Implementing Real-Time Analysis with Hadoop in Azure HDInsight

Lab 2 - Getting Started with Storm (Java)

Overview

In this lab, you will provision HDInsight Storm cluster, and use Java to create a simple Storm topology that captures a stream of simulated sensor data. You will then enhance the topology to implement guaranteed message processing, before finally implementing a sliding window that aggregates data over a specified timespan.

What You'll Need

To complete the labs, you will need the following:

- A web browser
- A Microsoft account
- A Microsoft Azure subscription
- A Windows, Linux, or Mac OS X computer
- The lab files for this course

Note: To set up the required environment for the lab, follow the instructions in the **Setup** document for this course. Specifically, you must have signed up for an Azure subscription.

Provisioning HDInsight Clusters

The first task you must perform is to provision an HDInsight Storm cluster to capture and process as real-time stream of data, and an HBase cluster in which the captured data will be stored.

Note: The Microsoft Azure portal is continually improved in response to customer feedback. The steps in this exercise reflect the user interface of the Microsoft Azure portal at the time of writing, but may not match the latest design of the portal exactly.

Provision an HDInsight Storm Cluster

To use Storm, you must provision an HDInsight Storm cluster.

1. In the Microsoft Azure portal, in the Hub Menu, click **New**. Then in the **Data + Analytics** menu, click **HDInsight**.
2. In the **New HDInsight Cluster** blade, enter the following settings, and then click **Create**:

- **Cluster name**: *Enter a unique name (and make a note of it!)*
- **Subscription**: *Select your Azure subscription*
- **Cluster type**:
    - **Cluster Type**: Storm
    - **Cluster Operating System**: Linux
    - **Version**: *Choose the latest version of Storm available.*
    - **Cluster Tier**: Standard
- **Resource Group**: *Create a new resource group with a unique name*
- **Credentials:**
    - **Cluster Login Username**: *Enter a user name of your choice (and make a note of it!)*
    - **Cluster Login Password**: *Enter and confirm a strong password (and make a note of it!)*
    - **SSH Username:** *Enter another user name of your choice (and make a note of it!)*
    - **SSH Password:** *Use the same password as the cluster login password*
- **Storage:**
    - **Create a new storage account**: *Enter a unique name consisting of lower-case letters and numbers only (and make a note of it!)*
    - **Choose Default Container**: *Enter the cluster name you specified previously*
- **Applications:** *None*
- **Cluster size**: *Configure the following settings in the **Pricing** blade, and then click **Select***
    - **Number of Supervisor nodes**: 1
    - **Supervisor Node Size**: *Leave the default size selected*
    - **Nimbus Node Size**: *Leave the default size selected*
    - **Zookeeper Node Sizes**: *Leave the default size selected*
- **Advanced Settings**: *None*

3. At the top of the page, click **Notifications** and verify that deployment has started. Then, while waiting for the cluster to be deployed (which can take a long time – often 30 minutes or more), move onto the next procedure

Note: As soon as an HDInsight cluster is running, the credit in your Azure subscription will start to be charged. Free-trial subscriptions include a limited amount of credit limit that you can spend over a period of 30 days, which should be enough to complete the labs in this course as long as clusters are deleted when not in use. If you decide not to complete this lab, follow the instructions in the *Clean Up* procedure at the end of the lab to delete your cluster to avoid using your Azure credit unnecessarily.

Creating a Storm Topology

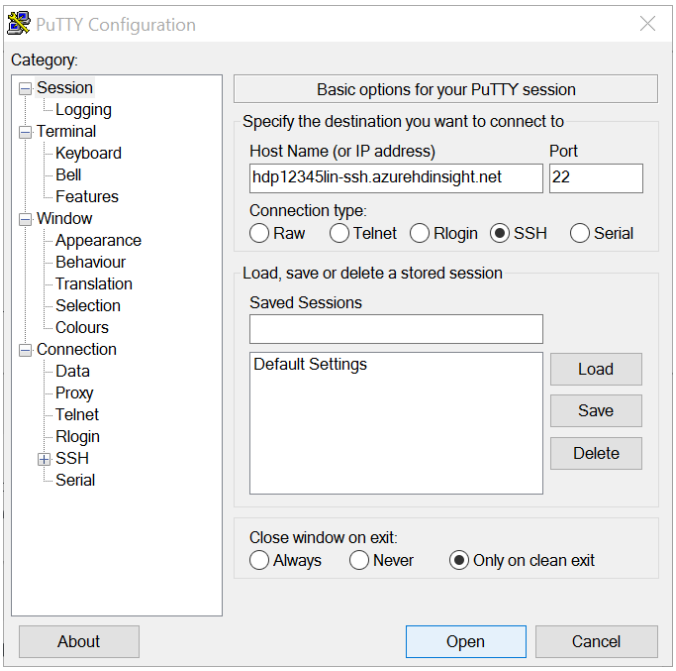
In the previous exercise, you provisioned a Storm HDInsight cluster. In this exercise, you will create a simple Storm topology based on the default template provided with the Azure SDK.

Open a Secure Shell Connection to the Cluster

To work with Storm in your cluster, you will open a secure shell (SSH) connection.

If you are using a Windows client computer:

1. In the Microsoft Azure portal, browse to the blade for your HDInsight cluster, click **Secure Shell**, and then in the **Secure Shell** blade, under **Windows users**, copy the **Host name** (which should be ***your_cluster_name*-ssh.azurehdinsight.net**) to the clipboard.
2. Open PuTTY, and in the **Session** page, paste the host name into the **Host Name** box. Then under **Connection type**, select **SSH** and click **Open**.



3. If a security warning that the host certificate cannot be verified is displayed, click **Yes** to continue.
4. Wen prompted, enter the SSH username and password you specified when provisioning the cluster (not the cluster login).

If you are using a Mac OS X or Linux client computer:

1. In the Microsoft Azure portal, on the **HDInsight Cluster** blade for your HDInsight cluster, click **Secure Shell**, and then in the **Secure Shell** blade, under **Linux, Unix, and OS X users**, note the command used to connect to the head node.
2. Open a new terminal session, and enter the following command, specifying your SSH user name (not the cluster login) and cluster name as necessary:

```
ssh your_ssh_user_name@your_cluster_name-ssh.azurehdinsight.net
```

Note: The commands and code used in this lab are case-sensitive.

3. If you are prompted to connect even though the certificate can't be verified, enter **yes**.
4. When prompted, enter the password for the SSH username.

Note: If you have previously connected to a cluster with the same name, the certificate for the older cluster will still be stored and a connection may be denied because the new certificate does not match the stored certificate. You can delete the old certificate by using the **ssh-keygen** command, specifying the path of your certificate file (**f**) and the host record to be removed (**R**) - for example:

```
ssh-keygen -f "/home/usr/.ssh/known_hosts" -R clstr-ssh.azurehdinsight.net
```

Install and Configure Maven

The Java Developer Kit (JDK) is installed on HDInsight head nodes, and can be used to develop Java components for Storm topologies. Maven is a project build system for Java that simplifies the process of developing and building Java packages with the JDK.

Note: In a real scenario, you would typically install the development tools on your local workstation, develop and build your solutions there, and then deploy them to the cluster to run them. However, for the purposes of this lab, installing the tools on the cluster node simplifies setup.

1. In the SSH console window, enter the following command to view the version of the JDK installed on the head node:

   ```
   javac -version
   ```

2. Enter the following command to verify that the **JAVA_HOME** system variable is set:

   ```
   echo $JAVA_HOME
   ```

3. Enter the following command to install Maven:

   ```
   sudo apt-get install maven
   ```

4. If you are prompted to confirm the installation, enter **Y** and wait for the installation to complete (this may take a few minutes).
5. After Maven is installed, enter the following command to view details of the Maven installation:

   ```
   mvn -v
   ```

6. Note the Maven home path (which should be similar to **/usr/share/maven**), and then enter the following command to add the **bin** subfolder of this path to the system **PATH** variable:

   ```
   export PATH=/usr/share/maven/bin:$PATH
   ```

Create a Maven Project

1. In the SSH console, enter the following command (on a single line) to create a new Maven project named **SensorStream**:

   ```
   mvn archetype:generate -DarchetypeArtifactId=maven-archetype-quickstart
   -DgroupId=lex.microsoft.com -DartifactId=SensorStream -
   DinteractiveMode=false
   ```

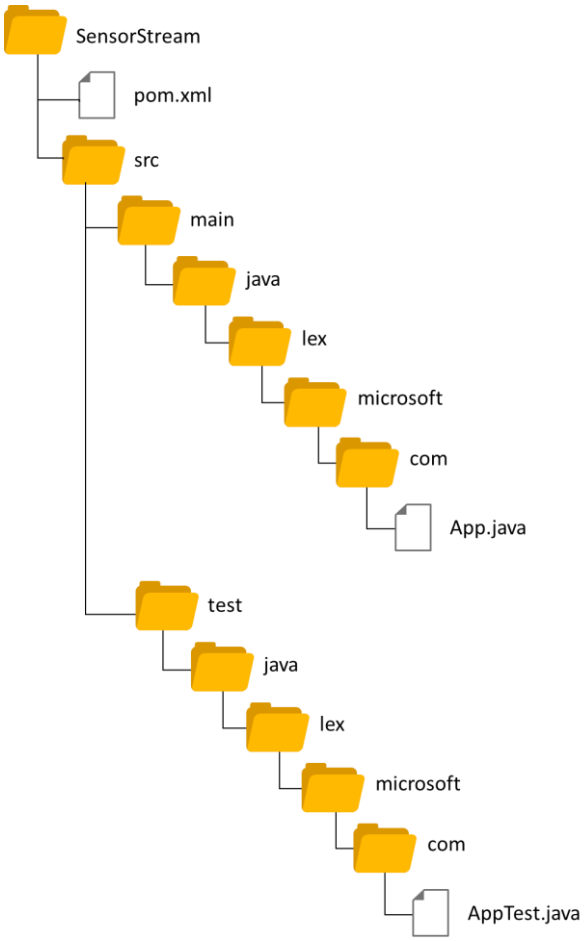
2. After the project is created, enter the following command to change the current directory to the directory for your **SensorStream** project:

   ```
   cd SensorStream
   ```

3. Enter the following command to view the directory hierarchy created for the project:

   ```
   ls -R
   ```

4. View the directory listing, and note that the directory structure for the project matches the following image:



5. You will create your own application code file, so enter the following command to delete the default code file generated from the project template:

```
rm src/main/java/lex/microsoft/com/App.java
```

6. You will not require the test harness generated by from the project template, so enter the following command to delete it.

```
rm -r src/test
```

Configure the Project

1. Enter the following command to open the **pom.xml** file in the Nano text editor. This file contains configuration information for the project:

```
nano pom.xml
```

2. Edit the file to remove the dependency on **junit,** which is not required since you won't be using the test harness (to remove the current line in Nano, press **CTRL+K**). Then add the sections indicated in **bold** below (you can copy this from the **pom.xml** file in the **Lab02\Java** folder where you extracted the lab files for this course). This adds a dependency on the storm-core library (which is provided by the cluster, and so will not be packaged in the JAR file for the compiled project), and adds plug-ins that make it easier to test and compile the project.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>lex.microsoft.com</groupId>
  <artifactId>SensorStream</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>SensorStream</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>org.apache.storm</groupId>
      <artifactId>storm-core</artifactId>
      <version>0.10.0</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>exec-maven-plugin</artifactId>
        <executions>
          <execution>
          <goals>
            <goal>exec</goal>
          </goals>
          </execution>
        </executions>
        <configuration>
          <executable>java</executable>
          <includeProjectDependencies>true</includeProjectDependencies>
          <includePluginDependencies>false</includePluginDependencies>
          <classpathScope>compile</classpathScope>
          <mainClass>${storm.topology}</mainClass>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

3. Exit the Nano editor (enter **CTRL+X**), saving the **pom.xml** file (enter **Y** and **ENTER** when prompted).

Implement a Spout

1. Enter the following command to open Nano and create a code file named **SensorSpout.java** in the **src/main/java/lex/microsoft/com** directory:

   ```
   nano src/main/java/lex/microsoft/com/SensorSpout.java
   ```

2. In Nano, enter the following code - you can copy and paste this from **SensorSpout.java** in the **Lab02\Java** folder where you extracted the lab files for this course:

```
package lex.microsoft.com;

import backtype.storm.spout.SpoutOutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichSpout;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Values;
import backtype.storm.utils.Utils;

import java.util.Map;
import java.util.Random;

//This spout randomly emits sensor readings
public class SensorSpout extends BaseRichSpout {
  //Collector used to emit output
  SpoutOutputCollector _collector;
  //Used to generate a random number
  Random _rand;

  //Open is called when an instance of the class is created
  @Override
  public void open(Map conf, TopologyContext context,
SpoutOutputCollector collector) {
  //Set the instance collector to the one passed in
    _collector = collector;
    //For randomness
    _rand = new Random();
  }

  //Emit data to the stream
  @Override
  public void nextTuple() {
    //Sleep for a bit
    Utils.sleep(100);
    // Generate a random sensor (0 to 5) to represent a turnstile
    Integer sensorNumber = _rand.nextInt(4) + 1;
    String sensorName = "Turnstile " + String.valueOf(sensorNumber);
    //Emit the sensor name
    _collector.emit(new Values(sensorName));
  }

  //Ack is not implemented since this is a basic example
  @Override
  public void ack(Object id) {
  }
```

```
  //Fail is not implemented since this is a basic example
  @Override
  public void fail(Object id) {
  }

  //Declare the output fields.
  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("sensor"));
  }
}
```

3. Review the code, noting that it emits random integer values that simulate readings from sensors, which are randomly named in the format **Turnstile*N***. Each time a sensor is triggered, it represents a turnstile being used.
4. Exit the Nano editor, saving the **SensorSpout.java** file.

Implement a Bolt

1. Enter the following command to open Nano and create a code file named **SensorBolt.java** in the **src/main/java/lex/microsoft/com** directory:

```
nano src/main/java/lex/microsoft/com/SensorBolt.java
```

2. In Nano, enter the following code - you can copy and paste this from **SensorBolt.java** in the **Lab02\Java** folder where you extracted the lab files for this course:

```
package lex.microsoft.com;

import java.util.HashMap;
import java.util.Map;

import backtype.storm.topology.BasicOutputCollector;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseBasicBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;

//There are a variety of bolt types. In this case, we use BaseBasicBolt
public class SensorBolt extends BaseBasicBolt {
  //For holding sensor entry counts
    Map<String, Integer> entries = new HashMap<String, Integer>();

    //execute is called to process tuples
    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {
      //Get the sensor name from the tuple
      String sensor = tuple.getString(0);
      Integer count = entries.get(sensor);
      if (count == null)
        count = 0;
      //Increment the count and store it
      count++;
      entries.put(sensor, count);
```

```
      //Emit the sensor and the current count of entries
      collector.emit(new Values(sensor, count));
    }

    //Declare that we will emit a tuple containing two fields; sensor
and entries
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("sensor", "entries"));
    }
}
```

3. Review the code, noting that it reads the sensor name from the stream and maintains a running count of the number of time each sensor has been triggered (which represents the number of entries there have been through each turnstile). In a real application, this data would typically be aggregated and stored in a database or HBase table.
4. Exit the Nano editor, saving the **SensorBolt.java** file.

Create a Topology

1. Enter the following command to open Nano and create a code file named **SensorTopology.java** in the **src/main/java/lex/microsoft/com** directory:

```
nano src/main/java/lex/microsoft/com/SensorTopology.java
```

2. In Nano, enter the following code - you can copy and paste this from **SensorTopology.java** in the **Lab02\Java** folder where you extracted the lab files for this course:

```
package lex.microsoft.com;

import backtype.storm.Config;
import backtype.storm.LocalCluster;
import backtype.storm.StormSubmitter;
import backtype.storm.topology.TopologyBuilder;
import backtype.storm.tuple.Fields;

import lex.microsoft.com.SensorSpout;

public class SensorTopology {

  //Entry point for the topology
  public static void main(String[] args) throws Exception {
  //Used to build the topology
    TopologyBuilder builder = new TopologyBuilder();
    //Add the spout, with 2 executors
    builder.setSpout("sensorspout", new SensorSpout(), 2);
    //Add the SensorBolt bolt, with 5 executors
    //fieldsgrouping ensures that the same sensor is sent to the same
bolt instance
    builder.setBolt("sensorbolt", new SensorBolt(),
5).fieldsGrouping("sensorspout", new Fields("sensor"));

    //new configuration
    Config conf = new Config();
    conf.setDebug(true);
```

```
    //If there are arguments, we are running on a cluster
    if (args != null && args.length > 0) {
      //parallelism hint to set the number of workers
      conf.setNumWorkers(3);
      //submit the topology
      StormSubmitter.submitTopology(args[0], conf,
builder.createTopology());
    }
    //Otherwise, we are running locally
    else {
      //Cap the maximum number of executors that can be spawned
      //for a component to 3
      conf.setMaxTaskParallelism(3);
      //LocalCluster is used to run locally
      LocalCluster cluster = new LocalCluster();
      //submit the topology
      cluster.submitTopology("SensorTopology", conf,
builder.createTopology());
      //sleep
      Thread.sleep(20000);
      //shut down the cluster
      cluster.shutdown();
    }
  }
}
```

3. Review the code, noting that it creates a topology that consists of two instances of the sensor spout and five instances of the bolt. A **fieldsGrouping** configuration is used to send all events for each sensor to the same bolt instance.
4. Exit the Nano editor, saving the **SensorTopology.java** file.

Build and Run the Topology

1. Ensure that you are still in the **SensorStream** directory, and enter the following command to build and run the Maven project locally to test it:

   ```
   mvn compile exec:java -Dstorm.topology=lex.microsoft.com.SensorTopology
   ```

2. Wait for the project to compile, and observe as the topology runs locally to test the spout and bolt. After a while, the topology should stop and you should be able to see running counts for each turnstile sensor in the output.
3. Enter the following command to compile the project to a JAR file:

   ```
   mvn clean package
   ```

4. When the package is compiled, enter the following command to view the compiled JAR file, which should be named **SensorStream-1.0-SNAPSHOT.jar**:

   ```
   ls target
   ```

5. Enter the following command to submit the topology to the Storm cluster:

   ```
   storm jar target/SensorStream-1.0-SNAPSHOT.jar
   lex.microsoft.com.SensorTopology sensorstream
   ```

6. Wait for the topology to be submitted, and then in your browser, in the Azure portal, in the blade for your cluster, click **Dashboard** and enter the HTTP credentials for your cluster when prompted.
7. In the Ambari dashboard, in the list of services, click Storm, and verify that the **Topologies** metric shows the value **1**. This indicates that there is one topology running.
8. Open a new tab in your browser and navigate to the following address (substituting ***cluster*** for your cluster name). If prompted, log in using your cluster HTTP credentials:

   https://***cluster***.azurehdinsight.net/stormui

9. In the **Storm UI** page, under **Topology Summary**, note that the **sensorstream** topology is running. Then click **sensorstream** to view details of the topology.
10. In the **Topology Summary** page, under **Topology Stats**, view the number of tuples emitted; noting that the **Acked** and **Failed** metrics are 0.
11. Under **Topology Actions**, click **Kill**, and if prompted, confirm that you want to stop the topology within 30 seconds.
12. Switch back to the Ambari Storm service page, and refresh it to verify that the **Topologies** metric now shows the value **0**.

Implementing Retry Logic

In the previous exercise, you created and submitted a Storm topology that captures and counts simulated turnstile sensor readings. However, the topology is not transactional and has no retry logic; and in the event of a bolt being unable to process a tuple (for example, due to intermittent network connectivity issues), some data may be lost. In this exercise, you will enhance the topology to add retry logic to help mitigate tuple processing failures.

Storm supports a guaranteed messaging framework in which downstream bolts send acknowledgements (or *acks*) to the upstream components to verify that tuples have been processed. If a bolt fails to process a tuple, it can send a failure (*fail*) message back to the originating component so that the tuple can be retried.

Add Retry Logic to the Spout

The **BasicBolt** interface you used to process the turnstile sensor readings automatically sends *ack* and *fail* notifications to upstream components. All you need to do is to add code in the originating spout to:

- Assign a unique identifier to each emitted tuple.
- Cache each tuple as it is emitted.
- Remove cached tuples when they have been acked.
- Re-emit cached tuples when they fail.

Note: You can copy and paste the code in this procedure from **SensorSpout-Ack.java** in the **Lab02\Java** folder where you extracted the lab files for this course.

1. Enter the following command to open **SensorSpout.java** in Nano:

   ```
   nano src/main/java/lex/microsoft/com/SensorSpout.java
   ```

2. Add the following import statement under the existing import statements:

   ```
   import java.util.HashMap;
   ```

3. In the **SensorSpout** class, under the existing variable declarations above the **open** function, add the following variable declaration:

```
// Cache for emitted tuples
Map<String, Values> cache = new HashMap<String, Values>();
```

4. In the **nextTuple** function, find the following code:

```
//Emit the sensor name
_collector.emit(new Values(sensorName));
```

And replace it with this code:

```
// Create a values object
Values values = new Values(sensorName);

// generate an ID
String id =
values.toString().hashCode()+"_"+System.currentTimeMillis();

// cache the tuple and its id
cache.put(id, values);

//Emit the values
_collector.emit(values, id);
```

5. Add the following code to the **ack** function:

```
// remove item from cache
System.out.println(id +" acked, so remove it...");
cache.remove(id);
```

6. Add the following code to the **fail** function:

```
Values values = cache.get(id);
if(values != null){
    _collector.emit(values, id);
    System.out.println("\tReplay: " + values);
}
```

7. Exit the Nano editor, saving the **SensorSpout.java** file.

Build and Run the Topology

1. Ensure that you are still in the **SensorStream** directory, and enter the following command to build and run the Maven project locally to test it:

```
mvn compile exec:java -Dstorm.topology=lex.microsoft.com.SensorTopology
```

2. Wait for the project to compile, and observe as the topology runs locally to test the spout and bolt. After a while, the topology should stop and you should be able to see messages indicating that tuples were acked and removed from the cache.
3. Enter the following command to compile the project to a JAR file:

```
mvn clean package
```

4. When the package is compiled, enter the following command to view the compiled JAR file, which should be named **SensorStream-1.0-SNAPSHOT.jar**:

```
ls target
```

5. Enter the following command to submit the topology to the Storm cluster:

```
storm jar target/SensorStream-1.0-SNAPSHOT.jar
lex.microsoft.com.SensorTopology sensorstream
```

6. Wait for the topology to be submitted, and then in your browser, navigate to the Storm UI page at the following address (substituting ***cluster*** for your cluster name). If prompted, log in using your cluster HTTP credentials:

   https://***cluster***.azurehdinsight.net/stormui

7. In the **Storm UI** page, under **Topology Summary**, note that the **sensorstream** topology is running. Then click **sensorstream** to view details of the topology.
8. In the **Topology Summary** page, under **Topology Stats**, view the number of tuples emitted; noting the **Acked** metrics.
9. Refresh the page after a few seconds to verify that the number of tuples emitted and acked is increasing.
10. Under **Topology Actions**, click **Kill**, and if prompted, confirm that you want to stop the topology within 30 seconds.

Implement a Sliding Window

Streaming data processing solutions often aggregate data over temporal windows. For example, in the turnstile sensor solution you have built in the previous exercises in this lab, you may want to keep a running count of the number of turnstile entries within a recent timespan, such as the last hour. Furthermore, you may want to update this timespan aggregation at frequent intervals, for example every minute; so that the count recorded at 1:00pm includes entries from 12:00 to 1:00, the count at 1:01pm includes entries from 12:01 to 1:01, and so on. This kind of temporal scope for an aggregation is referred to as a sliding window.

Implement a Bolt for the Sliding Window

To implement a sliding window, you will add a new bolt to the topology. This bolt will consume both the default stream of turnstile sensor readings, and a system stream that contains *tick tuples* at a specified interval. The bolt will maintain a running count of sensor entries received from the default stream, but each time a tick tuple is received it will discard all entries that are older than the timespan that defines the sliding window.

Note: To make it easy to see the effects of the sliding window, you will implement a window that spans 10 seconds and which is updated every second; a real solution for this particular scenario may use longer time intervals (though many high-volume streaming solutions operate across very short time windows).

1. Enter the following command to open Nano and create a code file named **SlidingBolt.java** in the **src/main/java/lex/microsoft/com** directory:

```
nano src/main/java/lex/microsoft/com/SlidingBolt.java
```

2. In Nano, enter the following code - you can copy and paste this from **SlidingBolt.java** in the **Lab02\Java** folder where you extracted the lab files for this course:

```
package lex.microsoft.com;
```

```
import java.util.Map;
import java.util.Date;
import java.util.ArrayList;
import java.util.Iterator;
import backtype.storm.Config;
import backtype.storm.Constants;
import backtype.storm.topology.BasicOutputCollector;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseBasicBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;


public class SlidingBolt extends BaseBasicBolt {
  //For holding sensor entry counts
  ArrayList entries = new ArrayList();

  @Override
  public Map<String, Object> getComponentConfiguration() {
      Config conf = new Config();
      int tickFrequencyInSeconds = 1;
      conf.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS,
               tickFrequencyInSeconds);
      return conf;
  }

//execute is called to process tuples
    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {


if(tuple.getSourceStreamId().equals(Constants.SYSTEM_TICK_STREAM_ID)
&& tuple.getSourceStreamId().equals(Constants.SYSTEM_TICK_STREAM_ID)){
         System.out.println("********TICK_TUPLE RECEIVED*******");
         Date d = new Date();
         Long time = d.getTime();
         Integer before = entries.size();
         Iterator i = entries.iterator();
         while(i.hasNext()){
           Long t = (Long) i.next();
           if(t < (time - 10000)){
             i.remove();
           }
         }
         Integer after= entries.size();
         Integer deleted = before - after;
         System.out.println(Integer.toString(deleted) + " deleted.");

        } else {
          Date d = new Date();
          entries.add(d.getTime());
      }
          Integer count = entries.size();
```

```
            System.out.println(Integer.toString(count) + " entries in
last 10 seconds");

          //Emit the sensor and the count of entries in the last 10
seconds
            collector.emit(new Values(count));
    }

    //Declare that we will emit a tuple containing a single field
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("entries"));
    }
}
```

3. Review the code, and note the following features:
    - An ArrayList is used to store the time value for each turnstile sensor entry received by the bolt.
    - A procedure named **getComponentConfiguration** returns a **Map** object in which configuration settings for this bolt are defined. These settings override the default configuration for the bolt, and in this case they include a **TOPOLOGY_TICK_TUPLE_FREQ_SECS** setting with a value of 1 second. This causes the bolt to receive a tick tuple every second from a system stream.
    - In the execute function, the code first checks to see if a tick tuple has been received. If so, all entries older than 10,000 milliseconds (10 seconds) are removed from the ArrayList used to store entry reading times.
    - When a tuple is received from the default stream, the current time value (which is a measure of milliseconds since January 1st 1970) is stored in the ArrayList.
    - The execute method determines the number of entries in the ArrayList (which represents the number of entries in the past 10 seconds) and emits it.
4. Exit the Nano editor, saving the **SlidingBolt.java** file.

Add the New Bolt to the Topology

To include the **SlidingBolt** bolt in the topology, you must modify the **SensorTopology** class and specify from which component the bolt should receive tuples, and how many instances of the bolt should be created.

1. Enter the following command to open **SensorTopology.java** in Nano:

    ```
    nano src/main/java/lex/microsoft/com/SensorTopology.java
    ```

2. In the main function, modify the existing code to build the topology by adding the code in **bold** below - you can copy and paste this from **SensorTopology-Sliding.java** in the **Lab02\Java** folder where you extracted the lab files for this course:

    ```
    //Used to build the topology
    TopologyBuilder builder = new TopologyBuilder();
    //Add the spout, with 2 executors
    builder.setSpout("sensorspout", new SensorSpout(), 2);
    //Add the SensorBolt bolt, with 5 executors
    //fieldsgrouping ensures that the same sensor is sent to the same bolt
    instance
    ```

```
builder.setBolt("sensorbolt", new SensorBolt(),
5).fieldsGrouping("sensorspout", new Fields("sensor"));
//Add the Sliding bolt, with 1 executors
builder.setBolt("slidingbolt", new SlidingBolt(),
1).shuffleGrouping("sensorspout");
```

Note that this code adds a single instance of bolt named **slidingbolt** to the topology and configures it to receive tuples from the **sensorspout** spout.

3. Exit the Nano editor, saving the **SensorTopology.java** file.

Build and Run the Topology

1. Ensure that you are still in the **SensorStream** directory, and enter the following command to build and run the Maven project locally to test it:

```
mvn compile exec:java -Dstorm.topology=lex.microsoft.com.SensorTopology
```

2. Wait for the project to compile, and observe as the topology runs locally to test the topology. After a while, the topology should stop and you should be able to review the output and see messages indicating:
   - The number of entries in the last 10 seconds
   - When a tick tuple was received.
   - The number of entries deleted with each tick tuple.

3. Enter the following command to compile the project to a JAR file:

```
mvn clean package
```

4. When the package is compiled, enter the following command to view the compiled JAR file, which should be named **SensorStream-1.0-SNAPSHOT.jar**:

```
ls target
```

5. Enter the following command to submit the topology to the Storm cluster:

```
storm jar target/SensorStream-1.0-SNAPSHOT.jar
lex.microsoft.com.SensorTopology sensorstream
```

6. Wait for the topology to be submitted, and then in your browser, navigate to the Storm UI page at the following address (substituting ***cluster*** for your cluster name). If prompted, log in using your cluster HTTP credentials:

   https://***cluster***.azurehdinsight.net/stormui

7. In the **Storm UI** page, under **Topology Summary**, note that the **sensorstream** topology is running. Then click **sensorstream** to view details of the topology.
8. In the **Topology Summary** page, under **Bolts (All time)**, click the **slidingbolt** bolt.
9. In the **Component Summary** page, under Input stats (All time), note the Executed value for each stream received by the bolt - these should include the **__tick** stream from the **__system** component and the **default** stream from the **sensorspout** component (if the **__tick** stream is not listed, wait a few seconds and then refresh the page).
10. Under **Component Summary**, click the **sensorstream** topology to return to the **Topology Summary** page.

11. Under **Topology Actions**, click **Kill**, and if prompted, confirm that you want to stop the topology within 30 seconds.
12. Close the **Storm UI** tab.
13. Exit the SSH console.

Clean Up

Now that you have finished using Storm, you can delete your cluster. This ensures that you avoid being charged for cluster resources when you are not using them. If you are using a trial Azure subscription that includes a limited free credit value, deleting the clusters maximizes your credit and helps to prevent using it all before the free trial period has ended.

Delete the Resource group

1. In the Azure portal, on the **Resource groups** menu, select the resource group you created for your clusters. This resource group contains your clusters and the associated storage account.
2. In the blade for your resource group, click **Delete**. When prompted to confirm the deletion, enter the resource group name and click **Delete**.
3. Wait for a notification that your resource group has been deleted.
4. Close the browser.