Stream Processing

Lecture 5

2022/2023

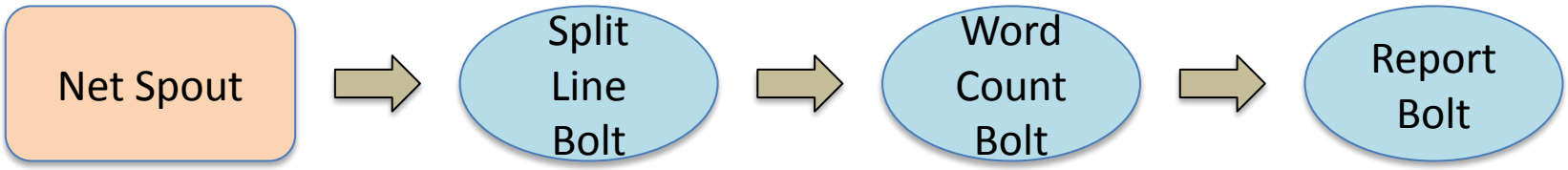
Table of Contents

- Continuous processing
  - **Apache Storm**
  - Spark Streaming
  - 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

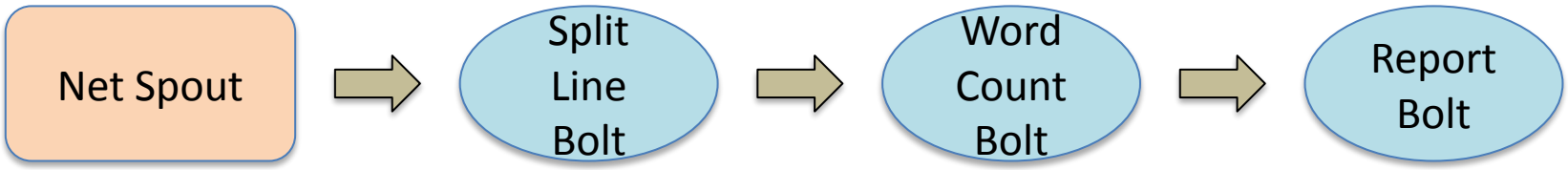


- *Topology* - a directed graph that embodies the logic for a realtime data-flow application
  - Vertices - represent a data computation
    - (spouts + bolts)
  - Edges - represent the data flow between components
  - Cycles are allowed
  - Runs continuously

Concepts: stream and tuples

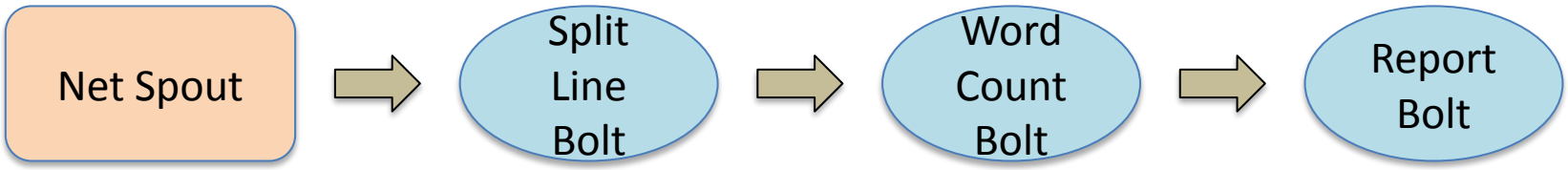
- A **stream** is an unbounded sequence of tuples that is processed and created in parallel in a distributed fashion.
- Streams are defined with a schema that names the fields in the stream's **tuples**.

Concepts: spout



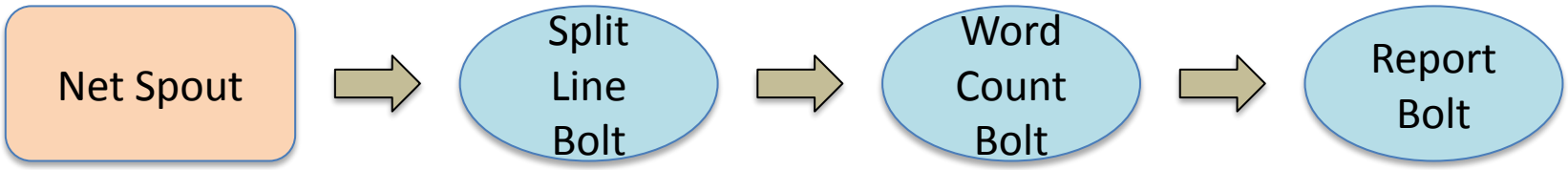
- A spout is a **source** of streams in a topology.
  - Spouts read tuples from an external source and emit tuples into the topology – e.g. Kafka queue.
  - can emit more than one stream [nextTuple]
  - must **not block** when asked to emit the next tuple

Concepts: spout



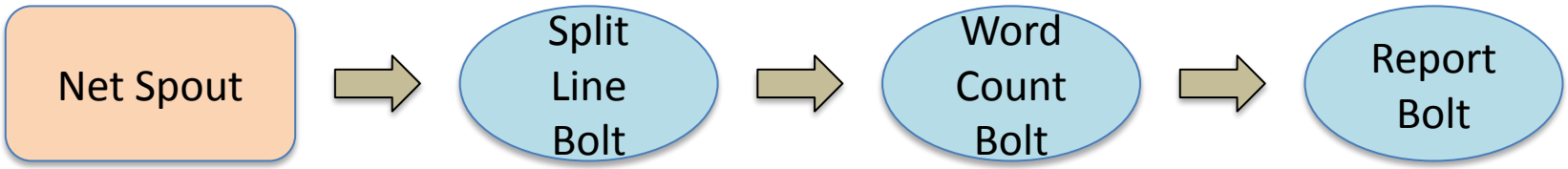
- Spouts can be **reliable** or **unreliable**.
  - A reliable spout is capable of replaying a tuple if its processing failed.
  - An unreliable spout forgets about the tuple as soon as it is emitted.

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.

Concepts: bolts



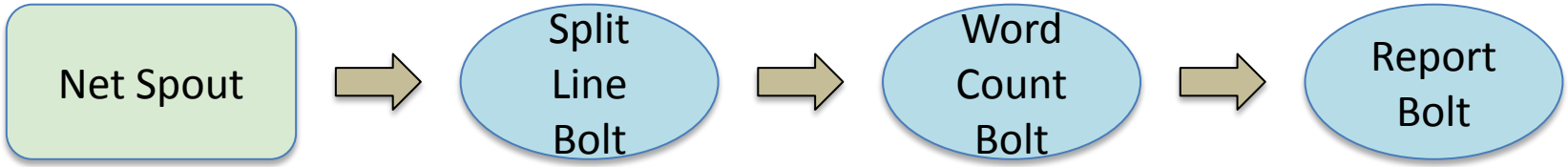
- Doing complex stream transformations often requires multiple steps and thus multiple bolts.
- Inputs are **explicitly** declared by subscribing streams emitted by other components (spouts, bolts).

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.
- The logical topology is mapped into a physical topology composed of multiple tasks.

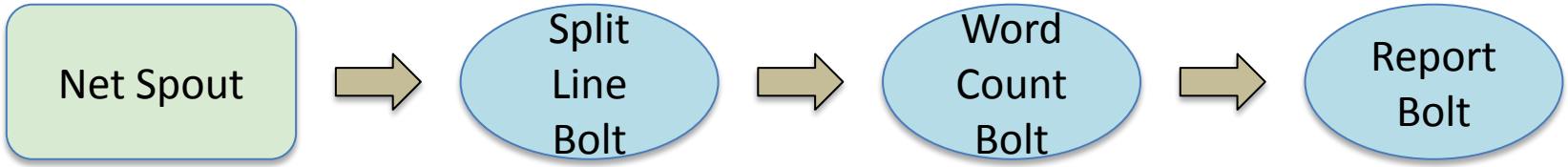
Example: word count

- Logical topology

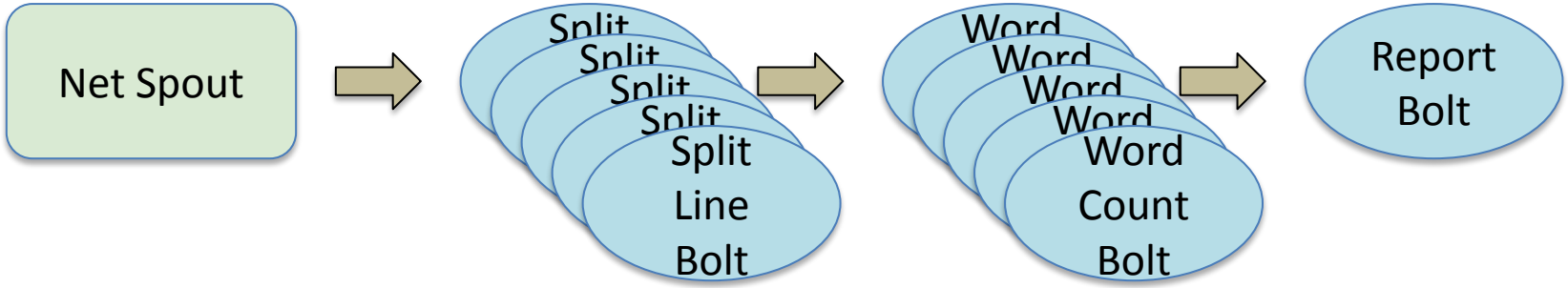


Example: word count

- Logical topology



- Physical topology



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

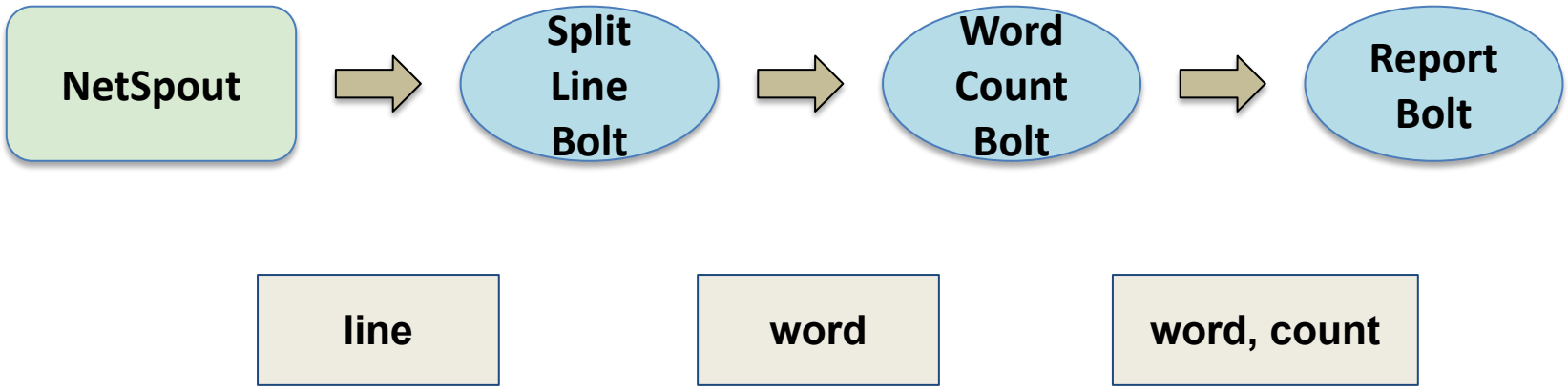
Storm: Reliability (2)

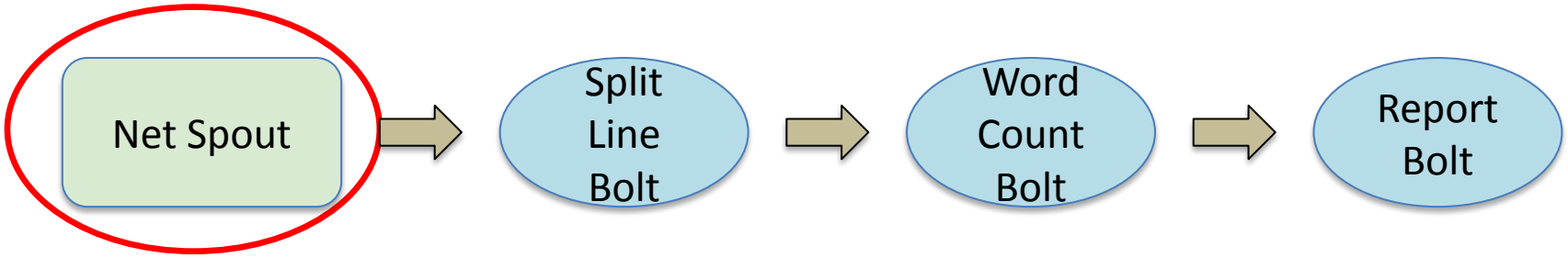
- Storm provides two basic tuple processing semantics under failures
  - **at least once** - a tuple is guaranteed to be processed at least once; **idempotence** must be implemented by the application
  - **at most once** (best effort) - a tuple is processed once or, not at all, in case of failure
  - 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)

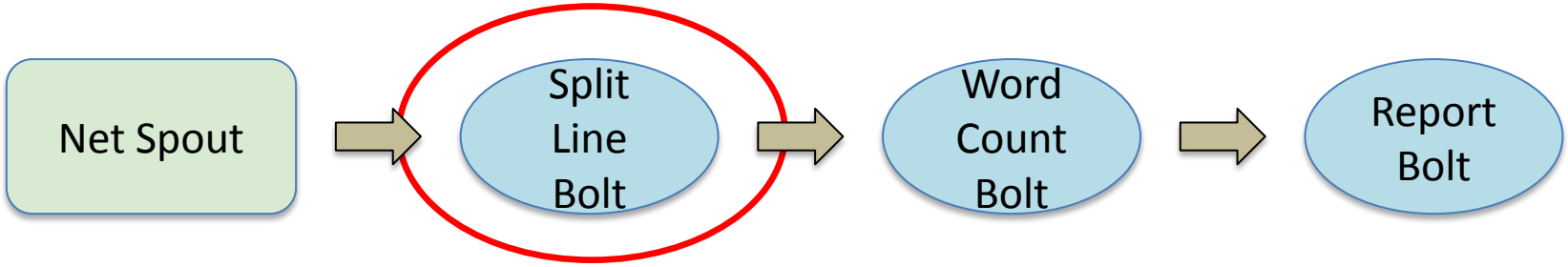
Programming Example (WordCount)



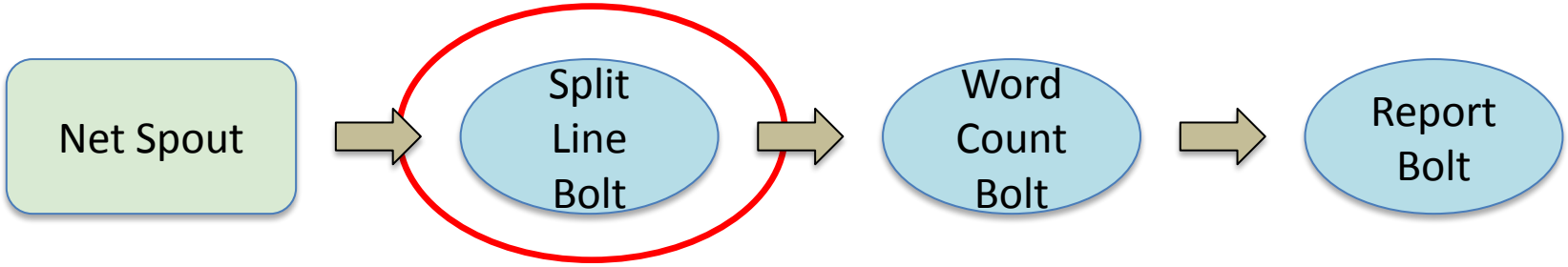



```
public class NetSpout extends BaseRichSpout {
        private BufferedReader reader;
        private SpoutOutputCollector collector;
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
                declarer.declare(new Fields(”line"));
        }
        public void open(Map 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
                }
        }
}
```

Method invoked when reading new data. In this case, reads a line from the socket.

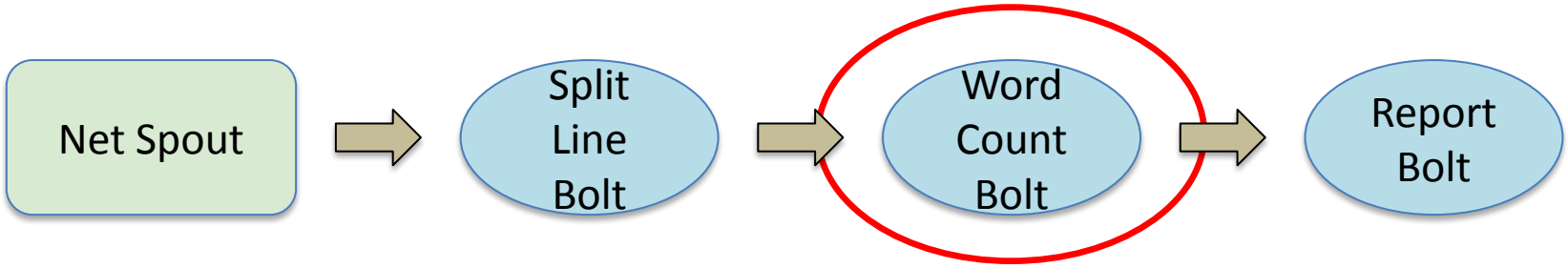



```
public class SplitSentenceBolt extends BaseRichBolt{
        private OutputCollector collector;
        public void prepare(Map config, TopologyContext context, OutputCollector collector) {
                this.collector = collector;
        }
        public void execute(Tuple tuple) {
                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"));
        }
}
```

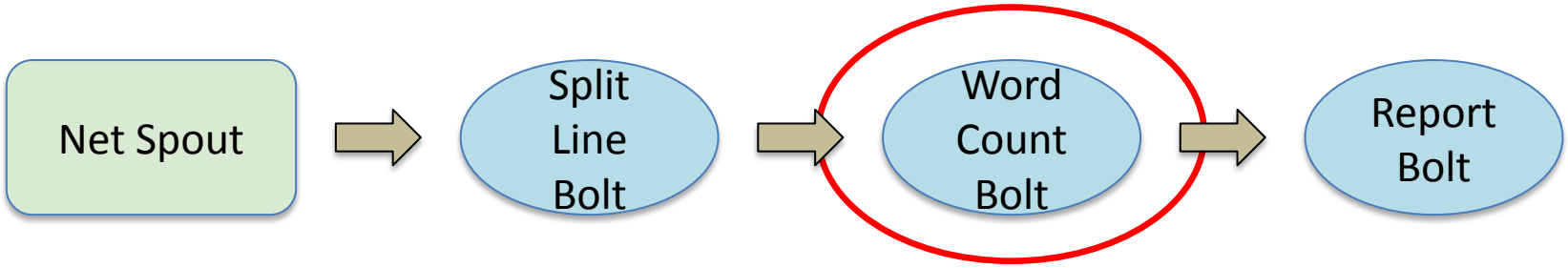



```
public class SplitSentenceBolt extends BaseRichBolt{
        private OutputCollector collector;
        public void prepare(Map config, TopologyContext context, OutputCollector collector) {
                this.collector = collector;
        }
        public void execute(Tuple tuple) {
                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"));
        }
}
```

Defines how to process a line. In this case, splits the line and emits each word in a tuple.



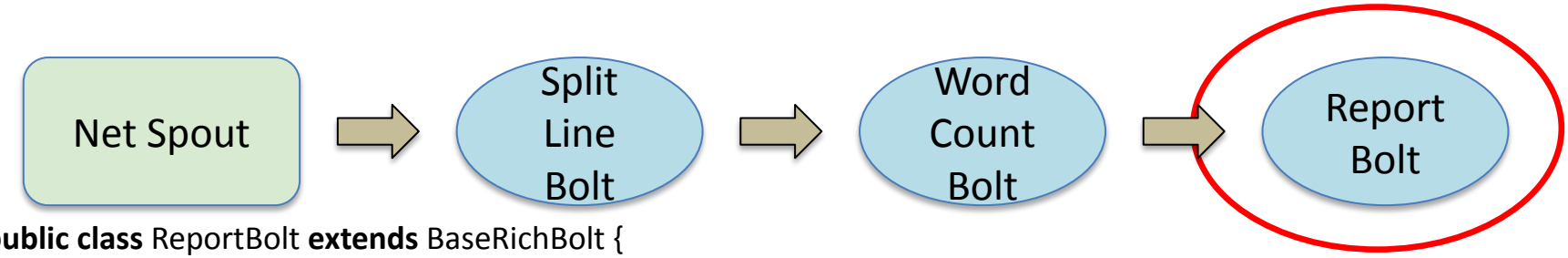

```
public class WordCountBolt extends BaseRichBolt{
        private OutputCollector collector;
        private HashMap<String, Long> counts = null;
        public void prepare(Map config, TopologyContext context, OutputCollector collector) {
                this.collector = collector;
                this.counts = new HashMap<>();
        }
        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"));
        }
}
```



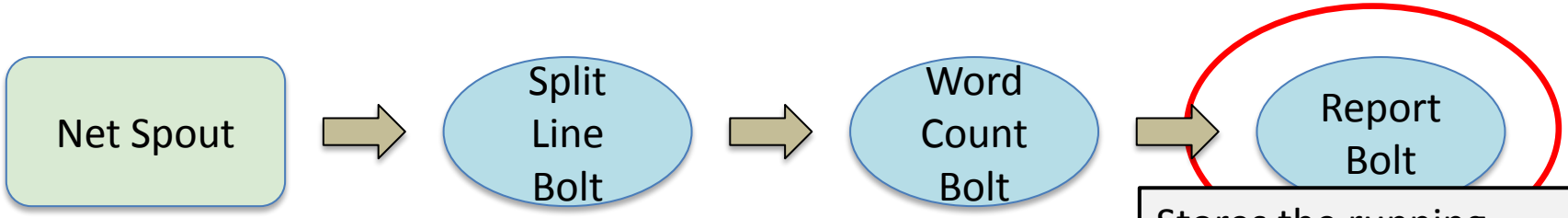

```
public class WordCountBolt extends BaseRichBolt{
        private OutputCollector collector;
        private HashMap<String, Long> counts = null;
        public void prepare(Map config, TopologyContext context, OutputCollector collector) {
                this.collector = collector;
                this.counts = new HashMap<>();
        }
        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"));
        }
}
```

Stores the running counts.

Defines how to process a word. In this case, updates the count for the word and forwards it to the next Bolt.




```
public class ReportBolt extends BaseRichBolt {
        private TreeMap<String, Long> counts = null;
        public void prepare(Map config, TopologyContext context, OutputCollector collector) {
                this.counts = new TreeMap<>();
        }
        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 ---");
                for (Map.Entry<String, Count>  e : counts.entrySet())
                        System.out.println(e.getKey() + " : " + e.getValue());
        }
}
```

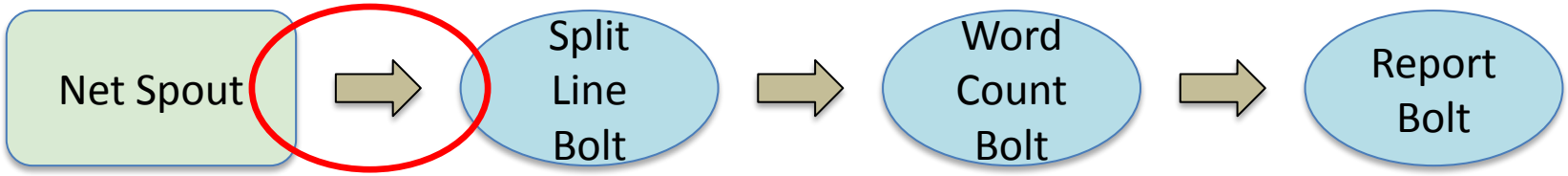



```
public class ReportBolt extends BaseRichBolt {
        private TreeMap<String, Long> counts = null;
        public void prepare(Map config, TopologyContext context, OutputCollector collector) {
                this.counts = new TreeMap<>();
        }
        public void execute(Tuple tuple) {
                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 ---");
                for (Map.Entry<String, Count>  e : counts.entrySet())
                        System.out.println(e.getKey() + " : " + e.getValue());
        }
}
```

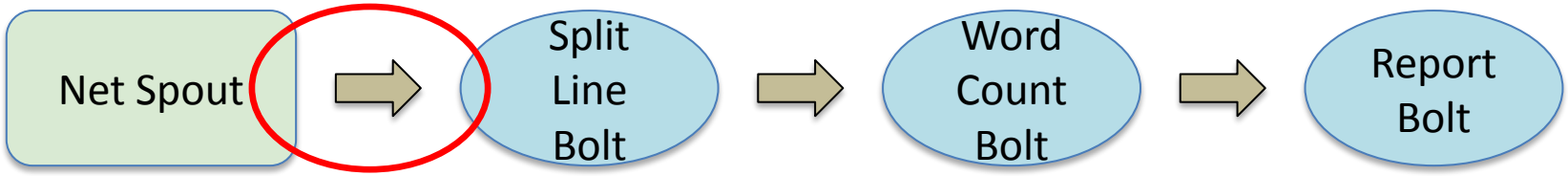
Stores the running counts.

Defines how to process a new tuple. In this case, just updates the count associated with each word.

Defines what to do when finishing. In this case, just dump to the console.

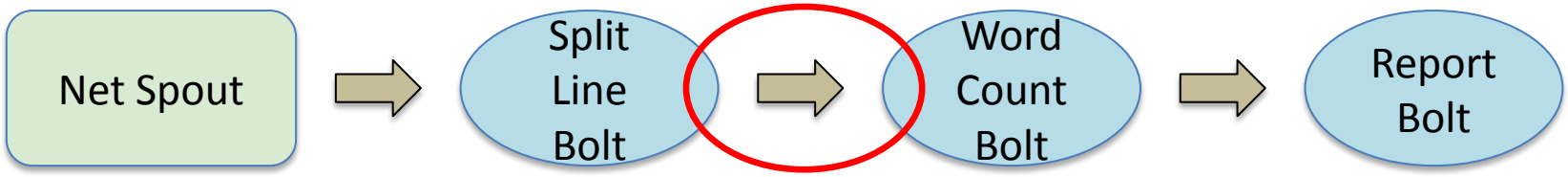



```
NetSpout spout = new NetSpout();
SplitLineBolt splitBolt = new SplitLineBolt();
WordCountBolt countBolt = new WordCountBolt();
ReportBolt reportBolt = new ReportBolt();

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());
```



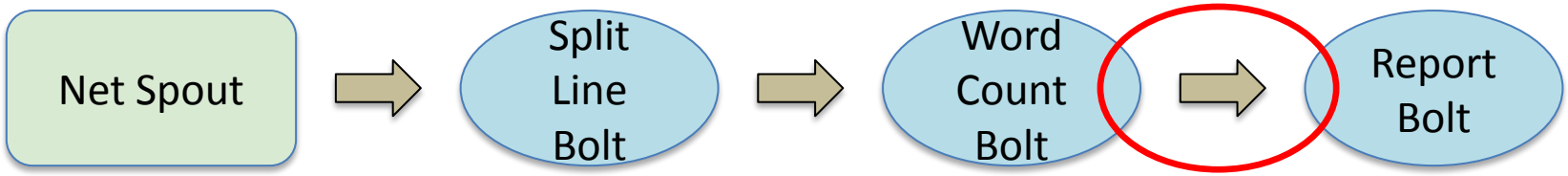

```
NetSpout spout = new NetSpout();
SplitLineBolt splitBolt = new SplitLineBolt();
WordCountBolt countBolt = new WordCountBolt();
ReportBolt reportBolt = new ReportBolt();

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());
```

Defines how to connect elements. In this case, lines are shuffled by the split line bolts.




```
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());
```

Defines how to connect elements. In this case, all tuples with the same word go to the same bolt.




```
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",
// 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());
```

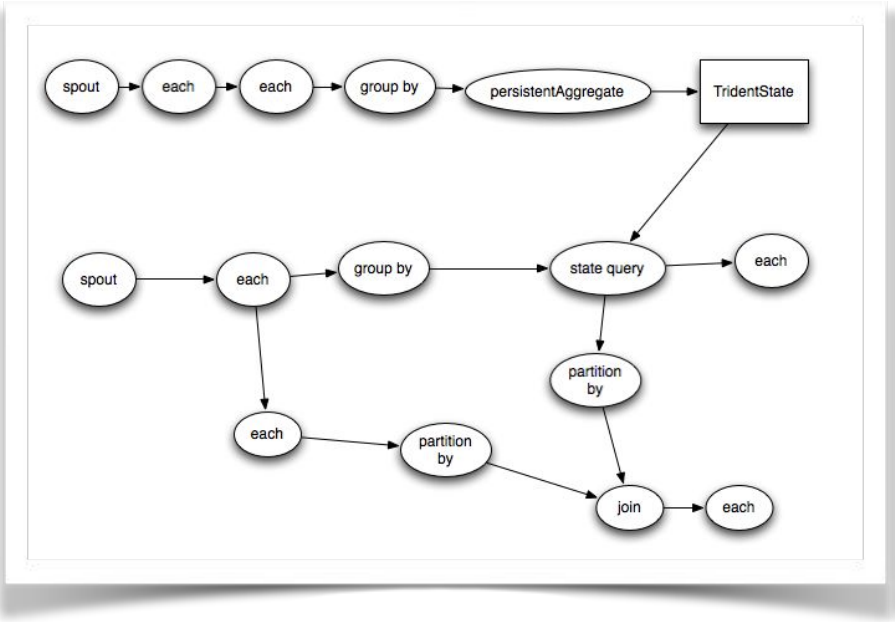
Defines how to connect elements. In this case, all tuples are sent to a single bolt.

Trident

- A high-level abstraction for doing realtime computing on top of Storm
  - (much like Apache 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** (i.e., moving data around)



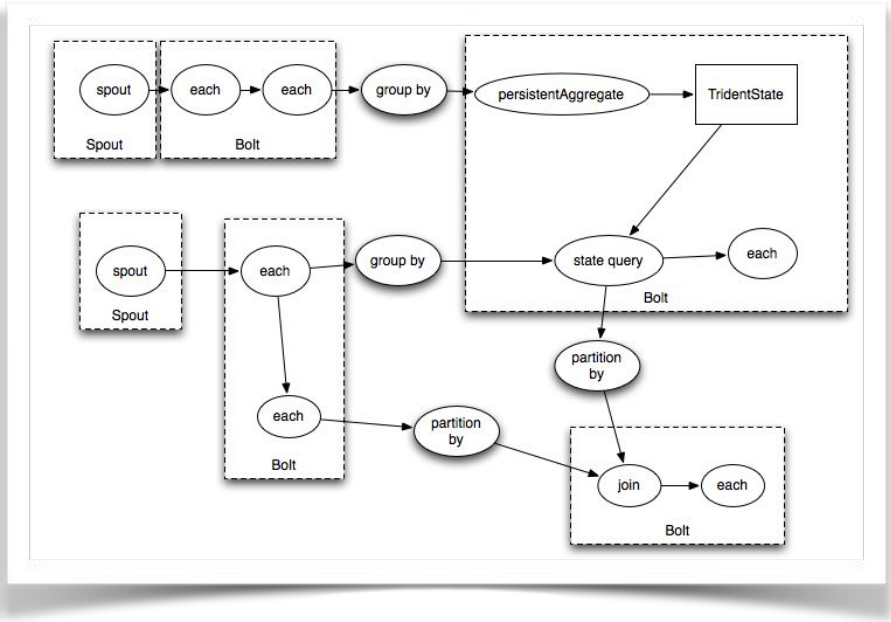


Table of Contents

- Continuous processing
  - Apache Storm
  - **Spark Streaming**
  - Apache Flink

pause

Continuous mode in Spark Streaming

- **Continuous processing** is an experimental streaming execution mode that enables low (~1 ms) end-to-end latency with at-least-once, no fault-tolerance guarantees.
- Supported operation (> v. 2.4.5):
  - Only projections (select, map, flatMap, mapPartitions, etc.) and selections (where, filter, etc.) are supported.
  - Aggregations are not supported.

Continuous mode in Spark Streaming

```
spark \
 .readStream \
 .format("kafka") \
 .option("kafka.bootstrap.servers",\
        "host1:port1,host2:port2") \
 .option("subscribe", "topic1") \
 .load() \
 .selectExpr("CAST(key AS STRING)",\
        "CAST(value AS STRING)") \
 .writeStream \
 .format("kafka") \
 .option("kafka.bootstrap.servers", \
        "host1:port1,host2:port2") \
 .option("topic", "topic1") \
 .trigger(continuous="1 second") \
 .start()
```

Only change necessary to the program.
1 second is the checkpointing interval.

Table of Contents

- Continuous processing
  - Apache Storm
  - Spark Straming
  - **Apache Flink**

Apache Flink

Stateful Computations over Data Streams

Apache Flink

- In many cases, data can be seen as a **stream of events**. We can identify two dimensions:

- **Bounded** vs. **unbounded** streams:
  - Bounded streams have a defined start and end; size is limited.
  - Unbounded streams have a start but no defined end; it is not possible to wait for all input data to arrive.

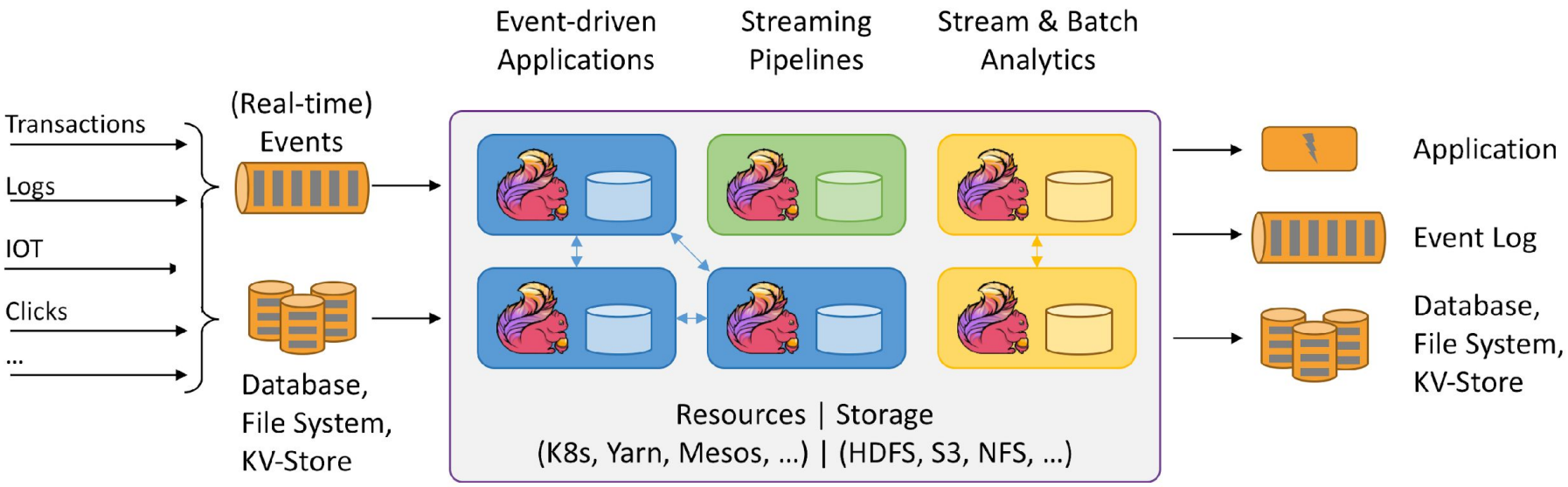
- **Real-time** vs. **recorded** streams:
  - Real-time refers to processing the stream as it is generated.
  - Recorded refers to persisting the stream before processing it.

Apache Flink (2)

- **Batch processing** corresponds to processing a bounded recorded stream.
  - Can be processed by ingesting all data before performing any computations;
  - Ordered ingestion is not required, as a bounded dataset can always be sorted.
- **Stream processing** corresponds to processing an unbounded real-time stream.
  - 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.

Apache Flink (2)

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



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)*.

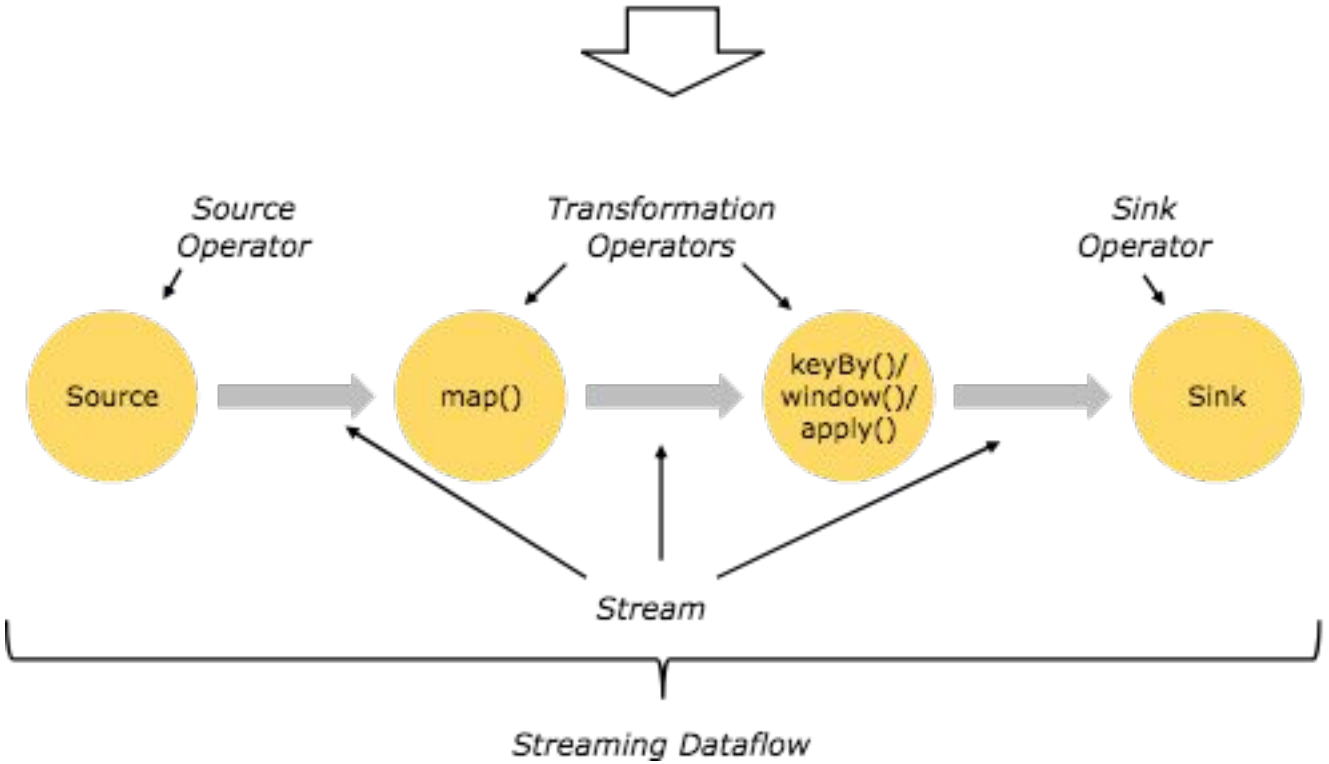
```
DataStream<String> lines = env.addSource(
                                new FlinkKafkaConsumer<>(…));

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

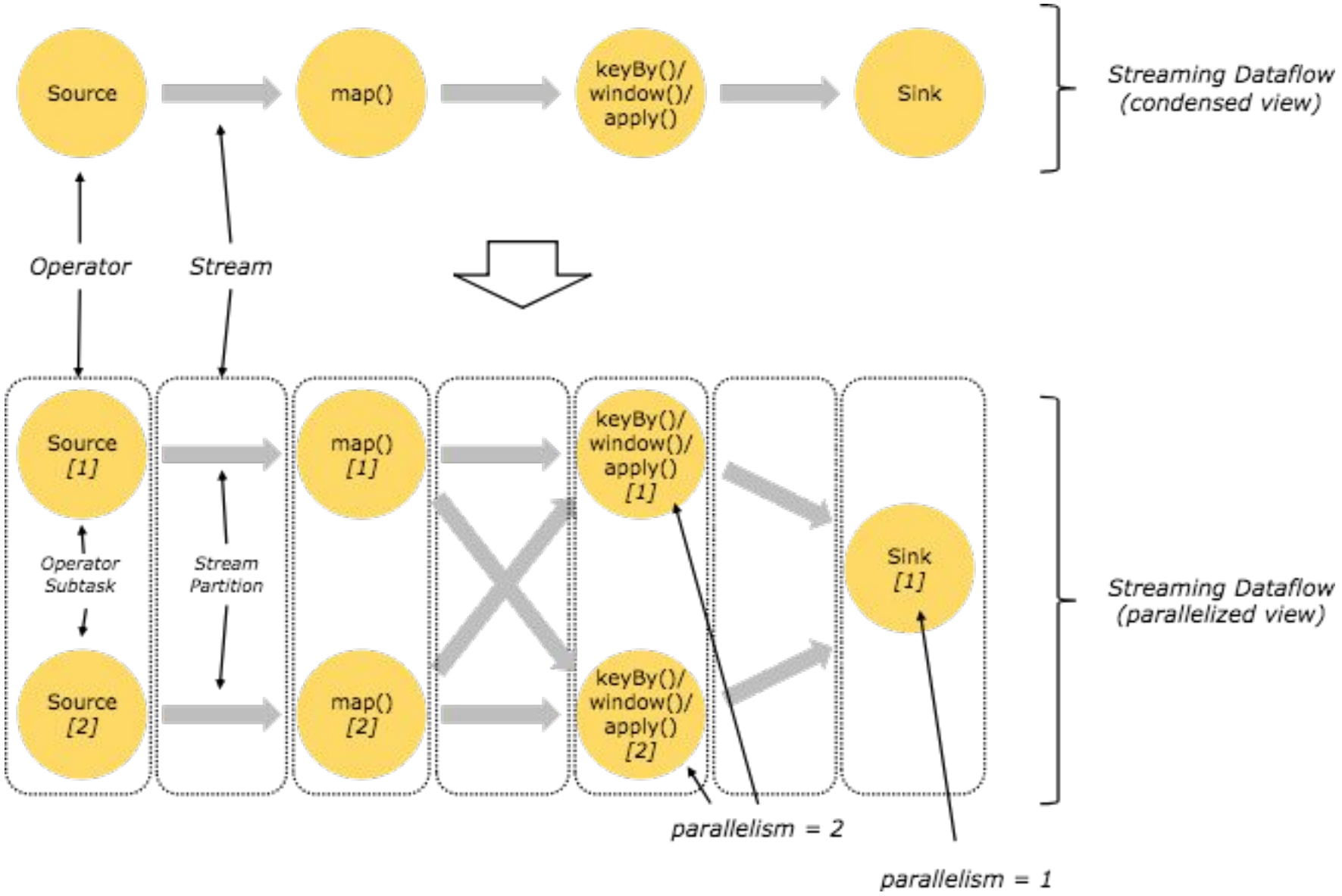


Parallel dataflows

- 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

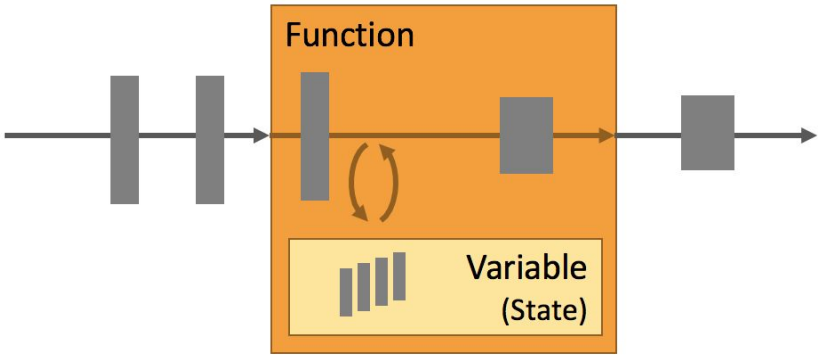

Source
map()
keyBy()/
window()/
apply()
Sink
Streaming Dataflow
(condensed view)
Operator
Stream
Source
[1]
Source
[2]
Operator
Subtask
Stream
Partition
map()
[1]
map()
[2]
keyBy()/
window()/
apply()
[1]
keyBy()/
window()/
apply()
[2]
Sink
[1]
Streaming Dataflow
(parallelized view)
parallelism = 2
parallelism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

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.

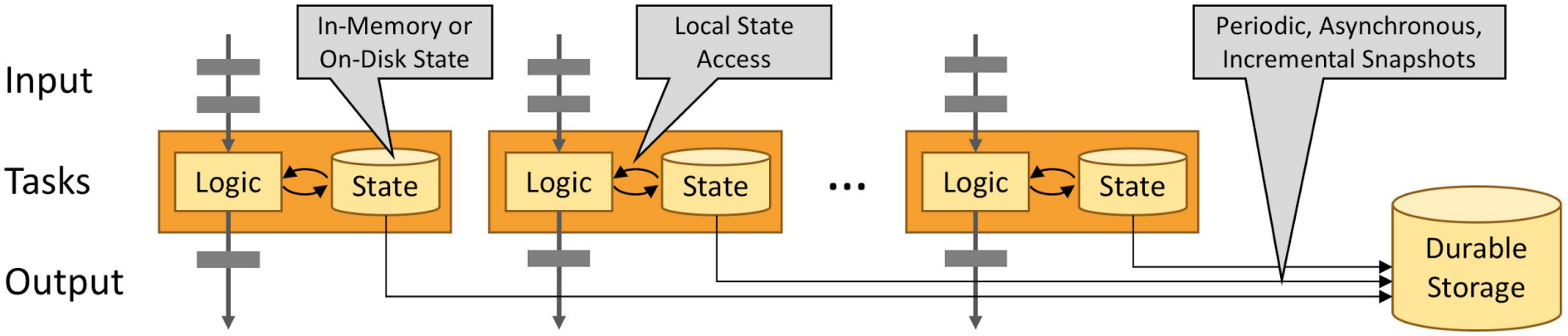


State (2)

- **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.

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

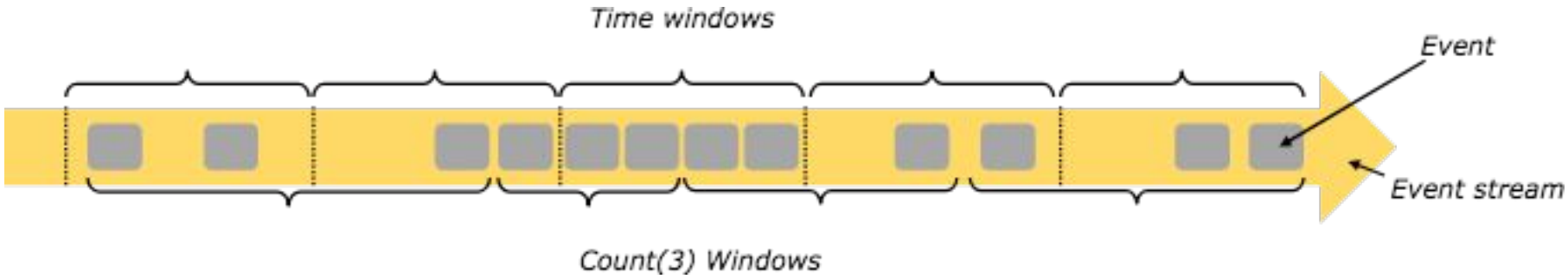
- Combines **stream replay** and **checkpointing**.
- A checkpoint is related to a specific point in each of the input streams along with the corresponding state for each of the operators.
- A streaming dataflow can be resumed from a checkpoint while maintaining consistency *(exactly-once processing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

Time

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

Windows

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

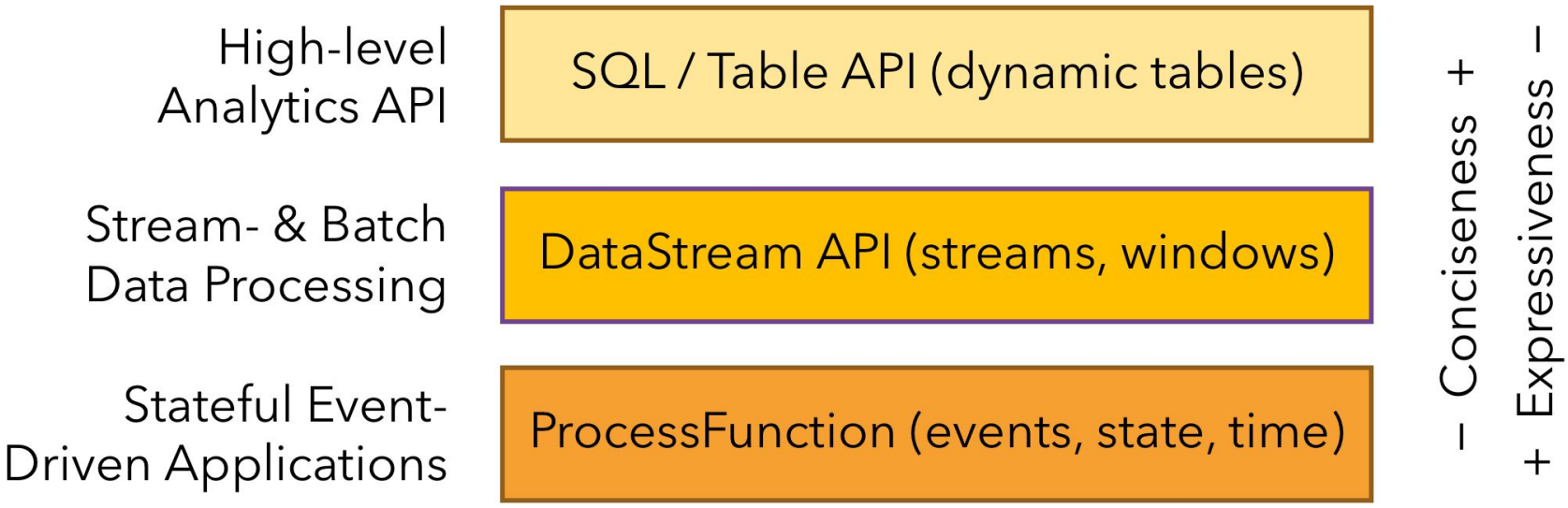


Example

```
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(...)
```

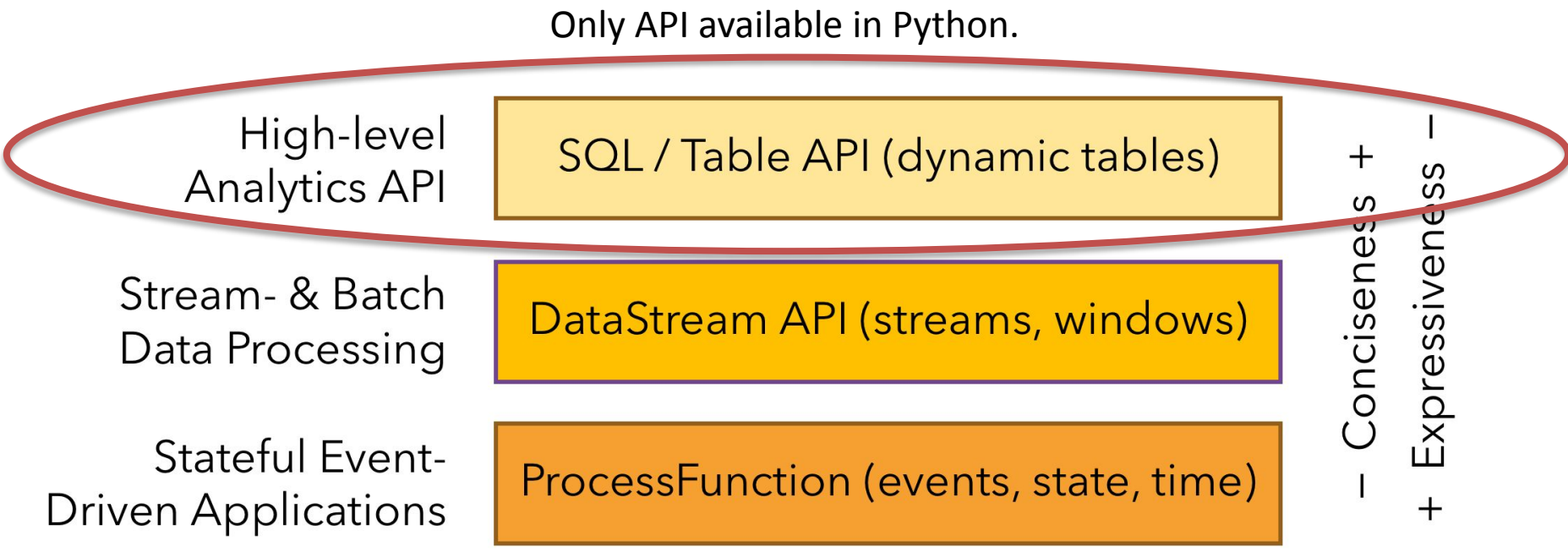
Layered architecture

- Possible to program using different levels of abstractions



Layered architecture

- Possible to program using different levels of abstractions (similar to Spark Streaming).



Datastream API

- Provides programming abstractions at a level similar to Spark Core

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

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 slide deck are from Flink and Storm sites.