O'REILLY®

Compliments of Lightbend

Stream Processing with Apache Flink

FUNDAMENTALS, IMPLEMENTATION, AND OPERATION OF STREAMING APPLICATIONS

Fabian Hueske & Vasiliki Kalavri

Lightbend Fast Data Platform encapsulates the **best of breed** stream processing engines (Spark, Flink, and Akka Streams) with the backplane of Kafka, plus management and monitoring tools for **maximum reliability** with **minimum effort**.

Lightbend
Fast Data Platform

Sign Up for Early Access

Stream Processing with Apache Flink

This Preview Edition of *Stream Processing with Apache Flink*, Chapters 2 and 3, is a work in progress. The final book is currently scheduled for release in November 2017 and will be available at oreilly.com and other retailers once it is published.

Fabian Hueske and Vasiliki Kalavri

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Stream Processing with Apache Flink
by Fabian Hueske and Vasiliki Kalavri

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Tim McGovern **Interior Designer:** David Futato

February 2017: First Edition

Revision History for the First Edition
2017-02-14: First Release

978-1-491-97429-2

[LSI]

Table of Contents

CHAPTER 1

Stream Processing Fundamentals

So far, we have seen how stream processing addresses limitations of traditional batch processing and how it enables new applications and architectures. We have discussed the evolution of the open-source stream processing space and we have got a brief taste of what a Flink streaming application looks like. In this chapter, we enter the streaming world for good and we provide the necessary background for the rest of this book.

This chapter is still rather independent of Flink. Its goal is to introduce the fundamental concepts of stream processing and discuss the requirements of stream processing frameworks. We hope that after reading this chapter, you will have gained the necessary knowledge to understand the requirements of stream processing applications and you will be able to evaluate features and solutions of modern stream processing systems.

Introduction to dataflow programming

Before we delve into the fundamentals of stream processing, we must first introduce the necessary background on *dataflow* programming and establish the terminology that we will use throughout this book.

Dataflow graphs

As the name suggests, a dataflow program describes how data flows between operations. Dataflow programs are commonly represented as directed graphs, where nodes are called operators and represent computations and edges represent data dependencies. Operators are the basic functional units of a dataflow application. They consume data from inputs, perform a computation on them, and produce data to outputs for further processing. Operators without input ports are called data sources and operators without output ports are called data sinks. A dataflow graph must have at least one data source and one data sink. Figure 2.1 shows a dataflow program that extracts and counts hashtags from an input stream of tweets.

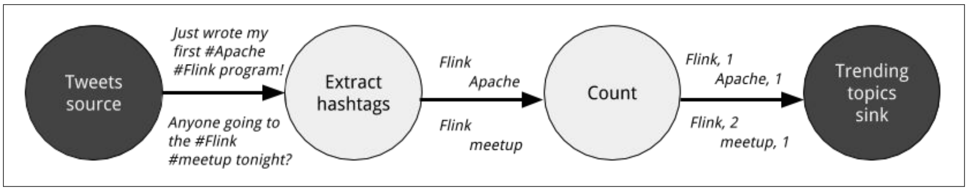


Figure 1-1. A logical dataflow graph to continuously count hashtags. Nodes represent operators and edges denote data dependencies.

Dataflow graphs like the one of Figure 2.1 are called logical because they convey a high-level view of the computation logic. In order to execute a dataflow program, its logical graph is converted into a physical dataflow graph, which includes details about how the computation is going to be executed. For instance, if we are using a distributed processing engine, each operator might have several parallel tasks running on different physical machines. Figure 2.2 shows a physical dataflow graph for the logical graph of Figure 2.1. While in the logical dataflow graph the nodes represent operators, in the physical dataflow, the nodes are tasks. The “Extract hashtags” and “Count” operators have two parallel operator tasks, each performing a computation on a subset of the input data.

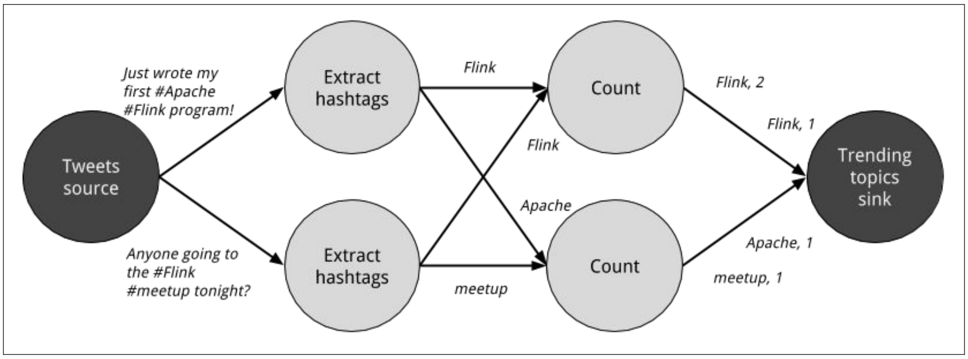


Figure 1-2. A physical dataflow plan for counting hashtags. Nodes represent tasks.

Data parallelism and task parallelism

We can exploit parallelism in dataflow graphs in different ways. First, we can partition our input data and have tasks of the same operation execute on the data subsets in parallel. This type of parallelism is called data parallelism. Data parallelism is useful because it allows for processing large volumes of data and spreading the computation load across several computing nodes. Second, we can have tasks from different operators performing computations on the same or different data in parallel. This type of parallelism is called task parallelism. Using task parallelism we can better utilize the computing resources of a cluster.

Data exchange strategies

Data exchange strategies define how data items are assigned to tasks in a physical dataflow graph. Data exchange strategies can be automatically chosen by the execution engine depending on the semantics of the operators or explicitly imposed by the dataflow programmer. Here, we briefly review some common data exchange strategies.

- The **forward** strategy sends data from a task to a receiving task. If both tasks are located on the same physical machine (which is often ensured by task schedulers), this exchange strategy avoids network communication.
- The **broadcast** strategy sends every data item to all parallel tasks of an operator. Because this strategy replicates data and involves network communication, it is fairly expensive.
- The **key-based** strategy partitions data by a key attribute and guarantees that data items having the same key will be processed by the same task. In Figure 2.2, the output of the "Extract hashtags" operator is partitioned by key (the hashtag), so that the count operator tasks can correctly compute the occurrences of each hashtag.
- The **random** strategy uniformly distributes data items to operator tasks in order to evenly distribute the load across computing tasks.

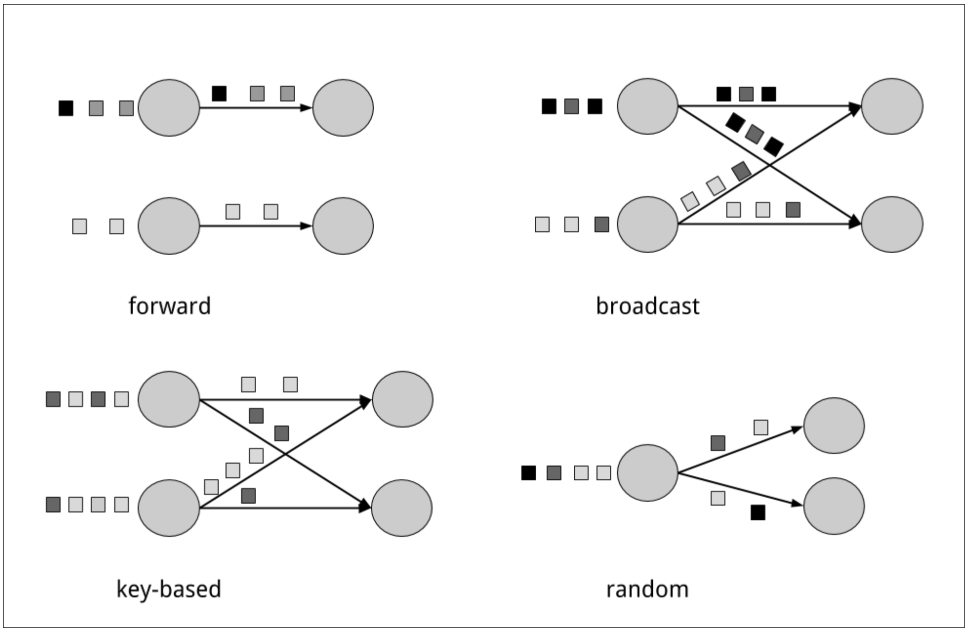


Figure 1-3. Data exchange strategies.

Processing infinite streams in parallel

Now that we have become familiar with the basics of dataflow programming, it's time to see how these concepts apply to processing data streams in parallel. But first, we define the term *data stream*:

A data stream is a potentially unbounded sequence of events

Events in a data stream can represent monitoring data, sensor measurements, credit card transactions, weather station observations, online user interactions, web searches, etc. In this section, we are going to learn the concepts of processing infinite streams in parallel, using the dataflow programming paradigm.

Latency and throughput

In the previous chapter, we saw how streaming applications have different operational requirements from traditional batch programs. Requirements also differ when it comes to evaluating performance. For batch applications, we usually care about the total execution time of a job, or how long it takes for our processing engine to read the input, perform the computation, and write back the result. Since streaming applications run continuously and the input is potentially unbounded, there is no notion of total execution time in data stream processing. Instead, streaming applications must provide results for incoming data *as fast as possible* while being able to handle high *ingest rates* of events. We express these performance requirements in terms of *latency* and *throughput*.

Latency

Latency indicates how long it takes for an event to be processed. Essentially, it is the time interval between receiving an event and seeing the effect of processing this event in the output. To understand latency intuitively, consider your daily visit to your favorite coffee shop. When you enter the coffee shop, there might be other customers inside already. Thus, you wait in line and when it is your turn you make an order. The cashier receives your payment and passes your order to the barista who prepares your beverage. Once your coffee is ready, the barista calls your name and you can pick up your coffee from the bench. Your service latency is the time you spend in the coffee shop, from the moment you enter until you have the first sip of coffee.

In data streaming, latency is measured in units of time, such as milliseconds. Depending on the application, we might care about *average* latency, *maximum* latency, or *percentile* latency. For example, an average latency value of 10ms means that events are processed within 10ms on average. Instead, a 95th-percentile latency value of 10ms means that 95% of events are processed within 10ms. Average values hide the true distribution of processing delays and might make it hard to detect problems. If the barista runs out of milk right before preparing your cappuccino, you will have to wait until they bring some from the supply room. While you might get annoyed by this delay, most other customers will still be happy.

Ensuring low latency is critical for many streaming applications, such as fraud detection, raising alarms, network monitoring, and offering services with strict service level agreements (SLAs). Low latency is what makes stream processing attractive and enables what we call *real-time* applications. Modern stream processors, like Apache Flink, can offer latencies as low as a few milliseconds. In contrast, traditional batch processing latencies typically range from a few minutes to several hours. In batch processing we first need to gather the events in batches and only then we are able to process them. Thus, the latency is bounded by the arrival time of the last event in each batch and naturally depends on the batch size. True stream processing does not introduce such artificial delays and therefore can achieve really low latencies. In a true streaming model, events can be processed as soon as they arrive in the system and latency reflects the actual work that has to performed on each event.

Throughput

Throughput is a measure of the system's processing capacity, i.e. its *rate* of processing. That is, throughput tells us how many events the system can process per time unit.

Revisiting the coffee shop example, if the shop is open from 7am to 7pm and it serves 600 customers in one day, then its average throughput would be 50 customers per hour. While we want latency to be as low as possible, we generally want throughput to be as high as possible.

Throughput is measured in events or operations per time unit. It is important to note that the rate of processing depends on the rate of arrival; low throughput does not necessarily indicate bad performance. In streaming systems we usually want to ensure that our system can handle the maximum expected rate of events. That is, we are primarily concerned with determining the *peak* throughput, i.e. the performance limit when our system is at its maximum load. To better understand the concept of peak throughput, let us consider that our system resources are completely unused. As the first event comes in, it will be immediately processed with the minimum latency possible. If you are the first customer showing up at the coffee shop right after it opened its doors in the morning, you will be served immediately. Ideally, we would like this latency to remain constant and independent of the rate of the incoming events. However, once we reach a rate of incoming events such that the system resources are fully used, we will have to start buffering events. In the coffee shop example, you will probably see this happening right after lunch. Many people show up at the same time and you have to wait in line to place your order. At this point we have reached the peak throughput and further increasing the event rate will only result in worse latency. If the system continues to receive data at a higher rate than it can handle, buffers might become unavailable and data might get lost. This situation is commonly known as *backpressure* and there exist different strategies to deal with it. In Chapter 3, we look at Flink's backpressure mechanism in detail.

Latency vs. throughput

At this point, it should be quite clear that latency and throughput are not independent metrics. If events take long to travel in the data processing pipeline, we cannot easily ensure high throughput. Similarly, if a system's capacity is small, events will be buffered and have to wait before they get processed.

Let us revisit the coffee shop example to clarify how latency and throughput affect each other. First, it should be clear that there is an optimal latency in the case of no load. That is, we will get the fastest service if we are the only customer in the coffee shop. However, during busy times, customers will have to wait in line and latency will increase. Another factor that affects latency and consequently throughput is the time it takes to process an event, or the time it takes for each customer to be served in our coffee shop. Imagine that during Christmas holiday season, baristas have to draw a Santa Claus on the cup of each coffee they serve. This way, the time to prepare a single beverage will increase, causing each person to spend more time in the coffee shop, thus lowering the overall throughput.

Then, can we somehow get both low latency and high throughput or is this a hopeless endeavour? One way we can lower latency is by hiring a more skilled barista, i.e. one that prepares coffees faster. At high load, this change will also increase throughput, because more customers will be served in the same amount of time. Another way to achieve the same result is to hire a second barista, that is, to exploit parallelism. The main take-away here is that lowering latency actually increases throughput. Naturally, if a system can perform operations faster, it can perform more operations at the same amount of time. In fact, that is what we achieve by exploiting parallelism in a stream processing pipeline. By processing several streams in parallel, we can lower the latency while processing more events at the same time.

Operations on data streams

Stream processing engines usually provide a set of built-in operations to ingest, transform, and output streams. These operators can be combined into dataflow processing graphs to implement the logic of streaming applications. In this section, we describe the most common streaming operations.

Operations can be either *stateless* or *stateful*. Stateless operations do not maintain any internal state. That is, the processing of an event does not depend on any events seen in the past and no history is kept. Stateless operations are easy to parallelize, since events can be processed independently of each other and of their arriving order. Moreover, in the case of a failure, a stateless operator can be simply restarted and continue processing from where it left off. On the contrary, stateful operators may maintain information about the events they have received before. This state can be updated by incoming events and can be used in the processing logic of future events. Stateful stream processing applications are more challenging to parallelize and operate in a fault tolerance manner because state needs to be efficiently partitioned and reliably recovered in the case of failures. We further discuss stateful stream processing, failure scenarios, and consistency in the end of this chapter.

Data ingestion and data egress

Data ingestion and data egress operations allow the stream processor to communicate with external systems. Data *ingestion* is the operation of fetching raw data from external sources and converting it into a format that is suitable for processing. Operators that implement data ingestion logic are called *data sources*. A data source can ingest data from a TCP socket, a file, a Kafka topic, or a sensor data interface. Data *egress* is the operation of producing output in a form that is suitable for consumption by external systems. Operators that perform data egress are called *data sinks* and examples include files, databases, message queues, and monitoring interfaces.

Transformation operations

Transformation operations are single-pass operations that process each event independently. These operations consume one event after the other and apply some transformation to the event data, producing a new output stream. The transformation logic can be either integrated in the operator or provided by a user-defined function (UDF). UDFs are written by the application programmer and implement custom computation logic.

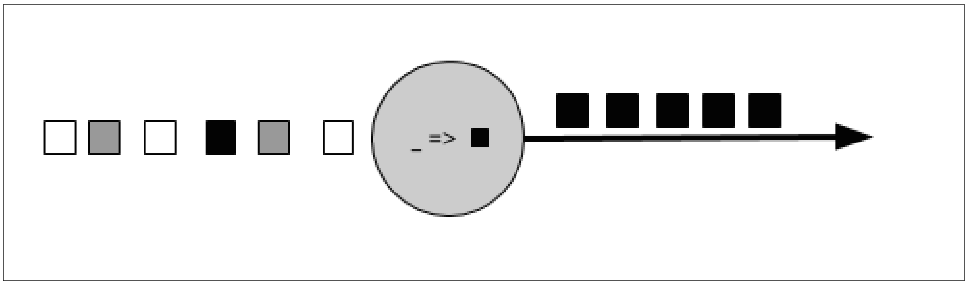


Figure 1-4. A streaming operator with a UDF that turns each incoming event into a black event.

Operators can accept multiple inputs and produce multiple output streams. They can also modify the structure of the dataflow graph by either splitting a stream into multiple streams or merging streams into a single flow. We discuss the semantics of all operators available in Flink in Chapter 5.

Rolling aggregations

A rolling aggregation is an aggregation, such as sum, minimum, and maximum, that is continuously updated for each input event. Aggregation operations are stateful and combine the current state with the incoming event to produce an updated aggregate value. Note that to be able to efficiently combine the current state with an event and produce a single value, the aggregation function must be associative and commutative. Otherwise, the operator would have to store the complete stream history. Figure 2.10 shows a rolling minimum aggregation. The operator keeps the current minimum value and accordingly updates it for each incoming event.

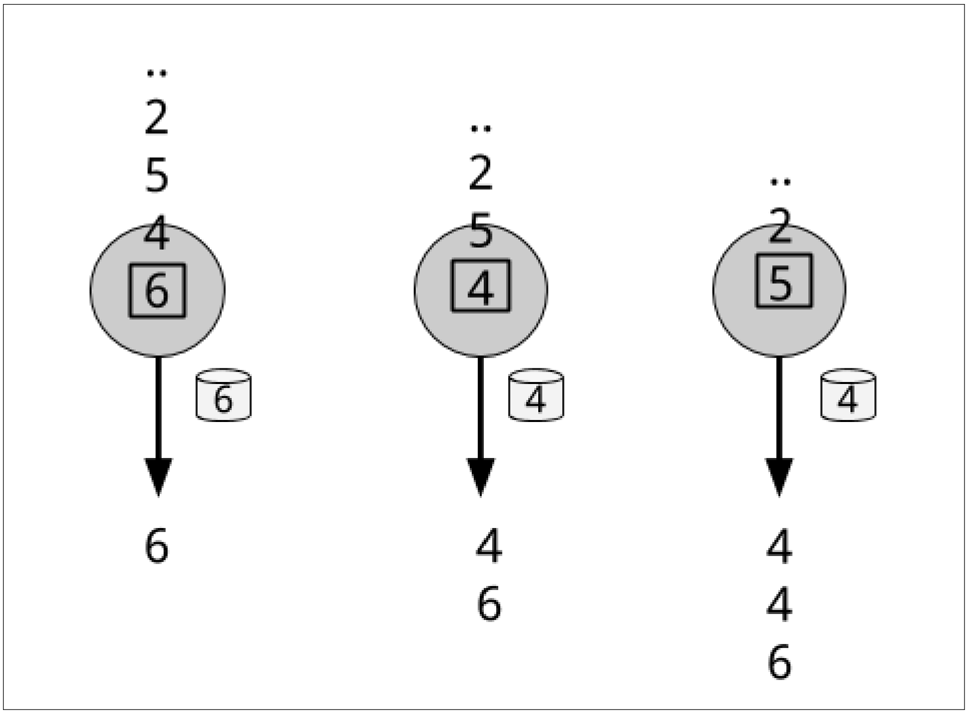


Figure 1-5. A rolling minimum aggregation operation.

Windows

Transformations and rolling aggregations process one event at a time to produce output events and potentially update state. However, some operations must collect and buffer records to compute their result. Consider for example a streaming join operation or a holistic aggregate, such as median. In order to evaluate such operations efficiently on unbounded streams, we need to limit the amount of data these operations maintain. In this section, we discuss window operations, which provide such a mechanism.

Apart from having a practical value, windows also enable semantically interesting queries on streams. We have seen how rolling aggregations encode the history of the whole stream in an aggregate value and provide us with a low-latency result for every event. This is fine for some applications, but what if we are only interested in the most recent data? Consider an application that provides real-time traffic information to drivers so that they can avoid congested routes. In this scenario, we want to know if there has been an accident in a certain location within the last few minutes. Knowing whether there ever happened an accident is not so valuable. What's more, by reducing the stream history to a single aggregate, we lose the information about how our data varies over time. For instance, we might want to know how many vehicles cross an intersection every 5 minutes.

Window operations continuously create finite sets of events called buckets from an unbounded event stream and let us perform computations on these finite sets. Events are usually assigned to buckets based on data properties or based on time. To properly define window operator semantics, we need to answer two main questions: "*how are events assigned to buckets?*" and "*how often does the window produce a result?*". The behavior of windows is defined by a set of policies. Window policies decide when new buckets are created, which events are assigned to which buckets, and when the contents of a bucket get evaluated. The latter decision is based on a trigger condition. When the trigger condition is met, the bucket contents are sent to an evaluation function that applies the computation logic on the bucket elements. Evaluation functions can be aggregations like sum or minimum or custom operations applied on the bucket's collected elements. Policies can be based on time (e.g. events received in the last 5 seconds), on count (e.g. the last 100 events), or on a data property. In this section, we describe the semantics of common window types.

- **Tumbling** windows assign events into non-overlapping buckets of fixed size. When the window border is passed, all the events are sent to an evaluation function for processing. Count-based tumbling windows define how many events are collected before triggering evaluation. Figure 2.6 shows a count-based tumbling window that discretizes the input stream into buckets of 4 elements. Time-based tumbling windows define a time interval during which events are buffered in the bucket. Figure 2.7 shows a time-based tumbling window that gathers events into buckets and triggers computation every 10 minutes.

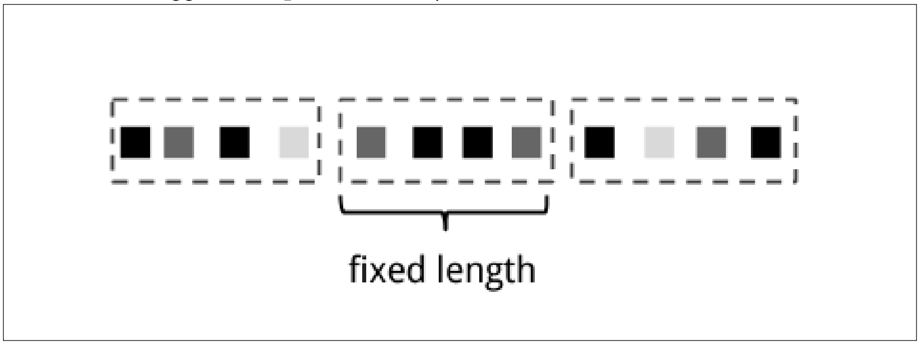


Figure 1-6. Count-based tumbling window.

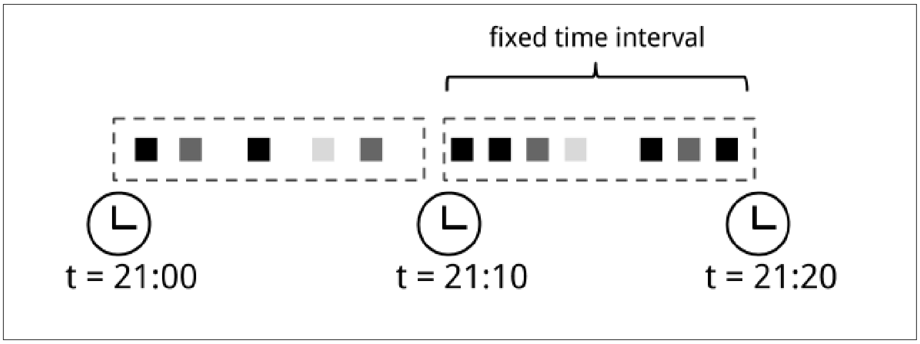


Figure 1-7. Time-based tumbling window.

- **Sliding** windows assign events into overlapping buckets of fixed size. Thus, an event might belong to multiple buckets. We define sliding windows by providing their length and their *slide*. The slide value defines the interval at which a new bucket is created. The sliding count-based window of Figure 2.13 has a length of 4 events and slide of 3 events.

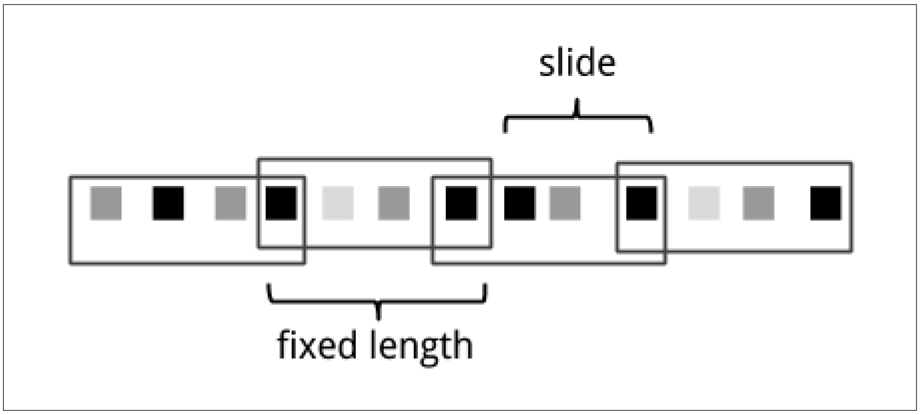


Figure 1-8. Sliding count-based window with a length of 4 events and a slide of 3 events.

- **Session** windows are useful in a common real-world scenario where neither tumbling nor sliding windows can be applied. Consider an application that analyzes online user behavior. In such applications, we would like to group together events that origin from the same period of user activity or *session*. Sessions comprise of a series of events happening in adjacent times followed by a period of inactivity. For example, user interactions with a series of news articles one after the other could be considered a session. Since the length of a session is not defined beforehand but depends on the actual data, tumbling and sliding windows cannot be applied in this scenario. Instead, we need a window operation that assigns events belonging to the same session in the same bucket. Session windows group events in session based on a *session gap* value that defines the time of inactivity to consider a session closed. Figure 2.9 shows a session window.

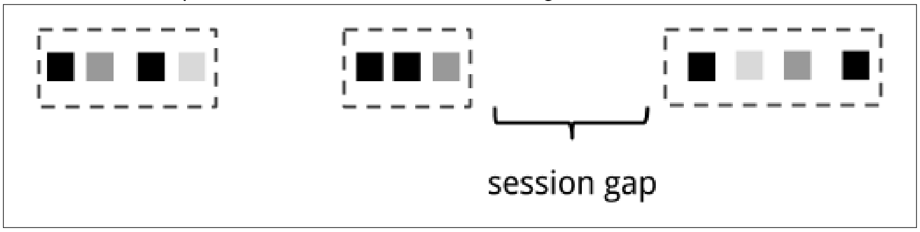


Figure 1-9. Session window.

All the window types that we have seen so far are *global* windows and operate on the full stream. In practice though we might want to partition a stream into multiple logical streams and define *parallel* windows. For instance, if we are receiving measurements from different sensors, we probably want to group the stream by sensor id before applying a window computation. In parallel windows, each partition applies the window policies independently of other partitions. Figure 2.10 shows a parallel count-based tumbling window of length 2 which is partitioned by event color.

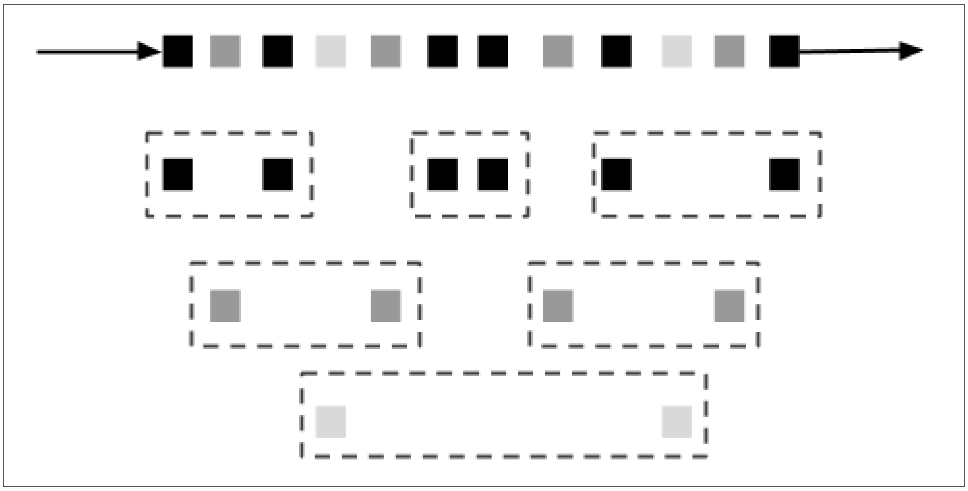


Figure 1-10. A parallel count-based tumbling window of length 2.

Window operations are closely related to two dominant concepts in stream processing: time semantics and state management. Time is perhaps the most important aspect of stream processing. Even though low latency is an attractive feature of stream processing, its true value is way beyond just offering fast analytics. Real-world systems, networks, and communication channels are far from perfect, thus streaming data can often be delayed or arrive out-of-order. It is crucial to understand how we can deliver accurate and deterministic results under such conditions. What's more, streaming applications that process events as they are produced should also be able to process historical events in the same way, thus enabling offline analytics or even time travel analyses. Of course, none of this matters if our system cannot guard state against failures. All the window types that we have seen so far need to buffer data before performing an operation. In fact, if we want to compute anything interesting in a streaming application, even a simple count, we need to maintain state. Considering that streaming applications might run for several days, months, or even years, we need to make sure that state can be reliably recovered under failures and that our system can guarantee accurate results even if things break. In the rest of this chapter, we are going to look deeper into the concepts of time and state guarantees under failures in data stream processing.

Time semantics

In this section, we introduce time semantics and describe the different notions of time in streaming. We discuss how a stream processor can provide accurate results with out-of-order events and how we can support historical event processing and time travel with streaming.

What is the meaning of one minute?

When dealing with a potentially unbounded stream of continuously arriving events, time becomes a central aspect of applications. Let's assume we want to compute results continuously, for example every one minute. What would *one minute* really mean in the context of our streaming application?

Consider a program that analyzes events generated by users playing online mobile games. The application receives a user's activity and provides rewards in the game, such as extra lives and level-ups, based on how fast the user meets the game's goals. For example, if users pop 500 bubbles within one minute, they get a level-up. Alice is a devoted player who plays the game every morning during her commute to work. The problem is that Alice lives in Berlin and she takes the subway to work. And everyone knows that the mobile internet connection in the Berlin subway is lousy. Consider the case where Alice starts popping bubbles while her phone is connected to the network and sends events to the analysis application. Then suddenly, the train enters a tunnel and her phone gets disconnected. Alice keeps on playing and the game events are buffered in her phone. When the train exits the tunnel, she comes back online, and pending events are sent to the application. What should the application do? What's the meaning of one minute in this case? Does it include the time Alice was offline or not?

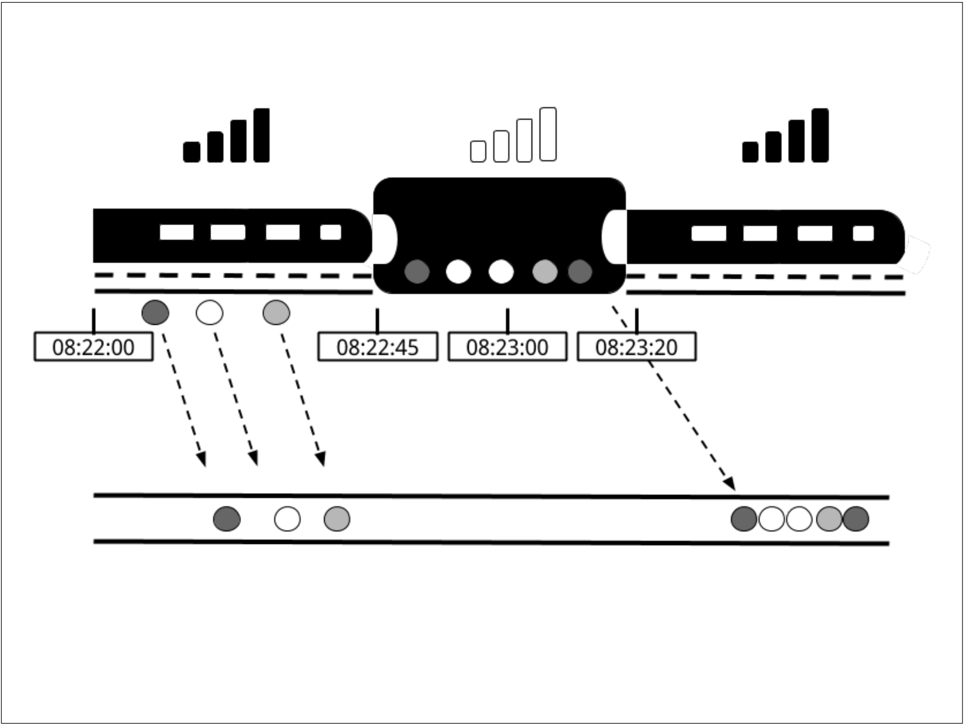


Figure 1-11. Playing online mobile games in the subway. An application receiving game events would experience a gap when the train goes through a tunnel and network connection is lost. Events are buffered in the player's phone and delivered to the application when the network connection is restored.

Online gaming is a simple scenario showing how operator semantics should depend on the time when events actually happen and not the time when the application receives the events. In the case of a mobile game, consequences can be as bad as Alice getting disappointed and never playing again. But there are much more time-critical applications whose semantics we need to guarantee. If we only consider how much data we receive within one minute, our results will vary and depend on the speed of the network connection or the speed of the processing. Instead, what really defines the amount of events in one minute is the time of the data itself.

In Alice's game example, the streaming application could operate with two different notions of time, Processing time or Event time. We describe both notions in the following sections.

Processing time

Processing time is the time of the local clock on the machine where the operator processing the stream is being executed. A processing-time window includes all events that happen to have arrived at the window operator within a time period, as measured by the wall-clock of its machine. In Alice's case, a processing-time window would continue counting time when her phone gets disconnected, thus not accounting for her game activity during that time:

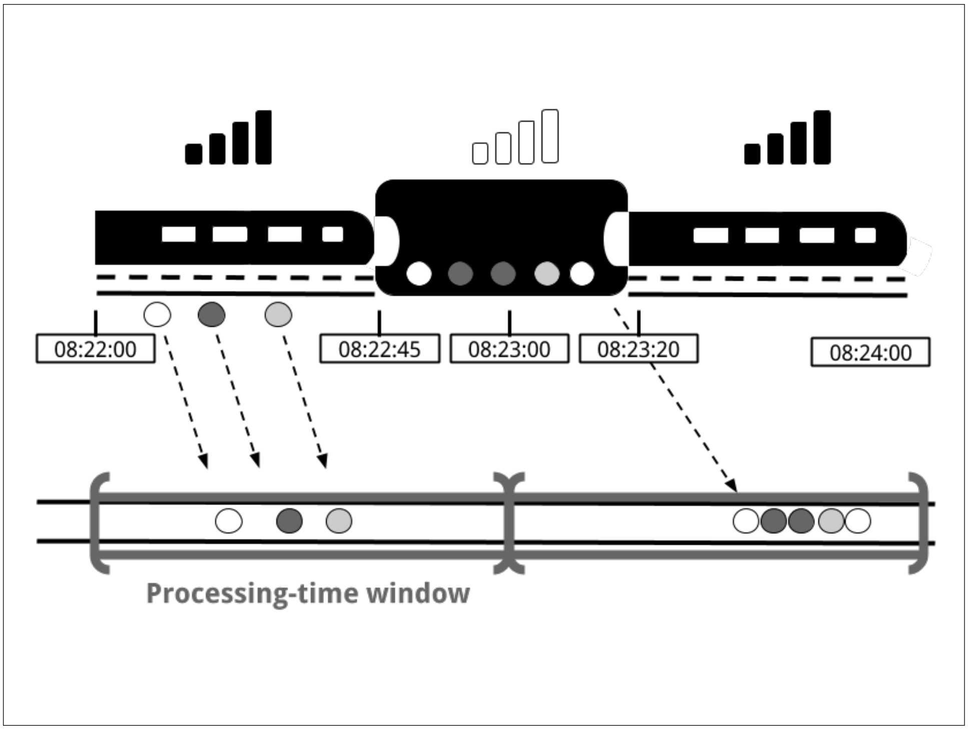


Figure 1-12. A processing-time window continues counting time when Alice's phone gets disconnected.

Event time

Event-time is the time when an event in the stream actually happened. Event time is based on a *timestamp* that is attached on the events of the stream. Timestamps usually exist inside the event data before they enter the processing pipeline (e.g. event creation time). In Alice's case, an event-time window would correctly place events in a window, reflecting the reality of how things happened, even though some events were *delayed*:

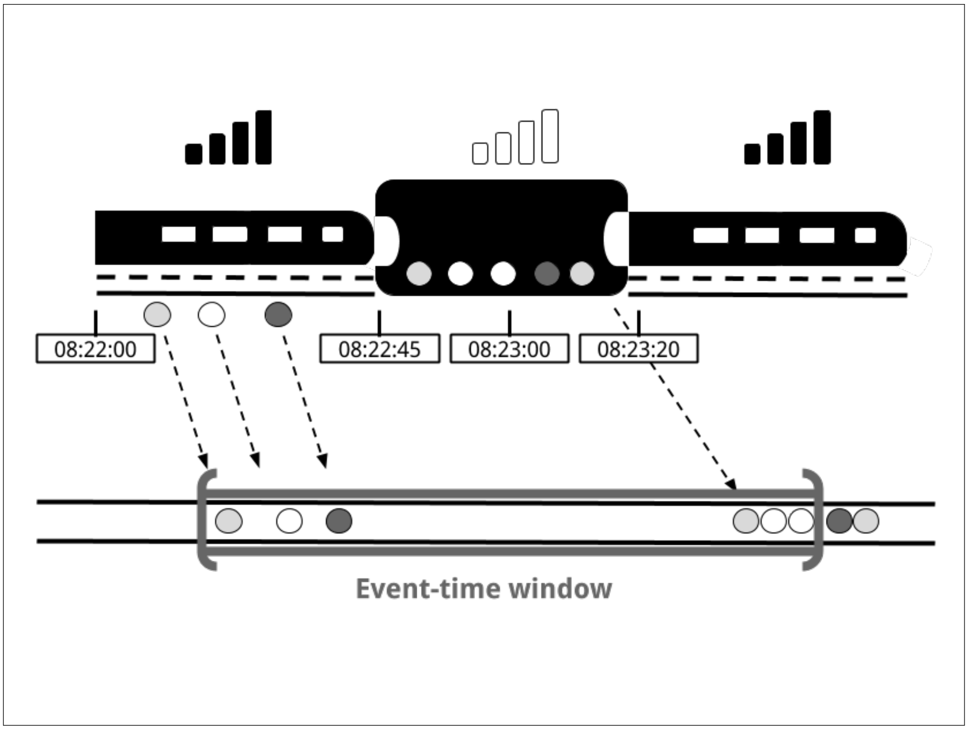


Figure 1-13. Event-time correctly places events in a window, reflecting the reality of how things happened.

Event-time completely decouples the processing speed from the results. Operations based on event-time are predictable and their results deterministic. An event-time window computation will yield the same result no matter how fast the stream is processed or when the events arrive at the operator.

Handling delayed events is only one of the challenges that we can overcome with event time. Except from experiencing network delays, streams might be affected by many other factors resulting in events arriving *out-of-order*. Consider Bob, another player of the online mobile game, who happens to be on the same train as Alice. Bob and Alice play the same game but they have different mobile providers. While Alice's phone loses connection when inside the tunnel, Bob's phone remains connected and delivers events to the gaming application.

By relying on event time, we can guarantee result correctness even in such cases. What's more, when combined with replayable streams, the determinism of timestamps gives us the ability to *fast-forward* the past. That is, we can re-play a stream and analyze historic data as if events are happening in real-time. Additionally, we can fast-forward the computation to the present so that once our program catches up with the events happening now, it can continue as a real-time application using exactly the same program logic.

Watermarks

In our discussion about event-time windows so far, we have overlooked one very important aspect: *how do we decide when to trigger an event-time window*? That is, how long do we have to wait before we can be certain that we have received all events that happened before a certain point of time? And how do we even know that data will be delayed? Given the unpredictable reality of distributed systems and arbitrary delays that might be caused by external components, there is no categorically correct answer to these questions. In this section, we will see how we can use the concept of *watermarks* to configure event-time window behavior.

A watermark is a global progress metric that indicates a certain point in time when we are confident that no more delayed events will arrive. In essence, watermarks provide a logical clock which informs the system about the current event time. When an operator receives a watermark with time T, it can assume that no further events with timestamp less than T will be received. Watermarks are essential to both event-time windows and operators handling out-of-order events. Once a watermark has been received, operators are signaled that all timestamps for a certain time interval have been observed and either trigger computation or order received events.

Watermarks provide a configurable trade-off between results confidence and latency. *Eager* watermarks ensure low latency but provide lower confidence. In this case, late events might arrive after the watermark and we should provide some code to handle them. On the other hand, if watermarks are too slow to arrive, we have high confidence but we might unnecessarily increase processing latency.

In many real-world applications, the system does not have enough knowledge to perfectly determine watermarks. In the mobile gaming case for example, it is practically impossible to know for how long a user might remain disconnected; they could be going through a tunnel, boarding a plane, or never playing again. No matter if watermarks are user-defined or automatically generated, tracking global progress in a distributed system can be problematic, especially in the presence of straggler tasks. Hence, simply relying on watermarks might not always be a good idea. Instead, it is crucial that the stream processing system provides some mechanism to deal with events that might arrive after the watermark. Depending on the application requirements, we might want to ignore such events, log them, or use them to correct previous results.

Processing time vs. event time

At this point, you might be wondering: *Since event time solves all of our problems, why even bother considering processing time?* The truth is that processing time can indeed be useful in some cases. Processing-time windows introduce the lowest latency possible. Since we do not take into consideration late events and out-of-order events, a window simply needs to buffer up events and immediately trigger computation once the specified time length is reached. Thus, for applications where speed is more important than accuracy, processing time comes handy. Another case is when we need to periodically report results in real-time, independently of their accuracy. An example application would be a real-time monitoring dashboard that displays event aggregates as they are received. To recap, processing time offers low latency but results depend on the speed of processing and are not deterministic. On the other hand, event time guarantees deterministic results and allows us to deal with events that are late or even out-of-order.

State and consistency models

We now turn to examine another extremely important aspect of stream processing, state. State is ubiquitous in data processing. It is required by any non-trivial computation. To produce a result, a UDF accumulates state over a period or number of events, e.g. to compute an aggregation or detect a pattern. Stateful operators use both incoming events and internal state to compute their output. Take for example a rolling aggregation operator that outputs the current sum of all the events it has seen so far. The operator keeps the current value of the sum as its internal state and updates it every time it receives a new event. Similarly, consider an operator that raises an alert when it detects a “high temperature” event followed by a “smoke” event within 10 minutes. The operator needs to store the “high temperature” event in its internal state, until it sees the “smoke” event or the until 10-minute time period expires.

The importance of state becomes even more evident if we consider the case of using a batch processing system to analyze an unbounded data set. In fact, this has been a common implementation choice before the rise of modern stream processors. In such a case, a job is executed repeatedly over batches of incoming events. When the job finishes, the result is written to persistent storage, and all operator state is lost. Once the job is scheduled for execution on the next batch, it cannot access the state of the previous job. This problem is commonly solved by delegating state management to an external system, such as a database. On the contrary, with continuously running streaming jobs, manipulating state in the application code is substantially simplified. In streaming we have durable state across events and we can expose it as a first-class citizen in the programming model.

Since streaming operators process potentially unbounded data, caution should be taken to not allow internal state to grow indefinitely. To limit the state size, operators usually maintain some kind of summary or *synopsis* of the events seen so far. Such a summary can be a count, a sum, a sample of the events seen so far, a window buffer, or a custom data structure that preserves some property interesting to the running application.

As one could imagine, supporting stateful operators comes with a few implementation challenges. First, the system needs to efficiently manage the state and make sure it is protected from concurrent updates. Second, parallelization becomes complicated, since results depend on both the state and incoming events. Fortunately, in many cases, we can partition the state by a key and manage the state of each partition independently. For example, if we are processing a stream of measurements from a set of sensors, we can use partitioned operator state to maintain state for each sensor independently. The third and biggest challenge that comes with stateful operators is ensuring that the state can be recovered and that results will be correct in the presence of failures. In the next section, we look into task failures and result guarantees in detail.

Task failures

Operator state in streaming jobs is very valuable and should be guarded against failures. If state gets lost during a failure, results will be incorrect after recovery. Streaming jobs run for long periods of time, thus state might be collected over several days or even months. Reprocessing all input to reproduce lost state in the case of failures would be both very expensive and time-consuming.

In the beginning of this chapter, we saw how we can model streaming programs as dataflow graphs. Before execution, these are translated into physical dataflow graphs of many connected parallel tasks, each running some operator logic, consuming input streams and producing output streams for other tasks. Typical real-world setups can easily have hundreds of such tasks running in parallel on many physical machines. In long-running, streaming jobs, each of these tasks can fail at any time. How can we ensure that such failures are handled transparently so that our streaming job can continue to run? In fact, we would like our stream processor to not only continue the processing in the case of task failures, but also provide correctness guarantees about the result and operator state. We discuss all these matters in this section.

What is a task failure?

For each event in the input stream, a task performs the following steps: (1) receive the event, i.e. store it in a local buffer, (2) possibly update internal state, and (3) produce an output record. A failure can occur during any of these steps and the system has to clearly define its behavior in a failure scenario. If the task fails during the first step, will the event get lost? If it fails after it has updated its internal state, will it update it again after it recovers? And in those cases, will the output be deterministic?

We assume reliable network connections, such that no records are dropped or duplicated and all events are eventually delivered to their destination in FIFO order. Note that Flink uses TCP connections, thus these requirements are guaranteed. We also assume perfect failure detectors and that no task will intentionally act maliciously; that is, all non-failed tasks follow the above steps.

In a batch processing scenario, we can solve all these problems easily since all the input data is available. The most trivial way would be to simply restart the job, but then we would have to replay all data. In the streaming world, however, dealing with failures is not a trivial problem. Streaming systems define their behavior in the presence of failures by offering result guarantees. Next, we review the types of guarantees offered by modern stream processors and some mechanisms that systems implement to achieve those guarantees.

Result guarantees

Before we describe the different types of guarantees, we need to clarify a few points that are often the source of confusion when discussing task failures in stream processors. In the rest of this chapter, when we talk about "result guarantees" we refer to the consistency of the internal state of the stream processor. That is, we are concerned with what the application code sees as state value after recovering from a failure. Note that stream processors can normally only guarantee result correctness for state that lives inside the stream processor itself. However, guaranteeing exactly-once delivery of results to external systems is not possible in general. For example, once data has been emitted to a sink, it is hard to guarantee result correctness, since the sink might not provide transactions to revert results that have been previously written.

At-Most-Once

The simplest thing to do when a task fails is to do nothing to recover lost state and replay lost events. At-most-once is the trivial case that guarantees processing of each event at-most-once. In other words, events can be simply dropped and there is no mechanism to ensure result correctness. This type of guarantee is also known as "no-guarantee" since even a system that drops every event can fulfil it. Having no guarantees whatsoever sounds like a terrible idea, but it might be fine, if we can live with approximate results and all we care about is providing the lowest latency possible.

At-Least-Once

In most real-world applications, the minimum requirement is that events do not get lost. This type of guarantee is called at-least-once and it means that all events will definitely be processed, even though some of them might be processed more than once. Duplicate processing might be acceptable if application correctness only depends on the completeness of information. For example, determining whether a specific event occurs in the input stream can be correctly realized with at-least-once guarantees. In the worst case, we will locate the event more than once. However, counting how many times a specific event occurs in the input stream might return the wrong result under at-least-once guarantees.

In order to ensure at-least-once result correctness, we need to have a mechanism to replay events, either from the source or from some buffer. Persistent event logs write all events to durable storage, so that they can be replayed if a task fails. Another way to achieve equivalent functionality is using record acknowledgements. This method stores every event in a buffer until its processing has been acknowledged by all tasks in the pipeline, at which point the event can be discarded.

Exactly-Once

This is the strictest and most challenging to achieve type of guarantee. Exactly-once result guarantees means that not only there will be no event loss, but also updates on the internal state will be applied exactly once for each event. In essence, exactly-once guarantees mean that our application will provide the correct result, as if a failure never happened.

Providing exactly-once guarantees requires at-least-once guarantees, thus a data replay mechanism is again necessary. Additionally, the stream processor needs to ensure internal state consistency. That is, after recovery, it should know whether an event update has already been reflected on the state or not. Transactional updates is one way to achieve this result, however, it can incur substantial performance overhead. Instead, Flink uses a lightweight snapshotting mechanism to achieve exactly-once result guarantees. We discuss Flink's fault-tolerance algorithm in Chapter 3.

End-to-end Exactly-Once

The types of guarantees we have seen so far refer to the stream processor component only. In a real-word streaming architecture however, it is common to have several connected components. In the very simple case, there will be at least one source and one sink apart from the stream processor. End-to-end guarantees refer to result correctness across the data processing pipeline. To assess end-to-end guarantees, one has to consider all the components of an application pipeline. Each component provides its own guarantees and the end-to-end guarantee of the complete pipeline would be the weakest of each of its components. It is important to note that sometimes we can get stronger semantics with weaker guarantees. A common case is when a task performs idempotent operations, like maximum or minimum. In this case, we can achieve exactly-once semantics with at-least-once guarantees.

Summary

In this chapter, we have introduced data stream processing by describing fundamental concepts and ideas. We have explained the dataflow programming model and we have shown how streaming applications can be expressed as distributed dataflow graphs. Next, we have looked into the requirements of processing infinite streams in parallel and we have discussed the importance of latency and throughput for stream applications. We have introduced basic streaming operations and we have showed how we can compute meaningful results on unbounded input data using windows. We have wondered about the meaning of time in stream processing and we have compared the notions of event time and processing time. Finally, we have seen why state is important in streaming applications and how we can guard it against failures and guarantee correct results.

Up to this point, we have considered streaming concepts independently of Apache Flink. In the rest of this book, we are going to see how Flink actually implements these concepts and how you can use its DataStream APIs to write applications that use all of the features that we have introduced so far.

CHAPTER 2

The Architecture of Apache Flink

The previous chapter discussed important concepts of distributed stream processing, such as parallelization, time, and state. In this chapter we give a high-level introduction into Flink's architecture and describe how Flink addresses the aspects of stream processing that we discussed before. In particular, we explain how Flink is able to provide low latency and high throughput processing, how it handles time in streaming applications, and how its fault tolerance mechanism works. This chapter provides relevant background information to successfully implement and operate advanced streaming applications with Apache Flink. It will help you to understand Flink's internals and to reason about the performance and behavior of streaming applications.

System Architecture

Flink's architecture follows the master-worker pattern implemented by many distributed systems. In Flink, the master process is called *JobManager*[1] and a worker process is called *TaskManager*. A Flink cluster consists of at least one JobManager and one or more TaskManagers. *Client* processes submit applications to the JobManager. Every process runs in a separate Java Virtual Machine (JVM).

Executing an Application Step-by-Step

The responsibilities of the JobManager, TaskManagers, and Client processes are best explained by describing the execution of a streaming application at a high level. The following figure shows the steps of submitting and executing a streaming application on Flink.

1 see discussion of highly-available setups later in this chapter.

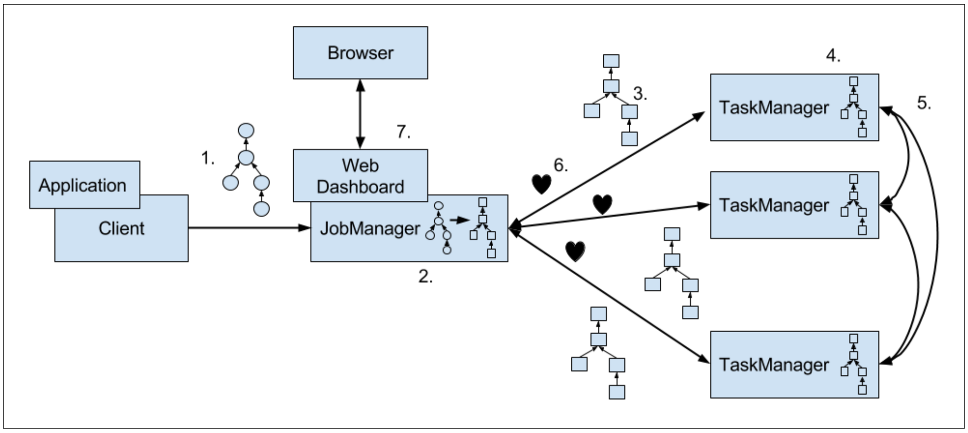


Figure 2-1. Job Submission and Execution.

1. A streaming application is a regular Java or Scala program. Flink's DataStream API, internally construct a so-called *JobGraph*. The JobGraph is Flink's representation of a non-parallel logical dataflow graph (see Chapter 2). Flink's client code, which is embedded in the program, ships the JobGraph and all required resources of the application (classes, libraries, and configuration files) to the JobManager.
2. The JobManager receives an application and converts the JobGraph into a parallelized physical dataflow graph called the *ExecutionGraph*. The ExecutionGraph consists of at least one task for each logical operator. Operators with multiple tasks will be executed in a data-parallel fashion. Applications with multiple operators benefit from task parallelism.
3. The JobManager tracks the current state of each task in the lifecycle and assigns each task to a TaskManager for execution.
4. When a TaskManager receives a task, it immediately initializes and starts executing the task.
5. Running tasks exchange data with their preceding and succeeding tasks. If sending and receiving tasks run on different TaskManagers, the data is sent over the network. In case sender and receiver run in the same process, data is exchanged locally. In both cases the TaskManager is responsible for the data transfer between tasks. We discuss the data transfer between TaskManagers in more detail in a later section of this chapter.
6. TaskManager tells to the JobManager whenever a task changes its state (starting, running, failed). In addition, it periodically reports certain performance metrics to the JobManager.

7. Flink's web dashboard runs as part of the JobManager and publishes details about currently running or terminated applications. The JobManager exposes the same information also via a REST interface.

Resource and Task Isolation

A TaskManager is able to execute several tasks at the same time. These tasks can be of the same operator (data parallelism), a different operator (task parallelism), or even from a different application (job parallelism). A TaskManager provides a certain number of processing Slots to control the number of tasks that can be executed concurrently. A processing slot is able to execute one slice of an application, i.e., one task of each operator of the job. The following figure visualizes the relationship of TaskManagers, slots, tasks, and operators.

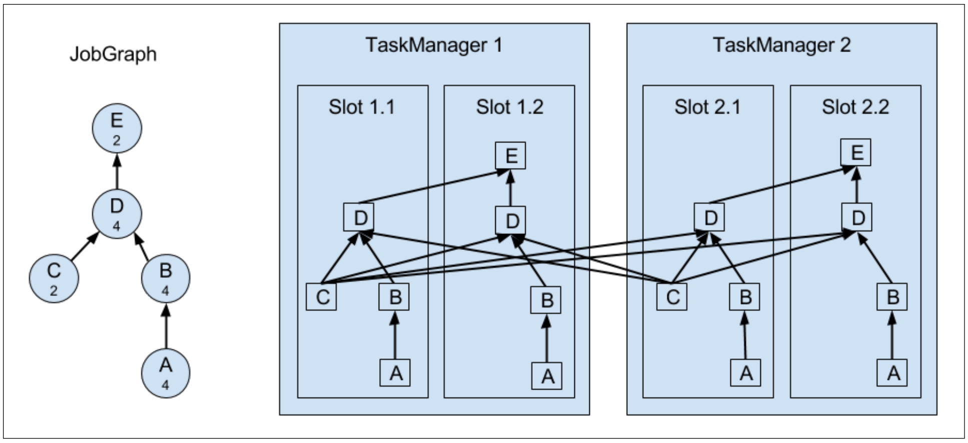


Figure 2-2. Operators, Tasks, and Processing Slots.

On the left hand side we see a JobGraph consisting of five operators. Operators A and C are sources and operator E is a sink. Operators C and E have a parallelism of two. The other operators have a parallelism of four. Since the maximum operator parallelism is four, the job requires at least four available processing slots to be executed. Given two TaskManagers with two processing slots each, this requirement is fulfilled. The JobManager parallelizes the JobGraph into an ExecutionGraph and assigns the tasks to the four available slots. The tasks of the operators with a parallelism of four are assigned to each slot. The two tasks of operators C and E are assigned to slots 1.1 and 2.1 and slots 1.2 and 2.2, respectively. Scheduling tasks as job slices to slots has the advantage that many tasks are co-located on the TaskManager which means that they can efficiently exchange data without accessing the network.

TaskManagers execute tasks as threads in the same JVM process. Threads are more lightweight than individual processes and have lower communication costs but do not strictly isolate tasks from each other. Hence, a single misbehaving task can kill the whole TaskManager process and all tasks which are being executed on the TaskManager. Flink features flexible deployment configurations and first class integration with cluster resource managers such as YARN and Apache Mesos (see Chapter X). Therefore, it is possible to isolate jobs by starting a Flink cluster per job or configuring multiple TaskManagers with a single processing slot per physical machine. By leveraging thread-parallelism inside of a TaskManager and the option to deploy several TaskManager processes per host, Flink offers a lot of flexibility to trade-of performance and resources isolation when deploying applications. We will discuss the configuration and setup of Flink clusters in detail in Chapter X.

The Networking Layer

In a running streaming application, processing tasks are continuously exchanging data. The TaskManagers take care of shipping data from sending tasks to receiving tasks. Although Flink is often perceived as a record-at-a-time streaming engine, the communication between tasks uses buffers to collect records before they are shipped to achieve satisfying throughput. This batching technique is basically the same mechanism as found in networking or disk IO protocols. Each TaskManager has a pool of byte buffers (by default 32KB in size) which are used to send and receive data.

If the sender task and receiver task run in the same TaskManager process, the sender task serializes the outgoing records into a byte buffer and puts the buffer into a queue once it is filled. The receiving task takes the buffer from the queue and deserializes the incoming records. Hence, no network communication is involved.

If the sender and receiver tasks run in separate TaskManager processes, the communication happens via the network stack of the operating system. For streaming applications, the data exchange must happen in a pipelined fashion, i.e., each pair of TaskManagers maintains a permanent TCP connection to exchange data [2]. In case of a shuffle connection pattern, each sender task needs to be able to send data to each receiving task. A TaskManager that executes a sender task needs for each receiving task one byte buffer into which the sender task serializes its outgoing data. Once a buffer is filled, it is shipped via the network to the receiving task. On the receiver side, a TaskManager needs one byte buffer for each receiving task. The following figure visualizes this architecture.

2 Batch applications can as well exchange data in a bulk fashion. In batch mode, the outgoing data is collected at the sender and transmitted as a batch over a temporary TCP connection to the receiver.

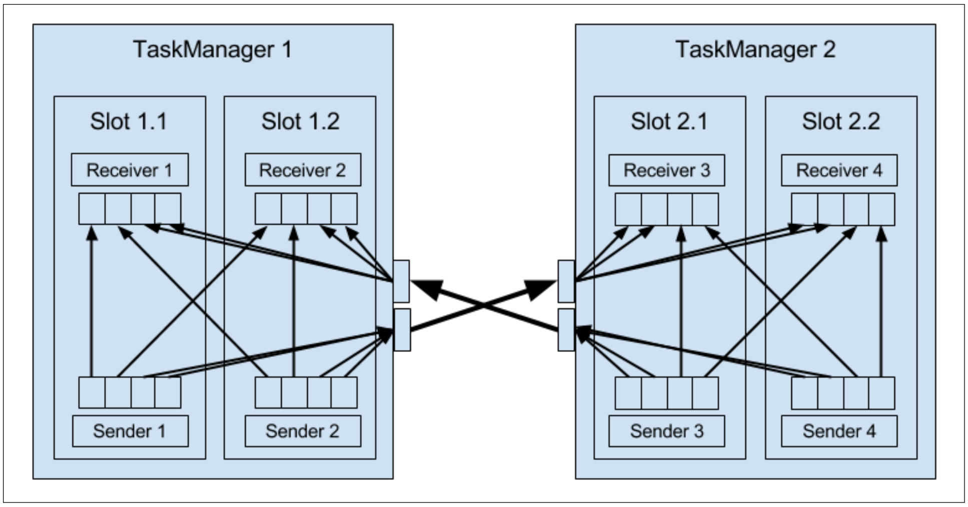


Figure 2-3. Flink's Networking Layer.

The figure shows four sender and four receiver tasks. Each sender task has four network buffers to send data to each receiver task and each receiver task has four buffers to receive data. Buffers which need to be sent to the other TaskManager are multiplexed over the same network connection. In order to enable a smooth pipelined data exchange, a TaskManager must be able to provide enough buffers to serve all outgoing and incoming connections concurrently. In case of a shuffle or broadcast connection, each sending task needs a buffer for each receiving task, i.e, the number of required buffers is quadratic to the parallelism of the involved operators.

High Throughput and Low Latency

Sending individual records over the wire is very inefficient and causes significant overhead. Buffering is a mandatory technique to fully utilize the network connections and to achieve a high network throughput. One disadvantage of buffering is that it adds latency because records are collected in a buffer instead of being immediately shipped. If a sender task does only rarely produce records for a specific receiving task, it might take a long time until a buffer is filled and shipped which would cause high processing latencies. In order to avoid such situations, Flink ensures that each buffer is shipped after a certain period of time regardless of how much it is filled. This timeout can be interpreted as an upper bound for the latency added by a network connection. However, the threshold does not serve as a strict latency SLA for the job as a whole because a job might involve multiple network connections and it does also not account for delays caused by the actual processing.

Flow Control with Back Pressure

Streaming applications which ingest streams with high volume can easily come to a point where a task is not able to process its input data at the rate at which it arrives. This might happen if the volume of an input stream is too high for the amount of resources allocated to a certain operator or if the input rate of a stream significantly varies and causes spikes of high load. Regardless of the reason why an operator cannot handle its input, this situation should never be a reason for a stream processor to terminate an application. Instead the stream processor should gracefully throttle the rate at which a streaming application ingests its input to the maximum speed at which the application can process the data. With a decent monitoring infrastructure in place, a throttling situation can be easily detected and usually resolved by adding more compute resources and increasing the parallelism of the bottleneck operator. The described flow control technique is called backpressure and an important feature of stream processors.

Flink naturally supports backpressure due to the design of its network layer. The following figure illustrates the behavior of the network stack when a receiving task is not able to process its input data at the rate at which it is emitted by the sender task.

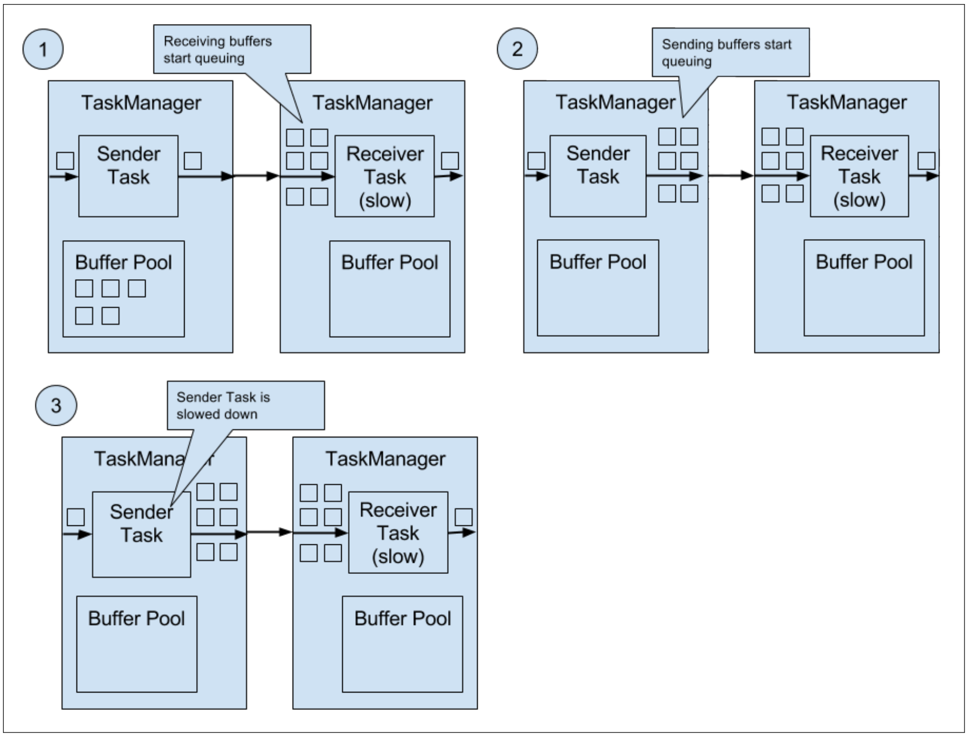


Figure 2-4. Backpressure in Flink.

The figure shows a sender and a receiver task running on different machines. (1) When the input rate of the application increases, the sender task can cope with the load but the receiver task starts to fall behind and is no longer able to handle its input load. Now the receiving TaskManager starts to queue buffers with received data. At some point the buffer pool of the receiving TaskManager is empty [3] and can no longer continue to buffer arriving data. (2) Next, the sending TaskManager starts queuing buffers with outgoing records until its own buffer pool is empty. (3) Finally, the sender task cannot emit anymore data and blocks until a new buffer becomes available. A task which is blocked due to a slow receiver behaves itself like a slow receiver and in turn slows down its predecessors. The slowdown escalates up to the sources of the streaming application. Eventually, the whole application is slowed down to the processing rate of the slowest operator.

Due to the flexible assignment of network buffers to queue outgoing or incoming data, Flink is able to handle temporary load spikes or slowed down tasks in a very graceful manner.

Handling Event Time

In Chapter 2, we highlighted the importance of time semantics for stream processing applications and explained the differences between processing time and event time. While processing time is easy to understand because it is based on the local time of the processing machine, it produces somewhat arbitrary, inconsistent, and non-reproducible results. In contrast, event time semantics yield reproducible and consistent results which is a hard requirement for many stream processing use-cases. However, event-time applications require some additional configuration compared to applications with processing-time semantics. Also the internals of stream processors that support event-time are more involved than the internals of a system that purely operates in processing-time. Flink provides intuitive and easy-to-use primitives for common event-time processing use cases but also allows implementing more advanced event-time applications with custom operators. For such advanced applications, a good understanding of Flink's internals is often helpful and sometimes also required. The previous chapter introduced two concepts that Flink leverages to provide event-time semantics: record timestamps and watermarks. In the following we will describe how Flink internally implements and handles timestamps and watermarks to support streaming applications with event-time semantics.

3 A TaskManager ensures that each task has at least one incoming and one outgoing buffer and respects additional buffer assignment constraints to avoid deadlocks and maintain smooth communication.

Timestamps

A Flink streaming application with event-time semantics must ensure that every record has a timestamp. The timestamp associates a record with a specific point in time. Although the semantics of the timestamps are up to the application, they often encode the time at which an event happened. When processing a data stream in event-time mode, Flink evaluates time-based operators based on the timestamps of records. For example, a time-based window operator assigns records to windows according to their associated timestamp. Flink encodes timestamps as 16-byte long values and attaches them as metadata to records. Its built-in operators interpret the long value as a Unix timestamp with millisecond precision, i.e., the number of milliseconds since 1970-01-01-00:00:00.000. However, custom operators can have their own interpretation and, for example, adjust the precision to microseconds.

Watermarks

In addition to record timestamps, a Flink event-time application must also provide watermarks. Time-based operators need to know the current logical time of the application in order to make progress. For instance, a time-based window operator needs to know that the end-time of the window has passed in order to emit the result and close it. While processing time operators simply ask the operating system for the current time, event-time operators compute the event time (the logical time as embedded in the record timestamps) from watermarks. In Flink, watermarks are implemented as special records holding a timestamp long value. Watermark records are injected into a stream of regular records with annotated timestamps as the following figure shows.

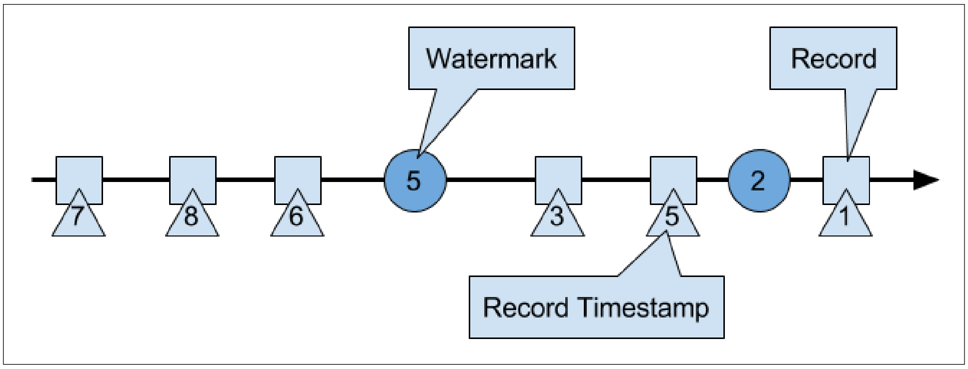


Figure 2-5. A stream with timestamped records and watermarks.

Watermarks must be monotonically increasing to ensure that the logical time computed by operators is progressing and not going backwards. In addition, watermarks are also related to record timestamps. A watermark indicates that all records following the watermark should have a timestamp greater than (i.e., later than) the watermark's timestamp [4]. Records that violate this condition are so-called late records. When an operator advances its logical time to the latest watermark and therefore triggers a computation, late records do not contribute to the result of the computation which consequently will be inaccurate or incomplete. Flink provides mechanisms to properly handle late records. We will discuss these features later in Chapter 6. The requirement of monotonically increasing watermarks and the objective to limit the number of late elements does also mean that the timestamps of the records in a stream should be in general increasing as well. Monotonically increasing timestamps are not required though because watermarks are a mechanism to tolerate records with out-of-order timestamps, such as the records with timestamps 3 and 5 in the above figure.

Extraction and Assignment of Timestamps and Watermarks

Timestamps and watermarks can be interpreted as properties of a data stream. They are usually generated when a stream is ingested by a streaming application. Since record timestamps and watermarks are tightly related with each other, extracting and assigning timestamps and watermarks goes hand in hand in Flink. A Flink application can assign timestamps and watermarks in two ways.

The first option is a data source that immediately attaches timestamps and emits watermarks when ingesting a data stream into the application. This approach is mainly applicable for custom data source functions because timestamp extraction depends on the often custom data type of a source function. Flink's built-in connectors are usually more generic and do not extract and assign timestamps and watermarks (see Chapter X for details on connectors).

The second and more common approach is to use a so-called timestamp and watermark extraction operator. The operator is usually applied immediately after a source function and consists of two user-defined functions[5]. The first function extracts a timestamp from a record. Flink calls the timestamp extraction function for each record and attaches the extracted long value as metadata to the record. The second function generates a watermark and there are two ways to do this:

1. Generation of periodic watermarks: Flink periodically asks the user-defined function for the current watermark timestamp. The polling interval can be configured.
2. Generation of punctuated watermarks: Flink calls the user-defined function for each passing record. The user-defined function may or may not extract a water-

4 In fact this is a bit oversimplified. Watermarks can be used to control result completeness and latency. We will discuss this property of watermarks in Chapter X.

5 The syntax of these functions is discussed in Chapter X.

mark from the record. Punctuated watermarks are useful if the stream contains special records that encode watermark information.

In both cases, Flink injects the watermarks returned by the function into the stream. A timestamp and watermark extractor operator can also be used to override existing timestamps and watermarks, for example timestamps and watermarks generated by a source function. However, it should be noted that this is usually not a good idea and should be done very carefully.

Handling of Timestamps and Watermarks

Once timestamps have been assigned and watermarks have been generated and injected into a stream, subsequent operators can process the records of the stream with event-time semantics. Except for the Process operator[6], all built-in operators of Flink's DataStream API handle timestamps and watermarks internally and do not give developers access to them. None of the built-in operators (except for timestamp extractors and watermark generators which were discussed before) expose APIs to adjust or change timestamps or watermarks.

Instead, the operators take care of maintaining and emitting record timestamps and watermarks. Some operators have to adjust timestamps of existing records or compute timestamps for new records. For instance, a time window operator attaches the end time of a window as timestamp to all records emitted by the window and subsequently forwards the watermarks that triggered the window's computation. By handling record timestamps and watermarks internally, developers of Flink event-time applications only have to provide streams with annotated timestamps and watermarks and Flink's operators will take care of everything else.

Computing Event Time from Watermarks

Flink is a distributed system and executes streaming applications in a data parallel fashion (see Chapter 2) by splitting streams into stream partitions which are concurrently processed. Streams are usually ingested by multiple concurrently running source tasks in order to achieve the required throughput. Consequently, watermarks are generated in parallel as well. However, parallel watermark generators do only observe the record timestamps of their local stream partition such that a single watermark can only define the logical time in its local partition of the stream.

As mentioned before, watermarks are implemented by Flink as special records which are received and emitted by operators. Watermarks emitted by an operator task are

6 The Process operator is invoked per record and gives access to the record's timestamp, the current watermark, and allows to register timers which invoke a callback function once the time is reached. The Process function is later discussed in Chapter X.

broadcasted to all connected tasks of a subsequent operator. For example the watermarks of a stream which is partitioned on a key attribute are broadcasted to each following operator task. A task maintains for each preceding task a channel watermark. When it receives a new watermark from a preceding task, it updates the respective channel watermark if the new watermark is greater (later) than its current timestamp. If the channel watermark was updated, the task updates its task watermark, i.e., its current event time, as the minimum timestamp of all its channel watermarks. Finally, the task broadcasts its new watermark to all succeeding tasks. This watermark propagation algorithm ensures that the event time of each task is monotonically increasing and the correct rate, i.e, that no late events are produced. The following figure illustrates the computation of watermarks.

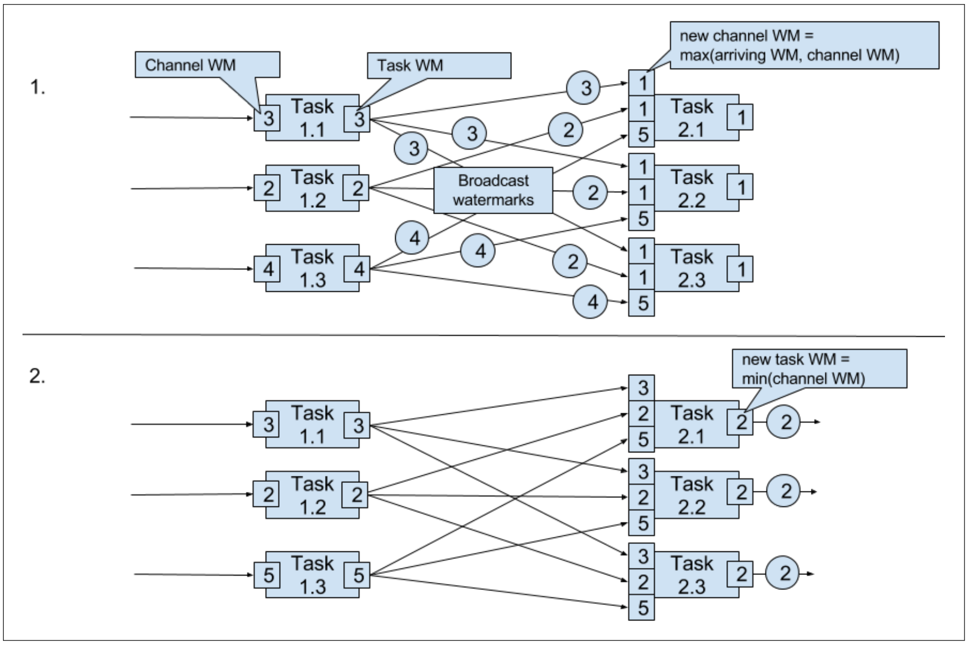


Figure 2-6. Watermark Propagation and Event Time Computation.

Note that the algorithm to compute the task watermark as the minimum of all channel watermarks is also used for operators that consume two (or more) streams such as union or join operators. For such operators it is important that the watermarks of both streams are not too far apart. Otherwise, Flink might need to buffer a lot of data of the stream which is ahead in time.

Fault Tolerance and Failure Recovery

In Chapter 2 we discussed operator state and pointed out that the vast majority of streaming applications is stateful. Most operators require and continuously update some kind of state such as records collected in a window, reading positions of an input source, or custom, application-specific operator state. In Flink, all state - regardless of built-in or custom operators - is treated the same. Each task manages its own state and does not share it with other tasks (belonging to the same or other operators). The state is locally stored on the processing machine, i.e., Flink does not rely on some kind of global mutable state. Since operator state must never be lost, Flink needs to protect it from any kinds of failures which are common in distributed systems such as killed processes, failing machines, and interrupted network connections. For streaming applications with strict latency requirements, the overhead of preparing for failures as well as recovering from them needs to be low.

Flink's approach to recover applications and the state of their operators from failures is based on consistent checkpoints of the complete state of an application. This technique can provide exactly-once result guarantees. Flink implements a sophisticated and lightweight algorithm to take consistent checkpoints by copying local operator state to a remote storage in a distributed and asynchronous fashion. Moreover, Flink can be configured to operate in a highly-available cluster mode without a single point of failure to be able to tolerate JobManager and TaskManager failures.

In the following sections, we discuss how Flink internally manages state. We explain what a consistent checkpoint is and how it is used to recover from failures. We discuss Flink's checkpointing algorithm in detail and finally explain how Flink's highly-available mode works.

State Backends

During regular processing, the tasks of a stateful application need efficient read and write access to their state. Therefore, tasks locally store their state on their TaskManager for fast access. However as in any distributed system, TaskManagers may fail at any point in time such that their storage must be considered as volatile. Therefore, Flink periodically checkpoints the local operator state to a remote and persistent storage.

There are different options how operator state can be locally and remotely stored. For example, TaskManagers can store local state in-memory or on their local disk. While the former approach gives very good performance, it limits the maximum size of the state. On the other hand, writing state to disk is slower but allows for larger state. The remote storage for checkpointing can be a distributed file system or a database system. Flink features a pluggable mechanism called state backend to configure how state is locally and remotely stored. Flink comes with different state backends that serve different needs. We will discuss the different state backends and their pros and cons in more detail in Chapter X.

Recovery from Consistent Checkpoints

Flink periodically takes consistent checkpoints of a running streaming application. A consistent checkpoint of a stateful streaming application is a copy of the state of each of its operators at a point when all operators have processed exactly the same input. This is easier to understand when looking at the steps of a naive algorithm to take a consistent checkpoint from a streaming application.

1. Pause the ingestion of streams at all data sources.
2. Wait for all in-flight data to be completely processed.
3. Copy the state of each task to a remote, persistent storage such as HDFS.
4. Report the storage locations of all tasks' checkpoints to the JobManager.
5. Resume the ingestion of all streams.

Note that Flink does not implement this naive algorithm. We present Flink's more sophisticated checkpointing algorithm after we discussed how Flink uses consistent checkpoints to recover from failures.

In order to protect a streaming application from failures, Flink periodically takes consistent checkpoints from the application. The JobManager holds pointers to the locations where the checkpoints of every task are stored. The following picture shows an application with a single source task that consumes a stream of increasing numbers. The numbers are partitioned into a stream of even and odd numbers and on each stream is a running sum computed. The source task stores the current offset of its input stream as state, the sum tasks persist the current sum value as state. When taking a checkpoint, all tasks write their state into HDFS and report the storage location to the JobManager.

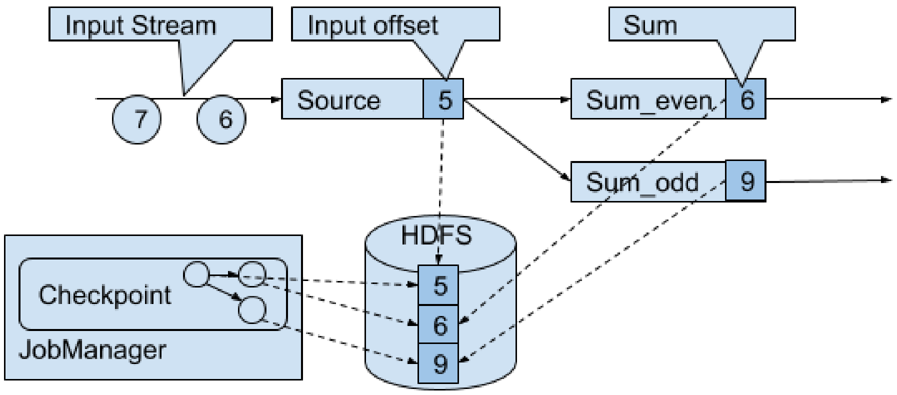


Figure 2-7. Taking a consistent checkpoint from a streaming application.

If at some point one or more tasks of the application fail, Flink recovers from this failure by resetting the applications state to the most recently completed checkpoint as the following figure shows. The recovery process is depicted in the following figure.

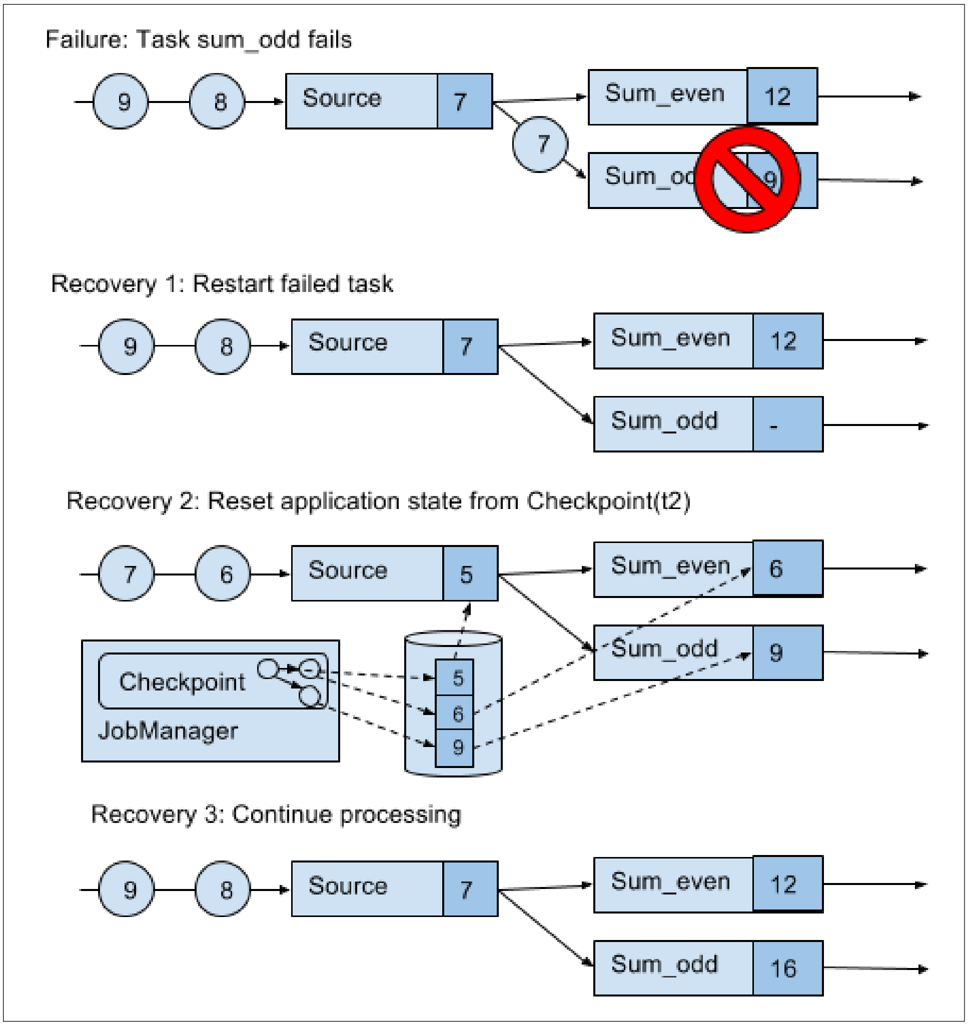

Failure: Task sum_odd fails
9
8
Source
7
Sum_even
12
7
Sum_odd
9
Recovery 1: Restart failed task
9
8
Source
7
Sum_even
12
Sum_odd
-
Recovery 2: Reset application state from Checkpoint(t2)
7
6
Source
5
Sum_even
6
Sum_odd
9
Checkpoint
JobManager
5
6
9
Recovery 3: Continue processing
9
8
Source
7
Sum_even
12
Sum_odd
16

Figure 2-8. Recovery from a task failure using a consistent checkpoint.

The recovery happens in three steps:

1. Bring up the failed task (or tasks).
2. Reset the state of the whole application to the latest checkpoint, i.e., resetting the state of each task. The JobManager provides the location to the most recent checkpoints of task state. Note that, depending on the topology of the application, certain optimizations are possible and not all tasks need to be reset.
3. Resume the processing of all tasks.

This checkpointing and recovery mechanism is able to provide exactly-once consistency for applications, given that all operators checkpoint and restore all of their state and that all data sources of the application checkpoint their reading position in their input stream and are able to continue reading from a position that was reset by a checkpoint. Whether a data source is resettable or not depends on its implementation and the external system or interface from which a stream is consumed. For instance, Apache Kafka supports to read records from a previous offset of the stream. In contrast, a stream consumed from a socket cannot be reset because sockets do not buffer data which has been emitted. Consequently, an application can only be operated under exactly-once failure semantics if it consumes all data from resettable data sources such as Flink's connector for Apache Kafka [7]

When an application is restarted from a checkpoint, its internal state is exactly the same as when the checkpoint was taken. When it restarts, the application consumes and processes all data that was processed between the checkpoint and the failure another time. Although this means that some messages are processed twice (before and after the failure) by Flink operators, the mechanism still achieves exactly-once state semantics because the state of all operators has been reset to a point that had not seen this data yet. We also need to point out that the checkpointing and recovery mechanism does only reset the internal state of a streaming application. Flink is in general not able to invalidate records that have been sent out or to rollback changes done to external systems. Even if the state of an application is guaranteed to have exactly-once semantics, a data sink might emit a record more than once during recovery. Nonetheless, it is also possible to achieve exactly-once output semantics for selected data sinks. The challenge of end-to-end exactly-once applications is discussed in detail in Chapter X.

Taking Consistent Checkpoints without Stopping the Worlds

Flink's recovery mechanism is based on consistent application checkpoints. The naive approach to take a checkpoint from a streaming application, i.e, to pause, checkpoint, and resume the application, suffers from its "stop-the-world" behavior and is not applicable for applications that have even moderate latency requirements. Instead, Flink implements an algorithm which is based on the well-known Chandy-Lamport algorithm for distributed snapshots. The algorithm does not pause the application but allows to continue processing while checkpointing the state of operators. In the following we explain how this algorithm works.

At the core of Flink's checkpointing algorithm is a special type of record which is called checkpoint barrier. Similar to watermarks, checkpoint barriers are injected into

[7] Note that there are other resettable sources. In addition it is possible to implement a source function that buffers records inside operator state to replay them in case of a failure.

the regular data stream and cannot overtake or be passed by other records. A checkpoint barrier logically splits a stream into two parts and the position of the barrier in the stream defines the extent of a checkpoint. All state modifications due to records that precede a barrier are included in the checkpoint while the records following the barrier do not modify the checkpointed state. In order to identify the checkpoint to which a barrier belongs, each barrier carries a checkpoint ID.

We continue to explain the algorithm step-by-step using the example of a simple streaming application. The application consists of two source tasks which both consume a stream of increasing numbers. The output of the source tasks is partitioned into substreams of even and odd numbers. Each substream is processed by a task that keeps the highest observed number as state and forwards all records to a stateless sink. The application is depicted in the following figure.

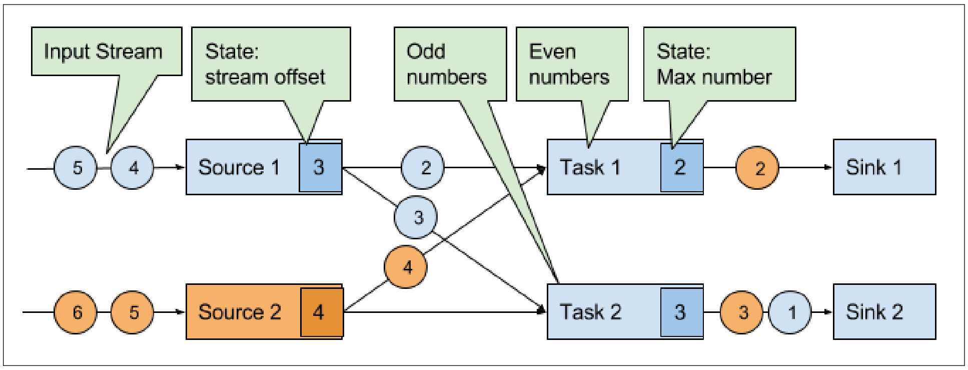


Figure 2-9. An example streaming application with two stateful sources, two stateful tasks, and two stateless sinks.

A checkpoint is initiated by the JobManager by sending a message with a new checkpoint ID to each data source task as the following figure shows.

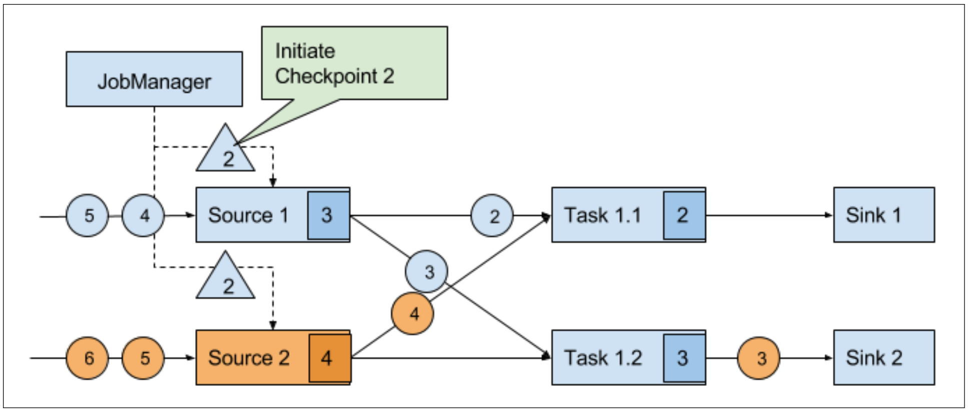


Figure 2-10. The JobManager initiates a checkpoint by sending a message to all sources.

When a data source task receives the message, it pauses its operation, checkpoints its local state to the remote storage defined by the state backend, and emits a checkpoint barrier with the checkpoint ID into its regular output stream. After the barrier is emitted the source continues its regular operations. By injecting the barrier into its output stream, the source function defines the stream position on which the check-point is taken. The following figure shows a streaming application after each source checkpointed its local state and emitted a checkpoint barrier.

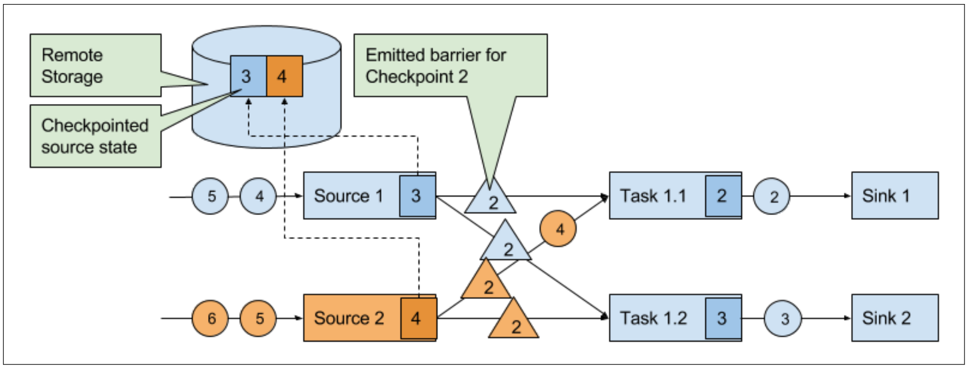
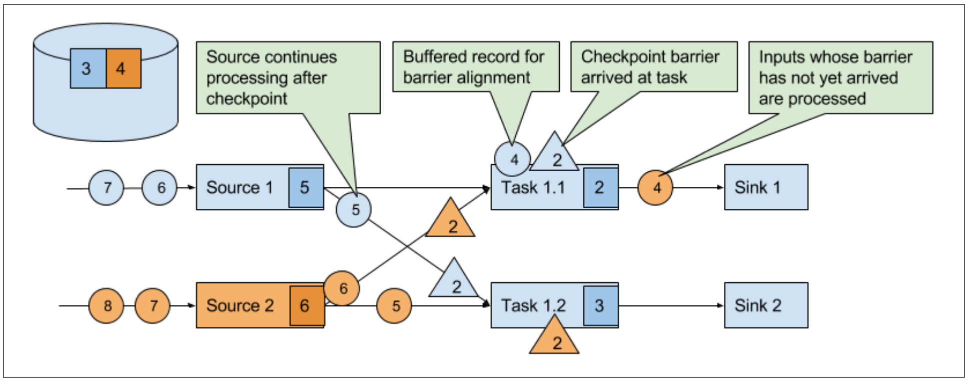


Figure 2-11. Sources checkpoint their state and emit a checkpoint barrier.

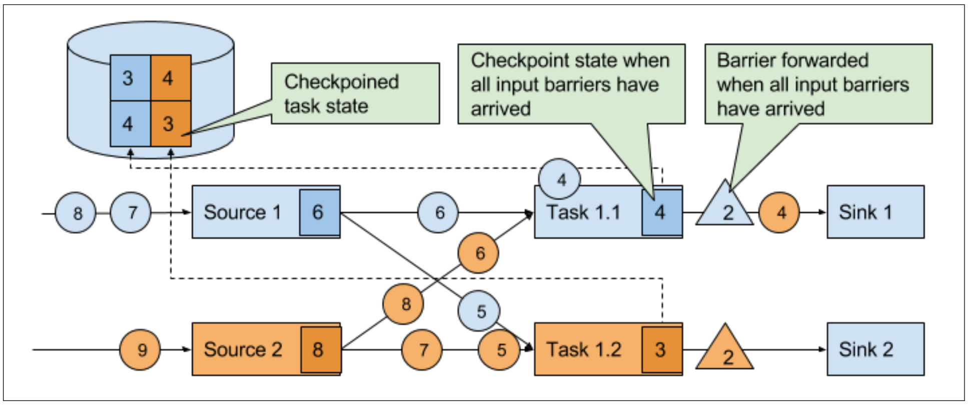
A checkpoint barrier emitted by a source task starts flowing to the next tasks. Similar to watermarks, checkpoint barriers are forwarded to all connected parallel tasks, i.e., they are broadcasted over channels that apply a partitioning strategy for regular records. By forwarding checkpoint barriers to all connected tasks, Flink ensures that each task receives a checkpoint from each of its input streams, i.e., all downstream connected tasks. When a task receives a barrier for a new checkpoint, it waits for the arrival of all barriers for the checkpoint. While it is waiting, it continues processing records from input streams that did not contain a barrier yet. Records which arrive via channels that forwarded a barrier already must not be processed and need to be buffered. The process of waiting for all barriers to arrive is called barrier alignment.



3
4
Source continues processing after checkpoint
Buffered record for barrier alignment
Checkpoint barrier arrived at task
Inputs whose barrier has not yet arrived are processed
7
6
Source 1
5
5
4
2
Task 1.1
2
4
Sink 1
2
2
8
7
Source 2
6
6
5
Task 1.2
3
2
Sink 2

Figure 2-12. Tasks wait until they received a barrier on each input stream. Meanwhile they buffer records of input streams for which a barrier did arrived. All other records are regularly processed.

Once all barriers have arrived, the task checkpoints its operator state to the remote storage configured by the state backend and forwards a checkpoint barrier to each of its downstream connected tasks.



3
4
4
3
Checkpoined task state
Checkpoint state when all input barriers have arrived
Barrier forwarded when all input barriers have arrived
8
7
Source 1
6
6
4
Task 1.1
4
2
4
Sink 1
6
8
5
9
Source 2
8
7
5
Task 1.2
3
2
Sink 2

Figure 2-13. Tasks checkpoint their state once all barriers have been received. Subsequently they forward the checkpoint barrier.

After the checkpoint barrier has been emitted, the task starts to process the buffered records and subsequently continues to process its input streams.

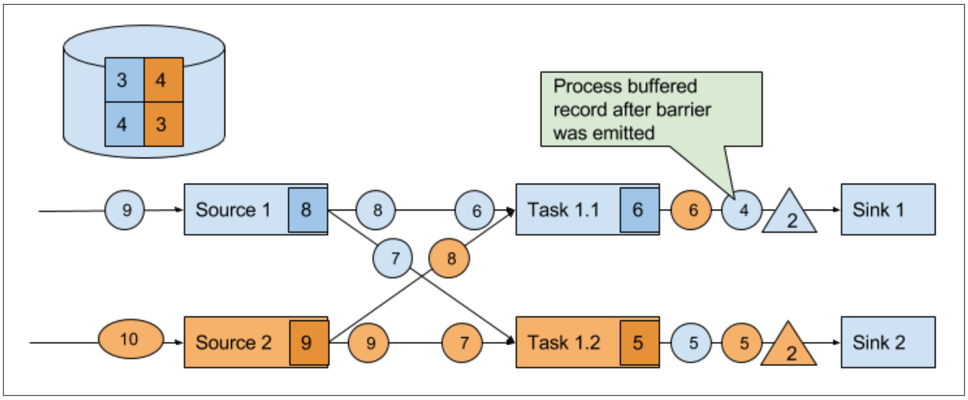


Figure 2-14. Tasks continue their regular processing after the checkpoint barrier was forwarded.

Eventually, the checkpoint barriers arrive at a sink task. When a sink task receives a barrier, it performs a barrier alignment, checkpoints its own state, and acknowledges the reception of the barrier to the JobManager. The JobManager records the checkpoint of an application as completed once it receives a barrier acknowledgement from each sink task of the application. Now, the checkpoint can be used to recover the application from a failure as described before.

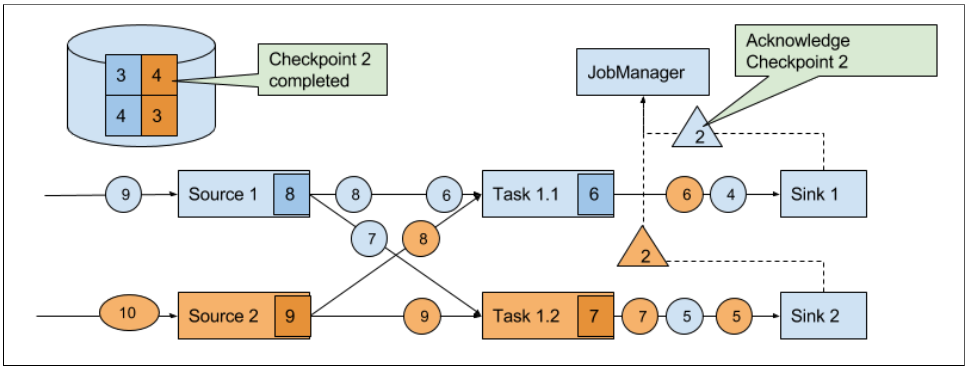


Figure 2-15. Sinks acknowledge the reception of a checkpoint barriers to the JobManager. A checkpoint is complete when all sinks acknowledge the barriers of a checkpoint.

The discussed algorithm produces consistent distributed checkpoints from streaming applications without stopping the whole application. However, it has two properties that can increase the latency of an application. Flink's implementation features tweaks that improve the performance of the application under certain conditions.

The first spot is the process of checkpointing the state of a task. During this step, a task is blocked and its input is buffered. Since operator state can become quite large and checkpointing means sending the data over the network to a remote storage system, taking a checkpoint can take up to several seconds. To overcome this issue, Flink can perform asynchronous checkpoints under certain conditions [Footnote: This feature depends on the type of state and the capabilities of the state backend]. A task creates an asynchronous checkpoint by creating an immutable local copy of its state. Once the copy is complete, the task continues processing and asynchronously writes the copy to the remote storage. The overall checkpoint of an application is then completed when all barriers have arrived at sinks and all copy processes have been finished.

Another reason for increased latency can result from the record buffering during the barrier alignment step. For applications that require consistently very low latency and can tolerate at-least-once state guarantees, Flink can be configured to process all arriving records during buffer alignment instead of buffering those for which the barrier has already arrived. Once all barriers for a checkpoint have arrived, the operator checkpoints the state, which might now also include modifications from records that would usually belong to the next checkpoint. In case of a failure, these records will be processed again which means that the checkpoint provides at-least-once instead of exactly-once guarantees.

Highly-Available Flink Clusters

Flink's checkpointing and recovery techniques prevent data loss in case of a task or worker failure. However in order to be able to restart an application, Flink requires a sufficient amount of processing slots. Given a Flink setup with four TaskManagers that provide two slots each, a streaming application can be executed with a maximum parallelism of eight. If one of the TaskManagers fails, the number of available slots is reduced to six which is not enough to recover a streaming application with a parallelism of eight. For stand-alone cluster setups, this issue can be solved by having standby TaskManagers which can take over the work of failed workers. In cluster setups with resource managers such as YARN and Apache Mesos, new TaskManager processes can be automatically started.

A more challenging problem than TaskManager failures are JobManager failures. The JobManager controls the execution of a streaming application and keeps metadata about the execution such as the JobGraph and handles to completed checkpoints. A streaming application cannot continue processing if the associated JobManager disappears. By default, the JobManager is a single-point-of-failure in Flink. To overcome this problem, Flink features a high-availability mode to ensure that another JobManager can take over the responsibilities of a failed master.

Flink's high-availability mode is based on Apache ZooKeeper, a system for distributed services that require coordination and consensus. Flink uses ZooKeeper for leader election and as an available and durable data store. When operating in high-availability mode, a JobManager writes the JobGraph and all required metadata such as the application's JAR file into a remote storage system, which is configured by the state backend. In addition, the JobManager writes a pointer to the storage location into ZooKeeper's data store. During execution of an application, the JobManager receives the state handles (storage locations) of the individual task checkpoints. Upon the completion of a checkpoint, i.e., when all tasks have successfully persisted their state into the remote storage, the JobManager writes the state handles to the remote storage and a pointer to this location to ZooKeeper. Hence, all data that is required to recover from a JobManager failure is stored in the remote storage and ZooKeeper holds pointers to the storage locations. The following figure illustrates this setup.

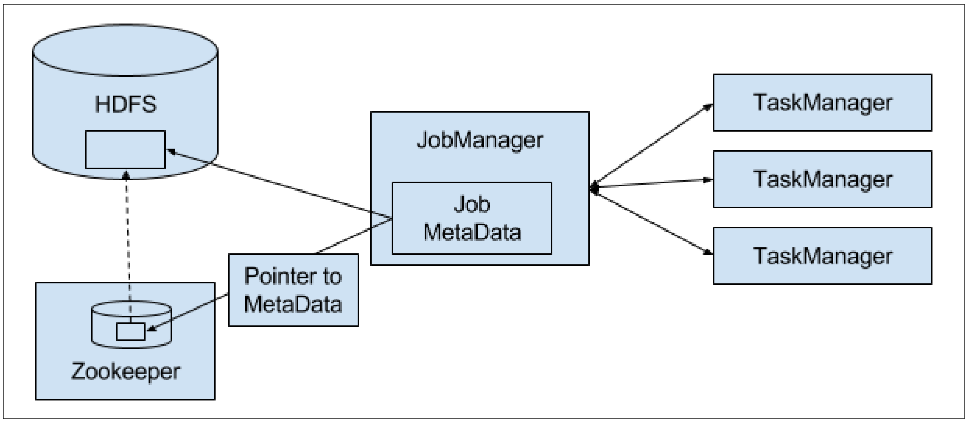


Figure 2-16. Highly-available JobManager setup with Apache ZooKeeper.

When a JobManager fails, all respective tasks are automatically cancelled. A new Job-Manager that takes over the work of the failed master performs the following steps

1. Connect to all available TaskManagers.
2. Request the storage locations from ZooKeeper to fetch the JobGraph, the JAR file, and all state handles of the last checkpoint of all running applications from the remote storage.
3. Restart all applications and reset the state of all their tasks to the last completed checkpoints, using the retrieved JobGraphs and task state handles.

A highly-available stand-alone cluster setup requires at least two JobManagers, one active and one or more stand-by masters. Flink uses ZooKeeper to elect the active JobManager. If Flink runs on a resource manager, such as YARN or Apache Mesos, stand-by masters are not required because a new JobManager is automatically started. We will discuss the configuration of highly available Flink setups later in Chapter X.

Summary

In this chapter we have discussed Flink's high-level architecture and the internals of its networking stack, event-time processing mode, and failure recovery mechanism. Knowledge about these internals can be helpful when designing advanced streaming applications, setting up and configuring clusters, and operating streaming applications as well as reasoning about their performance.