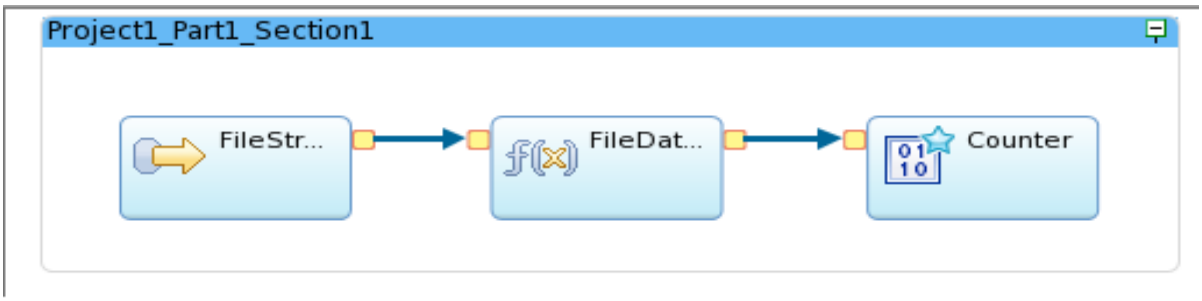
InfoSphere-based Stream Application

Word Count Application



Functionalities:

The application can read an existing file and count the total number of lines, words and characters, and it will print out the result on the terminal.

Graph Description:

In order to achieve above goal, I predefined a type of tuple which consists of the statistic data of each line, which are "the number of lines", “the number of words” and”the number of characters”. The name of this kind of tuple is *LineSta*and the format would be "type LineStat = tuple<int32 lines, int32 words, int32 chars>".

As it is shown on above picture, firstly, *FileSource* operator is employed to read data from an existing file and produce tuples as outputs using line format. In other words, each tuples emitted from this *FileSource* will contain a line of the source file. After that, these tuples are transmitted to a *Functor* operator, where the number of lines,the number of each line's words and characters are calculated by using tokenisation function. These counting information is then outputted as a format of the predefined *LineSta* tuple. Finally, these *LineSta* tuples are accumulated in a *Custom* operator, which will output the final result when it hits the *Sys.FinalMarker* punctuation.

How to run the application

1. Assume that folder Project1_Part1_Section1 is located at path MyPath.

2. Make sure that the file you would like to compute is located at./MyPath/Project1_Part1_Section1/data

2. Open up terminal and use cd command to get in MyPath.

3. Type the following command and you will see the result in a few seconds.

Sample input and command

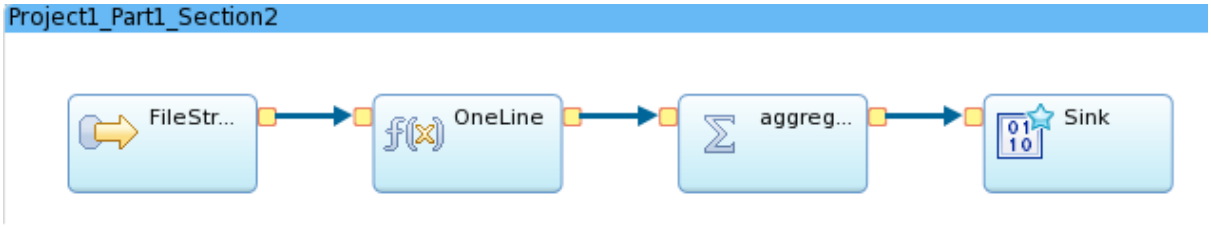
Assume the file is shaks12.txt which is located at data folder, the command would be:

./Project1_Part1_Section1/output/application.Project1_Part1_Section1/Standalone/bin/standalone application::Project1_Part1_Section1.file=shaks12.txt

Sample output

124456 line(s), 901325 word(s), 5582655 character(s).

Windowing Word Count Application



Functionalities:

The application can read an existing file and count the number of lines, words and characters over the last 100 lines of text, and it will print out the result on the terminal.

Graph Description:

In this application, the predefined tuple *LineSta*<int32 lines, int32 words, int32 chars>is employed to store the counting statistic data.

As it is shown on above picture, firstly, *FileSource* operator is applied to read data from an existing file and output tuples using line format .Secondly, these tuples are transmitted to a Functor operator, where the number of each line's words and characters are calculated by using tokenisation function. These counting information is outputted as *LineSta* tuples. Thirdly, *LineSta* tuples are accumulated in an *Aggregate* operator, where a tumbling window is configured as count(100) so that the counting will be performed over every 100 lines and after each 100-line aggregation the accumulated counting data would be cleared to zero.

Why we chose to use Aggregate operator?

Aggregate operator is used to compute user-specified aggregations over tuples gathered in a window, which perfectly match up our requirement that is performing counting over a fixed size of window. Additionally, the built-in *Custom* custom output function is helpful in accumulating counts and clearing counts.

How to run the application

1. Assume that folder Project1_Part1_Section2 is located at path MyPath.

2. Make sure that the file you would like to compute is located at ./MyPath/Project1_Part1_Section2/data

2. Open up terminal and use cd command to get in MyPath.

3. Type the following command and you will see the result in a few seconds which depends on the size of the file.

Sample input and command

Assume the file is shaks12.txt which is located at data folder, the command would be: ./Project1_Part1_Section1/output/application.Project1_Part1_Section2/Standalone/bin/standalone application::Project1_Part1_Section2.file=shaks12.txt

Sample output

100 line(s), 674 word(s), 4305 character(s).

100 line(s), 683 word(s), 4299 character(s).

100 line(s), 624 word(s), 4168 character(s).

100 line(s), 638 word(s), 4014 character(s).

100 line(s), 671 word(s), 4093 character(s).

100 line(s), 686 word(s), 4113 character(s).

100 line(s), 691 word(s), 4007 character(s).

100 line(s), 675 word(s), 4065 character(s).

...

Word Count Application with Submission Arguments

Functionalities:

The application can read an existing file and count the number of lines, words and characters over a user-specified size of window, and it will print out the result on the terminal.

Graph and Application Description

The graph layout is the same as Section2, the only difference is that Section 3's application can accept a submission time argument which specifies the size of the tumbling window used in Aggregation operator.

How to run the application

1. Assume that folder Project1_Part1_Section2 is located at path MyPath.

2. Make sure that the file you would like to compute is located at ./MyPath/Project1_Part1_Section2/data

2. Open up terminal and use cd command to get in MyPath.

3. Type the following command and you will see the result in a few seconds which depends on the size of the file.

Sample input and command

Assume the file is shaks12.txt which is located at data folder and we would like to perform counting over a window whose size is 50, the command would be:

./Project1_Part1_Section3/output/application.Project1_Part1_Section3/Standalone/bin/standalone application::Project1_Part1_Section3.file=shaks12.txt Project1_Part1_Section3.windowSize=50

Sample output

50 line(s), 351 word(s), 2196 character(s).

50 line(s), 327 word(s), 2129 character(s).

50 line(s), 157 word(s), 1243 character(s).

50 line(s), 239 word(s), 1608 character(s).

50 line(s), 318 word(s), 2036 character(s).

50 line(s), 381 word(s), 2290 character(s).

50 line(s), 333 word(s), 2150 character(s).

...

TCP Word Count Application

Functionalities:

The functionalities that Section 4 is aimed to achieve are the same as those of Section 3, the only difference is that instead of reading data from a FileSource operator, the application reads data from an external TCP client.

Here, we predefine the application will communicate with localhost:23456.

How to run the application

1. Assume that folder Project1_Part1_Section4 is located at path MyPath.

2. Make sure that the file you would like to compute is located at ./MyPath/Project1_Part1_Section4/data

2. Open up terminal and use cd command to get in MyPath.

3. Type the following command and you will see the result in a few seconds which depends on the size of the file.

Sample input and command

Assume the file is shaks12.txt which is located at data folder and we would like to perform counting over a window whose size is 50, the command would be:

./Project1_Part1_Section3/output/application.Project1_Part1_Section4/Standalone/bin/standalone application:: Project1_Part1_Section4.windowSize=50

Now execute the following Python script to send file data to localhost:23456

python tcp-txt.py -i shaks12.txt -s localhost:23146

Sample output

50 line(s), 351 word(s), 2196 character(s).

50 line(s), 327 word(s), 2129 character(s).

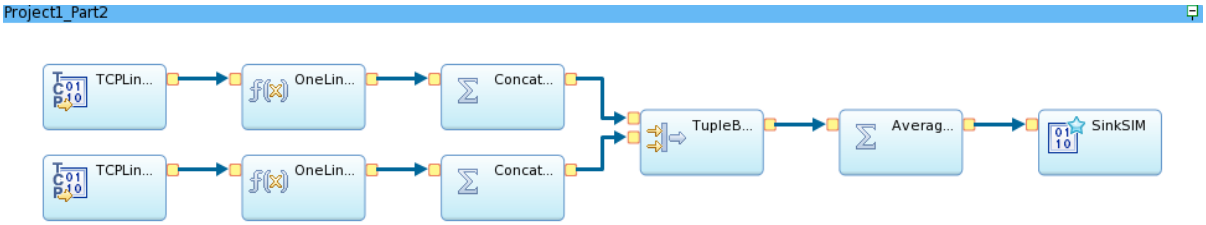
50 line(s), 157 word(s), 1243 character(s).

50 line(s), 239 word(s), 1608 character(s).

50 line(s), 318 word(s), 2036 character(s).

...

Document Similarity Application



Functionalities:

This application will take data (text) from two TCP sources and compute their similarity scores based on Formula 2.1. The scores will be printed on the terminal.

$$\frac{\sum_{i,j}(g(w_i,w_j))}{(\sqrt{\sum_{w_i \in d_i}(f_i)^2}) * (\sqrt{\sum_{w_j \in d_j}(f_j)^2})}$$

Formula 2.1

Main Idea

Why we did not choose to calculate similarity scores line by line? According to above formula, to compute the similarity score between two documents, we need to know the total number of words in each document as well as the frequency of each word. However, in the context of stream computing, the entire data source is unknown; therefore, we need to apply Window concept to cope with this stream application issue. One possible way would be that we take one line from each TCPSource to compute their similarity score and then we will average the scores over a window of lines, such as over a counter-based sliding window. Another possible way would be that rather than computing similarity line by line, we would compute the similarity over a bunch of lines together, which could decently improve the result accuracy. To achieve the latter way, before calculating the similarity scores, line tuples would be transmitted to an Aggregate operator first to be concatenated in a sliding window. For the sake of obtaining more accurate result, we decided to apply the second way.

How did we correlate two document data and compute their similarity? Initially, Barrier operator was applied to correlate two document data; however, one potential drawback of using such way was that if Filestream1 would emit 100 lines whereas FileStream2 would generate 1000 lines, only 100 lines' similarity score could be calculated. To resolve such issue, one possible way would be using Join operator instead of Barrier. In this way, the similarity calculating process would not be blocked in forementioned situation.

However, Joint operator would not work in a number of special cases, particularly could not synchronise tuples from two TCP sockets as Barrier does. One example would be if FileStream2 starts receiving data when FileStream1 already finished sending out tuples.

After making a trade-off between the capabilities of Barrier and Join, we eventually decided to employ Barrier operator for the sake of more accurate similarity results guaranteed by the tuple-synchronisation ability.

What is the problem of averaging similarity scores? The biggest issue of performing averaging is that it could not identify outliers data. However, we could not come up with any ideas to mitigate the impact brought by averaging calculation.

Graph Description

Step 1: FileStream1 and FileStream2 are two TCPSurce operators listening to two TCP sockets and output data in line format.

Step 2: Line tuples are appended with a space character, which would be useful in following concatenation operation.

Step 3: Lines from each file stream are concatenated in a sliding window.

Step 4: Barrier operator is used to correlate line data from previous operator.

Step 5: Average operator is used to perform averaging calculation.

Step 6: Similarity scores of each widow of the original file data are printed out here.

How to run the application

1. The two TCP sockets used by this application are 127.0.0.1:23146 and 127.0.0.1:23147.

2. Assume that folder Project1_Part2 is located at path MyPath.

3. Make sure that the files you would like to compare are located at ./MyPath/Project1_Part2/data/

4. Open up terminal and use cd command to get in MyPath.

5. Type the following command and you will see similarity scores being printed out in a few seconds.

Sample input and command

Assume the files to be compared are shaks12.txt and shaks13.txt being located at data folder, the command would be:

./Project1_Part2/output/application.Project1_Part2/Standalone/bin/standalone application::Project1_Part2.windowSize=600 Project1_Part2.avgWindowSize=30

Once the application is running, if there is any data coming from those two predefined sockets, the similarity score will be printed on the terminal. You can use following command to send file data to the specified servers/ports.

python tcp-txt.py -i shaks12.txt -s localhost:23146

python tcp-txt.py -i shaks13.txt -s localhost:23147

Sample output

The similarity of two document is: 1

The similarity of two document is: 1

The similarity of two document is: 1

The similarity of two document is: 1

The similarity of two document is: 1

...