Lab8: Getting Familiar with Infosphere Streams, Streams Studio and Streams Processing Language (SPL)

Department of Electrical and Computer Engineering
Iowa State University
Spring 2015

Purpose

Data Streams are continuously flowing data, e.g sensor data or network packets. We will use Virtual Machine for real time analysis of streams, i.e. analysis of streams as and when they arrive, without storing them in the system. The goal of this lab is to get familiar with writing, compiling and running a program using Streams Processing Language (SPL) on IBM Infosphere Streams, a platform for data streams analysis. More information can be found in the following link:
http://pic.dhe.ibm.com/infocenter/streams/v3r0/index.jsp

Submission

Create a zip (or tar) archive with the following and hand it in through blackboard.

- Commented Code for your programs. Include all source files needed for compilation.
- Output of your programs.

Using the VMWare player

A Virtual Machine, or a VM, is a software-based emulation of a computer, which runs an OS and applications just like a physical machine, except that there is no physical machine. A VMWare player is an application that allows you to create and run virtual machines (VM) on the systems where the VMWare player player is installed. IBM Infosphere Streams can only run on Red Hat Linux (RHEL), so VMWare player, along with the VM, gives us the freedom of using Infosphere Streams irrespective of any Operating System used. We have created a VM Image with RHEL OS and Infosphere Streams installed in the OS. In the lab computers, the VM Image is located in C drive, in a folder named 'VMImage'. In order to run the VM, double click the file with .vmx format. Wait for the system to boot up and you have an RHEL system ready for you to use. You will be logged in as user "streamsadmin".

Once you have logged into the VM, you will find many icons on the desktop of the VM. Explore and use these to your advantage.

Understanding IBM Infosphere Streams and SPL

Infosphere Streams is a software platform that enables analysis and processing of massive data streams. Users can create applications using SPL without having to understand the lower-level stream-specific operations.

SPL has the following main components:
a) Data Streams / Tuples

b) Operators: act on the data from incoming stream to produce the desired output streams. These streams are either the final output stream or an input to some other operator. There are existing standard operators. However, new operators can be created using either SPL or other programming languages such as C++/Java.

c) Processing Elements (PE): Operators and their relationships are broken into smaller units called PE, which are then executed independently.

d) Jobs: To run an SPL application, the corresponding jobs are submitted to Infosphere Streams after compilation of the SPL application.

Getting Started with SPL – Compiling and Understanding the basics

The following example application illustrates basic features of SPL :

The application reads a text file and outputs the same text file with each line prefixed with the corresponding line number.

For instance, if the input to the program is the following text,

```
The Unix utility "cat" is so called
because it can con"cat"enate files.
Our program behaves like "cat -n",
listing one file and numbering lines.
```

then, the expected output is as follows:

```
1 The Unix utility "cat" is so called
2 because it can con"cat"enate files.
3 Our program behaves like "cat -n",
4 listing one file and numbering lines.
```

The SPL code for the above example is as follows :

```
composite NumberedCat {                                                  //1
 graph                                                                   //2
  stream<rstring contents> Lines = FileSource() {                        //3
   param  format        : line;                                          //4
      file          : getSubmissionTimeValue("file");                    //5
  }                                                                      //6
  stream<rstring contents> Numbered = Functor(Lines) {                   //7
   logic  state         : { mutable int32 i = 0; }                       //8
      onTuple Lines : { i++; }                                           //9
   output Numbered      : contents = (rstring)i + " " + contents;        //10
  }                                                                      //11
  () as Sink = FileSink(Numbered) {                                      //12
   param  file          : "result.txt";                                  //13
      format        : line;                                              //14
  }                                                                      //15
}                                                                        //16
```

Understanding the Code

The above code has a main composite operator. A main composite operator is an operator that contains stream graph comprising of operators and streams connecting the operators.

There are 3 operators in the graph: **FileSource()**, **Functor()**, **FileSink().** *FileSource()* operator reads data from a file in the form of stream. Line 4 indicates that the file is read by *FileSource()* operator one line at a time. Each line of the file is a tuple. The input file name is given during runtime. Line 3 indicates that this operator produces an output stream called *Lines* having only one attribute (or field), named *contents*, which of type rstring. For instance, "*The Unix utility "cat" is so called* " is a tuple and the entire line is assigned to the variable *contents* which is of type rstring.

The output of *FileSource()* is the stream *Lines* which is an input to the operator *Functor()* at line 7. This operator initializes a variable "i" and increments it's value for each line (or tuple) received. It concatenates the value of "i" with the remaining line such that the output stream "Numbered" contains each line of the file along with the line number as shown in the output. Please note, that the variable "i" has to be defined as "mutable" else incrementing "i" will not be possible.

The output stream of the *Functor()* operator, *Numbered*, is taken as an input to the operator *FileSink()* which then outputs the result, again one line at a time in a file called "result.txt".

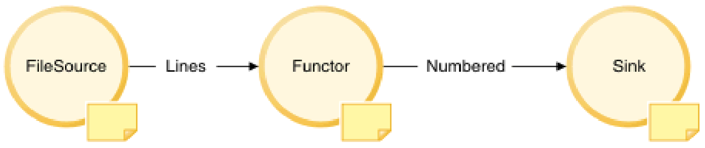
SPL provided standard operators are given in the following link:

http://pic.dhe.ibm.com/infocenter/streams/v3r2/index.jsp?topic=%2Fcom.ibm.swg.im.infosphere.streams.spl-standard-toolkit-reference.doc%2Fdoc%2Fsplstdtlkt-container.html

To go through some more examples of SPL, check the below link. The description of these sample programs are given in the file named `Streams Samples` located in your VM desktop.
https://www.ibm.com/developerworks/mydeveloperworks/files/app?lang=en#/person/120000AMMH/file/ae7c2619-646b-45f0-98d1-ee0bc80d1a8f

Streams Dataflow Graph

The dataflow in the above program can be visualized as the following graph:



Generating SPL program:

You can write, compile and run an SPL program either by using the Linux console, or by using an Eclipse-based IDE, Streams Studio. Streams Studio gives two options to write and edit SPL codes – SPL Graphical Editor and SPL Editor. The SPL Graphical Editor provides a graphical representation of the SPL source code and provides a graphical user interface to create SPL applications. You drag elements from the palette onto the canvas to create SPL applications in the SPL Graphical Editor. On the other hand, the SPL Editor provides a textual representation of the SPL source code. You type SPL code in the SPL Editor to create SPL applications, and the corresponding dataflow graph, as one shown above, is created, The SPL Graphical Editor is the default editor for editing SPL source files. Use the Open With option to change the default editor for the SPL source files. We recommend you to use the IDE for writing and compiling applications.

1) Using the SPL Graphical Editor in Streams Studio

a) Creating SPL application project

1. When you open the Streams Studio, you will be asked to select the location of your workspace.
2. Once you enter the Streams Studio, from the toolbar, select File > New > Project.
3. Select InfoSphere Streams Studio > SPL Application Project.
4. Click Next.
5. In the SPL Application Project window, do the following:
    a. In the Project name field, enter the name of the application project. Name it “Numberedcat”.
    b. By default, the Use default location checkbox is selected and Streams Studio creates the SPL project in the current Eclipse workspace.
    c. The Namespace field contains a default value. Either accept the default or enter a different namespace.
    d. The Main composite field defaults to the name you entered in the Project name field. Either accept the default name or enter a different name for the main composite.
    e. By default, the wizard creates an SPL source (*.spl) file.
    f. Click Finish to accept the default project configuration.

Result: Studio creates a new SPL project named NumberedCat. A main composite graph named NumberedCat is created and opened within the SPL Graphical Editor. The right side of the editor is called the canvas, which is the area where you draw the graph. The left side of the editor is called the palette which contains the items that you can drag onto the canvas to create a graph. The default graph contains a single main composite operator as shown on the canvas.

b) Creating Operators

We refer to the dataflow graph shown above to create the operators.

1. Type fil in the Find field at the top of the palette.
2. Select the FileSource operator from the palette, then click and drag the mouse pointer onto the NumberedCat composite in the canvas.
3. Similarly find the Functor and the FileSink operator and drag them onto the NumberedCat composite in the canvas.

c) Creating streams to connect the operators

Click the output port of the FileSource operator and drag the mouse pointer to the input port of the Functor operator and click. A stream is created that connects the FileSource operator to the Functor operator. Repeat the process starting with the output port of the Functor operator and connecting it to the input port of the FileSink operator.

Hint: To optimize how the graph is drawn, click the Layout button on the toolbar of the editor.

d) Define the parameters for the operators

1. Double click the FileSource operator.
2. In the Properties view, select the *Param* tab. Click Add, select *file* and *format*, and then click OK.
3. In the Value field for the file parameter, type *getSubmissionTimeValue("file")* (You can directly enter the file name instead of entering it at submission time by including the file name in double quotes). In the Value field for the *format* parameter, type *line*.
4. Click anywhere in the table to exit edit mode.
5. Similarly, for the FileSink Operator, select the Param tab, and add file and format. Type *"result.txt"*, including the quotation marks, in the Value field for the file parameter and type *line* in the Value field for the format parameter.

Hint: You will be able to see the corresponding code if you bring your mouse over the operators.

e) Define the schema for the streams

1. Select the stream that connects the FileSource and Functor operators.
2. In the Properties view, select the General tab.
3. In the Name field, click Rename, type *Lines* and then click OK.
4. This will rename the stream. You might see an error message because the rename operation compiles the file. You will not get this error after you complete defining the stream schema.
5. Select the Schema tab.
6. In the Name field, replace varName with *contents*.
7. In the Type field, enter *rstring*.
8. Click anywhere in the table to exit edit mode.
9. Similarly, for the stream connecting from Functor to FileSink, rename the stream to *Numbered*. In the Schema tab, replace varName with *contents* and varType with *rstring*.

f) Define Logic and Output clause for Functor operator

1. Double click the Functor operator. In the Properties view, select the Logic tab. Enter the following code snippet:
   ```
   state      : { mutable int32 i = 0; }
   onTuple Lines : { i++; }
   ```
2. In the Output tab, enter `(rstring)i + " " + contents` in the Value field of *contents.*

g) Save the graph to generate the SPL code.

a. From the menu bar, click **File > Save** to save the contents of the application/NumberedCat.spl file.

h) Open the SPL file to see the generated code

a. In the Streams Graphical Editor, right-click and click **Open With > SPL Editor**.

Result: Studio opens the SPL file in the SPL Editor.

i) Testing the stream application built from graph. An SPL application can run as a stand-alone program or in a distributed cluster as a job (The above code is a simple one so running it as a standalone application is not a bad idea. But, applications requiring more memory and resources can be dealt with by distributing the load amongst more than one machine in a cluster. In this lab, we are not working with cluster, however, to get a feel of it, we still learn to run the application as a job in a cluster of one machine, that is the VM that you are currently working on.)

To test our first SPL application in stand-alone mode:

1. Create the input source file.
   a. In the Project Explorer view, expand **NumberedCat > Resources**, select **data**, right-click and click **New** > **File**.
   b. In the **File name** field, type `cat.txt` and click **Finish**.
   c. Copy and paste the following sample data into the new file, then save and close the file.
   ```
   The Unix utility "cat" is so called
   because it can con"cat"enate files.
   Our program behaves like "cat -n",
   listing one file and numbering lines.
   ```
2. Create a stand-alone build configuration.
   a. In the Project Explorer view, expand **NumberedCat > application**, select **Numberedcat [Build: Distributed]**, right-click and click **New > Standalone Build**.
   b. In the Main – Standalone window, keep the default settings and click **OK**.
   c. Select **Standalone**, right-click and click **Build**.
3. Run the stand-alone build configuration.
   a. In the Project Explorer view, expand **NumberedCat > application** > **NumberedCat** > **Standalone**, right-click and click **Launch**.
   b. In the Edit Configuration window, in the Submission Time Values section, enter `cat.txt` under the Value tab. Click **Continue**.
   c. Click **Yes** when you are prompted to save the changes.

   Result: The output from the NumberedCat program is displayed in the Console view.

To test our first SPL application in distributed mode:

1. Create the input source file. (this step must have been done if you ran the application in standalone mode)
   a. In the Project Explorer view, expand **NumberedCat** > **Resources**, select **data**, right-click and click **New** > **File**.
   b. In the **File name** field, type `cat.text` and click **Finish**.
   c. Copy and paste the following sample data into the new file, then save and close the file.
   ```
   The Unix utility "cat" is so called
   because it can con"cat"enate files.
   Our program behaves like "cat -n",
   listing one file and numbering lines.
   ```
2. Create an instance or use the existing one.
   a. To create an instance, in the Streams Explorer view, expand **InfoSphere Streams**, select **Instances**, right-click and click **Make Instance**.
3. Start the default instance.
   a. In the Streams Explorer view, expand **Instances**, select the instance you want to start, right-click and click **Start Instance**. Say you have started instance named `test@streamsadmin`
4. Create a distributed build configuration.
   a. In the Project Explorer view, expand **NumberedCat > application** > **NumberedCat**, select **Distributed**, right-click and click **Set active**.
   b. Select **Distributed [Active]**, right-click and click **Build**.
5. Run the distributed build configuration.
   a. In the Project Explorer view, expand **NumberedCat > application** > **NumberedCat [Build: Distributed]**.
   b. Select **Distributed [Active]**, right-click and click **Launch**.
   c. In the Edit Configuration window, in the Submission Time Values section, enter `cat.txt` under the Value tab. Click the **Browse** button next to the **Instance** field, select `test@streamsadmin`, click **OK**, and then click **Continue**.
   d. Click **Yes** when you are prompted to save the changes.

   Result: The output from the NumberedCat program is sent to the PE console log. To see the output, in the Streams Explorer view, select the PE you want to monitor, right-click and click **Show PE Console**.

Related link:
http://pic.dhe.ibm.com/infocenter/streams/v3r0/topic/com.ibm.swg.im.infosphere.streams.studio.doc/concepts/creatingasketchfilebyusinginfospherestreamsstudio.html
http://pic.dhe.ibm.com/infocenter/streams/v3r0/nav/5

2) Using the Linux console:

<u>Compiling the Above Code:</u>

a) Create a folder "<ISU Username>/Lab1/NumberedCat" under the home folder and enter the folder.

```
$ cd ~
$ mkdir <ISU Username>
$ cd <ISU Username>
$ mkdir Lab1
$ cd Lab1
$ mkdir NumberedCat
$ cd NumberedCat
```

b) Copy the above code in a file, "NumberedCat.spl", and save it in the *NumberedCat* folder.

c) The code can be run in a standalone mode or as a job in a distributed mode.
To Run the following command to compile the code in a standalone mode:

```
$ sc -T -M NumberedCat
```

The "-T" option runs the above code in a single machine as a standalone process. The "-M NumberedCat" indicates that the main composite operator containing the main program and the stream graph is named "NumberedCat".
(We will discuss about running the program as a job later in this section.)

d) The input to SPL programs should be placed under the subfolder "data" which is generated after compiling the code. Create a file "cat.txt" in the *data/* subfolder and add the following text:

```
The Unix utility "cat" is so called
because it can con"cat"enate files.
Our program behaves like "cat -n",
listing one file and numbering lines.
```

e) If you did not get any compilation error, then an executable "standalone" is created in */output/bin* folder.
Run the executable from the current directory "NumberedCat":

```
$ ./output/bin/standalone file="cat.txt"
```

f) Output files are generated in the "data" subfolder. Check the output in "~/Lab1/NumberedCat/data/results.txt".
Expected output is:

```
1 The Unix utility "cat" is so called
2 because it can con"cat"enate files.
3 Our program behaves like "cat -n",
4 listing one file and numbering lines.
```

a) Compile the code. This time you do not use the parameter "T" because it is not a standalone application.

```
$sc -M NumberedCat
```

b) Create a streams instance – it is a single instantiation of the streams runtime system.

```
$streamtool mkinstance --template developer
```

c) Start the instance to start all the services associated with the instance.

```
$streamtool startinstance
```

d) After starting the runtime instance, the job corresponding to the above application is submitted to the runtime instance. After the compilation of the code, an Application Descriptive Language (ADL) file, is created in the output directory which contains information about the PEs. This ADL filename needs to be mentioned while submitting the job. The submission time value is given using the "-P" option.

```
$streamtool submitjob -P file=cat.txt output/NumberedCat.adl
```

e) In order to monitor the status of the PEs, run the following command. The PEs should show as "healthy". If it is not showing as healthy, then something is wrong.

```
$streamtool lspes
```

f) If you submit more than one jobs, then you can see the list of jobs by using the following commad:

```
$streamtool lsjobs
```

g) Once you have got the output in the file "data/result.txt", cancel the job. You can submit more than one job to the instance. For every new job that you submit without cancelling the previous job, your job ID increases by 1, starting from 0. Please take care to give the correct job ID while cancelling the job else you might end up cancelling the wrong job. If "0" is the job ID,

```
$streamtool canceljob 0
```

h)It is recommended to stop the instance once you are done with running your applications. You may or may not delete the instance.

```
$streamtool stopinstance                      // stop the runtime instance
$streamtool rminstance                        // remove the runtime instance
```

Note: You can use any of the above mentioned methods to write your application. You are not assigned with one specific lab machine so at the end of the lab, save your work in your U drive and delete your work from the VM.

Exercise 1:

Use the above program to output lines, prefixed with line numbers, for the text file "big.txt". The output must be directed to a file "bigresults.txt". Also, modify the program such that it outputs the number of occurrences of the words "history" and "adventure" in a file called "counts.txt".

<u>Hints:</u>
1) Add another standard operator to the code. Use either ***Functor()*** or ***Custom()*** operator.
2) Use "tokenize()" function. Check the code in "902_word_count" folder to get an idea of how tokenize works from the following link:
https://www.ibm.com/developerworks/mydeveloperworks/files/app?lang=en#/person/120000AMMH/file/ae7c2619-646b-45f0-98d1-ee0bc80d1a8f

The example code in "902_word_count" just retrieves the number of tokens generated. You can retrieve the tokens as follows: *list<rstring>tokens = tokenize(line, " \t", false).* The notations of this statement is from the word count example in "902_word_count".

The *tokenize()* function description is as follows:

public list<rstring> tokenize (rstring str, rstring delim, boolean keepEmptyTokens)
Tokenize string.

Parameters:

- ***str*** *:* input string
- ***delim*** *:* delimeters to use. Each character in delim is a possible delimiter
- ***keepEmptyTokens :*** keep empty tokens in the result

Returns:

```
        list of tokens
```

Know more about tokenize() and other standard types and functions from the following link:
http://pic.dhe.ibm.com/infocenter/streams/v3r2/index.jsp?topic=%2Fcom.ibm.swg.im.infosphere.streams.spl-standard-toolkit-builtins.doc%2Fspl-builtins.html

3) You can use conditional statements, such as “if”, “else”, etc, and loops such as “for”, “while” in the code. These conditional statements and/or loops can be included in the Functor() or Custom() operators.

Exercise 2:

An online shopping website gives a 1% discount to customers who make a minimum purchase of $250, 2% discount for a minimum purchase of $500 and 5% discount for a minimum purchase of $1000.

The input data stream has tuples of the form :
< Transaction ID, First Name, Last Name, Purchase Amount($) >

Process the input stream to generate an output of the following form:
<Transaction ID, Customer Name, Purchase Amount ($), Discount ($)>, where “Discount” is the amount in dollars which is computed using the above given logic.

The data is saved in a file “shopping.csv”. Direct the output to a file named “CustomerStatus.txt”

<u>Hints:</u>
1) Do not tokenize each line. Instead, in the ***FileSource()*** operator's parameter, use format type “csv”.
2) You can use Functor() or Filter() operator.