MANNING

Flink IN ACTION

MEAP

Sameer Wadkar
Hari Rajaram

MEAP Edition
Manning Early Access Program
Flink in Action
Version 2

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP for *Flink in Action*. We are excited to deliver this book as large-scale stream processing using Flink and Google Data Flow is fast gaining in popularity. Stream processing is much more than just processing records one at a time as they arrive. True stream processing needs support for concepts such as event time processing to ensure stream processing systems are just as accurate as the batch processing system. There is a need for one system the performs both stream and batch processing. Apache Flink is that system.

As we started exploring Apache Flink, we discovered the subtle challenges that are inherent in stream processing. These challenges are intrinsic to how stream processing is performed. Unlike batch processing, where all data is available when processing begins, stream processing must be able to handle incomplete data, late arrivals, and out-of-order arrivals—without compromising performance or accuracy—and be resilient to failure. We tackle all these challenges in this book.

Writing this book has been a challenge, partly because the technology is changing rapidly as we write and partly because we want to make this complex topic of streaming easy to understand in the context of everyday use cases. We believe that eventually streaming systems will become the norm, because the real world operates in the streaming mode. Real-world events occur and are captured continuously in transaction systems. The reporting systems that aggregate these transactions into reports operate in batch-processing mode due to technology limitations. These limitations are now being addressed by systems such as Apache Flink. We hope this book helps you develop a strong foundation in the concepts and the challenges of implementing streaming systems capable of handling high-velocity and high-volume streaming data.

Please be sure to post any questions, comments, or suggestions you have about the book in the Author Online forum. Your feedback is essential in developing the best book possible.

— Sameer B. Wadkar and Hari Rajaram

brief contents

1

Introducing Apache Flink

This chapter covers

- Why stream processing is important
- What is Apache Flink
- Apache Flink in the context of a real world example

This book is about handling streaming data with Apache Flink. Every business is composed of a series of events. Imagine a large retail store or public news site that is serving customers all over the world; events are constantly being generated. What distinguishes a streaming system from a batch system is that the event stream is unbounded or infinite from a system perspective. Decision-makers need to analyze these streaming events together to make business decisions. For example:

- A retail store chain is constantly selling products in various locations. People making decisions need to know how the various products are selling. Most current systems do this via nightly extract, transform, and load (ETL) processing, which is common in enterprise environments, requires decision makers to wait an entire day before reports become available. Ideally these decision makers would like to be able to inquire in near real-time the performance of sales across the stores and regions.
- A popular news website is constantly serving user requests. Each request/response can be considered an event. The stream of events need to be analyzed in near real-time to understand how the news articles are performing with respect to page-views and to determine which advertisements should be displayed to the readers as they are browsing the website.
- Near real-time systems are especially valuable in fraud detection systems. Determining that a credit card transaction was a fraud within moments of performing it is crucial in

preventing as well as minimizing the cost of fraud.

In this book we will discuss the main challenges of stream processing at scale. We will discuss the features of Flink that provide solutions to these challenges.

If you are reading this book, it's likely that you have prior Big Data experience working with Apache Hadoop or Apache Spark. Both are used predominantly in the batch-processing domain, although Spark has a streaming module. We will draw comparisons between these systems and Flink when we describe key features of streaming applications.

Understanding the basics of streaming systems

The following articles by Tyler Akidau, one of the key members of the Google Cloud DataFlow team, provide an excellent introduction to capabilities required of a stream processing system:

- **World beyond batch: Streaming 101 -** https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101
- **World beyond batch: Streaming 102 -** https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102

1.1 Stream event processing

First let us understand what is an event and its relationship to streams. From a business perspective an event is business transaction:

- A customer buying a product in a store (online or physical).
- A mobile device emitting the current geographic location.
- Clicking a link on news website.
- Buying or selling stocks via an online brokerage site.
- Every action during online gaming.
- Messaging on social media.

Typically, the information in an event can be broadly classified into the following four categories:

- When – The time the event occurred
- Where – The location associated with an event
- Who – The agent (user or machine) associated with the event
- What – Business information captured by the event

Examples of event information are:

- For a buy event in a store this information is, the transaction time, the store identifier which identifies the location (for online purchases the IP address of the customer identifies the same information, customer, the product identifier, number of items purchased, cost per unit and the transaction id associated with all the products purchased as part of this transaction.
- If a user is accessing a News website and clicks on a link to read a news item, an event

can be considered to be a combination of the following pieces of information: the time the response was generated and sent to the reader, duration of page access, IP address of the reader which identifies the location, the page URL, topic and keywords associated with the page.

Figure 1.1 shows the information content of an event processing system such as transaction event associated with buying an item in a store or information content of the clickstream event associated with the user of a news website clicking and reading a news item.

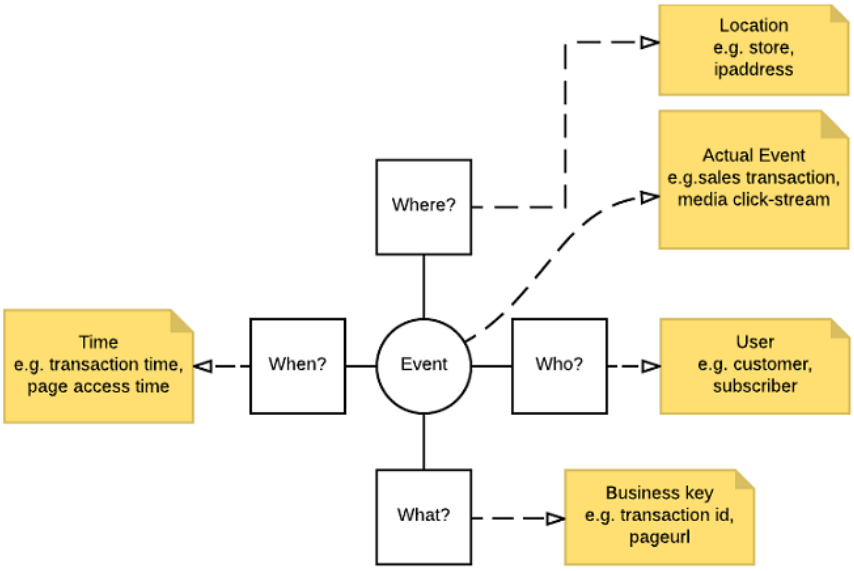

Figure 1.1. The content of a product purchase event by a customer in a retail store along the dimensions of when, where, who and what.

These events need to be analyzed and presented as reports to decision makers. Consider a typical use case of generating aggregate information. For example:

- What were the sales of a product at various intervals (every 5 minutes, 15 minutes, 30 minutes, 1 hour, 24 hours, etc.)?
- Did the sales for a product increase in the day following the introduction of a promotion scheme?
- How many hits did the page generate in the first 1 hour of being published?
- What are the key words associated with the user activity on our news website in the last 5 seconds? Which ads do we show to this user based on this information?

The most common method of implementing such solutions is to collect the daily transactions in a staging database and perform aggregations and other analytics operations in a batch processing mode. The output of this batch processing system is then collected in a Data

Warehouse and consumed by business users via Business Intelligence applications to make decisions. The problem with this mode of operation is, Decision makers have to wait for an entire day to receive information. While using batch processing for generating analytics may be acceptable for certain types of use cases (Retail store analytics), it will produce sub-optimal results when used for use cases like selecting advertisements to display to an online user.

Streaming systems like Flink were designed to solve the delayed-information delivery problem. Flink can deliver the aggregation results or event alerts at various levels of granularity almost as soon as the information is available. But how soon is "almost as soon?" The answer is complex as we will discuss this in detail soon. We will see how Flink allows you to make controlled trade-offs between latency of information availability and accuracy. Latency of information availability is defined by the how fast the information is presented to the user after the event occurs in the real world. Accuracy is a measure of how accurate the information is. We will soon see how Flink allows you to trade off speed of information delivery with accuracy of the delivered information. In other words, you could get the information sooner if you are willing to tolerate approximate aggregation results or an occasional wrong alert. Or you can wait a little longer to get the most accurate information. Or you can implement a system that will provide approximate results very close to real-time, which will be eventually corrected.

Figure 1.2 shows the difference batch and stream processing.

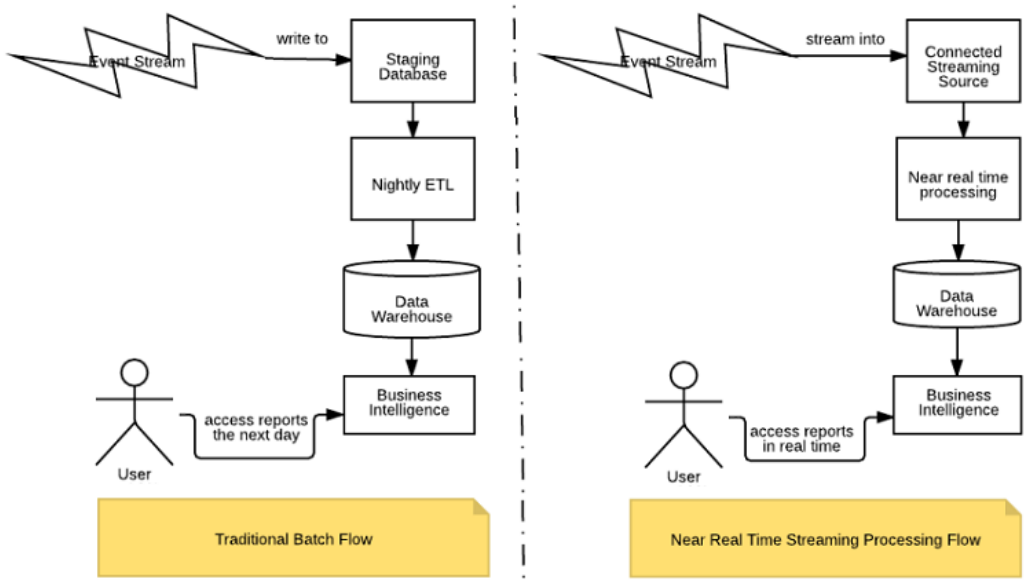

Figure 1.2. Comparing batch workflow with streaming workflow.

In an ETL batch-processing scenario, the transactions databases collect event data during the day, which is bulk loaded to a staging database at night. A large ETL process then executes to produce aggregations on this data and store these results to the data warehouse. The Business Intelligence applications then present the reports based on these aggregates to the business users the following day. Users have to wait an entire day to get insights into business operations.

Streaming systems operate differently. The transactions database will stream the logs in near real-time which are processed by a streaming system which will produce aggregations within intervals. The most intuitive interval is time. For example, calculate aggregates like total sales by region for every 5-minute interval. We will soon see that other types of intervals are also possible in Flink. These aggregates will be stored in the Data Warehouse which provide a more real-time insight into business operations to the business users. The intervals should not be too small (ex. 5-seconds) because Data Warehouses are typically designed for optimal reads (lots of indexes) which will slow down the writing of the granular aggregates and cause the overall system to have bottlenecks.

Note that is not possible to calculate these streaming aggregates in a batch system like Hadoop as you would need to collect transaction data which occurred in the required interval in files and process these files as a batch job. Hadoop jobs have a large (a few seconds to a couple of minutes) of startup overhead. Also it is difficult to know for sure when all the events which occurred in a given interval have arrived. Flink supports what is known as Event Time Processing which guarantees this. We will see examples of this soon.

Spark can support processing of these mini-batches in a Spark Streaming. However, Spark's current support for Event Time Processing is limited. It will probably evolve. But as of now Spark does not support true Event Time Processing.

Additionally, when treating a batch system like Spark to process streaming in mini-batches you can only perform aggregates in time intervals which are multiples of the minimum interval for a batch. And we cannot make it too small as it will cause write bottlenecks on a Data Warehouse which is optimized for read and is slower to write data to when the indexes are active. Most ETL processes disable indexes during a bulk load at night after the ETL process has finished aggregating the data.

1.2 How Flink works

In this section we will delve deeper into how Flink works in the context of a lifecycle of events. We will trace the event lifecycle as it is generated in the real world, arrives into Flink, is processed with other events and finally written out to an external system or storage.

Figure 1.3 shows the reference architecture for a typical Flink implementation for handling a streaming data.

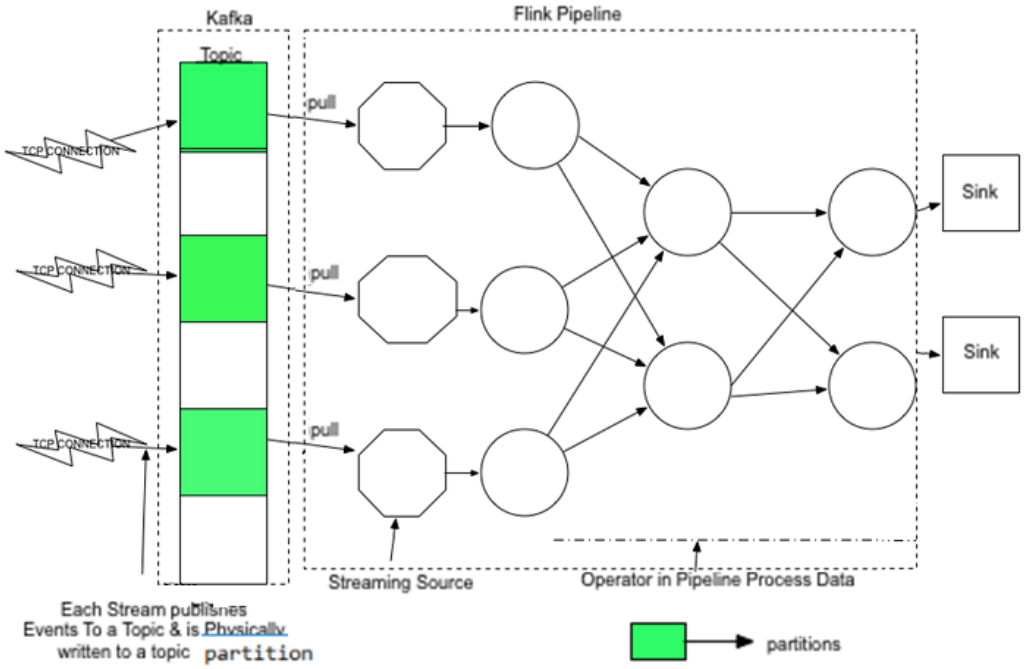

Figure 1.3. Flink Reference Architecture-Events are persisted in a durable store (Kafka), read using a streaming source component, processed in a Flink Pipeline composed of a graph of operators streaming events from upstream to downstream. Results are finally persisted in an external data store or data stream called a sink.

The main steps in the stream processing are:

1. Events are streamed over a reliable TCP connection from a real world event source.
2. The listener on these TCP connections will persist these events into a durable, partitioned, failure resilient store which maintains the temporal order (there are exceptions, which will be described in chapter 5) of the event arrival. The product most commonly used is Apache Kafka which is a durable and distributed publish/subscribe messaging system capable of handling event velocity of over 100K messages per second. The events are published to a Kafka Topic. A Kafka topic consists of multiple partitions and each partition can run on a separate machine which allows events published to a Kafka topic to be processed in parallel. The events are now ready to be processed by the Flink application which we will refer to as the Flink Pipeline.
3. The streaming source is the first component of the Flink Pipeline. Each streaming source instance will pull data from the durable event store partition in the order it was received and pass it to a chain of operators for processing.
4. After the event data is processed the last step in the Flink Pipeline will write the data to an external data store or stream which is known as a sink.

1.2.1 The need for a durable event store like Kafka

It is natural to wonder why a system like Apache Kafka is needed. Why can the streaming source in the Flink Pipeline not connect directly to the network sources which stream the events? This is certainly possible. However, by doing so you have given up your ability to recover from failure. A distributed system like Flink has many tasks (each operator instance in the pipeline runs as a task) running on multiple nodes. It is possible for nodes to fail. Flink will restart the tasks on a healthy node. But what about the data which was delivered to the task for processing. If the data is processed in flight, it is cannot be resent.

Apache Kafka is a distributed, durable publish-subscribe messaging system that is resilient to failure. Events arriving into Kafka are published to a Topic in Kafka. The actual event data is written to a partition of a topic. Partitions of a topic are usually maintained on separate Kafka brokers (servers). Partitions allow multiple writers to write to the same topic as well as multiple consumers read from the same topic in parallel. Each write to a partition is persisted to a disk and replicated on disks on the cluster where Kafka is deployed. If the event data needs to be re-read by the Flink pipeline to re-execute a failed operator, the Streaming Source can simply rewind the pointer to the Kafka partition and re-read the data and re-send it down the pipeline for re-processing. In a nutshell, persisting event data before processing, allows Flink to be recover from failure as well as support re-processing of data. We will revisit this topic again when we discuss checkpoints and snapshots.

1.2.2 Flink application pipeline

The Flink Application pipeline is depicted in figure 1.3 (reproduced again for convenience).

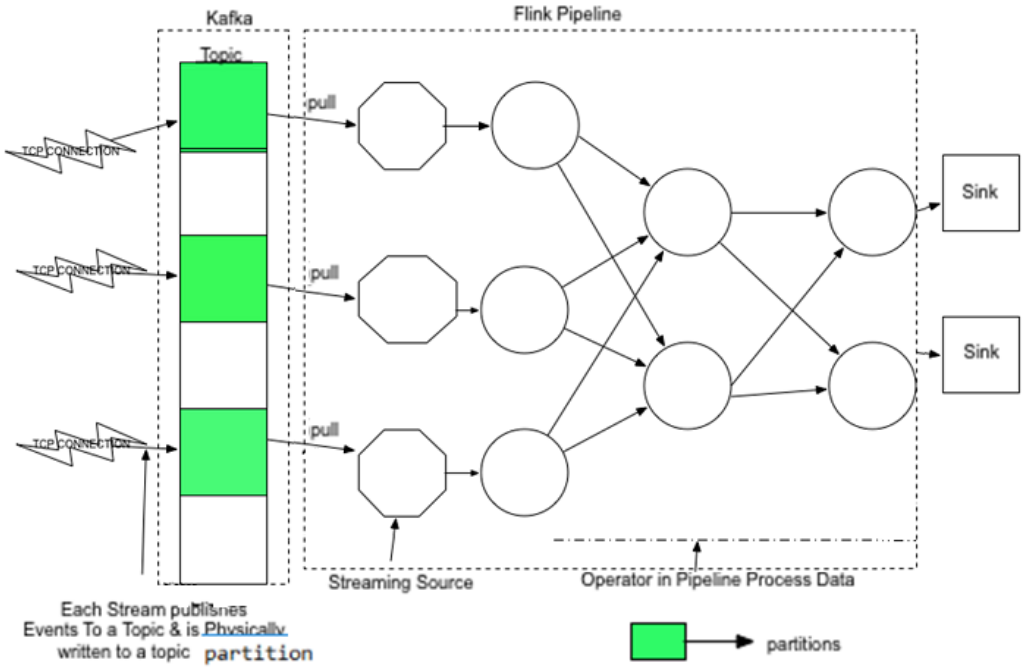

Figure 1.3. A sample Flink application pipeline. The Streaming Source operator receives data from an external data stream. The next layer is a Map type operator which processes events one at a time, followed by a configuration of Map or Reduce type operators which finally write to a Sink operator which pipe data into an external data store.

This section assumes a basic knowledge of MapReduce.

Basics of MapReduce

We will assume that the reader has a basic knowledge of how MapReduce works. There are many sources online that describe MapReduce. The following StackOverflow link does a reasonable job of explaining what it is - http://stackoverflow.com/questions/28982/please-explain-mapreduce-simply.

One of the authors of this book has also published a book on Hadoop, Pro Apache Hadoop (https://www.amazon.com/Pro-Apache-Hadoop-Jason-Venner/dp/1430248637). The first three chapters of that book cover the basics of YARN (Hadoop 2) and MapReduce. The first seven chapters cover MapReduce in detail.

There are four distinct layers in a Flink Application Pipeline:

- The first is the Streaming Source component. This is a custom component which logically speaking, reads data from each event stream which can be a TCP connection. Physically, it may read from a Kafka partition or another data source to which the event

stream is persisted.

- A Streaming Source component is directly connected to a Mapper-type operator. Each event is streamed into a Mapper type operator instance. The Map operator will process events one at a time and stream them out to the next layer in the pipeline.
- The subsequent layers can consist of either Map type or Reduce type operators configured in the form of a processing pipeline. Map type operators process an event stream one event at a time and produce a Key-Value pair. The Reduce type component receive input as a list of values for a given key and produce one to many Key-Value pairs.
- The Reduce type components can be connected to other Map or Reduce type operators. In this respect Flink is similar to Spark which allows you to configure a DAG (Directed Acyclic Graph) of tasks (Mapper or Reducer component) through which the data flows to achieve your computational goal. Basic MapReduce only allows you to have a job with Mapper and Reducer configuration. If you need a more complex configuration you have the create a chain of MapReduce jobs. The output of the upstream job(s) becomes the input to the downstream job(s). This leads to excessive IO which has serious performance implications. The Flink (and Spark) style of creating complex processing pipeline is more performant.
- The operators in the final layer of the processing pipeline (leaf nodes of the processing pipeline) are connected to a sink type operator. The sink operator receives Key-Value pairs and writes them to an external data store that can be persistent storage like a NoSQL store, HDFS, local file system or event another data stream.

The key question now is when a Map type operator streams data to a Reduce type operator, it is shuffled by the Key from the Key-Value pairs emitted by the Map operators. In batch processing the Reduce operator instance will reduce all the values for the given key. But in case of stream processing, the stream is potentially infinite and there is no concept of all values for a key. Still a Reduce operator needs to operate on list of values for a given key. What values make up this list? This brings us to the concept of windows in Flink. Each such list is known as a Window.

1.3 Basics of windowing in Flink

In stream processing, a window is a dynamic runtime component that contains multiple events that arrive in a stream. An event can be assigned to a single window or multiple windows. Various types of operations such as aggregations are applied on elements (events) in a window. The concept of windowing is crucial to stream processing. Without windows it is impossible to do data processing on streams because most data processing use cases are required to perform aggregations and aggregations cannot be performed on a single data element. Aggregations by nature can only be performed on multiple data elements.

The lifecycle of each Key-Value pair from arrival to a Reduce operator instance to being processed and emitted to the next stage in the pipeline is depicted in figure 1.4.

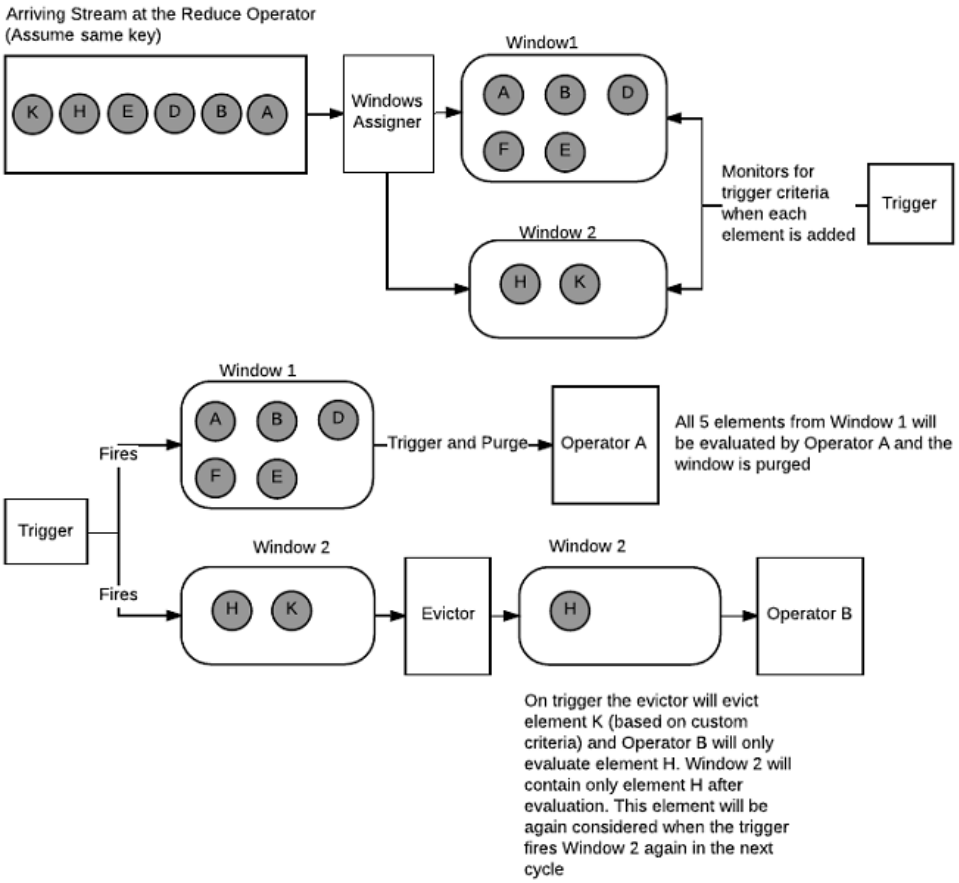

Figure 1.4. How elements arriving for a given key at the Reduce type operator instance are assigned to windows and finally evaluated using the Windows Assigner, Trigger and the optional Evictor components.

The key components participating in this lifecycle are as follows:

- A Windows Assigner and Trigger pair of components
- An optional Evictor component
- The Reduce side operator (denoted as Operator A and Operator B in the figure) instances.

Each of the components in the figure 1.4 work as a system. The order described in the figure is an indication of the order in which events are processed and how the individual components are chained to one another. The key steps in the processing are as follows:

- When a data element arrives at the Reduce operator instance shuffled by key, the Windows Assigner component will assign it to one or more windows. Assignment to windows may be based on time (all events from 12:00:00 to 12:00:05), count (all events are assigned to the same window called a Global window), session (a window per session where a session is defined as a period of activity followed by a period of in-activity) or custom criteria.
- The Trigger component is responsible for evaluating if a window is ready for processing. Logically, each pane gets its own Trigger component. The Trigger component checks if the window is ready for evaluation each time an element is assigned to it or when a timer assigned to the window expires.
- When the Trigger component decides that the elements of the window should be evaluated (called the firing of the window) it takes all the elements in the window and sends it to the Reduce type instance for evaluation. The Reduce operator instance is invoked for the given key and list of values for the key assigned to the window just fired. The Window can then be purged which reclaims the memory. This type of processing typically occurs in time based windows. For example, windows where all events between 12:00:00 or 12:00:05 are processed together.
- The Window can be evaluated by an optional evictor component before it is sent for processing by the Reduce type operator instance. The evictor will iterate through all the elements of the window, possibly remove some and only send the remaining elements for processing by the operator instance. In such cases the Window is not purged and the remaining elements in the window are available for processing again the next time the window is triggered. We will discuss the role of Evictor component when we discuss count-based windows in Chapter 4.

In the next section we will discuss two types of windows in detail:

- Time based windows
- Count based windows

We will also describe how each of these types of windows can be further subdivided into tumbling and sliding windows.

We will defer the discussion on session and custom windows to advanced section of the book.

1.4 Time-based windows

In this section we will look at time-based windows. In such windows, events are allocated to windows based on time. But which time? There are three types of times associated with an event:

- **Event time** – This is the time the event occurs in the real world. It is one of the attributes of the event data. It is assigned by the source of the event in the real world.
- **Processing time** – This is the time the event arrives at the operator for evaluation.

For a given event the processing time assigned to it changes as it arrives at each operator. Processing time is a dynamic attribute of the event based on when it arrives for processing at an operator instance.

- **Ingestion time** – This is the time the event arrives inside the Flink system. It is similar to event time in that, it is assigned once and never changes. It is different from event time since it is assigned by the Flink system and not by the source of the event.

Time based windows are used to calculate aggregations based on time. Some applications of time based windows are:

- Streaming Extract Transform Load (ETL) operation which generates sales by product line per store for a large retail chain for every 5-minute interval. Decision makers will be able to access reports with at most 5-minute delay on how various product lines are performing.
- Calculate aggregations like average temperature every 5 second intervals in an Internet of Things (IoT) application.
- Page visits per user in one minute intervals to decide what advertisements to show a user.

To understand these window types, we will consider the lifecycle of three events in an event stream being processed by a Flink processing pipeline. We also focus our attention on two operators (called Operator A and Operator B) in the pipeline. Table 1.1 shows the various times associated with the events.

Table 1.1 Times of three events in a stream processed by a pipeline

Event Id	Event-Time	Ingestion-Time	Arrival time at Operator A	Arrival time at operator B
101	12:00:01	12:00:07	12:00:09	12:00:17
102	12:00:06	12:00:07	12:00:09	12:00:17
103	12:00:06	12:00:07	12:00:11	12:00:17

1.4.1 Event time-based windows

When windows are based on event time, they are assigned to a time window based on when they occurred in the real world. In this chapter we will assume that the task of assigning this timestamp to the event rests with the streaming source operator responsible for reading in event stream into the Flink processing pipeline. There are other options to assign the timestamp to the event which will be described in chapter 5.

The streaming source operator is custom developed to extract the event timestamp from the event payload. It assigns this timestamp to the event so that the Flink framework can extract it from the event at any stage of processing in the pipeline. The Windows Assigner will extract this timestamp from the event and use it to assign the event to the correct window. Note that the event will be assigned to a pane of the window based on the key.

But how is this window triggered? In order to trigger this window, the Trigger needs to determine that all the events for its window have arrived. Or in other words the Trigger needs to determine that no more events will arrive for its window. If an operator is receiving events from multiple sources, the trigger needs to determine that all events for the time period have arrived.

For the purpose of this discussion we will make several simplifying assumptions which will be relaxed in chapter 5:

- Timestamps have a minimum resolution of 1 second. This makes the explanations easier to follow as compared to using a resolution of 1 millisecond.
- Events arrive at a streaming source operator in the increasing order of their timestamp.
- Events are never late.
- A time window will contain events which include the starting timestamp but exclude the ending timestamp of the interval. For example, the window 12:00:00-12:00:05 will include events with timestamp 12:00:00 but the events with timestamp 12:00:05 will go to the window 12:00:05-12:00:10.

The information that all events for a time period have arrived is contained in special lightweight events known as Watermarks. A watermark is a timestamp *based on* the last seen event timestamp by a given source operator. Each source operator produces its own watermarks. Assume that each source operator will generate a watermark event which one second less than that current timestamp it observes. We will assume that the source operator will generate a watermark event for every event processed. This has an overhead and chapter 5 will describe how to avoid it. For the purpose of this chapter we will assume we get one watermark event per event processed.

The source operator will generate a watermark event which is equal to a timestamp which is one second less than the timestamp of the event it processed. For example, if the event had a timestamp 12:00:01, it will generate a watermark with timestamp 12:00:00. If the next event also has timestamp 12:00:01, the next watermark will have timestamp 12:00:00 again. If this is followed by an event with timestamp 12:00:02, the watermark will advance to 12:00:01. Why did we subtract one second from the current timestamp? Because watermarks make an assertion that no more events with the timestamp represented by the watermark will arrive after the watermark. If the source sees an event for 12:00:01 it means (given our assumption) that all events for 12:00:00 have arrived. But as described in our example, events with 12:00:01 can still arrive as there may be multiple events with that timestamp. We cannot produce a watermark with timestamp 12:00:01 yet. We can only do that when the source sees an event for timestamp 12:00:02. Figure 1.5 demonstrates how this simplified watermark generation mechanism works.

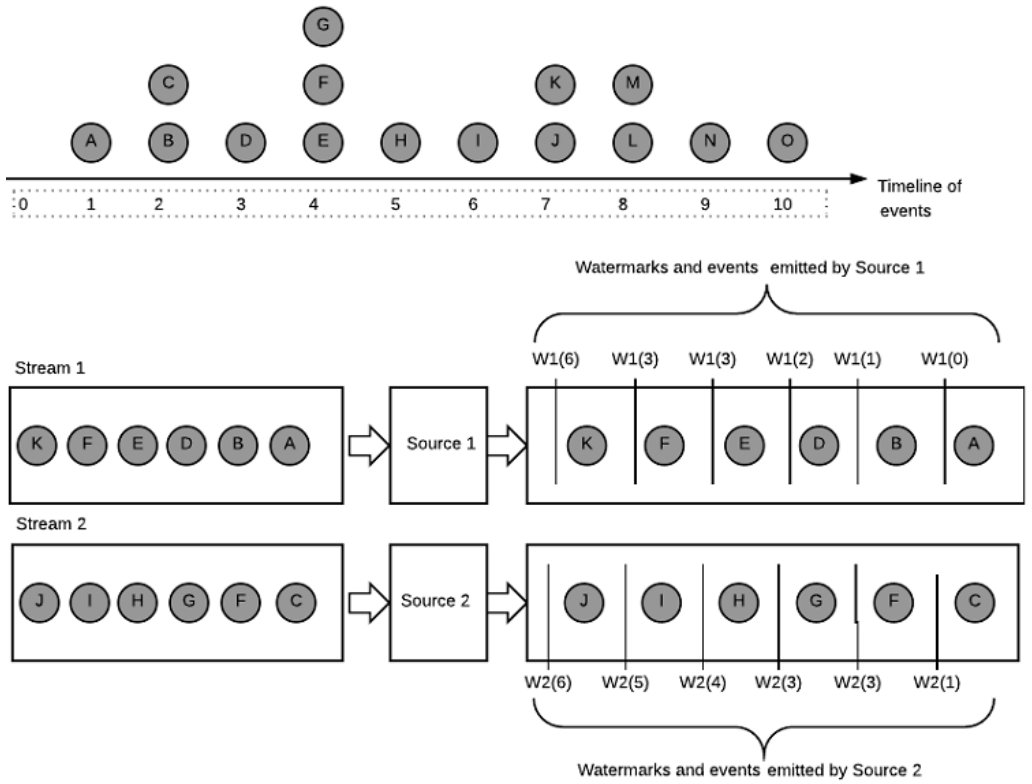

Figure 1.5. A simplistic watermark generation mechanism. Each source generates a watermark event when it observes and event. It emits the event and a watermark with a timestamp one second less than the event timestamp.

To recap the key points about the watermark event at each source are:

- You control watermark generation with custom code. We will describe all options to do that in chapter 5.
- Watermarks only move forward with respect to time. If an event arrives slightly out of order (chapter 5) you do not move the watermark back. Instead you leave it at its current level. In figure 1.5 we see how events with the same timestamp arrive one after the other. The watermark stays at the same level for each subsequent event of the same timestamp.

These watermark events from each source will flow through the Flink application pipeline into all operators. An operator knows how many streams are sending data to it. It receives watermarks from all these streams. Assume that we have defined time windows with length of five (5) second. When an operator is receiving data from two separate streams and it receives

the watermark event with timestamp 12:00:05 (or higher) from both streams, the trigger attached to the window, 12:00:00-12:00:05, knows based on this information that no more events will arrive at this the window and it can be triggered. The trigger needs to make evaluation each time a watermark event arrives at the window. A Trigger will only fire a window if all sources indicate that time has progressed to cover the window interval (Watermark arrival of 12:00:04 from both sources for the time window 12:00:00-12:00:05 to fire). In our example if Stream 1 has sent watermark for timestamp 12:00:04 but Stream 2 is only at 12:00:03, it means that the Stream 2 can still send events with timestamp 12:00:04. The window can only fire when both streams send the watermark 12:00:04 or higher. Note that we say 12:00:04 or higher because the Source2 may directly see events with timestamp 12:00:06 after events with timestamp 12:00:04 in which case the watermark for source 2 will advance to 12:00:05 directly from 12:00:03.

Table 1.2 shows the time windows in which our three events described in Table 1.1 would have been processed when using event time based processing.

Table 1.2 Event time processing ensures events are processed consistently from operator to operator in the Flink application pipeline

Operator	Time windows			
	12:00:00-12:00:05	12:00:05-12:00:10	12:00:10-12:00:15	12:00:15-12:00:20
A	101	102, 103		
B	101	102,103		

What happens if one of the stream is delayed substantially? In such cases we can define custom triggers which function based on our specific business needs. Flink provides the API level control to handle these situations. We will defer this discussion until chapter 7.

The concept of watermarks is extremely important to understanding event-time based processing. We will discuss it in more detail in chapter 4 and chapter 7.

1.4.2 Processing time-based windows

When windows are based on processing time, the Windows Assigner will assign events to a window based on when it arrives at that operator. Processing time for a given event is the time the Windows Assigner reads the event for assignment to windows. Assume we are performing aggregations on 5 second windows. For a window based on time interval 12:00:00-12:00:05 where we assign events which arrive with timestamps starting 12:00:00 and ending (but not including) 12:00:05. We have another window 12:00:05-12:00:10 and so on. Events are assigned to one or the other window based on when they arrive for processing at the operator.

The trigger is applied when a timer assigned to the window expires. Thus when the time 12:00:05 based on the wall clock of the machine elapses, the window representing the

interval 12:00:00-12:00:05 is triggered. Similarly, at 12:00:10 the window for the interval 12:00:05-12:00:10 is triggered.

The important point to note about processing time windows is, even for windows of the same size in a Flink pipeline, events processed in a window for certain time interval, can find themselves in different time interval window in a downstream operator since the window assignment is based on when the events are evaluated by the Windows Assigner of an operator. Network delays can cause events to be assigned to different windows at each operator. Table 1.3 illustrates this situation. Note how event 101 is processed in a different window from events 102 and 103 for operator 1, but operator 2 processes them together in the same window.

Table 1.3 Processing time based processing is fast but can lead to inconsistent assignment to windows across multiple operators in the processing pipeline

Operator	Time windows			
	12:00:00-12:00:05	12:00:05-12:00:10	12:00:10-12:00:15	12:00:15-12:00:20
A		101	102,103	
B				101,102,103

1.4.3 Ingestion time-based windows

Ingestion time windows are similar in function to event time windows except that the timestamp is assigned based on the arrival time at the streaming source operator. The rest of the operations are identical to the event-time based windows. Watermarks are added based on the ingestion time using the same mechanism as event-time based processing.

Like event-time based processing (and unlike processing-time based processing), ingestion-time processing ensures that events are consistently assigned to the same time windows as they flow through the application pipeline. Unlike event-time based processing there is no risk of late arriving events as the timestamps are assigned on arrival. This implies that watermark events can never be delayed. This also means that, once a window is evaluated for an operator, it will never be used again and can be safely discarded.

Ingestion time based processing has a very slight overhead as compared to processing time based processing. But results are more consistent compared to processing-time based processing. Table 1.4 illustrates how our three events are processed using ingestion-time based processing. Due to a delays in arrival of events 101 and 102, they end up getting processed in the same window.

Table 1.4 Ingestion time processing also ensures events are processed consistently from operator to operator in the Flink application pipeline.

Operator	Time windows			
	12:00:00-12:00:05	12:00:05-12:00:10	12:00:10-12:00:15	12:00:15-12:00:20
A		101,102,103		
B		101,102,103		

1.4.4 Using various time windows to support Lambda Architecture style of processing

It is common to use processing-time windows to implement the Lambda Architecture. In the Lambda Architecture (http://lambda-architecture.net/) streaming events are processed as quickly as possible at the cost of some accuracy to provide decision makers with fast (but approximate and somewhat inaccurate) results. Eventually a batch processing system will correct these approximate results. In Flink processing-time or ingestion-time based processing can provide the fast approximate results. Another Flink pipeline with the same configuration but operating using event-time based processing will also operate on this event stream and correct any inaccuracies in the results produced by the processing or ingestion time based pipeline. Event-time based processing will always follow processing-time and ingestion-time based processing due to the delays introduced by watermark injection.

Due to its consistency of windows assignment through the processing pipeline, ingestion-time should be preferred to processing-time to implement the fast approximate phase of Lambda architecture based processing. Table 1.5 combines tables 1.2, 1.3, and 1.4 to provide a side-by-side comparison of various types of time windows.

Table 1.5 Comparing various types of time windows. Note the consistency of window assignment between operators of a pipeline in event time and ingestion time processing as compared to processing time based processing.

Window Type	Operator	Time windows			
		12:00:00-12:00:05	12:00:05-12:00:10	12:00:10-12:00:15	12:00:15-12:00:20
Event Time	A	101	102,103		
	B	101	102,103		
Processing Time	A		101	102,103	
	B				101,102,103
Ingestion Type	A		101,102,103		
	B		101,102,103		

1.4.5 Tumbling versus sliding time windows

Time windows can be defined as tumbling or sliding. In all our examples so far we have considered tumbling windows. Tumbling windows do not have overlapping portions. Thus 12:00:00-12:00-05 and 12:00:05-12:00:10 are tumbling windows. Each event is assigned to one only one window. Figure 1.6 shows and example of a tumbling window. Notice how events can only be assigned exactly one window.

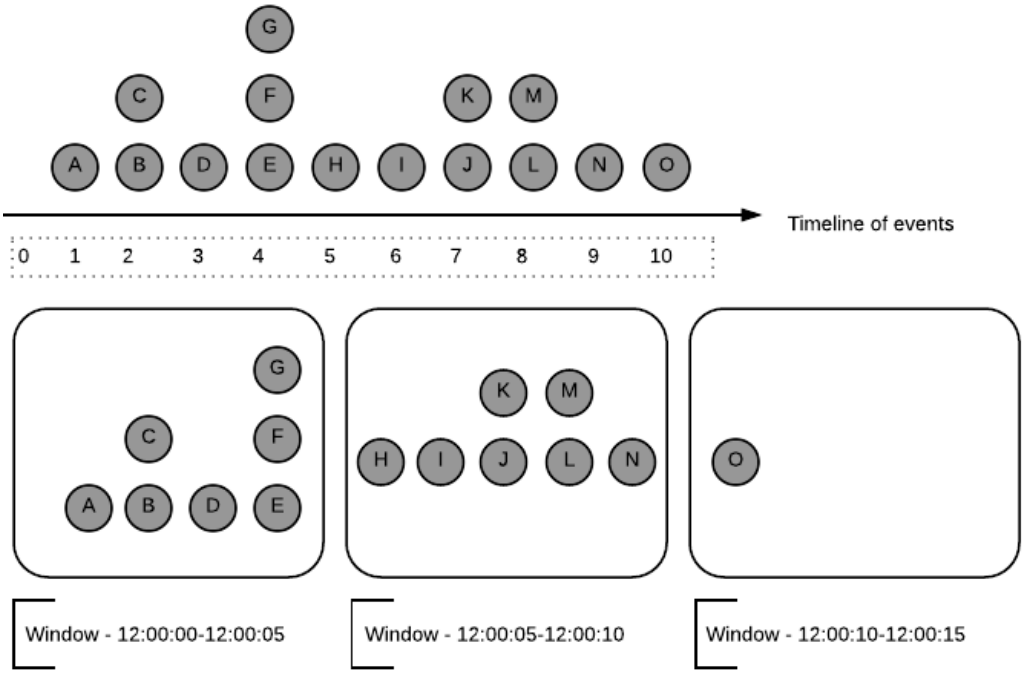

Figure 1.6. Tumbling windows. Events are assigned to only one window and windows are non-overlapping

Time windows can also be sliding. An example of a sliding window is, a window which 5 seconds long which slides every 1 second. For examples, 12:00:00-12:00:05, 12:00:01:12:00:06, 12:00:02:12:00:07 are examples of sliding windows. Events will be assigned to more than one window based on their timestamp. Figure 1.7 shows an example of sliding windows. Notice how windows are overlapping and events are assigned to one more than one window.

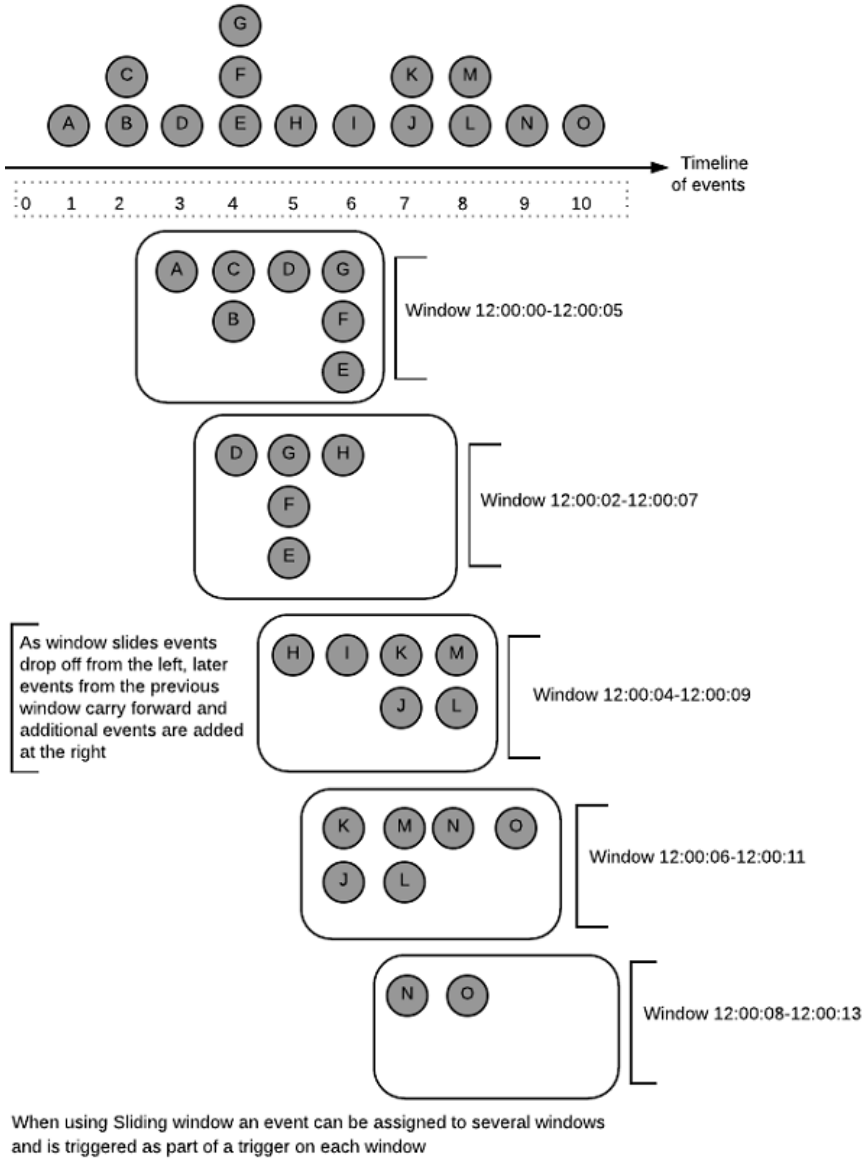

Figure 1.7. Sliding windows. Events are assigned to multiple overlapping windows.

1.5 Count-based windows

When count-based windows are used, the Reduce type operator uses a single window known as the Global Window. All elements arriving at the operator are assigned to the same window.

Each key gets its own pane in the window. The Trigger associated with each window pane will fire the window pane based on number of elements in the pane. When this count crosses a predefined threshold, the Trigger associated with the pane will fire the pane and hand those elements to the operator for evaluation.

The count window can be one of the two types:

- Tumbling count window – In this type of count windows, the pane is fired when the element count in the pane reaches a pre-defined level. For example, if this threshold is set to 100, the pane will fire when the pane has 100 elements. The trigger will send only those 100 elements for evaluation and remove these evaluated elements. Events will continue to accumulate in the pane and when the number of elements in the pane reached 100, it will fire again. The process repeats itself with an event being evaluated exactly once per operator.
- Sliding count window – In this type of count window the trigger mechanism depends on two parameters, the size of the window pane and how much it slides by. Imagine a configuration defined for a sliding window with size 100 and slide size 10. When the size of a window pane first reaches 100, the Trigger hands it to the Evictor component. The Evictor component will iterate through these elements and sends all 100 elements for evaluation. The evaluated elements are not removed from the window pane. Elements continue to be added to the window pane. When the size of the pane reaches 110, the pane is once again handed to the Evictor component. This time the Evictor component removes the 10 oldest elements in the window pane and hands the remaining 100 elements for evaluation. Thus 90 most recent elements from the earlier evaluation are re-evaluated along with 10 new elements. This process repeats itself. During each evaluation the 10 oldest elements are moved from the window pane and exactly 100 elements (most recently arrived) are evaluated by the operator.

Tumbling Count windows can be used for the following types of use cases:

- To calculate trends in website like twitter. For example, to calculate which hashtags are trending we can calculate the sum for each hashtag for the current date each time a certain threshold (ex. 1000) tweets of a given hashtag are made.
- To calculate leader boards and award badges. For example, give a bronze badge each time a user responds to 10 questions on an online question-answer forum like Stack Overflow. Next grant a silver badge to a user when they answer 50 questions and gold badge when the user answers 100 questions.

Sliding Count windows can be used when you need to look for patterns in the last n elements. If you are monitoring a temperature sensor and need to send an alert if three consecutive readings are above a threshold you will use a sliding count window with size equal to three (3) and slide equal to one (1). For each new reading this window triggers and determines if the last 3 elements are above the threshold temperature. A tumbling window of size 3 is

inadequate for such as use-case because the three consecutive reading may be in different tumbling window boundaries.

1.6 Batch processing as special case of stream processing

Finally, we are ready to understand how Flink can treat batch processing as a special case of stream processing. Flink can perform batch processing by using the Global Window assignment which we discussed in the count-based window. A custom trigger will evaluate the pane only when all the elements for that pane have arrived. The elements from the pane will then be discarded. The results of this evaluation will be exactly the same as if we had used a separate batch processing API like Spark.

But, batch processing is still very commonly used. And a batch processing framework can make several performance and memory optimizations based on the knowledge that it is processing all elements at once. We will explore what constitutes this knowledge in the next chapters of the book. To take advantage of these optimizations Flink provides a separate batch processing API called the Dataset API. This API uses the same underlying streaming framework components to execute the job but has some specialized pieces which enable the batch processing jobs to execute faster. Dataset API is the topic for chapter 3.

Regardless of the option you choose, the processing operators you develop in Flink and the ones available in the Flink libraries can be reused across both (streaming and batch) API's in Flink.

1.7 Pipelined processing and backpressure handling

Backpressure is a situation where the Flink system is receiving data faster than it can process it. This results in the backlog at the source. A system not designed to handle backpressure elegantly will end up in one of these two undesirable states:

- The system components will run out of memory. It is unacceptable for an enterprise system to crash when it receives a deluge of data is unacceptable.
- The system will start dropping data elements at the source. Enterprise systems rely on accurate processing of data. Loss of data is unacceptable.

In this section we will explore how Flink uses its pipelined processing architecture to elegantly handle backpressure without running out of memory or dropping data elements at the source.

In a typical batch processing scenario, a downstream operator in the topology cannot start until the upstream operators have finished processing. This is not practical in a streaming scenario because streams are essentially infinite.

Flink operators are provided input and output buffers. Data is received into its input buffers and when evaluated by the operator, it is pushed to its output buffers. The system is configured to push these buffers to downstream operators (over a TCP connection if operators are on different nodes) when the buffers are full. The size of the buffers can be configured to control latency in a Flink system. Flink also allows for buffers to be pushed downstream when

a user defined timer elapses per buffer. This feature needed to control latency when data is not arriving fast enough to fill the buffers. This is known as pipelined processing and it allows data to be constantly streamed and evaluated using user-defined settings to control latency of responses.

1.7.1 Backpressure handling

Flink handles backpressure gracefully. It does not run out of memory or drop data, which would cause loss of data. The key components of backpressure handling are:

- Durable event store which delivers event data in the order it was received. We have looked at Kafka as an example of such as a store.
- Pipelined processing using Input/Output buffers

Consider figure 1.8, which illustrates what a Flink system looks like when there is no backpressure. It contains two operators, a Map operator and a Sum operator. A word count application configured to operate in the streaming mode will have this configuration.

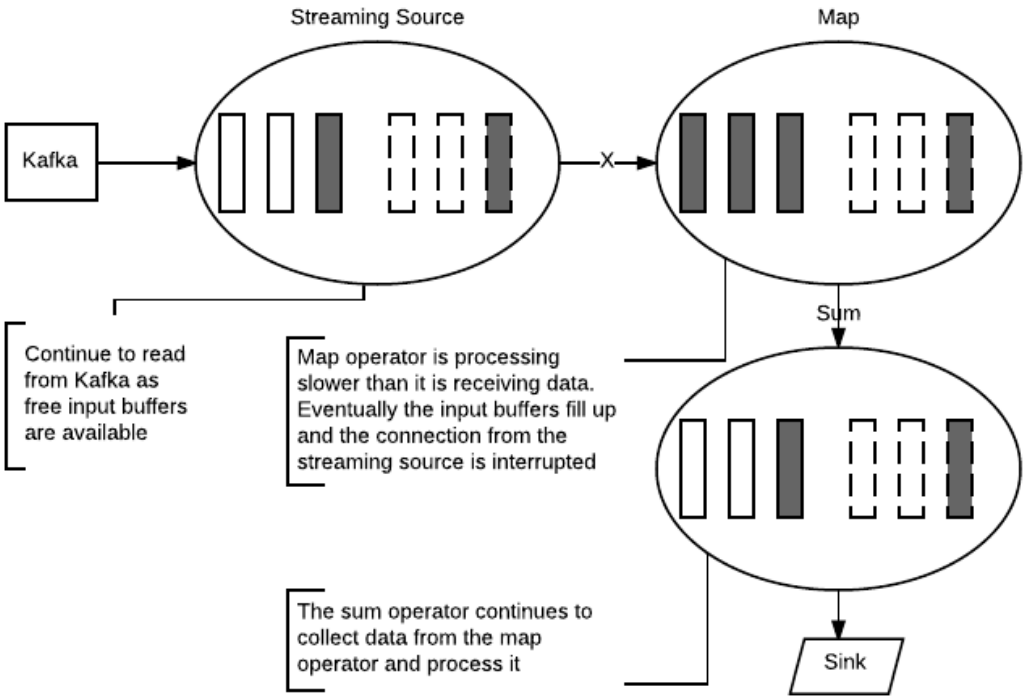

Figure 1.8. Steady sate with no backpressure. The system is processing the data faster than it arrives.

In this situation, there is no backpressure and the system continues reading data from Kafka and processing it through the entire topology of operations.

Figure 1.9 shows a situation where the backpressure starts to build up from the Map operator.

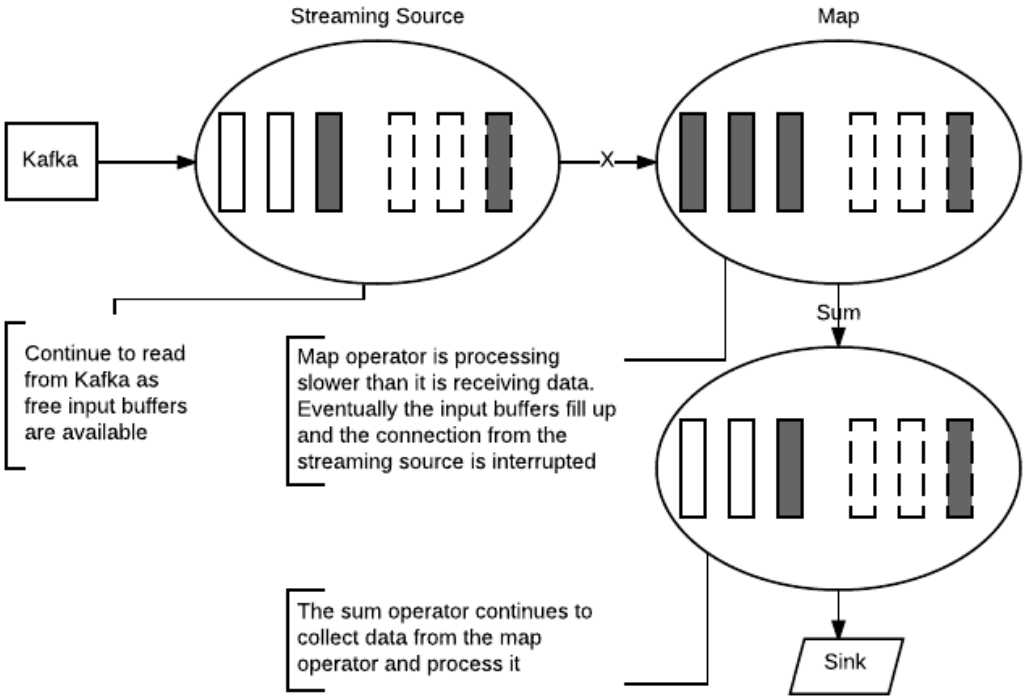

Figure 1.9. The Map operator is processing slower than it receives data. The input buffers fill up until no space is available in them. The connection with the streaming source is interrupted.

The Map operator is now processing data slower than it is receiving it. Consequently, the input buffers on the Map operator fill up until no more input buffers are available. This causes the connection to the streaming source to get interrupted. The Streaming source continues to read data from Kafka as it has available input/output buffers. The Sum operator also continues processing due to the availability of input/output buffers.

Figure 1.10 shows what happens if the Mapper does not release its input buffers soon enough. The backpressure will then accumulate in the streaming source as its input/output buffers fill up.

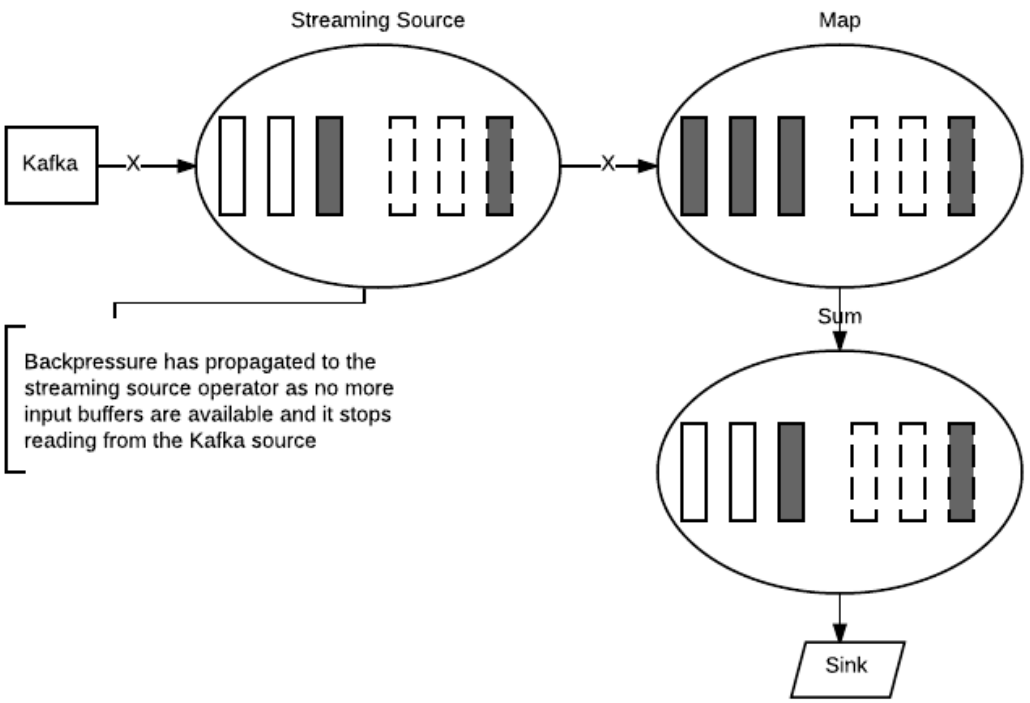

Figure 1.10. Backpressure has now built up into the Streaming Source and it stops reading data from the Kafka source.

At this point the streaming source stops reading data from Kafka. Note that data continues to be added to Kafka from external sources. This is the main reason why we need a durable messaging system. Without a system like Kafka we would either run out of memory or be forced to drop data elements to handle backpressure.

Durable staging of data is the key to handling backpressure

One of the reasons Flink does not connect to external streaming sources of data directly is to be able to handle backpressure gracefully. A system like Kafka guarantees that all data received is persisted in a failure resilient storage. Flink is free to stop processing this data to catch up on a large backlog of events. When it resumes processing, it reads from the last read point in Kafka (or other durable storage system being used).

Eventually, the Map operator catches up on its processing and proceeds to have available input buffers. This causes the streaming source to empty its output buffers into the input buffers of the Mapper. The streaming source now starts reading out of the Kafka system again. However, now the Sum operator has no free input buffers and it stops collecting data

from the Mapper operators. This will cause the connection to the Mapper to be interrupted. Figure 1.11 illustrates this situation.

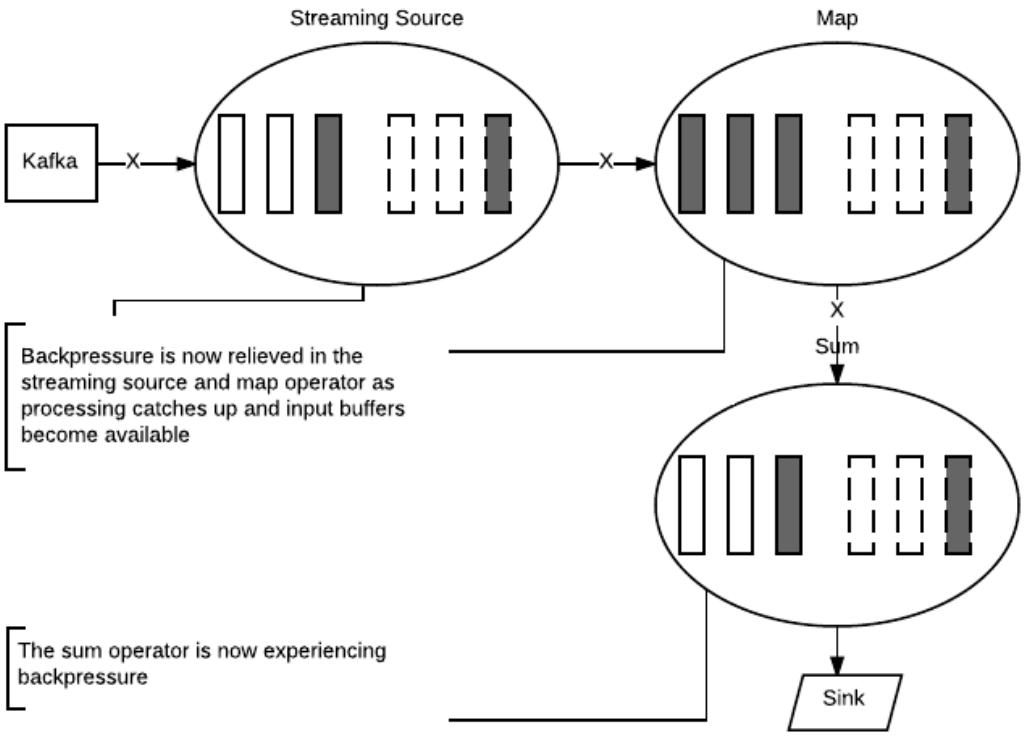

Figure 1.11. Backpressure has relieved on the streaming source and Mapper as the Mapper catches up with its processing but the Sum operator does not have any input buffers available.

Thus each stage of the Flink processing pipeline can handle backpressure gracefully without running out of memory or loss of data.

1.8 Failure recovery and exactly once processing using checkpoints

One of the features that distinguish Flink from other streaming frameworks is its support for exactly once processing. Most frameworks offer at-least once processing (meaning an event may be re-processed giving duplicate results) or at-most once processing (meaning an event can be lost where we get missing data). These trade-offs are typical in a distributed system where components of the processing pipeline may fail due to the hardware failures in a large cluster. Distributed frameworks need to be able to handle such failure. Ignoring such failure

leads to at-most once processing and simply re-processing the data on a healthy node may lead to some elements being re-processed again resulting in at-least once processing.

Flink supports exactly-once processing by taking period snapshots called checkpoints. A checkpoint represents a time interval which is global to the Flink cluster. This means that the checkpoint start and end time instance are the same across the cluster nodes.

The checkpoint contains the state of the Kafka pointers at the start and end of the snapshot. Events read as a part of the checkpoint flow through the Flink processing pipeline. Each operator in the pipeline can will store the results of evaluating the events which are part of the checkpoint. Thus a checkpoint captures the following information:

- Kafka pointers at the start and end of the global time period representing the snapshot.
- State of each operator just before it emits the results of its evaluation for the data elements which are part of the snapshot.

Checkpoint is considered complete only when all the sinks commit for the data elements which hare part of the same snapshot. If Flink needs to re-process data, it will start the entire pipeline from the last successful snapshot. This will cause it to read Kafka pointers from the location contained in the last successful checkpoint and the pipeline will continue processing from there.

Flink also provides the option of choosing at-least once processing by relaxing some of the strict consistency constraints placed on how checkpoints are captured and propagated through the system. It is faster than exactly-once processing but can result in duplicate processing occasionally. Thus you can make trade-offs between performance and accuracy on a use case basis.

Details about how consistent checkpoints are taken across the system will be discussed in a chapter dedicated to the concepts of failure recovery.

1.9 Reprocessing using save points

Imagine you need to do one of more of the following:

- You can perform application upgrades and initiate processing from a point in time in the past
- You need to perform A/B testing on a currently executing pipeline. A/B testing allows two versions of the code to be tested using the same initial conditions (save points) to compare results or performance. For example, you have collected user activity per session where a session is defined a period of activity followed by a period of inactivity for 15 minutes. You use these results in the recommendation engine. Your business analyst wants to know how the recommendations produced by the engine change if session is defined based on an inactivity period of 5 minutes. You will need to re-process data from a point in time in the past and compare the results of the two recommendation engines.
- You have just deployed a new version of the application. After running it for one day

you discover there is a bug in the application. You deploy the code fixes but you will need to re-process the data from the time the new code was deployed yesterday to ensure the results are correct.

Flink supports the above use cases using the concept of save-points. A save-point is a user initiated snapshot of the system at a point in time. A Flink application pipeline can be restarted from a save point. The Fink application will start processing data from the state of the system stored in the save point. Save points enable the recovery use cases mention as follows:

- Initiate a save point just prior to the application upgrade. Then perform the application upgrade. Initiate processing from the save point.
- Periodically create save points for your program. For the new definition of session (5 minutes of inactivity) initiate a new instance of your Flink pipeline from one of your save points which define the session as a period of activity followed by over 5 minutes of inactivity. Run your current pipeline based on the old definition of session (15 minutes of inactivity) from the same save point. Collect results and train your recommendation engines using both datasets and compare the results.
- Just before you deploy your new version of the application take a save point. When a bug fix is deployed you will just need to re-process from that save point.

Save points will be discussed in more detail in a chapter dedicated to failure, recovery and state management in Flink.

1.10 Real world example – news website

In this book, we will use an example of a fictitious news website called *Newsflink*. Newsflink is very popular website, and a large number users visit the site daily to read news articles. As the website serves user request for news articles, for each response it also generates a newsfeed event, which is sent to the Flink system for analysis. Newsflink, being a very popular website, has a large number of web-servers serving the requests simultaneously. Figure 1.12 shows the overall workflow of events within the Newsflink infrastructure.

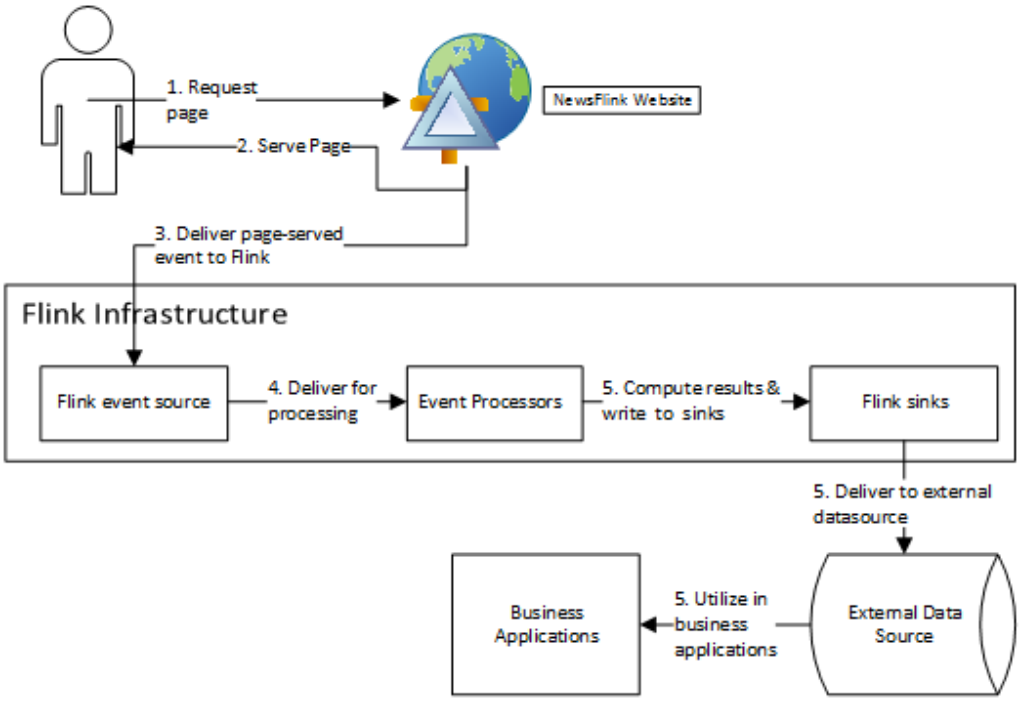

Figure 1.12. Users request news articles, which are served to the user. The events representing this response are sent to the Flink system for Streaming Analysis.

The workflow of the events in the Newsflink infrastructure is as follows:

1. The user (anonymous or logged-in) of the website makes a request for a news article.
2. The webserver serves the news article.
3. Simultaneously, the webserver also generates an event representing this response and sends it to the Flink system for streaming analysis.
4. The Flink system first persists the event (Kafka can be used to persist events) to ensure it is never lost and can be re-processed if necessary, and then delivers it to the Flink processing units for processing.
5. The results (typically user defined aggregations) are computed and delivered to the business systems.
6. The business applications then use these results for various purposes. For example, the results can be used to feed recommendation engines that are used to propose similar articles to the user.

Why did we select Newsflink as an example?

Streaming applications are used to support a wide array of exotic use-cases ranging from real-time ad generation based on online user behavior to real-time fraud detection. Why then did we select Newsflink as our use case. The reason we did that is because we want to show that Streaming can be used to provide business value in everyday ETL style applications.

We will also use very limited use case examples to illustrate the capability of Flink. For example, we will frequently use the use case of finding aggregate page count per section and sub-section of the website. This is done on purpose. We do not want you to get lost into the business details of understanding a use-case. Instead we want to use simple yet realistic use-cases to illustrate what Flink can and cannot do.

Where appropriate we will describe other use cases for a given topic but will revert to a familiar one for source code illustrations. Our goal is to provide enough details to enable you to apply this knowledge to your own use-cases.

1.10.1 Event Schema

The schema of the event sent to Flink System for analysis is depicted in figure 1.13

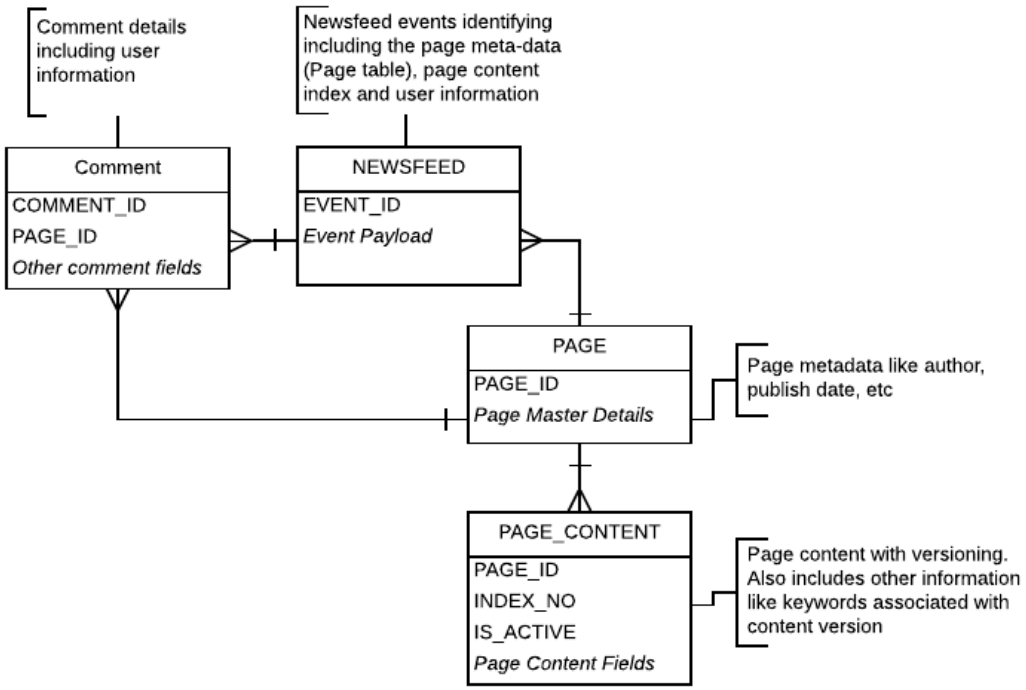

Figure 1.13. The data model for our Newsflink website. The master data sources are the PAGE and the PAGE_CONTENT tables. The NEWSFEED and COMMENT are the streaming sources.

The data model of the Newsflink corporation includes the following sources (or tables in the traditional sense):

- PAGE – This source contains information about the page such as the name of the author, publish date, section (Ex. Sports), sub-section (Ex. Football), topic (Ex. Super bowl), etc.
- PAGE_CONTENT – This source contains the content for a given page. The content can be modified regularly in response to viewership statistics. Hence a page can support multiple version. Each version is also associated with key words associated with the content of that row. Only one version can be active at a time.
- NEWSFEED – This source identifies all activity by user for a given page. Each time a user accesses a page this event is generated. This event contains the information about the user, and the current state of the page. This is the most important source for the analytics use case. It is also very large. Thus a website receiving 100 million unique requests per day with an event payload of 1KB per request, will produce 100 GB of data each day. The newsfeed events will constantly stream into our system.
- COMMENT – A user may comment on a given page. This table stores each comment for a page along with the information of the user writing the comment. The comment

events will also be constantly streaming into our system.

We will use this data model in the book to describe the capabilities of Flink in next chapters. We will expand each data source at the appropriate time during the rest of the book.

1.11 Summary

- The Apache Flink framework supports processing of streams of events. Event streams are how businesses execute in the real world. True stream processing allows these events to be processed in near real time, allowing the decision makers to have faster access to information.
- Flink supports rich and complex windowing semantics. Flink supports windows based on time, count, session and custom criteria. Flink handles time based windows based on event-time, processing-time and ingestion-time. Windows can be non-overlapping(tumbling) or overlapping(sliding). These complex windows allow Flink to implement faster versions of the famous Lambda Architecture.
- Flink employs pipelined processing data where each stage of the pipeline will keep flowing the data to the subsequent stage. Pipelined processing allows graceful handling of backpressure situations where the system is processing slower than it is ingesting data. Flink handles backpressure without running out of memory or allowing data loss.
- Flink allows the user to select between at-least once or exactly-once processing in exchange of slightly increased latency. Flink's ability to take precise snapshot of the system state based on a system-wide global marking of time supports the exactly-once processing.
- Flink allows creation of user-defined snapshots called save points. Snapshots allow you to go back to a point in time to reprocess data, to perform A/B testing and to apply application upgrades or bug fixes.
- Flink treats batch processing as special case of stream processing. Flink operators can be reused across batch and streaming topologies. Treating batch as special case of streaming provides Flink with a more fine-grained ability to process data.

In the next chapter we will show you how to install Flink and write simple programs in Flink using the `DataSet`, `DataStream` and `Table` APIs of Flink.