Kafka and Streaming

Introduction

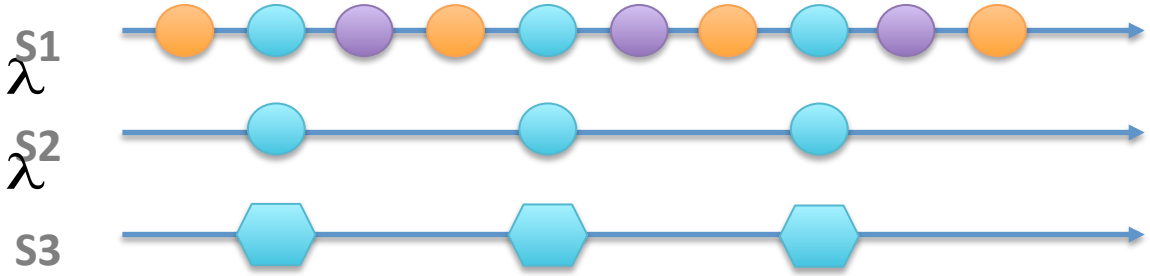
- Kafka Streaming: An alternative for streaming to and from Kafka
  - Part of the Apache Kafka
  - Powerful
    - Highly scalable, fault-tolerant
    - Rich feature set with support for both stateless and stateful processing
    - Support for fault-tolerant local state
  - Lightweight
    - Just a library integrated in Kafka (no external dependencies)
    - No need for dedicated clusters
  - Near real-time
    - Millisecond processing latency
- Kafka and Spark / Flink

Two API's

- Kafka supports a low level API that can be used to manipulate streams

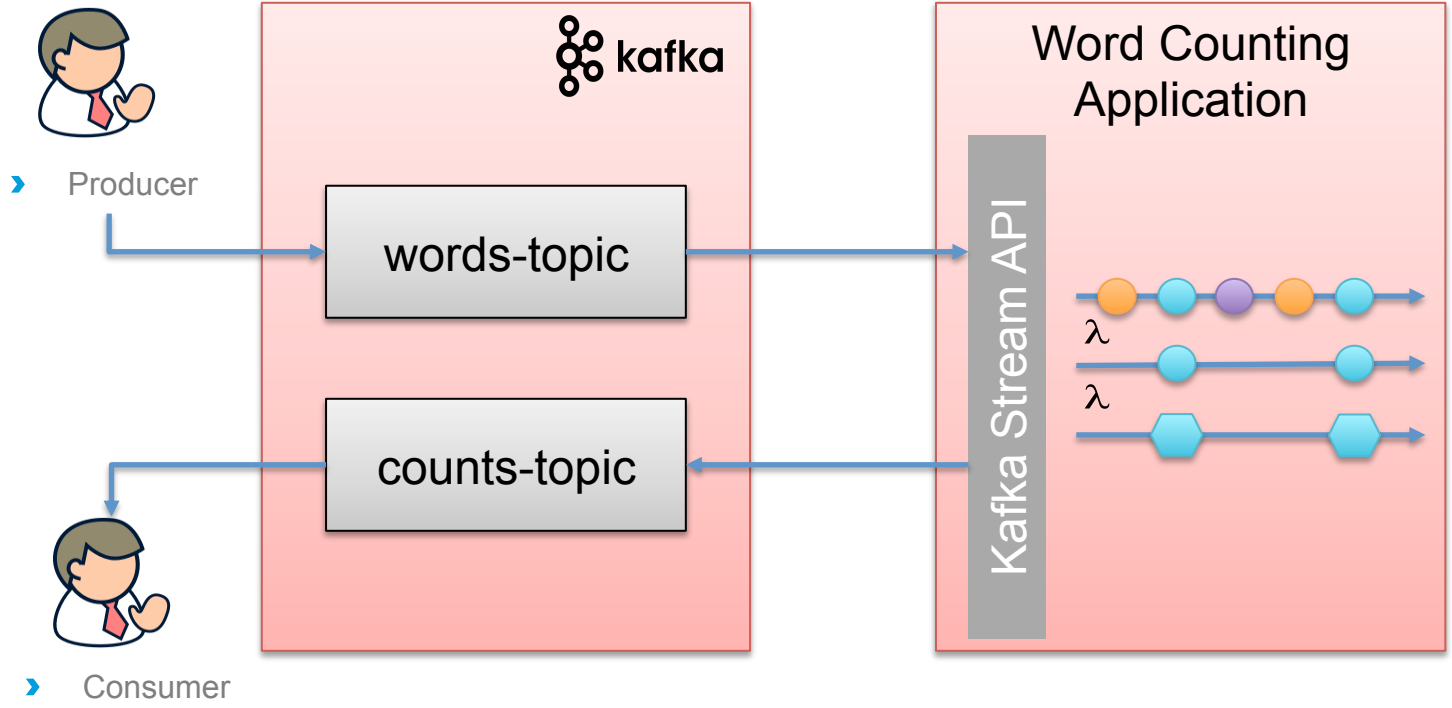
- We'll focus on the Kafka Stream `DSL`
  - Higher level
  - Closer to other streaming ideas
    - Brining in some of the functional programming paradigms

What is Stream Processing?



- Programming with streams of data (even unbounded streams)
- Typical traits
  - Functional
    - Apply a dataflow or a sequence of functions to the streams
    - Combining functions provides a powerful expression language (lambda expression)
  - Reactive
    - Process incoming stream data immediately upon receipt

The Obligatory Word Count Example



Kafka Stream Classes

- StreamsConfig
  - A wrapper around a map used for configuration
- KStreamBuilder
  - The key class for setting up the stream procoessing
  - Based on the GoF Builder Pattern
    - Fluent API
    - Constructs the processing dataflow
- KafkaStreams
  - A wrapper of the actual stream

Enable Kafka Stream (Using Maven)

```
<dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-streams</artifactId>
        <version>0.10.1.1</version>
</dependency>
```

- Kafka Streams are distributed as a simple jar file
- Simply include the jar file in your program and you'll be able to use the streaming library

Programming to Stream

```
// Configure
Properties props = new Properties();
pros.put(StreamsConfig.CLIENT_ID_CONFIG, "...")
...
StreamsConfig config = new StreamsConfig(props);

// Serialization/deserialization
Serde<String> serDeser = Serdes.String();

// Create the data processing pipeline
KStreamBuilder bld = new KStreamBuilder();
bld.stream(serDeser, serDeser, "topic")
.map(...) ...

// Create stream
KafkaStreams sp = new KafkaStreams(bld, config);

// Start stream
kafkaStreams.start();
```

Word Count Streaming

```
StreamsConfig config = new StreamsConfig(props);
Serde<String> serde = Serdes.String();

KStreamBuilder bld = new KStreamBuilder();
bld.stream(serde, serde, "words-topic")
  .flatMapValues(text -> asList(text.split(" ")))
  .map((key, word) -> new KeyValue<>(word, word))
  .countByKey(serde, "Counts")
  .toStream()
  .map((word, count) ->
       new KeyValue<>(word, word + ":" + count))
  .to(serde, serde, "counts-topic");

KafkaStreams s= new KafkaStreams(bld, config);
s.start();
```

kafkacat – A Useful Tool

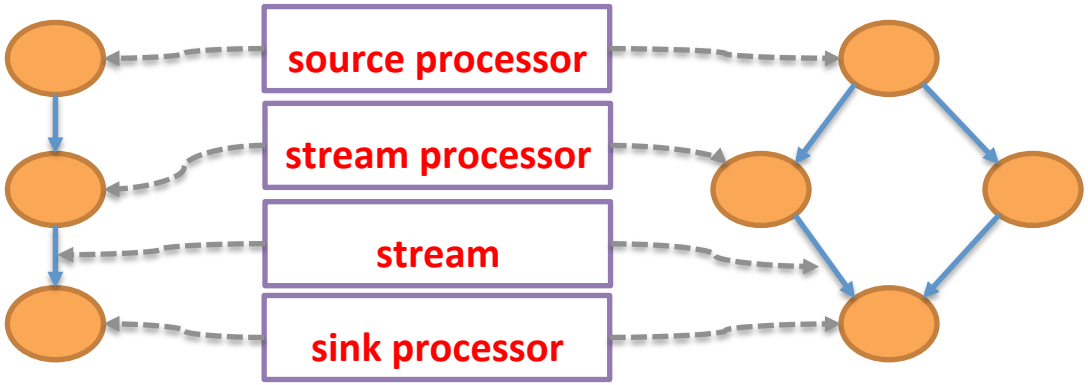
- kafkacat is a tool created by Magnus Edenhill
  - https://github.com/edenhill/kafkacat
  - Doesn't require JVM
  - Similar to 'netcat'
  - Can act as a Producer and consumer
  - Useful also to query metadata from Kafka
- To provide the word list to the application, we could simply use kafkacat instead of writing new producers and consumers
- Use as a producer
  - kafkacat −P −b myBroker −t topic1
- Use as a consumer
  - kafkacat −b myBroker −G mygroup topic1 topic 2

Installing kafkacat

- The installation of kafkacat is trivial on Mac most Linux distrobutions
  - Mac
    - brew install kafkacat
  - Ubuntu or Debian
    - [sudo] apt-get install kafkacat

- Our Kafka docker image is using Alpine distribution so if you want to run it in the docker image you'll have to:
  - apk add --update alpine-sdk bash python cmake
  - curl https://codeload.github.com/edenhill/kafkacat/tar.gz/master | tar xzf - && cd kafkacat-* && bash ./bootstrap.sh

Processor Topology



- The topology defines the computational logic of the data processing pipeline
- The topology typically forms a chain, but more advanced computation may be defined as a graph

Processing and Time

- In streaming time is an important design consideration
  - When working with concepts such as windowing, it is essential to establish which time to use
- Event-time
  - The point in time when an event or data record was created by the source
- Processing-time
  - The point in time when an event was processed by the stream processor
- Ingestion-time
  - The point in time when an event was stored in a topic partition by one of the Kafka brokers

Stateful Stream Processing

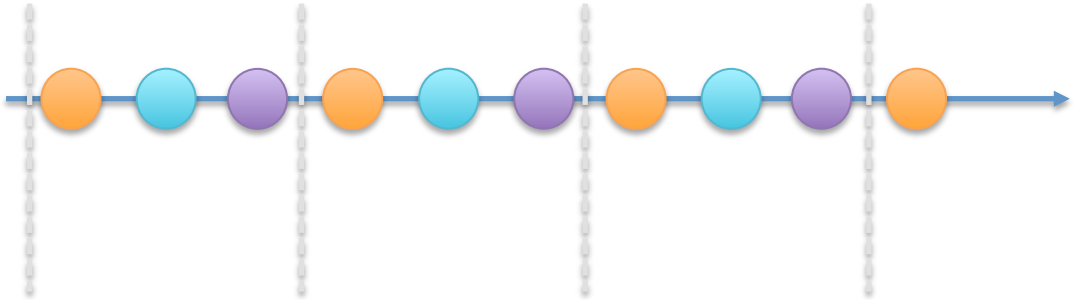
- Typically we try to make our stream processing stateless

- However, for some use cases, stateful streaming may be essential
- The Kafka Stream DSL provides support for stateful processing

Stream vs. Table

- Kafka streams introduces the concept of tables, where
  - A stream can be viewed as a table: KStream → KTable
  - A table can be viewed as a stream: KTable → KStream
  - (called stream-table duality)
- Streams as tables
  - A stream can be considered a changelog of a table
  - One can convert the stream into a table by replaying the event stream
- Table as streams
  - A table can be be converted into a stream by iterating over each key-value entry

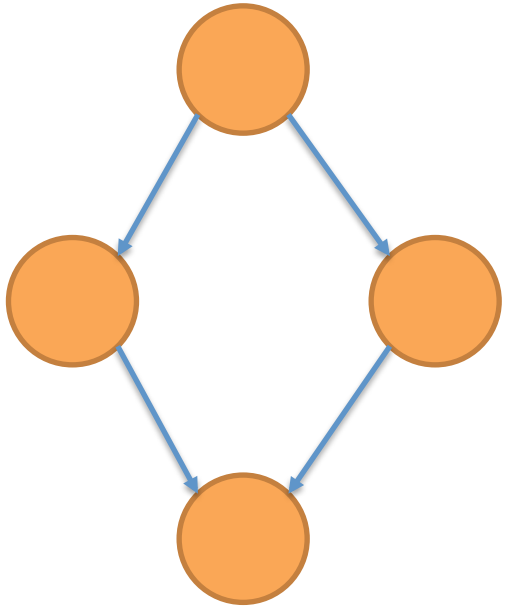
Windowing



- We may have to process data in time buckets, called windows
- Typically as part of some form of aggregation
- Windowing operations are available in the Kafka Stream DSL
- A window has a retention period to handle possible late arriving events

Parallel Stream Processing Pipelines

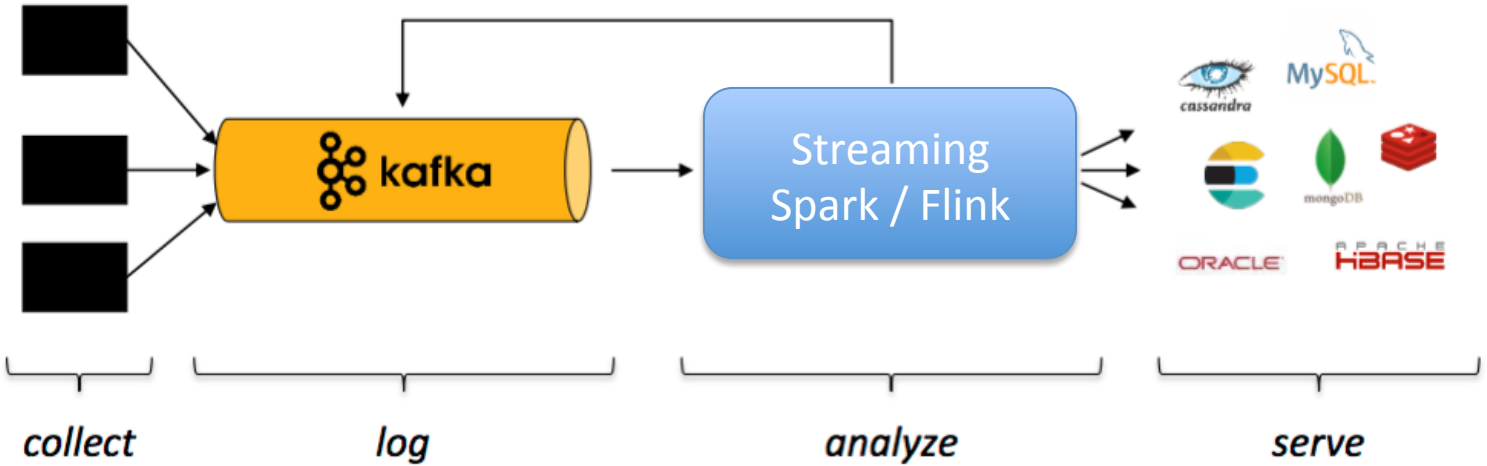
- The Kafka Stream DSL supports ways to parallelize and rendezvous processing steps
- Join
  - This operation merges two streams into a single stream
- Aggregation
  - Takes one input stream and yields a new stream by combining multiple input events into a single output event



Backpressure

- Kafka Streams don't need a backpressure mechanism
- Kafka uses a depth-first processing strategy
  - Each event consumed from Kafka goes through the complete processor topology before the next message is processed

Kafka with Spark or Flink



- Gather and backup streams
- Offer streams for consumption
- Provide stream recovery

- Analyze and correlate streams
- Create derived streams and state
- Provide these to downstream systems

Spark Streaming and Kafka

- We can set up Spark to use Kafka stream as the source of events

- Code structure:
  - A bit of configuration for Spark Streaming
  - A bit of configuration for Kafka parameters
  - Create stream from Kafka

- At this point, we have a DStream and we can do the normal Spark Streaming processing on it

Spark Streaming with Kafka – Spark Configuration

```
// Consume command line parameters
val Array(brokers, topics, interval) = args

// Create Spark configuration
val sparkConf = new SparkConf().setAppName("SparkKafka")

// Create streaming context, with batch duration in ms
val ssc = new StreamingContext(sparkConf, Duration(interval.toLong))
ssc.checkpoint("./output")
```

Spark Streaming with Kafka – Kafka Configuration

```
// Create a set of topics from a string
val topicsSet = topics.split(",").toSet

// Define Kafka parameters
val kafkaParams = Map[String, Object](
  "bootstrap.servers" -> brokers,
  "key.deserializer" -> classOf[StringDeserializer],
  "value.deserializer" -> classOf[StringDeserializer],
  "group.id" -> "use_a_separate_group_id_for_each_stream",
  "auto.offset.reset" -> "latest",
  "enable.auto.commit" -> (false: java.lang.Boolean))
```

Create a Kafka Stream in Spark and Use It

```
// Create a Kafka stream
val stream = KafkaUtils.createDirectStream[String, String](
  ssc, PreferConsistent, Subscribe[String, String](topicsSet,kafkaParams))
// Get messages - lines of text from Kafka
val lines = stream.map(consumerRecord => consumerRecord.value)
// Split lines into words
val words = lines.flatMap(_.split(" "))
// Map every word to a tuple
val wordMap = words.map(word => (word, 1))
// Count occurrences of each word
val wordCount = wordMap.reduceByKey(_ + _)
//Print the word count
wordCount.print()
```

Summary

- Kafka provides a stream processing library
- Two APIs
  - Low level API
  - Kafka Stream DSL
- The Kafka Stream DSL enables the definition of processing topologies
- Kafka streams support stateful stream processing
- Kafka has a reach support of functions and features
  - Windowing
  - Aggregation
  - Join
  - Functions (map, flatMap, etc.)
- Integration with other streaming systems: Spark, Flink, …

Lab

- We have 3 labs for streaming
- Lab 1: Word count using Kafka Streams
  - I will show this in slides
- Lab 2: Word count in Spark
  - I'll leave that for you to do after the class
- Lab 3: IoT example in Spark
  - I'll walk you through the code
- Example 4: IoT example implemented in Kafka Streams
  - I'll walk you through the code

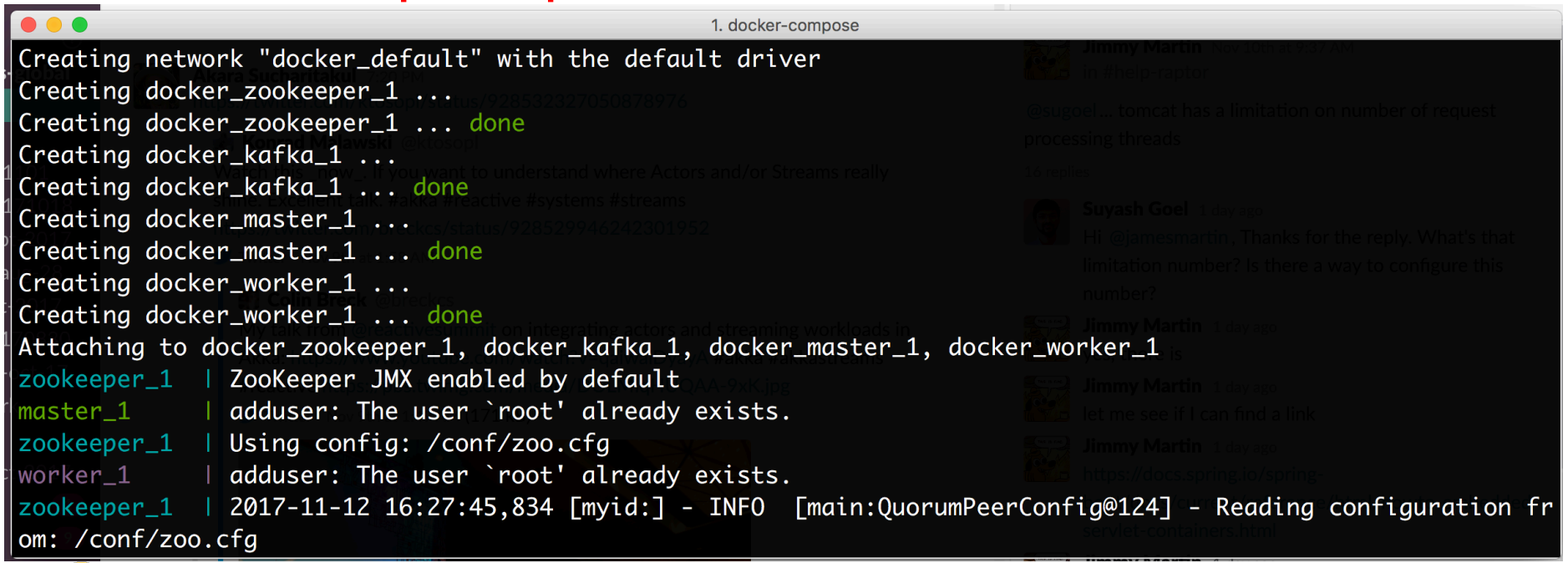
Step 1: Setup Your docker.yml File

- We've done this before…
- The **`docker-compose.yml`** file is in the directory:
  - **`kafka-lab/labs/06-Streaming/docker`**
- Modify the **`KAFKA_ADVERTISED_HOST_NAME`** to refer to the IP of your machine
- Notice that in this docker-compose file we have more processes:
  - ZooKeeper
  - Kafka
  - Spark Master
  - Spark Worker

Step 2: Start Your Docker Processes

- Start your processes with:

  `docker-compose up`



Step 3: Create Some Topics

- Start a new terminal (you may want to keep this terminal to eventually shutdown the docker containers)
- For the word-count we'll have two topics:
  - stream-input
  - stream-output
- Use the docker-compose exec to create the topics:

```
sh
LM-SJN-21001415:docker pgraff$ docker-compose exec kafka /opt/kafka_2.11-0.10.1.1/bin/kafka-topics.sh --creat
e --zookeeper zookeeper:2181 --replication-factor 1 --partitions 1 --topic stream-input
Created topic "stream-input".
LM-SJN-21001415:docker pgraff$ docker-compose exec kafka /opt/kafka_2.11-0.10.1.1/bin/kafka-topics.sh --creat
e --zookeeper zookeeper:2181 --replication-factor 1 --partitions 1 --topic stream-output
Created topic "stream-output".
LM-SJN-21001415:docker pgraff$
```

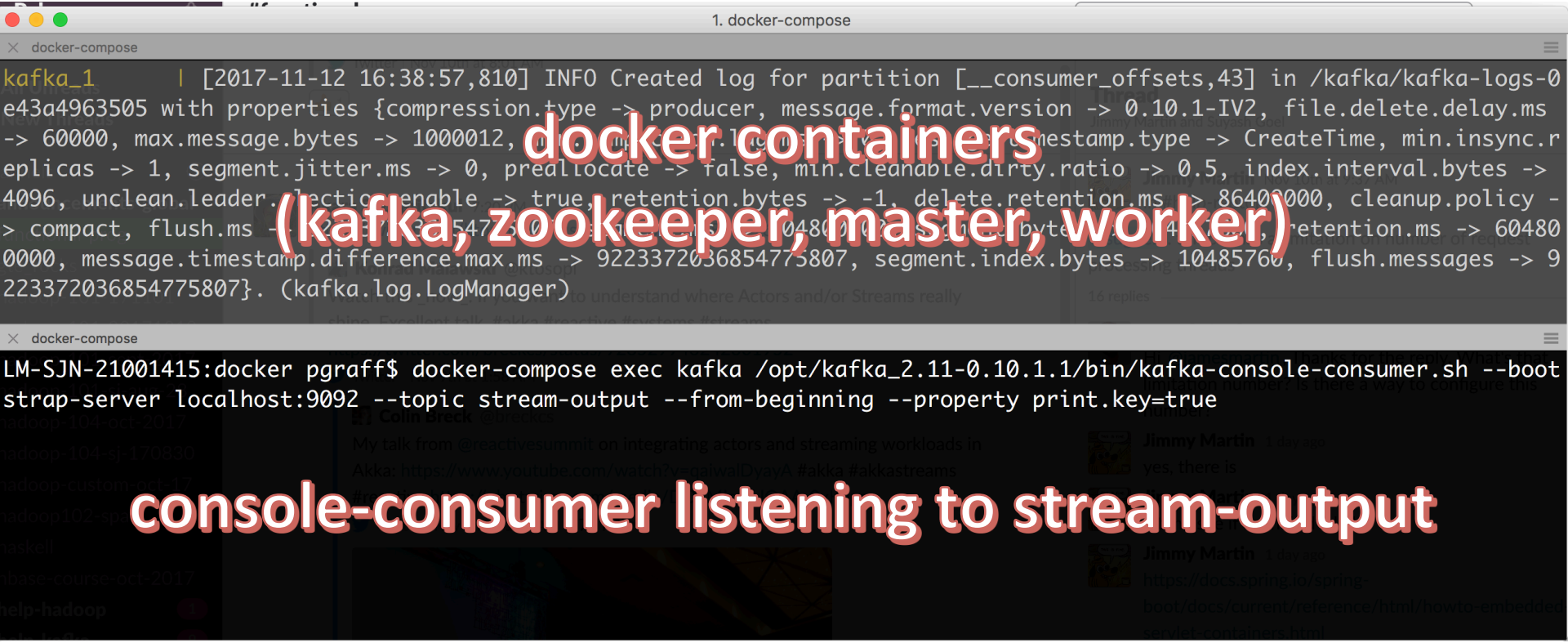
Step 4: Create a Console Consumer

- Let's create a console consumer to listen to the end result
- In a separate terminal
  - Change to the docker directory
  - Run a console consumer listening to the stream-output:

```
docker-compose exec kafka
/opt/kafka/bin/kafka-console-consumer.sh
--bootstrap-server localhost:9092
--topic stream-output
--from-beginning
--property print.key=true
```

Prints the key as well as the value of each message

Where Are We?

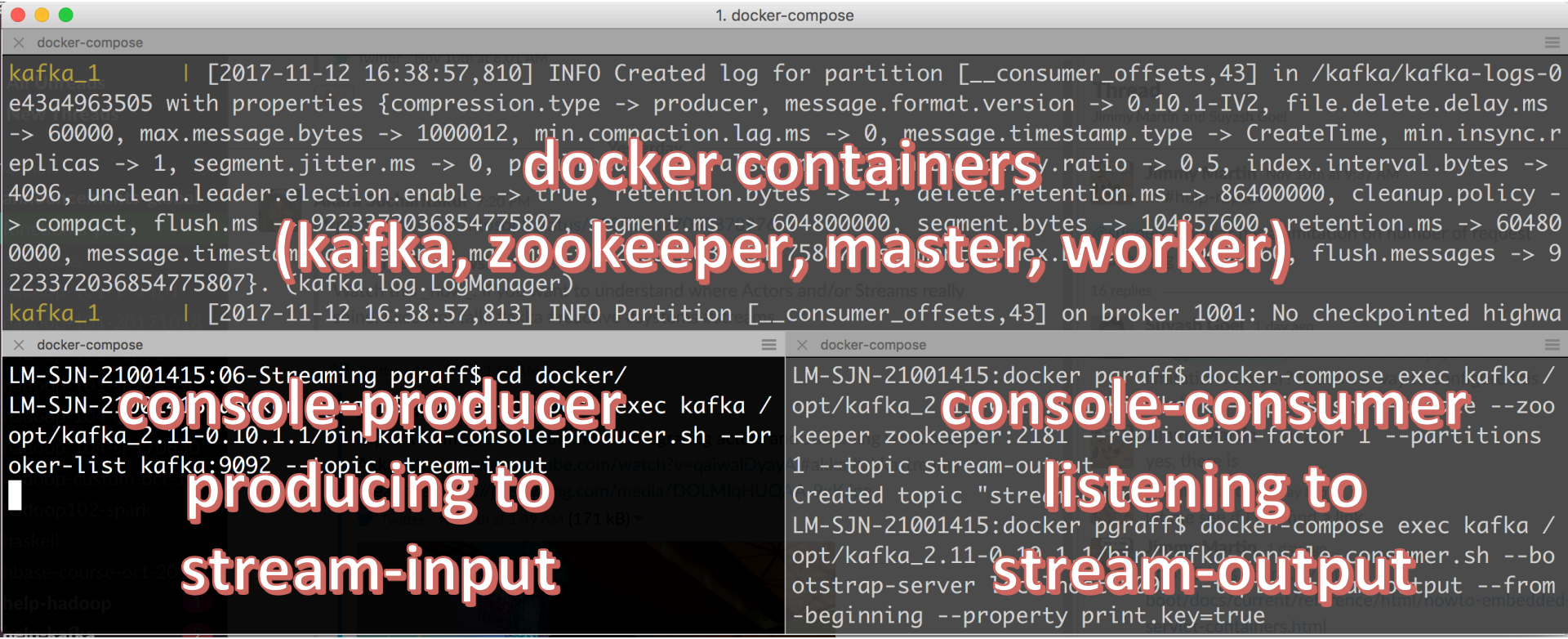

1. docker-compose
docker-compose
kafka_1 | [2017-11-12 16:38:57,810] INFO Created log for partition [__consumer_offsets,43] in /kafka/kafka-logs-0
e43a4963505 with properties {compression.type -> producer, message.format.version -> 0.10.1-IV2, file.delete.delay.ms
-> 60000, max.message.bytes -> 1000012, message.timestamp.type -> CreateTime, min.insync.r
eplicas -> 1, segment.jitter.ms -> 0, preallocate -> false, min.cleanable.dirty.ratio -> 0.5, index.interval.bytes ->
4096, unclean.leader.election.enable -> true, retention.bytes -> -1, delete.retention.ms -> 86400000, cleanup.policy -
> compact, flush.ms
0000, message.timestamp.difference.max.ms -> 9223372036854775807, segment.index.bytes -> 10485760, flush.messages -> 9
223372036854775807}. (kafka.log.LogManager)
docker containers
(kafka, zookeeper, master, worker)
docker-compose
LM-SJN-21001415:docker pgraff$ docker-compose exec kafka /opt/kafka_2.11-0.10.1.1/bin/kafka-console-consumer.sh --boot
strap-server localhost:9092 --topic stream-output --from-beginning --property print.key=true
console-consumer listening to stream-output

Step 5: Create a Console Producer

- We also need to be able to produce the input data
- Open a terminal and change directory to the docker directory
- Run the console-producer:

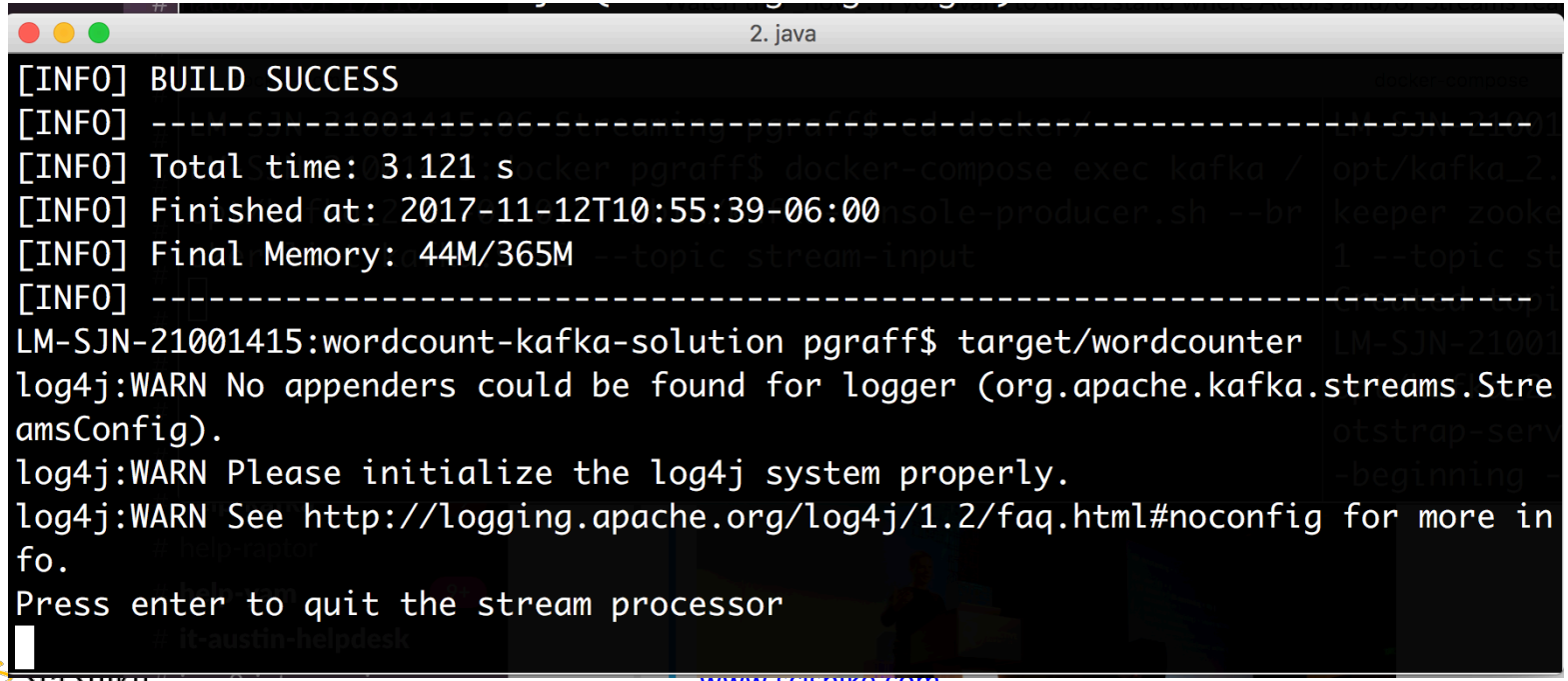
```
 docker-compose exec kafka
/opt/kafka/bin/kafka-console-producer.sh
--broker-list kafka:9092
--topic stream-input
```

Where Are We?

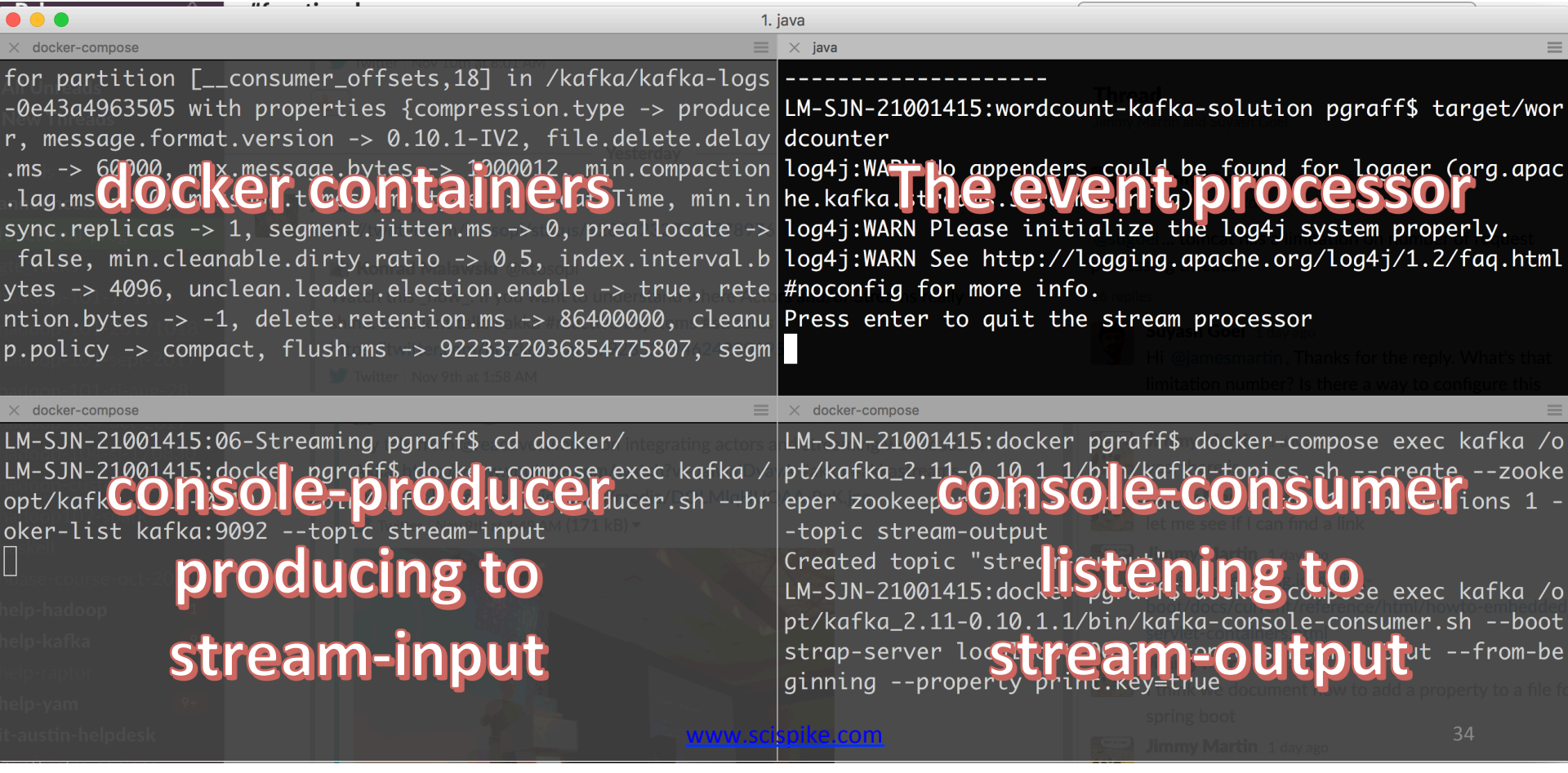

1. docker-compose
docker-compose
kafka_1 | [2017-11-12 16:38:57,810] INFO Created log for partition [__consumer_offsets,43] in /kafka/kafka-logs-0
e43a4963505 with properties {compression.type -> producer, message.format.version -> 0.10.1-IV2, file.delete.delay.ms
-> 60000, max.message.bytes -> 1000012, min.compaction.lag.ms -> 0, message.timestamp.type -> CreateTime, min.insync.r
eplicas -> 1, segment.jitter.ms -> 0, p
4096, unclean.leader.election.enable -> true, retention.bytes -> -1, delete.retention.ms -> 86400000, cleanup.policy -
> compact, flush.ms -> 9223372036854775807, segment.ms -> 604800000, segment.bytes -> 104857600, retention.ms -> 60480
0000, message.timest
223372036854775807}. (kafka.log.LogManager)
kafka_1 | [2017-11-12 16:38:57,813] INFO Partition [__consumer_offsets,43] on broker 1001: No checkpointed highwa
docker containers
(kafka, zookeeper, master, worker)
docker-compose
LM-SJN-21001415:06-Streaming pgraff$ cd docker/
LM-SJN-21001415:docker pgraff$ docker-compose exec kafka /
opt/kafka_2.11-0.10.1.1/bin/kafka-console-producer.sh --br
oker-list kafka:9092 --topic stream-input
console-producer
producing to
stream-input
docker-compose
LM-SJN-21001415:docker pgraff$ docker-compose exec kafka /
opt/kafka_2.11-0.10.1.1/bin/kafka-console-consumer.sh --zoo
keeper zookeeper:2181 --replication-factor 1 --partitions
1 --topic stream-output
Created topic "stream-output".
LM-SJN-21001415:docker pgraff$ docker-compose exec kafka /
opt/kafka_2.11-0.10.1.1/bin/kafka-console-consumer.sh --bo
otstrap-server kafka:9092 --topic stream-output --from
-beginning --property print.key=true
console-consumer
listening to
stream-output

Step 6: Build and Run the Stream Processor

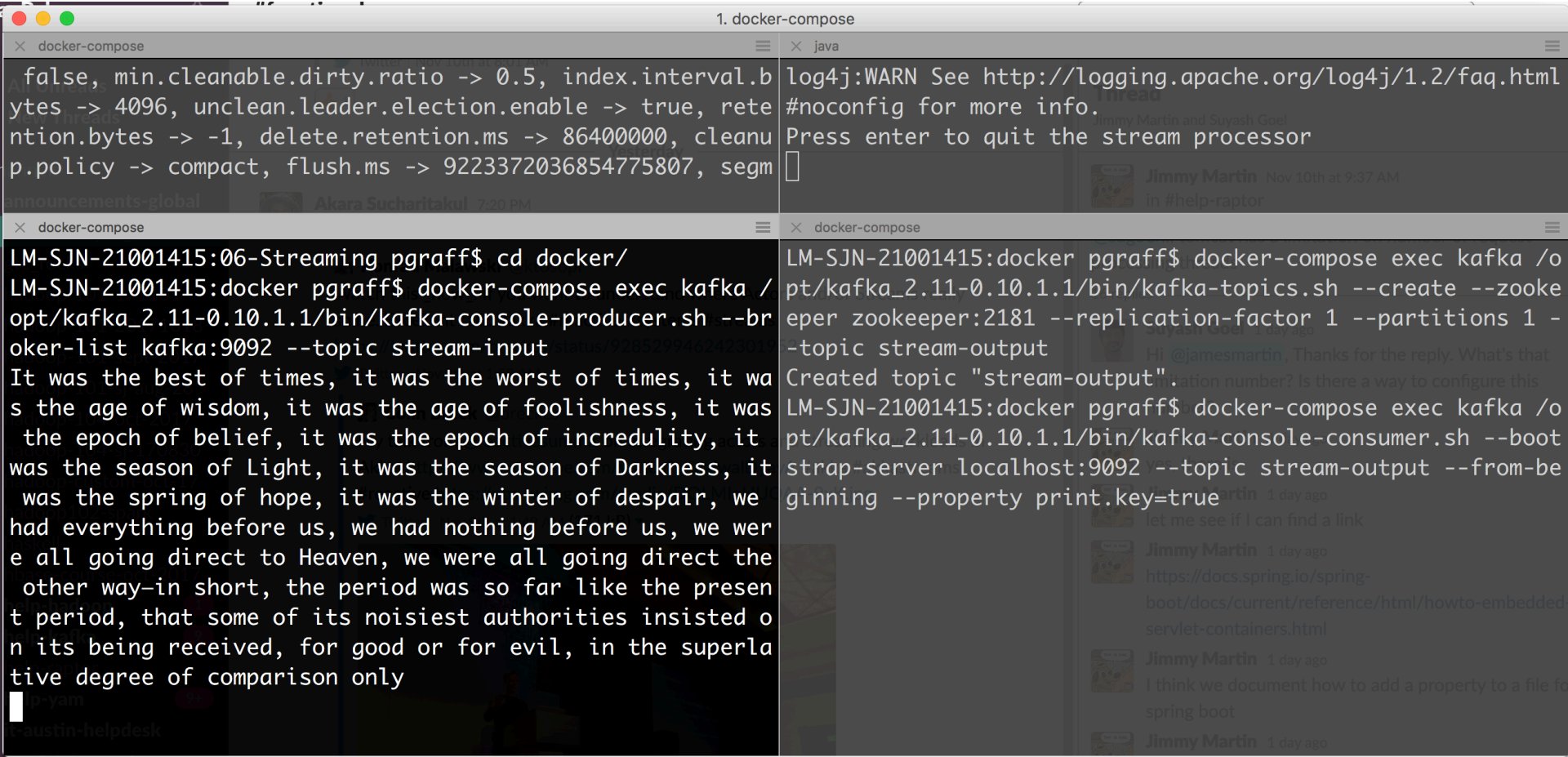
- Open a terminal and change directory to the **wordcount-kafka-solution**
- Run **mvn package** and after that **target/wordcounter**



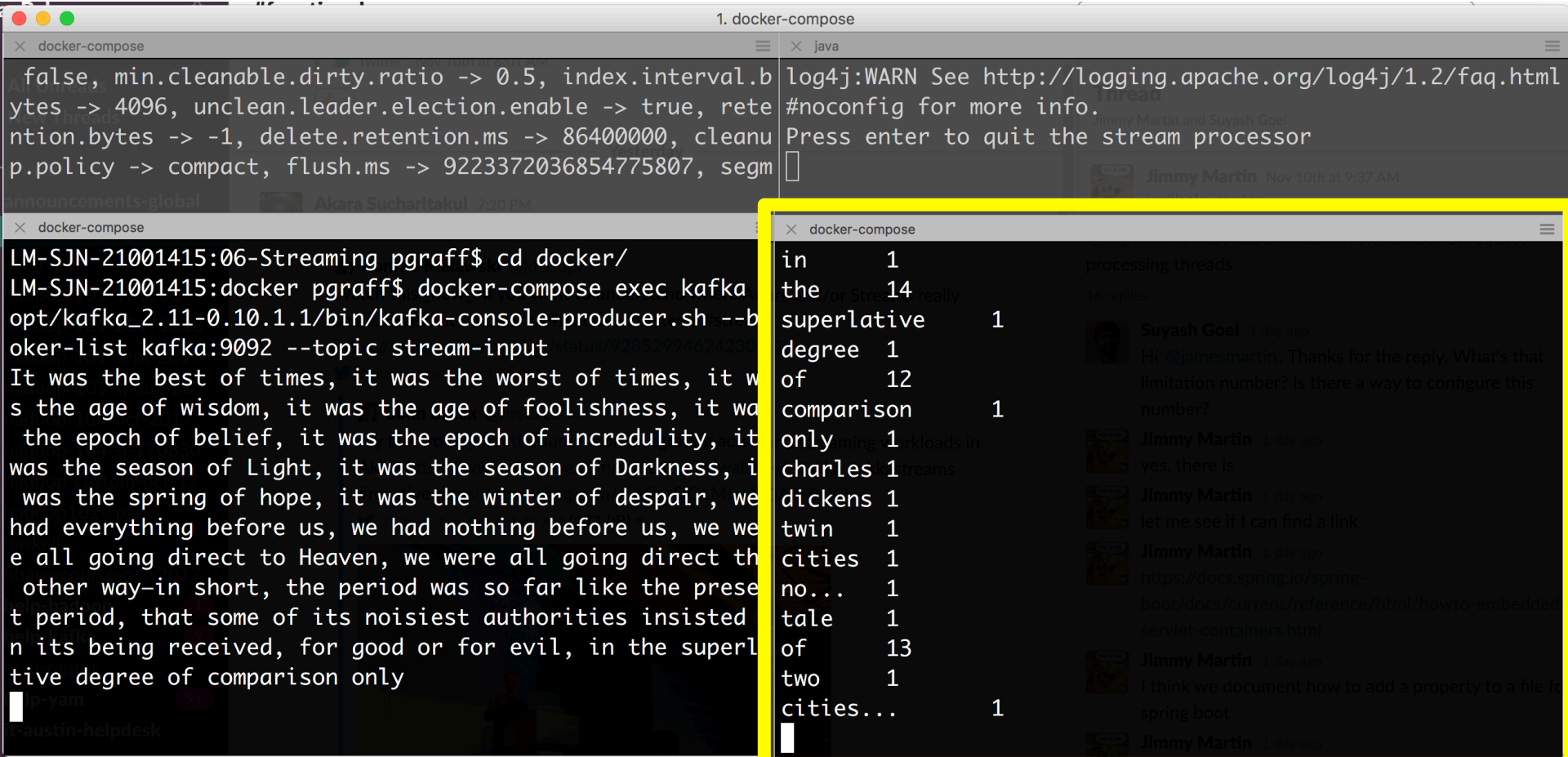
Where Are We?


1. java
docker-compose
for partition [__consumer_offsets,18] in /kafka/kafka-logs
-0e43a4963505 with properties {compression.type -> produce
r, message.format.version -> 0.10.1-IV2, file.delete.delay
.ms -> 60000, max.message.bytes -> 1000012, min.compaction
.lag.ms -> 0, message.timestamp.type -> CreateTime, min.in
sync.replicas -> 1, segment.jitter.ms -> 0, preallocate ->
false, min.cleanable.dirty.ratio -> 0.5, index.interval.b
ytes -> 4096, unclean.leader.election.enable -> true, rete
ntion.bytes -> -1, delete.retention.ms -> 86400000, cleanu
p.policy -> compact, flush.ms -> 9223372036854775807, segm
java
LM-SJN-21001415:wordcount-kafka-solution pgraff$ target/wor
dcounter
log4j:WARN No appenders could be found for logger (org.apac
he.kafka.streams.StreamsConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html
#noconfig for more info.
Press enter to quit the stream processor
docker containers
The event processor
docker-compose
LM-SJN-21001415:06-Streaming pgraff$ cd docker/
LM-SJN-21001415:docker pgraff$ docker-compose exec kafka /
opt/kafka_2.11-0.10.1.1/bin/kafka-console-producer.sh --br
oker-list kafka:9092 --topic stream-input
console-producer
producing to
stream-input
docker-compose
LM-SJN-21001415:docker pgraff$ docker-compose exec kafka /o
pt/kafka_2.11-0.10.1.1/bin/kafka-topics.sh --create --zooke
eper zookeeper:2181 --replication-factor 1 --partitions 1 -
-topic stream-output
Created topic "stream-output".
LM-SJN-21001415:docker pgraff$ docker-compose exec kafka /o
pt/kafka_2.11-0.10.1.1/bin/kafka-console-consumer.sh --boot
strap-server localhost:9092 --topic stream-output --from-be
ginning --property print.key=true
console-consumer
listening to
stream-output

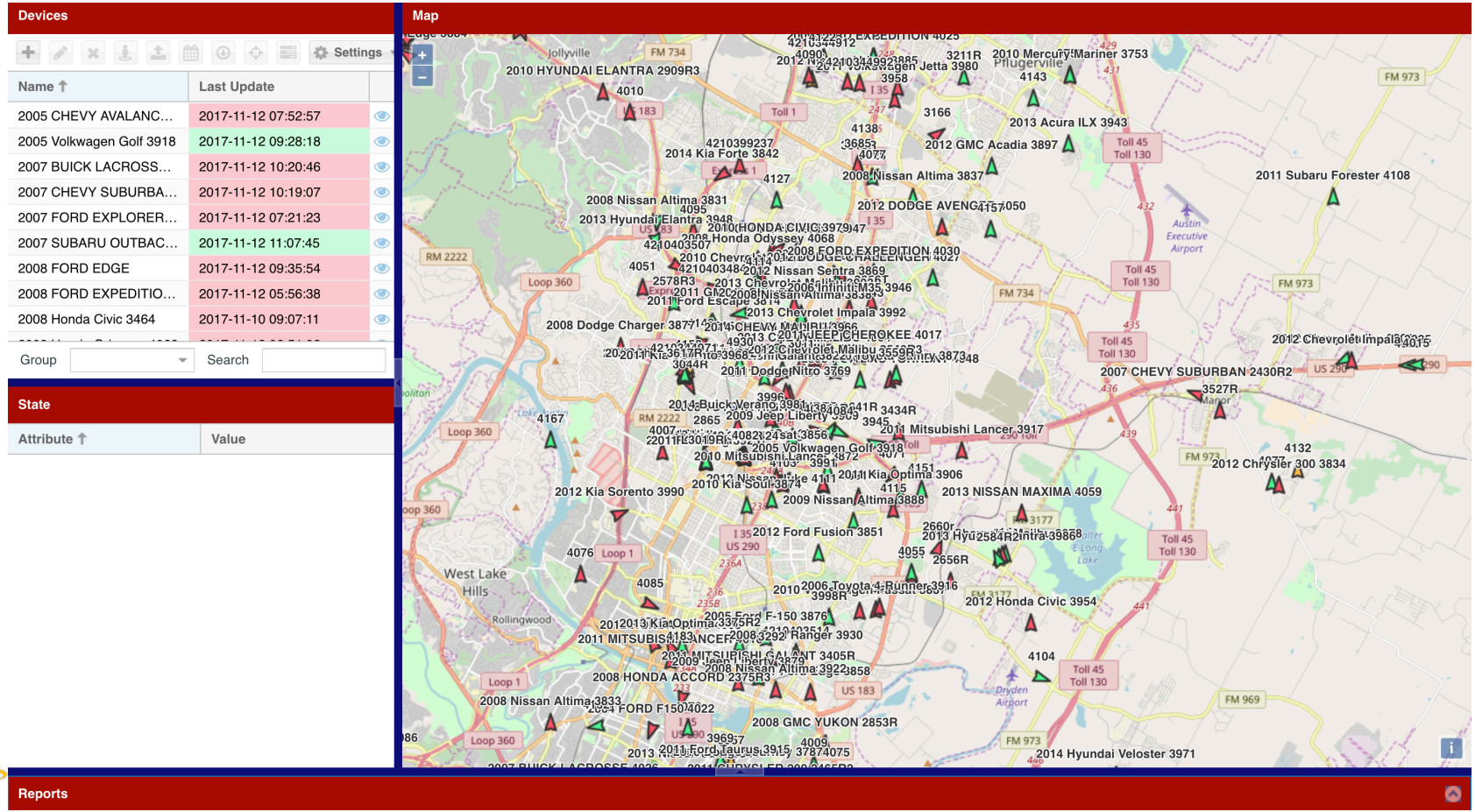
Step 7: Enter Some Text in the Producer



After some time...



Lab/Example: Another Problem



Devices

Settings

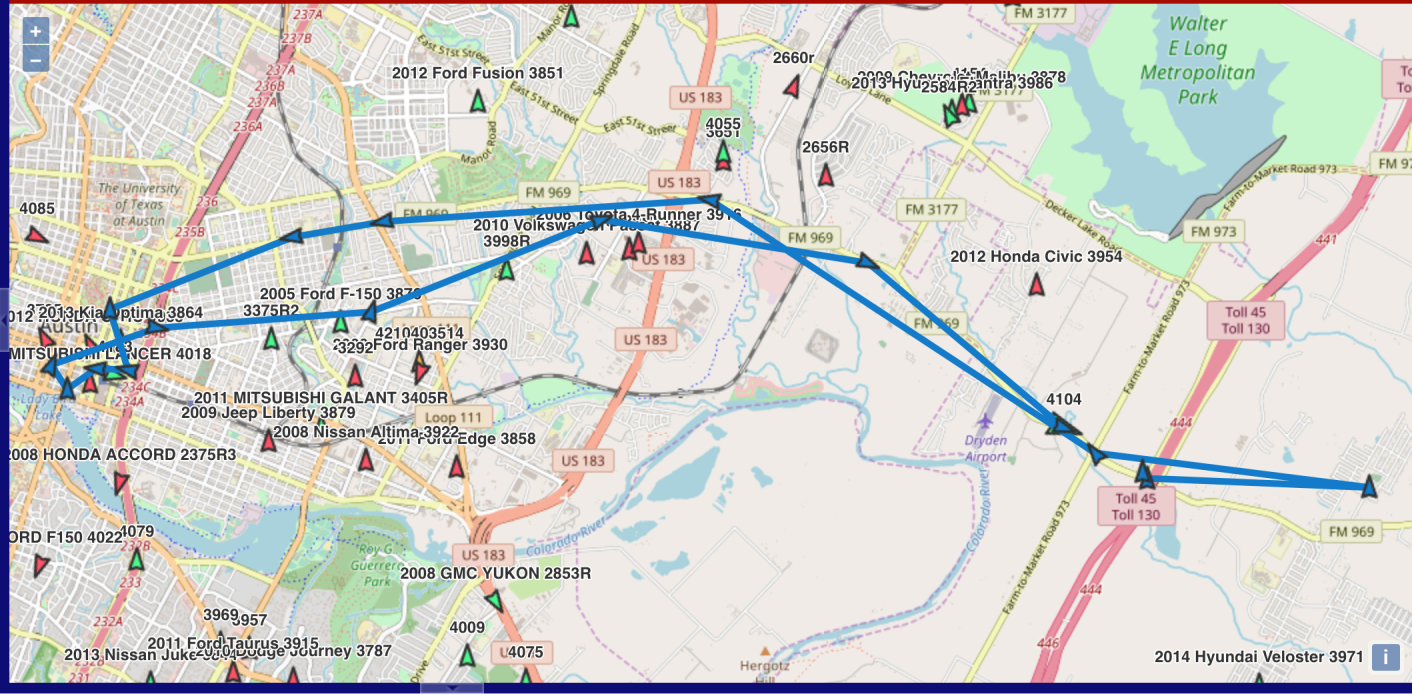
Name	Last Update
2005 CHEVY AVALANC…	2017-11-12 07:52:57
2005 Volkswagen Golf 3918	2017-11-12 09:28:18
2007 BUICK LACROSS…	2017-11-12 10:20:46
2007 CHEVY SUBURBA…	2017-11-12 10:19:07
2007 FORD EXPLORER…	2017-11-12 07:21:23

Group Search

State

Attribute	Value
Address	display
Battery	No
Color	BLACK
Course	E
Ignition	Yes
Ip	80.65.250.116
Latitude	30.259717
Longitude	-97.615567

Map



Device 4104 From 2017-11-12 00:00 To 2017-11-12 23:59 Show Clear

Event	Time	Latitude	Longitude	Speed	Address
.. moving	2017-11-12 01:22:00	30.259717	-97.615567	50.6 mph	display
.. moving	2017-11-12 01:17:00	30.278067	-97.640783	51.8 mph	display
.. moving	2017-11-12 01:12:00	30.282883	-97.675117	31.1 mph	display
.. moving	2017-11-12 01:07:00	30.272633	-97.704783	12.7 mph	display
.. moving	2017-11-12 01:02:00	30.270833	-97.732417	16.1 mph	display
.. moving	2017-11-12 00:57:00	30.266733	-97.745950	20.7 mph	display
.. moving	2017-11-12 00:52:00	30.264017	-97.743933	2.3 mph	display

Example Description

- We get a stream of vehicle locations
- Each location contains:
  - Vehicle ID
  - Data of observation
  - Speed
  - Direction
  - Latitude
  - Longitude

Problem and Plan

- We want to know where the vehicle is most likely to be parked
- Our plan
  1. Parse the vehicle data from an input stream
  2. Filter out the only those locates where the vehicle was at rest
  3. Convert the vehicle locate into keys that we can accumulate. The key would be based on
     - Vehicle ID
     - Geo location
  4. Group the locates together by key and count them
  5. Stream the frequent parking count to another stream

Input File: vehicle_1_to_1000.tsv

```
678  120 2016-08-16 16:31:21 30  162 -97.79678333333334  30.466516666666667
679  120 2016-08-16 16:31:31 5 159 -97.79646666666666  30.465716666666665
680  113 2016-08-16 16:31:37 5 159 -97.79645 30.465733333333333
681  120 2016-08-16 16:31:41 5 159 -97.79646666666666  30.465716666666665
682  120 2016-08-16 16:31:51 5 159 -97.79646666666666  30.465716666666665
683  120 2016-08-16 16:32:01 14  154 -97.79638333333334  30.465566666666668
684  120 2016-08-16 16:32:11 22  163 -97.79596666666667  30.464583333333334
685  120 2016-08-16 16:32:17 28.4  159 -97.795695  30.463865
686  120 2016-08-16 16:32:21 26  159 -97.79548333333334  30.463366666666666
687  107 2016-08-16 16:32:29 25  159 -97.79508333333334  30.4624
688  120 2016-08-16 16:32:31 24  157 -97.795 30.462216666666666
689  120 2016-08-16 16:32:41 7 160 -97.79466666666667  30.4615
690  120 2016-08-16 16:32:51 2 158 -97.79463333333334  30.461433333333332
691  113 2016-08-16 16:32:52 2.1 160 -97.79464166666666  30.461433333333332
692  120 2016-08-16 16:33:01 2 158 -97.79463333333334  30.461433333333332
693  111 2016-08-16 16:33:08 2 161 -97.79468333333334  30.46145
```

Message Pump: Simulator

- We've created a message pump application to simulate the vehicle input
- You can find this implementation in the directory:
  `kafka-lab/labs/06-Streaming/iot-kafka-solution/gps-pump`
- The implementation reads the TSV file and produces messages into a topic called:
  `gps-locations`
- In the real world, we would have to write some edge servers that receives the data from the vehicles and publish the topic into the topic

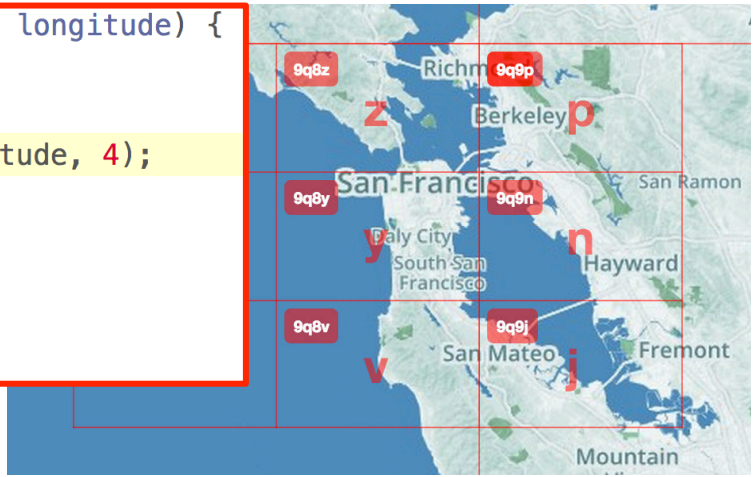
Event Processor

- We've created an event processor based on kafka-streaming (and another on Spark that we'll look at later)
- You can find this implementation in the directory: **`kafka-lab/labs/06-Streaming/iot-kafka-solution/processor`**
- This processor uses kafka-streams to read the events from the topic **`gps-locations`** and produces messages on the topic **`frequent-parking`**
  - The messages produced:
    - Key: Vehicle and location key based on GeoHash
    - Value: parking count

GeoHash

- We build a key based on the vehicle id and a GeoHash
- GeoHash allows us to focus on an area instead of absolute locates

```
 9    public LocationKey(String id, double latitude, double longitude) {
10      this.objectId = id;
11      try {
12        this.geoHash = GeoHash.encodeHash(longitude, latitude, 4);
13      }
14      catch (Exception e) {
15        System.out.println(e.getMessage());
16      }
17    }
```



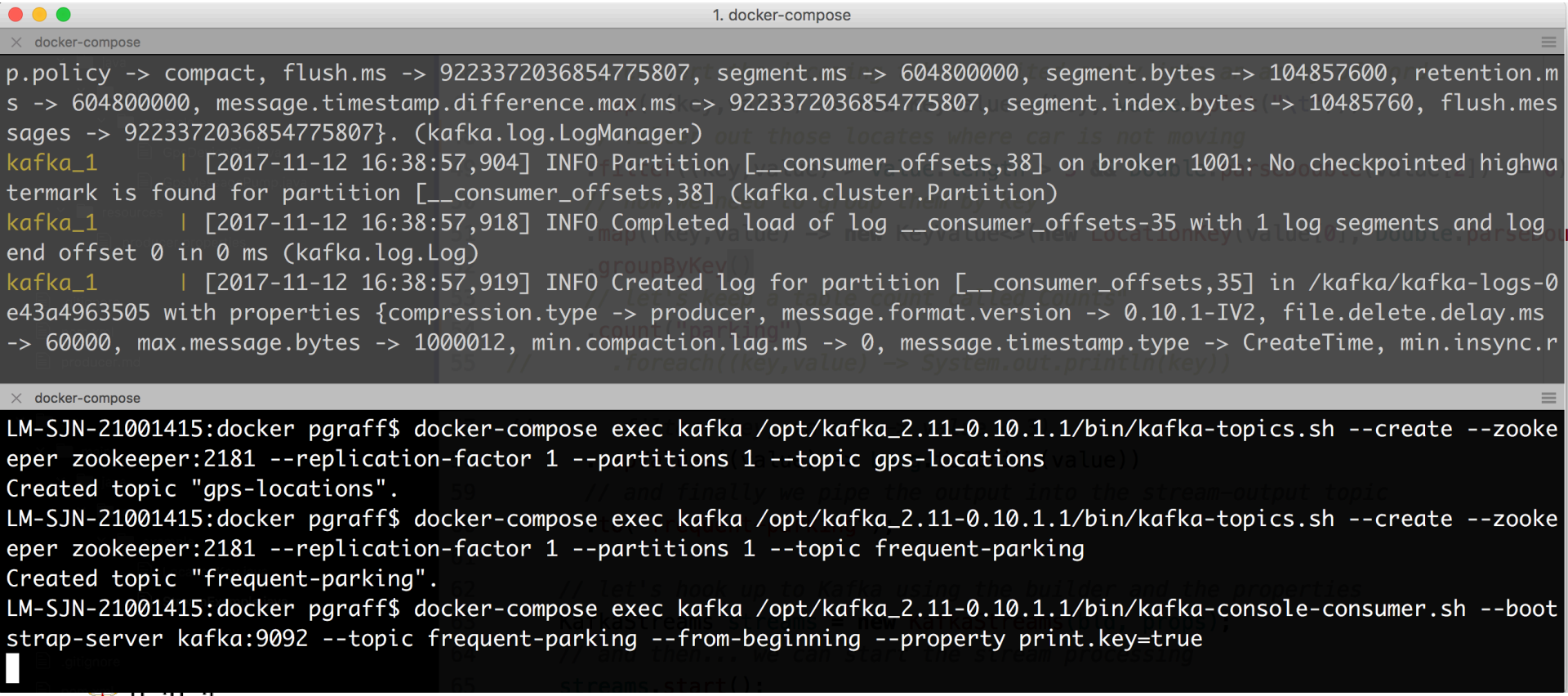
Kafka Stream Solution

```
source
  .map( (key,value) -> new KeyValue<>(key, value.split("\t")))
  .filter((key,value)-> value.length > 5 && Double.parseDouble(value[2]) == 0)
  .map((key,value) ->
      new KeyValue<>(
        new LocationKey(value[0],
          Double.parseDouble(value[4]),
          Double.parseDouble(value[5])).toString(),
          value[0]))
  .groupByKey()
  .count("parking")
  .mapValues((value) -> Long.toString(value))
  .to("frequent-parking");
```

Run Solution in Kafka

1. Modify the docker-compose file to use your (which you may already have done in the word-count exercise)
2. Create the topics:
   - gps-locations
   - frequent-parking
3. Run a kafka-consumer to listen to the frequent-parking topic (with the show key property)
4. Build and run the message pump
5. Build and run the processor
6. Wait to see the parking counts...

Creating the Topics and Start the Console Consumer



Build and Start the Message Pump

```
^CLM-SJN-21001415:gps-pump pgraff$ mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building gps-message-pump 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --
[WARNING]
```

sh

```
[INFO] --- maven-assembly-plugin:2.6:single (make-assembly) @ gps-message-pump ---
[INFO] Building jar: /Users/pgraff/kafka/kafka-lab/labs/06-Streaming/iot-kafka-solu
mp-1.0-SNAPSHOT-jar-with-dependencies.jar
[INFO]
[INFO] --- really-executable-jar-maven-plugin:1.1.0:really-executable-jar (default)
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2.559 s
[INFO] Finished at: 2017-11-12T11:51:04-06:00
[INFO] Final Memory: 26M/313M
[INFO] ------------------------------------------------------------------------
LM-SJN-21001415:gps-pump pgraff$
LM-SJN-21001415:gps-pump pgraff$ target/gps-message-pump ../vehicle_1_1000.tsv
```

Output

```
java
Sent: 120        2016-08-16 15:54:03     0       0       -97.83058333333334      30.446433333333335
Sent: 120        2016-08-16 15:54:12     0       0       -97.83058333333334      30.446433333333335
Sent: 120        2016-08-16 15:54:22     0       0       -97.83058333333334      30.446433333333335
Sent: 120        2016-08-16 15:54:32     0       0       -97.83058333333334      30.446433333333335
Sent: 120        2016-08-16 15:54:42     0       0       -97.83058333333334      30.446433333333335
Sent: 120        2016-08-16 15:54:52     0       0       -97.83058333333334      30.446433333333335
Sent: 120        2016-08-16 15:55:02     0       0       -97.83058333333334      30.446433333333335
Sent: 120        2016-08-16 15:55:12     0       0       -97.83058333333334      30.446433333333335
Sent: 113        2016-08-16 15:55:52     3.5     325     -97.83055833333333      30.446461666666668
Sent: 120        2016-08-16 15:55:56     0       0       -97.83058333333334      30.446433333333335
Sent: 107        2016-08-16 15:55:58     0       0       -97.83049166666666      30.446375
Sent: 120        2016-08-16 15:56:06     0       0       -97.83058333333334      30.446433333333335
Sent: 120        2016-08-16 15:56:16     0       0       -97.83058333333334      30.446433333333335
```

Build and Start the Processor

```
sh
LM-SJN-21001415:processor pgraff$ mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building iot-example 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
```

```
sh
[INFO]
[INFO] --- really-executable-jar-maven-plugin:1.1.0:really-executable-jar (default) @ iot-example ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4.037 s
[INFO] Finished at: 2017-11-12T11:55:59-06:00
[INFO] Final Memory: 50M/368M
[INFO] ------------------------------------------------------------------------
LM-SJN-21001415:processor pgraff$ target/parking-processor
```

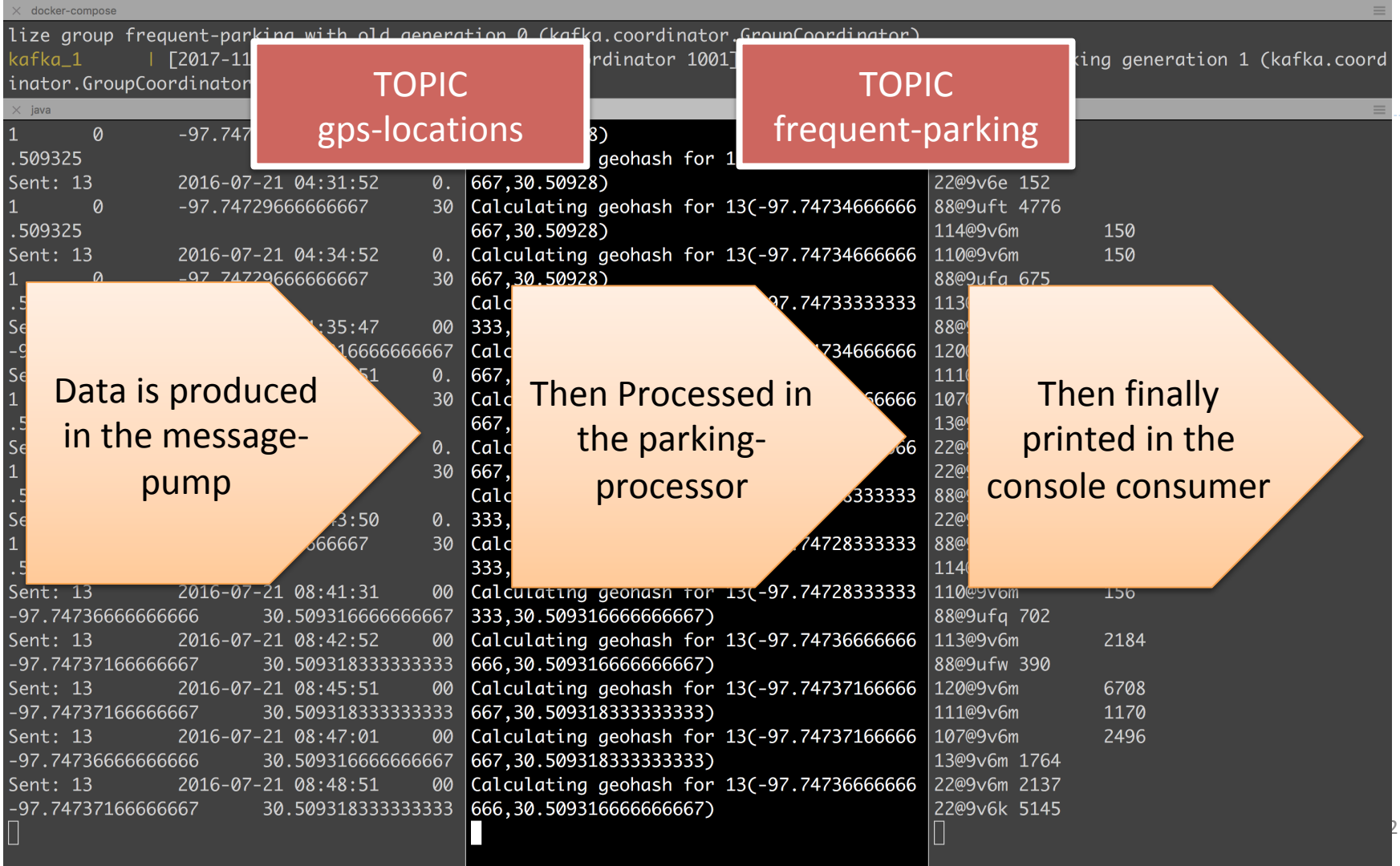
Output

java

```
Calculating geohash for 88(-97.40266666666666,27.765366666666665)
Calculating geohash for 88(-97.40266666666666,27.765366666666665)
Calculating geohash for 88(-97.40266666666666,27.76546666666667)
Calculating geohash for 88(-97.40266666666666,27.76546666666667)
Calculating geohash for 22(-97.89346666666667,30.369886666666666)
Calculating geohash for 22(-97.89346666666667,30.369883333333334)
Calculating geohash for 22(-97.9058,30.37845)
Calculating geohash for 22(-97.9058,30.378416666666666)
Calculating geohash for 88(-97.40265,27.765433333333334)
Calculating geohash for 88(-97.40266333333334,27.765446666666666)
Calculating geohash for 88(-97.40263333333333,27.76546666666667)
Calculating geohash for 88(-97.40264833333333,27.765475)
Calculating geohash for 88(-97.4027,27.7654)
Calculating geohash for 88(-97.402715,27.76541)
Calculating geohash for 88(-97.40266666666666,27.76545)
Calculating geohash for 88(-97.40267333333334,27.765455)
Calculating geohash for 88(-97.40267333333334,27.765455)
```

docker-compose

```
120@9v6m        5934
111@9v6m        1035
107@9v6m        2208
13@9v6m 1575
22@9v6m 1903
22@9v6k 4588
88@9v6m 2464
22@9v6e 146
88@9uft 4587
114@9v6m        144
110@9v6m        144
88@9ufq 648
113@9v6m        2016
88@9ufw 360
120@9v6m        6192
111@9v6m        1080
107@9v6m        2304
13@9v6m 1638
22@9v6m 1981
22@9v6k 4773
```


TOPIC
gps-locations
TOPIC
frequent-parking
Data is produced in the message-pump
Then Processed in the parking-processor
Then finally printed in the console consumer