Apache Spark and Apache Flink Comparison

Content- Mangement und Suchtechnologien
Florian Thom s0558101
Daniel Nagel s0559090

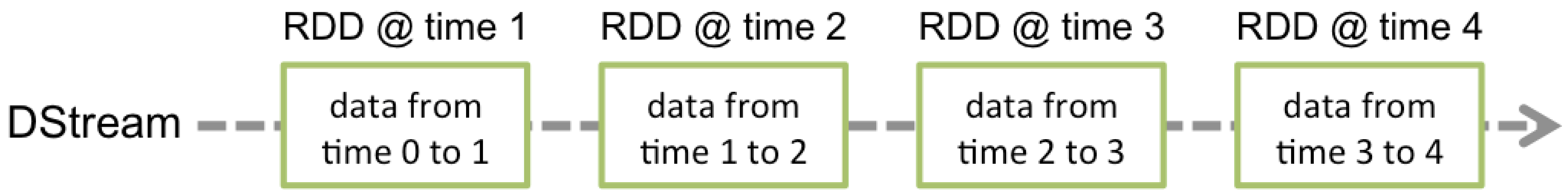
Apache Spark: Streaming

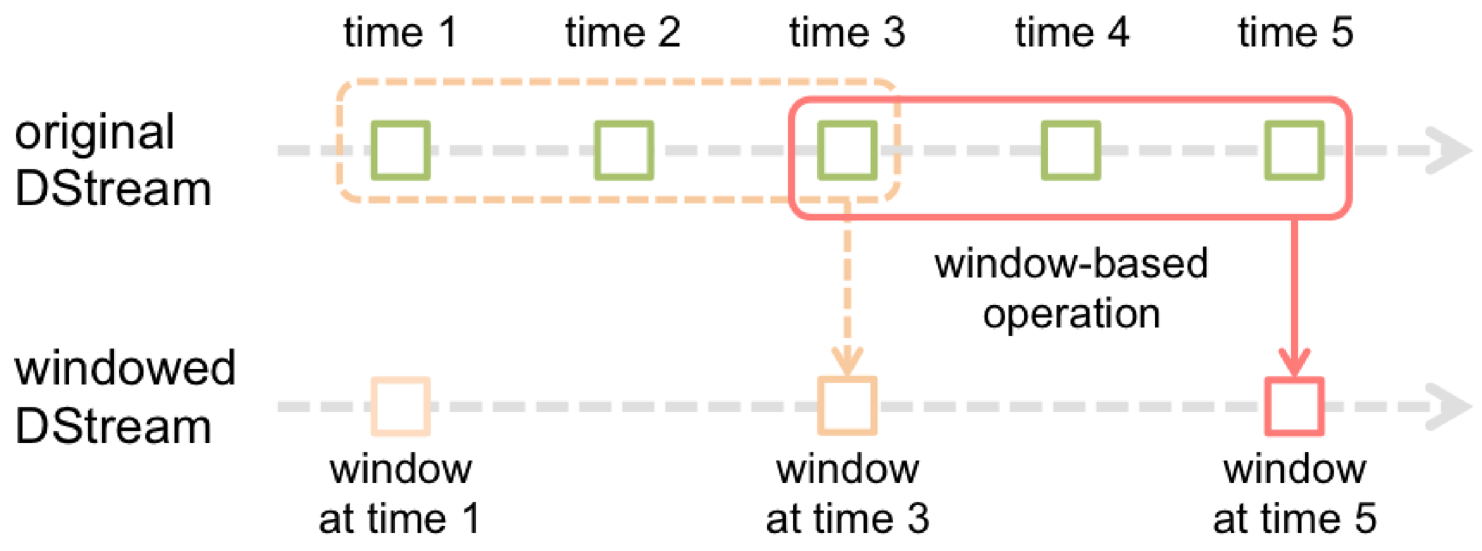
»Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.«

- Spark Offical Documentation

- **descretized stream (DStream) represents a continuous stream of data**
  - internally represented by a series of RDDs
- **DStream provides higher-order functions**
  - all operations are working on the underlying RDDs
- **two built-in streaming sources: basic and advanced**
  - basic: file systems and socket connections
  - advanced: Apache Kafka, Apache Flume, Amazon Kinesis

Apache Spark: DStream



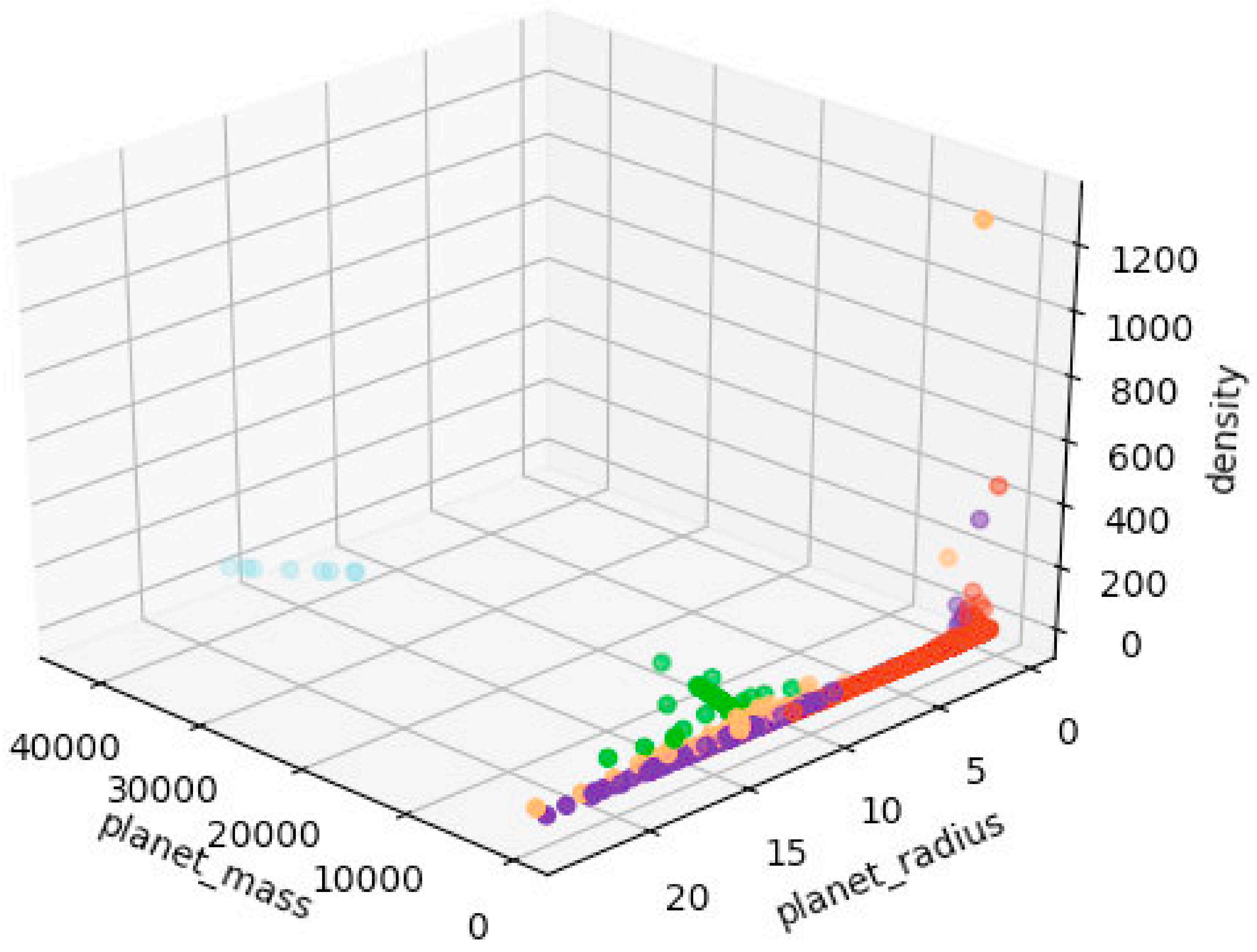


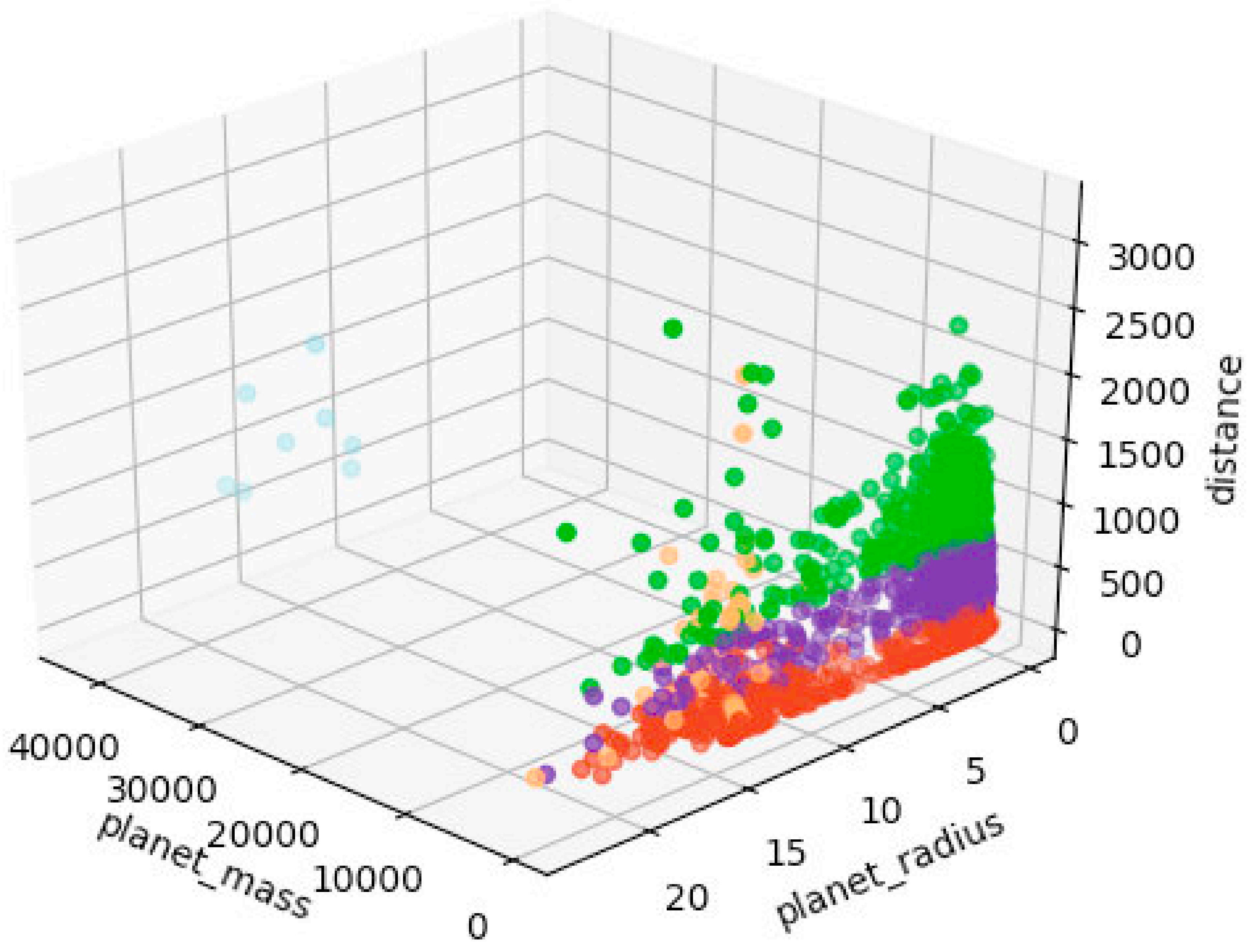
- DStream is split into batches with RDDs as data representation
  - multiple batches can be stored in a single window
- StreamingContext contains the execution threads and batch interval
  - the batch interval defines when current batch ends and a new batch starts
  - batches can be grouped into windows
  - window interval needs to be a multiple of the batch interval
- use foreachRDD to access the underlying RDD representation

Apache Spark: Streaming Example

```
val sc = spark.sparkContext
val ssc = new StreamingContext(sc, Seconds(2))

var predicted: RDD[(Vector, Int)] = sc.emptyRDD
var kmodel = new KMeansModel(Array.fill(numClusters)(Vectors.zeros(numDim)))
val data = ssc.textFileStream("file://" + relPath.getCanonicalPath)
        .map(parseCSV(_))
        .filter(!containsEmpty(_))
        .map(transformToTuple(_))
val vectors = data.map(f => Vectors.dense(f._4, f._5, f._7))

vectors.foreachRDD(rdd => {
        kmodel = new KMeans().setK(numClusters).setInitialModel(kmodel).setMaxIterations(numIter).run(rdd)
        val current = rdd.map(f => (f, kmodel.predict(f)))
        predicted = predicted.union(current)
})

ssc.start()
ssc.awaitTerminationOrTimeout(20000)
ssc.stop(false)
```

Apache Spark: Streaming Example



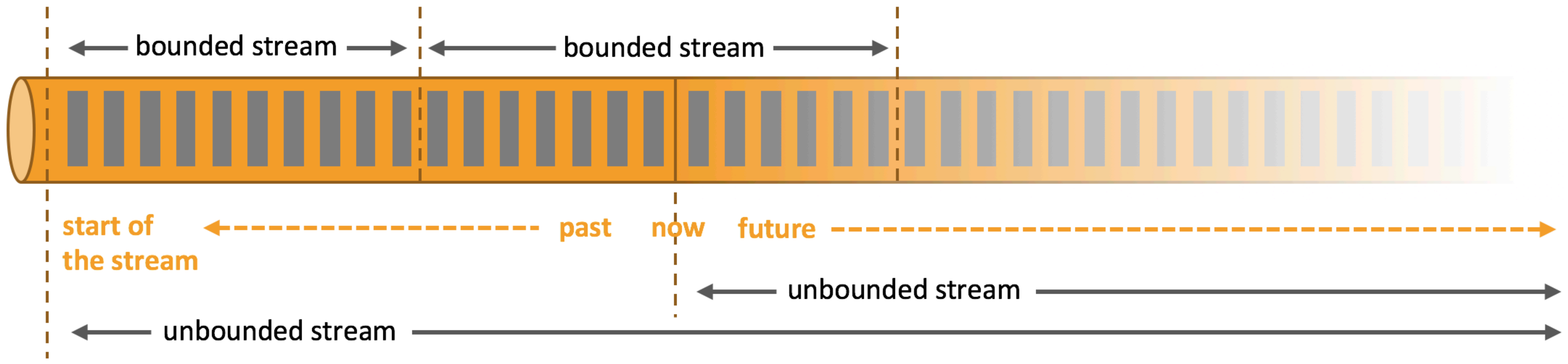


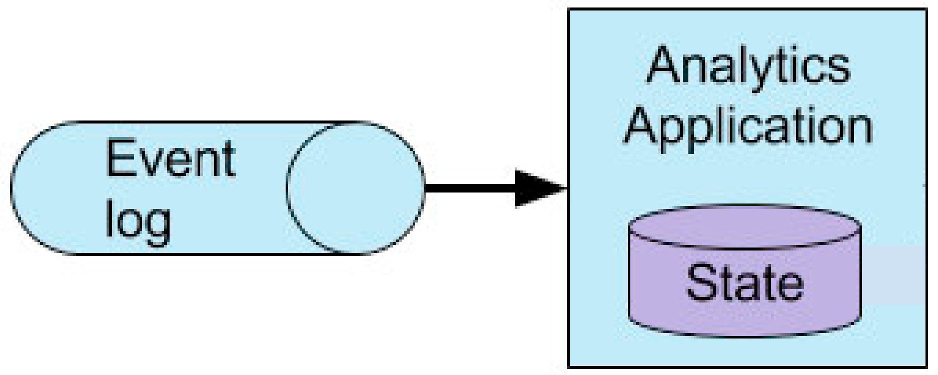
- exoplanet archive dataset from caltech and nasa
- clustering results can be exported at runtime

Apache Flink: Streaming I

»Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams.«

- Flink Offical



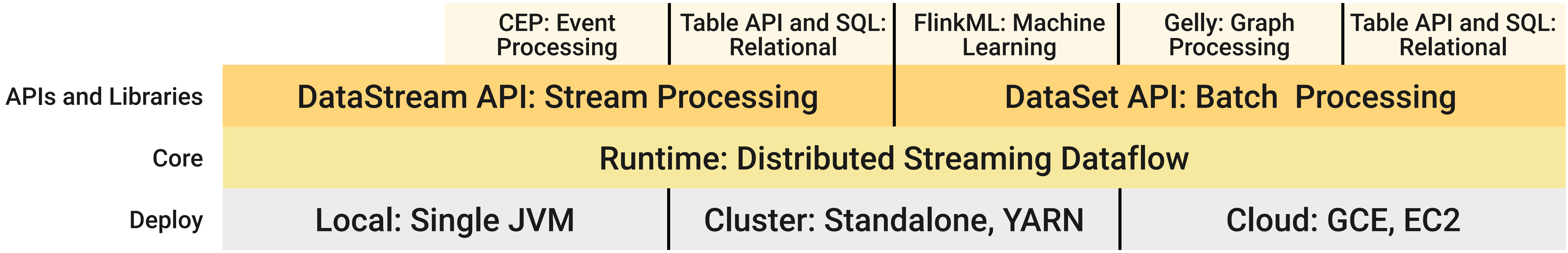


- **bounded datastreams (near real-time, simliar to Apache Spark)**
  - **have a defined start and end**
- **unbounded stream (real-time)**
  - **have a start but no defined end**
- **stateful functions and operators store data across the process events**

Apache Flink: Streaming II

- high throughput, scalable
- low latency due real time computation
  - no waiting required (window / batch)
- exactly-once semantics for stateful computations
  - Flink: Standard
  - Spark: only with much efforts, not applicable to windowed operations
- base implementation in Java

Apache Flink: API I



CEP: Event Processing

Table API and SQL: Relational

FlinkML: Machine Learning

Gelly: Graph Processing

Table API and SQL: Relational

APIs and Libraries

DataStream API: Stream Processing

DataSet API: Batch Processing

Core

Runtime: Distributed Streaming Dataflow

Deploy

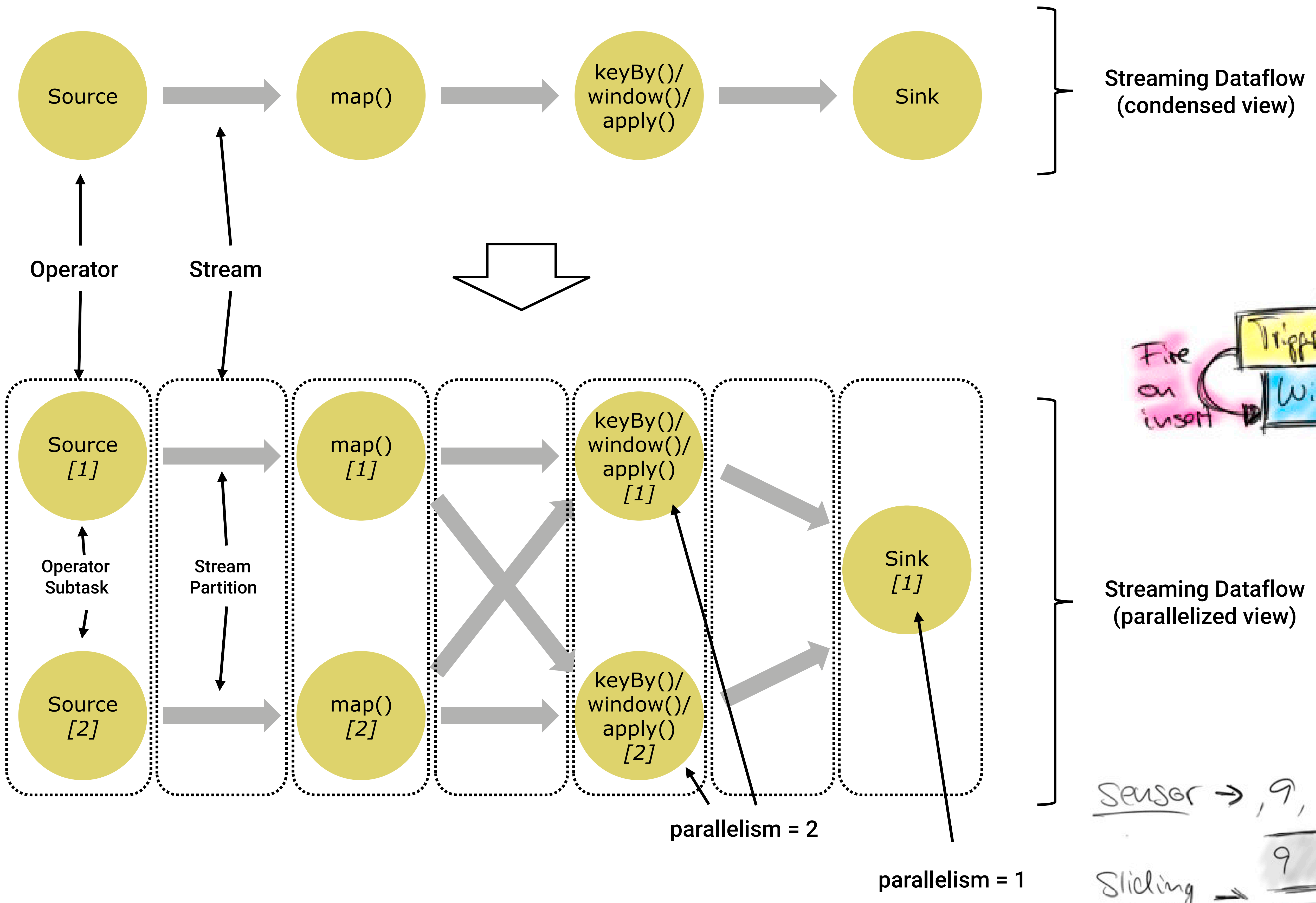
Local: Single JVM

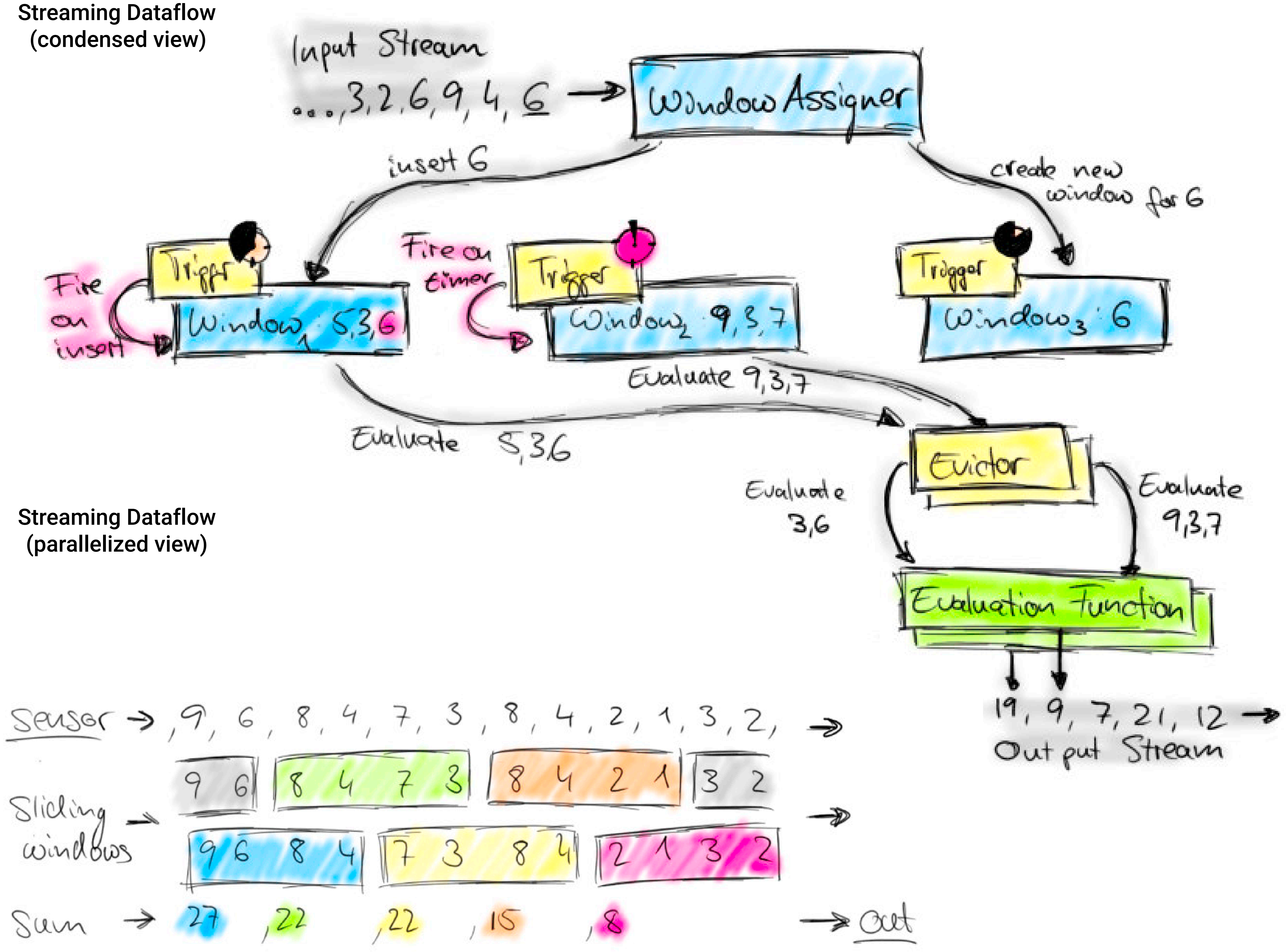
Cluster: Standalone, YARN

Cloud: GCE, EC2

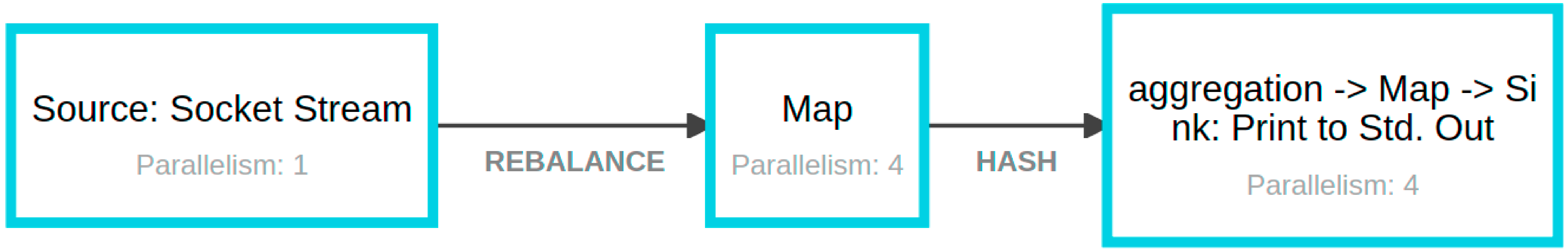
- DataSet: base data structure for batch processing
    - internally represented as bounded stream
- DataStream: base data structure for stream processing
    - internal representation as directed acyclic graph (job graph)
    - start: data source / connectors (build-in: Apache Kafka, RabbitMQ, ...)
    - end: data sink / connectors (build-in: Apache Kafka, Cassandra, Redis, ...)
- DataSet and DataStream can not be combined

Apache Flink: API II





Apache Flink: Streaming Example



Source: Socket Stream
Parallelism: 1

REBALANCE

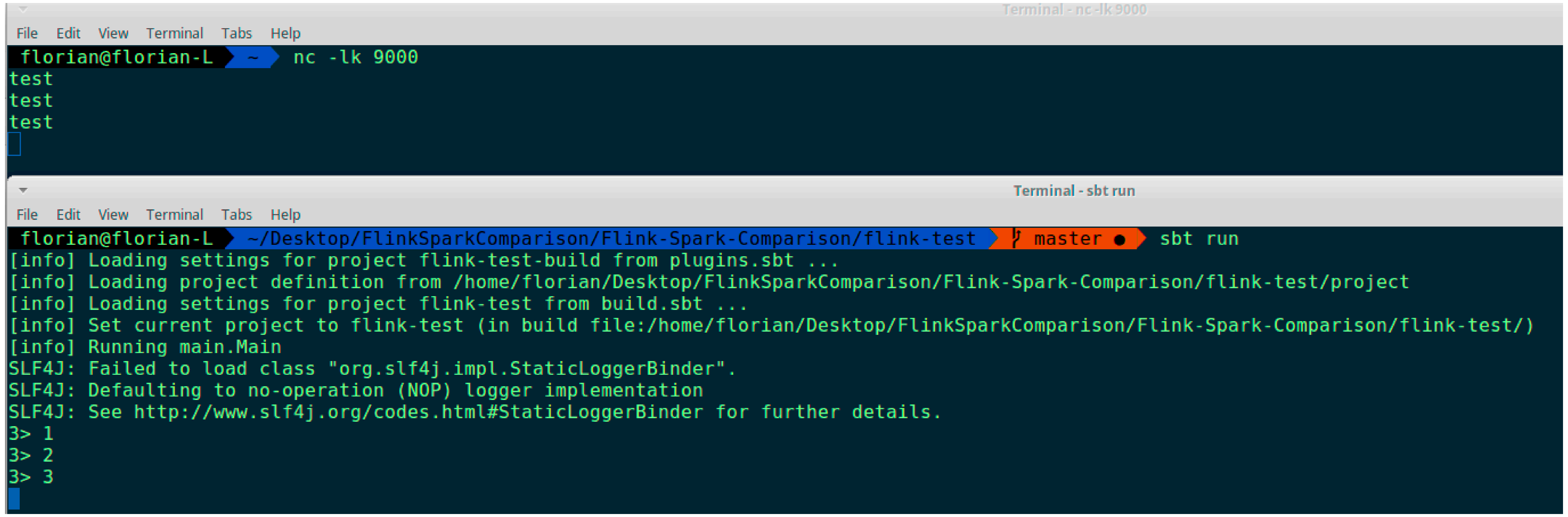
Map
Parallelism: 4

HASH

aggregation -> Map -> Sink: Print to Std. Out
Parallelism: 4

```
val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment
env.setMaxParallelism(4)
val text: DataStream[String] = env.socketTextStream("127.0.0.1", 9000, '\n')
text.map(a=>(1,1))
        .keyBy(0)
        .sum(0)
        .map(b=>b._1)
        .print()
env.execute()
```

Apache Flink: Streaming Example



Terminal - nc -lk 9000
File Edit View Terminal Tabs Help

```
florian@florian-L  ~  nc -lk 9000
test
test
test
```

Terminal - sbt run
File Edit View Terminal Tabs Help

```
florian@florian-L  ~/Desktop/FlinkSparkComparison/Flink-Spark-Comparison/flink-test  master ●  sbt run
[info] Loading settings for project flink-test-build from plugins.sbt ...
[info] Loading project definition from /home/florian/Desktop/FlinkSparkComparison/Flink-Spark-Comparison/flink-test/project
[info] Loading settings for project flink-test from build.sbt ...
[info] Set current project to flink-test (in build file:/home/florian/Desktop/FlinkSparkComparison/Flink-Spark-Comparison/flink-test/)
[info] Running main.Main
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
3> 1
3> 2
3> 3
```

- stateful behaviour
- job graph
  - define data source
  - transformations (keyBy: repartitions the given stream and ensures parallelism, sum: stateful continuous elements)
  - execute graph

Apache Spark and Flink Benchmark

- example should be understandable and implementable
- test algorithm is a simple word count over a given document
- lipsum dataset is used with 50 paragraphs
  - replicated multiple times to reach 10, 100, 1000 and 2000MB
- spark and flink have the same base configuration
  - Spark is using a 2 second batch interval
  - we decided to disable windowing (2 second interval) in Flink because of no performance gain
- only looking at the duration of the main job
  - ignoring latency, throughput, backpressure

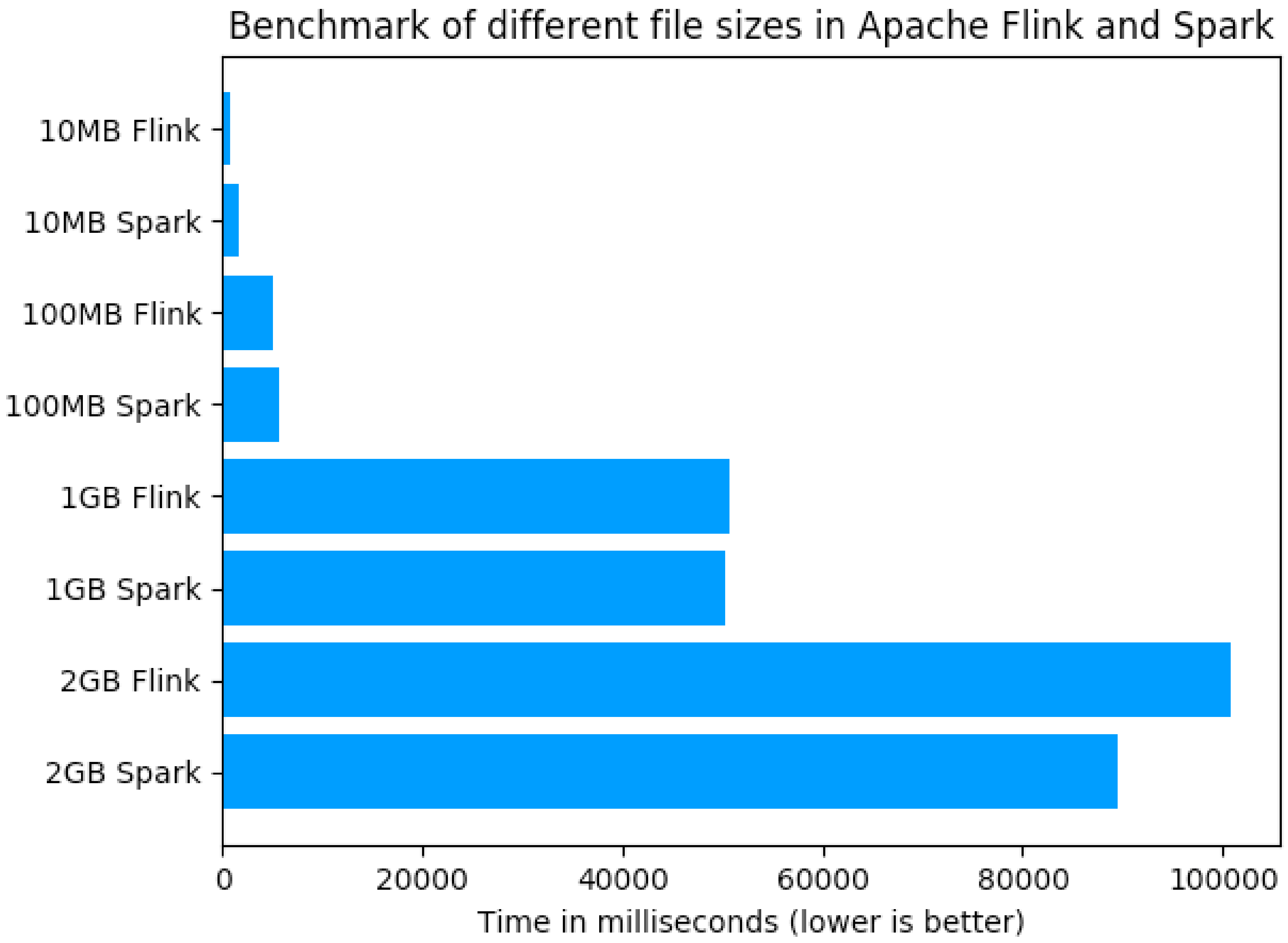
Benchmark: Spark Implementation

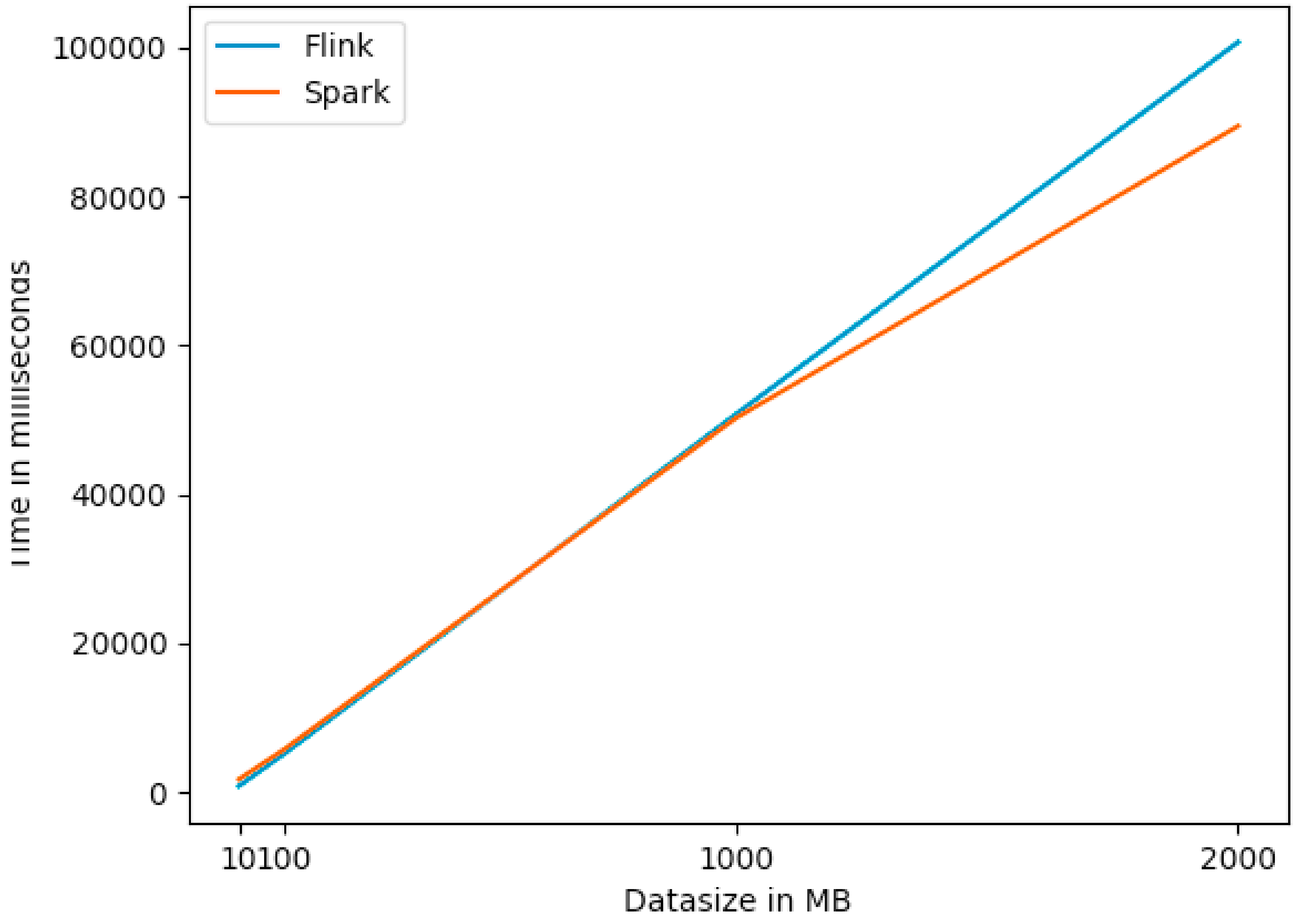
```
def split(line: String): Array[String] =
{
    line.toLowerCase().replaceAll(",", " ").replaceAll(".", " ").split(" ")
}

val sc = spark.sparkContext
val ssc = new StreamingContext(sc, Seconds(2))
var counts: RDD[(String, Long)] = sc.emptyRDD

val data = ssc.textFileStream("file://...")
    .filter(!_.isEmpty())
    .flatMap(split(_))
val words = data.map(f => (f, 1L)).reduceByKey((a, b) => a + b)
words.foreachRDD(rdd => {
    val merged = counts.union(rdd)
    counts = merged.reduceByKey((a, b) => a + b)
})

ssc.start()
ssc.awaitTerminationOrTimeout(20000)
ssc.stop(false)
```

Benchmark: Flink Implementation

```
case class WordWithCount(word:String,count:Long)

class LoremWordCount
{
    val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment
    env.setParallelism(4)
    val text: DataStream[String] = env.readTextFile("...",StandardCharsets.UTF_8.name())
    text.flatMap(line => line
        .toLowerCase()
        .replaceAll(",", " ")
        .replaceAll(".", " ")
        .split("\\s")
      )
      .map(w => WordWithCount(w, 1))
      .keyBy("word").sum("count")
      .addSink(new DiscardingSink[WordWithCount]())
    env.execute()
}

class Sink[T] extends SinkFunction[T] {}
```

Benchmark: Results



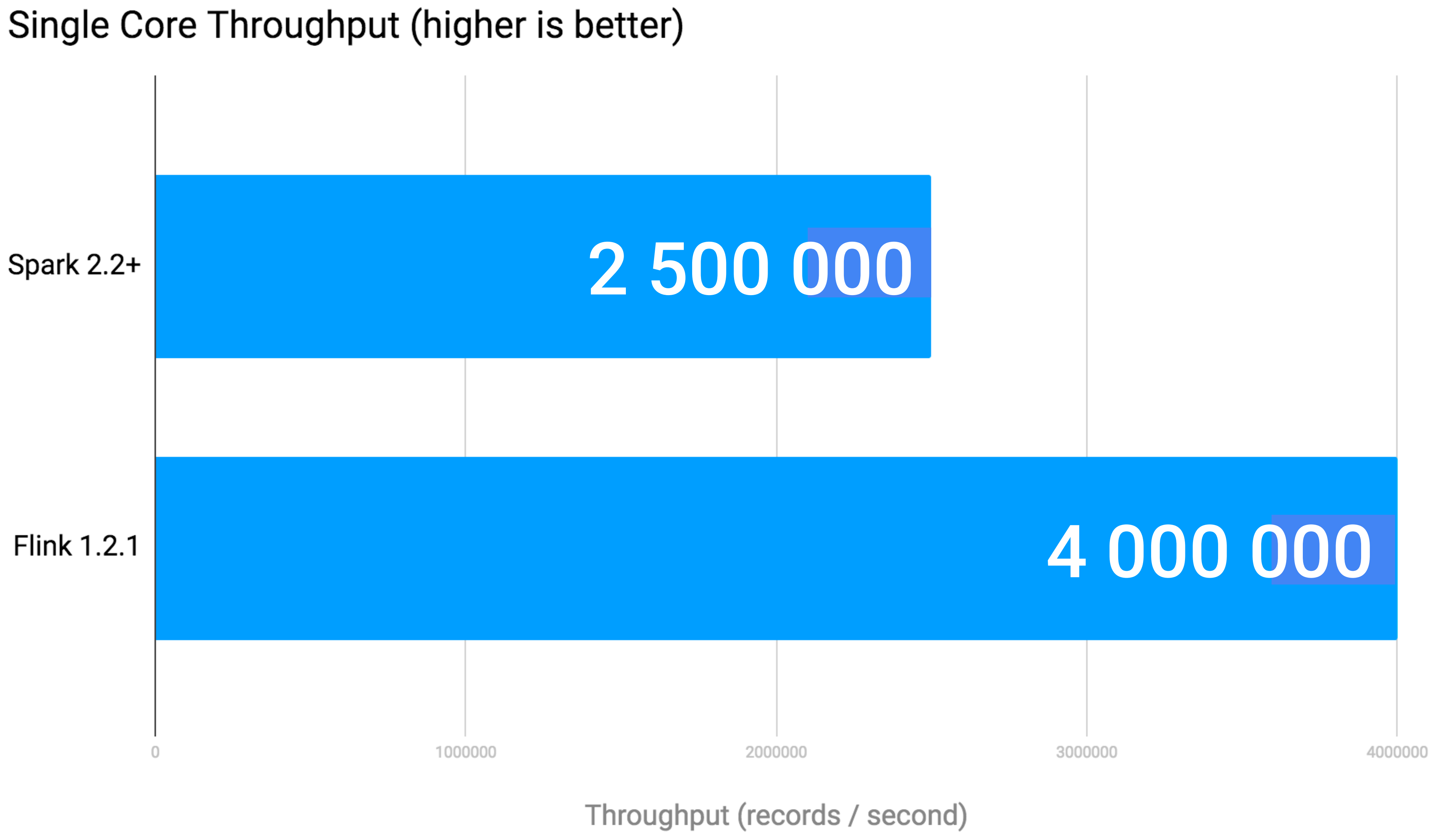


- node specifications
  - ubuntu 16.04.5 LTS
  - Intel Core i5-7200U @ 2.50GHz (4 Threads), 8GB DDR4, 256 Samsung SSD

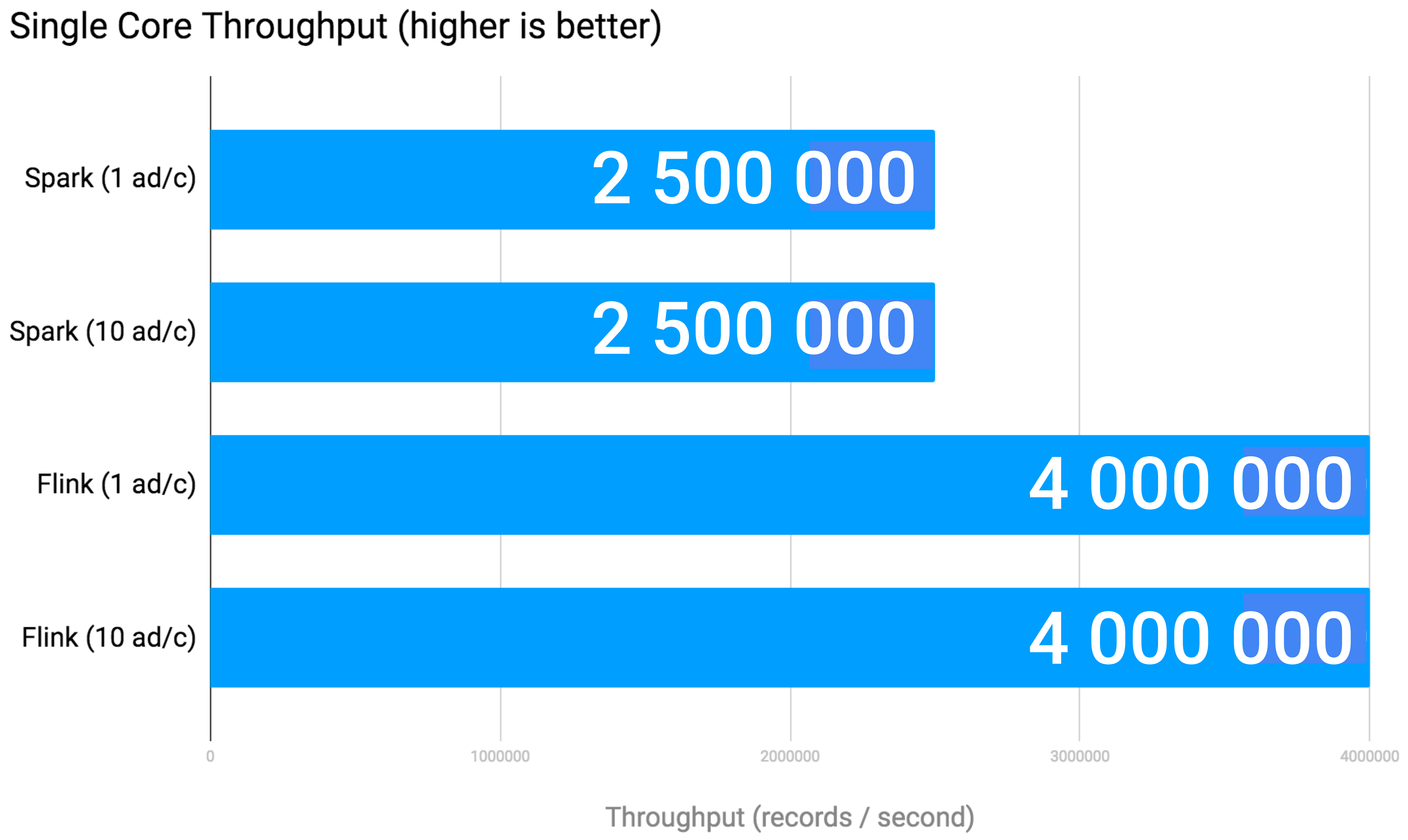
Benchmark: Problems

- only a single data source
  - reading from hard drive / ssd
- the entry point is hard to define due to endless stream
  - other metrics are needed to find the right comparisons
- data will be discarded and not processed further
- one algorithm used
  - different business logic has different performance requirements
- local machine, scalability not tested
  - maby CPU throttling and other issues

Apache Spark and Flink other Benchmarks



Single Core Throughput (higher is better)



Single Core Throughput (higher is better)

»... that benchmark results most often represent a narrow combination of business logic and configuration options, deployed in an artificial environment.«

- data Artisans

Sources

Apache Spark

spark.apache.org/streaming/

spark.apache.org/docs/2.2.0/streaming-programming-guide.html

spark.apache.org/docs/2.2.0/mllib-clustering.html

github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/clustering/

Apache Flink

“Benchmarking Distributed Stream Processing Engines”: Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanern, Volker Markl url: arxiv.org/pdf/1802.08496.pdf [19/12/2018]

www.oreilly.com/library/view/stream-processing-with/9781491974285/

www.kdnuggets.com/2016/10/beginners-guide-apache-flink-explained.html

ci.apache.org/projects/flink/flink-docs-release-1.7/concepts/programming-model.html

flink.apache.org/news/2015/12/04/Introducing-windows.html

thirdeyedata.io/apache-flink/

other Benchmarks

www.data-artisans.com/blog/curious-case-broken-benchmark-revisiting-apache-flink-vs-databricks-runtime

github.com/yahoo/streaming-benchmarks

datasets

exoplanetarchive.ipac.caltech.edu

www.lipsum.com

Appendix: Flink / Spark Code Comparision

```
object WordCount {

  def main(args: Array[String]) {

    val env = new SparkContext("local","wordCount")

    val data = List("text1","text21 text22 text23","text3")

    val dataRDD = env.parallelize(data)

    val words = dataRDD.flatMap(value => value.split("\\s+"))

    val mappedWords = words.map(value => (value,1))

    val sum = mappedWords.reduceByKey(_+_)

    println(sum.collect())

  }

}
```

```
object WordCount {

  def main(args: Array[String]) {

    val env = ExecutionEnvironment.getExecutionEnvironment

    val data = List("text1","text21 text22 text23","text3")

    val dataDataSet = env.fromCollection(data)

    val words = dataDataSet.flatMap(value => value.split("\\s+"))

    val mappedWords = words.map(value => (value,1))

    val grouped = mappedWords.groupBy(0)

    val sum = grouped.sum(1)

    println(sum.collect())
  }

}
```