CS 591 K1:
Data Stream Processing and Analytics

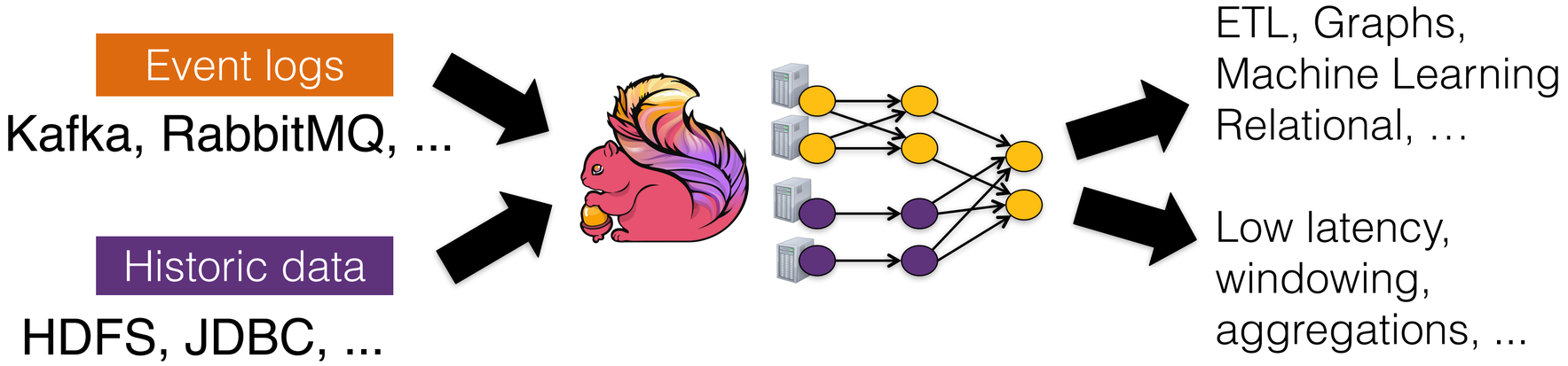
Spring 2020

1/30: Introduction to Apache Flink and Apache Kafka

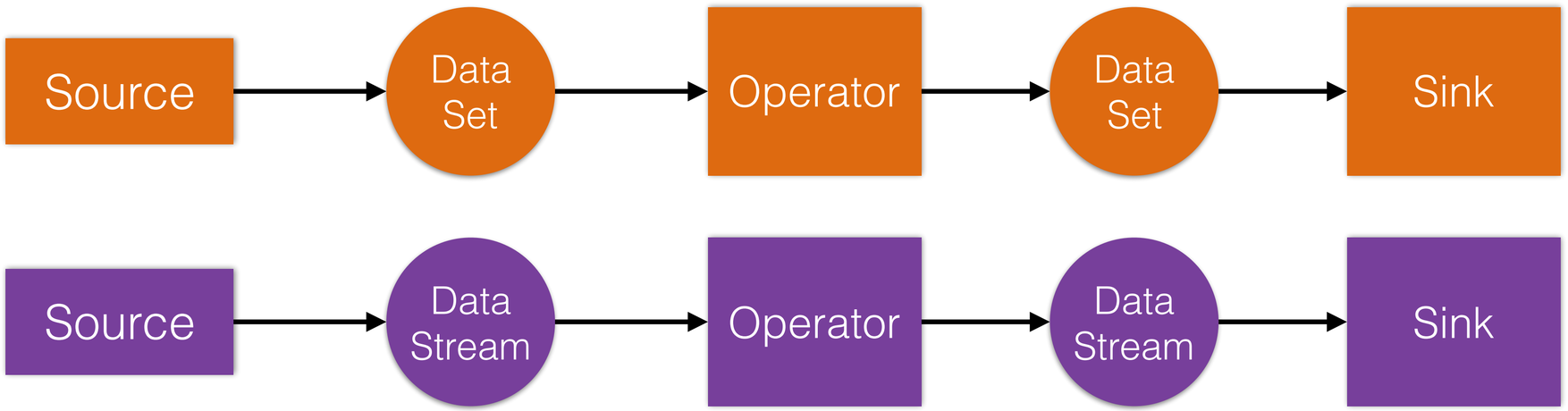
Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

Apache Flink

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



Basic API Concept

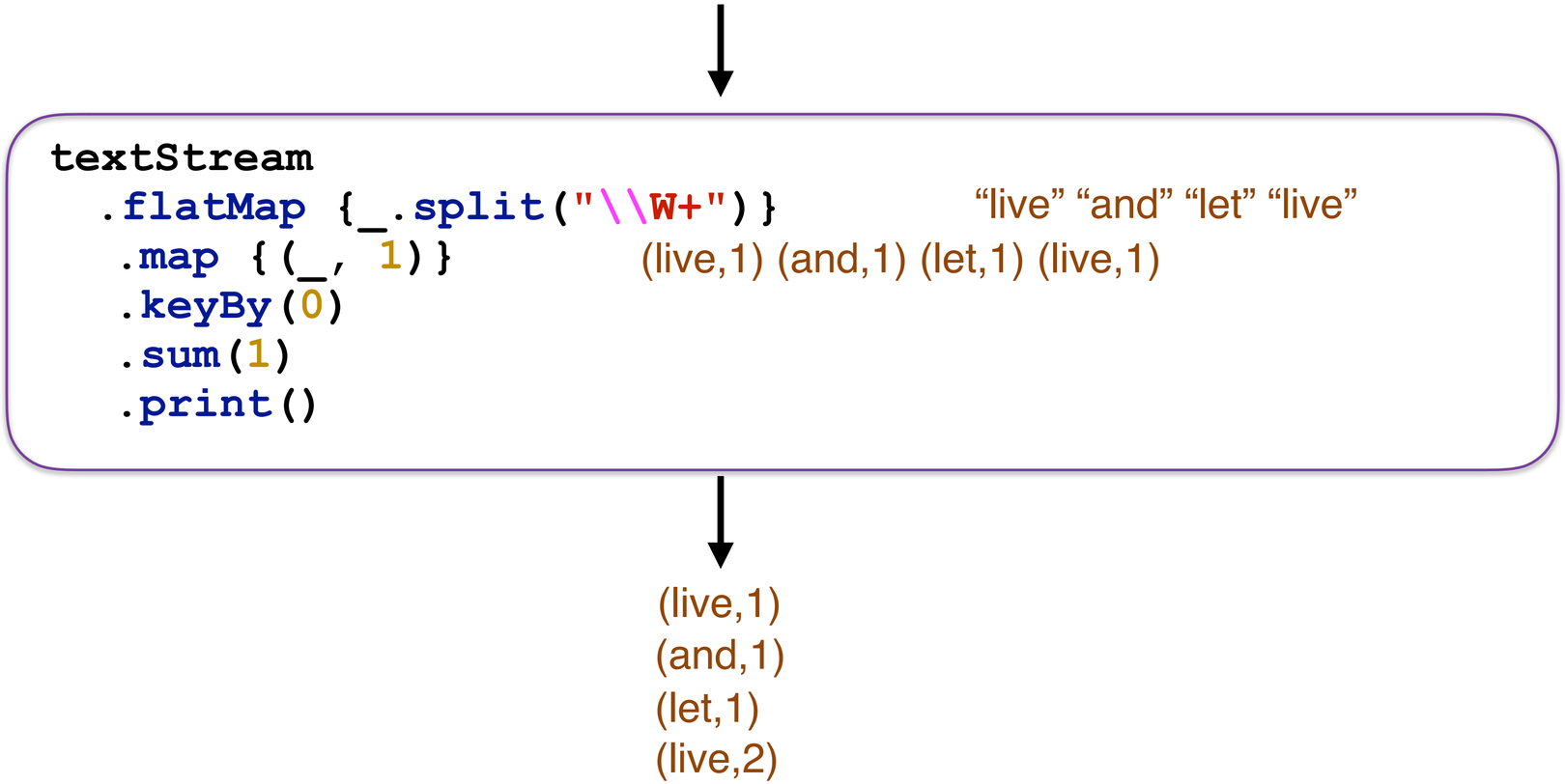


Writing a Flink Program

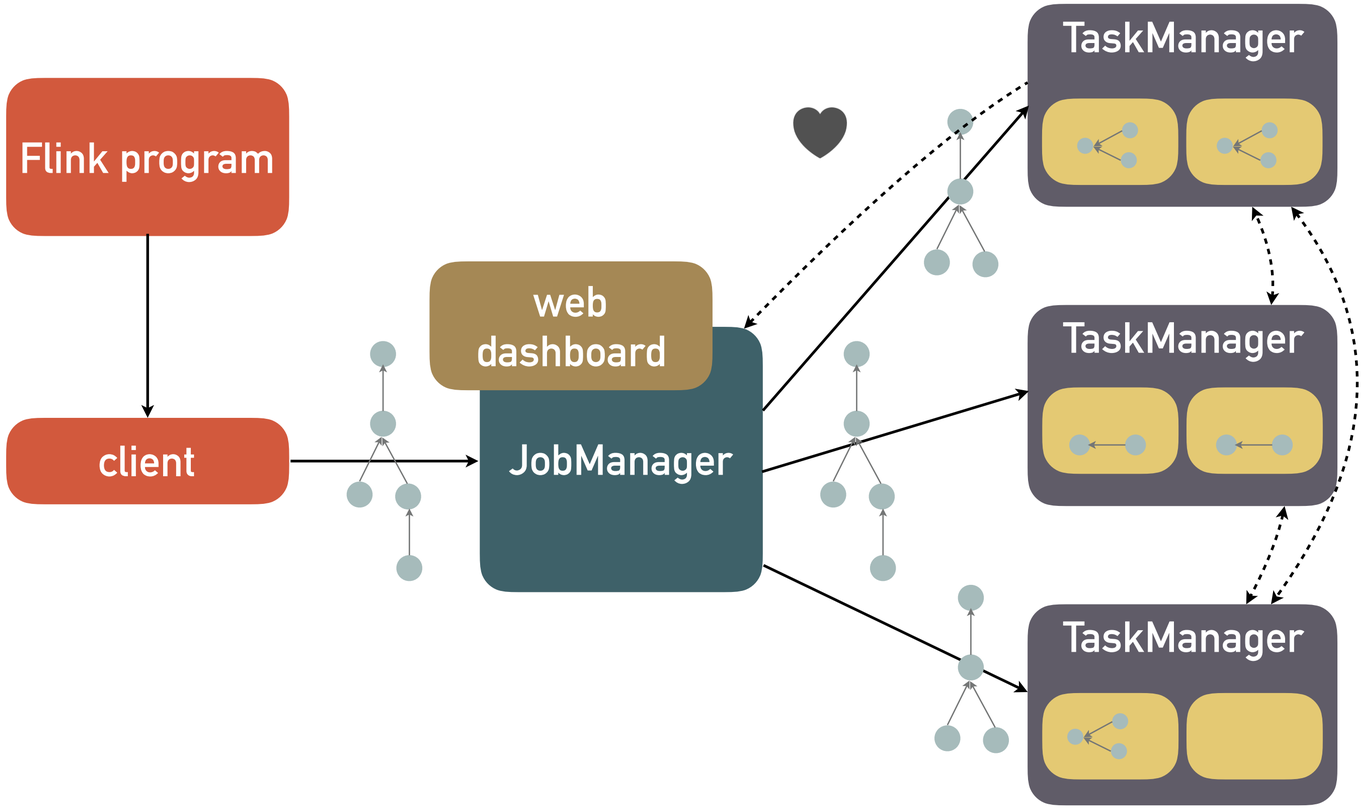
1. Bootstrap Sources
2. Apply Operators
3. Output to Sinks

Streaming word count

“live and let live”



Distributed architecture



DataStream API Basics

Example: Sensor Readings

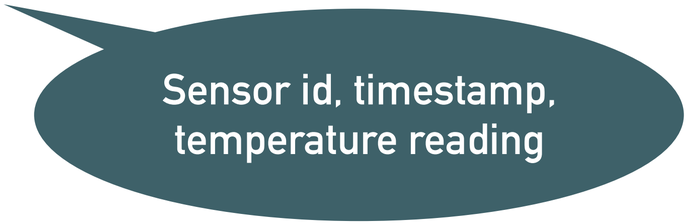
```
case class Reading(id: String, time: Long, temp: Double)


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!

Map

```
val inputStream: DataStream[(Int, Double)] = ...
val result: DataStream[Int] = inputStream.map(new MyMapFunction)

class MyMapFunction extends MapFunction[(Int, Double), Int] {
    override def map(value: (Int, Double)): Int = {
        value._1
    }
}
```

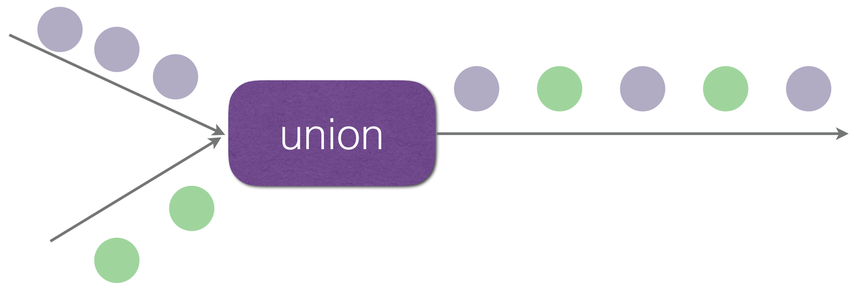
Filter

```
val inputStream: DataStream[Int] = ...
val result: DataStream[Int] = inputStream.filter(_ > 0)
```

union

```
val tempStream: DataStream[SensorReading] = ...
val humidityStream: DataStream[SensorReading] = …
val pressureStream: DataStream[SensorReading] = …

val measurements = tempStream.union(humidityStream, pressureStream)
```

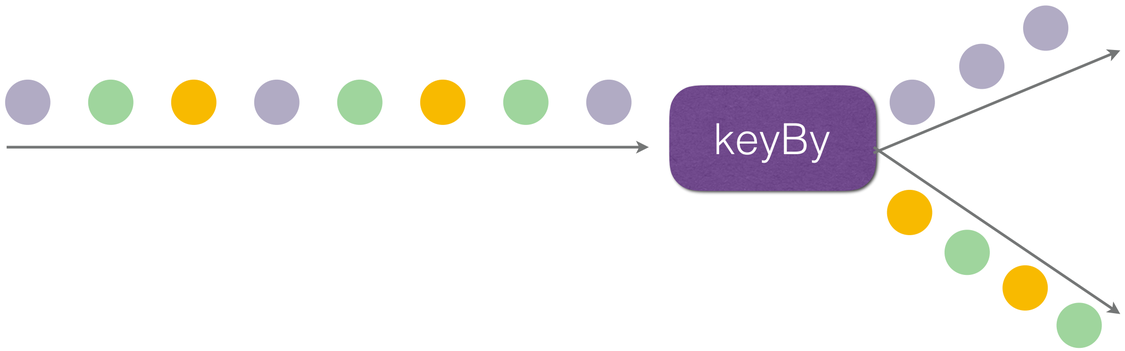


keyBy [DataStream -> KeyedStream]

```
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?



coMap / coFlatMap

```
val factors: DataStream[(String, Double)] =  …
val priceRequests: DataStream[Item] = ...

factors.connect(priceRequests).flatMap(
 new CoFlatMapFunction[(String, Double), Item, Offer] {

   // shared state between the two streams
   val factorValues: HashMap[String, Double] = HashMap.empty

   // flatMap method for the stream of factors
   override def flatMap1(value: (String, Double), out: Collector[Offer])= {
     factorValues.put(value._1, value._2)
   }

   // flatMap method for the stream of price requests
   override def flatMap2(item: Item, out: Collector[Offer]) = {
     out.collect(computePrice(item, factorValues))
   }
 })
```

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/

A distributed and fault-tolerant publish-subscribe messaging system and serves as the ingestion, storage, and messaging layer for large production streaming pipelines.

Kafka is commonly deployed on a cluster of one or more servers.

Terminology

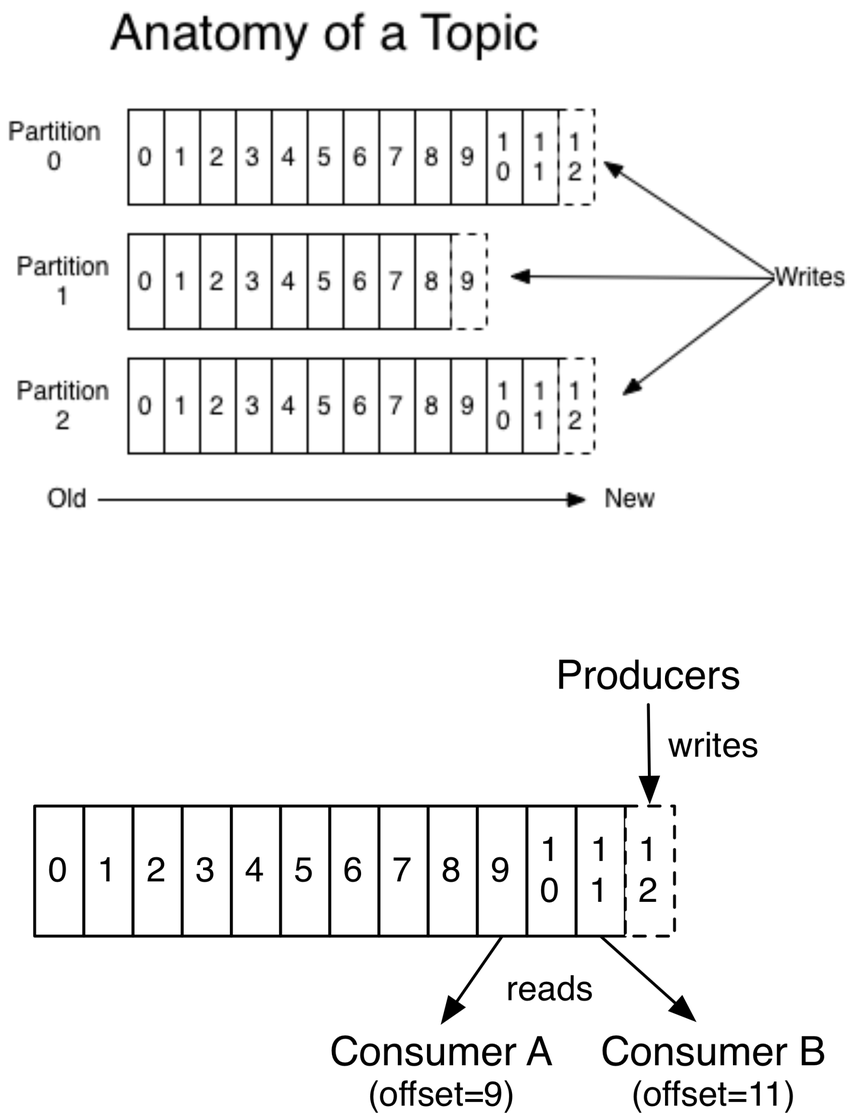
A **topic** identifies a category of stream records stored in a Kafka cluster.
Records consist of a key, a value, and a timestamp.

A **producer** publishes a stream of records to a Kafka topic and a **consumer** subscribes to one or more topics and processes the stream of records published in them.

Topics are multi-subscriber, i.e. a topic can have zero, one, or many consumers that subscribe to the data written to it. For each topic, the Kafka cluster maintains a **partitioned log**. Each **partition** is an ordered, immutable sequence of records that is continually appended to—a structured commit log.

An **offset** is a sequential id number assigned to records within a partition. It uniquely identifies records within each partition.

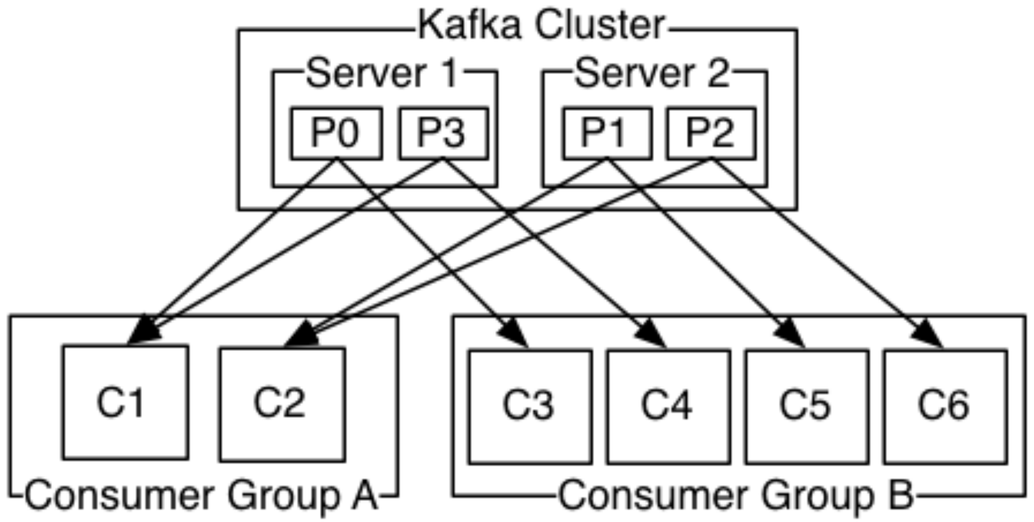
The **retention policy** defines a time period after a record is published that it is available for consumption. Records are discarded after their retention time to free up disk space.



Partitions allow the log to scale beyond a size that will fit on a single server.

Each individual partition must fit on the servers that host it, but a topic may have many partitions so it can handle an arbitrary amount of data.

The number of partitions determines the unit of parallelism.



Consumers label themselves with a *consumer group* name, and each record published to a topic is delivered to one consumer instance within each subscribing consumer group.

Consumer instances can be in separate processes or on separate machines.

If all the consumer instances have the same consumer group, then the records will effectively be **load balanced** over the consumer instances.

If all the consumer instances have different consumer groups, then each record will be **broadcast** to all the consumer processes.

Guarantees

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent:
  - if a record M1 is sent by the same producer as a record M2, and M1 is sent first, then M1 will have a lower offset than M2 and appear earlier in the log.
- A consumer instance sees records in the order they are stored in the log.
- For a topic with replication factor N, we will tolerate up to N-1 server failures without losing any records committed to the log.

Resources

- Documentation
  - https://kafka.apache.org/
- Community
  - https://kafka.apache.org/contact
- Conference
  - https://kafka-summit.org/