A Quick Dive into Cloud Data Streaming Technology

Dennis Gannon
School of Informatics and Computing
Indiana University

(This paper adapted from the blog at www.esciencegroup.com)

Introduction

Cloud computing evolved from the massive data centers that were built to handle the "big data" challenges that confronted the designers of on-line services like search and e-mail. For the most part, data from these services accrued into large collections in the cloud where they could be analyzed by massively parallel, batch computing jobs. The types of knowledge derived from this analysis is used to improve the services that generated the data in the first place. For example, data analysis of cloud system log files can yield valuable information about how to improve performance of the cloud system. Analysis of user search terms can improve the search index. Analysis of vast collections of text can be used to create new machine learning based services such as natural language translation services.

While batch analysis of big collections is extremely important, it is often the case that the results of the analysis must be available as soon as the data is available. For example, analyzing data from instruments that control complex systems, such as the sensors onboard an autonomous vehicle or an energy power grid. In these instances, the data analysis is critical to driving the system. In some cases, the value of the results diminishes rapidly as it gets older. For example, trending topics in a twitter stream is not very interesting if it is no longer trending. In other cases, the volume of data that arrives each second is so large that it cannot be retained and real-time analysis or data reduction is the only way to handle it. This is true of some extremely large science experiments.

We refer to the activity of analyzing data coming from unbounded streams as data stream analytics. While many people think this is a brand new topic, there is a longer history that goes back to some basic research on complex event processing in the 1990s at places like Stanford, Caltech and Cambridge. These projects created some of the intellectual foundation for today's systems.

In the paragraphs that follow we will describe some of the recent approaches to stream analytics that have been developed by the open source community and the public cloud providers. As we shall see there are many factors that determine when a particular technology is appropriate for a particular problem. While it is tempting to think that one open source solutions can cover all the bases, this may not be the case. In fact there is an entire zoo of interesting solutions including Spark Streaming which has been derived from the Spark parallel data analysis system, Twitter's Storm system which has been redesigned by Twitter as Heron, Apache Flink from the German Stratosphere project, Googles Dataflow which is becoming Apache Beam which will run on top of Flink, Spark and Google's cloud. Other university projects include Borealis from Brandeis, Brown and MIT, Neptune and the Granules project at Colorado State. In addition to Google Cloud dataflow other commercial cloud providers have contributed to the available toolkit: Amazon Kinesis, Azure Streaming and IBM Stream Analytics are a few examples. In some cases, the analysis of instrument data streams needs to move closer to the source and tools are emerging to do "pre-analysis" to decide what data should go back to the cloud for deeper analysis. For example, the Apache Quark edge-analytics tools are designed to run in very small systems such as the

Raspberry Pi. A good survey of many of these stream processing technologies is by Kamburugamuve and Fox. They cover many issues not discussed here.

Basic Design Challenges of Streaming Systems

Before continuing it is useful to address several basic problems that confront the designers of these system. A major problem is the question of correctness and consistency. Here is the issue. Data in an unbounded stream is unbounded in time. But if you want to present results from the analytics, you can't wait until the end of time. So instead you present results at the end of a reasonable window of time. For example, a daily summary based on a complete checkpoint of events for that day. But what if you want results more frequently? Every second? The problem is that if the processing is distributed and the window of time is short you may not have a way to know about the global state of the system and some events may be missed or counted twice. In this case the reports may not be consistent. Strongly consistent event systems will guarantee that each event is processed once and only once. A weakly consistent system may give you approximate results that you can "back up" by a daily batch run on the daily checkpoint file. This gives you some ground-truth to fall back on if you suspect your on-line rapid analysis reporting is less reliable. Designs based on combining a streaming engine with a separate batch system is called the Lambda Architecture. The goal of many of the systems described below is to combine the batch computing capability with the streaming semantics so having a separate batch system is not necessary.

The other issue is the design of the semantics of time and windows. Many event sources provide a time stamp when an event is created and pushed into the stream. However, the time at which an events is processed will be later. So we have event time and processing time. To further complicate things events may be processed out of event-time order. This raises the question of how we reason about event time in windows defined by processing time.

There are at least four types of windows. Fixed Time windows divide the income stream into logical segments that correspond to a specified interval of processing time. The intervals do not overlap. Sliding windows allow for the windows to overlap. For example, windows of size 10 seconds that start every 5 seconds. Per-session windows divide the stream by sessions of activity related to some key in the data. For example, mouse clicks from a particular user may be bundled into a sequence of sessions of clicks nearby in time. Finally, there is the global window that can encapsulate an entire bounded stream. Associated with windows there must be a mechanism to trigger an analysis of the content of the window and publish the summary. Each of the systems below support some windowing mechanisms and we will discuss some of them and provide some concluding remarks at the end. A great discussion of this and many related issues is found in a pair of articles by Tyler Akidau.

Another design involves the way the system distributes the work over processors or containers in the cloud and the way parallelism is achieved. As we shall see the approaches to parallelism of the systems described here are very similar. This paper will not discuss performance or scalability issues. That is another topic we will return to later.

Finally, we note that operations on streams often resemble SQL-like relational operators. However, there are difficulties with this comparison. How do you do a join operation on two streams that are unbounded? The natural solution involves dividing streams by windows in time and doing the join over each window.

Vijayakumar and Plale have looked at this topic extensively. The CEDR system from MSR illustrated how SQL-like temporal queries can have a well-defined semantics.

Cloud Providers and the Open Source Streaming Tools.

One way to distinguish the streaming engines is look at the approach to the programming model. In one camp is an approach based on batch processing as derived from Hadoop or Spark, and the other is based on the pipelined execution of a directed acyclic graph.

Spark Streaming

Spark streaming is a good example that illustrates how one can adapt a batch style processing analytics tool to a streaming case. The core idea is very simple. You break the stream into a bunch of little batches.

To illustrate this and a few of the other technologies discussed here we will frame the discussion in terms of a hypothetical science application. Assume we have a large set of environmental sensor distributed over some area. Each sensor is connected by WiFi to the internet and each sends a sequence of messages to a cloud address for analysis. The sensors may be weather, sound, co2, light, motion, vibration or image capture. The size of the messages may only be a few bytes (time stamp + geo-location + temperature) or a few megabytes of sound or images. The goal of the project may be environmental restoration where you are interested in the health and development of the flora and fauna in some devastated forest. Or it may be something like the NSF ocean observatories project which has a large number of wired as well as untethered instruments along the U.S. coastal waters.

Spark streaming works by taking the input from a stream data source aggregator such as

1. A high throughput publish-subscribe system like RabbitMQ or a more highly scalable system like Apache Kafka or
2. The Microsoft Azure Event Hub (which we described in another post) or
3. Amazon Kinesis.

Kinesis is a robust data aggregator in that it can take from many sources at high rates of speed and it will retain the stream records for up to seven days. A Kinesis producer is a source of a stream of data records. Each Producer uses a partition key, such as “co2 sensor” that is attached to each data record as it is sent to Kinesis. Internally Kinesis partitions data into “shards” and each shard can handle up to 2 MB/sec or 1000 records per second of input data. Kinesis uses the partition key to map your data record to a shard. The total capacity of your stream is the sum of the capacity of the shards that it contains.

A Kinesis client is the program that pulls the data records from the Kinesis shards and processes it. You can roll your own client or you can use spark streaming or one of the other systems described here to do the processing. Spark streaming is just a version of Spark that processes data in batches where each batch is defined by a time interval. The Spark name for a stream is a DStream which is a sequence or Spark RDDs (Resilient Distributed Dataset). We covered Spark in a previous article here. Spark Streaming provides a nice adaptor which will automatically read the data from Kinesis shards and repackage them into DStreams so that they can be consumed by the Spark Engine as shown in Figure 1.

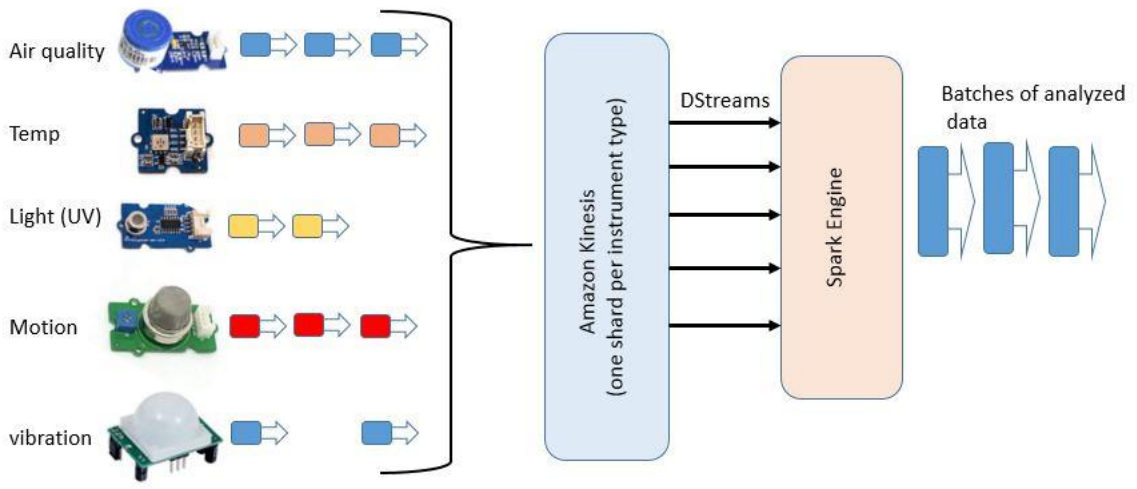

Figure 1. Environmental sensor analysis stream example.

Each RDD in the DStream is represents the data in a window of time from the shard associated with the instrument stream. This RDD is processed in parallel by the spark engine. Another level of parallelism is exploited by the fact that we have DStreams associated with each shard and we may have many of them. There is one processing thread for each shard. This is nicely illustrated in Figure 2 from the Spark Streaming Guide.

DStreams can be transformed into new DStreams using the Spark Streaming library. For example there are the *map*() and *filter*() functions that allows us to apply an analysis or filter on a DStream to produce a new one. DStreams can be merged together by the *union*() operator or, if there is a common key, such as a timestamp, one can apply a *join*() operator to create a new DStream with events with the same key tied together. Because each RDD in the DStream is process completely by the Spark engine, the results are strongly consistent. There is a very good technical paper from the Berkeley team that created spark streaming and it is well worth a read.

To illustrate spark streaming let's assume that every second our sensors from figure 1 each transfer a byte array that encodes a json string representing its output every second. Suppose we are interested in receiving a report of the average temperature for each 10 second window at each location where we have a temperature sensor. We can write a Python Spark Streaming program to do this as follows. First we need to create a streaming context and Kinesis connector to grab the stream of instrument data.

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.streaming.kinesis import KinesisUtils, InitialPositionInStream

sc = SparkContext("....", "sensortest")
ssc = StreamingContext(sc, 10)

ks = KinesisUtils.createStream(
     sc, [Kinesis app name], [Kinesis stream name], [endpoint URL],
     [region name], [initial position], [checkpoint interval],[StorageLevel])
```

Ks should now be a DStream where each RDD element is the set of events collected by Kinesis in the last 10 seconds. (Note: I have not yet actually tried this, so some details may be wrong. This example is adapted from a Kafka version from Jon Haddad and the Kenisis integration guide)

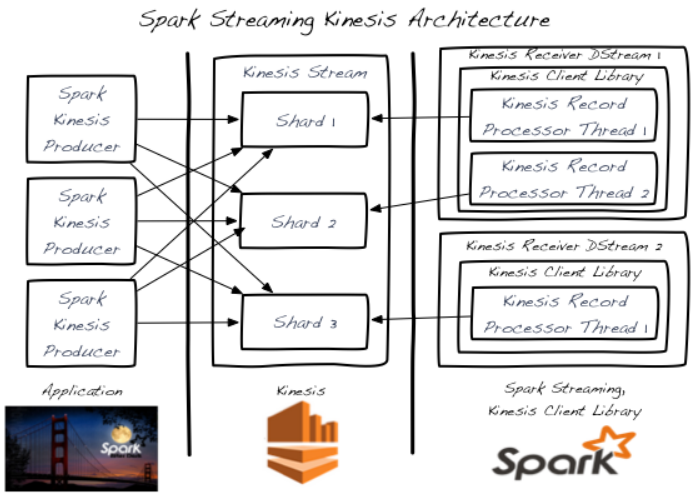

Figure 2. Spark Streaming with Kinesis (image from Spark streaming kinesis integration guide)

Next we will need to convert byte array for each sensor record to a json Python dictionary. From there we will filter out all but the temperature sensors, then using a simple map-reduce compute the average temperature for each sensor (which we identify by its location). To do this we can use the *reduceByKey*() method which will give us a sum and count for each sensor. We can then map that into a new DStream taking the form of a dictionary of sensor locations and average temperature for that interval as follows.

```
temps = ks.filter(lambda x: x["sensortype"] == "tempsensor")    \
          .map(lambda x: (x["location"], (x["value"], 1))       \
          .reduceByKey(lambda (x1,y1),(x2,y2): (x1+x2,y1+y2))   \
          .map(lambda z: {"location": z[0], "average temp": z[1][0]/z[1][1]})
```

We may now dump our result DStream temps to storage at the end of the processing of this RDD. Alternatively, we can join this DStream with a static DStream to compute a running average temperature.

Storm and Heron: Streaming with a DAG dataflow style.

There are several significant systems based on executing a directed graph of tasks in a "dataflow" style. We will give a brief overview of three of these. One of the earliest was Storm which was created by Nathan Marz and released as open source by Twitter in late 2011. Storm was written in a dialect of Lisp called Clojure that works on the Java VM. In 2015 Twitter rewrote Storm and it is has deployed it under the name Heron which is being released as an Apache project. The Heron architecture was described in an article in the ACM SIGMOD 2015 conference. Heron implements the same programming model and API as Storm, so we will discuss Storm first and then say a few words about the Heron design.

Storm (and Heron) run "topologies" which are directed acyclic graphs whose nodes are Spouts (data sources) and Bolts (data transformation and processing). Actually Storm has two programming models. One of these we can call classic and the other is called Trident which is built on top of the classic model. In both cases Storm (and Heron) topologies are directed acyclic graphs as shown in Figure 3.

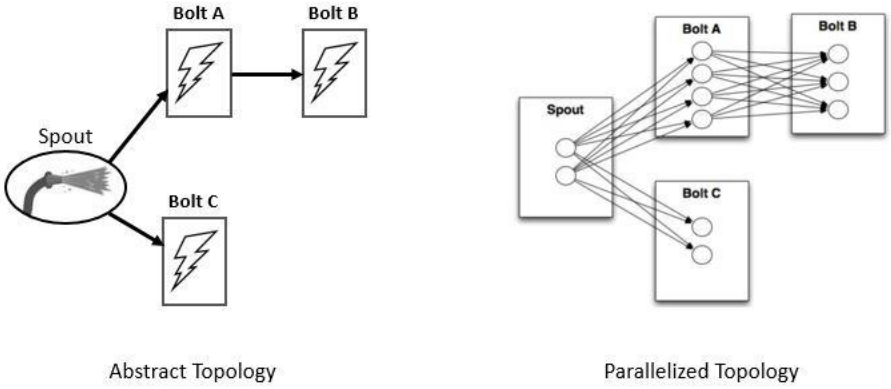

Figure 3. Storm/Heron topology. On the left is the abstract topology as defined by the program and on the right is the unrolled parallel topology for runtime.

The programming model is based on extending the basic spout and bolt classes and then using a topology builder to tie it all together. A basic template for a Bolt is shown below. There are three required methods. *The prepare*() method is a special constructor that is called when the actual instance is deployed on the remote JVM. It is supplied with context about the configuration and topology as well as a special object called the *OuputCollector* which is used to connect the Bolts output to the output stream defined by the topology. The *prepare*() method is also where you instantiate your own data structures.

The basic data model for Storm/Heron is a stream of *Tuples*. A tuple is just that: a tuple of items where each item need only be serializable. Some of the fields in a tuple have names that are used for communicating a bit of semantics between bolts. The method *declareOutputFields*() is used to declare the name of the fields in a stream. More on this point later. The heart of the bolt is the method *execute*(). This is invoked for each new tuple that is sent to the bolt and it contains the computational core of the bolt. It is also where results from the Bolts process is sent to its output streams.

The main programming API for Storm is Java, so we will touch briefly on that here. There are several base classes and styles of bolts, but this is the basic template. One of the specialized Bolt classes is for sliding and tumbling windows. Spouts are very similar classes, but the most interesting ones are the Spouts that connect to event providers like Kafka or EventHub.

```
public class MyBolt extends BaseRichBolt{
	private OutputCollector collector;
	public void prepare(Map config, TopologyContext context,
			OutputCollector collector) {
		this.collector = collector;
	}
	public void execute(Tuple tuple) {
		/*
		*execute is called when a new tuple has been delivered.
		*do your real work here.  for example,
		*create a list of words from the tuple and then emit them
		*to the default output stream.
		*/
		for(String word : words){
			this.collector.emit(new Values(word));
		}
	}
	public void declareOutputFields(OutputFieldsDeclarer declarer) {
	    /*
		* the declarer is how we declare out output fields in the default
		* output stream.  you can have more than one output stream
		* using declarestream. the emit() in execute needs to identify
		* the stream for each output value.
		*/
		declarer.declare(new Fields("word"));
	}
}
```

The topology builder class is to build the abstract topology and provide instructions for how the parallelism should be deployed. The key methods of the build are *setBolt*() and *setSpout*(). These each take three arguments: the name of the spout or bolt instance, an instance of your spout or bolt class and an integer that tells the topology how many tasks will be assigned to execute this instance. A task is a single thread that is assigned to a spout or bolt instance. This is the parallelism number. The code below shows how to create the topology of Figure 3.

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("Spout", new MySpout(), 2);
builder.setBolt("BoltA", new MyBoltA(), 4).shuffleGrouping("spout");
builder.setBolt("BoltB", new MyBoltB(), 3).fieldsGrouping("BoltA", new Fields("word"));
builder.setBolt("BoltC", new MyBoltC(), 2).shuffelGrouping("spout")

Config config = new Config();
LocalCluster cluster = new LocalCluster();
cluster.submitTopology("mytopology", config, builder.createTopology());
```

As you can see, there are 2 tasks for the spout, 4 for bolt A, 3 for bolt B and 2 for bolt C. Note that the 2 tasks for the spout are sent to 4 for Bolt B. How do we partition the 2 output streams over the 4 tasks? To do this we use a stream grouping function. In this case we have used Shuffle grouping which randomly distributed them. In the second case we map the 4 outputs streams from Bolt A to the 3 tasks of bolt B using a field grouping based on a field name. This makes sure that all tuples with the same field name are mapped to the same task.

As mentioned above the Twitter team has redesigned storm as Heron. The way a topology is executed is that a set of container instances are deployed to manage it as shown in Figure 4.

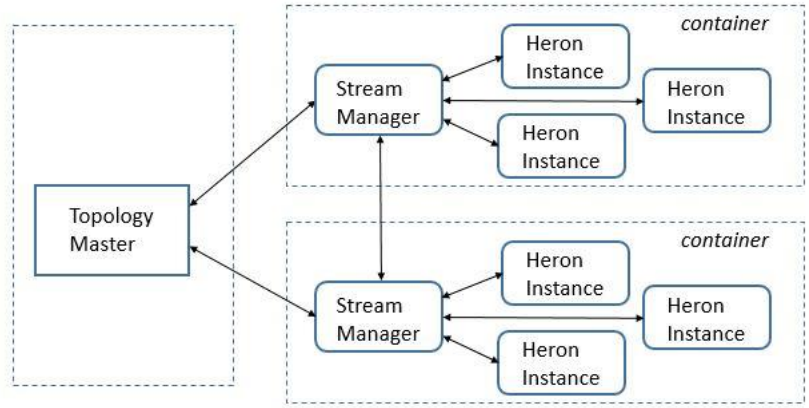

Figure 4. Heron architecture detail.

The topology master coordinates the execution of the topology on a set of other containers that each contain a stream manager and heron instance processes which execute the tasks for the bolts and spouts. The communication between the bolts and spouts are mediated by the stream manager and all the stream managers are connected together in an overlay network. (The topology master makes sure they are all in communication.) Heron provides great performance improvements over Storm. One improvement of the architecture is better flow control of data from spouts when the bolts are falling behind. Please look at the full paper for more detail. Some of the best Storm tutorial material comes from Michael Noll's blog (here is a good example).

Trident

As mentioned Storm has another programming model that is implemented on top of the basic spout bolt library. This is called Trident. The classic Storm programming model is based on the topology instance. You construct the flow graph by adding spouts and bolts. It is building a graph by adding the nodes. Trident is somewhat of a dual concept: it is about the edges. The central figure in Trident is the stream. The first thing to note is that trident processes all events in a stream in batches and Trident works very hard to make sure that each tuple is processed once and only once. However, in real life failure happens and retries may be required. Since tuples originate from spouts defining the retry semantics must be closely tied to the spout. Trident has several configurations for spout depending on the semantics required. Some are transactional, meaning every batch has a transaction identifier (*txid*) and a tuple does not appear in any other batch. Using the *txid* we can make sure we never process a tuple more than once. If the tuple caused the processing of the batch to fail, we can re-issue the entire batch. Regular Storm spouts are non-transactional. Another type of spout is "opaque transactional" the third category which guarantees that each tuple is processed exactly once but, if not, it may appear in another batch.

Let's begin by declaring a trivial artificial (non-transactional) spout that has a single word in each tuple called "name". I want the batch size to be 50 tuples. The code will look something like this.

```
TridentTopology topology = new TridentTopology();
FixedBatchSpout spout = new FixedBatchSpout(new Fields("name"), 50, the word list )
Stream str1 = topology.newStream("spout", spout)
```

Now that we have a stream we can start making transformations to it. For example, we can expand the

tuple so each tuple contains the word and also the number of characters in the word. We can do this by creating a function object that takes the string from the tuple and emits its length.

```
public static class Getlength extends BaseFunction {
  @Override
  public void execute(TridentTuple tuple, TridentCollector collector) {
    collector.emit(new Values(tuple.getString(0).length()));
  }
}
```

We apply this function to the stream to create a new stream.

```
Stream str2 = str1.each(new Fields("name"), new Getlength, new Fields("length"));
```

Notice that the function only emitted the length. The *each()* function has the strange property that it appends new field to the end of the tuple, so now each tuple has labels [“name”, “length”]. Next suppose we only want names from a particular list *mynames* and we want to drop the others. We will write a filter function to do that and then create a new filtered stream

```
public static class NameFilter extends BaseFilter {
  List<String> nameslist

  public NameFilter(List<String> names) {
    this.namelist = names;
  }
  @Override
  public boolean isKeep(TridentTuple tuple) {
    return namelist.contains(tuple.getString(0));
  }
}

Stream str3 = str2.each(new Fields("name","length"), new NameFilter(mynames));
```

Now let’s partition the stream by the name field and compute the counts of each. The result is of type TridentState.

```
TridentState counts =
   str3.groupBy(new Fields("name"))
       .persistentAggregate(new MemcachedState.opaque(serverLocations),
                            new Count(), new Fields("count"))
```

The details about how the data is sent to the databases behind the memcashe are not important here by the idea is we can now keep track of the aggregate state of the stream.

The final thing we should look at is how parallelism is expressed. This is actually fairly simple annotations to the stream. Putting all the steps above into one expression we can show how this is done.

```
TridentState counts =
topology.newStream("spout", spout)
        .parallelismHint(2)
        .shuffle()
        .each(new Fields("name"), new Getlength, new Fields("length"))
        .parallelismHint(5)
        .each(new Fields("name","length"), new NameFilter(mynames))
        .groupBy(new Fields("name"))
        .persistentAggregate(new MemcachedState.opaque(serverLocations),
                             new Count(), new Fields("count"));
```

This version creates two instances of the spout and five instances of the Getlength() function and uses the random shuffle to distribute the tuple batches to the instances. There is much more to classic Storm and Trident and there are several good books on the subject.

Google's Dataflow and Apache Beam

The most recent entry to the zoo of solutions we will discuss is Apache Beam (now in Apache's incubation phase.) Beam is the open source release of the Google Cloud Dataflow system. Much of what is said below is a summary of their document. An important motivation for Beam (from now on I will use that name because it is shorter than writing "Google Cloud Dataflow") is to treat the batch and streaming cases in a completely uniform way. The important concepts are

1. Pipelines – which encapsulates the computation in the model.
2. PCollections – the data as it moves through a Pipeline.
3. Transforms – the computational transformations that operate on PCollections and produce PCollections
4. Sources and Sinks.

PCollections

The idea is that a PCollection can be either a very large but fixed size set of element or a potentially unbounded stream. The elements in any PCollection are all of the same type, but that type maybe any serializable Java type. The creator of a PCollection often appends a timestamp to each element at creation time. This is particularly true of unbounded collections. One very important type of PCollection that is used often is the Key-Value PCollection *KV<K, V>* where *K* and *V* are the Key and Value types. Another important thing to understand about PCollections is that they are immutable. You can't change them but you can use transforms to translate them into new PCollections.

Without going into the details of how you initialize a pipeline, here is how we can create a PCollection of type *PCollection<String>* of strings from a file.

```
Pipeline p = Pipeline.create(options);
PCollection<String> pc =
        p.apply(TextIO.Read.from("/home/me/mybigtextfile.txt"))
```

We have used the pipeline operator *apply()* which allows us to invoke the special transform TextIO to read the file. There are other pipeline operators, but we will not discuss many of them. Now, in a manner

similar to the way Trident uses the *each*() operator to create new Trident streams, we will create a sequence of PCollections using the *apply*() method of the PCollection class.

There are five basic transform types in the library. Most takes a built-in or user defined function object as an argument and applies the function object to each element of the PCollection to create a new PCollection.

1. *Pardo* - apply the function argument to each element of the of the input PCollection. This is done in parallel by workers tasks that are allocated to this activity. This is basic embarrassingly parallel map parallelism
2. *GroupByKey* – apply this to a *KV<K,V>* type of PCollection with group all the elements with the same key into the a single list, so the resulting PCollection is of type *KV<K, Iterable<V>>*. In other words, this is the shuffle phase of a map-reduce.
3. *Combine* – apply an operation that reduces a PCollection to a PCollection with a single element. If the PCollection is windowed the result is a Pcollection with the combined result for each window. Another type of combining is for key-grouped PCollections.
4. *Flatten* – combine PCollections of the same type into a single PCollection.
5. *Windowing and Triggers* – These are not transformations in the usual sense, but defining mechanisms for the window operations.

To illustrate some of these features let's redo the environmental sensor example again but we will compute the average temperature for each location using a sliding window. For the sake of illustration, we will use an imaginary pub-sub system to get the events from the instrument steam and let's suppose the events are delivered to our system in the form of a Java object from the class *InstEvnt*. That would be declared as follows.

```
@DefaultCoder(AvroCoder.class)
static class InstEvent{
       @Nullable String instType;
       @Nullable String location;
       @Nullable Double reading;
       public InstEvent( ....)
       public String getInstType(){ ...}
       public String getLocation(){ ...}
       public String getReading(){ ...}
}
```

This class definition illustrates how a custom serializable type looks like in Beam. We can now create our stream from our fictitious pub-sub system with this line.

```
PCollection<InstEvnt> input =
      pipeline.apply(PubsubIO.Read
                              .timestampLabel(PUBSUB_TIMESTAMP_LABEL_KEY)
                              .subscription(options.getPubsubSubscription()));
```

We next must filter out all but the "tempsensor" events. While we are at it, let's convert the stream so that the output is a stream of key-value pairs corresponding to (location, reading). To do that we need a special function to feed to the ParDo operator.

```
static class FilterAndConvert extends DoFn<InstEvent, KV<String, Double>> {
    @Override
    public void processElement(ProcessContext c) {
         InstEvent ev = c.element();
         if (ev.getInstType() == "tempsensor")
              c.output(KV<String, Double>.of(ev.getLocation(), ev.getReading));
    }
}
```

Now we can apply the Filter and Convert operator to our input stream. Let us also create a sliding window of events of duration five minutes that is created every two minutes. We note that the window is measured in terms of the timestamps on the events and not on the processing time.

```
PCCollection<KV<String, Float>> reslt = input
       .apply(Pardo.of(new FilterAndConvert())
       .apply(Window.<KV<String, Double>> into(SlidingWindows.of(
                            Duration.standardMinutes(5))
                            .every(Duration.standardMinutes(2))))
```

Our stream *reslt* is now a *KV<String,Double>* type and we can apply a *GroupByKey* and *Combine* operation to reduce this to a *KV<String,Double>* where each location key maps to the average temperature. To make life easy Beam has a number of variations of this simple map-reduce operation and one exists that is perfect for this case: *Mean.perKey*() which combines both steps in one transformation.

```
PCollection<KV<String, Double>> avetemps
       = reslt.apply(Mean.<String, Double>perKey());
```

Finally we can now take the set of average temperatures for each window and send them to an output file.

```
PCollection<String> outstrings = avetemps
       .apply(Pardo.of(new KVToString())
       .apply(TextIO.Write.named("WritingToText")
              .to("/my/path/to/temps")
              .withSuffix(".txt"));
```

The function class *KVToString*() is one we define in a manner similar to the *FilterAndConvert* class above. There are two things to notice in what happened above. First, we have used an implicit trigger that generates the means and output at the end of the window. Second, note that because the windows overlap, events will end up in more than one window.

Beam has several other types of triggers. For example, you can have a data driven trigger looks at the data as it is coming and fires when some condition you have set is met. The other type is based on a concept introduce by Google Dataflow called the watermark. The idea of the watermark is based on event time. It is used to emit results when the system estimates that it has seen all the data in a given window. There are actually several very sophisticated ways to define triggers based on different ways to specify the watermark. We won't go into them here and we refer you to the Google Dataflow documents.

Apache Flink

Flink is now one of the "runners" for Beam because it is possible to implement the Beam semantics on top of Flink. Many of the same core concepts exist in Flink and Beam. As with the other systems, Flink takes input streams from one or more sources, which are connected by a directed graph to a set of sinks.

Like the others, the system is based on a Java virtual machine and the API is rendered in Java and Scala. There is also an (incomplete) Python API where there is also a similarity to Spark Streaming. To illustrate this, we can compare the Flink implementation of our instrument filter for figure 1 to the Spark Streaming example above.

The Flink Kinesis Producer is still a "work in progress", so this code was tested by reading a stream from a CSV file. The Flink data types do not include the Python dictionary/Json types so we use here a simple tuple format. Each line of the input stream looks like

```
instrument-type string, location string, the word "value", floating point number
```

For example,

```
tempsensor, pine street and second, value, 72.3
```

After reading from the file (or Kinesis shard) the records in the stream *data* are now 4-tuples of type (STRING, STRING, STRING, FLOAT). The core of the Flink version of the temperature sensor averager is shown below.

```
class MeanReducer(ReduceFunction):
    def reduce(self, x, y):
        return (x[0], x[1], x[2], x[3] + y[3], x[4] + y[4])

env = get_environment()
data = env.add_source(FlinkKinesisProducer( … ) … )

resuts = data \
    .filter(lambda x: x[0]=='tempsensor') \
    .map(lambda x: (x[0], x[1], x[2], x[3], 1.0)) \
    .group_by(1) \
    .reduce(MeanReducer()) \
    .map(lambda x: 'location: '+x[1]+' average temp %f' % (x[3]/x[4]))
```

The filter operation is identical to the Spark Streaming case. After filtering the data we turn each record into a 5-tuple by appending 1.0 to the end of the 4-tuple. The *group_by*(1) and reduce using the *MeanReducer* function. The *group_by*(1) is a signal to shuffle these so that they are keyed by field in position 1 which corresponds to the *location string* and then we apply the reduction to each of the grouped tuple sets. This operation is the same as the reduceByKey function in the Spark Streaming example. The final map converts each element to a string that gives the average temperature for each location.

This example does not illustrate is Flink's windowing operators, which are very similar to Beam's, nor does it illustrate the underlying execution architecture. In a manner similar to the other systems described

here, Flink parallelizes the stream and tasks during execution. For example, our temperature sensor example has a logical view as tasks which may be executed in parallel as shown in Figure 5.

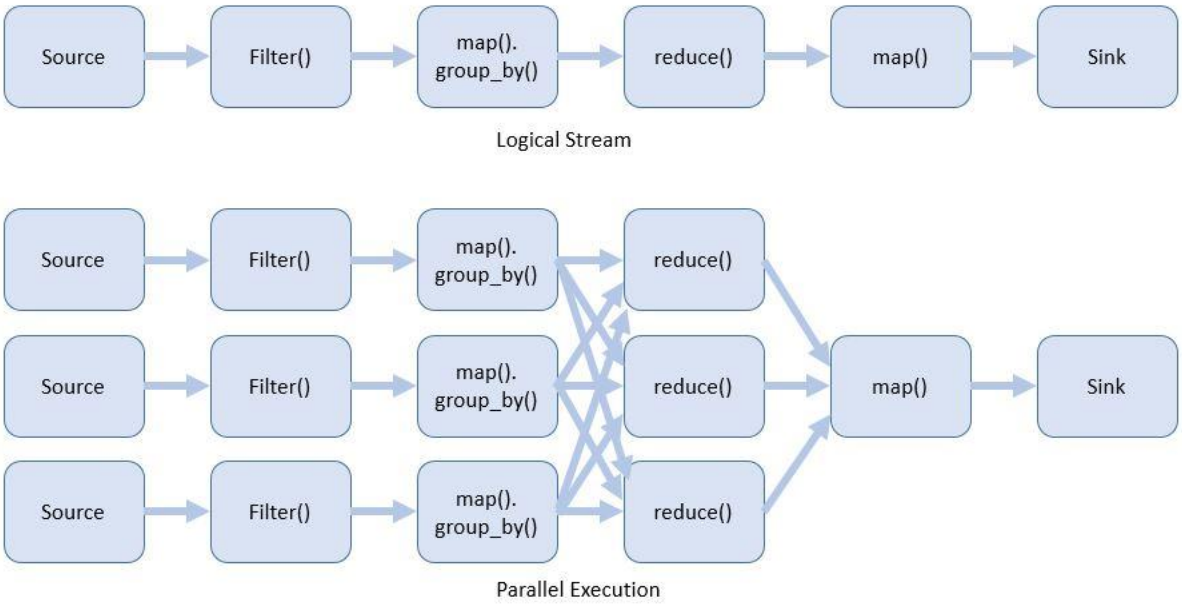

Figure 5. Flink logical task view and parallel execution view.

The Flink distributed execution engine is based on a standard master worker model. The Flink source program is compiled into an execution data flow graph and sent to a job manager node by a client system. The job manager executes the stream and transformations on remote Java VMs which run a task manager. The task manager partitions its available resources into task slots where the individual tasks defined by the graph execution nodes are assigned. The job manager and task managers manage the data communication streams between the graph nodes. This is all very nicely illustrated by a figure from the Apache Flink documentation. This documentation also describes the Flink windowing and other details of the implementation and programming model.

Summary and Conclusions

We have looked at four different systems, Spark Streaming, Storm/Heron, Google Dataflow/Beam and Flink. Each of these has been used in critical production deployments and proven successful for their intended applications. While we have only illustrated each with a trivial example we have seen that they all share some of the same concepts and create pipelines in very similar ways. One obvious difference is in the way Storm/Heron explicitly constructs graphs from nodes and edges and the others use a very functional style of pipeline composition. (Storm does have the Trident layer that allows a functional pipeline composition but it is not clear if this will be supported in the Heron version.)

Conceptually the greatest difference arises when comparing Spark Streaming to the others and, in particular, Beam. Akidau and Perry make a very compelling argument for the superiority of the Beam model in comparison to Spark Streaming. They make a number of important points. One obvious one is that Spark is a batch system for which a streaming mode has been attached and Beam was designed from the ground up to be streaming with obvious batch capabilities. The implication is that the windowing for Spark is based on the RDD in the DStream and this is clearly not as flexible as Beam windows. A more

significant point revolves around Beam's recognition that event time and processing time are not the same. Where this becomes critical is in dealing with out of order events, which are clearly possible in widely distributed situations. Beam's introduction of event-time windows, triggers and watermarks are a major contribution and clarifies a number of important correctness issues when events are out of order while still allowing you to get approximate results in a timely manner.

In terms of performance of these systems, we will leave it to another time to address this issue. In fact, it would be a very interesting exercise to create a set of meaningful benchmarks that each system can be measured against. It would be a non-trivial exercise to design the experiments, but well worth the effort.