Amazon Kinesis and Apache Storm

Building a Real-Time Sliding-Window Dashboard over Streaming Data

Rahul Bhartia

October 2014

Contents

Abstract

Apache Storm developers can use Amazon Kinesis to quickly and cost effectively build real-time analytics dashboards and applications that can continuously process very high volumes of streaming data, such as clickstream log files and machine-generated data.

This whitepaper outlines a reference architecture for building a system that performs real-time sliding-windows analysis over clickstream data using Amazon Kinesis and Apache Storm. We will use Amazon Kinesis for managed ingestion of streaming data at scale with the ability to replay past data, and we'll run sliding-window analytics to power dashboards in Apache Storm. We build the system based on the reference architecture, which demonstrates everything from ingestion, processing, and storing to visualizing of the data in real-time.

Introduction

Streams of data are ubiquitous today – clickstreams, log data streams, event streams, and more. The need for real-time processing of high-volume data streams is pushing the limits of traditional data processing infrastructures. Building a clickstream monitoring system, for example, where data is in the form of a continuous clickstream rather than discrete data sets, requires the use of continuous queries, rather than ad-hoc one-time queries.

In this whitepaper, we propose a reference architecture for ingesting, analyzing, and processing vast amounts of clickstream data generated at very high rates in a smart and cost-efficient way using Amazon Kinesis with Apache Storm. We also explore the use of Amazon ElastiCache (Redis) as an in-memory data store for aggregated counters and use of its Pub/Sub facility to publish the counters on a simple dashboard.

We demonstrate the architecture by implementing a time-sensitive, sliding-window analysis over continuous clickstream data to determine how many *visitors* originated from specific *referrers*. Sliding-window analysis is a common way to analyze and identify trends in the clickstream data. The term *sliding-window analysis* refers to a common pattern analysis of real-time and continuous data and uses rolling counts of incoming data to examine trending, volumes, and rankings.

Reference Architecture

The reference architecture outlined below demonstrates using Amazon Kinesis for real-time data ingestion and Apache Storm for processing streaming data. To demonstrate the reference architecture, we create an application that puts simulated URLs and referrers into an Amazon Kinesis stream. Then we use Apache Storm to process the Amazon Kinesis stream and calculate how many visitors originated from a particular referrer. Storm also persists the aggregated counters in ElastiCache and publishes the counter on a pubsub channel. Using Node.Js, we further translate the messages from the Pub/Sub channel into server-side events (SSEs). Epoch, which uses D3, is used to build a real-time visualization by consuming the server-side events.
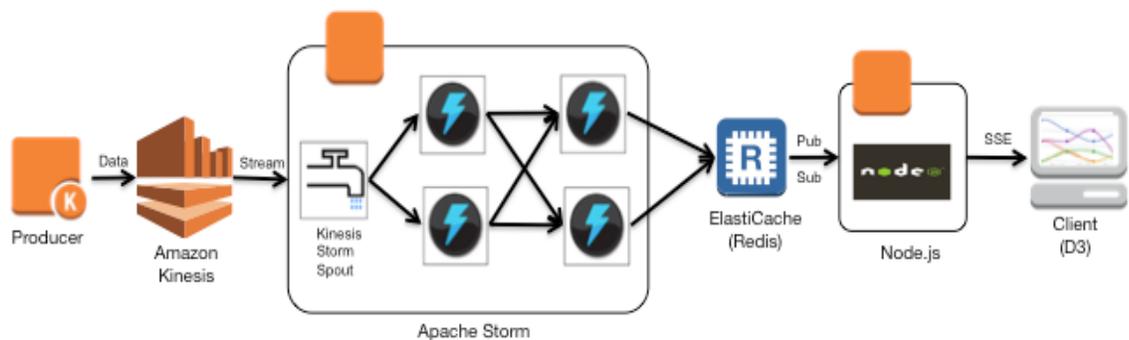
Let's start by taking a look at each component of our reference architecture and the role it plays in our architecture.

Amazon Kinesis

Amazon Kinesis handles streaming of high-volume, continuously generated data in our reference architecture. Amazon Kinesis is a fully managed service for real-time processing of streaming data at massive scale. Using Amazon Kinesis, developers can continuously capture and store terabytes of data per hour from hundreds of thousands of sources, including website clickstreams, financial transaction data, social media feeds, IT logs, location-tracking events, and more. Developers can then write stream processing applications that consume the Kinesis streams to take action on real-time data, and power real-time dashboards, generate alerts, implement real-time business logic, or even emit data to other big data services such as Amazon Simple Storage Service (Amazon S3), Amazon Redshift, and more.

The Kinesis Client Library enables developers to build streaming data processing applications. Leveraging the client library, developers can focus on business logic and let the client library automatically handle complex issues like adapting to changes in stream volume, load-balancing streaming data, coordinating distributed services, and processing data with fault-tolerance. Amazon Kinesis Connector Library further helps developers integrate Amazon Kinesis with other services from AWS, such as Amazon

DynamoDB, Redshift, and Amazon S3. Amazon Kinesis also has connectors for other applications and distributed systems, like Apache Storm.

Apache Storm

Apache Storm handles continuous processing of the Amazon Kinesis streams in our reference architecture. Apache Storm is a real-time distributed computing technology for processing streaming messages on a continuous basis. Individual logical processing units (known as *bolts* in Storm terminology) are connected like a pipeline to express the series of transformations while also exposing opportunities for concurrent processing.

Data streams are ingested in Storm via *spouts*. Each spout emits one or more sequences of tuples into a cluster for processing. A given bolt can consume any number of input spouts, process the tuples, and emit transformed tuples. A *topology* is a multi-stage distributing computation application composed of a network of spouts and bolts. A topology (which you can think of as a full application) runs continuously in order to process incoming data.

We'll use the Amazon Kinesis Storm Spout to integrate Amazon Kinesis with Storm for real-time computation of our clickstream logs.

Amazon ElastiCache

Amazon ElastiCache handles the persistent storage of aggregated counters in an in-memory store and publication of these counters in our reference architecture. ElastiCache is a web service for forming the data store and notification layer that makes it easy to deploy, operate, and scale an in-memory cache in the cloud. The service improves the performance of web applications by allowing you to retrieve information from fast, managed, in-memory caches, instead of relying entirely on slower disk-based databases.

ElastiCache also supports Redis—a popular open-source in-memory key-value store that supports data structures such as sorted sets and lists. Redis also has a subscription model that client apps can use to receive notifications. In this model, our client app—the visualization layer—is notified of the change and then updates the visualizations to reflect that change.

While our stack mainly uses the Redis Pub/Sub facility to stream the updates to the visualization layer, you can also use Redis to persist the aggregated counters in the simple key-value store that Redis provides. Using such a persistent store, multiple applications across an enterprise can present different views or bootstrap to a known state in case of failures.

Node.js

Node.js is the component handing the conversion of published counters into server side events in our reference architecture. Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices. The Node.js application subscribes to the Redis Pub/Sub channel and translates those notifications to Server-Sent Events (SSEs). SSE is a W3C specification that describes how a server can push events to browser clients using the standard HTTP protocol.

Epoch and D3

Epoch is a real-time charting library built on D3, which handles the visualization of the server-side events to provide a real-time dashboard. D3.js is a JavaScript library for manipulating documents based on data. D3 helps bring data to life using HTML, SVG, and CSS. Epoch is a general purpose library for application developers and visualization designers. It focuses on two different aspects of visualization programming: basic charts for creating historical reports, and real-time charts for displaying frequently updating time-series data.

Deploying on AWS

We deploy our application using AWS CloudFormation, a service that gives developers and systems administrators an easy way to create and manage a collection of related AWS resources, provisioning and updating them in an orderly and predictable fashion. The sample AWS CloudFormation template included here provisions three Amazon EC2 instances and starts all the applications on them. The AWS CloudFormation stack also creates an IAM Role to allow the application to authenticate your account without the need for you to provide explicit credentials. See Using IAM Roles for EC2 Instances with the SDK for Java for more information.

You can run the AWS CloudFormation template to create the stack we described. In the following sections we walk through the details about processing in each individual layer.

Streaming Data in Amazon Kinesis

In a real-world scenario, your log servers, application servers, or app running on smartphones/ tablets are producing the clickstream data captured by your Amazon Kinesis stream. For our application, we'll use a sample producer app, HttpReferrerStreamWriter.java to generate simulated URLs in JSON format. This producer will automatically create an Amazon Kinesis stream for you and continuously emit simulated data into the stream.

This producer is also a part of the [Amazon Kinesis Data Visualization Sample Application](), which demonstrates how to interact with Amazon Kinesis to generate meaningful statistics from a stream of data and visualize those results in real time.

Accessing Kinesis Stream

After the AWS CloudFormation template completes deployment, navigate to the Amazon Kinesis console and click the stream name, which you will find prefixed by the template name, followed by 'KinesisStream' with two shards, and 'Active' status.

On the Stream Details page for the stream, the first two graphs display write capacity of two MB/sec and read capacity of four MB/sec based on two shards available for the stream, as shown in the following illustration. The Amazon Kinesis stream is now ingesting data submitted to it by the producer.
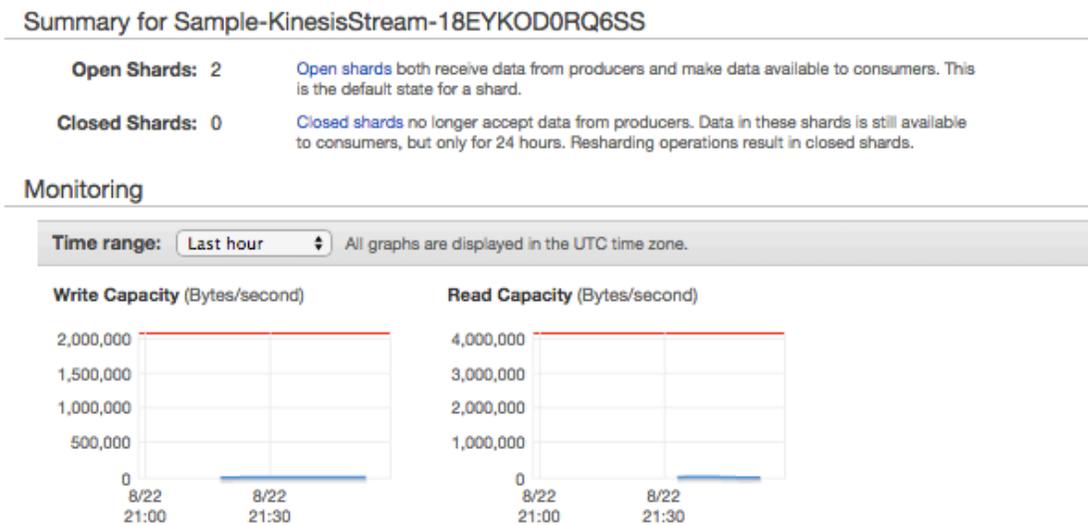
Continuous Processing Using Storm

Amazon Kinesis Storm Spout fetches data records from Amazon Kinesis and emits them as tuples. The spout state is stored in Apache ZooKeeper, part of the Storm cluster, to track the current position in the stream.

To use this spout, we start creating our Storm topology as shown in the following example:

```
…
public static void main(String[] args) throws ...
{
…
```

```
if (
…
propertiesFile = args[0];
mode = args[1];
…
configure(propertiesFile);

final KinesisSpoutConfig config = new
KinesisSpoutConfig(streamName,
zookeeperEndpoint).withZookeeperPrefix(zookeeperPrefix)
.withInitialPositionInStream(initialPositionInStream)
.withRegion(Regions.fromName(regionName));

final KinesisSpout spout = new KinesisSpout(config, new
CustomCredentialsProviderChain(), new
ClientConfiguration());
…
// Using number of shards as the parallelism hint for the
spout.
builder.setSpout("Kinesis", spout, 2);
…
```

In the excerpt above, note the following:

- The `configure` function initializes the variable from the properties file passed as an argument to the application.

- `KinesisSpoutConfig` configures the spout, including the Storm topology name, the Amazon Kinesis stream name endpoint for connecting to ZooKeeper, and the prefix for the ZooKeeper paths where the spout state is stored.

- `KinesisSpout` constructs an instance of the spout, using your AWS credentials and the configuration specified in `KinesisSpoutConfig`. Each task executed by the spout operates on a distinct set of Amazon Kinesis shards and periodically commits their state to ZooKeeper.

The Kinesis Spout by default emits a tuple of `(partitionKey, record)` as specified by `DefaultKinesisRecordScheme`. If you want to emit more structured data, you can provide your own implementation of `IKinesisRecordScheme`. In this whitepaper we use a *parse* bolt to process the *record* field back into original referrer and resource components using the following code:

```
public void execute(Tuple input,  BasicOutputCollector
collector) {
```

```
Record record =
(Record)input.getValueByField(DefaultKinesisRecordScheme.FI
ELD_RECORD);
ByteBuffer buffer = record.getData();
…
data = decoder.decode(buffer).toString();
JSONObject jsonObject = new JSONObject(data);

String referrer = jsonObject.getString("referrer");
```

We add this bolt to our topology and connect with the Kinesis spout:

```
builder.setBolt("Parse", new ParseReferrerBolt(),
6).shuffleGrouping("Kinesis");
```

At this point, we have a partial storm topology that reads data from Amazon Kinesis and emits the simulated referrer for each record generated by our producer application. In the next section we implement a sliding-window counter by using the storm-starter package library and connect it to the Redis Pub/Sub channel.

Implementing a Sliding Window

The storm-starter project provides sample implementations of various real-time data processing topologies, including the RollingTopWords topology, which can be used for computing trending topics. In this whitepaper, we reuse most of the RollingCountBolt topology to perform the rolling counts of incoming objects, i.e., sliding-window based counting. The bolt is configured by two parameters:

•   Length of the sliding window in seconds (which influences the output data of the bolt, i.e., how it counts objects)

•   Emit frequency in seconds (which influences how often the bolt will output the latest window counts)

The RollingCountBolt uses tick tuple to emit the count at the specified frequency and is configured in the bolt as shown in the following example:

```
public Map<String, Object> getComponentConfiguration()
{

Map<String, Object> conf = new HashMap<String, Object>();
conf.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS,emitFrequency
InSeconds);
```

amazon
webservices

```
    return conf;
    }
```

In the `execute` method of the bolt, we use the helper function `isTickTuple` to identify whether the incoming tuple is a tick tuple. If the incoming tuple is a tick tuple, we proceed with emitting the current count. Otherwise, we increment the counter for the sliding window.

```
…
public void execute(Tuple tuple)
{
 if (TupleHelpers.isTickTuple(tuple))
 {
   LOG.debug("Received tick tuple, triggering emit of
current window counts");
   emitCurrentWindowCounts();
 }
 else {
   countObjAndAck(tuple);
 }
}
```

The bolt uses two main data structures to implement a sliding-window counter:

- `SlidingWindowCounter` provides a sliding-window count for each tracked object. The window in our case, however, does not advance with time, but whenever (and only when) the method `getCountsThenAdvanceWindow()` is called during the `emit` method.

- `NthLastModifiedTimeTracker` is used to track the actual duration of the sliding window. It is included in case the expected sliding-window length (as configured by the user) is different from the actual length.

```
    private void emitCurrentWindowCounts() {
        Map<Object, Long> counts =
    counter.getCountsThenAdvanceWindow();
        int actualWindowLengthInSeconds =
    lastModifiedTracker.secondsSinceOldestModification();
        lastModifiedTracker.markAsModified();
        if (actualWindowLengthInSeconds !=
    windowLengthInSeconds) {
```

```
    LOG.warn(String.format(WINDOW_LENGTH_WARNING_TEMPLATE,
    actualWindowLengthInSeconds, windowLengthInSeconds));
            }
            emit(counts, actualWindowLengthInSeconds);
        }

        private void countObjAndAck(Tuple tuple) {
            Object obj = tuple.getValue(0);
            counter.incrementCount(obj);
            collector.ack(tuple);
        }
```

At this point we have all the pieces in place to compute a sliding-window count of referrers coming in a stream of data from Amazon Kinesis. We finally modify the `emit` function of the `RollingCountBolt` to publish the current count over a Pub/Sub channel of an ElastiCache Redis channel.

```
    for (Entry<Object, Long> entry : counts.entrySet())
    {
      …
      JSONObject msg = new JSONObject();
      …
      msg.put("name", referrer);
      msg.put("time", currentEPOCH);
      msg.put("count", count);
      …
      jedis.publish("ticker",msg.toString());
```

Accessing the Storm User Interface

The AWS CloudFormation template also deploys Storm and Zookeeper over a single Amazon EC2 instance, and starts the topology described above. You can find the URL for the Storm UI from the key named "StormURL" in the output section of the cloud-formation template. Navigate to the URL and click the topology named 'SampleTopology'.

Under the topology, you should see that the spout named "Kinesis" has two tasks; each task reading data in parallel from individual shards.

Topology summary

Name	Id	Status	Uptime
SampleTopology	SampleTopology-1-1408743330	ACTIVE	24m 39s

Topology actions

Activate Deactivate Rebalance Kill

Topology stats

Window	Emitted	Transferred	Complete latency
10m 0s	40380	40380	51.814
3h 0m 0s	163940	163940	20.336
1d 0h 0m 0s	163940	163940	20.336
All time	163940	163940	20.336

Spouts (All time)

Id	Executors	Tasks	Emitted	Transferred	Complete la
Kinesis	2	2	81900	81900	20.336

Bolts (All time)

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed
Count	6	6	0	0	0.004	0.141	85020	0.015	82220	0
Parse	6	6	82040	82040	0.000	0.036	81960	0.035	81960	0

Building a Real-Time Data Visualization

Next we move to the last task—to present streaming data. Over the years, web standards have made great progress toward building data-centric applications with ease. We'll use SSEs, which allow a web app to subscribe to an event-stream instead of polling to create real-time visualizations.

Generating Server-Side Events

We use Node.js for its event-driven model to generate the SSEs. Our code creates a basic webserver and serves the content using the Connect middleware for Node.

```
connect()
.use(connect.static(__dirname))
.use(function(req,res) {
if(req.url == '/ticker') {
```

amazon
web services

```
  ticker(req,res);
 }
}
)
.listen(9000);
```

The `ticker` function subscribes to the Redis Pub/Sub channel, which is being constantly updated via our Storm topology. When a new counter event arrives on the Pub/Sub channel, the Node server appends the data and publishes it back as a server side event.

```
function ticker(req,res) {
 req.socket.setTimeout(Infinity);

  var subscriber =
redis.createClient(6379,process.argv[2]);

  subscriber.subscribe("pubsubCounters");

  // When we receive a message from the redis connection
  subscriber.on("message", function(channel, message) {

                                              res.j
son(message);
  });

  //send headers for event-stream connection
  res.writeHead(200, {
    'Content-Type': 'text/event-stream',
    'Cache-Control': 'no-cache',
    'Connection': 'keep-alive'
  });

  res.json = function(obj) { res.write("data:
"+obj+"\n\n"); }
  res.json(JSON.stringify({}));

  // The 'close' event is fired when a user closes their
  browser window.
  req.on("close", function() {
    subscriber.unsubscribe();
    subscriber.quit();
  });
}
```

Visualization Using Epoch

The index.html page served as default by Connect middleware has the client-side JavaScript that uses Epoch to render the real-time chart. In the script, we create an empty chart bound to the respective referrer variable.

```
var siteArray = [ "amazon", "yahoo", "bing", "reddit",
"google", "stackoverflow" ];
...
 var chartDiv = document.createElement('div');
 chartDiv.id = siteArray[i]+'Chart';
 chartDiv.className = "epoch category20";
 chartDiv.style.width='89%';
 chartDiv.style.height='100%';
 chartDiv.style.position='relative';
 chartDiv.style.float='left';


...

 eval("var "+siteArray[i]+"Data =
[{label:\""+siteArray[i]+"\",values:[{time:currentTime,y:0}
]}];")
 eval("var "+siteArray[i]+" =
$('#"+siteArray[i]+"Chart').epoch({type:'time.area',data:"+
siteArray[i]+"Data,axes:['left','bottom','right']});")
 }
```

The script adds an event-listener on the event stream that is published by the Node server.

```
var source = new EventSource('/ticker');
source.addEventListener('message',tick);
```

The `event-listener` function simply parses the payload to find the time when the counter was generated and its value. The function then updates the dataset with the new dataset and redraws the graph created on every event.

```
function tick(e) {
 if(e)
 {
  var eventData = JSON.parse(e.data);
    window[eventData.name].push([{ time: eventData.time, y:
eventData.count}]);
```

```
      }
    }
```

Accessing the Dashboard

Our AWS CloudFormation template also deploys Node.js on a single-node Amazon EC2 instance for the server. You can find the URL for the Node server from the key named "VisualizationURL" under the output section of the AWS CloudFormation template. Navigate to the dashboard at that URL and you should see the graph being updated in real time as the data arrives.

Demo: real-time clickstream visualization

The clickstream data is being streamed into Amazon Kinesis and Apache storm is used to compute the count by referrers over a sliding window of 2 secs. The couters are made available to the dashboard using server-side events, which are further composed from Redis PubSub channel. The visualization was done using D3 and Epoch
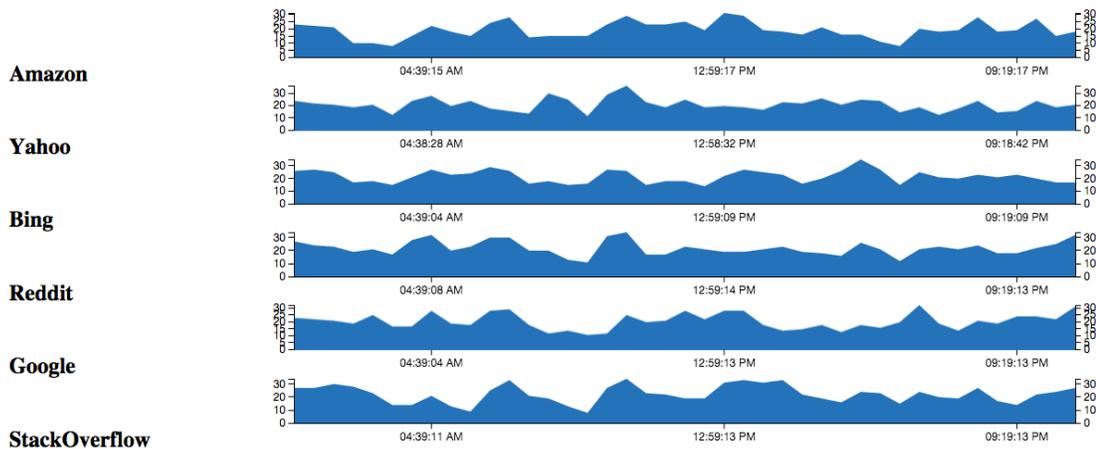
Deleting the CloudFormation Stack

Once you've run through the stack and understand how several pieces of our reference architecture fit together, you can terminate the application resources created by the AWS CloudFormation stack. To do so, navigate to the AWS Management Console (https://console.aws.amazon.com/) and select the AWS CloudFormation service. Select the stack that you created and then click on **Delete Stack**. When AWS CloudFormation is finished removing all resources, the stack should be removed from the list.

Conclusion

In this whitepaper, we looked at how to implement real-time visualization of sliding-windows analysis over streaming data. In actual scenarios, real-time streaming data not only entails collection of data at scale but also being able to process, store, and deliver the results on a real-time basis. This brings in a number of challenges and requires a decoupled architecture for streaming, processing, storage, and delivery. Using systems for each layer, which can scale but easily integrate with other layers, allows for the ever-expanding data velocity in real-time streaming system.

Amazon Kinesis also supports multi-reader applications, allowing you to build multiple paths for the same data running at different latencies from a single stream. Using Amazon Kinesis connectors, you can also deliver the same stream of data to the other stores of your choice—Amazon S3, Amazon Redshift, and Amazon DynamoDB. Amazon S3 offers highly durable and available cloud storage for a variety of content, ranging from web applications to media files, and can be used to build a cost-effective archive by running an application once every few hours. You can also use Amazon EMR clusters to read and process Amazon Kinesis streams directly, and use familiar tools like Hive, Pig, and MapReduce to analyze your data.