CS 591 K1:
Data Stream Processing and Analytics

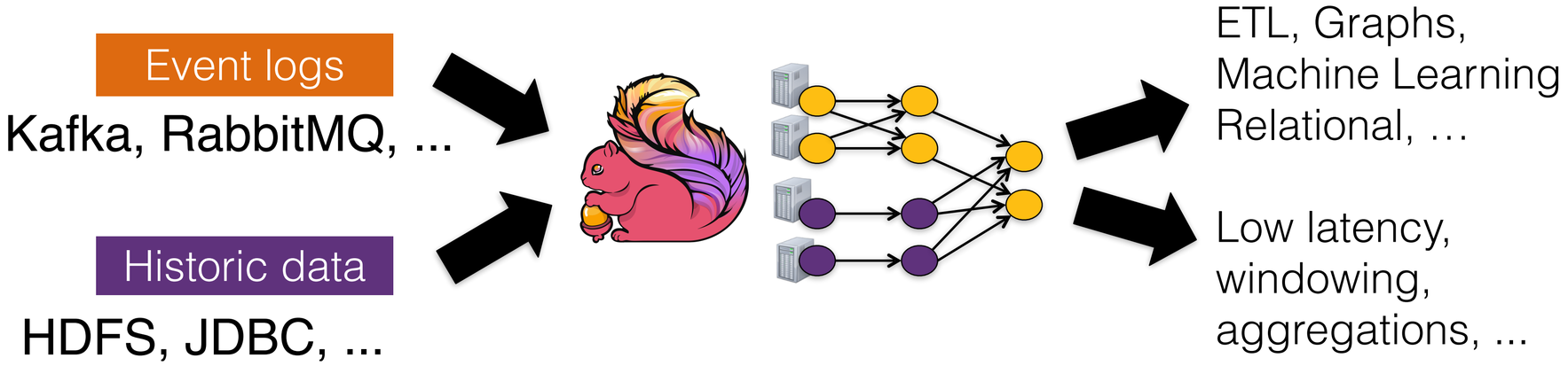
Spring 2021

Introduction to Apache Flink

Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

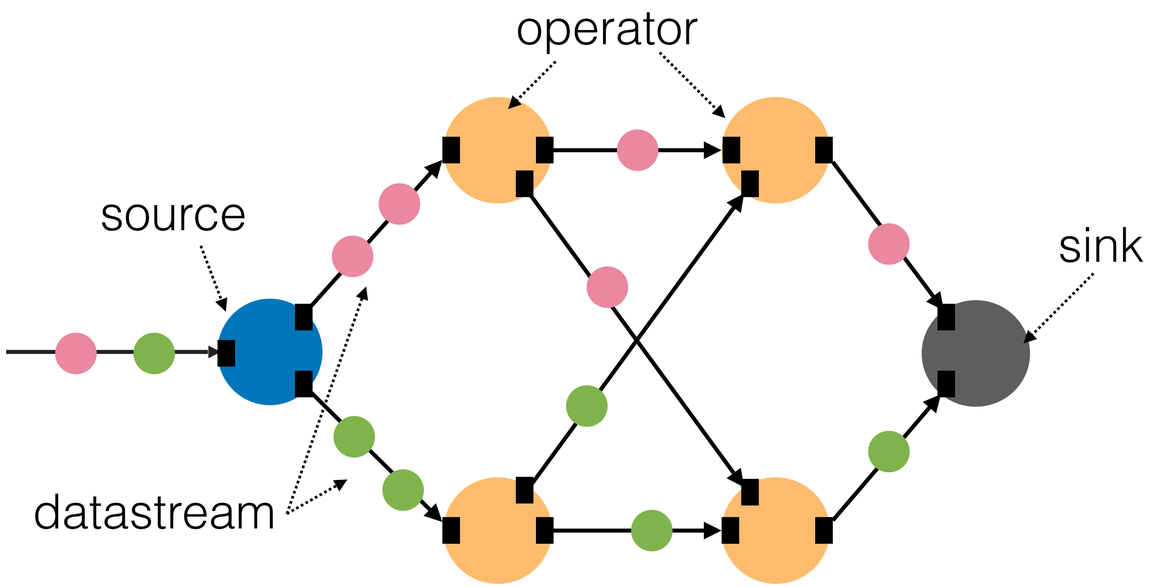
Apache Flink

- An open-source, distributed data analysis framework
- True streaming at its core
- Streaming & Batch API



Dataflow programming model

Dataflow graph: a Directed Acyclic Graph (DAG)



Writing a Flink Program

1. Bootstrap Sources
2. Apply Operators
3. Output to Sinks

Dataflow graph

- operators are nodes, data channels are edges
- channels have FIFO semantics
- streams of data elements flow continuously along edges

Operators

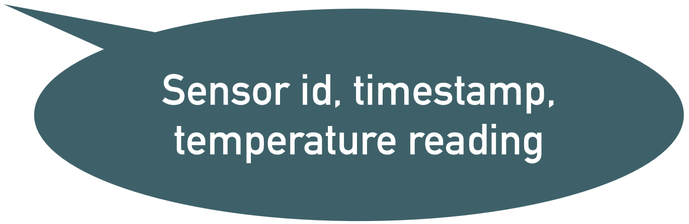
- receive one or more input streams
- perform tuple-at-a-time, window, logic, pattern matching transformations
- output one or more streams of possibly different type

Example: Sensor Readings

```
case class Reading(id: String, time: Long, temp: Double)


object MaxSensorReadings {
  def main(args: Array[String]) {
  val env = StreamExecutionEnvironment.getExecutionEnvironment
    val sensorData = env.addSource(new SensorSource)
    val maxTemp = sensorData
      .map(r => Reading(r.id,r.time,(r.temp-32)*(5.0/9.0)))
      .keyBy(_.id)
      .max("temp")
    maxTemp.print()
    env.execute("Compute max sensor temperature”)
  }
}
```

Example: Sensor Readings

```
case class Reading(id: String, time: Long, temp: Double)


object MaxSensorReadings {
  def main(args: Array[String]) {
  val env = StreamExecutionEnvironment.getExecutionEnvironment
    val sensorData = env.addSource(new SensorSource)
    val maxTemp = sensorData
      .map(r => Reading(r.id,r.time,(r.temp-32)*(5.0/9.0)))
      .keyBy(_.id)
      .max("temp")
    maxTemp.print()
    env.execute("Compute max sensor temperature”)
  }
}
```



Example: Sensor Readings

```
case class Reading(id: String, time: Long, temp: Double)

object MaxSensorReadings {
  def main(args: Array[String]) {
  val env = StreamExecutionEnvironment.getExecutionEnvironment
    val sensorData = env.addSource(new SensorSource)
    val
                                          time,(r.temp-32)*(5.0/9.0)))
      .max("temp")
    maxTemp.print()
    env.execute("Compute max sensor temperature”)
  }
}
```

Flink programs are defined in regular Scala/Java methods

Set up the execution environment: local, cluster, I/O, time semantics, parallelism, …

Example: Sensor Readings

```
case class Reading(id: String, time: Long, temp: Double)


object MaxSensorReadings {
  def main(args: Array[String]) {
  val env = StreamExecutionEnvironment.getExecutionEnvironment
    val sensorData = env.addSource(new SensorSource)
    val maxTemp = sensorData
      .map(r => Reading(r.id,r.ti                    0/9.0)))
      .keyBy(_.id)
      .max("temp")
    maxTemp.print()
    env.execute("Compute max sensor temperature”)
  }
}
```

Ingest a stream of sensor readings

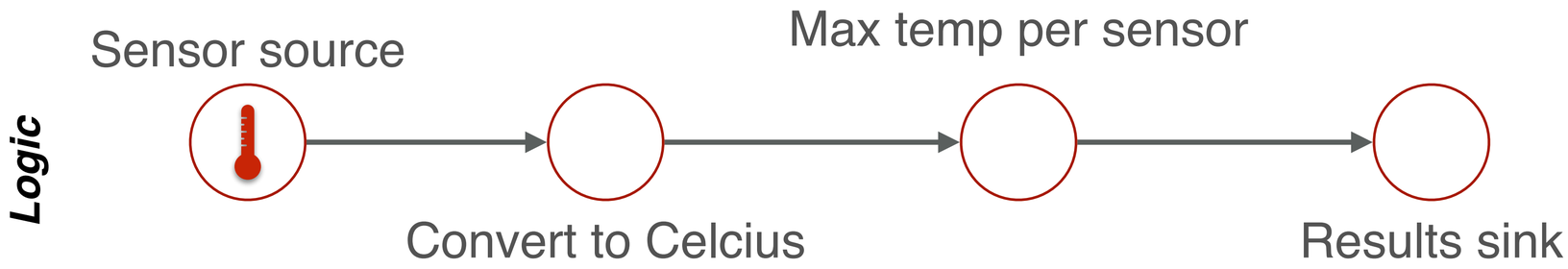
Example: Sensor Readings

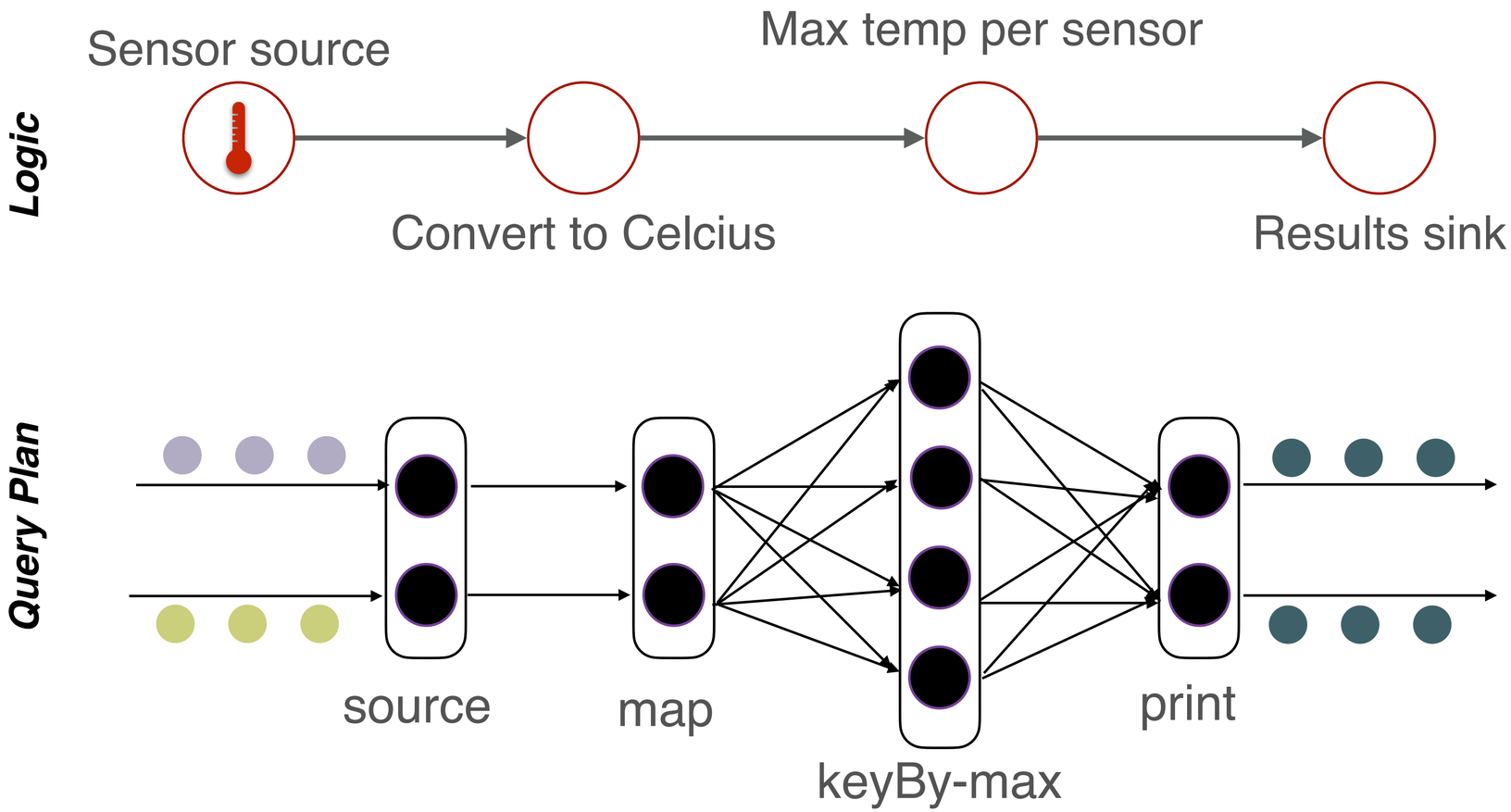
```
case class Reading(id: String, time: Long, temp: Double)

object MaxSensorReadings {
  def main(args: Array[String]) {
  val env = StreamExecutionEnvironment.getExecutionEnvironment
    val sensorData = env.addSource(new SensorSource)
    val maxTemp = sensorData
      .map(r => Reading(r.id,r.time,(r.temp-32)*(5.0/9.0)))
      .keyBy(_.id)
      .max("temp")
    maxTemp.print()
    env.execute("Compute max sensor temperature”)
  }
}
```

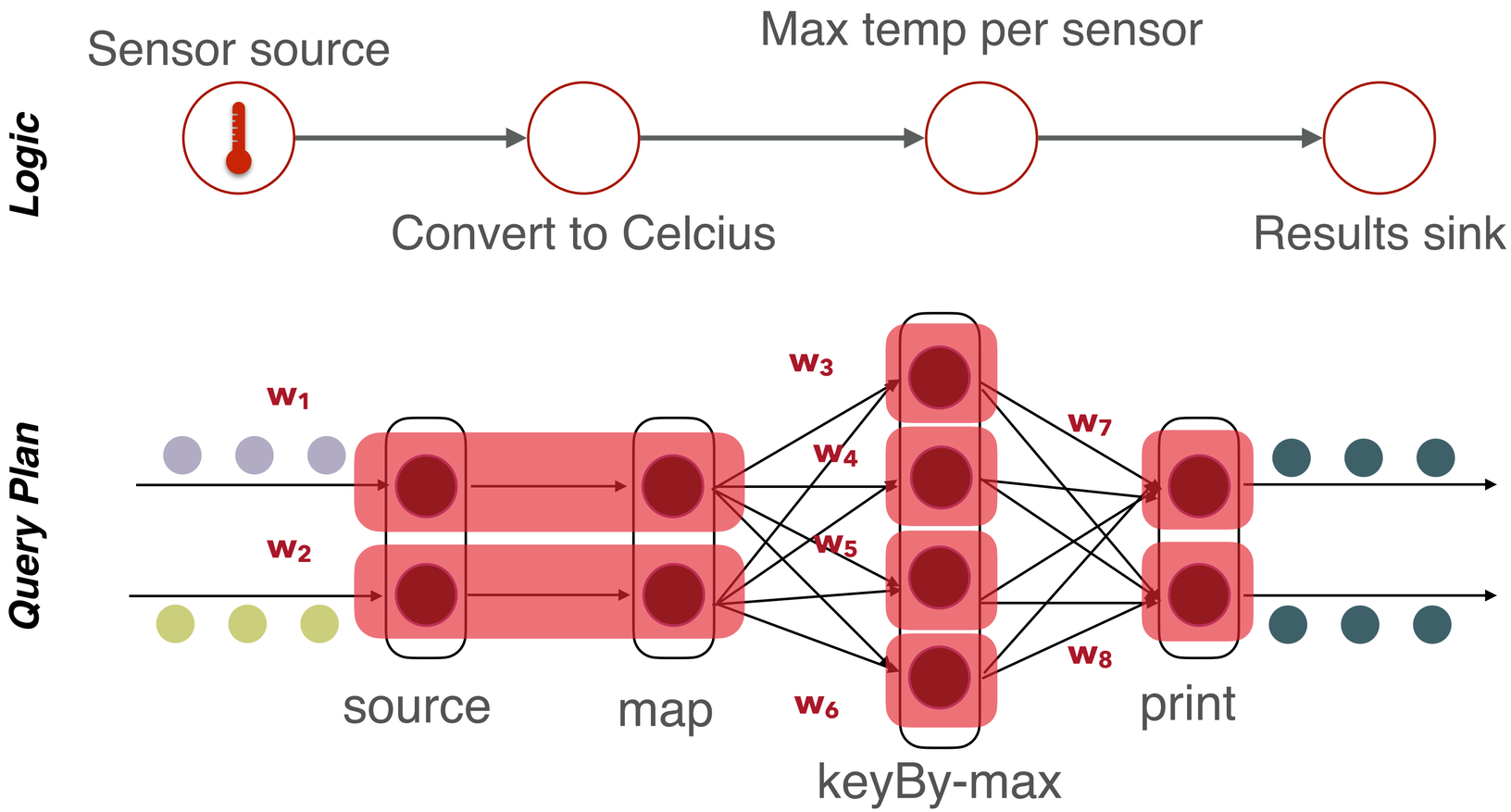
Example: Sensor Readings

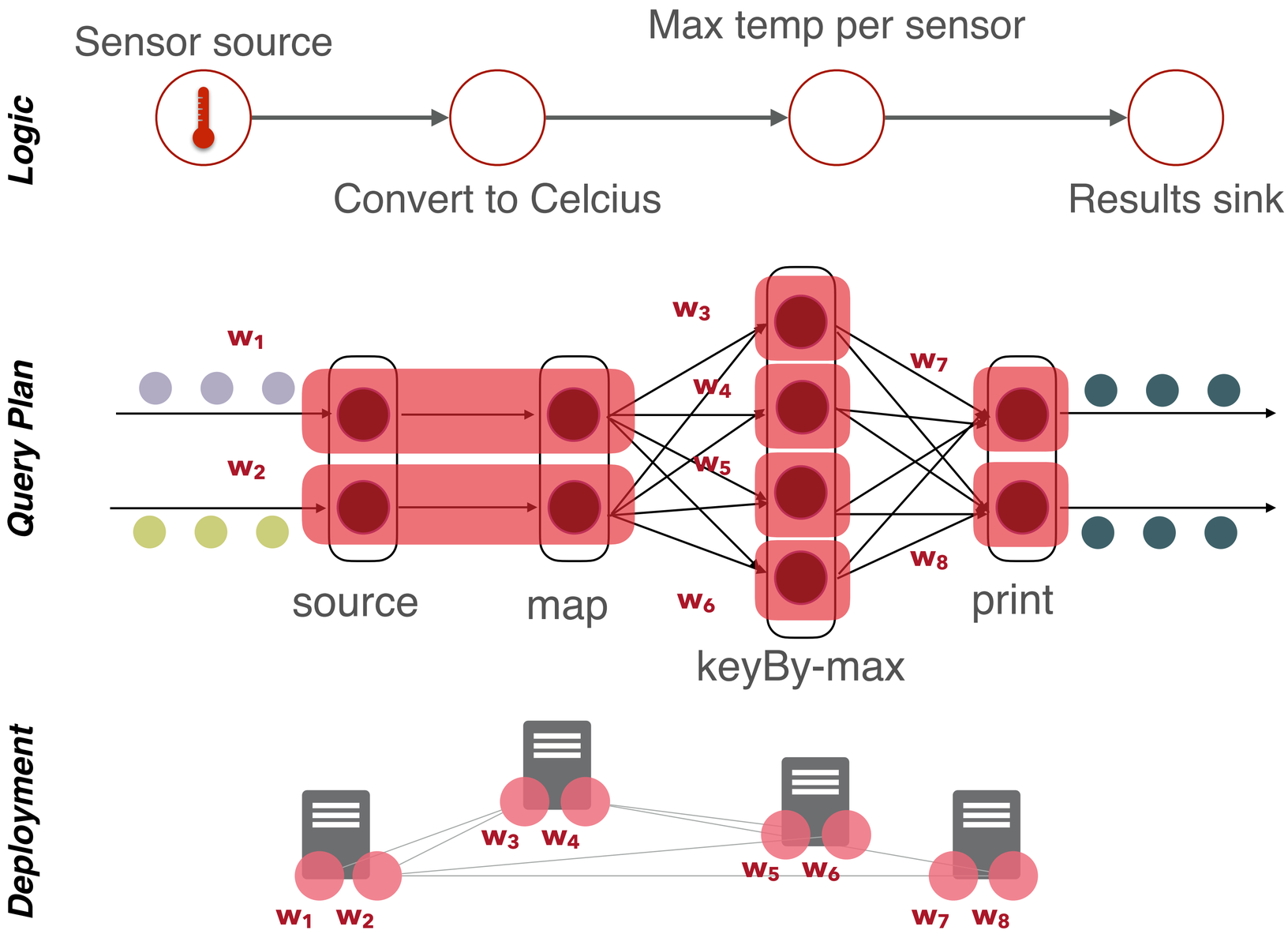
```
case class Reading(id: String, time: Long, temp: Double)

object MaxSensorReadings {
  def main(args: Array[String]) {
  val env = StreamExecutionEnvironment.getExecutionEnvironment
    val sensorData = env.addSource(new SensorSource)
    val maxTemp = sensorData
      .map(r => Reading(r.id, r.time, (r.temp-32)*(5.0/9.0)))
      .keyBy(_.id)
      .max("temp")
    maxTemp.print()
    env.execute("Compute max sensor temperature”)
  }
}
```

Output and execute the program!

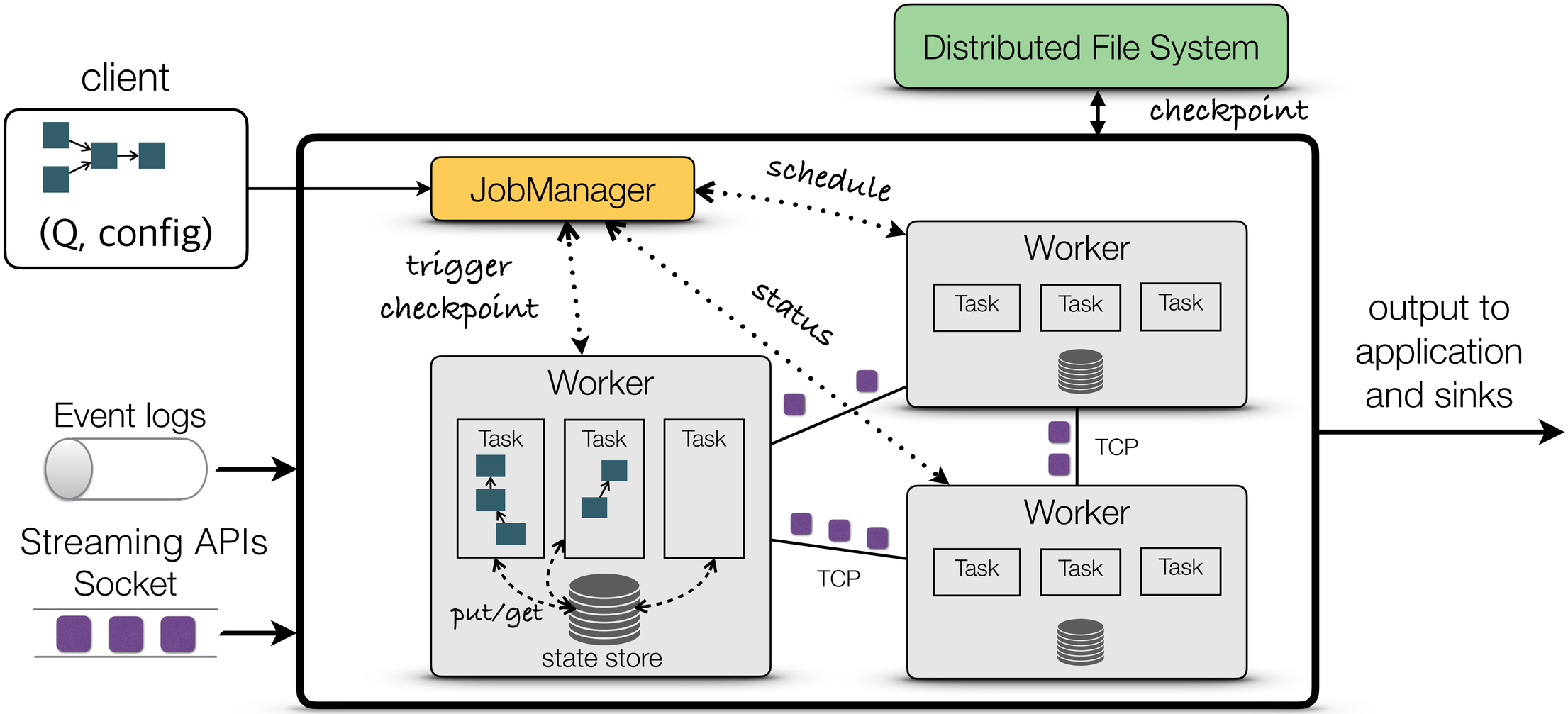

Logic
Sensor source
Max temp per sensor
Convert to Celcius
Results sink


Logic
Sensor source
Max temp per sensor
Convert to Celcius
Results sink
Query Plan
source
map
keyBy-max
print


Logic
Sensor source
Max temp per sensor
Convert to Celcius
Results sink
Query Plan
w1
w2
w3
w4
w5
w6
w7
w8
source
map
keyBy-max
print


Logic
Sensor source
Max temp per sensor
Convert to Celcius
Results sink
Query Plan
w1
w2
w3
w4
w5
w6
w7
w8
source
map
keyBy-max
print
Deployment
w1
w2
w3
w4
w5
w6
w7
w8

Job submission and architecture



JobManager

The *JobManager* coordinates the distributed execution of Flink jobs:

- Task scheduling and monitoring
- Checkpoint coordination & recovery

The *ResourceManager* handles resource allocation and provisioning and can interact with resource provides such as YARN, Mesos, Kubernetes and with standalone Flink clusters.

The *Dispatcher* provides a REST interface to submit Flink applications for execution and runs the Flink WebUI to provide information about job executions.

The *JobMaster* manages the execution of a single job. A Flink cluster might have multiple concurrent jobs running on it, but each will have its own JobMaster.

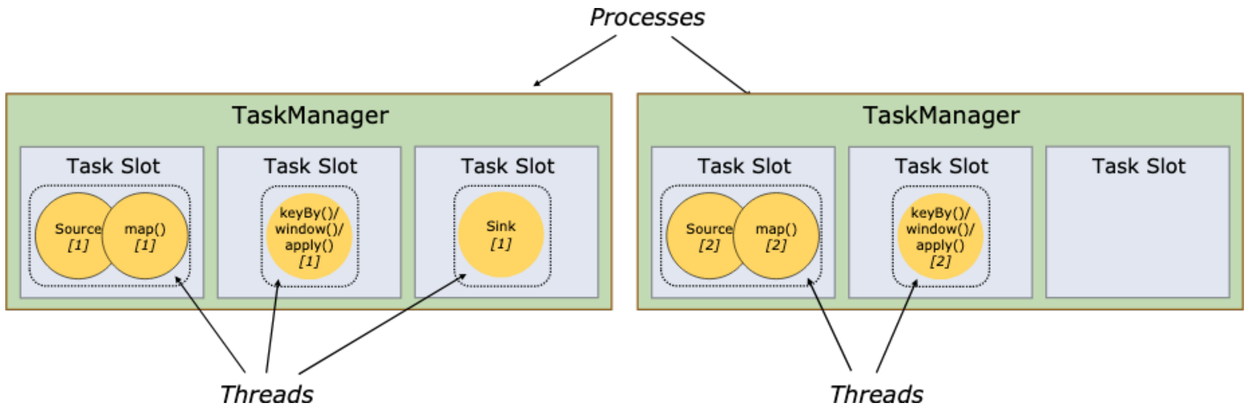
Tasks & TaskManagers

The TaskManagers are Flink's workers. They execute dataflow tasks and take care of data buffering and exchange.

Each TaskManager has a set of **task slots** where it can schedule tasks.

The number of slots defines the number of tasks that can run concurrently on the TaskManager.

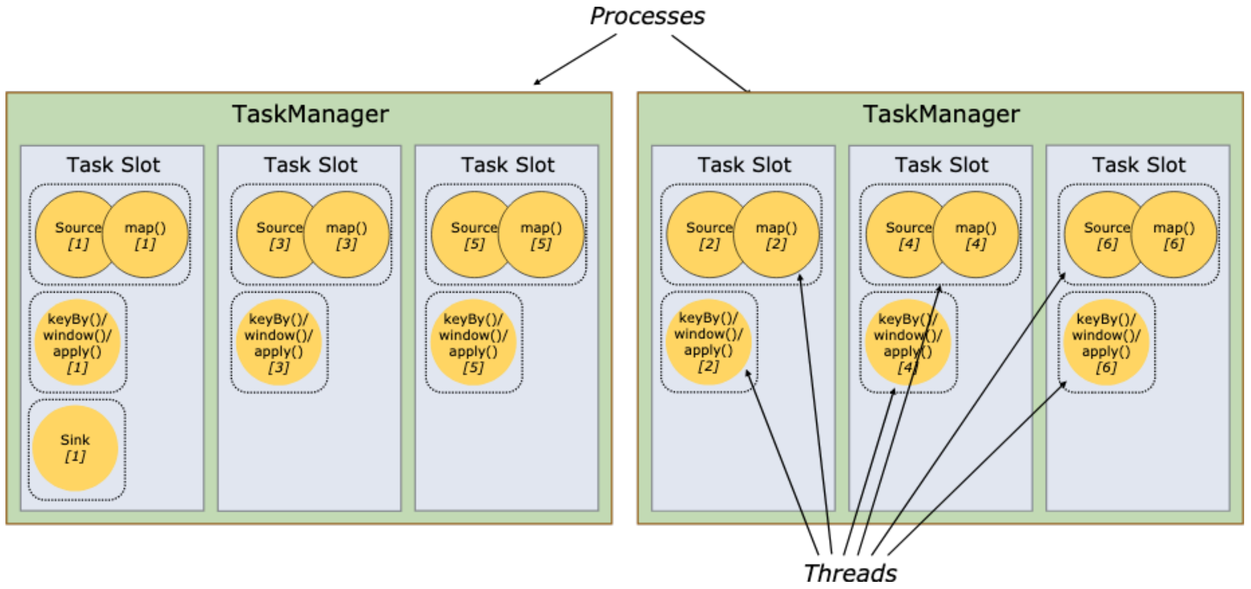
- Each TaskManager is a *JVM process*, and may execute one or more subtasks in separate threads.
- Each *task slot* represents a fixed subset of resources of the TaskManager. Slotting the resources means that a subtask will not compete with subtasks from other jobs for managed memory, but instead has a certain amount of reserved managed memory. Note that no CPU isolation, as slots only separate the managed *memory* of tasks.
- Tasks in the same JVM share TCP connections (via multiplexing) and heartbeat messages.



Slot sharing

Flink allows subtasks **to share slots** if they belong to the same job, so that one slot may hold an entire pipeline of the job.

- Slot sharing improves **resource utilization**
- Without sharing, non-intensive subtasks block as many resources as the resource intensive subtasks.
- You can manually control slot sharing using the `slotSharingGroup()` method.



Map

Datastream —> DataStream

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

Filter

Datastream —> DataStream

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

FlatMap

Datastream —> DataStream

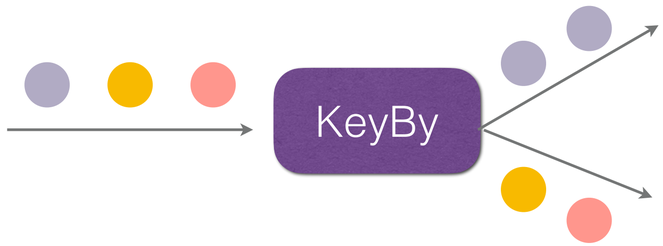
```
dataStream.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public void flatMap(String value, Collector<String> out)
        throws Exception {
        for(String word: value.split(" ")){
            out.collect(word);
        }
    }
});
```

keyBy

Datastream —> KeyedStream

Logically partitions the stream into disjoin partitions so that records with the same key end au in the same partition.

```
// Key by field "someKey"
dataStream.keyBy(value -> value.getSomeKey())

// Key by the first element of a Tuple
dataStream.keyBy(value -> value.f0)
```



```
val inputStream = env.fromElements(
    (1, 2, 2), (2, 3, 1), (2, 2, 4), (1, 5, 3))

inputStream.keyBy(0).sum(1).print()
```

```
val inputStream = env.fromElements(
    (1, 2, 2), (2, 3, 1), (2, 2, 4), (1, 5, 3))

inputStream.keyBy(0).sum(1).print()
```

what's the output for each key?

```
val inputStream = env.fromElements(
    (1, 2, 2), (2, 3, 1), (2, 2, 4), (1, 5, 3))

inputStream.keyBy(0).sum(1).print()
```

what's the output for each key?

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

```
val inputStream = env.fromElements(
    (1, 2, 2), (2, 3, 1), (2, 2, 4), (1, 5, 3))

inputStream.keyBy(0).sum(1).print()
```

what's the output for each key?

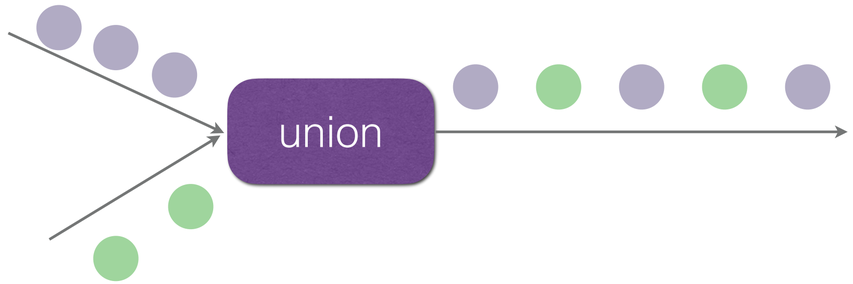
```
keyedStream.sum(0);
keyedStream.min("key");
keyedStream.max(0);
keyedStream.minBy("key");
keyedStream.maxBy(0);
```

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

Union

Datastream* —> DataStream

```
dataStream.union(otherStream1, otherStream2, …);
```



Configuration options

`conf/flink-conf.yaml` contains the configuration options as a collection of key-value pairs with format key:value

Common options you might need to adjust:

jobmanager.heap.size: JVM heap size for the JobManager (coordinator).

taskmanager.heap.size: JVM heap size for the TaskManagers (workers).

parallelism.default: Default parallelism for jobs. You can override this option by using **env.setParallelism()** in your application.

taskmanager.numberOfTaskSlots: The number of parallel operator or user function instances that a single TaskManager can run. This value is typically proportional to the number of physical CPU cores that the TaskManager's machine has (e.g., equal to the number of cores, or half the number of cores).

Flink commands

Start Flink:
./bin/start-cluster.sh

Stop Flink:
./bin/stop-cluster.sh

Run an application with no arguments:
./bin/flink run ./examples/batch/WordCount.jar

Run an application with input and output arguments:
```
./bin/flink run ./examples/batch/WordCount.jar \
                --input file:///home/user/hamlet.txt --output file:///home/user/wordcount_out
```

Run with a class entry point and arguments:
```
./bin/flink run -c org.apache.flink.examples.java.wordcount.WordCount \
                 ./examples/batch/WordCount.jar \
                 --input file:///home/user/hamlet.txt --output file:///home/user/wordcount_out
```

Resources

- Documentation
  - https://flink.apache.org/
- Community
  - https://flink.apache.org/community.html#mailing-lists
- Conference
  - http://flink-forward.org/