


Chapter 11. Stream Processing

Kafka was traditionally seen as a powerful message bus, capable of delivering streams of events but without processing or transformation capabilities. Kafka's reliable stream delivery capabilities make it a perfect source of data for stream processing systems. Apache Storm, Apache Spark Streaming, Apache Flink, Apache Samza and many more stream processing systems were built with Kafka often being their only reliable data source.

Industry analysts sometimes claim that all those stream processing systems are just the same as complex event processing (CEP) systems that have been around for 20 years. They are surprised that stream processing systems are so successful where CEP only saw very modest adoption. We think the main issue for CEP was the lack of streams of events to process. With the raise of popularity of Apache Kafka, first as simple message bus and later as a data integration system, many companies had a system containing many streams of interesting data, stored for long amounts of time and perfectly ordered. Just waiting for some stream processing framework to show up and process them. In other words, in the same way that data processing was significantly more difficult before databases were invented - stream processing was held back by a lack of stream processing platform that provides reliable storage and integration for stream processing.

Starting from version 0.10.0, Kafka does more than provide a reliable source of data streams to every popular stream processing framework. Now Kafka includes a powerful stream processing library as part of its collection of client libraries. This allows developers to consume, process and produce events in their own apps, without relying on an external processing framework.

We'll begin the chapter by explaining what we mean by stream processing (since this term is frequently misunderstood), we'll then discuss some of the basic concepts of stream processing and the design patterns that are common to all stream processing systems. We'll then dive into Apache Kafka's stream processing library - its goals and architecture. We'll give a small example of how to use Kafka Streams to calculate a moving average of stock prices. We'll then discuss other examples for good stream processing use-cases and finish off the chapter by providing few criteria you can use when choosing which stream processing framework (if any) to use with Apache Kafka. This chapter is intended as a brief introduction to stream processing and will not provide a complete coverage of every Kafka Streams feature or an attempt to discuss and compare every stream processing framework in existence - those topics deserve entire books on their own, possibly several.

What Is Stream Processing

There is a lot of confusion about what stream processing means. There are many definitions that mix up implementation details, performance requirements, data models and many other aspects of software engineering. I've seen the same thing play out in the world of relational databases - the abstract definitions of the relational model are getting forever entangled in the implementation details and specific limitations of the popular database engines.

The world of stream processing is still evolving, and just because a specific popular implementation does things in specific ways or has specific limitations doesn't mean that those details are an inherent part of processing streams of data.

Lets start at the beginning: What is a data stream (also called "event stream" or "streaming data")? First and foremost, a data stream is an abstraction representing an unbounded dataset. Unbounded means infinite and ever growing. The dataset is unbounded because over time, new records keep arriving. This definition is used by Google, Amazon and pretty much everyone else.

Note that this simple model (a stream of events) can be used to represent pretty much every business activity we care to analyze. We can look at a stream of credit card transactions, stock trades, package deliveries, network events going through a switch, events reported by sensors in manufacturing equipment, emails sent, moves in a game, etc, etc. The list of examples is endless because pretty much everything can be seen as a sequence of events.

There are few other attributes of event streams model, in addition to their unbounded nature: * **Event streams are ordered** - There is an inherent notion of which events occur before or after other events. This is clearest when looking at financial events - a sequence in which I first put money in my account and later I spend the money is very different from a sequence at which I first spend the money and later I cover my debt by depositing money back. The latter will incur overdraft charges while the former will not. Note that this is one of the differences between an event stream and a database table - records in a table are always considered unordered and the "order by" clause of SQL is not part of the relational model, it was added to assist in reporting.

- **Immutable data records** - Events, once occured, can never be modified. A financial transaction that is cancelled does not disappear, instead an additional event is written to the stream, recording a cancellation of previous transaction. A customer that returns merchandize to a shop does not mean the previous sell is deleted, rather the return is recorded as an additional event. This is another difference between a data stream and a database table - I can delete or update records in a table, but those are all additional transactions that occure in the database, and as such can be recorded in a stream of events that records all transactions. If you are familiar with the concepts of binlogs, WALs or redo logs in databases you can see that if I insert a record into a table and later delete it, the table will no longer contain the record - but the redo log will contain two transactions - the insert and the delete.

- **Event streams are replayable** - This is a desirable property. While it is easy to imagine non-replayable streams (tcp packets streaming through a socket are generally non-replayable), for more business purposes it is critical to be able to replay a raw stream of events that occured months (and sometimes years) earlier. This is required in order to correct errors, try new methods of analysis or perform audits. This is the reason we believe Kafka made stream processing so successful in modern businesses - it allows to capture and replay a stream of events. Without this capability, stream processing can never be more than a lab toy for data scientists.

It is worth noting that the definition says nothing about the data contained in the events and the number of events per second. The data differs from system to system - events can be tiny (sometimes only few bytes) or very large (XML messages with many headers), they can be completely unstructured, key-value pairs, semi-structured JSON or structured Avro or Protobuf messages. While it is often assumed that data streams are "big data" and involve millions of events per second, the same techniques we'll discuss apply equally well (and often better) to smaller streams of events with only few events per second or minute.

Now that we know what are event streams, its time to make sure we understand what is "stream processing". Stream processing refers to on-going processing of one or more event streams. Stream processing a programming paradigm - just like request-response and batch processing. Lets look at how the three paradigms compare to get a better understanding of how stream processing fits into software architectures:

- Request-response: This is the lowest latency paradigm, with response times ranging from sub-milliseconds to few milliseconds, and usually the expectation that response times will be highly consistent. The mode of processing is usually blocking - an app sends a request and waits for the processing system to respond. In the database world, this paradigm is known as OLTP. Point-of-sales systems, credit card processing and time-tracking systems typically work in this paradigm.

- Batch processing: This is the high-latency / high-throughput option. The processing system wakes up at set times - every day at 2am, every hour on the hour, etc. It reads all required input (either all data available since last execution, all data from beginning of month, etc), writes all required output and goes away until the next time it is scheduled to run. Processing times range from minutes to hours and users expect to read stale data when they are looking at results. In the database world those are the DWH/BI systems - data is loaded in huge batches once a day, reports are generated and users look at the same reports until the next data load occurs. This paradigm often has great efficiency and economy of scale, but in recent years businesses need the data available in shorter time-frames in order to make decision making more timely and efficient. This puts huge pressure on those systems that were written to exploit economy of scale and not provide low-latency reporting.

- Stream processing: This paradigm is intended to fill the gap between the other two. We'll notice that most business processes don't require immediate sub-millisecond response but also can't wait for the next day to know results either. Most business processes happen continuously, and as long as the business reports are updated continuously and the line of business apps can continuously respond, the processing can proceed without anyone waiting for a specific response and needs it within milliseconds. Business processes like alerting on suspicious credit transactions or network activity, adjusting prices in real-time based on supply and demand or tracking deliveries of packages are all natural fit for continuous but non-blocking processing.

  It is important to note that the definition doesn't mandate any specific framework, API or feature. As long as you are continuously reading data from an unbounded dataset, doing something to it and emitting output - you are doing stream processing. But the processing has to be continuous and on-going. A process that starts every day at 2am, reads 500 records from the stream, outputs a result and goes away doesn't quite cut it as far as stream processing goes.

Stream Processing Concepts

Much of stream processing is very similar to any type of data processing - you write code that receives data, does something with the data - few transformations, aggregates, enrichments, etc and then place the result somewhere. There are some key concepts that are unique to stream processing and often cause confusion when someone who has data processing experience first attempts to write stream processing applications. Lets take a look at few of those and try to clarify.

Time

Time is probably the most important concept in stream processing and often the most confusing. For an idea of how complex time can get when discussing distributed systems, I recommend Justin Sheehy's excellent "There is No Now" (http://queue.acm.org/detail.cfm?id=2745385) paper. In the context of stream processing having a common notion of time is critical because most stream applications perform operations on time windows - for example our stream application may calculate a moving 5-minute average of stock prices. In that case, we need to know what to do when one of our producers goes offline for 2 hours due to network issues and returns with 2 hours worth of data - most of the data will be relevant for 5-min time windows that have long passed and for which the result was already calculated and stored.

Stream processing systems typically refer to the following notions of time:

- **Event Time** - This is the time the events we are tracking occurred and the record was created. The time a measurement was taken, an item at was sold at a shop, a user viewed a page on our website, etc. In versions 0.10.0 and higher, Kafka automatically adds current time to Producer records at the time they are created. If this does not match your application's notion of "event time", for example in cases where the Kafka record is created based on a database record some time after the event occurred, you should add the event time as a field in the record itself. Event time is usually the time that matters most when processing stream data.

- **Log Append Time** - This is the time the event arrived to the Kafka broker and stored there. In versions 0.10.0 and higher, Kafka brokers will automatically add this time to records they receive if Kafka is configured to do so or if the records arrive from older producers and contain no timestamps. This notion of time is typically less relevant for stream processing, since we are usually interested in the times the events occurred. For example, if we calculate number of devices produced per day, we want to count devices that were actually produced on that day - even if there were network issues and the event only arrived to Kafka the following day. However, in cases where the real event time was not recorded, Log Append time can still be used consistently since it does not change after the record was created.

- **Processing Time** - This is the time at which a stream processing application received the event in order to perform some calculation. This time can be milliseconds, hours or days after the event occurred. This notion of time assigns different timestamps to the same event - depending on exactly when each stream processing application happened to read the event. It can even differ for two threads in the same application! Therefore this notion of time is highly unreliable and best avoided.

When working with time, it is important to be mindful of timezones. The entire data pipeline should standardize on a single timezone - or results of stream operations will be confusing and often meaningless. If you must handle data streams with different timezones, you need to make sure you can convert events to a single timezone before performing operations on time windows. Often it means storing the timezone in the record itself.

State

As long as you only need to process each event individually, stream processing is a very simple problem. For example, if all you need to do is to read a stream of online shopping transactions from Kafka, find the transactions over $10,000 and email the relevant sales person - you can probably write this in just few lines of code using a Kafka consumer and SMTP library.

Stream processing becomes really interesting when you have operations that involve multiple events: Counting number of events by type, moving averages, joining two streams to create an enriched stream of information. In those cases, it is not enough to look at each event by itself, you need to keep track of more information - how many events of each type did we see this hour, any events this should be joined with, total sum of value of all events grouped by type, etc. We call the information that is stored between events a "state".

It is often tempting to store the state in some variables local to the stream processing app. For example: a simple hash-table to store moving counts. In fact, we did just that in many examples in the book. However, this is not a reliable approach for managing state in stream processing - when the stream processing application is stopped, the state is lost, which changes the results. This is usually not the desirable outcome - and therefore care should be taken to persist the most recent state and recover it when starting the application.

Stream processing refer to several types of state:

- **Local** or **internal** state - state that is accessible only by a specific instance of the stream processing application. This state is usually maintained and managed with an embedded, in-memory database running within the application. The advantage of local state is that it is extremely fast. The disadvantage is that you are limited to the amount of memory available. As a result, many of the design patterns in stream processing focus on ways to partition the data into sub-streams that can be processed using a limited amount of local state.
- **External** state - state that is maintained in an external datastore, often a NoSQL system like Cassandra. The advantages of an external state is its virtually unlimited size and the fact that it can be accessed from multiple instances of the application or even from different applications. The downside is the extra latency and complexity introduced with an additional system. Most stream processing apps try to avoid having to deal with an external store, or at least limit the latency overhead by caching information in the local state and communicating with the external store as rarely as possible. This usually introduces challenges with maintaining consistency between the internal and external state.

Stream-Table duality

We are all familiar with database tables - A table is a collection of records, each identified by its primary key and contains a set of attributes as defined by a schema. Table records are mutable (i.e. tables allow update and delete operations) and querying a table allows checking the state of the data at a specific point in time. For example, by querying the CUSTOMERS_CONTACTS table in a database, we expect to find current contact details for all our customers. Unless the table was specifically designed to include history, we will not find their past contacts in the table.

One way to compare tables to streams is to note that streams contain changes - streams are a stream of events and each event caused a change. A table contains a current state of the world, a state that is a result of many changes. From this description, it is clear that tables and streams are two sides of the same coin - the world always changes and sometimes we are interested in the events that caused those changes while other times we are interested in the current state of the world. Systems that allow you to transition back and forth between the two ways of looking at data are more powerful than systems that support just one.

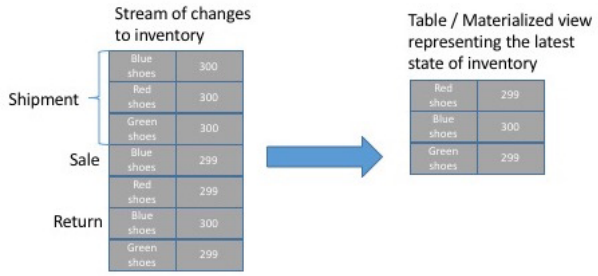
In order to convert a table to a stream, we need to capture the changes that modify the table. Take all those "insert", "update" and "delete" events and store them in a stream. Most databases offer "Change Data Capture" (CDC) solutions for capturing these

changes and there are many Kafka Connectors that can pipe those changes into Kafka where they will be available for stream processing.

In order to convert a stream to a table, we need to "apply" all the changes that the stream contains. This is also called "materializing" the stream. We create a table, either in memory, in an internal state store or in an external database - and start going over all the events in the stream from beginning to end, changing the state as we go. When we finish - we have a table representing a state at a specific time that we can use.

Suppose we have a store selling shoes. A stream representation of our retail activity can be a stream of events: "Shipment arrived with red, blue and green shoes" "Blue shoes sold" "Red shoes sold" "Blue shoes returned" "Green shoes sold"

If we want to know what our inventory contains right now or how much money we made until now, we want to materialize the view - we currently have blue and yellow shoes and $170 in the bank. If we want to know how busy the store is, we can look at the entire stream and see that there were 5 transactions. We may also want to investigate why the blue shoes were returned.

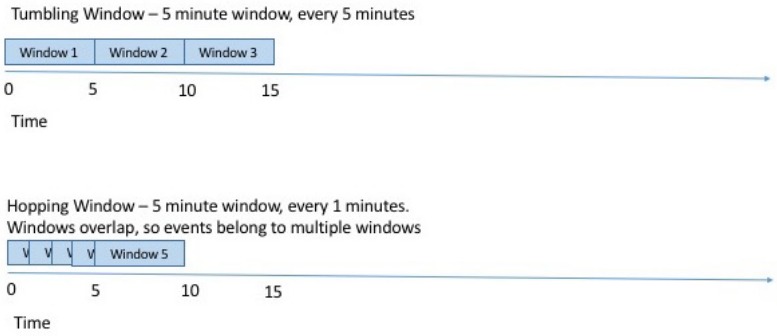


Time Windows

Most operations on streams are windowed operations - operating on slices of time: Moving averages, top products sold this week, 99 percentile load on the system, etc. Join operations on two streams are also windowed - we join events that occurred at the same slice of time. Very few people stop and think about the type of window they want for their operations. For example, when calculating moving averages, we want to know:

- Size of the window - average on 5 minutes? 15? day? smaller windows means you'll know about changes faster, but the measurement is more noisy. Larger windows are smoother, but they lag more - if price increases it will take longer to notice than with a smaller window.
- How often the window moves ("advance interval") - 5 minute averages can update every minute, second or every time there is a new event. When the "advance interval" is equal to the window size, this is sometimes called a "tumbling window". When the window moves on every record, this is sometimes called a "sliding window".
- How long the window remains updatable - Our 5-minute moving average calculated the average for 00:00-00:05 window. Now an hour later, we are getting few more results with their "event time" showing 00:02. Do we update the result for the 00:00-00:05 period? Or do we let bygones be bygones? Ideally, we'll be able to define a certain time-period at which events will get added to their respective time-slice - If the events were up to 4 hours late, lets update, but ignore events that are even more late.

  Windows can be aligned to clock time - i.e a 5 minute window that moves every minute will have the first slice as 00:00-00:05 and the second is 00:01-00:06. Or it can be un-aligned and simply start whenever the app started and then the first slice can be 03:17-03:22. Sliding windows are never aligned because they move whenever there is a new record.



Stream Processing Design Patterns

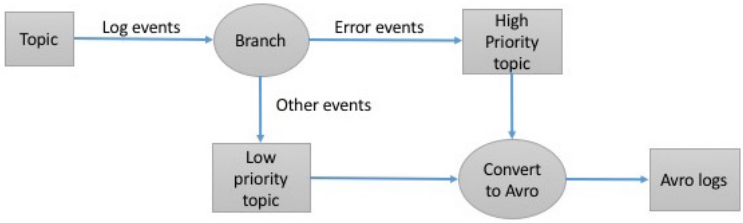
Every stream processing system is different - from the basic combination of a consumer, processing logic and producer to involved clusters like Spark Streaming with their machine learning libraries. And much in between. But there are some basic design patterns, known solutions to common requirements of stream processing architectures. We'll review few of those well known patterns here, and then show how they are used in 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 map/filter pattern since 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 where 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 would be 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 can be an application that reads events from stream and modifies them from JSON to Avro. Such applications do need to maintain state within the application since each event can be handled independently. This means that recovering from app failures or load-balancing is incredibly easy - there is no need to recover state, you can simply hand the events off to another instance of the app to process.

This pattern can be easily handled with a simple producer and consumer:

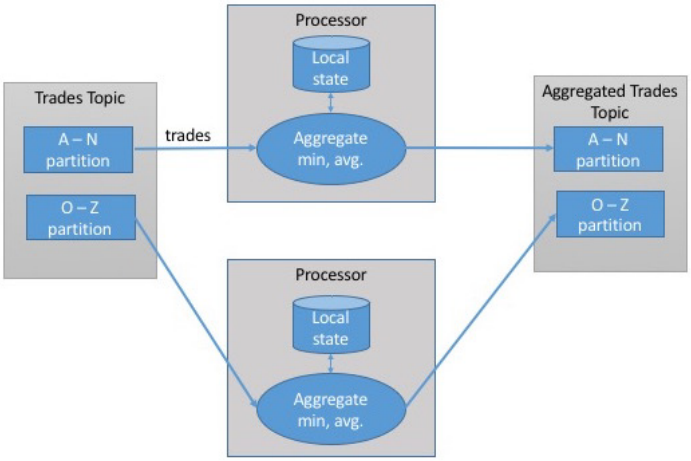


Processing with Local State

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

These aggregations require maintaining a **state** for the stream. In our example, in order to calculate minimum and average price each day we need to store the minimum and maximum values we've seen until the current time and compare each new value in the stream to the stored minimum and maximum.

All these can be done using **local** state (rather than a shared state) - because each operation in our example is a "group by" aggregate. I.e. 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.



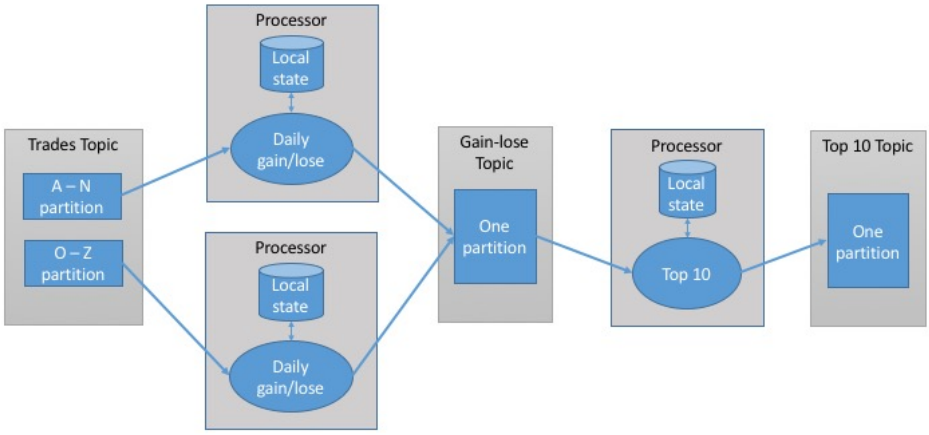
Stream processing applications become significantly more complicated when the application has local state and there are several issues a stream processing application must address:

- Memory usage - the local state must fit into the memory available to the application instance
- Persistence - we need to make sure the state is not lost when an application instance is shutting down, and that the state can be recovered when the instance is started again or when it is replaced by a different instance. This is something that KafkaStreams 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 Streams node goes down, the local state is not lost - it can be easily re-created by re-reading 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 later we can re-populate the local cache from this data. Kafka uses log compaction for these topics to make sure the topic doesn't grow endlessly and re-creating the state is always feasible.
- Rebalancing - partitions sometimes get re-assigned to a different consumer. When this happens, the instance that lose the partition must store the last good state and the instance that receives the partition must know to recover the correct state.

Different stream processing frameworks differ in how much they help the developer manage the local state they need. If your application requires maintaining local state, be 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.

Multi-phase processing / Re-partitioning

Local state is great if you need a "group by" type of aggregate. But what if you need a result that uses all available information? For example, suppose we want to publish "top 10" stocks each day - the 10 stocks that gained the most from opening to closing on each day of trading. Obviously, nothing we do on each application instance locally is enough - because all the top-10 stocks could be in partitions assigned to other instances. What we need is a 2-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.

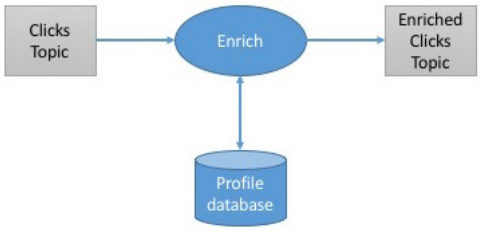


This type of multi-phase processing is very familiar to those who wrote map-reduce code, where you often have to resort to multiple reduce phases. If you ever wrote map-reduce code, you'll remember that you need a separate app for each reduce step. Unlike MapReduce, most stream processing frameworks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

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.

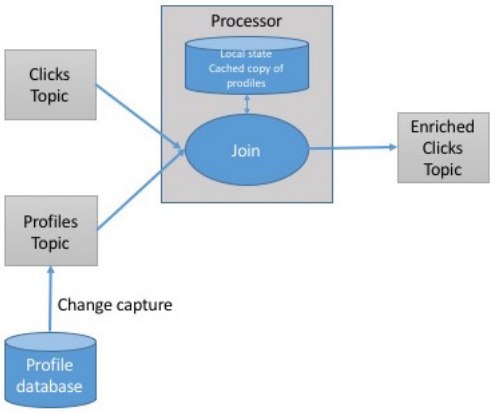
The obvious idea on how to perform an external lookup for data enrichment is something like this: For every click event in the stream, look up the user in the profile database and write an event that includes the original click plus the user age and gender to another topic.



The problem with this obvious idea is that an external lookup adds significant latency to the processing of every record - usually between 5-15 milliseconds. In many cases, this is not feasible. Often the additional load this places on the external datastore 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. We want a solution that scales better.

In order to get good performance and scale, we will want 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 CDC - Change Data Capture, and if you use Kafka Connect you will find multiple connectors capable of performing CDC and converting database tables to a stream of change events. This allows you to keep your own private copy of the table, and you will be notified whenever there is a database change event, so you can update your own copy accordingly.



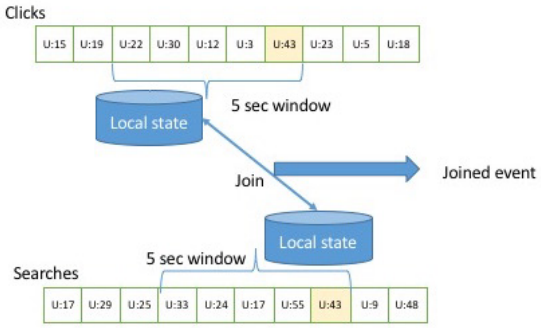
And then when you get click events, you can look up the user_id at your local cache and enrich the event. And because you are using a local cache, this scales a lot better and will not affect the database and other apps using it.

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

Streaming Join

Sometimes you want to join two real event streams rather than a stream with a table. What makes a stream "real"? If you'll recall the discussion at the beginning of the chapter, streams are unbounded. When you use a stream to represent a table, you can ignore most of the history in the stream - you only care about the current state in the table. But when you join two streams, you 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, lets 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 we'll 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 second long window on each stream and match the results from each window.



The way this works in Kafka Streams is that both 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 cache, and this is how it can perform the join

Out of Sequence Events

Handling events that arrive at the stream at the wrong timing is a challenge not just in stream processing but also in traditional ETL systems. Out of sequence events happen quite frequently and expectedly at IoT (Internet of Things) scenarios: A mobile device loses wifi signal for few hours and sends 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).

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

- 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.
- Defines a time period during which it will attempt to reconcile out-of-sequence events. Perhaps a 3 hour delay should be reconciled but events over 3 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 I have a daily batch job and few events arrived after the job completed, I can usually just re-run 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 events and new at any given moment.
- Have the ability 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 streams 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 time older or newer than current processing time. This is typically done by maintaining multiple aggregation windows available for update in the local state and giving the 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.

Kafka's Stream APIs 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 needs to be updated as a result of a late event, Kafka Streams will simply write a new result for this aggregation window, which will overwrite the previous result.

Re-Processing

The last important pattern is that of re-processing 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 re-process the event stream and re-calculate 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 stream processing apps writing two result streams is simply a matter of spinning up the new version of the application as a new consumer group, starting with the first offset of the input topics (so it will get its own copy of all events in the input streams), following the results stream, and switching the client applications to the new result stream when the new version of the processing job has caught up. This is possible with every stream processing framework that uses Apache Kafka.

The second use-case is more challenging - it requires "resetting" an existing app to start processing back at the beginning of the input streams, and also 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, it allows comparing results between versions and it doesn't have the risk of losing critical data or introducing errors in the cleanup process.

Kafka Streams By Example

In order to demonstrate how those patterns are implemented in practice, we'll show few examples using Apache Kafka's streams API. We are using this specific API because it is relatively simple to use and it ships with Apache Kafka, which you 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 while the examples are quite specific.

Apache Kafka has two streams APIs - a low level Processor API and a high level Streams DSL. We will use Kafka Streams DSL in our examples. The DSL allows you 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 you to create your own transformations, but as you'll see - this is rarely required.

An application that uses the DSL API always starts with using the StreamBuilder to create a processing **Topology** - a directed graph (DAG) of transformations that are applied to the events in the streams. Then you 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 you close KafkaStreams object.

We'll look at 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

Lets 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 configuring the KafkaStreams engine. KafkaStreams has a large number of possible configurations, which we won't discuss here, but you can find in the documentation (http://kafka.apache.org/documentation/#streamsconfigs) and in addition, you can also configure the producer and consumer embedded in the streams engine 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.KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName()); ❸
        props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
```

❶ Every Kafka Streams application must have an application id. This 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 Kafka Streams application always read data from Kafka topic and write their output to Kafka topics. As we'll discuss later, they also use Kafka for coordination. So we 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, lets build our streams topology:

```
        KStreamBuilder builder = new KStreamBuilder();

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


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

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

❶ We create a KStreamBuilder 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 So we count how many events we got in each collection. The result of count is a Long, so we convert it to a String so it will be easier for us to read the results from Kafka.

❻ Only one thing left - write the results back to Kafka

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

```
        KafkaStreams streams = new KafkaStreams(builder, 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 since the input data is finite.
        Thread.sleep(5000L);

        streams.close(); ❸

    }
}
```

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

❷ Start the KafkaStreams engine

❸ and after a while, stop it.

Thats it! In just few short lines we demonstrated how easy it is to implement **single event processing** pattern (We applied a map and a filter on the events). We then **re-partitioned** 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.

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

One thing you'll notice is that you can run the entire example on your own machine without installing anything except Apache Kafka. This is similar to the experience you may have seen when using Spark in something like "Local Mode". The main different is that if your input topic contains multiple partitions, you can run multiple instances of the WordCount application (just run the app in several different terminal tabs) and this gives you your first Kafka Streams processing cluster. The instances of the WordCount application itself talk to each other and coordinate the work. One of the biggest barriers to entry with Spark is that local mode is very easy to use, but then to run a production cluster, you need to install YARN or Mesos and then install Spark on all those machines and then learn how to submit your app to the cluster. With Kafka's Streams API you just start multiple instances of your app - and you have a cluster. The exact same thing is running on your 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 while 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 an output streams that contains few windowed statistics: * Best (i.e minimum) ask price for every 5 sec window * Number of trades for every 5 sec window * Avg ask price for every 5 sec window

All those will be updated every second.

For the simplicity of the example, we'll assume our exchange only has 10 different stock tickers trading in it.

The setup and configuration is very similar to those we used in the word-count example above.

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

The main different is the Serde clases used. In the word-count example, we used Strings for both key and value and therefore we 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 serializer and deserializer this object (and few other objects we used in this small app), I used Gson library from Google, which allowed me to generate JSon serializer and deserializer from my Java object and then I created a small wrapper that created a Serde object from those. Here is how I 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 you need to remember to provide a Serde object for every object you'll want to store in Kafka - input, output and in some cases also intermediate results. To make this easier, I recommend generating these Serdes - through projects like GSon, Avro, Protobufs or similar.

Now that we got everything configured, its time to build our topology:

```
KStream<TickerWindow, TradeStats> stats = source.groupByKey() ❶
        .aggregate(TradeStats::new,  ❷
            (k, v, tradestats) -> tradestats.add(v),  ❸
            TimeWindows.of(5000).advanceBy(1000), ❹
            new TradeStatsSerde(), ❺
            "trade-stats-store") ❻
        .toStream((key, value) -> new TickerWindow(key.key(), key.window().start())) ❼
        .mapValues((trade) -> trade.computeAvgPrice()); ❽

        stats.to(new TickerWindowSerde(), new TradeStatsSerde(),  "stockstats-output"); 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 we 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 After we ensured correct partitioning, we start the windowed aggregation. The "aggregate" method will split the stream into overlapping windows (a 5 second window every second), and then it applies an aggregate method on all the events in the window. The first parameter this method takes is a new object that will contain 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, avg price, number of trades.

❸ 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.

❹ We define the window - in this case a window of 5 seconds (5000 milliseconds), advancing every second.

❺ Then we provide a Serde object for serializing and deserializing the results of the aggregation (the Tradestats object)

❻ As mentioned in the patterns section, 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 name of the state store. This can be any unique name.

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. Since it represents tallied results and a specific state calculated from a stream of changes (See the "Concepts" section for more explanation of stream-table duality). I wanted to turn the table back into a stream of events and also, instead of having the entire window definition in my key, I chose to use my own object that contains just the ticker and the start-time of the window. This toStream method converts the table into a stream and also converts the key into my TickerWindow object.

❽ The last step is to update the average price - right now the aggregation results include 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.

❾ And finally - we write the results back to "stockstats-output" stream

After we defined the flow, we use it to generate a KafkaStreams object and we run it, just like we did in the word-count example.

This examples 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 give a name to the state store. Yet this application will scale to multiple instances and will 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 the Architecture section.

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 bit of a 360-view into each user activity - for each minute, what did he search for? what did he click? what are the interested in the profile? These kinds of joins provide a rich data collection for analytics. Product recommendations are often based on this kind of information - user searched for bikes, clicked on links for “Trek” and is interested in travel - 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, lets skip this part and take a look at the topology for joining multiple streams:

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

KStream<Integer, UserActivity> viewsWithProfile = views.leftJoin(profiles, ❸
    (page, profile) -> new UserActivity(profile.getUserID(), profile.getUserName(), profile.getZipcode(), profile.getI

KStream<Integer, UserActivity> userActivityKStream = viewsWithProfile.leftJoin(searches, ❺
    (userActivity, search) -> userActivity.updateSearch(search.getSearchTerms()), ❻
    JoinWindows.of(1000), Serdes.Integer(), new UserActivitySerde(), new SearchSerde()); ❼
```

❶ First thing we do is create streams objects for the two streams we want to join - clicks and searches

❷ We also define a KTable for the user profiles. A KTable is a local cache 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, you 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 they 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 stream to 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 - stream to stream join is a join with a time-window. Joining all clicks and all 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 1 second. Clicks that happen within 1 second of the search are considered relevant, and the search terms will be included in the activity record that contains the click and the user profile. This will allow a full analysis of searches and their results.

After we defined the flow, we use it to generate a KafkaStreams object and we run it, just like we did in the word-count example.

This examples show two different join patterns possible in stream processing - one is joining a stream with a table to enrich all streaming events with information in the table. This is a bit 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.

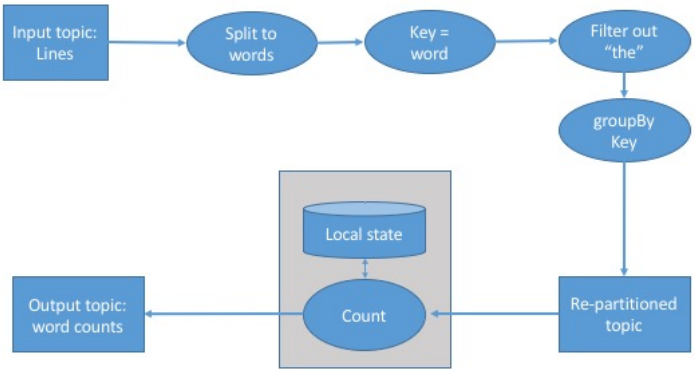
As usual, you can find the complete example including instructions for running it on Github.

Kafka Streams - Architecture Overview

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

Building a Topology

Every streams applications implements and executes at least 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. In the Word Count example, the topology is:

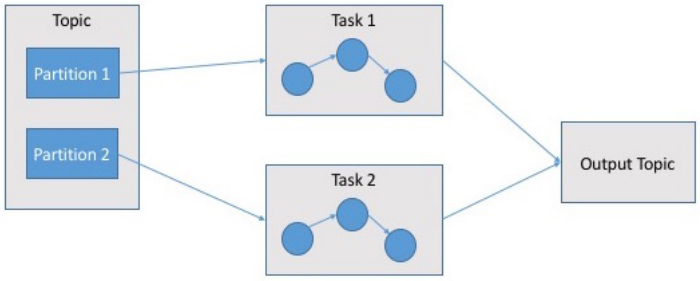


Even a simple app has a non-trivial topology. The topology is made up of Processors - those are the nodes in the topology graph (represented by circles in our diagram). Most processors implement an operation of the data - filter, map, aggregate, etc. There are also Source Processors which consume data from a topic and pass it on and Sink Processors which takes 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.

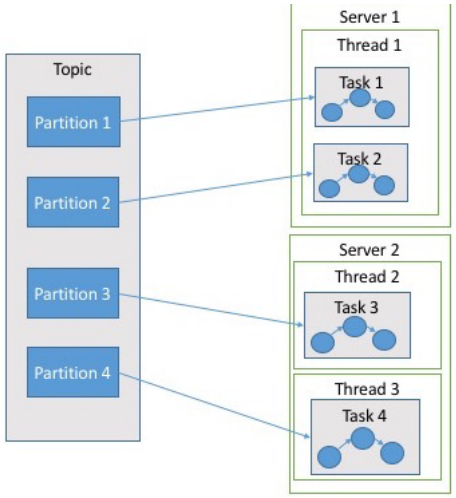
Scaling the Topology

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. You 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 with 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 the Kafka streams engine, because each task can execute independently of others.



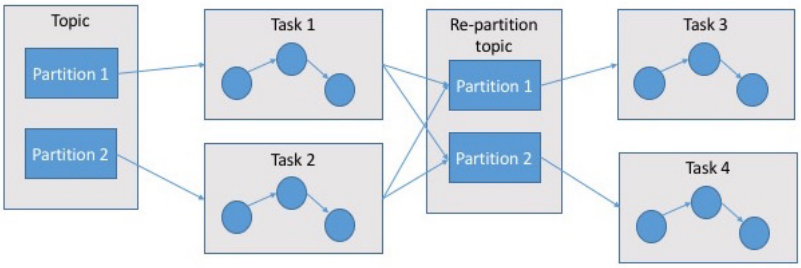
The developer of the application can choose the number of threads each application instance will execute. If multiple threads are available, every thread will execute a subset of the tasks that the application creates. If multiple instances of the application are running on multiple servers, different tasks will execute for each thread on each server. This is the way streaming applications scale: You will have as many tasks as you have partitions in the topics you are processing. If you want to process faster, add more threads. If you 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.



You may have noticed that sometimes a processing step may require results from multiple partitions, which could create dependencies between tasks. For example, if we join two streams, as we did in the Click Stream example, you need data from a partition in each stream before you can emit a result. Kafka streams handles this situation by assigning all the partitions needed for one join to the same task, so this 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 will have the same number of partitions and be partitioned based on the join key.

Another way we can introduce dependencies between tasks is when our application requires re-partitioning. For example, in the Click Stream example all our events are keyed by the user ID. But what if we want to generate statistics per page? or per zipcode? we'll need to re-partition the data by the zipcode 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 re-partitions the data (groupBy operation), it will need to **shuffle** the events - send events to other tasks to process them. Unlike other stream processor frameworks, Kafka streams re-partitions by writing the events to a new topic with the new keys and partitions, and have another set of tasks read events from that topic and continue processing. The re-partitioning steps breaks our topology into two sub-topologies, each with its own tasks. The second set of tasks depends on the first, because it processes results of the first sub-topology. However, they can still run independently 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 same servers. This is one of the more useful things Kafka does - reduce dependencies between different parts of a pipeline.



Surviving Failures

The same model that allows us to scale our application also allows us to gracefully handle failures. First, all the data, including a change-log of the local state is persisted to Kafka - which is 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 (for example, 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 by 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.

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 asking what are the common use-cases for stream processing. We explained in the beginning of the chapter that stream processing, or continuous processing is useful in cases where you want your events to get processed in quick order, rather than wait for hours until the next batch, but you are not really standing around waiting for a response to arrive in milliseconds. This is all true, but also very abstract. Lets look at few real examples of problems that can be solved with stream processing:

- **Customer Service:** Suppose that you just reserved a room at a large hotel chain and you expect an email confirmation and receipt. Few minutes after reserving, when the confirmation still didn't arrive, you call customer service to confirm your reservation. Suppose the customer service desk tells you "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 I've had this conversation more than once with a large hotel chain. What we really want is every system in the hotel chain to get an update about new reservations seconds or minutes after the reservation is made. Including the customer service center, the hotel, the system that sends email confirmations, the website, etc. You also want the customer service center to be able to immediately pull up all the details about any of your past visits in any of the hotels in the chain and the reception desk at the hotel to know that you are a loyal customer so they can give you an upgrade. Building all those systems using stream processing applications allows them to receive and process updates in near real-time and this makes for a better customer experience. With such system I'd receive a confirmation email within minutes, my credit card would be charged on time, the receipt would be sent and the service desk can immediately answer my questions regarding the reservation.
- **Internet of Things**: Internet of things can mean many things - from a home device for adjusting temperatures 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 - manufacturing, telecommunications (identify faulty cellphone towers), cable TV (identify 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 large scale and identify patterns that signal that a device requires maintenance. These patterns can be dropped packets for a switch, more force requires to tighten screws in manufacturing or users restarting the box more frequently for Cable TV.

- **Fraud Detection**: Fraud detection, also known as anomaly detection, is also a very wide field that focuses on catching "cheaters" or bad actors in the system. Credit card fraud, stock trading fraud, video game cheaters or cybersecurity risks. In all these fields there are large benefits to catching fraud as early as possible - before they cause large damage. 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 3 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 cyber security there is a fraud method known as beaconing, where the hacker plants malware inside the organization, and this malware occasionally reaches outside to receive commands. Usually 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 the pattern of communication as abnormal, detecting that this host typically doesn't access those specific IPs, the security organization can be alerted early, before farther harm is made.

How to Choose a Stream Processing Framework

When you compare two stream processing systems, the most important thing you can do is consider the use-case or cases you have in mind.

There are few important categories of applications you may be considering:

- Ingest - where the goal is to get data from one system to another, with some modification to the data on the way that will make it conform to the target system.
- Low milliseconds actions - Any application that requires almost immediate response. Some of fraud detection use-cases fall within this bucket.
- Asynchronous microservices - these microservices perform a simple action on behalf of a larger business process. Perhaps update the inventory of a store. These applications may need to maintain a 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 re-consider 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 re-consider 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 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 very well with your message bus of choice (Kafka, hopefully), that has change capture capabilities in order to easily deliver upstream changes to the microservice local caches and good support of a local store that can serve as a cache or materialized view of the microservice data.
- If you are building 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 in order 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 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 re-process data?
- Usability of APIs and ease of debugging: I've seen orders of magnitude differences in the time it takes to write high quality app between different versions of the same framework. Development time and time-to-market is 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 caches, 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 you'll consider are going to be open source. In open source software, there's no replacement for a vibrant and active community. Good community means you get new and exciting features on regular basis, the quality is relatively good (no one wants to work on bad software), bugs gets fixes quickly and user questions get answers in timely manner. It also means that if you get a strange error and google for it, you'll find information - because many other people are using this system and are seeing the same issues.

Placeholder

PREV
10. Monitoring Kafka

NEXT
A. Installing Kafka on Other Operating Systems