Stream Processing

Lecture 9

2018/2019

Table of Contents

- Continuous processing
  - **Apache Storm**
- Event-driven processing
  - Apache Flink

Continuous stream processing

- **Storm** : A decentralized continuous data processing system [designed at Twitter]
- **Trident** : High level framework for assembling Storm topologies (standing queries)

Concepts - topology

- *Topology* - a directed graph that embodies the logic for a realtime data-flow application
  - Vertices - represent a data computation
    - (spouts + bolts)
  - Edges - represent the data flow between components
  - Cycles are allowed
  - Runs continuously

Concepts: stream and tuples

- A **stream** is an unbounded sequence of tuples that is processed and created in parallel in a distributed fashion.
- Streams are defined with a schema that names the fields in the stream's **tuples**.

Concepts: spout

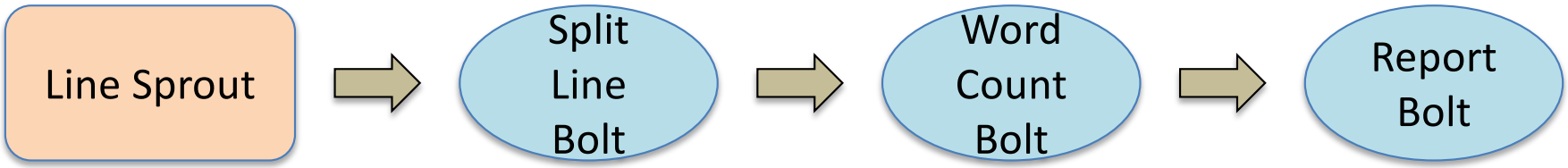
- A spout is a **source** of streams in a topology.
  - Spouts read tuples from an external source and emit tuples into the topology – e.g. Kafka queue.
  - can emit more than one stream [nextTuple]
  - must **not block** when asked to emit the next tuple
- Spouts can be **reliable** or **unreliable**.
  - A reliable spout is capable of replaying a tuple if its processing failed.
  - An unreliable spout forgets about the tuple as soon as it is emitted.

Concepts: bolts

- **Bolts** process incoming tuples and emit tuples to the next stage.
  - In Storm, all processing is done in bolts.
- Bolts can do anything from filtering, functions, aggregations, joins, talking to databases, and more.
- Doing complex stream transformations often requires multiple steps and thus multiple bolts.
- Inputs are **explicitly** declared by subscribing streams emitted by other components (spouts, bolts).

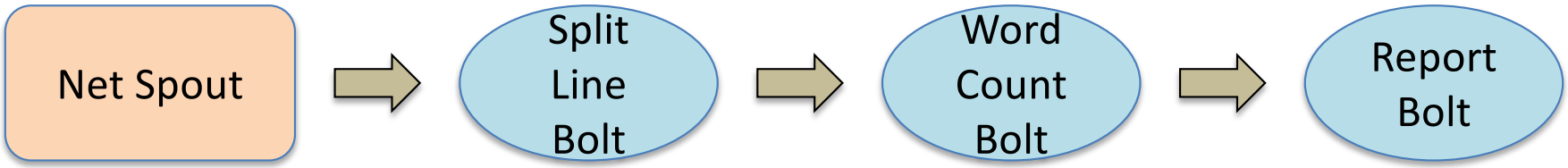
Example: word count

- Logical topology

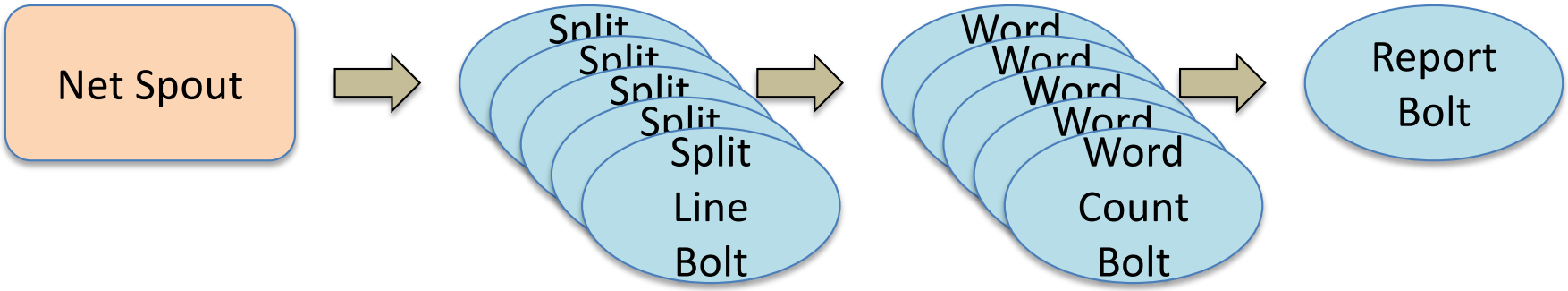


Example: word count

- Logical topology



- Physical topology



Storm execution

- *Tasks* - an instance of a spout or bolt; the level of parallelism is determined by the number of tasks chosen/hinted;
- A stream grouping defines how that stream should be partitioned among the bolt's tasks.

Stream grouping (partial)

- **Shuffle grouping**: Tuples are randomly distributed across the bolt's tasks.
- **Fields grouping**: The stream is partitioned by the fields specified in the grouping. E.g.: if the stream is grouped by the ”word" field, tuples with the same ”word" will go to the same task.
- **All grouping**: The stream is replicated across all the bolt's tasks.
- **Global grouping**: The entire stream goes to a single one of the bolt's tasks. Specifically, it goes to the task with the lowest id.

Storm: Reliability (1)

- Every spout tuple will be fully processed by the topology
  - tuples are tracked across the topology
  - a timeout causes a spout tuple to be replayed
  - the application/bolt must manage dependencies among tuples, upon emitting.
  - the application/bolt must **ack** the tuples to signal they have been fully processed

Storm: Reliability (2)

- Storm provides two basic tuple processing semantics under failures

  - **at least once** - a tuple is guaranteed to be processed at least once; **idempotence** must be implemented by the application

    - Storm keeps a XOR checksum that tracks progress of the tuple within the topology (see paper)

  - **at most once** (best effort) - a tuple is processed once or, not at all, in case of failure

  - also available in later versions [**Trident**]

  - **exactly once** - expensive; provided via *transactional-topologies* and Trident; involves tuple batches and issuing "DB" transactions.

Storm: Reliability (3)

- Stateless bolts forget everything upon a failure
  - – - replaying lost tuples is not enough for correct results
    - (consider sliding windows, moving averages, etc)
  - – - there is no automatic provision for checkpointing or to recreate state prior to the failure
  - – Stateful topologies can be achieved via Trident; or interfacing with external (in-memory) databases (ad hoc)

Example (1)

```
public class NetSpout extends BaseRichSpout {
        private BufferedReader reader;
        private SpoutOutputCollector collector;
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
                declarer.declare(new Fields(”line"));
        }
        public void open(Map config, TopologyContext context, SpoutOutputCollector collector) {
                this.collector = collector;
                reader = new BufferedReader( new InputStreamReader( new Socket(“localhost”,7070)));
        }
        public void nextTuple() {
                try {
                        this.collector.emit( reader.readLine());
                } catch( IOException e) {
                        // do nothing
                }
        }
}
```

Example (2)

```
public class SplitSentenceBolt extends BaseRichBolt{
        private OutputCollector collector;
        public void prepare(Map config, TopologyContext context, OutputCollector collector) {
                this.collector = collector;
        }
        public void execute(Tuple tuple) {
                String line = tuple.getStringByField(”line");
                String[] words = line.split(" ");
                for(String word : words){
                        this.collector.emit(new Values(word));
                }
        }
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
                declarer.declare(new Fields("word"));
        }
}
```

Example (3)

```
public class WordCountBolt extends BaseRichBolt{
	private OutputCollector collector;
	private HashMap<String, Long> counts = null;
	public void prepare(Map config, TopologyContext context, OutputCollector collector) {
		this.collector = collector;
		this.counts = new HashMap<String, Long>();
	}
	public void execute(Tuple tuple) {
		String word = tuple.getStringByField("word");
		Long count = this.counts.get(word);
		if(count == null){
			count = 0L;
		}
		count++;
		this.counts.put(word, count);
		this.collector.emit(new Values(word, count));
	}
	public void declareOutputFields(OutputFieldsDeclarer declarer) {
		declarer.declare(new Fields("word", "count"));
	}
}
```

Example (4)

```
public class ReportBolt extends BaseRichBolt {
	private HashMap<String, Long> counts = null;
	public void prepare(Map config, TopologyContext context, OutputCollector collector) {
		this.counts = new HashMap<String, Long>();
	}
	public void execute(Tuple tuple) {
		String word = tuple.getStringByField("word");
		Long count = tuple.getLongByField("count");
		this.counts.put(word, count);
	}
	public void declareOutputFields(OutputFieldsDeclarer declarer) {
		// this bolt does not emit anything
	}
	public void cleanup() {
		System.out.println("--- FINAL COUNTS ---");
		List<String> keys = new ArrayList<String>();
		keys.addAll(this.counts.keySet());
		Collections.sort(keys);
		for (String key : keys) {
			System.out.println(key + " : " + this.counts.get(key));
		}
	}
}
```

Example (5)

```
NetSpout spout = new NetSpout();
SplitLineBolt splitBolt = new SplitLineBolt();
WordCountBolt countBolt = new WordCountBolt();
ReportBolt reportBolt = new ReportBolt();

TopologyBuilder builder = new TopologyBuilder();
builder.setSpout(“net-spout”, spout);
        // NetSpout --> SplitLineBolt
builder.setBolt("split-bolt", splitBolt).shuffleGrouping(“net-spout”);
        // SplitLineBolt --> WordCountBolt
builder.setBolt("count-bolt", countBolt).fieldsGrouping("split-bolt", new Fields("word"));
        // WordCountBolt --> ReportBolt
builder.setBolt("report-bolt", reportBolt).globalGrouping("count-bolt");
Config config = new Config();
LocalCluster cluster = new LocalCluster();
cluster.submitTopology(TOPOLOGY_NAME, new Config(), builder.createTopology());
```

Trident

- A high-level abstraction for doing realtime computing on top of Storm
  - (much like Pig is for MapReduce)
  - extends Storm with primitives for doing **stateful**, **incremental processing** on top of any database or persistence store
  - **exactly-once semantics**, under failures

Trident : execution

- Trident topologies compile to efficient Storm topologies
  - the goal is to **avoid** unnecessary network **shuffles**

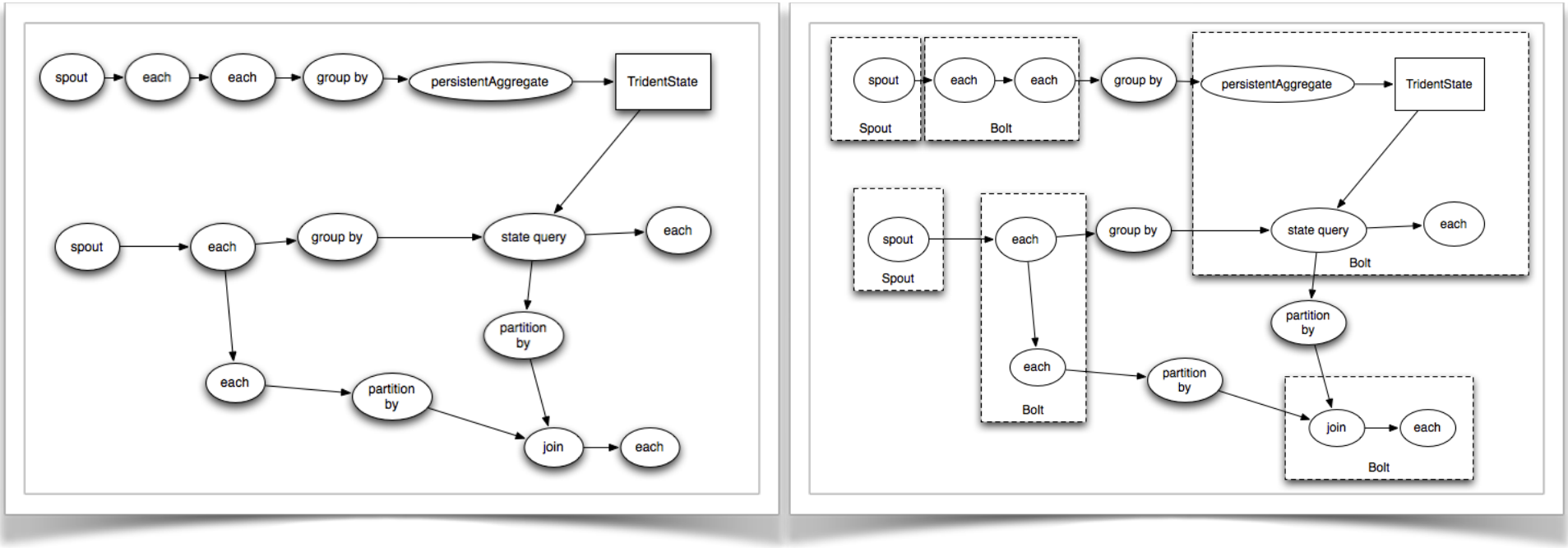


Table of Contents

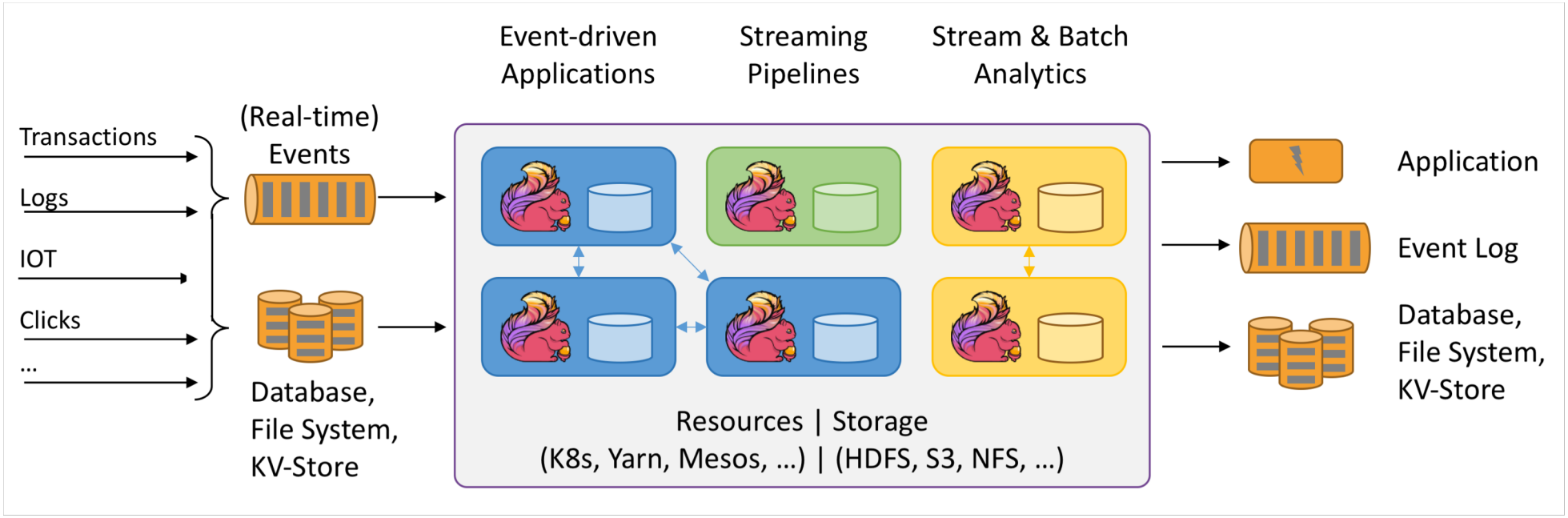
- Continuous processing
  - Apache Storm
- Event-driven processing
  - **Apache Flink**

Apache Flink

- Data is often produced as a stream of events. Data can be processed as *unbounded* or *bounded* streams.
- **Unbounded streams** have a start but no defined end.
  - Do not terminate and provide data as it is generated;
  - Must be continuously processed;
  - It is not possible to wait for all input data to arrive.
- **Bounded streams** have a defined start and end.
  - Can be processed by ingesting all data before performing any computations;
  - Ordered ingestion is not required, as a bounded data set can always be sorted.
  - Processing of bounded streams is also known as batch processing.

Apache Flink (2)

- Apache Flink is an open source platform for distributed stream processing.

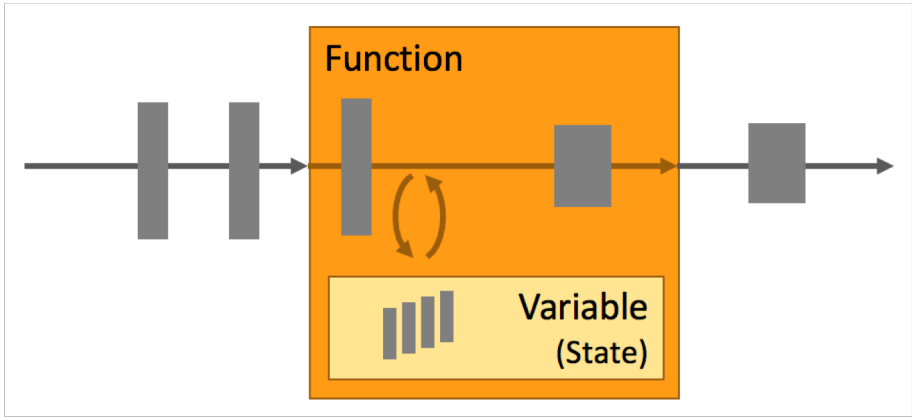


Concepts: streams

- Flink can handle any kind of stream.
- **Bounded** and **unbounded** streams: Streams can be unbounded or bounded, i.e., fixed-sized data sets.
- **Real-time** and **recorded** streams: All data are generated as streams. There are two ways to process the data. Processing it in real-time as it is generated or persisting the stream to a storage system and processed it later.

State

- Every non-trivial streaming application is stateful.
- Any application that runs basic business logic needs to remember events or intermediate results.



State (2)

- **Multiple State Primitives**: Flink supports different data types.
- **Pluggable State Backends**: Application state is managed in and checkpointed by a pluggable state backend.
- **Exactly-once state consistency**: Flink's checkpointing and recovery algorithms guarantee the consistency of application state in case of a failure.
- **Very Large State**: Flink is able to maintain application state of several terabytes due to its asynchronous and incremental checkpoint algorithm.
- **Scalable Applications**: Scales stateful applications by redistributing the state to more or fewer workers.

Time

- **Event-time Mode**: Applications that process streams based on timestamps of the events.
- **Processing-time Mode**: Applications that performs computations as triggered by the wall-clock time of the processing machine.

- **Watermark Support**: Flink employs watermarks to reason about time in event-time applications.
- **Late Data Handling**: Flink features multiple options to handle late events, such as rerouting them via side outputs and updating previously completed results.

Programming model: abstraction

- **Stream** is a (potentially never-ending) flow of data records
- **Transformation** is an operation that takes one or more streams as input, and produces one or more output streams as a result.

Programming model: execution

- Programs are mapped to **streaming dataflows**, consisting of **streams** and transformation **operators**.
- Each dataflow starts with one or more **sources** and ends in one or more **sinks**.
- The dataflows resemble arbitrary **directed acyclic graphs***(DAGs)*.

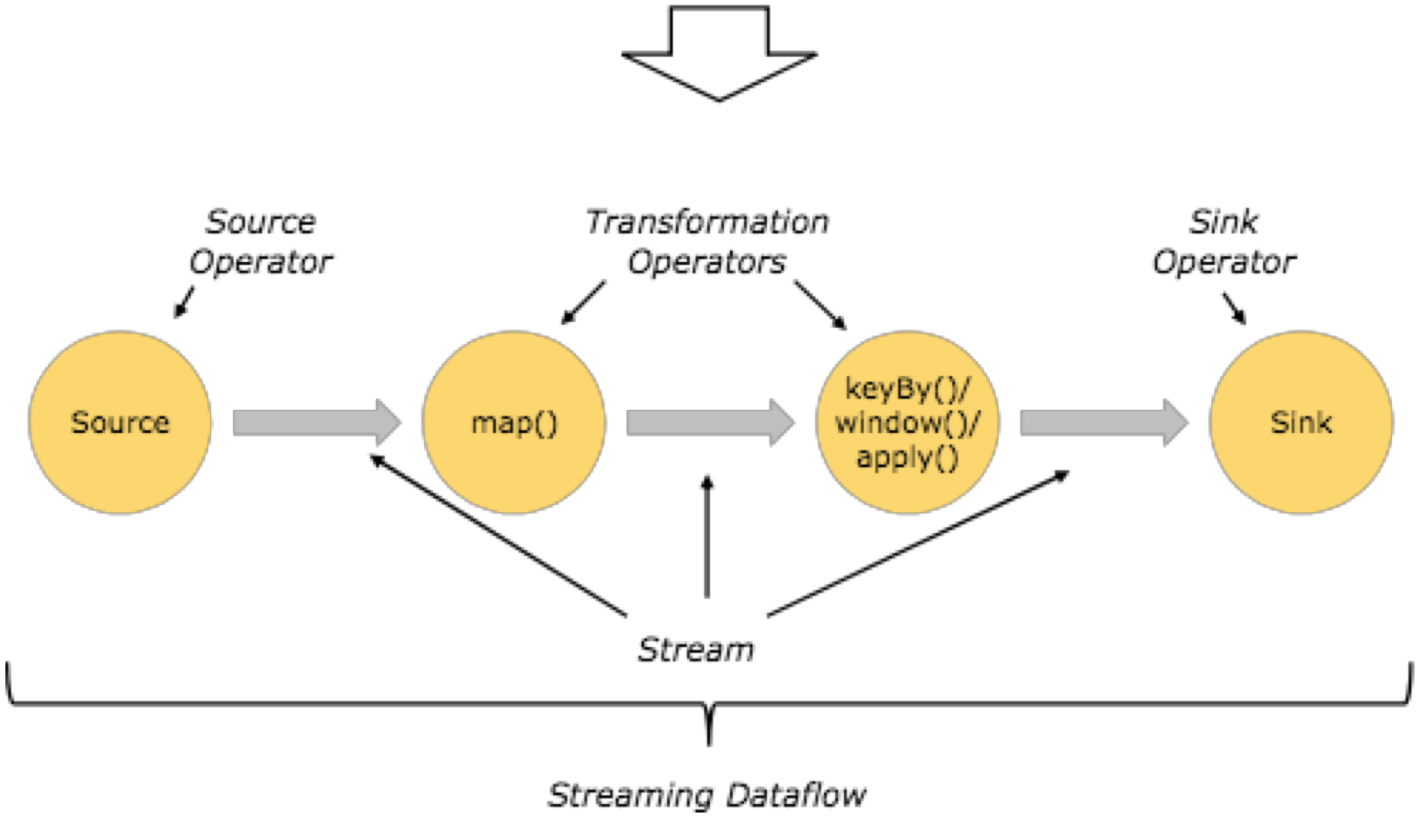
```
DataStream<String> lines = env.addSource(
                                new FlinkKafkaConsumer<>(…));

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

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

stats.addSink(new RollingSink(path));
```

Source

Transformation

Transformation

Sink

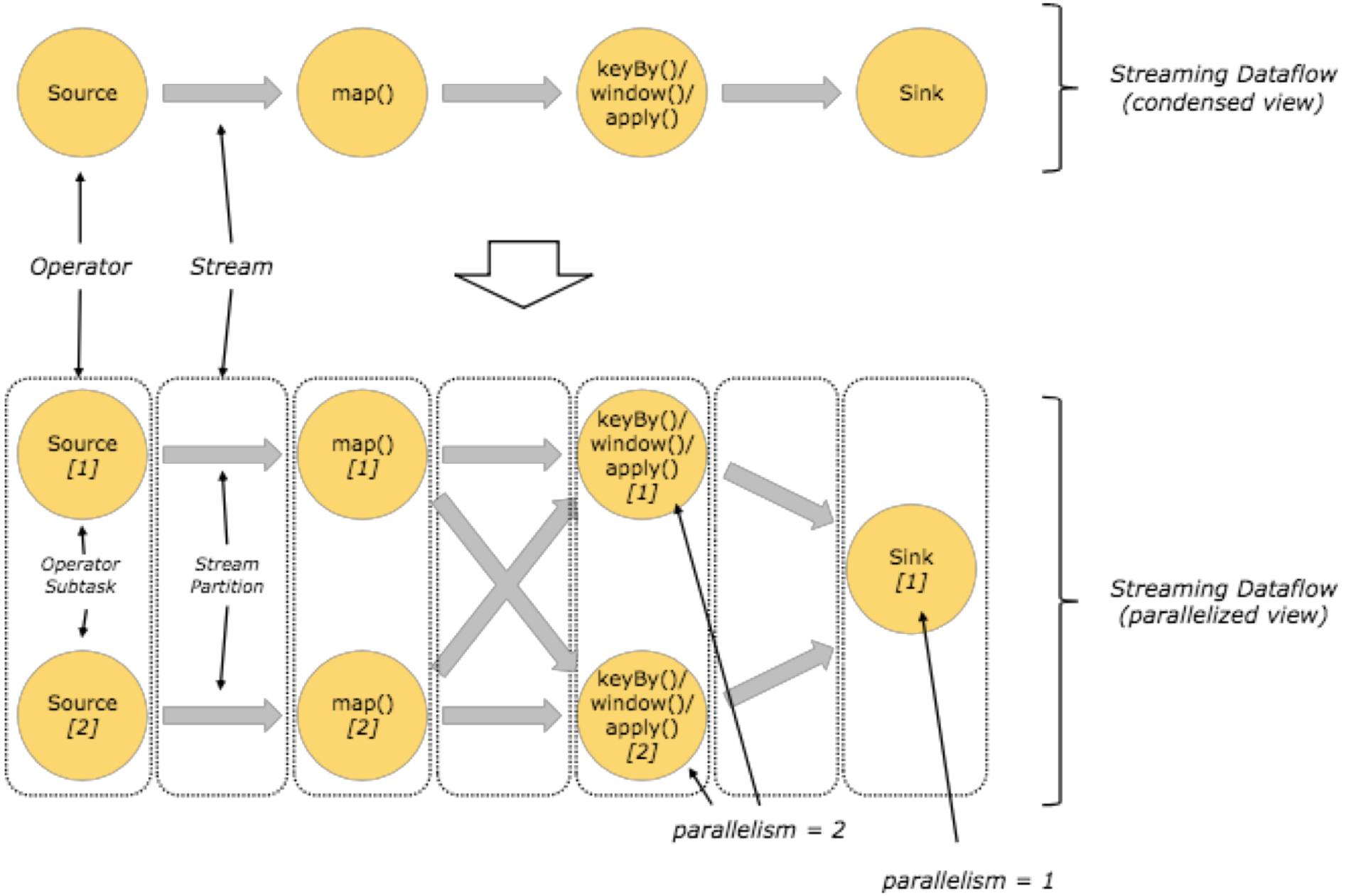


Parallel dataflows

- During execution, a *stream* has one or more **stream partitions**, and each *operator* has one or more **operator subtasks**.
- The operator subtasks are independent and execute in different threads/machines.
- The number of operator subtasks is the **parallelism** of the operator. The parallelism of a stream is always that of its producing operator. Different operators of the same program may have different levels of parallelism.

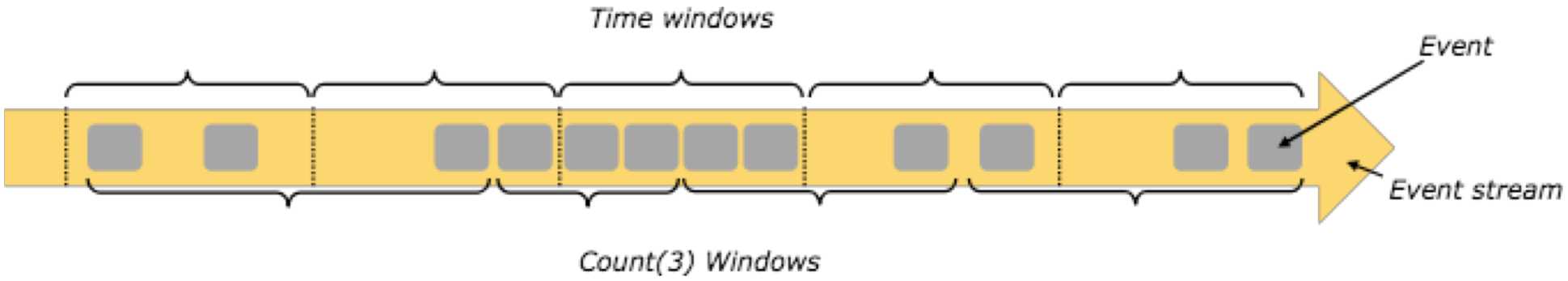
Parallel dataflows (cont.)

- Communication between operators
  - One-to-one
  - Redistributing


Source
map()
keyBy()/
window()/
apply()
Sink
Streaming Dataflow
(condensed view)
Operator
Stream
Source
[1]
Source
[2]
map()
[1]
map()
[2]
keyBy()/
window()/
apply()
[1]
keyBy()/
window()/
apply()
[2]
Sink
[1]
Operator
Subtask
Stream
Partition
Streaming Dataflow
(parallelized view)
parallelism = 2
parallelism = 1

Windows

- Windows allow to aggregate events from a defined time-period.

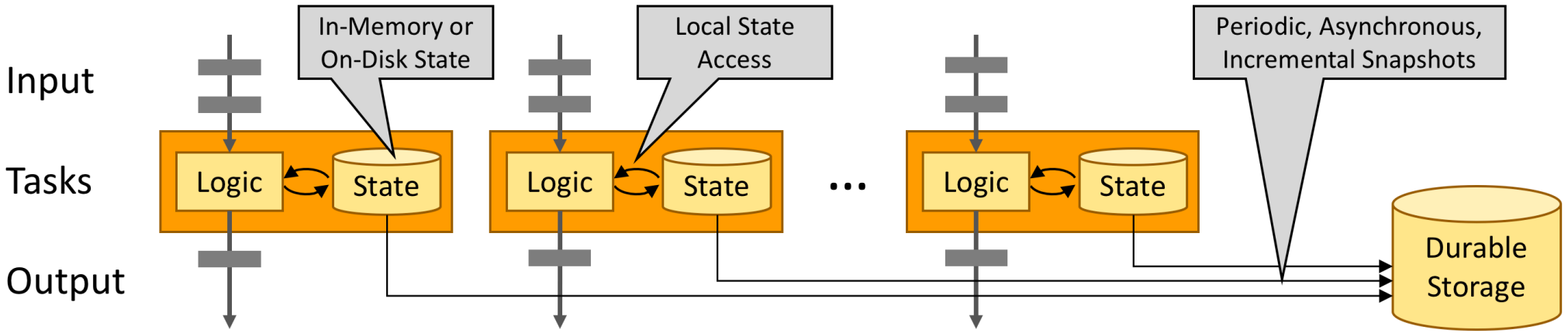


Example

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime)

// alternatively:
// env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime)
// env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

val stream: DataStream[MyEvent] = env.addSource(new FlinkKafkaConsumer09[MyEvent](topic, schema, props))

stream
    .keyBy( _.getUser )
    .timeWindow(Time.hours(1))
    .reduce( (a, b) => a.add(b) )
    .addSink(...)
```

Local State

- Task state is maintained in memory or, if the state size exceeds the available memory, in access-efficient on-disk data structures.
- Hence, tasks perform all computations by accessing local, often in-memory, state yielding very low processing latencies.



Local State (2)

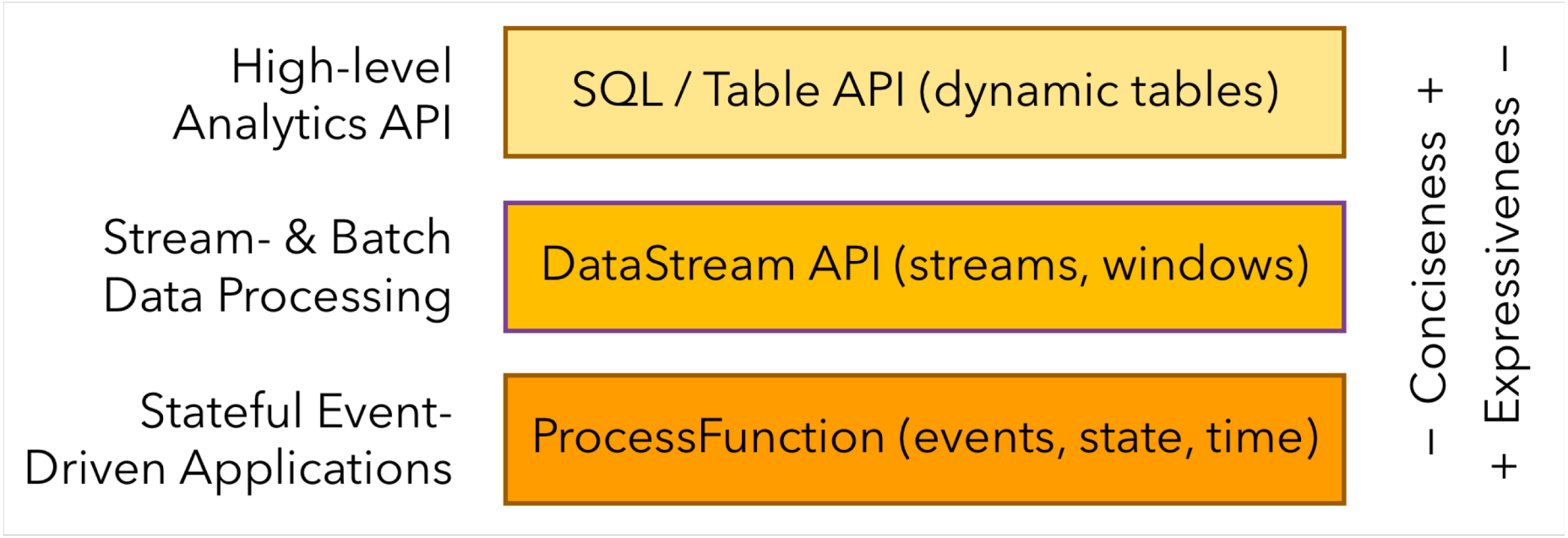
- Flink guarantees exactly-once state consistency in case of failures by periodically and asynchronously checkpointing the local state to durable storage.

Fault tolerance

- Combines **stream replay** and **checkpointing**.
- A checkpoint is related to a specific point in each of the input streams along with the corresponding state for each of the operators.
- A streaming dataflow can be resumed from a checkpoint while maintaining consistency *(exactly-once processing semantics)* by restoring the state of the operators and replaying the events from the point of the checkpoint.

Layered architecture

- Possible to program using different levels of abstractions



ProcessFunction

- ProcessFunctions are the most expressive interfaces.
  - Process individual events from input streams or window
  - Fine-grain control over time
  - Can arbitrarily modify its state and register timers that trigger a callback

ProcessFunction (example)

```
/** Called for each processed event. */
  @Override
  public void processElement( Tuple2<String, String> in, Context ctx,
                              Collector<Tuple2<String, Long>> out) throws Exception {
    switch (in.f1) {
     case "START":            // set the start time if we receive a start event.
       startTime.update(ctx.timestamp());
       break;
     case "END":              // emit the duration between start and end event
       Long sTime = startTime.value();
       if (sTime != null) {
         out.collect(Tuple2.of(in.f0, ctx.timestamp() - sTime));
         startTime.clear();
       }
    }
  }
```

DataStreamAPI: get environment

- The execution environment is used to control the execution of the programs

```
StreamExecutionEnvironment env =
    StreamExecutionEnvironment.getExecutionEnvironment();
…
env.execute("Window WordCount");
```

DataStreamAPI: data sources

- File-based:
  - **readTextFile(path)** - Reads text files, line-by-line and returns them as Strings.
  - **readFile(fileInputFormat, path)** - Reads (once) files as dictated by the specified file input format.
  - …
- Socket-based:
  - **socketTextStream(host, port)** - Reads from a socket. Elements can be separated by a delimiter.
- Collection-based:
  - **fromCollection(Collection)** - Creates a data stream from the Java Java.util.Collection.
  - **fromCollection(Iterator, Class**) - Creates a data stream from an iterator. The class specifies the data type of the elements returned by the iterator.
  - …
- Custom:
  - **addSource** - Attach a new source function.

DataStreamAPI: data sinks

- **writeAsText()** - Writes elements line-wise as Strings.
- **writeAsCsv(...)** - Writes tuples as comma-separated value files.
- **print()** / **printToErr()** - Prints the *toString()* value of each element on the standard out / standard error stream.
- **writeToSocket** - Writes elements to a socket according to a SerializationSchema
- **addSink** - Invokes a custom sink function. Flink comes bundled with connectors to other systems (such as Apache Kafka) that are implemented as sink functions.

DataStreamAPI: iterations

- Iterative streaming programs implement a step function and embed it into an IterativeStream

```
DataStream<Long> someIntegers = env.generateSequence(0, 1000);
IterativeStream<Long> iteration = someIntegers.iterate();

DataStream<Long> minusOne = iteration.map(new MapFunction<Long, Long>() {
    public Long map(Long value) throws Exception {
        return value - 1 ;
}  });

DataStream<Long> stillGreaterThanZero = minusOne.filter(new FilterFunction<Long>()
{
  @Override
  public boolean filter(Long value) throws Exception {
    return (value > 0);
} });
```

DataStreamAPI: loops

- It is possible to fed a stream into an iteration using the method **closeWith**. This allows to execute loops in computations. What is this good for?

```
DataStream<Long> someIntegers = env.generateSequence(0, 1000);
IterativeStream<Long> iteration = someIntegers.iterate();

DataStream<Long> minusOne = iteration.map(new MapFunction<Long,
Long>() {…});

DataStream<Long> stillGreaterThanZero = minusOne.filter(new
FilterFunction<Long>(){…});

iteration.closeWith(stillGreaterThanZero);
```

DataStreamsAPI: control latency

- By default, elements are buffered and transmitted when buffers are full. It is possible to force transmition after some time by using **env.setBufferTimeout(timeoutMillis)**

```
LocalStreamEnvironment env =
StreamExecutionEnvironment.createLocalEnvironment();
env.setBufferTimeout(timeoutMillis);

env.generateSequence(1,10).map(new MyMapper())
        .setBufferTimeout(timeoutMillis);
```

DataStreamsAPI: transformations

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

DataStreamsAPI: transformations

```
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);
        }
    }
});
```

DataStreamsAPI: transformations

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

DataStreamsAPI: transformations

Logically partitions a stream into disjoint partitions.

```
dataStream.keyBy("someKey") // Key by field "someKey"
dataStream.keyBy(0) // Key by the first element of a Tuple
```

DataStreamsAPI: transformations

A "rolling" reduce on a keyed data stream.

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

DataStreamsAPI: transformations

A "rolling" fold on a keyed data stream with an initial value.

```
DataStream<String> result =
  keyedStream.fold("start", new FoldFunction<Integer, String>() {
    @Override
    public String fold(String current, Integer value) {
        return current + "-" + value;
    }
  });
```

DataStreamsAPI: transformations

Rolling aggregations on a keyed data stream.

```
keyedStream.sum(0);
keyedStream.sum("key");
keyedStream.min(0);
keyedStream.min("key");
keyedStream.max(0);
keyedStream.max("key");
keyedStream.minBy(0);
keyedStream.minBy("key");
keyedStream.maxBy(0);
keyedStream.maxBy("key");
```

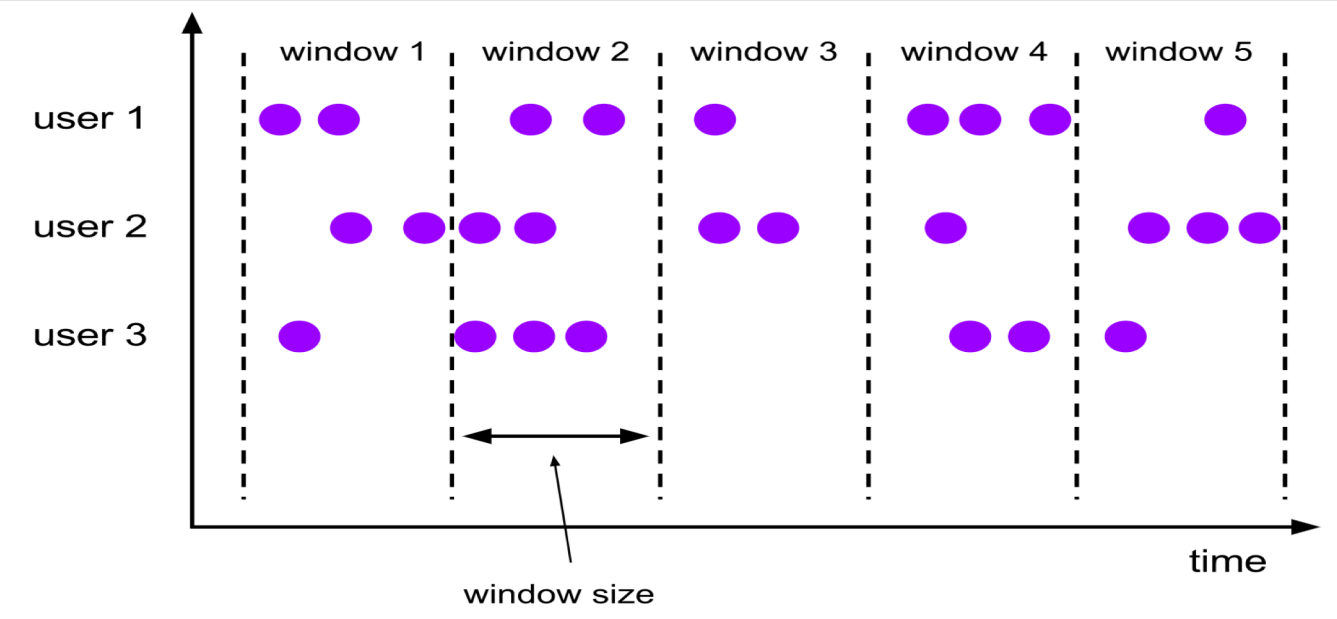
DataStreamAPI (example)

```
public class WindowWordCount {
   public static void main(String[] args) throws Exception {
      StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

      DataStream<Tuple2<String, Integer>> dataStream = env
            .socketTextStream("localhost", 9999)
            .flatMap(new Splitter())
            .keyBy(0)
            .timeWindow(Time.seconds(5))
            .sum(1);

      dataStream.print();

      env.execute("Window WordCount");
   }

   public static class Splitter implements FlatMapFunction<String, Tuple2<String, Integer>> {
      @Override
      public void flatMap(String sentence, Collector<Tuple2<String, Integer>> out) throws Exception {
         for (String word: sentence.split(" ")) {
            out.collect(new Tuple2<String, Integer>(word, 1));
} } }   }
```

Windows

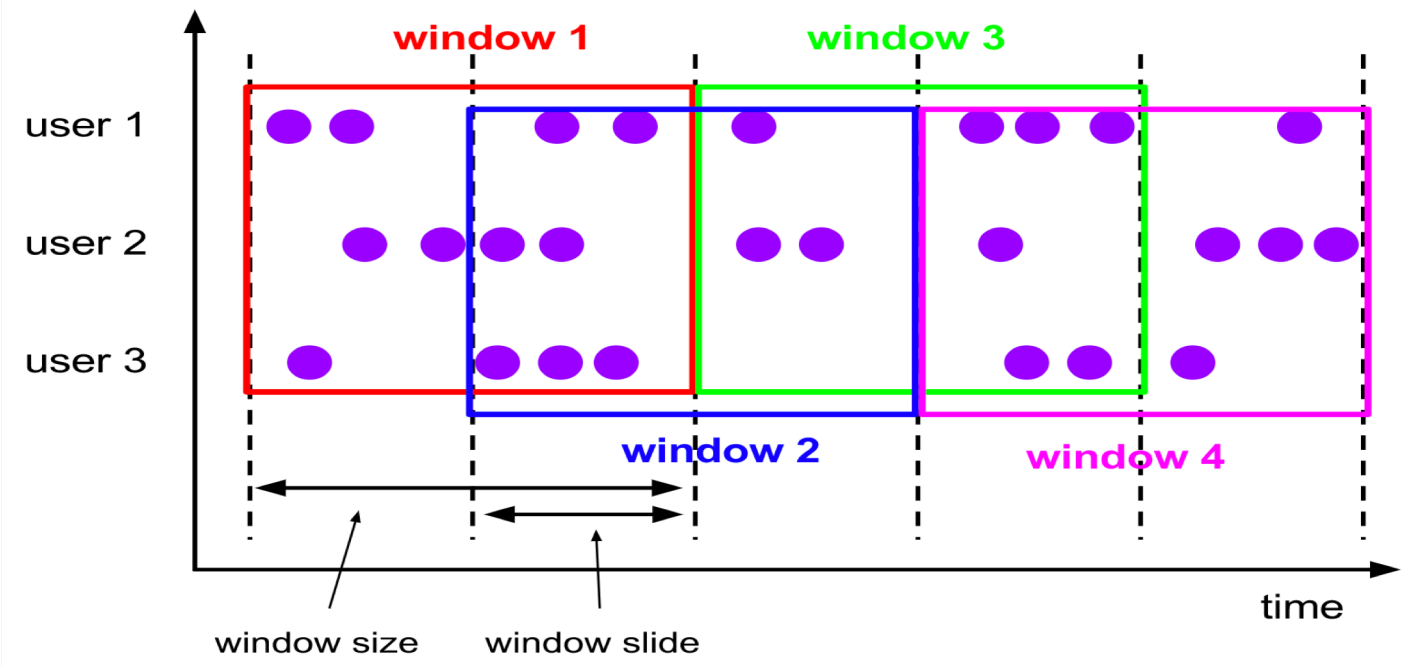
- A *tumbling windows* assigner assigns each element to a window of a specified *window size*.

```
dataStream.keyBy(0).window(
    TumblingEventTimeWindows.of(Time.seconds(5)));
```



Windows

- The *sliding windows* assigner assigns elements to windows of fixed length.dataStream.keyBy(0).window(SlidingEventTimeWindows.of(Time.seconds(10), Time.seconds(5)));



DataStreamsAPI: transformations

Aggregates the contents of a window.

```
windowedStream.sum(0);
windowedStream.sum("key");
windowedStream.min(0);
windowedStream.min("key");
windowedStream.max(0);
windowedStream.max("key");
windowedStream.minBy(0);
windowedStream.minBy("key");
windowedStream.maxBy(0);
windowedStream.maxBy("key");
```

Other windows operations

- Reduce
- Fold
- Apply : apply generic function

DataStreamsAPI: transformations

- Join two data streams on a given key and a common window.

```
dataStream.join(otherStream)
   .where(<key selector>).equalTo(<key selector>)
   .window(TumblingEventTimeWindows.of(Time.seconds(3)))
   .apply (new JoinFunction () {...});
```

Table API

- Start environment

```
import org.apache.flink.table.api.java.StreamTableEnvironment;

StreamExecutionEnvironment sEnv =
StreamExecutionEnvironment.getExecutionEnvironment();
// create a TableEnvironment for streaming queries
StreamTableEnvironment sTableEnv = StreamTableEnvironment.create(sEnv);
```

Table API

- Register a table source

```
// create a TableSource
TableSource csvSource = new CsvTableSource("/path/to/file", ...);

// register the TableSource as table "CsvTable"
tableEnv.registerTableSource("CsvTable", csvSource);
```

Table API

- Execute operation on table

```
// scan registered Orders table
Table orders = tableEnv.scan("Orders");
// compute revenue for all customers from France
Table revenue = orders
  .filter("cCountry === 'FRANCE'")
  .groupBy("cID, cName")
  .select("cID, cName, revenue.sum AS revSum");
```

Table API: SQL

- Use SQL

```
// compute revenue for all customers from France
Table revenue = tableEnv.sqlQuery(
    "SELECT cID, cName, SUM(revenue) AS revSum " +
    "FROM Orders " +
    "WHERE cCountry = 'FRANCE' " +
    "GROUP BY cID, cName"
  );
```

Bibliography

- Apache Flink: Stream and batch processing in a single engine. P. Carbone, et. al. IEEE Data Engineering Bulletin. 38.
- https://flink.apache.org/
- https://storm.apache.org/
- Storm @Twitter. Ankit Toshniwal, et. al. Sigmod'14.

- Acknowledgments: some images in this slides deck are from Flink and Storm sites.