Lab7: Getting Familiar with Infosphere Streams and Streams Processing Language (SPL)

Department of Electrical and Computer Engineering
Iowa State University
Spring 2013

Purpose

A data stream is a sequence of data items that is being continuously updated through the arrival of new items, e.g sensor data or network packets. We have setup an environment on the cloud for real-time analysis of streams. The goal of this lab is to get familiar with writing, compiling and running a program using the Streams Processing Language (SPL) on IBM Infosphere Streams. More information can be found in the following link:
http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp

Submission

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

- A write-up answering questions for each experiment in the lab. For output, cut and paste results from your terminal and summarize when necessary.
- Commented Code for your program. Include all source files needed for compilation.

Resources Needed

You will need the following before you start the lab. If you don't have these, please contact the instructors.

- IP Address of the Streams Node
- Login id and private key file

Logging Into the Cloud

Log into the cloud using the private key given to you in the same way you did for all previous labs. Note this key and the IP address are different from the one you used in prior labs.

Understanding IBM Infosphere Streams and SPL

Infosphere Streams is a software platform that enables analysis and processing of massive data streams. Programs can be written using the Streams Processing Language (SPL). SPL has the following main components.

a) Data Streams/Tuples. Each individual element of a stream is called a "tuple".

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 some basic features of SPL. The application reads a text file and outputs the same text file with numbered lines.

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:

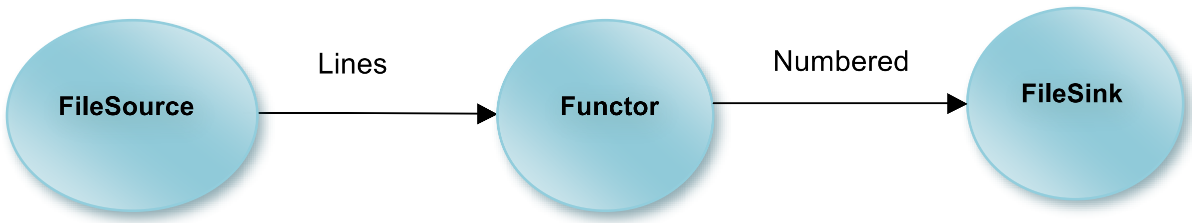
1 The Unix utility "cat" is so called
2 because it can con"cat"enate files.
3 Our program behaves like "cat -n",
4 listing one file and numbering lines.

The code 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
```

Streams Dataflow Graph

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



Compiling the Above Code:
a) After logging into the cloud, create a folder named "Lab1/Numbered" under your home folder.

```
$ cd ~
$ mkdir Lab1
$ cd Lab1
$ mkdir NumberedCat
```

b) Copy the above code in a file, "NumberedCat.spl", and save it in the *NumberedCat* folder. (You can create the *NumberedCat.spl* file in your local machine and then copy it in the cloud using scp/winscp).

c) Make sure you are in the "NumberedCat" folder.

```
$ cd ~/Lab1/NumberedCat
```

and run the following command to compile the code:

```
$ 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".

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 *NumberedCat/data/* folder 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
```

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.
```

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. To see how it is done, let us run the above code in a cluster of 10 machines:

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 an instance – it is a single instantiation of the runtime system. Do not create more than one instance because the system's load capacity is limited. If you want to create a fresh instance, delete the previous one.

```
$streamtool mkinstance --template developer –numhosts 8
```

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 probably 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. In the following command, "0" is the job ID. Because this is the first job you have submitted, the job ID is "0". 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.

```
$streamtool canceljob 0
```

h)There are other commands that you must be knowing about:

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

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().** The *FileSource()* operator reads data from a file and outputs it as a stream. Line 4 indicates that the file is read by *FileSource()* operator one line at a time. Each line of the file is a single tuple in the stream named "Lines".

The input file name is given during the job submission. 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. Note that the variable “i” has to be defined as “mutable” else incrementing “i” will be disallowed.

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”. “() as Sink” implies that the final output is not a stream, and it is named as “Sink”. You can have any name of your choice instead of “Sink”.

SPL provided standard operators are given in the following link:
http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp?nav=%2F2_1_7

To go through some more examples of SPL, check the below link:
https://www.ibm.com/developerworks/mydeveloperworks/files/app?lang=en#/person/120000AMMH/file/ae7c2619-646b-45f0-98d1-ee0bc80d1a8f

Exercise 1: (30 points)

Update the above program for the text file “big.txt” saved in the folder “/datasets” so that it outputs numbered line in the file “bigresults.txt”, and also outputs number of occurrences of “history” and “adventure” in a separate file called “counts.txt”. So the same code should output two files, “count.txt” and “bigresults.txt”. Draw a dataflow graph for the above streams program.

Attach the output file and the graph diagram in your submission.

Hints:
1) Add another standard operator to the code. Use either ***Functor()*** or ***Custom()*** operator. You can learn about the Custom() operator from the following link:
http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp?nav=%2F2_1_7_3
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 the other standard types and functions from the following link: SPL Standard Toolkit Types and Functions

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 also be included under the “logic onTuple” section of Functor() or Custom().

Exercise 2: (30 points)

Given a stream of data with bank balances of different customers, find a list of those customers whose balance is greater than $10M or less than $5000. The output tuple should be of the form <Customer Full Name, Account Number, Status>, where “Status” can assume one of the four values: ***Platinum*** for those customers whose balance is greater than $10M, ***Gold*** for those whose balance is greater than $5M, ***Silver*** for those whose balance is greater than $1M and ***Bronze*** for those customers whose balance is less than $1M.

The input dataset has tuples of the form :
< Account Number, Customer Last Name, Customer First Name , Balance >

The dataset is saved in “/datasets” folder as “bank.csv”

1) Write a code for this problem and save the result in output file “CustomerStatus.txt”
2) Draw a Streams dataflow graph for this problem.

Attach the output files and dataflow diagram to your submission

Hints:
1) You do not need to tokenize each line. Instead, in the ***FileSource()*** operator's parameter, use format type as “csv” instead of “line”.Get more information about the FileSource() and other operators from the following link: SPL Standard Toolkit Reference
2) Consider using the Functor() or the Filter() operators.