secret configuration. Apache Kafka only includes the framework that allows introduction of pluggable external config providers, an example provider that reads configuration from a file, and there are community-developed external config providers that integrate with Vault, AWS, and Azure.

Failure Handling

Assuming that all data will be perfect all the time is dangerous. It is important to plan for failure handling in advance. Can we prevent faulty records from ever making it into the pipeline? Can we recover from records that cannot be parsed? Can bad records get fixed (perhaps by a human) and reprocessed? What if the bad event looks exactly like a normal event and you only discover the problem a few days later?

Because Kafka can be configured to store all events for long periods of time, it is possible to go back in time and recover from errors when needed. This also allows replaying the events stored in Kafka to the target system if they were lost.

Coupling and Agility

A desirable characteristic of data pipeline implementation is to decouple the data sources and data targets. There are multiple ways accidental coupling can happen:

Ad hoc pipelines

Some companies end up building a custom pipeline for each pair of applications they want to connect. For example, they use Logstash to dump logs to Elasticsearch, Flume to dump logs to HDFS, Oracle GoldenGate to get data from Oracle to HDFS, Informatica to get data from MySQL and XML to Oracle, and so on. This tightly couples the data pipeline to the specific end points and creates a mess of integration points that requires significant effort to deploy, maintain, and monitor. It also means that every new system the company adopts will require building

additional pipelines, increasing the cost of adopting new technology, and inhibiting innovation.

Loss of metadata

If the data pipeline doesn't preserve schema metadata and does not allow for schema evolution, you end up tightly coupling the software producing the data at the source and the software that uses it at the destination. Without schema information, both software products need to include information on how to parse the data and interpret it. If data flows from Oracle to HDFS and a DBA added a new field in Oracle without preserving schema information and allowing schema evolution, either every app that reads data from HDFS will break or all the developers will need to upgrade their applications at the same time. Neither option is agile. With support for schema evolution in the pipeline, each team can modify their applications at their own pace without worrying that things will break down the line.

Extreme processing

As we mentioned when discussing data transformations, some processing of data is inherent to data pipelines. After all, we are moving data between different systems where different data formats make sense and different use cases are supported. However, too much processing ties all the downstream systems to decisions made when building the pipelines about which fields to preserve, how to aggregate data, etc. This often leads to constant changes to the pipeline as requirements of downstream applications change, which isn't agile, efficient, or safe. The more agile way is to preserve as much of the raw data as possible and allow downstream apps, including

Kafka Streams apps, to make their own decisions regarding data processing and aggregation.

When to Use Kafka Connect Versus Producer and Consumer

When writing to Kafka or reading from Kafka, you have the choice between using traditional producer and consumer clients, as described in Chapters 3 and 4, or using the Kafka Connect API and the connectors, as we'll describe in the following sections. Before we start diving into the details of Kafka Connect, you may already be wondering, "When do I use which?"

As we've seen, Kafka clients are clients embedded in your own application. It allows your application to write data to Kafka or to read data from Kafka. Use Kafka clients when you can modify the code of the application that you want to connect an application to and when you want to either push data into Kafka or pull data from Kafka.

You will use Connect to connect Kafka to datastores that you did not write and whose code or APIs you cannot or will not modify. Connect will be used to pull data from the external datastore into Kafka or push data from Kafka to an external store. To use Kafka Connect, you need a connector for the datastore to which you want to connect, and nowadays these connectors are plentiful. This means that in practice, users of Kafka Connect only need to write configuration files.

If you need to connect Kafka to a datastore and a connector does not exist yet, you can choose between writing an app using the Kafka clients or the Connect API. Connect is recommended because it provides out-of-the-box features

like configuration management, offset storage, parallelization, error handling, support for different data types, and standard management REST APIs. Writing a small app that connects Kafka to a datastore sounds simple, but there are many little details you will need to handle concerning data types and configuration that make the task nontrivial. What's more, you will need to maintain this pipeline app and document it, and your teammates will need to learn how to use it. Kafka Connect is a standard part of the Kafka ecosystem, and it handles most of this for you, allowing you to focus on transporting data to and from the external stores.

Kafka Connect

Kafka Connect is a part of Apache Kafka and provides a scalable and reliable way to copy data between Kafka and other datastores. It provides APIs and a runtime to develop and run *connector plug-ins*—libraries that Kafka Connect executes and that are responsible for moving the data. Kafka Connect runs as a cluster of *worker processes*. You install the connector plug-ins on the workers and then use a REST API to configure and manage *connectors*, which run with a specific configuration. *Connectors* start additional *tasks* to move large amounts of data in parallel and use the available resources on the worker nodes more efficiently. Source connector tasks just need to read data from the source system and provide Connect data objects to the worker processes. Sink connector tasks get connector data objects from the workers and are responsible for writing them to the target data system. Kafka Connect uses *convertors* to support storing those data objects in Kafka in different formats—JSON format support is part of Apache Kafka, and the Confluent Schema Registry provides Avro,

Kafka Streams version 2.6 or later includes a more efficient exactly-once implementation that requires Kafka brokers of version 2.5 or later. This efficient implementation can be enabled by setting `processing.guarantee` to `exactly_once_beta`.

Stream Processing Design Patterns

Every stream processing system is different—from the basic combination of a consumer, processing logic, and producer, to involved clusters like Spark Streaming with its machine learning libraries, and much in between. But there are some basic design patterns, which are known solutions to common requirements of stream processing architectures. We'll review a few of those well-known patterns and show how they are used with a few examples.

Single-Event Processing

The most basic pattern of stream processing is the processing of each event in isolation. This is also known as a *map/filter pattern* because it is commonly used to filter unnecessary events from the stream or transform each event. (The term *map* is based on the map/reduce pattern in which the map stage transforms events and the reduce stage aggregates them.)

In this pattern, the stream processing app consumes events from the stream, modifies each event, and then produces the events to another stream. An example is an app that reads log messages from a stream and writes `ERROR` events into a high-priority stream and the rest of the events into a low-priority stream. Another example is an application that reads events from a stream and modifies them from JSON to Avro. Such applications do not need to maintain state within the application because each event can be handled

independently. This means that recovering from app failures or load-balancing is incredibly easy as there is no need to recover state; we can simply hand off the events to another instance of the app to process.

This pattern can be easily handled with a simple producer and consumer, as seen in Figure 14-3.

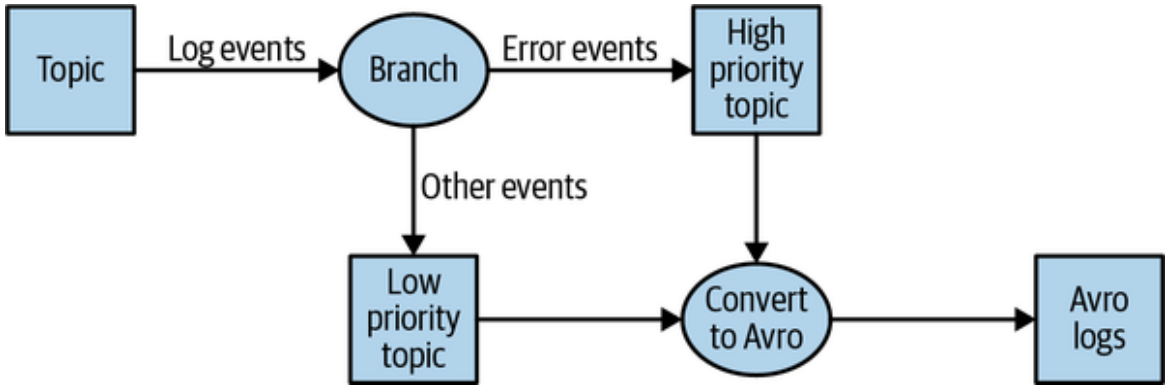

Figure 14-3. Single-event processing topology

Processing with Local State

Most stream processing applications are concerned with aggregating information, especially window aggregation. An example of this is finding the minimum and maximum stock prices for each day of trading and calculating a moving average.

These aggregations require maintaining a *state*. In our example, in order to calculate the minimum and average price each day, we need to store the minimum value, the sum, and the number of records we've seen up until the current time.

All this can be done using *local* state (rather than a shared state) because each operation in our example is a *group by* aggregate. That is, we perform the aggregation per stock symbol, not on the entire stock market in general. We use a Kafka partitioner to make sure that all events with the

same stock symbol are written to the same partition. Then, each instance of the application will get all the events from the partitions that are assigned to it (this is a Kafka consumer guarantee). This means that each instance of the application can maintain state for the subset of stock symbols that are written to the partitions that are assigned to it. See Figure 14-4.

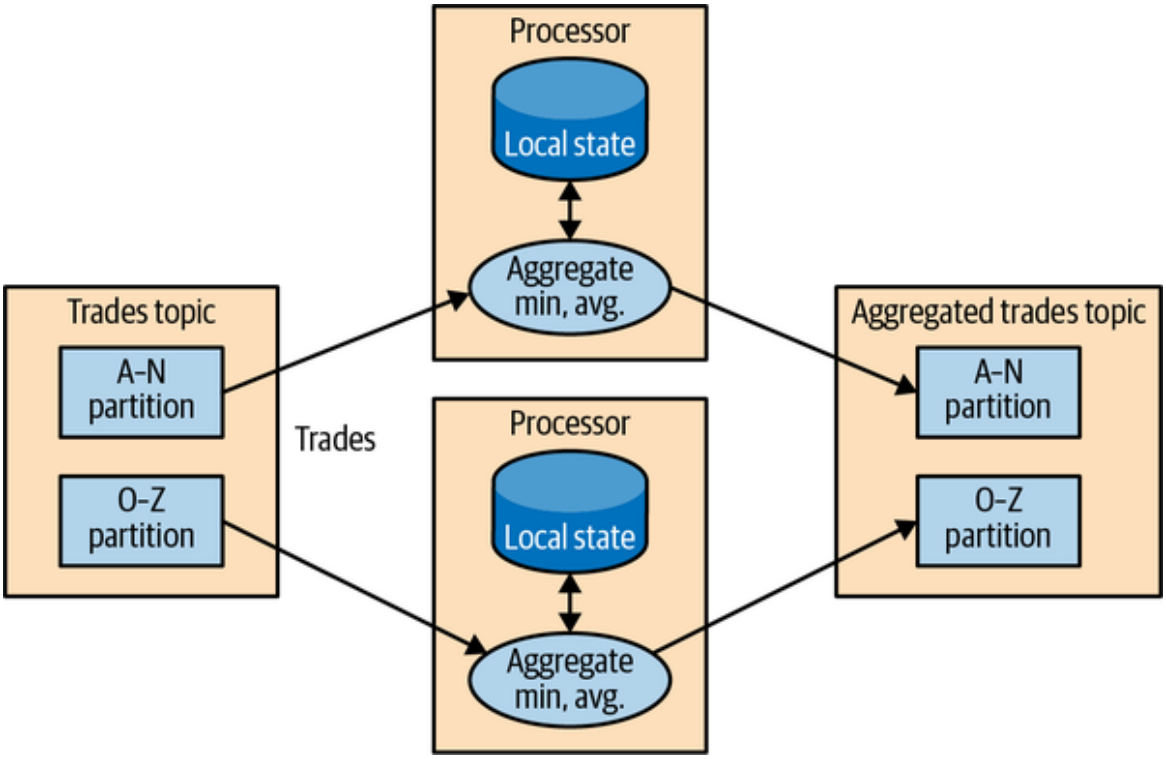

Figure 14-4. Topology for event processing with local state

Stream processing applications become significantly more complicated when the application has local state. There are several issues a stream processing application must address:

Memory usage

The local state ideally fits into the memory available to the application instance. Some local stores allow spilling to disk, but this has significant performance impact.

Persistence

We need to make sure the state is not lost when an application instance shuts down and that the state can be recovered when the instance starts again or is replaced by a different instance. This is something that Kafka Streams handles very well—local state is stored in-memory using embedded RocksDB, which also persists the data to disk for quick recovery after restarts. But all the changes to the local state are also sent to a Kafka topic. If a stream's node goes down, the local state is not lost—it can be easily re-created by rereading the events from the Kafka topic. For example, if the local state contains "current minimum for IBM = 167.19," we store this in Kafka so that later we can repopulate the local cache from this data. Kafka uses log compaction for these topics to make sure they don't grow endlessly and that re-creating the state is always feasible.

Rebalancing

Partitions sometimes get reassigned to a different consumer. When this happens, the instance that loses the partition must store the last good state, and the instance that receives the partition must know to recover the correct state.

Stream processing frameworks differ in how much they help the developer manage the local state they need. If our application requires maintaining local state, we make sure to check the framework and its guarantees. We'll include a short comparison guide at the end of the chapter, but as we all know, software changes quickly and stream processing frameworks doubly so.

Multiphase Processing/Repartitioning

Local state is great if we need a *group by* type of aggregate. But what if we need a result that uses all available information? For example, suppose we want to publish the top 10 stocks each day—the 10 stocks that gained the most from opening to closing during each day of trading. Obviously, nothing we do locally on each application instance is enough because all the top 10 stocks could be in partitions assigned to other instances. What we need is a two-phase approach. First, we calculate the daily gain/loss for each stock symbol. We can do this on each instance with a local state. Then we write the results to a new topic with a single partition. This partition will be read by a single application instance that can then find the top 10 stocks for the day. The second topic, which contains just the daily summary for each stock symbol, is obviously much smaller with significantly less traffic than the topics that contain the trades themselves, and therefore it can be processed by a single instance of the application. Sometimes more steps are needed to produce the result. See Figure 14-5.

This type of multiphase processing is very familiar to those who write MapReduce code, where you often have to resort to multiple reduce phases. If you've ever written map-reduce code, you'll remember that you needed a separate app for each reduce step. Unlike MapReduce, most stream processing frameworks allow including all steps in a single app, with the framework handling the details of which application instance (or worker) will run each step.

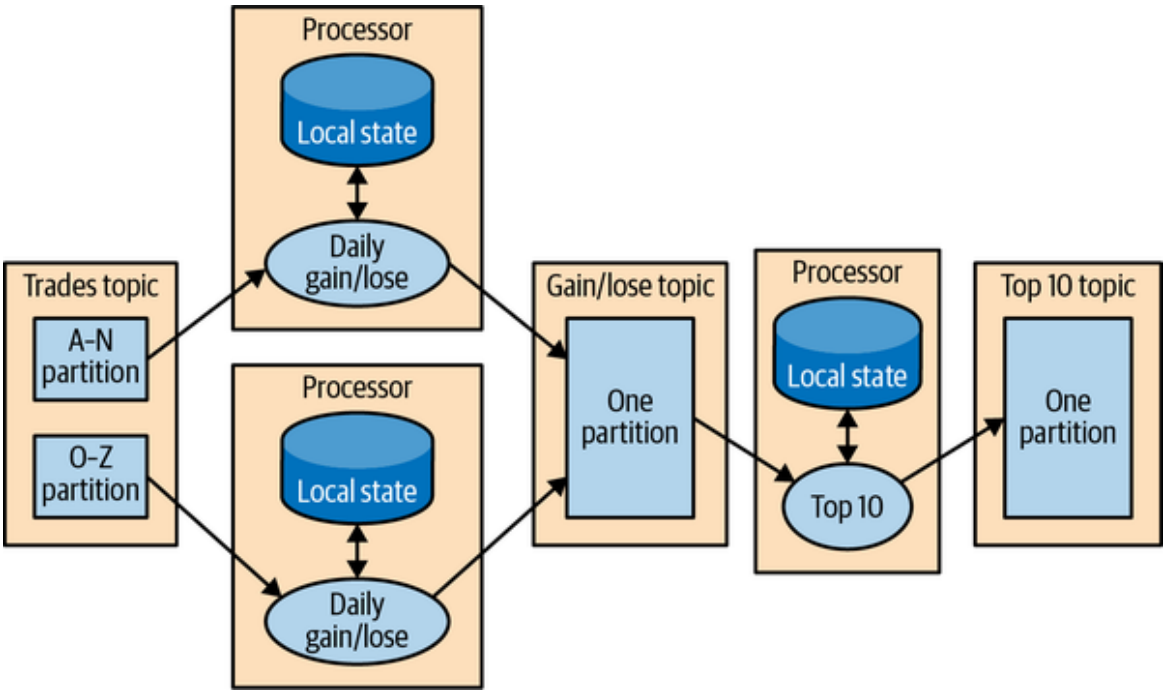

Figure 14-5. Topology that includes both local state and repartitioning steps

Processing with External Lookup: Stream-Table Join

Sometimes stream processing requires integration with data external to the stream—validating transactions against a set of rules stored in a database or enriching clickstream information with data about the users who clicked.

The obvious idea on how to perform an external lookup for data enrichment is something like this: for every click event in the stream, look up the user in the profile database and write an event that includes the original click, plus the user age and gender, to another topic. See Figure 14-6.

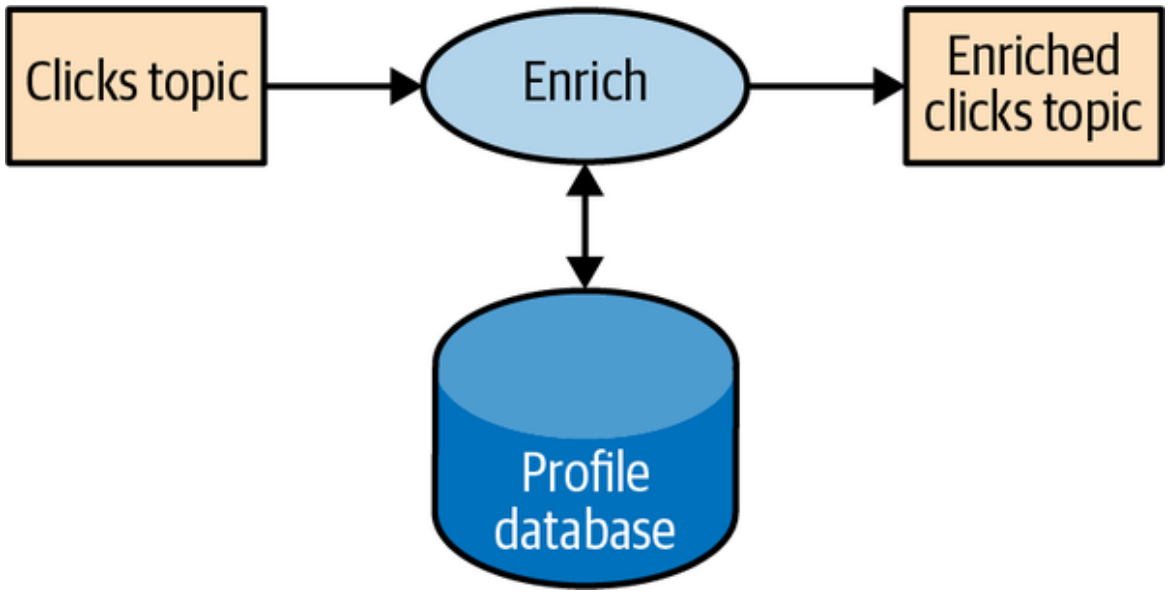

Figure 14-6. Stream processing that includes an external data source

The problem with this obvious idea is that an external lookup adds significant latency to the processing of every record—usually between 5 and 15 milliseconds. In many cases, this is not feasible. Often the additional load this places on the external data store is also not acceptable—stream processing systems can often handle 100K–500K events per second, but the database can only handle perhaps 10K events per second at reasonable performance. There is also added complexity around availability—our application will need to handle situations when the external DB is not available.

To get good performance and availability, we need to cache the information from the database in our stream processing application. Managing this cache can be challenging though—how do we prevent the information in the cache from getting stale? If we refresh events too often, we are still hammering the database, and the cache isn't helping much. If we wait too long to get new events, we are doing stream processing with stale information.

But if we can capture all the changes that happen to the database table in a stream of events, we can have our stream processing job listen to this stream and update the cache based on database change events. Capturing changes to the database as events in a stream is known as *change data capture* (CDC), and Kafka Connect has multiple connectors capable of performing CDC and converting database tables to a stream of change events. This allows us to keep our own private copy of the table and be notified whenever there is a database change event so we can update our own copy accordingly. See Figure 14-7.

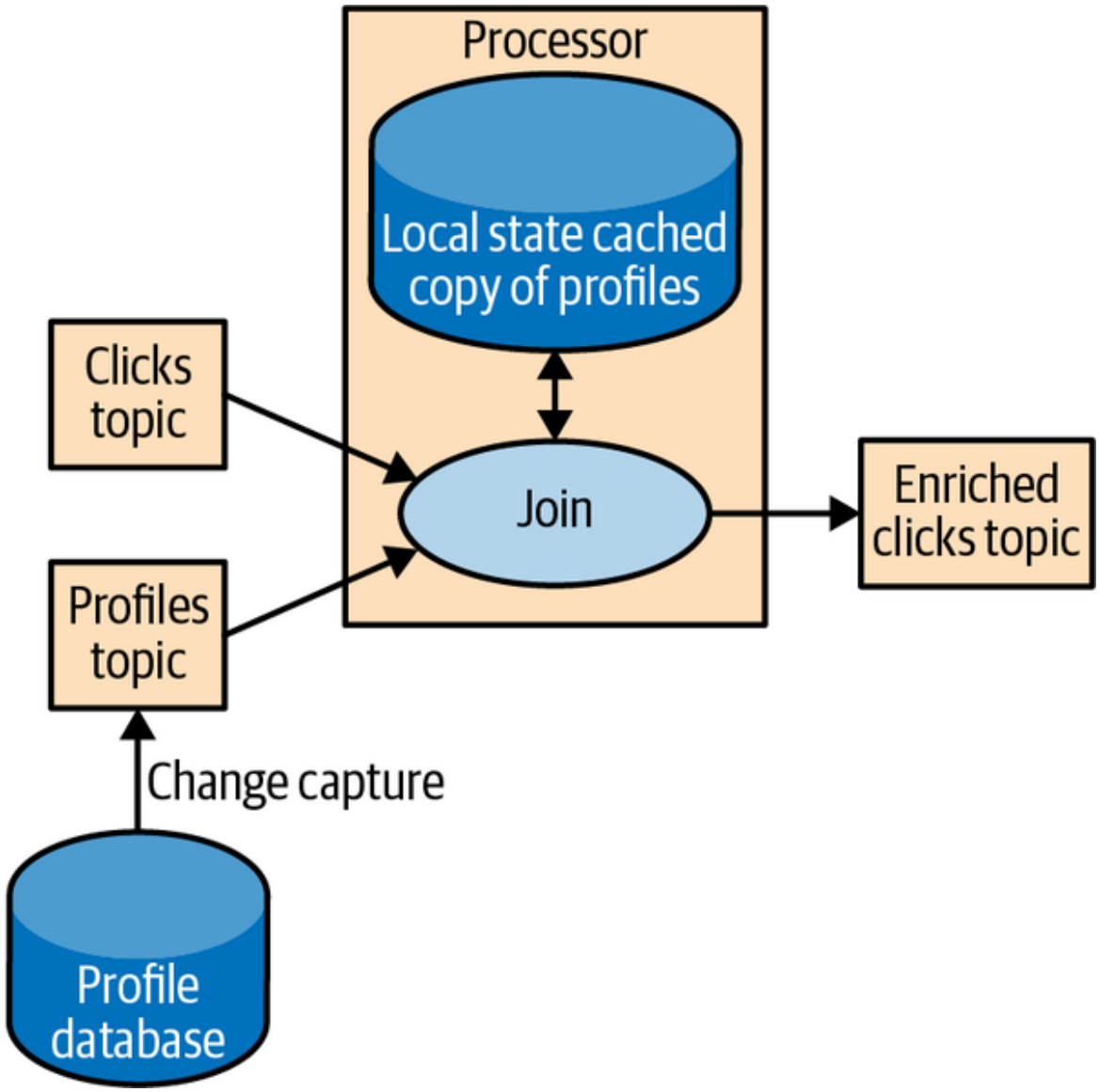

Figure 14-7. Topology joining a table and a stream of events, removing the need to involve an external data source in stream processing

Then, when we get click events, we can look up the `user_id` at our local state and enrich the event. And because we are using a local state, this scales a lot better and will not affect the database and other apps using it.

We refer to this as a *stream-table join* because one of the streams represents changes to a locally cached table.

Table-Table Join

In the previous section we discussed how a table and a stream of update events are equivalent. We've discussed in detail how this works when joining a stream and a table. There is no reason why we can't have those materialized tables in both sides of the join operation.

Joining two tables is always nonwindowed and joins the current state of both tables at the time the operation is performed. With Kafka Streams, we can perform an `equi-join`, in which both tables have the same key that is partitioned in the same way, and therefore the join operation can be efficiently distributed between a large number of application instances and machines.

Kafka Streams also supports `foreign-key join` of two tables—the key of one stream or table is joined with an arbitrary field from another stream or table. You can learn more about how it works in "Crossing the Streams", a talk from Kafka Summit 2020, or the more in-depth blog post.

Streaming Join

Sometimes we want to join two real event streams rather than a stream with a table. What makes a stream "real"? If you recall the discussion at the beginning of the chapter, streams are unbounded. When we use a stream to represent a table, we can ignore most of the history in the stream because we only care about the current state in the table. But when we join two streams, we are joining the entire history, trying to match events in one stream with events in the other stream that have the same key and happened in the same time windows. This is why a streaming join is also called a *windowed join*.

For example, let's say that we have one stream with search queries that people entered into our website and another

stream with clicks, which include clicks on search results. We want to match search queries with the results they clicked on so that we will know which result is most popular for which query. Obviously, we want to match results based on the search term but only match them within a certain time window. We assume the result is clicked seconds after the query was entered into our search engine. So we keep a small, few-seconds-long window on each stream and match the results from each window. See Figure 14-8.

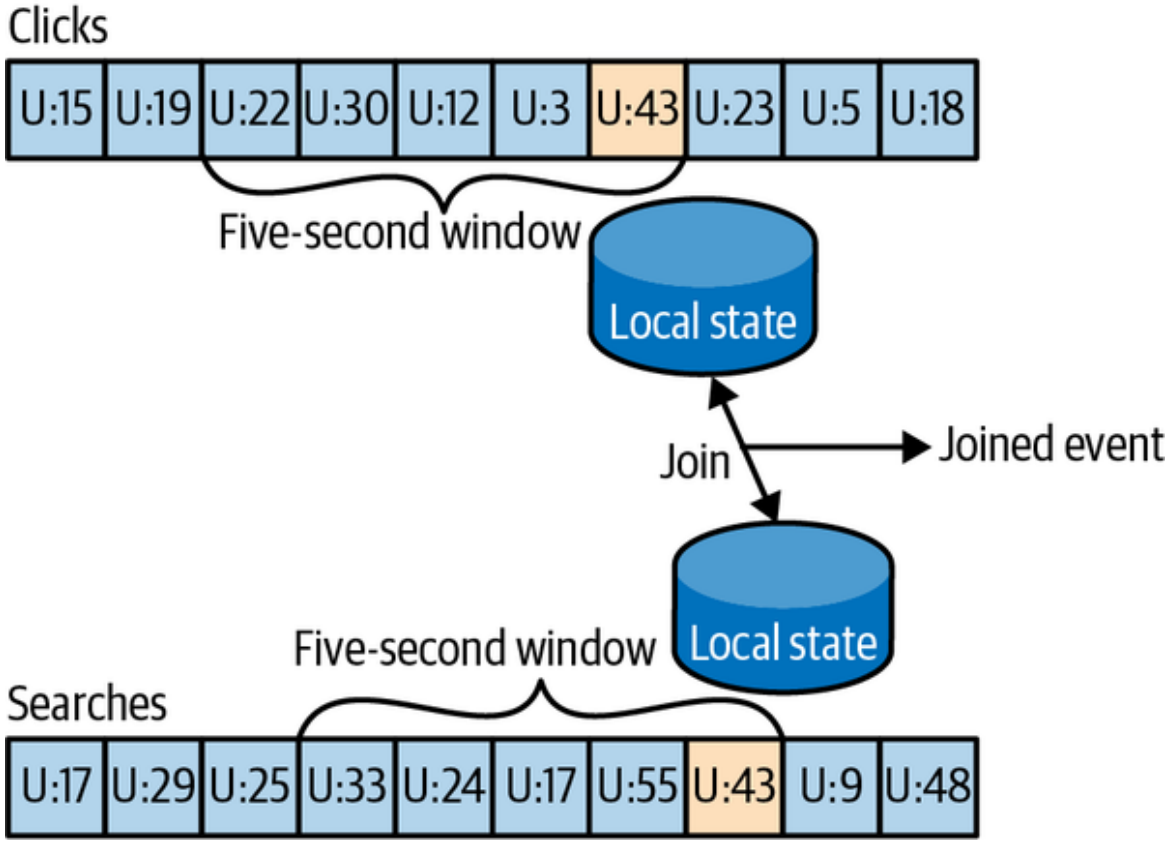

Figure 14-8. Joining two streams of events; these joins always involve a moving time window

Kafka Streams supports `equi-joins`, where streams, queries, and clicks are partitioned on the same keys, which are also the join keys. This way, all the click events from `user_id:42` end up in partition 5 of the clicks topic, and all the search

events for `user_id:42` end up in partition 5 of the search topic. Kafka Streams then makes sure that partition 5 of both topics is assigned to the same task. So this task sees all the relevant events for `user_id:42`. It maintains the join window for both topics in its embedded RocksDB state store, and this is how it can perform the join.

Out-of-Sequence Events

Handling events that arrive at the stream at the wrong time is a challenge not just in stream processing but also in traditional ETL systems. Out-of-sequence events happen quite frequently and expectedly in IoT scenarios (Figure 14-9). For example, a mobile device loses WiFi signal for a few hours and sends a few hours' worth of events when it reconnects. This also happens when monitoring network equipment (a faulty switch doesn't send diagnostics signals until it is repaired) or manufacturing (network connectivity in plants is notoriously unreliable, especially in developing countries).

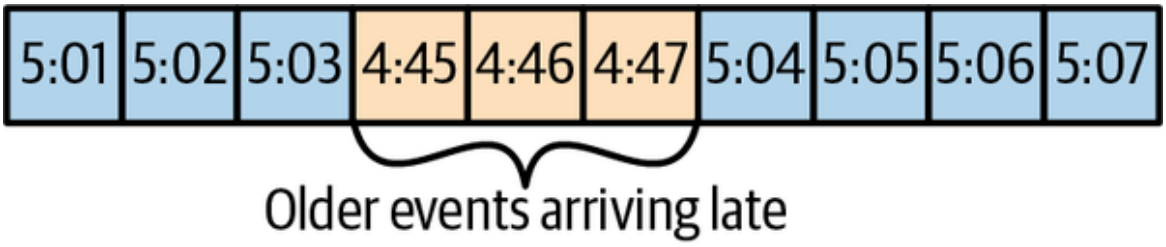

Figure 14-9. Out-of-sequence events

Our streams applications need to be able to handle those scenarios. This typically means the application has to do the following:

- Recognize that an event is out of sequence—this requires that the application examines the event time and discovers that it is older than the current time.

- Define a time period during which it will attempt to reconcile out-of-sequence events. Perhaps a three-hour delay should be reconciled, and events over three weeks old can be thrown away.
- Have an in-band capability to reconcile this event. This is the main difference between streaming apps and batch jobs. If we have a daily batch job and a few events arrived after the job completed, we can usually just rerun yesterday's job and update the events. With stream processing, there is no "rerun yesterday's job"—the same continuous process needs to handle both old and new events at any given moment.
- Be able to update results. If the results of the stream processing are written into a database, a *put* or *update* is enough to update the results. If the stream app sends results by email, updates may be trickier.

Several stream processing frameworks, including Google's Dataflow and Kafka Streams, have built-in support for the notion of event time independent of the processing time, and the ability to handle events with event times that are older or newer than the current processing time. This is typically done by maintaining multiple aggregation windows available for update in the local state and giving developers the ability to configure how long to keep those window aggregates available for updates. Of course, the longer the aggregation windows are kept available for updates, the more memory is required to maintain the local state.

The Kafka Streams API always writes aggregation results to result topics. Those are usually `compacted topics`, which means that only the latest value for each key is preserved.

In case the results of an aggregation window need to be updated as a result of a late event, Kafka Streams will simply write a new result for this aggregation window, which will effectively replace the previous result.

Reprocessing

The last important pattern is reprocessing events. There are two variants of this pattern:

- We have an improved version of our stream processing application. We want to run the new version of the application on the same event stream as the old, produce a new stream of results that does not replace the first version, compare the results between the two versions, and at some point move clients to use the new results instead of the existing ones.
- The existing stream processing app is buggy. We fix the bug, and we want to reprocess the event stream and recalculate our results

The first use case is made simple by the fact that Apache Kafka stores the event streams in their entirety for long periods of time in a scalable data store. This means that having two versions of a stream processing application writing two result streams only requires the following:

- Spinning up the new version of the application as a new consumer group
- Configuring the new version to start processing from the first offset of the input topics (so it will get its own copy of all events in the input streams)

- Letting the new application continue processing, and switching the client applications to the new result stream when the new version of the processing job has caught up

The second use case is more challenging—it requires "resetting" an existing app to start processing back at the beginning of the input streams, resetting the local state (so we won't mix results from the two versions of the app), and possibly cleaning the previous output stream. While Kafka Streams has a tool for resetting the state for a stream processing app, our recommendation is to try to use the first method whenever sufficient capacity exists to run two copies of the app and generate two result streams. The first method is much safer—it allows switching back and forth between multiple versions and comparing results between versions, and doesn't risk losing critical data or introducing errors during the cleanup process.

Interactive Queries

As discussed previously, stream processing applications have state, and this state can be distributed among many instances of the application. Most of the time the users of stream processing applications get the results of the processing by reading them from an output topic. In some cases, however, it is desirable to take a shortcut and read the results from the state store itself. This is common when the result is a table (e.g., the top 10 best-selling books) and the stream of results is really a stream of updates to this table—it is much faster and easier to just read the table directly from the stream processing application state.

Kafka Streams includes flexible APIs for querying the state of a stream processing application.

Kafka Streams by Example

To demonstrate how these patterns are implemented in practice, we'll show a few examples using the Apache Kafka Streams API. We are using this specific API because it is relatively simple to use and it ships with Apache Kafka, which we already have access to. It is important to remember that the patterns can be implemented in any stream processing framework and library—the patterns are universal, but the examples are specific.

Apache Kafka has two stream APIs—a low-level Processor API and a high-level Streams DSL. We will use the Kafka Streams DSL in our examples. The DSL allows us to define the stream processing application by defining a chain of transformations to events in the streams. Transformations can be as simple as a filter or as complex as a stream-to-stream join. The lower-level API allows us to create our own transformations. To learn more about the low-level Processor API, the developer guide has detailed information, and the presentation "Beyond the DSL" is a great introduction.

An application that uses the DSL API always starts with using the `StreamsBuilder` to create a processing *topology*—a directed acyclic graph (DAG) of transformations that are applied to the events in the streams. Then we create a `KafkaStreams` execution object from the topology. Starting the `KafkaStreams` object will start multiple threads, each applying the processing topology to events in the stream. The processing will conclude when we close the `KafkaStreams` object.

We'll look at a few examples that use Kafka Streams to implement some of the design patterns we just discussed. A simple word count example will be used to demonstrate the

map/filter pattern and simple aggregates. Then we'll move to an example where we calculate different statistics on stock market trades, which will allow us to demonstrate window aggregations. Finally, we'll use ClickStream enrichment as an example to demonstrate streaming joins.

Word Count

Let's walk through an abbreviated word count example for Kafka Streams. You can find the full example on GitHub.

The first thing you do when creating a stream processing app is configure Kafka Streams. Kafka Streams has a large number of possible configurations, which we won't discuss here, but you can find them in the documentation. In addition, you can configure the producer and consumer embedded in Kafka Streams by adding any producer or consumer config to the `Properties` object:

```
public class WordCountExample {

    public static void main(String[] args) throws Exception{

        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG,
          "wordcount"); ❶
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
          "localhost:9092"); ❷
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
          Serdes.String().getClass().getName()); ❸
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
          Serdes.String().getClass().getName());
```

❶ Every Kafka Streams application must have an application ID. It is used to coordinate the instances of the application and also when naming the internal local stores and the topics related to them. This name must be unique for each Kafka Streams application working with the same Kafka cluster.

❷ The Kafka Streams application always reads data from Kafka topics and writes its output to Kafka topics. As we'll discuss later, Kafka Streams applications also use Kafka for coordination. So we had better tell our app where to find Kafka.

❸ When reading and writing data, our app will need to serialize and deserialize, so we provide default Serde classes. If needed, we can override these defaults later when building the streams topology.

Now that we have the configuration, let's build our streams topology:

```
StreamsBuilder builder = new StreamsBuilder(); ❶

KStream<String, String> source =
  builder.stream("wordcount-input");


final Pattern pattern = Pattern.compile("\\W+");

KStream<String, String> counts  = source.flatMapValues(value->
  Arrays.asList(pattern.split(value.toLowerCase()))) ❷
        .map((key, value) -> new KeyValue<String,
           String>(value, value))
        .filter((key, value) -> (!value.equals("the"))) ❸
        .groupByKey() ❹
        .count().mapValues(value->
           Long.toString(value)).toStream();❺
counts.to("wordcount-output"); ❻
```

❶ We create a `StreamsBuilder` object and start defining a stream by pointing at the topic we'll use as our input.

❷ Each event we read from the source topic is a line of words; we split it up using a regular expression into a series of individual words. Then we take each word (currently a value of the event record) and put it in the

event record key so it can be used in a group-by operation.

❸ We filter out the word *the*, just to show how easy filtering is.

❹ And we group by key, so we now have a collection of events for each unique word.

❺ We count how many events we have in each collection. The result of counting is a `Long` data type. We convert it to a `String` so it will be easier for humans to read the results.

❻ Only one thing left—write the results back to Kafka.

Now that we have defined the flow of transformations that our application will run, we just need to…run it:

```
KafkaStreams streams = new KafkaStreams(builder.build(), props); ❶

streams.start(); ❷

// usually the stream application would be running forever,
// in this example we just let it run for some time and stop
Thread.sleep(5000L);

streams.close(); ❸
```

❶ Define a `KafkaStreams` object based on our topology and the properties we defined.

❷ Start Kafka Streams.

❸ After a while, stop it.

That's it! In just a few short lines, we demonstrated how easy it is to implement a single event processing pattern (we applied a map and a filter on the events). We repartitioned the data by adding a group-by operator and

then maintained simple local state when we counted the number of records that have each word as a key. Then we maintained simple local state when we counted the number of times each word appeared.

At this point, we recommend running the full example. The README in the GitHub repository contains instructions on how to run the example.

Note that we can run the entire example on our machine without installing anything except Apache Kafka. If our input topic contains multiple partitions, we can run multiple instances of the `WordCount` application (just run the app in several different terminal tabs), and we have our first Kafka Streams processing cluster. The instances of the `WordCount` application talk to one another and coordinate the work. One of the biggest barriers to entry for some stream processing frameworks is that local mode is very easy to use, but then to run a production cluster, we need to install YARN or Mesos, then install the processing framework on all those machines, and then learn how to submit our app to the cluster. With the Kafka's Streams API, we just start multiple instances of our app—and we have a cluster. The exact same app is running on our development machine and in production.

Stock Market Statistics

The next example is more involved—we will read a stream of stock market trading events that include the stock ticker, ask price, and ask size. In stock market trades, *ask price* is what a seller is asking for, whereas *bid price* is what the buyer is suggesting to pay. *Ask size* is the number of shares the seller is willing to sell at that price. For simplicity of the example, we'll ignore bids completely. We also won't

include a timestamp in our data; instead, we'll rely on event time populated by our Kafka producer.

We will then create output streams that contain a few windowed statistics:

- Best (i.e., minimum) ask price for every five-second window
- Number of trades for every five-second window
- Average ask price for every five-second window

All statistics will be updated every second.

For simplicity, we'll assume our exchange only has 10 stock tickers trading in it. The setup and configuration are very similar to those we used in "Word Count":

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "stockstat");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, Constants.BROKER);
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
  Serdes.String().getClass().getName());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
  TradeSerde.class.getName());
```

The main difference is the Serde classes used. In "Word Count", we used strings for both key and value, and therefore used the `Serdes.String()` class as a serializer and deserializer for both. In this example, the key is still a string, but the value is a `Trade` object that contains the ticker symbol, ask price, and ask size. In order to serialize and deserialize this object (and a few other objects we used in this small app), we used the Gson library from Google to generate a JSON serializer and deserializer from our `Java` object. Then we created a small wrapper that created a Serde object from those. Here is how we created the Serde:

```
static public final class TradeSerde extends WrapperSerde<Trade> {
    public TradeSerde() {
        super(new JsonSerializer<Trade>(),
          new JsonDeserializer<Trade>(Trade.class));
    }
}
```

Nothing fancy, but remember to provide a Serde object for every object you want to store in Kafka—input, output, and, in some cases, intermediate results. To make this easier, we recommend generating these Serdes through a library like Gson, Avro, Protobuf, or something similar.

Now that we have everything configured, it's time to build our topology:

```
KStream<Windowed<String>, TradeStats> stats = source
    .groupByKey() ❶
    .windowedBy(TimeWindows.of(Duration.ofMillis(windowSize))
                           .advanceBy(Duration.ofSeconds(1))) ❷
    .aggregate( ❸
        () -> new TradeStats(),
        (k, v, tradestats) -> tradestats.add(v), ❹
        Materialized.<String, TradeStats, WindowStore<Bytes, byte[]>>
            as("trade-aggregates") ❺
           .withValueSerde(new TradeStatsSerde())) ❻
    .toStream() ❼
    .mapValues((trade) -> trade.computeAvgPrice()); ❽

stats.to("stockstats-output",
    Produced.keySerde(
      WindowedSerdes.timeWindowedSerdeFrom(String.class, windowSize))); ❾
```

❶ We start by reading events from the input topic and performing a `groupByKey()` operation. Despite its name, this operation does not do any grouping. Rather, it ensures that the stream of events is partitioned based on the record key. Since we wrote the data into a topic with a key and didn't modify the key before calling `groupByKey()`, the data is still partitioned by its key—so this method does nothing in this case.

❷ We define the window—in this case, a window of five seconds, advancing every second.

❸ After we ensure correct partitioning and windowing, we start the aggregation. The `aggregate` method will split the stream into overlapping windows (a five-second window every second) and then apply an aggregate method on all the events in the window. The first parameter this method takes is a new object that will contain the results of the aggregation—`Tradestats`, in our case. This is an object we created to contain all the statistics we are interested in for each time window—minimum price, average price, and number of trades.

❹ We then supply a method for actually aggregating the records—in this case, an `add` method of the `Tradestats` object is used to update the minimum price, number of trades, and total prices in the window with the new record.

❺ As mentioned in "Stream Processing Design Patterns", windowing aggregation requires maintaining a state and a local store in which the state will be maintained. The last parameter of the aggregate method is the configuration of the state store. `Materialized` is the store configuration object, and we configure the store name as `trade-aggregates`. This can be any unique name.

❻ As part of the state store configuration, we also provide a Serde object for serializing and deserializing the results of the aggregation (the `Tradestats` object).

❼ The results of the aggregation is a *table* with the ticker and the time window as the primary key and the aggregation result as the value. We are turning the table back into a stream of events.

❽

The last step is to update the average price—right now the aggregation results include the sum of prices and number of trades. We go over these records and use the existing statistics to calculate average price so we can include it in the output stream.

9. And finally, we write the results back to the `stockstats-output` stream. Since the results are part of a windowing operation, we create a `WindowedSerde` that stores the result in a windowed data format that includes the window timestamp. The window size is passed as part of the Serde, even though it isn't used in the serialization (deserialization requires the window size, because only the start time of the window is stored in the output topic).

After we define the flow, we use it to generate a `KafkaStreams` object and run it, just like we did in "Word Count".

This example shows how to perform windowed aggregation on a stream—probably the most popular use case of stream processing. One thing to notice is how little work was needed to maintain the local state of the aggregation—just provide a Serde and name the state store. Yet this application will scale to multiple instances and automatically recover from a failure of each instance by shifting processing of some partitions to one of the surviving instances. We will see more on how it is done in "Kafka Streams: Architecture Overview".

As usual, you can find the complete example, including instructions for running it, on GitHub.

ClickStream Enrichment

The last example will demonstrate streaming joins by enriching a stream of clicks on a website. We will generate a stream of simulated clicks, a stream of updates to a fictional profile database table, and a stream of web searches. We will then join all three streams to get a 360-degree view into each user activity. What did the users search for? What did they click as a result? Did they change their "interests" in their user profile? These kinds of joins provide a rich data collection for analytics. Product recommendations are often based on this kind of information—the user searched for bikes, clicked on links for "Trek," and is interested in travel, so we can advertise bikes from Trek, helmets, and bike tours to exotic locations like Nebraska.

Since configuring the app is similar to the previous examples, let's skip this part and take a look at the topology for joining multiple streams:

```
KStream<Integer, PageView> views =
    builder.stream(Constants.PAGE_VIEW_TOPIC,
      Consumed.with(Serdes.Integer(), new PageViewSerde())); ❶
KStream<Integer, Search> searches =
    builder.stream(Constants.SEARCH_TOPIC,
      Consumed.with(Serdes.Integer(), new SearchSerde()));
KTable<Integer, UserProfile> profiles =
    builder.table(Constants.USER_PROFILE_TOPIC,
      Consumed.with(Serdes.Integer(), new ProfileSerde())); ❷


KStream<Integer, UserActivity> viewsWithProfile = views.leftJoin(profiles, ❸
                (page, profile) -> {
                    if (profile != null)
                        return new UserActivity(
                          profile.getUserID(), profile.getUserName(),
                          profile.getZipcode(), profile.getInterests(),
                          "", page.getPage()); ❹
                    else
                       return new UserActivity(
                         -1, "", "", null, "", page.getPage());
                    });
```

```
KStream<Integer, UserActivity> userActivityKStream =
    viewsWithProfile.leftJoin(searches, ❺
      (userActivity, search) -> {
          if (search != null)
              userActivity.updateSearch(search.getSearchTerms()); ❻
          else
              userActivity.updateSearch("");
          return userActivity;
      },
      JoinWindows.of(Duration.ofSeconds(1)).before(Duration.ofSeconds(0)),  ❼
                  StreamJoined.with(Serdes.Integer(),  ❽
                                    new UserActivitySerde(),
                                    new SearchSerde()));
```

❶ First, we create a streams objects for the two streams we want to join—clicks and searches. When we create the stream object, we pass the input topic and the key and value Serde that will be used when consuming records out of the topic and deserializing them into input objects.

❷ We also define a `KTable` for the user profiles. A `KTable` is a materialized store that is updated through a stream of changes.

❸ Then we enrich the stream of clicks with user profile information by joining the stream of events with the profile table. In a stream-table join, each event in the stream receives information from the cached copy of the profile table. We are doing a left-join, so clicks without a known user will be preserved.

❹ This is the `join` method—it takes two values, one from the stream and one from the record, and returns a third value. Unlike in databases, we get to decide how to combine the two values into one result. In this case, we created one `activity` object that contains both the user details and the page viewed.

❺

Next, we want to `join` the click information with searches performed by the same user. This is still a left `join`, but now we are joining two streams, not streaming to a table.

❻ This is the `join` method—we simply add the search terms to all the matching page views.

❼ This is the interesting part—a *stream-to-stream join* is a join with a time window. Joining all clicks and searches for each user doesn't make much sense—we want to join each search with clicks that are related to it, that is, clicks that occurred a short period of time after the search. So we define a join window of one second. We invoke `of` to create a window of one second before and after each search, and then we call `before` with a zero-seconds interval to make sure we only join clicks that happen one second after each search and not before. The results will include relevant clicks, search terms, and the user profile. This will allow a full analysis of searches and their results.

❽ We define the Serde of the join result here. This includes a Serde for the key that both sides of the join have in common and the Serde for both values that will be included in the result of the join. In this case, the key is the user ID, so we use a simple `Integer` Serde.

After we define the flow, we use it to generate a `KafkaStreams` object and run it, just like we did in "Word Count".

This example shows two different join patterns possible in stream processing. One joins a stream with a table to enrich all streaming events with information in the table. This is similar to joining a fact table with a dimension when

running queries on a data warehouse. The second example joins two streams based on a time window. This operation is unique to stream processing.

As usual, you can find the complete example, including instructions for running it, on GitHub.

Kafka Streams: Architecture Overview

The examples in the previous section demonstrated how to use the Kafka Streams API to implement a few well-known stream processing design patterns. But to understand better how Kafka's Streams library actually works and scales, we need to peek under the covers and understand some of the design principles behind the API.

Building a Topology

Every streams application implements and executes one *topology*. Topology (also called DAG, or directed acyclic graph, in other stream processing frameworks) is a set of operations and transitions that every event moves through from input to output. Figure 14-10 shows the topology in "Word Count".

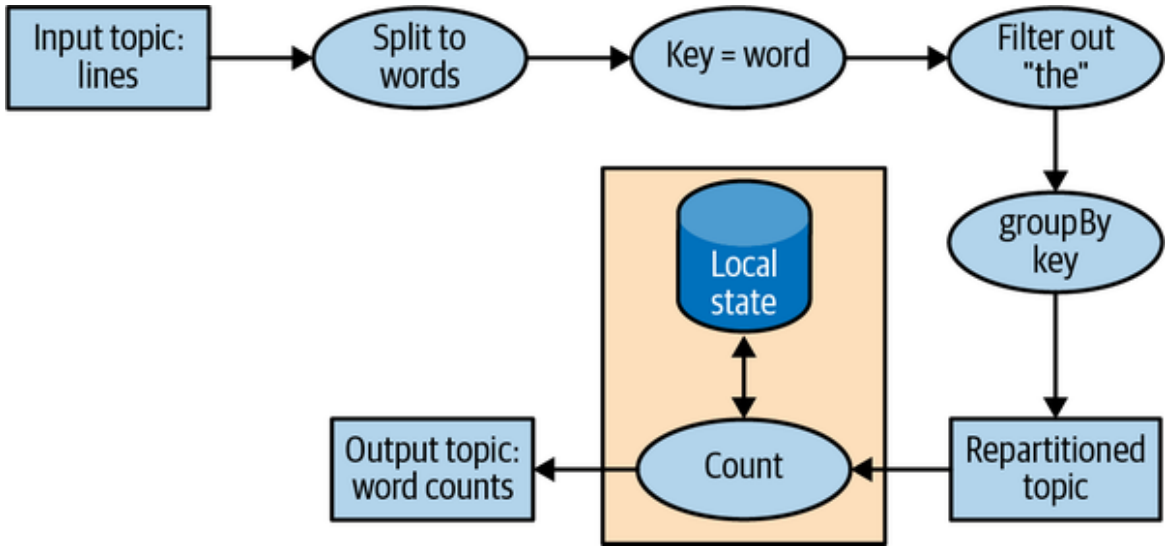

Figure 14-10. Topology for the word-count stream processing example

Even a simple app has a nontrivial topology. The topology is made up of processors—those are the nodes in the topology graph (represented by circles in our diagram). Most processors implement an operation of the data—filter, map, aggregate, etc. There are also source processors, which consume data from a topic and pass it on, and sink processors, which take data from earlier processors and produce it to a topic. A topology always starts with one or more source processors and finishes with one or more sink processors.

Optimizing a Topology

By default, Kafka Streams executes applications that were built with the DSL API by mapping each DSL method independently to a lower-level equivalent. By evaluating each DSL method independently, opportunities to optimize the overall resulting topology were missed.

However, note that the execution of a Kafka Streams application is a three-step process:

1. The logical topology is defined by creating `KStream` and `KTable` objects and performing DSL operations, such as filter and join, on them.
2. `StreamsBuilder.build()` generates a physical topology from the logical topology.
3. `KafkaStreams.start()` executes the topology—this is where data is consumed, processed, and produced.

The second step, where the physical topology is generated from the logical definitions, is where overall optimizations to the plan can be applied.

Currently, Apache Kafka only contains a few optimizations, mostly around reusing topics where possible. These can be enabled by setting `StreamsConfig.TOPOLOGY_OPTIMIZATION` to `StreamsConfig.OPTIMIZE` and calling `build(props)`. If you only call `build()` without passing the config, optimization is still disabled. It is recommended to test applications with and without optimizations and to compare execution times and volumes of data written to Kafka, and of course, validate that the results are identical in various known scenarios.

Testing a Topology

Generally speaking, we want to test software before using it in scenarios where its successful execution is important. Automated testing is considered the gold standard. Repeatable tests that are evaluated every time a change is made to a software application or library enable fast iterations and easier troubleshooting.

We want to apply the same kind of methodology to our Kafka Streams applications. In addition to automated end-to-end tests that run the stream processing application

against a staging environment with generated data, we'll want to also include faster, lighter-weight, and easier-to-debug unit and integration tests.

The main testing tool for Kafka Streams applications is `TopologyTestDriver`. Since its introduction in version 1.1.0, its API has undergone significant improvements, and versions since 2.4 are convenient and easy to use. These tests look like normal unit tests. We define input data, produce it to mock input topics, run the topology with the test driver, read the results from mock output topics, and validate the result by comparing it to expected values.

We recommend using the `TopologyTestDriver` for testing stream processing applications, but since it does not simulate Kafka Streams caching behavior (an optimization not discussed in this book, entirely unrelated to the state store itself, which is simulated by this framework), there are entire classes of errors that it will not detect.

Unit tests are typically complemented by integration tests, and for Kafka Streams, there are two popular integration test frameworks: `EmbeddedKafkaCluster` and `Testcontainers`. The former runs Kafka brokers inside the JVM that runs the tests, while the latter runs Docker containers with Kafka brokers (and many other components, as needed for the tests). `Testcontainers` is recommended, since by using Docker it fully isolates Kafka, its dependencies, and its resource usage from the application we are trying to test.

This is just a short overview of Kafka Streams testing methodologies. We recommend reading the "Testing Kafka Streams—A Deep Dive" blog post for deeper explanations and detailed code examples of topologies and tests.

Scaling a Topology

Kafka Streams scales by allowing multiple threads of executions within one instance of the application and by supporting load balancing between distributed instances of the application. We can run the Streams application on one machine with multiple threads or on multiple machines; in either case, all active threads in the application will balance the work involved in data processing.

The Streams engine parallelizes execution of a topology by splitting it into tasks. The number of tasks is determined by the Streams engine and depends on the number of partitions in the topics that the application processes. Each task is responsible for a subset of the partitions: the task will subscribe to those partitions and consume events from them. For every event it consumes, the task will execute all the processing steps that apply to this partition in order before eventually writing the result to the sink. Those tasks are the basic unit of parallelism in Kafka Streams, because each task can execute independently of others. See Figure 14-11.

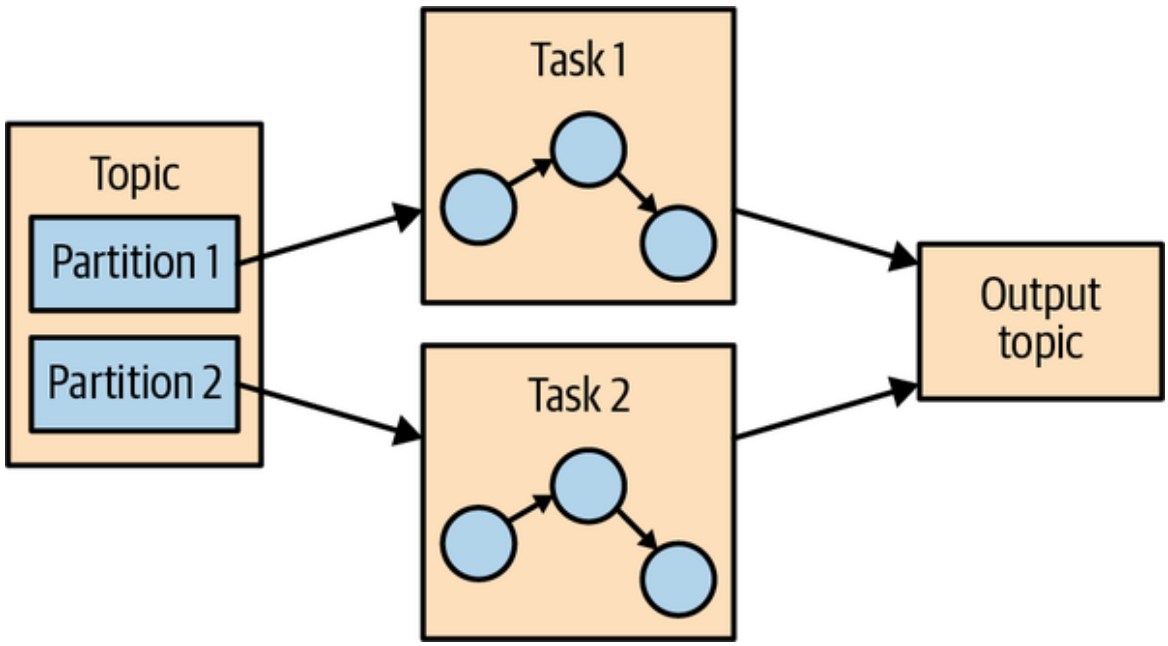

Figure 14-11. Two tasks running the same topology—one for each partition in the input topic

The developer of the application can choose the number of threads each application instance will execute. If multiple threads are available, every thread will execute a subset of the tasks that the application creates. If multiple instances of the application are running on multiple servers, different tasks will execute for each thread on each server. This is the way streaming applications scale: we will have as many tasks as we have partitions in the topics we are processing. If we want to process faster, add more threads. If we run out of resources on the server, start another instance of the application on another server. Kafka will automatically coordinate work—it will assign each task its own subset of partitions, and each task will independently process events from those partitions and maintain its own local state with relevant aggregates if the topology requires this. See Figure 14-12.

Sometimes a processing step may require input from multiple partitions, which could create dependencies between tasks. For example, if we join two streams, as we did in the ClickStream example in "ClickStream Enrichment", we need data from a partition in each stream before we can emit a result. Kafka Streams handles this situation by assigning all the partitions needed for one join to the same task so that the task can consume from all the relevant partitions and perform the join independently. This is why Kafka Streams currently requires that all topics that participate in a join operation have the same number of partitions and be partitioned based on the join key.

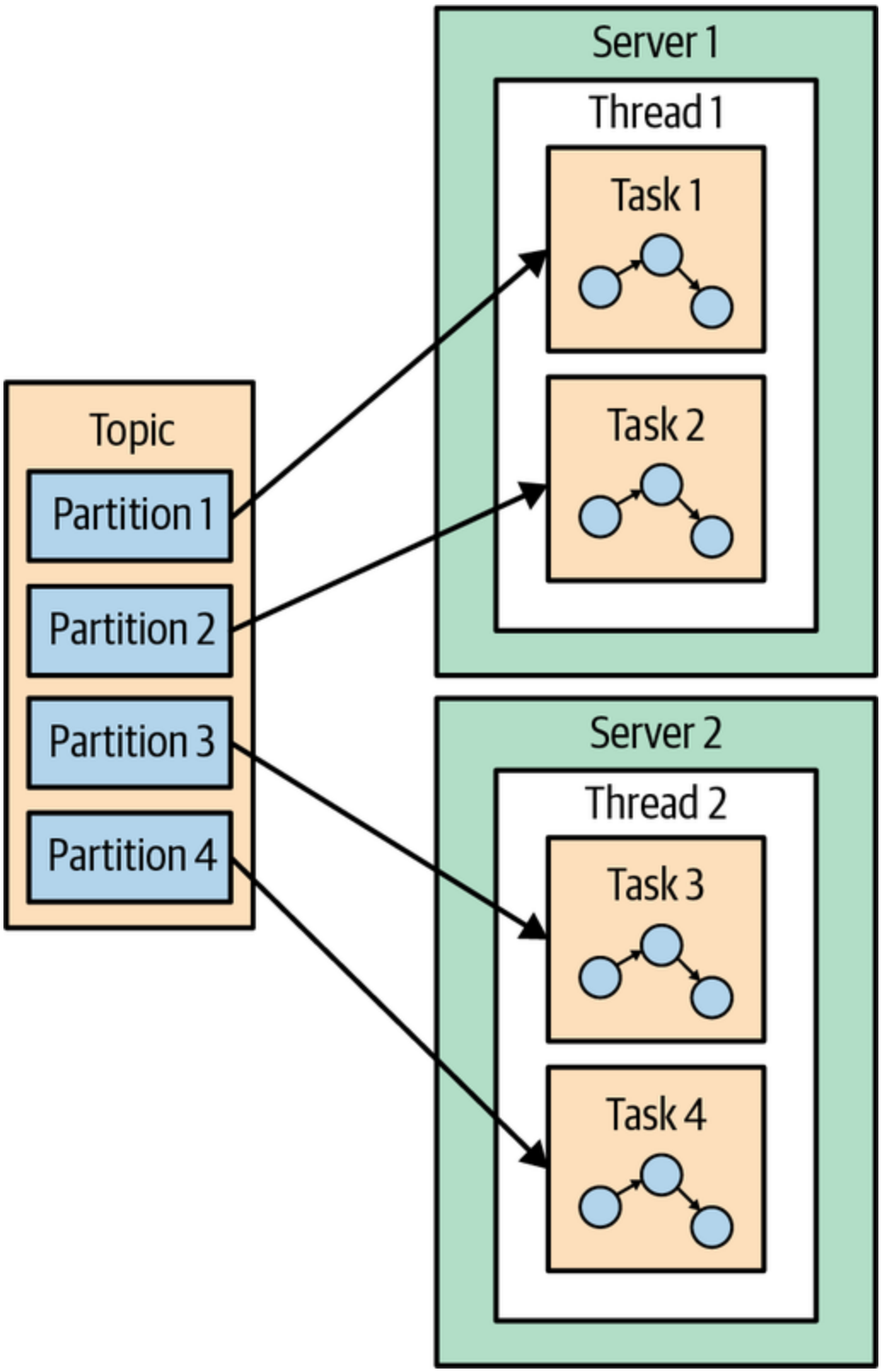

Figure 14-12. The stream processing tasks can run on multiple threads and multiple servers

Another example of dependencies between tasks is when our application requires repartitioning. For instance, in the ClickStream example, all our events are keyed by the user ID. But what if we want to generate statistics per page? Or per zip code? Kafka Streams will repartition the data by zip code and run an aggregation of the data with the new partitions. If task 1 processes the data from partition 1 and reaches a processor that repartitions the data (`groupBy` operation), it will need to *shuffle*, or send events to other tasks. Unlike other stream processor frameworks, Kafka Streams repartitions by writing the events to a new topic with new keys and partitions. Then another set of tasks reads events from the new topic and continues processing. The repartitioning steps break our topology into two subtopologies, each with its own tasks. The second set of tasks depends on the first, because it processes the results of the first subtopology. However, the first and second sets of tasks can still run independently and in parallel because the first set of tasks writes data into a topic at its own rate and the second set consumes from the topic and processes the events on its own. There is no communication and no shared resources between the tasks, and they don't need to run on the same threads or servers. This is one of the more useful things Kafka does—reduce dependencies between different parts of a pipeline. See Figure 14-13.

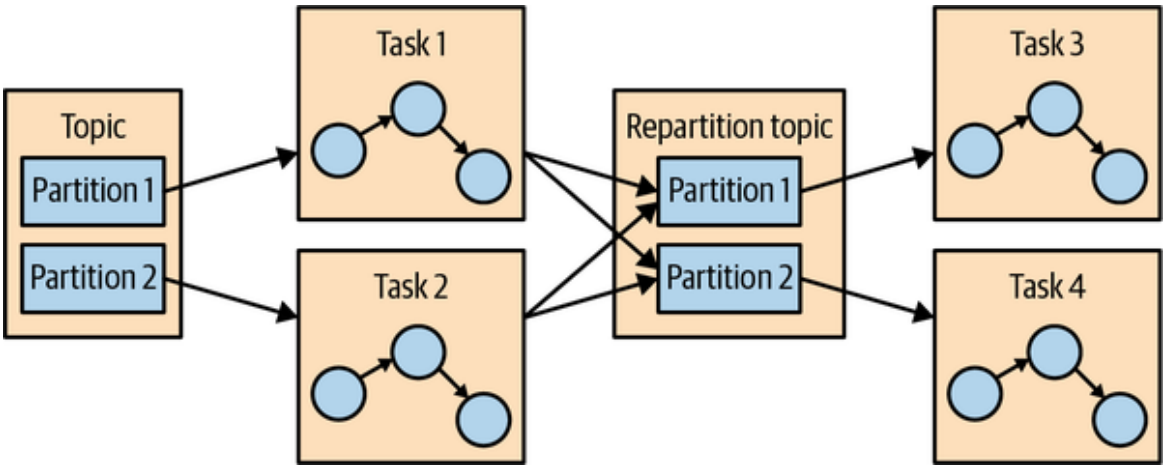

Figure 14-13. Two sets of tasks processing events with a topic for repartitioning events between them

Surviving Failures

The same model that allows us to scale our application also allows us to gracefully handle failures. First, Kafka is highly available, and therefore the data we persist to Kafka is also highly available. So if the application fails and needs to restart, it can look up its last position in the stream from Kafka and continue its processing from the last offset it committed before failing. Note that if the local state store is lost (e.g., because we needed to replace the server it was stored on), the streams application can always re-create it from the change log it stores in Kafka.

Kafka Streams also leverages Kafka's consumer coordination to provide high availability for tasks. If a task failed but there are threads or other instances of the streams application that are active, the task will restart on one of the available threads. This is similar to how consumer groups handle the failure of one of the consumers in the group by assigning partitions to one of the remaining consumers. Kafka Streams benefited from improvements in Kafka's consumer group coordination protocol, such as static group membership and cooperative

rebalancing (described in Chapter 4), as well as improvements to Kafka's exactly-once semantics (described in Chapter 8).

While the high-availability methods described here work well in theory, reality introduces some complexity. One important concern is the speed of recovery. When a thread has to start processing a task that used to run on a failed thread, it first needs to recover its saved state—the current aggregation windows, for instance. Often this is done by rereading internal topics from Kafka in order to warm up Kafka Streams state stores. During the time it takes to recover the state of a failed task, the stream processing job will not make progress on that subset of its data, leading to reduced availability and stale data.

Therefore, reducing recovery time often boils down to reducing the time it takes to recover the state. A key technique is to make sure all Kafka Streams topics are configured for aggressive compaction—by setting a low `min.compaction.lag.ms` and configuring the segment size to 100 MB instead of the default 1 GB (recall that the last segment in each partition, the active segment, is not compacted).

For an even faster recovery, we recommend configuring `standby replica`—those are tasks that simply shadow active tasks in a stream processing application and keep the current state warm on a different server. When failover occurs, they already have the most current state and are ready to continue processing with almost no downtime.

More information on both scalability and high availability in Kafka Streams is available in a a blog post and a Kafka summit talk on the topic.

Stream Processing Use Cases

Throughout this chapter we've learned how to do stream processing—from general concepts and patterns to specific examples in Kafka Streams. At this point it may be worth looking at the common stream processing use cases. As explained in the beginning of the chapter, stream processing—or continuous processing—is useful in cases where we want our events to be processed in quick order rather than wait for hours until the next batch, but also where we are not expecting a response to arrive in milliseconds. This is all true but also very abstract. Let's look at a few real scenarios that can be solved with stream processing:

Customer service

> Suppose that we just reserved a room at a large hotel chain, and we expect an email confirmation and receipt. A few minutes after reserving, when the confirmation still hasn't arrived, we call customer service to confirm our reservation. Suppose the customer service desk tells us, "I don't see the order in our system, but the batch job that loads the data from the reservation system to the hotels and the customer service desk only runs once a day, so please call back tomorrow. You should see the email within 2–3 business days." This doesn't sound like very good service, yet we've had this conversation more than once with a large hotel chain. What we really want is for every system in the hotel chain to get an update about a new reservation seconds or minutes after the reservation is made, including the customer service center, the hotel, the system that sends email confirmations, the website, etc. We also want the customer service center to be able to immediately pull up all the details about any of our past visits to any of

the hotels in the chain, and the reception desk at the hotel to know that we are a loyal customer so they can give us an upgrade. Building all those systems using stream processing applications allows them to receive and process updates in near real time, which makes for a better customer experience. With such a system, the customer would receive a confirmation email within minutes, their credit card would be charged on time, the receipt would be sent, and the service desk could immediately answer their questions regarding the reservation.

Internet of Things

IoT can mean many things—from a home device for adjusting temperature and ordering refills of laundry detergent, to real-time quality control of pharmaceutical manufacturing. A very common use case when applying stream processing to sensors and devices is to try to predict when preventive maintenance is needed. This is similar to application monitoring but applied to hardware and is common in many industries, including manufacturing, telecommunications (identifying faulty cellphone towers), cable TV (identifying faulty box-top devices before users complain), and many more. Every case has its own pattern, but the goal is similar: process events arriving from devices at a large scale and identify patterns that signal that a device requires maintenance. These patterns can be dropped packets for a switch, more force required to tighten screws in manufacturing, or users restarting the box more frequently for cable TV.

Fraud detection

Also known as *anomaly detection*, this is a very wide field that focuses on catching “cheaters” or bad actors in

the system. Examples of fraud-detection applications include detecting credit card fraud, stock trading fraud, video-game cheaters, and cybersecurity risks. In all these fields, there are large benefits to catching fraud as early as possible, so a near real-time system that is capable of responding to events quickly—perhaps stopping a bad transaction before it is even approved—is much preferred to a batch job that detects fraud three days after the fact, when cleanup is much more complicated. This is, again, a problem of identifying patterns in a large-scale stream of events.

In cybersecurity, there is a method known as *beaconing*. When the hacker plants malware inside the organization, it will occasionally reach outside to receive commands. It can be difficult to detect this activity since it can happen at any time and any frequency. Typically, networks are well defended against external attacks but more vulnerable to someone inside the organization reaching out. By processing the large stream of network connection events and recognizing a pattern of communication as abnormal (for example, detecting that this host typically doesn't access those specific IPs), the security organization can be alerted early, before more harm is done.

How to Choose a Stream Processing Framework

When choosing a stream processing framework, it is important to consider the type of application you are planning on writing. Different types of applications call for different stream processing solutions:

Ingest

Where the goal is to get data from one system to another, with some modification to the data to conform to the target system.

Low milliseconds actions

Any application that requires almost immediate response. Some fraud-detection use cases fall within this bucket.

Asynchronous microservices

These microservices perform a simple action on behalf of a larger business process, such as updating the inventory of a store. These applications may need to maintain local state caching events as a way to improve performance.

Near real-time data analytics

These streaming applications perform complex aggregations and joins in order to slice and dice the data and generate interesting, business-relevant insights.

The stream processing system you will choose will depend a lot on the problem you are solving:

- If you are trying to solve an ingest problem, you should reconsider whether you want a stream processing system or a simpler ingest-focused system like Kafka Connect. If you are sure you want a stream processing system, you need to make sure it has both a good selection of connectors and high-quality connectors for the systems you are targeting.

- If you are trying to solve a problem that requires low milliseconds actions, you should also reconsider your choice of streams. Request-response patterns are often better suited to this task. If you are sure you want a stream processing system, then you need to opt for one that supports an event-by-event low-latency model rather than one that focuses on microbatches.
- If you are building asynchronous microservices, you need a stream processing system that integrates well with your message bus of choice (Kafka, hopefully), has change capture capabilities that easily deliver upstream changes to the microservice local state, and has the good support of a local store that can serve as a cache or materialized view of the microservice data.
- If you are building a complex analytics engine, you also need a stream processing system with great support for a local store—this time, not for maintenance of local caches and materialized views but rather to support advanced aggregations, windows, and joins that are otherwise difficult to implement. The APIs should include support for custom aggregations, window operations, and multiple join types.

In addition to use case–specific considerations, there are a few global considerations you should take into account:

Operability of the system

Is it easy to deploy to production? Is it easy to monitor and troubleshoot? Is it easy to scale up and down when needed? Does it integrate well with your existing

infrastructure? What if there is a mistake and you need to reprocess data?

Usability of APIs and ease of debugging

I've seen orders-of-magnitude differences in the time it takes to write a high-quality application among different versions of the same framework. Development time and time-to-market are important, so you need to choose a system that makes you efficient.

Makes hard things easy

Almost every system will claim they can do advanced windowed aggregations and maintain local stores, but the question is: do they make it easy for you? Do they handle gritty details around scale and recovery, or do they supply leaky abstractions and make you handle most of the mess? The more a system exposes clean APIs and abstractions and handles the gritty details on its own, the more productive developers will be.

Community

Most stream processing applications you consider are going to be open source, and there's no replacement for a vibrant and active community. Good community means you get new and exciting features on a regular basis, the quality is relatively good (no one wants to work on bad software), bugs get fixed quickly, and user questions get answers in a timely manner. It also means that if you get a strange error and Google it, you will find information about it because other people are using this system and seeing the same issues.

Summary

We started the chapter by explaining stream processing. We gave a formal definition and discussed the common attributes of the stream processing paradigm. We also compared it to other programming paradigms.

We then discussed important stream processing concepts. Those concepts were demonstrated with three example applications written with Kafka Streams.

After going over all the details of these example applications, we gave an overview of the Kafka Streams architecture and explained how it works under the covers. We conclude the chapter, and the book, with several examples of stream processing use cases and advice on how to compare different stream processing frameworks.