On This Page

Learn Flink: Hands-On Training

Goals and Scope of this Training

This training presents an introduction to Apache Flink that includes just enough to get you started writing scalable streaming ETL, analytics, and event-driven applications, while leaving out a lot of (ultimately important) details. The focus is on providing straightforward introductions to Flink's APIs for managing state and time, with the expectation that having mastered these fundamentals, you'll be much better equipped to pick up the rest of what you need to know from the more detailed reference documentation. The links at the end of each section will lead you to where you can learn more.

Specifically, you will learn:

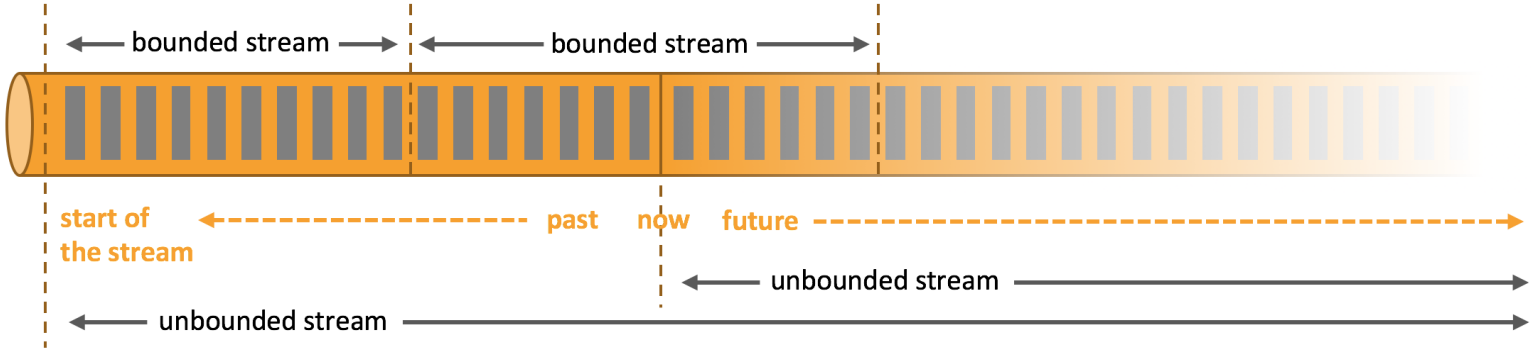
- how to implement streaming data processing pipelines
- how and why Flink manages state
- how to use event time to consistently compute accurate analytics
- how to build event-driven applications on continuous streams
- how Flink is able to provide fault-tolerant, stateful stream processing with exactly-once semantics

This training focuses on four critical concepts: continuous processing of streaming data, event time, stateful stream processing, and state snapshots. This page introduces these concepts.

> **Note:** Accompanying this training is a set of hands-on exercises that will guide you through learning how to work with the concepts being presented. A link to the relevant exercise is provided at the end of each section.

Stream Processing

Streams are data's natural habitat. Whether it is events from web servers, trades from a stock exchange, or sensor readings from a machine on a factory floor, data is created as part of a stream. But when you analyze data, you can either organize your processing around `bounded` or `unbounded` streams, and which of these paradigms you choose has profound consequences.



Batch processing is the paradigm at work when you process a bounded data stream. In this mode of operation you can choose to ingest the entire dataset before producing any results, which means that it is possible, for example, to sort the data, compute global statistics, or produce a final report that summarizes all of the input.

Stream processing, on the other hand, involves unbounded data streams. Conceptually, at least, the input may never end, and so you are forced to continuously process the data as it arrives.

In Flink, applications are composed of **streaming dataflows** that may be transformed by user-defined **operators**. These dataflows form directed graphs that start with one or more **sources**, and end in one or more **sinks**.

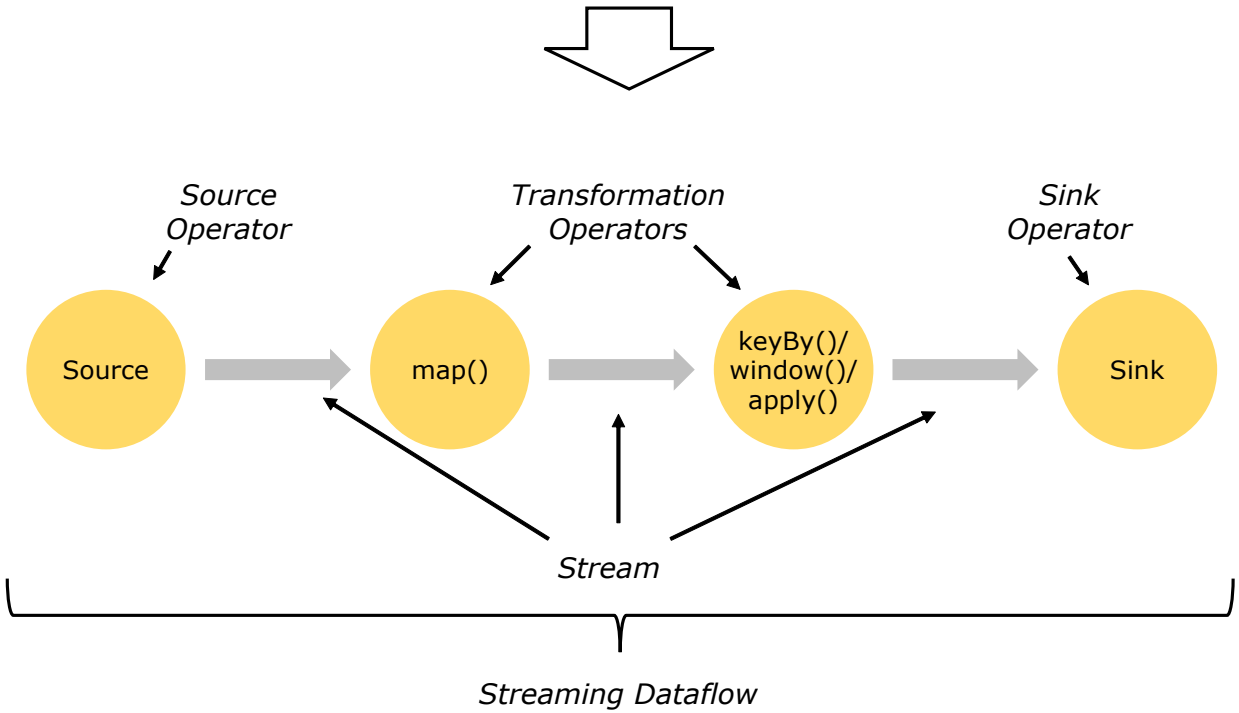
```
DataStream<String> lines = env.addSource(
                                new FlinkKafkaConsumer<>(…));

DataStream<Event> events = lines.map((line) -> parse(line));

DataStream<Statistics> stats = events
        .keyBy(event -> event.id)
        .timeWindow(Time.seconds(10))
        .apply(new MyWindowAggregationFunction());

stats.addSink(new MySink(...));
```

Source

Transformation

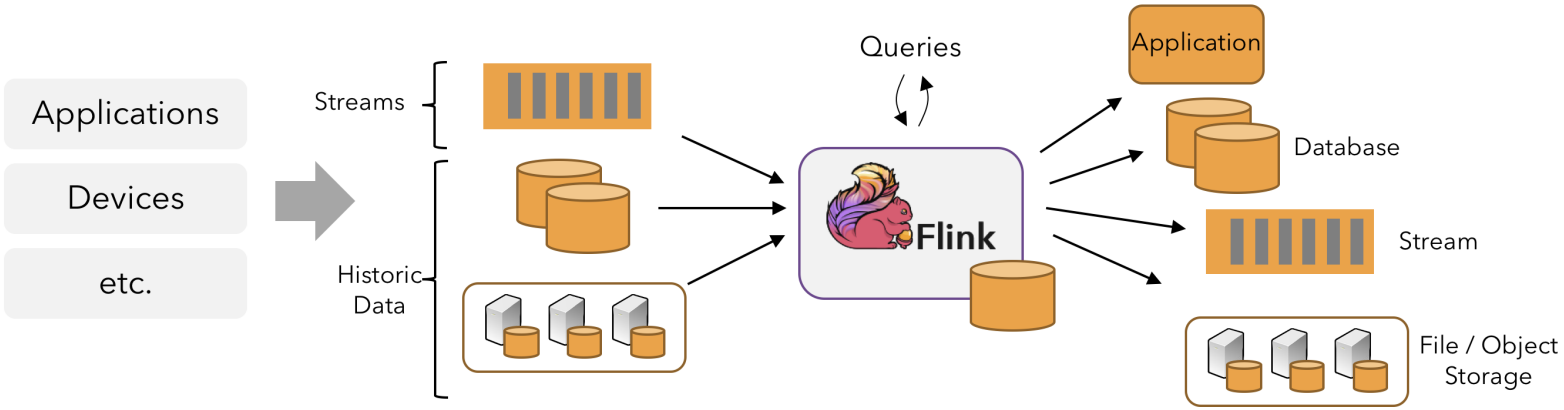
Transformation

Sink



Often there is a one-to-one correspondence between the transformations in the program and the operators in the dataflow. Sometimes, however, one transformation may consist of multiple operators.

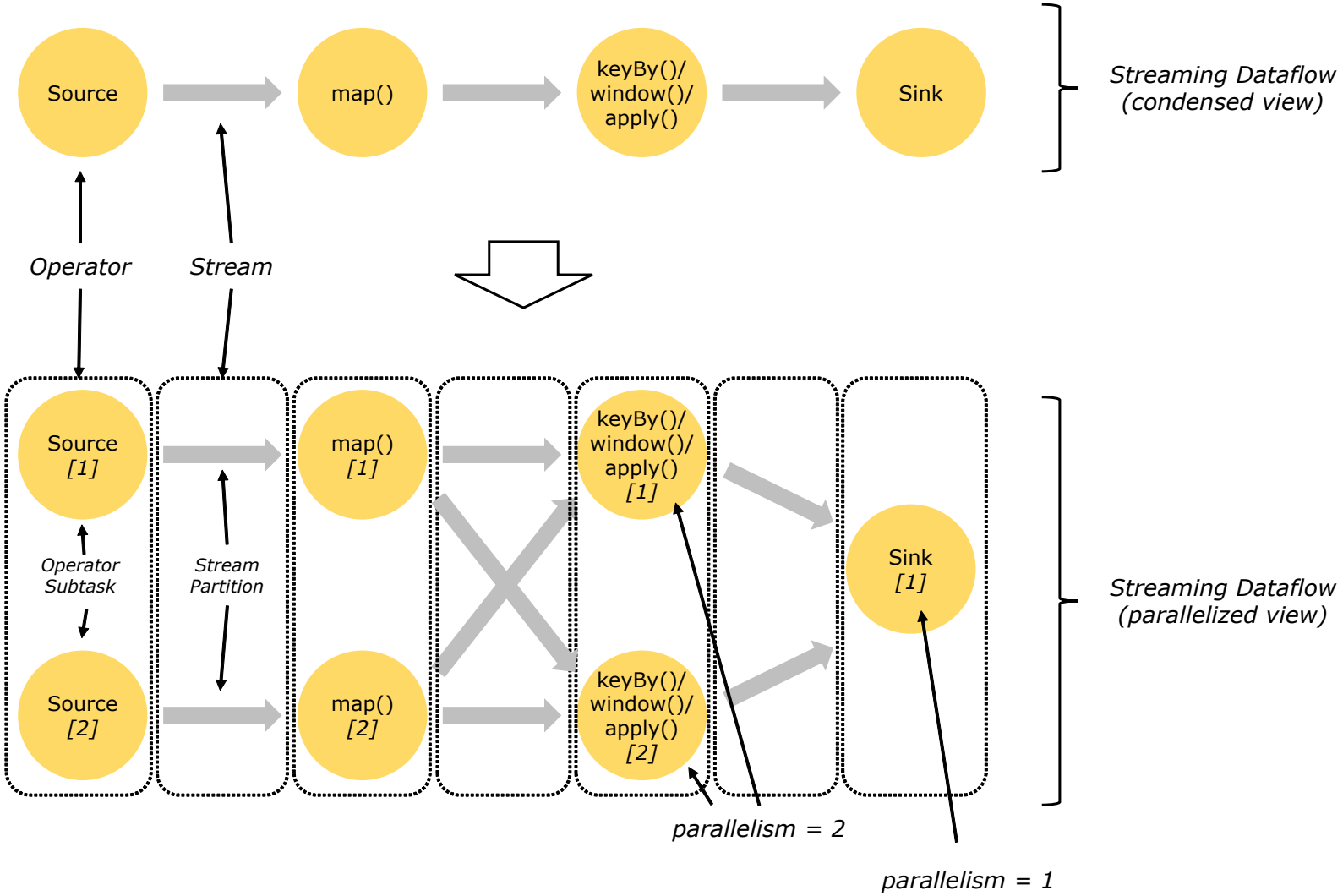
An application may consume real-time data from streaming sources such as message queues or distributed logs, like Apache Kafka or Kinesis. But flink can also consume bounded, historic data from a variety of data sources. Similarly, the streams of results being produced by a Flink application can be sent to a wide variety of systems that can be connected as sinks.



Parallel Dataflows

Programs in Flink are inherently parallel and distributed. During execution, a stream has one or more **stream partitions**, and each operator has one or more **operator subtasks**. The operator subtasks are independent of one another, and execute in different threads and possibly on different machines or containers.

The number of operator subtasks is the **parallelism** of that particular operator. Different operators of the same program may have different levels of parallelism.



Streams can transport data between two operators in a `one-to-one` (or forwarding) pattern, or in a `redistributing` pattern:

- **One-to-one** streams (for example between the Source and the map() operators in the figure above) preserve the partitioning and ordering of the elements. That means that subtask[1] of the map() operator will see the same elements in the same order as they were produced by subtask[1] of the Source operator.

- **Redistributing** streams (as between map() and keyBy/window above, as well as between keyBy/window and Sink) change the partitioning of streams. Each operator subtask sends data to different target subtasks, depending on the selected transformation. Examples are keyBy() (which re-partitions by hashing the key), broadcast(), or rebalance() (which re-partitions randomly). In a redistributing exchange the ordering among the elements is only

preserved within each pair of sending and receiving subtasks (for example, subtask[1] of map() and subtask[2] of keyBy/window). So, for example, the redistribution between the keyBy/window and the Sink operators shown above introduces non-determinism regarding the order in which the aggregated results for different keys arrive at the Sink.

Timely Stream Processing

For most streaming applications it is very valuable to be able re-process historic data with the same code that is used to process live data – and to produce deterministic, consistent results, regardless.

It can also be crucial to pay attention to the order in which events occurred, rather than the order in which they are delivered for processing, and to be able to reason about when a set of events is (or should be) complete. For example, consider the set of events involved in an e-commerce transaction, or financial trade.

These requirements for timely stream processing can be met by using event time timestamps that are recorded in the data stream, rather than using the clocks of the machines processing the data.

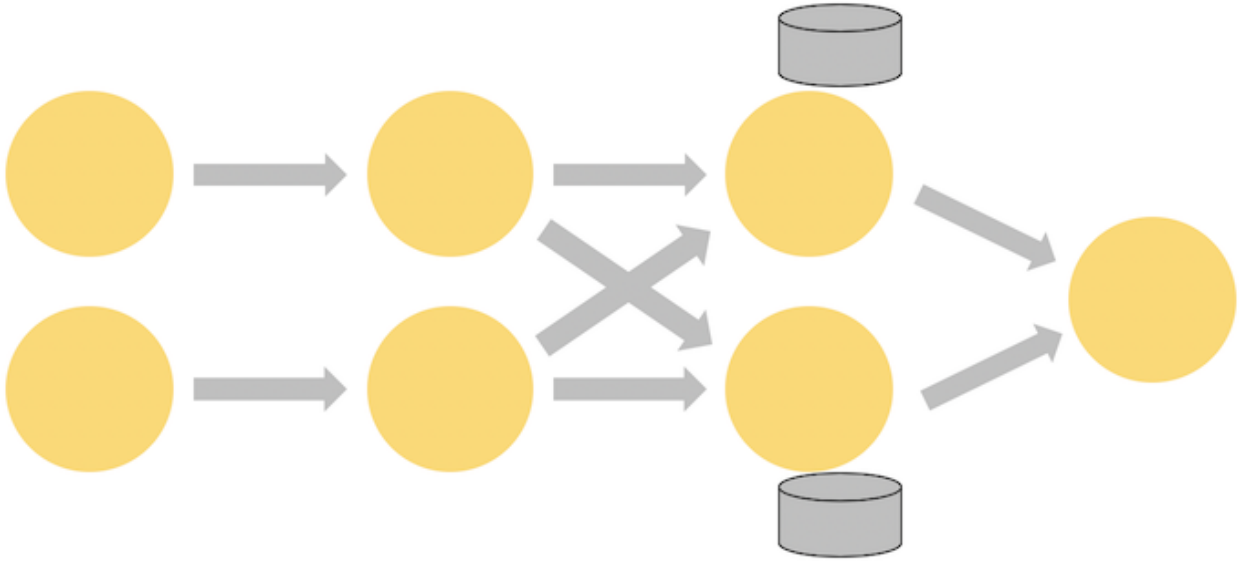
Stateful Stream Processing

Flink's operations can be stateful. This means that how one event is handled can depend on the accumulated effect of all the events that came before it. State may be used for something simple, such as counting events per minute to display on a dashboard, or for something more complex, such as computing features for a fraud detection model.

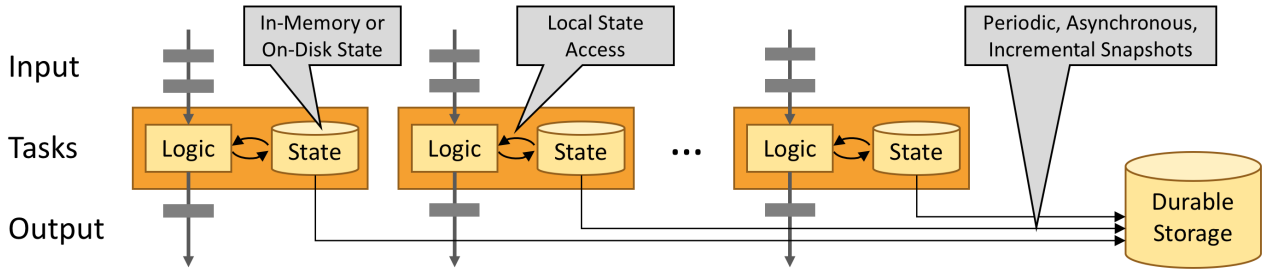
A Flink application is run in parallel on a distributed cluster. The various parallel instances of a given operator will execute independently, in separate threads, and in general will be running on different machines.

The set of parallel instances of a stateful operator is effectively a sharded key-value store. Each parallel instance is responsible for handling events for a specific group of keys, and the state for those keys is kept locally.

The diagram below shows a job running with a parallelism of two across the first three operators in the job graph, terminating in a sink that has a parallelism of one. The third operator is stateful, and you can see that a fully-connected network shuffle is occurring between the second and third operators. This is being done to partition the stream by some key, so that all of the events that need to be processed together, will be.



State is always accessed locally, which helps Flink applications achieve high throughput and low-latency. You can choose to keep state on the JVM heap, or if it is too large, in efficiently organized on-disk data structures.



Fault Tolerance via State Snapshots

Flink is able to provide fault-tolerant, exactly-once semantics through a combination of state snapshots and stream replay. These snapshots capture the entire state of the distributed pipeline, recording offsets into the input queues as well as the state throughout the job graph that has resulted from having ingested the data up to that point. When a failure occurs, the sources are rewound, the state is restored, and processing is resumed. As depicted above, these state snapshots are captured asynchronously, without impeding the ongoing processing.