Streaming processing for Dummies

Oleksandr Stepurko

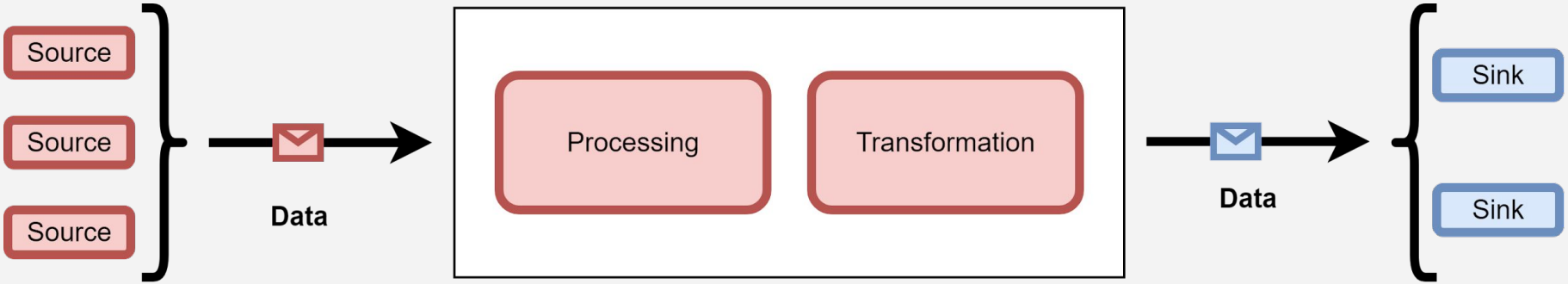
Senior Software Engineer. Andersen

Agenda

- Introduction to Streaming Processing
- Key Concepts in Streaming Processing
- Streaming Processing Frameworks
- Use Cases of Streaming Processing
- Streaming Processing in Action
- Challenges in Streaming Processing
- Conclusion and Q&A

Introduction to Streaming Processing

Stream processing is a data management technique that involves ingesting a continuous data stream to quickly analyze, filter, transform or enhance the data in real time. Once processed, the data is passed off to an application, data store or another stream processing engine.



Stream processing is needed to:

- Develop adaptive and responsive applications
- Help enterprises improve real-time business analytics
- Facilitate faster decisions
- Accelerate decision-making
- Improve decision-making
- Improve the user experience

Common stream processing use cases:

- Fraud detection
- Detecting anomalous events
- Tuning business application features
- Managing location data
- Personalizing customer experience
- Stock market trading
- Analyzing and responding to IT infrastructure events
- Digital experience monitoring
- Customer journey mapping
- Predictive analytics
- IoT data processing
- Network monitoring

Key Concepts in Streaming Processing

Data Streams: Unbounded sequences of data events.

Event Time vs. Processing Time: Distinguishing when an event occurred from when it's processed.

Stateful vs. Stateless Processing: Maintaining context across events.

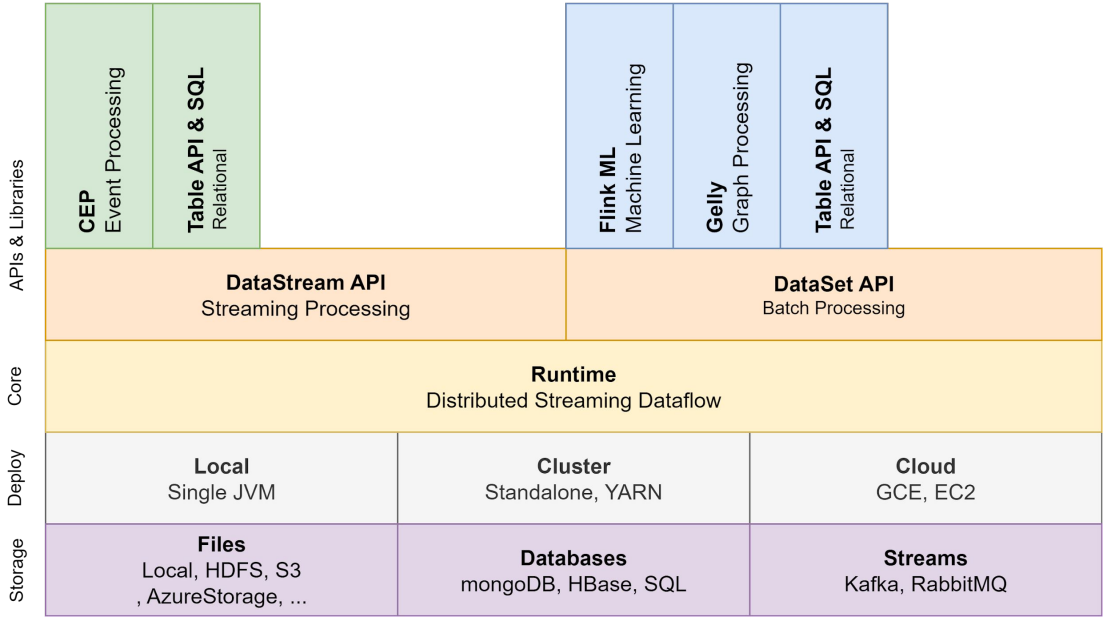
Windowing: Batching data over time or count-based windows.

Streaming Processing Frameworks

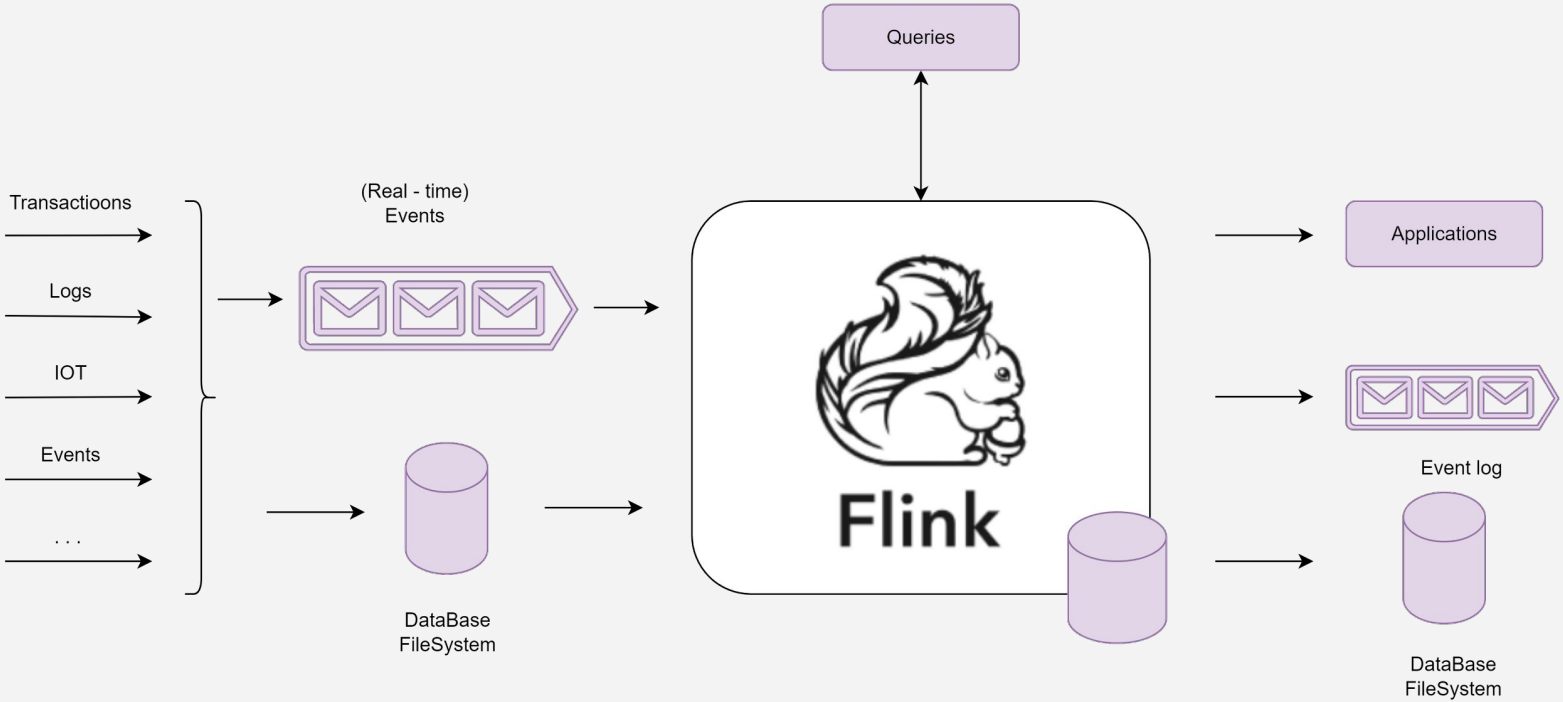
- Apache Kafka Streams
- Apache Flink
- Apache Spark Streaming
- AWS Kinesis
- Google Cloud Dataflow
- Microsoft Azure Stream Analytics

Apache Flink

Apache Flink is a real-time processing framework which can process streaming data. It is an open source stream processing framework for high-performance, scalable, and accurate real-time applications. It has true streaming model and does not take input data as batch or micro-batches.

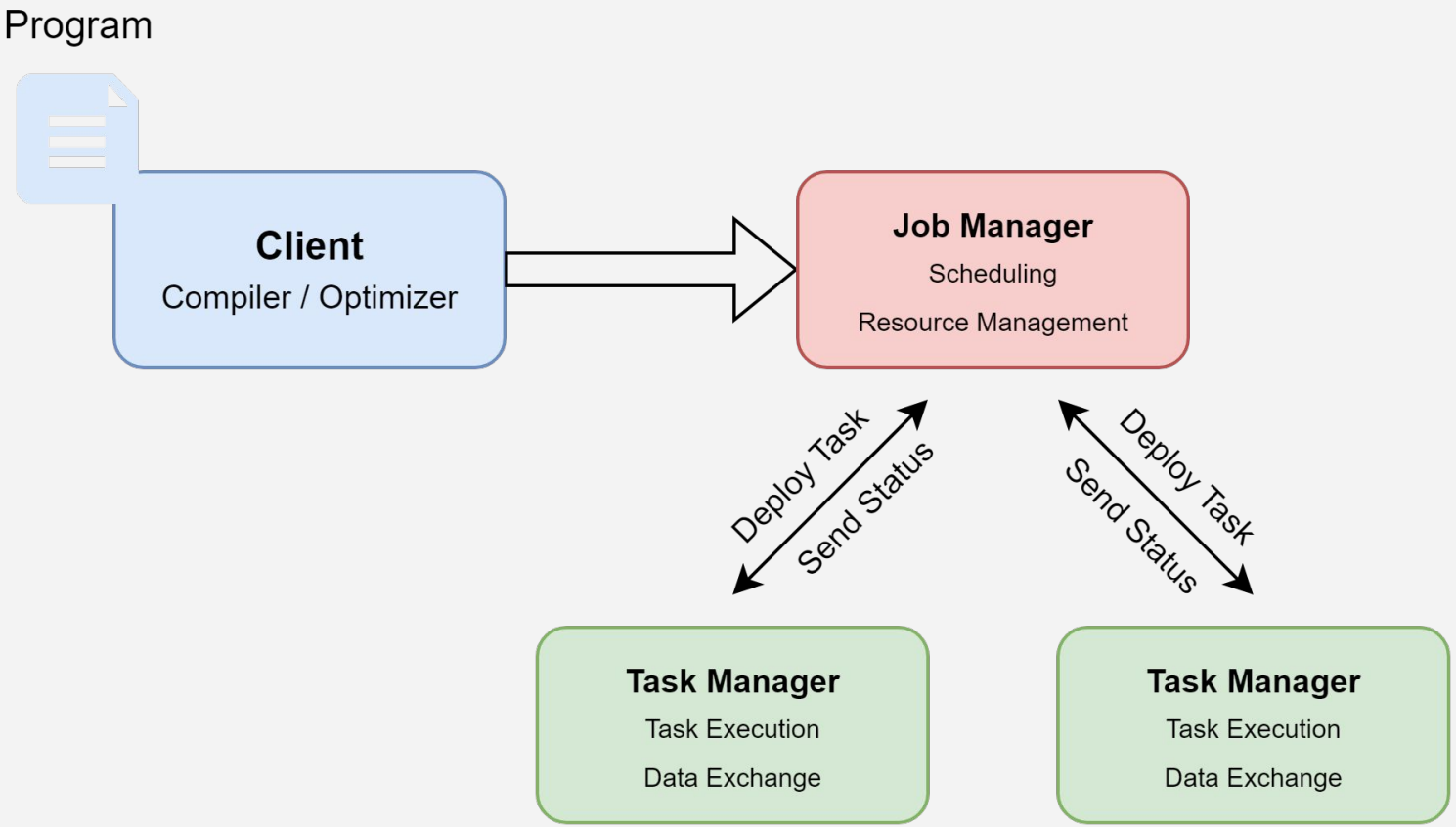


Apache Flink Architecture



Flink Kernel is the core element of the Apache Flink framework. The runtime layer provides distributed processing, fault tolerance, reliability, and native iterative processing capability. The execution engine handles Flink tasks, which are units of distributed computations spread over many cluster nodes. This ensures that Flink can run efficiently on large-scale clusters.

Apache Flink Job Execution Architecture



Apache Flink Architecture

Program

It is a piece of code, which you run on the Flink Cluster.

Client

It is responsible for taking code (program) and constructing job dataflow graph, then passing it to JobManager. It also retrieves the Job results.

JobManager

After receiving the Job Dataflow Graph from Client, it is responsible for creating the execution graph. It assigns the job to TaskManagers in the cluster and supervises the execution of the job.

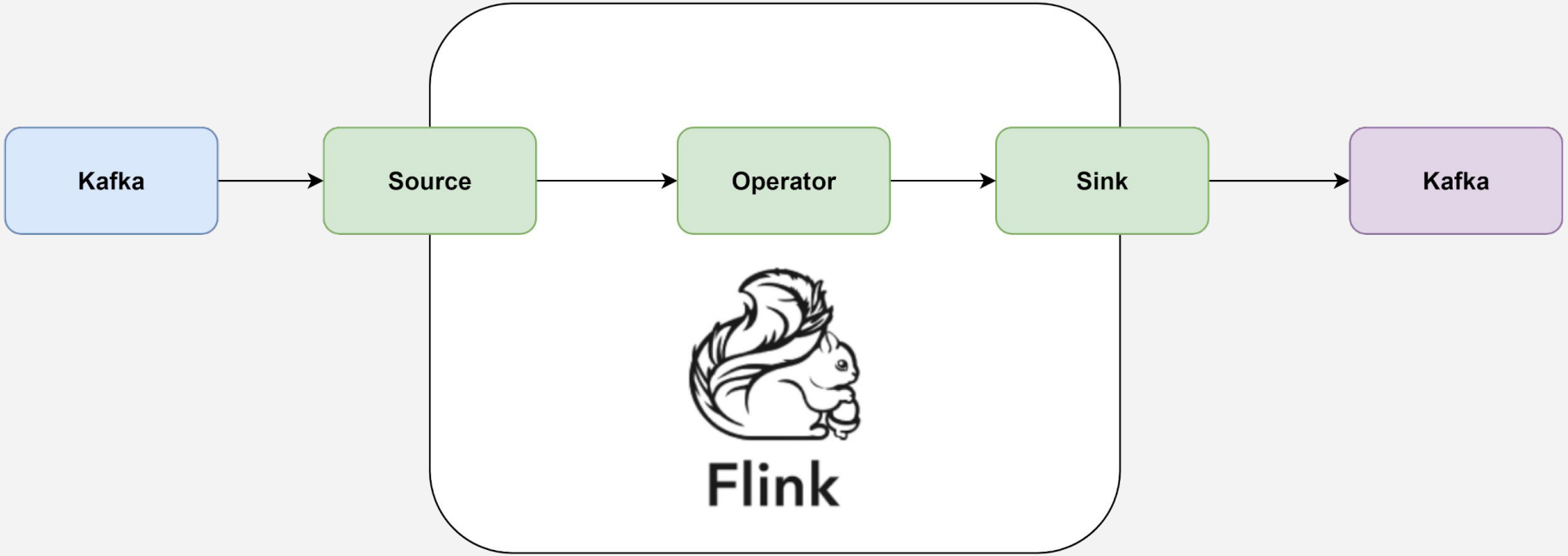
TaskManager

It is responsible for executing all the tasks that have been assigned by JobManager. All the TaskManagers run the tasks in their separate slots in specified parallelism. It is responsible to send the status of the tasks to JobManager.

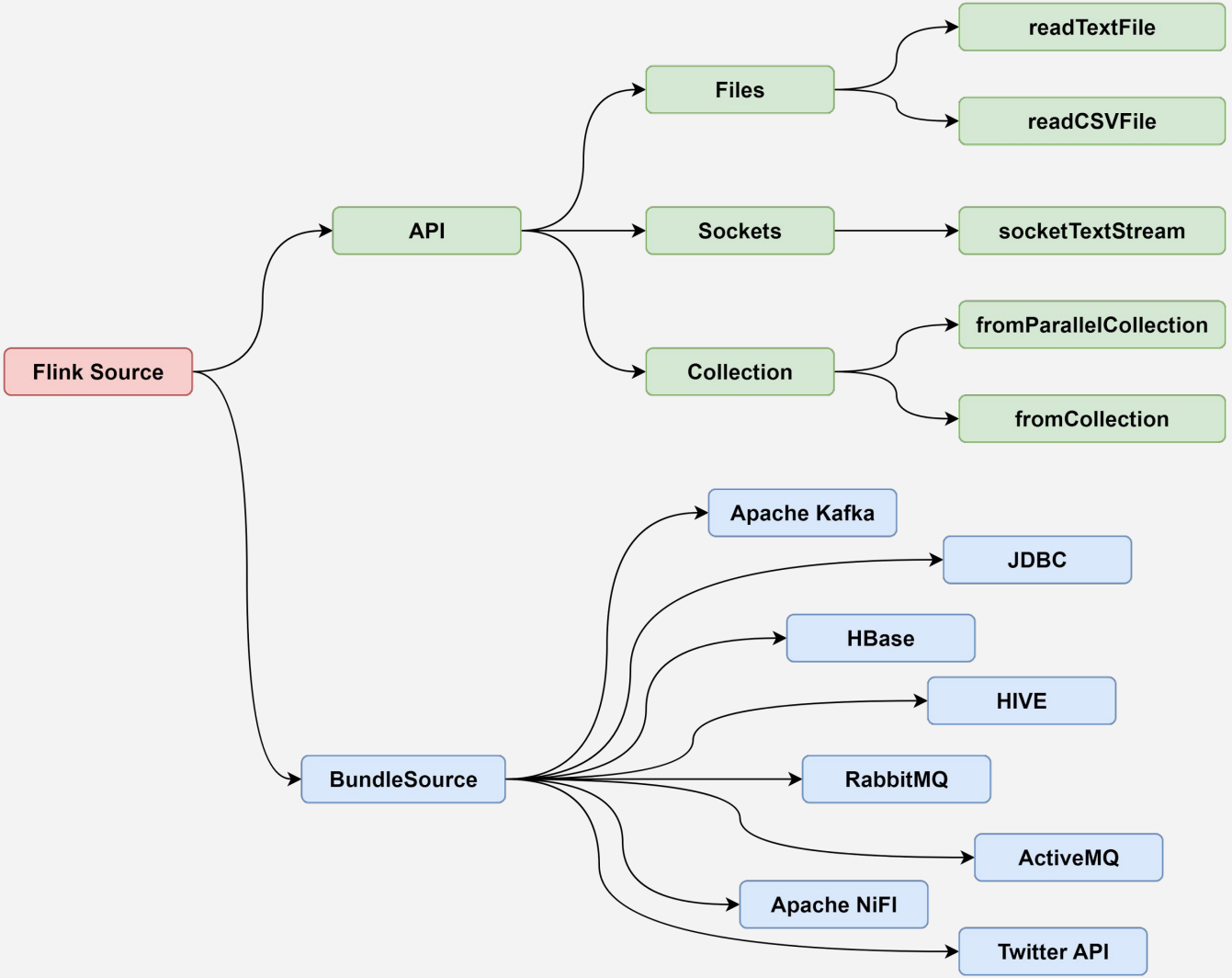
Apache Flink Connectors

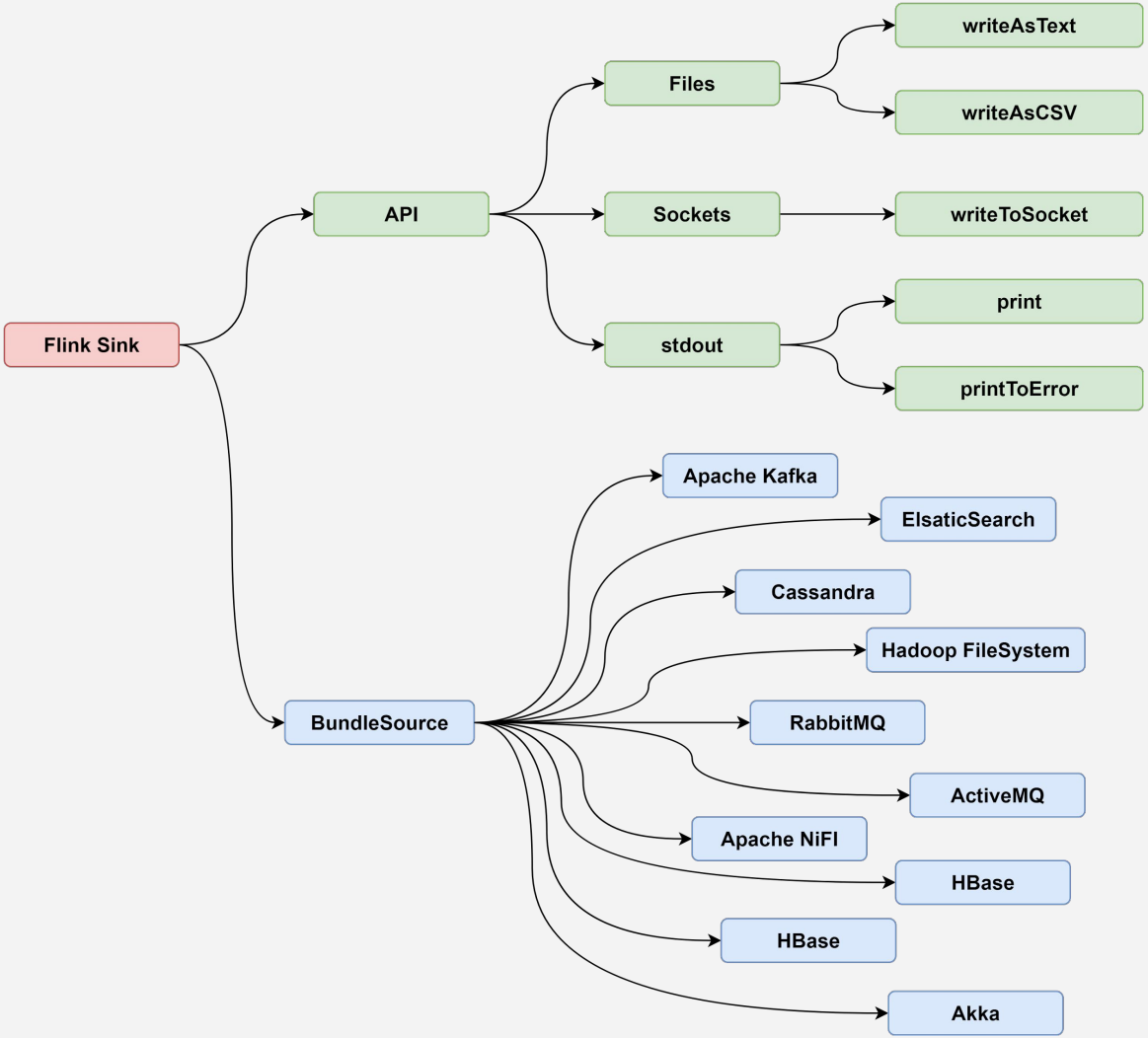
- **Important sources and destinations of Flink data:** Connectors are the bridge between Flink and external systems. For example, Kafka helps read data, process in Flink, and then re-write back to external systems such as HIVE and Elasticsearch.
- **Event control in data processing:** Event processing watermark and checkpoint alignment record.
- **Load balance**: Properly distribute data partitions based on loads of different concurrencies.
- **Data parsing and serialization:** Data may be stored in binary form in external systems and various forms of columns in the database. After writing data to Flink, it needs to be parsed before the subsequent data processing. Likewise, when writing data back to external systems, it also needs to be serialized, converting it into the corresponding storage format in the external system for storage.

Apache Flink Connectors



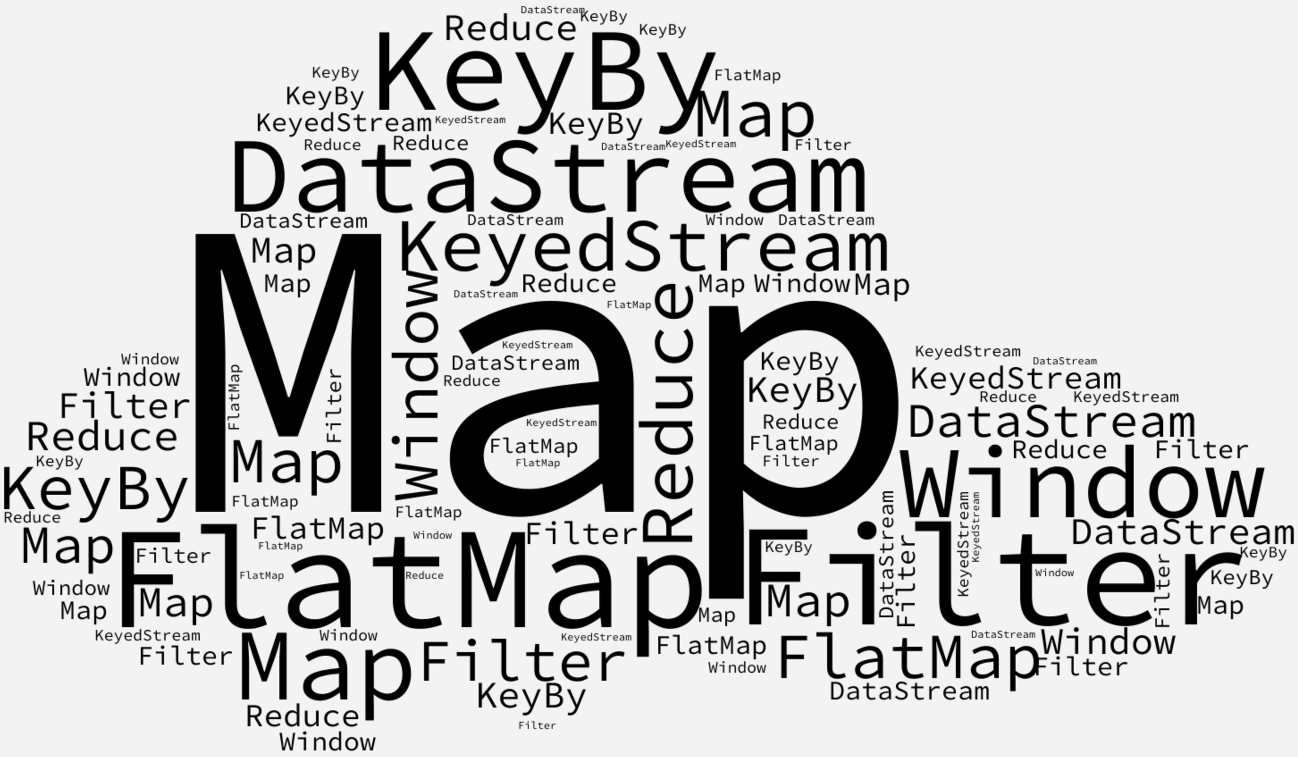
First, some of the records are read from Kafka topic through the Source. Then, these records are sent to the operators in Flink for computing. After that, the results are written to Kafka topic through the Sink. The Source and the Sink serve as the interfaces at both ends of the Flink job.


Flink Source
API
Files
readTextFile
readCSVFile
Sockets
socketTextStream
Collection
fromParallelCollection
fromCollection
BundleSource
Apache Kafka
JDBC
HBase
HIVE
RabbitMQ
ActiveMQ
Apache NiFI
Twitter API

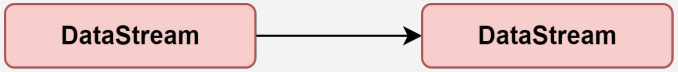

Flink Sink
API
Files
writeAsText
writeAsCSV
Sockets
writeToSocket
stdout
print
printToError
BundleSource
Apache Kafka
ElsaticSearch
Cassandra
Hadoop FileSystem
RabbitMQ
ActiveMQ
Apache NiFI
HBase
HBase
Akka

Operators

Operators transform one or more DataStreams into a new DataStream. Programs can combine multiple transformations into sophisticated dataflow topologies.



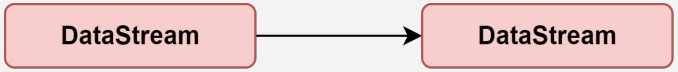
Map



Takes one element and produces one element. A map function that doubles the values of the input stream:

```
DataStream<Integer> dataStream = //...
dataStream.map(new MapFunction<Integer, Integer>() {
    @Override
    public Integer map(Integer value) throws Exception {
        return 2 * value;
    }
});
```

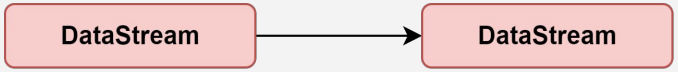
FlatMap



Takes one element and produces zero, one, or more elements. A flatmap function that splits sentences to words:

```
dataStream.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public void flatMap(String value, Collector<String> out)throws Exception {
        for(String word: value.split(" ")){
            out.collect(word);
        }
    }
});
```

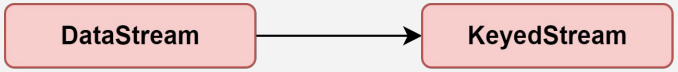
Filter



Evaluates a boolean function for each element and retains those for which the function returns true. A filter that filters out zero values:

```
dataStream.filter(new FilterFunction<Integer>() {
    @Override
    public boolean filter(Integer value) throws Exception {
        return value != 0;
    }
});
```

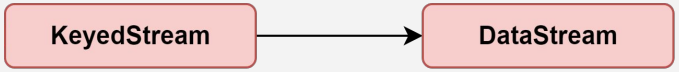
KeyBy



Logically partitions a stream into disjoint partitions. All records with the same key are assigned to the same partition. Internally, *keyBy()* is implemented with hash partitioning. There are different ways to specify keys.

```
dataStream.keyBy(value -> value.getSomeKey());
dataStream.keyBy(value -> value.f0);
```

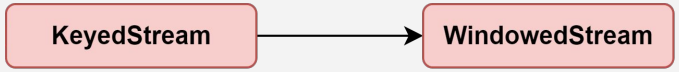
Reduce



A “rolling” reduce on a keyed data stream. Combines the current element with the last reduced value and emits the new value.

```
keyedStream.reduce(new ReduceFunction<Integer>() {
    @Override
    public Integer reduce(Integer value1, Integer value2)
    throws Exception {
        return value1 + value2;
    }
});
```

Window



Windows can be defined on already partitioned KeyedStreams. Windows group the data in each key according to some characteristic (e.g., the data that arrived within the last 5 seconds). See windows for a complete description of windows.

```
dataStream
  .keyBy(value -> value.f0)
  .window(TumblingEventTimeWindows.of(Time.seconds(5)));
```

Windowing is a technique that divides a stream of data into finite chunks, called windows, based on some criteria. For example, you can define windows based on time intervals (every 5 minutes), event counts (every 100 events), or session boundaries (when there is a gap of inactivity). Within each window, you can apply various operations on the data, such as aggregations (sum, average, count), transformations (map, filter, join), or complex business logic.

The type of window:

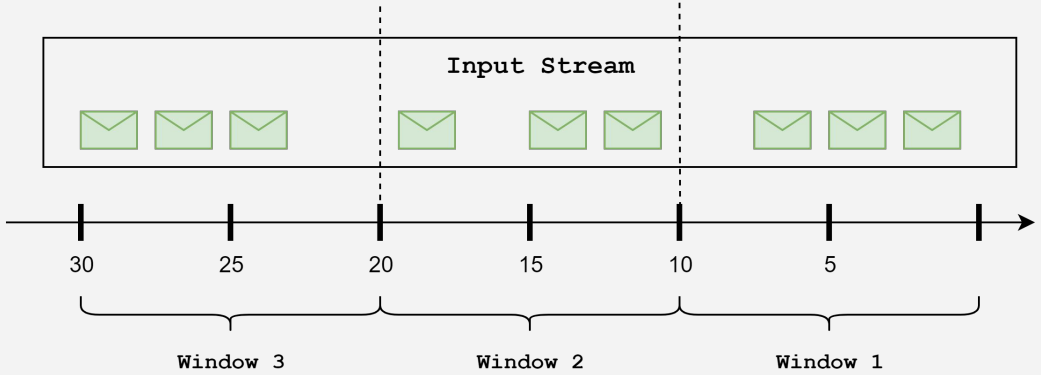
1. Tumbling window
2. Sliding window
3. Session window
4. Global window
5. User-defined window

Benefits of windowing

Windowing is a technique that divides a stream of data into finite chunks, called windows, based on some criteria. For example, we can define windows based on time (e.g., every 5 minutes), count (e.g., every 100 events), or session (e.g., until there is a gap of 10 minutes between events). Windowing allows us to apply aggregations, transformations, and other operations on each window, and produce intermediate or final results.

Tumbling window

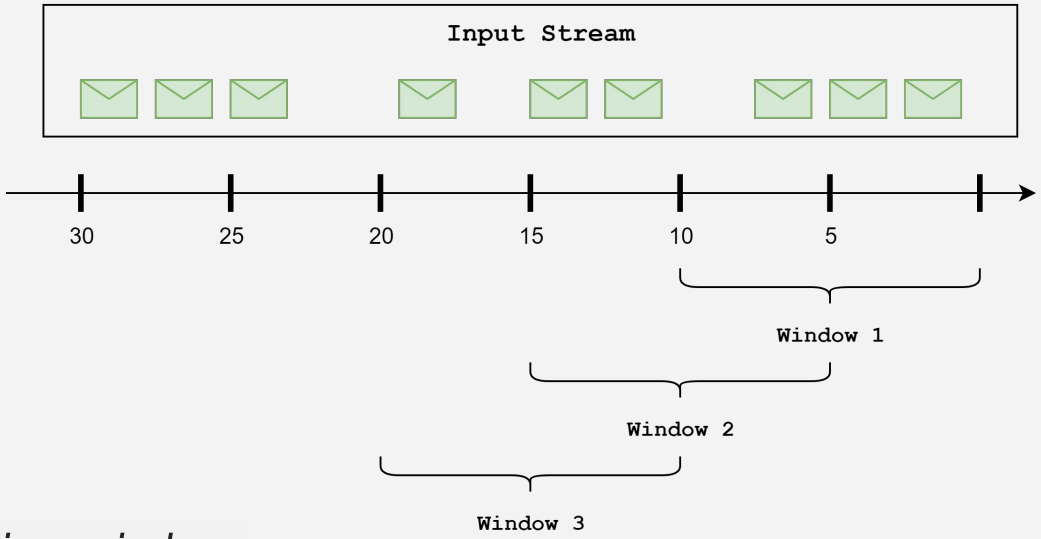
A tumbling window is an equal-sized, continuous and non-overlapping window. A tumbling window is defined with a window interval. For example as in fig a, if we have a tumbling window with a window interval 10 seconds, every incoming event from the data stream for a duration of 10 seconds will fall into the same window.




```
// tumbling event-time windows
stream.keyBy(<key selector>)
.window(TumblingEventTimeWindows.of(Time.seconds(5)))
```

Sliding window

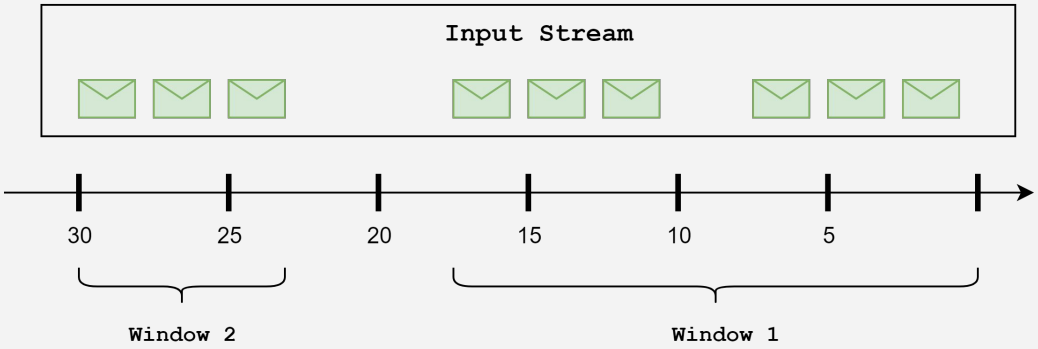
A sliding window is an overlapping window. A sliding window is defined with a window interval and a sliding offset. As we can see in figure b, the window size is 10 second which starts from 0–10 s. We have a sliding offset of 5 s and hence after 5 seconds, the window slides by 5 seconds and we get the second window in between 5–15 seconds.




```
// sliding event-time windows
stream.keyBy(<key selector>)
.window(SlidingEventTimeWindows.of(Time.seconds(10), Time.seconds(5)))
```

Session window

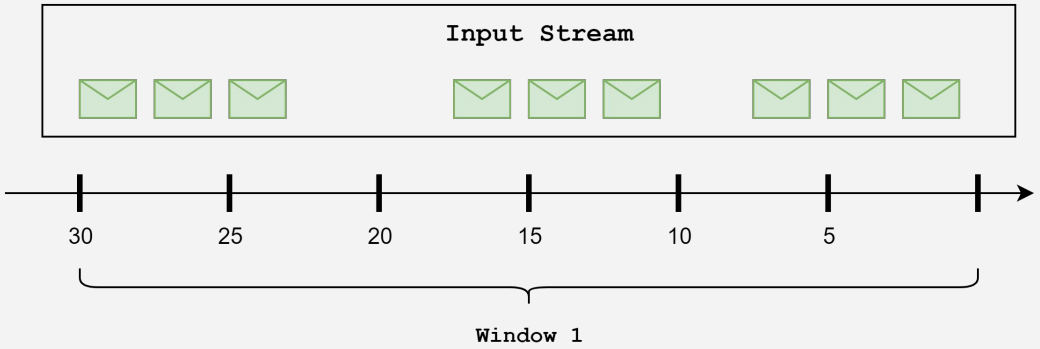
Session windows do not overlap and do not have a fixed start and end time. A session window assigner can be configured with either a static *session* gap or with a session gap extractor function which defines how long the period of inactivity is.




```
// session processing-time windows (static gap)
stream.keyBy(<key selector>).window(ProcessingTimeSessionWindows
.withGap(Time.seconds(5))) //session gap
```

Global window

A Global window considers the entire stream as a single window. As mentioned in figure, we will have all the events belonging to the same window.

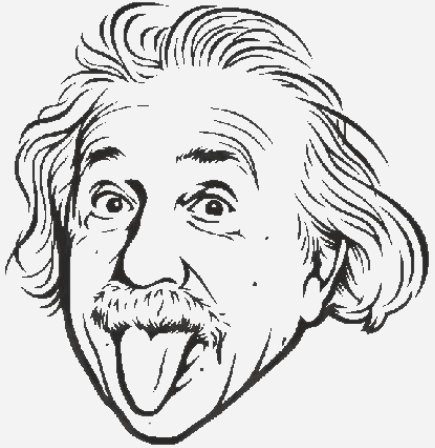



```
stream.keyBy(<key selector>)
.window(GlobalWindows.create())
```

User-defined window

This is a custom window defined by the user by extending the WindowAssigner class.

Education



THANK YOU

ANDERSEN

www.andersenlab.com