

Implementing Real-Time Analysis with Hadoop in Azure HDInsight

Lab 2 - Getting Started with Storm (C#)

Overview

In this lab, you will provision HDInsight Storm and HBase clusters, and use Microsoft C# to create a simple Storm topology that captures a stream of data and stores the captured data in HBase. You will then create a more complex Storm topology that aggregates data as it is captured.

What You'll Need

To complete the labs, you will need the following:

- A web browser
- A Microsoft account
- A Microsoft Azure subscription
- A Microsoft Windows computer containing:
 - Microsoft Visual Studio
 - The lab files for this course

Note: You can only complete this lab on a Windows client computer with Visual Studio 2015 and the latest version of the Microsoft .NET SDK for Azure installed. To set up the required environment for the lab, follow the instructions in the **Setup** document for this course, including the procedure to install Visual Studio and the .NET SDK for Azure.

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.

Create a Storage Account

You will use an Azure storage account to provide the distributed file system for your HDInsight cluster.

1. In a web browser, navigate to <http://portal.azure.com>, and if prompted, sign in using the Microsoft account that is associated with your Azure subscription.
2. In the Microsoft Azure portal, in the Hub Menu (on the left edge of the page), click **New**. Then in the **Data + Storage** menu, click **Storage account**.
3. In the **Storage account** blade, select the **Resource Manager** deployment model, and click **Create**.
4. In the **Create storage account** blade, enter the following settings, and then click **Create**:
 - **Name**: Enter a unique name for your storage account (and make a note of it!)
 - **Type**: Select **Locally Redundant**
 - **Subscription**: Your subscription
 - **Resource group**: Enter a name for a new resource group (and make a note of it!)
 - **Location**: Select any available region
 - **Pin to dashboard**: Unselected
5. At the top of the page, click **Notifications** and verify that deployment has started. Wait until your storage account has been created before proceeding to the next procedure. This should take a few minutes.

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!)
 - **Cluster Type**: Storm
 - **Cluster Operating System**: Windows
 - **HDInsight Version**: Choose the latest version of HDInsight
 - **Subscription**: Select your Azure subscription
 - **Resource Group**: Select the resource group you created for your storage account
 - **Credentials**: Configure the following settings in the **Cluster Credentials** blade, and then click **Select**
 - **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!)
 - **Enable Remote Desktop**: No
 - **Data Source**: The storage account you created in the first procedure
 - **Node Pricing Tiers**: Configure the following settings in the **Node Pricing Tiers** blade, and then click **Select**
 - **Number of Worker nodes**: 1
 - **Supervisor Nodes Pricing Tier**: View all and choose the smallest available size
 - **Nimbus Node Pricing Tier**: View all and choose the smallest available size
 - **Zookeeper Nodes Pricing Tier**: View all and choose the smallest available size
 - **Optional Configuration**: None
 - **Pin to Startboard**: Not selected
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. The free-trial subscription includes a credit limit of approximately \$200 (or local equivalent) that you can spend over a period of 30 days, which is 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 in order to avoid using your Azure credit unnecessarily.

Provision an HDInsight HBase Cluster

In addition to the Storm cluster, you will use an HBase cluster in some later exercises in this lab.

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!)
 - **Cluster Type**: HBase
 - **Cluster Operating System**: Windows
 - **HDInsight Version**: Choose the latest version of HDInsight
 - **Subscription**: Select your Azure subscription
 - **Resource Group**: The resource group you created for the storm cluster in the previous procedure
 - **Credentials**: Configure the following settings in the **Cluster Credentials** blade, and then click **Select**
 - **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!)
 - **Enable Remote Desktop**: Yes
 - **Expires on**: Select a date in the future.
 - **Remote Desktop Username**: Enter another user name of your choice (and make a note of it!)
 - **Remote Desktop Password**: Enter and confirm a strong password (and make a note of it!)
 - **Data Source**: The storage account you created in the first procedure
 - **Node Pricing Tiers**: Configure the following settings in the **Node Pricing Tiers** blade, and then click **Select**
 - **Number of Worker nodes**: 1
 - **Worker Nodes Pricing Tier**: View all and choose the smallest available size
 - **Head Node Pricing Tier**: View all and choose the smallest available size
 - **Zookeeper Nodes Pricing Tier**: View all and choose the smallest available size
 - **Optional Configuration**: None
 - **Pin to Startboard**: Not selected
3. At the top of the page, click **Notifications** and verify that deployment has started. Then, wait for the clusters to be deployed (as previously indicated, this can take a long time – often 30 minutes or more; so take a break – you’ve been working hard!)

Note: Remember - as soon as an HDInsight cluster is running, the credit in your Azure subscription will start to be charged.

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.

Create a Storm Project in Visual Studio


The Azure SDK adds a number of project templates to Visual Studio, including templates for Storm topology projects. You will use one of these templates to create a simple Storm topology.

1. Start Visual Studio and create a new project with the following settings:
 - **Template:** *In the **HDInsight** category, select **Storm Application**.*
 - **Name:** SensorStream
 - **Location:** *Browse to the **Lab02** folder in the directory where you extracted the lab files.*
 - **Solution name:** Storm Sensor Reader
2. When the project opens, view the Solution Explorer pane and note that the project template includes the following code files:
 - **Program.cs:** Contains code to initiate the topology.
 - **Spout.cs:** Contains code to implement a spout that consumes a stream from a data source and adds it to the topology.
 - **Bolt.cs:** Contains code to implement a bolt that consumes the stream from the spout and processes the event data it contains.
3. Open **Spout.cs** if it is not already open, and review the code it contains. Note that this code defines a class named **Spout** that includes the following methods:
 - **Spout:** This is the initializer method for the class, and defines the component schema for the event data. An output schema named **default** is declared, consisting of a list containing a single integer value. Because this class represents a spout that does not consume events from any other spouts in the topology, the input schema is null.
 - **Get:** This method returns a spout associated with the context passed to it. It is used by Storm to create instances of the spout on cluster nodes.
 - **NextTuple:** This method emits tuples containing event data to the topology, where they can be consumed by bolts. The default code in the template emits a list containing a single integer value, which is randomly generated using a C# **Random** class.
 - **Ack:** This method is called to acknowledge that a tuple emitted by the spout and identified by a sequence ID, has been successfully processed. The **Ack** method is used to implement retry logic, and is empty in the default template.
 - **Fail:** This method is called to indicate that a failure occurred when processing a tuple. The **Fail** method is used to implement retry logic, and is empty in the default template.
4. Open **Bolt.cs** if it is not already open, and review the code it contains. Note that this code defines a class named **Bolt** that includes the following methods:
 - **Bolt:** This is the initializer method for the class, and defines the component schema for the event data. The input schema named **default** is declared, consisting of a list containing a single integer value. Because this class represents a bolt that does not emit events for processing by other downstream bolts in the topology, the output schema is null.
 - **Get:** This method returns a bolt associated with the context passed to it. It is used by Storm to create instances of the bolt on cluster nodes.
 - **Execute:** This method is called each time a tuple containing event data is received from the stream, and contains code to process it. The code in the default template retrieves the first value from the tuple, which is assumed to be an integer, and adds it to a running count. The count value is then written to the log for the topology.
5. Open **Program.cs** if it is not already open, and review the code it contains. Note that the class includes a public **GetTopologyBuilder** function that is called by Storm to initiate the topology. This method creates a topology named **SensorStream** plus the current date and time, adds the **Spout** class as a spout, and adds the **Bolt** class as a bolt that consumes the stream from the

spout using a shuffle grouping (which means that the events are spread randomly across nodes on which the bolt is running).

Submit the Topology

The default code in the Storm Application template defines a valid topology that can be submitted to a Storm cluster and started. In this procedure, you will submit the topology to see it running.

1. On the **Build** menu, click **Build Solution**.
2. View the Server Explorer pane, and connect to your Azure subscription if you are not already connected. Then expand Azure, expand HDInsight, and right-click your Storm cluster (which is indicated by a ) and click **Properties**.
3. In the Properties pane, verify that the **Status** of your cluster is **Running**.
4. In the Solution Explorer pane, right-click the **SensorStream** project and click **Submit to Storm on HDInsight**. Then, when prompted, select your Storm cluster and click **Submit**.
5. After the storm topology has been submitted, in the **Storm Topology** window that is automatically opened, select the **SensorStream** topology to view its status. The topology should show the **Spout** spout transferring **default** tuples to the **Bolt** bolt, where they are executed. Click the **Refresh** icon at the top of the page to see an updated count of the number of events processed.
6. In your web browser, in the Azure portal, browse to your Storm HDInsight cluster.
7. At the top of the blade for your Storm cluster, click **Dashboard**, and when prompted, enter the cluster login credentials you specified for your Storm cluster.
8. In the Storm Dashboard, view the **Storm UI** tab, and note that the **Topology summary** list includes a topology named **SensorStream** and the current date and time. Click this topology to view its statistics.
9. Under **Topology actions**, click **Deactivate** to stop the topology. When prompted to confirm, click **OK**. Then close the Storm UI tab, but keep the Azure portal tab open.
10. Switch back to Visual Studio and refresh the Storm Topology view a few times. Now that the topology is inactive, no more events are processed.
11. On the toolbar in the Storm Topology page, click **Kill** to delete the topology; and when prompted to confirm, click **OK**. Then close the Storm Topology window within Visual Studio.
12. Keep the Visual Studio open, you will enhance the **SensorStream** project in the next exercise.

Creating a Custom Storm Topology

In the previous exercise, you created and submitted a Storm topology based on the default template. While this topology ran successfully, it didn't really do anything very useful. In this exercise, you will create a more complex topology that captures simulated sensor readings and stores them in an HBase table.

Create an HBase Table

Before creating a Storm topology, you must create an HBase table into which the captured sensor readings will be written.

Note: The commands in this procedure are case-sensitive.

1. In your web browser, in the Azure portal, browse to your HBase HDInsight cluster.
2. In the blade for your HBase cluster, click **Remote Desktop**.

3. On the **Remote Desktop** blade for your HBase cluster, click **Connect**, and open a remote desktop session to your HDInsight cluster using the remote desktop username and password you specified when provisioning the HBase cluster.
4. When the remote desktop window opens, on the desktop, double-click the **Hadoop Command Line** icon and view the syntax documentation for the Hadoop command line tool.
5. In the Hadoop Command Line console window, enter the following command to start the HBase shell:

```
%HBASE_HOME%\bin\hbase shell
```

6. Enter the following command to create a table named **Sensors** with a column family named **Reading**.

```
create 'Sensors', 'Reading'
```

7. Minimize the remote desktop window (you will return to the HBase shell later.)

Create a Sensor Simulator Class

The sensors in your topology will be simulated by a class that generates random sensor readings. Each reading will consist of the date and time the reading was taken, the name of the sensor, and the value of the reading.

1. In Visual Studio, add a new class file named **Sensor.cs** to the **SensorStream** project.

Tip: You can copy and paste code in the following instructions from **Sensor.txt** in the **Lab02\C#** folder.

2. In the **Sensor.cs** code file, replace the existing **using** statements with the following code:

```
using System;
using Microsoft.SCP;
```

3. Modify the **Sensor** class definition to be static, as shown here:

```
public static class Sensor
{
}
```

4. In the **Sensor** class, add the following code to define a static method named **GetSensorReading**:

```
public static Values GetSensorReading()
{
    Random r = new Random(DateTime.Now.Millisecond);
    var reading = new Values(DateTime.UtcNow,
        "Sensor " + r.Next(10).ToString(),
        r.Next());
    return reading;
}
```

This code initializes a new **Random** class, and then generates a **Values** list containing:

- The current date and time.
 - A string in the format *SensorN*, where N is a random number between 0 and 9.
 - A random integer value.
5. Save the modified **Sensor.cs** code file.

Modify the Spout Class

Now that you have created a source for the sensor readings, you must modify the spout to consume them.

Tip: You can copy and paste code in the following instructions from **Spout.txt** in the **Lab02\C#** folder.

1. Open the **Spout.cs** code file, and in the **Spout** initializer method, find the following code:

```
outputSchema.Add("default", new List<Type>() { typeof(int) });
```

2. Replace this code with the following code, which defines the output schema as a list containing a date, a string, and an integer:

```
outputSchema.Add("default", new List<Type>() { typeof(DateTime),  
                                                typeof(string),  
                                                typeof(int) });
```

3. In the **NextTuple** method, find the following code:

```
ctx.Emit(new Values(r.Next()));
```

4. Replace this code with the following code, which emits a sensor reading to the default stream:

```
Values sensorReading = Sensor.GetSensorReading();  
ctx.Emit(Constants.DEFAULT_STREAM_ID, sensorReading);
```

5. Above the **Spout** method, delete the following variable declaration, which is no longer required:

```
private Random r = new Random();
```

6. Save the modified **Spout.cs** code file.

Modify the Bolt Class

Now you're ready to modify the **Bolt** class that will process the sensor readings by writing them to the HBase table.

Tip: You can copy and paste code in the following instructions from **Bolt.txt** in the **Lab02\C#** folder.

1. On the **Tools** menu, point to **NuGet Package Manager**, and click **Manage NuGet Packages for Solution**.
2. In the NuGet Package Manager window, search nuget.org for *HBase Client*, and in the list of results, select **Microsoft.HBase.Client**.
3. Install the latest stable version of the HBase Client package in the **SensorStream** project. Then close the NuGet Package Manager window.
7. In the **Bolt.cs** code file, add the following **using** statements under the existing **using** statements:

```
using Microsoft.HBase.Client;  
using org.apache.hadoop.hbase.rest.protobuf.generated;
```

8. At the beginning of the class, find the following variable declarations:

```
private int count;  
private Context ctx;
```

9. Replace these declarations with the following ones, substituting the name of your HDInsight HBase cluster, your HBase cluster user name, and your HBase cluster password where indicated:

```
private Context ctx;
private DateTime readingTime;
private string sensorName;
private int sensorValue;
string clusterURL = "https://clustername.azurehdinsight.net";
string hadoopUsername = "username";
string hadoopUserPassword = "password";
```

10. In the **Bolt** initializer method, find the following code:

```
inputSchema.Add("default", new List<Type>() { typeof(int) });
```

11. Replace this code with the following code, which defines the input schema as a list containing a date, a string, and an integer:

```
inputSchema.Add("default", new List<Type>() { typeof(DateTime),
                                              typeof(string),
                                              typeof(int) });
```

12. In the **Execute** method, replace all of the existing code with the following code, which reads the data from the tuple received by the bolt and writes it to the HBase table using the sensor name as the row key:

```
// Get data from tuple
readingTime = (DateTime)tuple.GetValue(0);
sensorName = tuple.GetString(1);
sensorValue = tuple.GetInteger(2);

// log tuple
Context.Logger.Info("Sensor Reading", sensorName);

try
{
    // Insert live data into an HBase table.
    ClusterCredentials creds = new ClusterCredentials(
                                new Uri(clusterURL),
                                hadoopUsername,
                                hadoopUserPassword);
    HBaseClient hbaseClient = new HBaseClient(creds);

    CellSet cellSet = new CellSet();
    CellSet.Row cellSetRow = new CellSet.Row
    { key = Encoding.UTF8.GetBytes(sensorName) };
    cellSet.rows.Add(cellSetRow);
    Cell valueCell = new Cell
    { column = Encoding.UTF8.GetBytes("Reading:Value"),
      data = Encoding.UTF8.GetBytes(sensorValue.ToString()) };
    cellSetRow.values.Add(valueCell);
    Cell timeCell = new Cell
```



```

        { column = Encoding.UTF8.GetBytes("Reading:LastUpdated"),
          data = Encoding.UTF8.GetBytes(readingTime.ToString()) };
        cellSetRow.values.Add(timeCell);
        hbaseClient.StoreCells("Sensors", cellSet);
    }
    catch (Exception ex)
    {
        Context.Logger.Error("Sensor Write Error", ex.Message);
    }
}

```

13. Save the modified **Bolt.cs** code file.

Note: The connection between the Storm and HBase clusters in this lab is made over the Internet, even though both clusters are in the same Azure subscription and data center. This keeps the lab setup simple, and enables you to focus on learning the specifics of implementing a Storm topology in C#. In a real, production solution, you would provision both the Storm and HBase clusters in a virtual network so they can communicate directly without passing traffic out of the Azure data center across the Internet. For an example of deploying an HBase cluster in a virtual network, see <https://azure.microsoft.com/en-us/documentation/articles/hdinsight-hbase-provision-vnet/>.

Additionally, the code in this lab has been deliberately minimized to help you focus on learning the key principles of creating a Storm topology. In production code, you should store the cluster name and credentials in a configuration file and read them at run-time instead of hard-coding them. For a more comprehensive example of a Storm topology that writes data to HBase using best practices, create a new Visual Studio project based on the **Storm HBase Writer Sample** template.

Modify the Topology Initialization Code

Now that you have made changes to the event tuple schema, you must modify the code used to initiate the topology.

Tip: You can copy and paste code in the following instructions from **Program.txt** in the **Lab02\C#** folder.

1. In the **Program.cs** code file, modify the contents of the **GetTopologyBuilder** function to match the following code:

```

TopologyBuilder topologyBuilder = new
    TopologyBuilder("SensorStream");
topologyBuilder.SetSpout(
    "SensorReader_Spout",
    Spout.Get,
    new Dictionary<string, List<string>>()
    {
        { Constants.DEFAULT_STREAM_ID,
          new List<string>() {"readingTime", "sensorName", "sensorValue"} }
    },
    1);
topologyBuilder.SetBolt(
    "SensorWriter_Bolt",
    Bolt.Get,
    new Dictionary<string, List<string>>(),
    4).shuffleGrouping("SensorReader_Spout");

```

```
return topologyBuilder;
```

This code renames the spout and bolt to be more descriptive, modifies the stream schema to match the sensor reading data, and configures the parallelism hint for the bolt so that 4 executors are created.

2. Save the modified **Program.cs** file.

Submit the Topology

The default code in the Storm Application template defines a valid topology that can be submitted to a Storm cluster and started. In this procedure, you will submit the topology to see it running.

1. On the **Build** menu, click **Build Solution**.
2. In the Solution Explorer pane, right-click the **SensorStream** project and click **Submit to Storm on HDInsight**. If you are prompted to sign into your Azure account, do so using your Microsoft account credentials. Then, when prompted, select your Storm cluster and click **Submit**.
3. After the storm topology has been submitted, in the **Storm Topology** window that is automatically opened, select the **SensorStream** topology to view its status. The topology should show the **SensorReader_Spout** spout transferring **default** tuples to the **SensorWriter_Bolt** bolt, where they are executed. Click the **Refresh** icon at the top of the page to see an updated count of the number of events processed. Note that the spout runs one task, but the bolt uses four tasks to process the stream in parallel.
4. Refresh the view a few times until over a thousand events have been processed.

Query the HBase Table

Now that your Storm topology has captured and processed some sensor readings, you can view the readings in the HBase table.

1. In the remote desktop window, in the HBase shell, enter the following command to retrieve the data in the Sensors table

```
scan 'Sensors'
```
2. View the results, noting that they show the most recently captured value for each sensor.
3. Scan the table again, and observe that the values are being updated in real-time by the Storm topology.
5. Switch to Visual Studio and on the toolbar in the Storm Topology page, click **Kill** to stop and delete the topology; and when prompted to confirm, click **OK**.
4. Switch back to the remote desktop and scan the table once more. The data is now static.
5. Minimize the command line window. You will return to the HBase shell in the next exercise.

Implement Retry Logic

In the previous exercise, you created and submitted a Storm topology that captures simulated sensor readings and stores them in an HBase table. However, the topology is not transactional and has no retry logic; and in the event of a bolt being unable to write a sensor reading to HBase (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.

Enable the Acker

In order to implement retry logic in a non-transactional topology, you must enable the Storm *Acker* for the topology. This configuration setting enables downstream bolts to call the **Ack** method when a tuple has been processed successfully. Conversely, if processing fails for any reason, the downstream bolt can call the **Fail** method. The Acker routes these calls to the upstream spout or bolt that emitted the tuple to verify that a tuple has been processed or to initiate retry logic.

Tip: You can copy and paste code in the following instructions from **Program (Ack).txt** in the **Lab02\C#** folder.

1. In Visual Studio, in the **SensorStream** project, view the **Program.cs** file.
2. Modify the code in the **GetTopologyBuilder** function by adding an **enableAck** parameter with the value true to the **SetSpout** and **SetBolt** method calls as shown by the code in **bold** below.

```
public ITopologyBuilder GetTopologyBuilder()
{
    TopologyBuilder topologyBuilder = new TopologyBuilder("SensorStream");

    topologyBuilder.SetSpout(
        "SensorReader_Spout",
        Spout.Get,
        new Dictionary<string, List<string>>()
        {
            { Constants.DEFAULT_STREAM_ID,
              new List<string>() {"readingTime", "sensorName", "sensorValue"} }
        },
        1, true);

    topologyBuilder.SetBolt(
        "SensorWriter_Bolt",
        Bolt.Get,
        new Dictionary<string, List<string>>(),
        4, true).shuffleGrouping("SensorReader_Spout");

    return topologyBuilder;
}
```

3. Save Program.cs.

Modify the Bolt to Call *Ack* or *Fail* if Ack is Enabled

To initiate retry logic, the bolt in your topology must determine if the Acker is enabled; and if so, it must call **Ack** for each successfully processed tuple, or **Fail** if an exception occurs.

Tip: You can copy and paste code in the following instructions from **Bolt (Ack).txt** in the **Lab02\C#** folder.

1. Open the **Bolt.cs** file, and in the **Bolt** class declaration, under the existing variable declarations for the HBase cluster URL and credentials, add the following variable declaration:

```
private bool enableAck = false;
```

2. In the public **Bolt** initializer function, add the following code under the existing code that declares the component schema to set the **enableAck** variable based on the bolt configuration:

```
if (Context.Config.pluginConf.ContainsKey(
```

```

        Constants.NONTRANSACTIONAL_ENABLE_ACK))
    {
        enableAck = (bool) (Context.Config.pluginConf
            [Constants.NONTRANSACTIONAL_ENABLE_ACK]);
    }

```

3. In the **Execute** function, at the end of the **try** block, after the existing code that writes the data to HBase, add the following code to call **Ack** when the tuple has been successfully processed.

```

if (enableAck)
{
    ctx.Ack(tuple);
}

```

4. In the **Execute** function, in the **catch** block, after the existing code to log the error message, add the following code to call **Fail** if the tuple is not processed successfully.

```

if (enableAck)
{
    ctx.Fail(tuple);
}

```

5. Save Bolt.cs.

Modify the Spout to Add Retry Logic

The code to actually implement retry logic in your topology must be added to the spout. The basic approach is to maintain a cache of tuples that have been emitted until an Ack for those tuples is received. Each tuple emitted is identified by a sequence number so that tuples that have been successfully processed by all downstream bolts can be removed from the cache. If processing for a tuple fails, the spout retrieves the tuple from the cache and re-emits it.

Tip: You can copy and paste code in the following instructions from **Spout (Ack).txt** in the **Lab02\C#** folder.

1. Open the Spout.cs file, and in the **Spout** class declaration, under the existing variable declaration for the context, add the following variable declarations:

```

private bool enableAck = false;
private long tupSeqId = 0;
Dictionary<long, Values> cachedTuples = new Dictionary<long, Values>();

```

6. In the public **Spout** initializer function, add the following code under the existing code that declares the component schema to set the **enableAck** variable based on the spout configuration:

```

if (Context.Config.pluginConf.ContainsKey(
    Constants.NONTRANSACTIONAL_ENABLE_ACK))
{
    enableAck = (bool) (Context.Config.pluginConf
        [Constants.NONTRANSACTIONAL_ENABLE_ACK]);
}

```

2. In the **NextTuple** function, replace the existing code with the following code. This checks to see if the Acker is enabled; and if so, increments the sequence ID and includes it with the emitted tuple.

```

Values sensorReading = Sensor.GetSensorReading();

```

```

if (enableAck)
{
    // If ack is enabled, include a sequence ID
    tupSeqId++;
    cachedTuples.Add(tupSeqId, sensorReading);
    ctx.Emit(Constants.DEFAULT_STREAM_ID, sensorReading, tupSeqId);
}
else
{
    ctx.Emit(Constants.DEFAULT_STREAM_ID, sensorReading);
}

```

3. In the **Ack** function, add the following code to remove the cached tuple after it has been successfully processed:

```

// remove cached tuple
if (enableAck)
{
    cachedTuples.Remove(seqId);
}

```

4. In the **Fail** function, add the following code to retrieve the cached tuple and re-emit it if processing fails:

```

// retry failed tuple
if (cachedTuples.ContainsKey(seqId))
{
    ctx.Emit(Constants.DEFAULT_STREAM_ID, cachedTuples[seqId], seqId);
}

```

5. Save Spout.cs.

Test the Topology

1. In the Solution Explorer pane, right-click the **SensorStream** project and click **Submit to Storm on HDInsight**. If you are prompted to sign into your Azure account, do so using your Microsoft account credentials. Then, when prompted, select your Storm cluster and click **Submit**.
2. After the storm topology has been submitted, in the Storm Topology window that is automatically opened, select the **SensorStream** topology to view its status. Click the **Refresh** icon at the top of the page to see an updated count of the number of events processed.
3. Select the **Show Acker** checkbox, and refresh the topology, and note the **Ack** calls as the bolt processes the tuples (you may need to refresh several times before they appear).
4. Click **Kill** to stop and delete the topology; and when prompted to confirm, click **OK**.
5. Keep Visual Studio open, you will return to the SensorStream project in the next exercise.

Implement a Sliding Window

A common requirement in streaming data analysis is to aggregate data over a temporal period. For example, you may need to build an application that tracks the top ten trending hashtags over the previous hour. You can build this kind of solution by implementing a sliding window pattern in which the data within the required timeframe is aggregated at periodic intervals (for example, by determining the most commonly used hashtags for the last every hour at ten minute intervals).

In this exercise, you will implement a simple sliding window topology to count the number of readings generated by each sensor in the past minute. The counts will be updated every ten seconds.

Create a Bolt to Count Sensor Readings

Tip: You can copy and paste code in the following instructions from **Counter_Bolt.txt** in the **Lab02\C#** folder.

1. In Visual Studio, in Solution Explorer, right-click the **SensorStream** project, point to **Add**, and click **New Item**. Then add a new **Storm Bolt** named **Counter_Bolt.cs**.
2. In the new **Counter_Bolt.cs** class file, replace the existing **using** statements with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.SCP;
using Microsoft.HBase.Client;
using org.apache.hadoop.hbase.rest.protobuf.generated;
```

3. In the **Counter_Bolt** class declaration, before the **Execute** function, add the following variable declarations, substituting the name of your HDInsight HBase cluster, your HBase cluster user name, and your HBase cluster password where indicated:

```
// tuple fields
private DateTime readingTime;
private string sensorName;

// HBase settings
string clusterURL = "https://clustername.azurehdinsight.net";
string hadoopUsername = "username";
string hadoopUserPassword = "password";

// Cached readings
Dictionary<string, List<DateTime>> sensors =
    new Dictionary<string, List<DateTime>>();
```

This code includes a dictionary that will be used to store an entry for each sensor, containing the sensor name as a key and a list of DateTime values that indicate the reading time for each sensor reading.

4. In the **Counter_Bolt** class, before the **Execute** function, add the following initializer function:

```
public Counter_Bolt(Context ctx)
{
    this.ctx = ctx;
    Dictionary<string, List<Type>> inputSchema
        = new Dictionary<string, List<Type>>();
    inputSchema.Add("default", new List<Type>()
        { typeof(DateTime), typeof(string), typeof(int) });
    inputSchema.Add(Constants.SYSTEM_TICK_STREAM_ID, new List<Type>()
        { typeof(long) });
    this.ctx.DeclareComponentSchema(new ComponentStreamSchema
        (inputSchema, null));
}
```

This code defines two input schemas for the bolt. A default input schema that contains the sensor reading tuples emitted by the spout, and an input schema for the *tick tuple* stream that is sent to the bolt at a specified interval by Storm.

5. In the **Counter_Bolt** class, after the initializer function, add the following static function to return an instance of the bolt when initializing the topology.

```
public static Counter_Bolt Get(Context ctx,
                               Dictionary<string, Object> parms)
{
    return new Counter_Bolt(ctx);
}
```

6. In the **Counter_Bolt** class, after the **Get** function, add the following static function to determine whether a cached reading is stale (readings are considered stale if they are older than a minute).

```
private static bool StaleReading(DateTime d)
{
    return d < DateTime.Now.Subtract(new TimeSpan(0, 1, 0));
}
```

7. In the **Counter_Bolt** class, in the **Execute** function, add the following code.

```
var isTickTuple =
    tuple.GetSourceStreamId().Equals(Constants.SYSTEM_TICK_STREAM_ID);
if (!isTickTuple)
{
    // Get data from tuple
    readingTime = (DateTime)tuple.GetValue(0);
    sensorName = tuple.GetString(1);

    // cache this reading
    if (sensors.ContainsKey(sensorName))
    {
        sensors[sensorName].Add(readingTime);
    }
    else
    {
        sensors.Add(sensorName, new List<DateTime>() { readingTime });
    }
}
else
{
    // process each sensor for which we have readings
    foreach (KeyValuePair<string, List<DateTime>> readings in sensors)
    {
        // remove readings over a minute old
        readings.Value.RemoveAll(StaleReading);

        // count remaining readings
        int numReadings = readings.Value.Count();

        // write reading count to HBase
        ClusterCredentials creds = new ClusterCredentials(
            new Uri(clusterURL),
            hadoopUsername,
```

```

                                hadoopUserPassword);
HBaseClient hbaseClient = new HBaseClient(creds);

CellSet cellSet = new CellSet();
CellSet.Row cellSetRow =
    new CellSet.Row { key = Encoding.UTF8.GetBytes(readings.Key) };
cellSet.rows.Add(cellSetRow);
Cell valueCell =
    new Cell { column = Encoding.UTF8.GetBytes("Reading:Count"),
              data = Encoding.UTF8.GetBytes(numReadings.ToString()) };
cellSetRow.values.Add(valueCell);
hbaseClient.StoreCells("Sensors", cellSet);
    }
}

```

This code checks the input stream to determine if a tick tuple has been received. If not, the code processes a default tuple by adding the sensor reading time to the cached list of reading times for the specified sensor name.

If a tick tuple has been received, the code processes each sensor by removing all readings that are over a minute old, counting the remaining readings, and writing the count value to a column named **Reading:Count** in the **Sensors** HBase table.

8. Save Counter_Bolt.cs.

Add the Counter Bolt to the Topology

Now that you have implemented a bolt to count readings, you must add it to the topology.

Tip: You can copy and paste code in the following instructions from **Program (Counter).txt** in the **Lab02\C#** folder.

1. Open **Program.cs**, and in the **GetTopologyBuilder** function, after the existing **topologyBuilder.SetBolt** statement but before the **return topologyBuilder** statement, add the following code to define a configuration setting that enables a tick tuple every 10 seconds:

```

var boltConfig = new StormConfig();
boltConfig.Set("topology.tick.tuple.freq.secs", "10");

```

2. After the code you just added, but before the **return topologyBuilder** statement, add the following code to include the counter bolt in the topology.

```

topologyBuilder.SetBolt(
    "Counter_Bolt",
    Counter_Bolt.Get,
    new Dictionary<string, List<string>>(),
    1).fieldsGrouping("SensorReader_Spout",
    new List<int>() { 1 }).addConfigurations(boltConfig);

```

This code adds the **Counter_Bolt** class as a bolt, connecting it to the **Sensor_Reader** spout using a **fieldGrouping** connection so that all readings for a given **sensorName** field value are sent to the same instance of the bolt. This is necessary to ensure that there is a single, definitive count of readings for each sensor without having to collate results from multiple bolt instances. The configuration for the tick tuple is also added to this bolt.

3. Save Program.cs.

Test the Sliding Window

1. On the **Build** menu, click **Build Solution**.
2. In the Solution Explorer pane, right-click the **SensorStream** project and click **Submit to Storm on HDInsight**. If you are prompted to sign into your Azure account, do so using your Microsoft account credentials. Then, when prompted, select your Storm cluster and click **Submit**.
3. After the storm topology has been submitted, in the **Storm Topology** window that is automatically opened, select the **SensorStream** topology to view its status. The topology should show the **SensorReader_Spout** spout transferring **default** tuples to the **SensorWriter_Bolt** bolt, where they are executed. Click the **Refresh** icon at the top of the page to see an updated count of the number of events processed.
4. Refresh the view a few times until over a thousand events have been processed.
6. In the remote desktop window, in the HBase shell, enter the following command to retrieve the data in the Sensors table

```
scan 'Sensors'
```

7. View the results, noting that they now include a **Readings:Count** value for each sensor.
8. Scan the table again, and observe that the values are being updated every ten seconds by the Storm topology.
5. Enter **quit** to exit the HBase shell, close the Hadoop Command Line window, and sign out of the remote desktop session.
6. In Visual Studio, on the toolbar in the Storm Topology page, click **Kill** to stop and delete the topology; and when prompted to confirm, click **OK**.
9. Close Visual Studio.

Clean Up

Now that you have finished using Storm and HBase, you can delete your clusters. 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.