VRIJE UNIVERSITEIT BRUSSEL

ASSIGNMENT 1: AKKA STREAMS

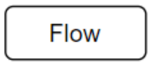
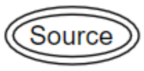
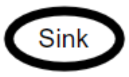
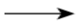
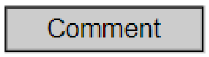
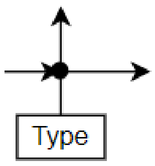
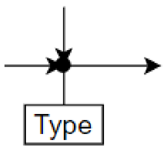
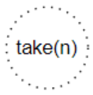
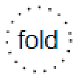
Software Architectures

NAME: ROBIN DE HAES
STUDENT NUMBER: 0547560

General

To process the stream of 200 000 text-based records of Maven dependency information a pipe-and-filter approach is used to split this task up in smaller, reusable parts that are executed sequentially or in parallel to incrementally come closer to the end result of extracting and collecting all dependencies and statistics in 2 separate files. First we will explain the smaller components that are used, after which we will show how these are combined to get the complete pipeline and the results we want.

We will first show the key for all diagrams that are used in this document.

Key	
	A “filter” component of Akka with one input and one output. An action is performed on the input data which transforms it in the output data being emitted.
	A Source component that has exactly one output and only emits data (but takes no input data).
	A Sink component that has exactly one input. It is only used to write away materialized values in our pipeline.
	The “pipe” connecting sources, flow and sinks.
	A comment providing additional information about a component’s behavior.
	Point where a stream of inputs is split in two or more output streams based on the specified type: *Broadcast*: A “copy” of each input element is emitted to each output stream. *Balance*: Each input element is emitted to the first available output stream. *alsoTo*: Each input element is sent to an attached Sink as well as to another component. *groupBy(attribute)*: Each input element is sent to a separate output stream based on the specified attribute of the input elements.
	Point where streams of inputs are combined to one output stream based on the specified type: *Merge(SubStreams)*: Input elements are merged one after another into the output stream. *Zip*: Combine an input element of each input stream into one element for the output stream.
	Only pass the specified number of elements (i.e. n elements) downstream.
	Fold all elements of the input stream into one emitted output element.

1. Source of LibraryDependency objects

Since flat text-based records don’t allow for the easiest processing, we mapped each record to a LibraryDependency object.

A **LibraryDependency** object is a direct mapping of one row in the text file and has 3 String-based fields: library, dependency and dependencyType with helper methods to quickly check if it is a Compile, Provided, Runtime or Test dependency. The library attribute could perhaps also be represented by an object with three components (GroupID, ArtifactID, Version) but this seemed redundant for our purposes. The attribute dependencyType could also have been made an object to ensure only valid dependencyTypes are present, but since all entries are valid this also seemed redundant.

When it comes to the actual processing we’re mostly interested in the stream of LibraryDependency objects as our source of elements and not the actual text file. This is why we used GraphDSL to create a Source component that abstracts away the generation of objects from the text file and simply emits a stream of LibraryDependency objects.

This **libraryDependencySource** is composed of:

1. A *txtSource* emitting the lines from the Maven dependencies text file
2. A *lineParsing* filter that splits each record up in its attributes (library, dependency and dependencyType)
3. An *invalidDependencyFilter* that checks if each record has 3 non-empty attributes. Further error handling could be added here (e.g. writing invalid entries to another file, more checks,…) but this is out-of-scope for the assignment since all entries can be assumed to be valid
4. A *conversionToLibraryDependency* filter that maps split records to LibraryDependency objects

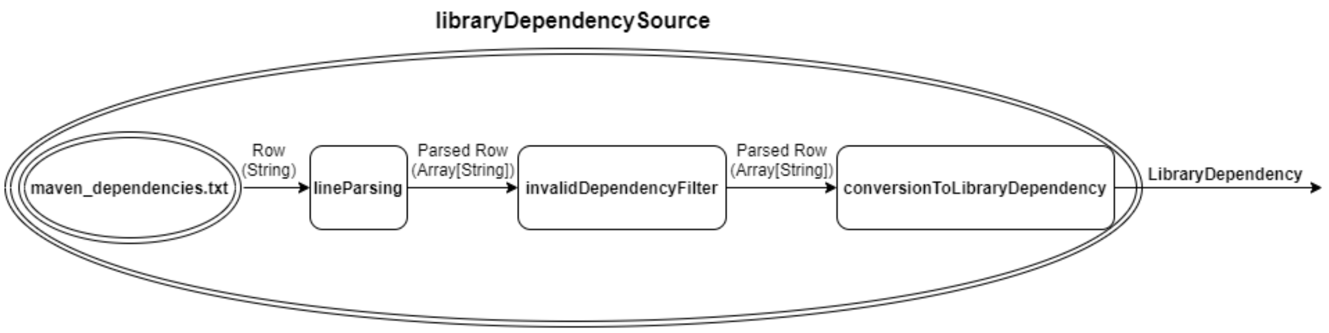


Figure 1. Diagram showing the composition of the libraryDependencySource component

2. Processing groups of LibraryDependency objects

Grouping

To allow processing of whole groups of libraries in parallel, where each pipeline processes one whole group of libraries at a time, we used the groupBy functionality.

This functionality splits up the stream of LibraryDependency objects from the libraryDependencySource into multiple substreams. We then fold all elements per substream into a List and merge the substreams again to end up with a stream of List[LibraryDependency] where each list contains all dependencies for one library. This whole behavior was abstracted away into the GraphDSL component **groupedByLibrary** for ease of use.

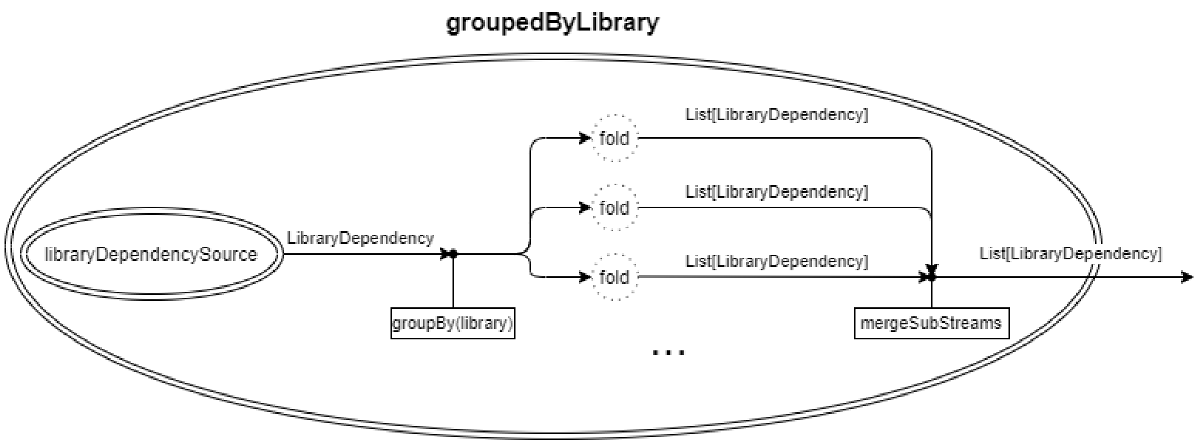


Figure 2. Diagram showing the composition of the groupedByLibrary component

<u>**Parallel processing**</u>

To process the groups in parallel, we use a balanced approach in which we send one List[LibraryDependency] to an available pipeline and then use a **flatMapFlow** component that performs flatMapConcat to convert this list back to a stream of LibraryDependency objects. Such a stream only contains LibraryDependency objects for the same library which makes processing them via **dependencyCounter** components and finally merging the results back together much easier. The balancing and merging is taken care of in the GraphDSL component **parallelDependencyCounters**. More information about the inner workings of the dependencyCounter component is given below.

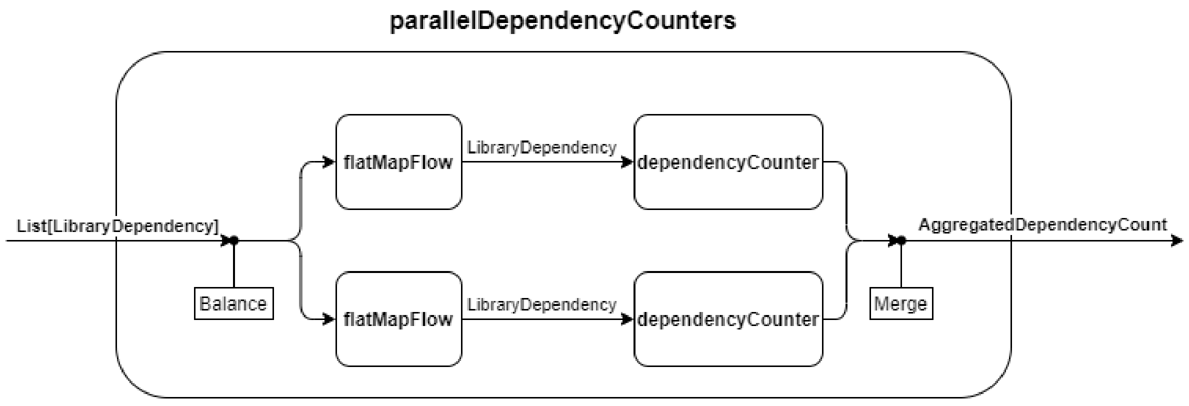


Figure 3. Diagram showing the composition of the parallelDependencyCounters component

<u>**Single pipeline**</u>

The GraphDSL component **dependencyCounter** creates one counter object for a processed stream of LibraryDependency objects. This counting functionality was abstracted away into a single component for easy reuse in parallel processing.

Some small, reusable filters were also made to be used in this component (and in the statistics pipeline later on):

- A generic elementCounter filter for counting the number of elements in a stream
- A filter that filters out LibraryDependency objects of a specific dependencyType (using the helper methods of the LibraryDependency class)

The dependencyCounter first broadcasts the incoming stream to five parallel streams: one for processing each dependencyType (so four in total) and one to keep track of the library name.

In the dependencyType processing streams we use the aforementioned filters to ensure we only have objects of the correct dependencyType and then we count all objects for that dependencyType. Since the library name is the same for all elements currently being processed, it is enough to only take one element in that stream.

Finally we zip all values back together to create an **AggregatedDependencyCount** object. It has the String field libraryName and Integer fields that each hold the count for one type of dependency (i.e. compileDependencyCount, etc.).

In summary, this component transforms a stream of LibraryDependency objects to a stream of AggregatedDependencyCount objects.

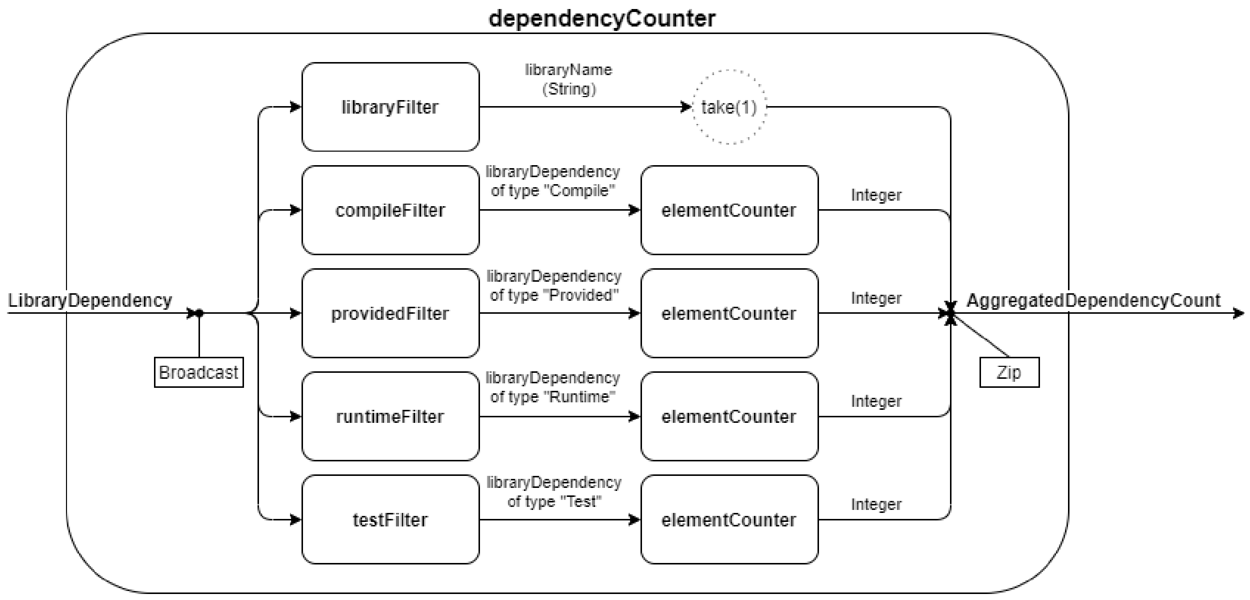


Figure 4. Diagram showing the composition of the dependencyCounter component

3. Sink for AggregatedDependencyCount objects

The previous processing steps result in a stream of AggregatedDependencyCount objects. Our **dependenciesSink** therefore simply needs to write these objects away to a *library_dependency_count.txt* file. AggregatedDependencyCount has a toString-method giving the required output format which makes this rather easy.

4. Statistics extension

To extract the statistics from the stream of AggregatedDependencyCount objects we send these elements to another GraphDSL component in addition to sending them to the aforementioned dependenciesSink.

This GraphDSL component has a similar approach as the previously described dependencyCounter. It first broadcasts the incoming stream to four parallel streams: one for processing each dependencyType. In each of the parallel streams it filters out objects that have a higher dependency count than a specified lower limit for that dependencyType and it eventually zips the results back into a statistics counter object.

This counter object is a **LibraryCount** object. It has the Integer field lowerLimit which is the specified lower limit that should be satisfied and four other Integer fields holding the count of libraries per dependencyType that satisfy this lower limit.

The previous processing steps result in a stream of LibraryCount objects. Our **statisticsSink** therefore simply needs to write these objects away to a *statistics.txt* file. LibraryCount has a toString-method giving the required output format which makes this rather easy.

*It should be noted that for testing purposes the GraphDSL component is created via a method **libraryCounter** taking a **lowerLimit** argument to be able to easily create components with different lower limits. The StreamApplication also has a field lowerLimit that is currently set to 2, but can be modified to test out different statistical computations.*

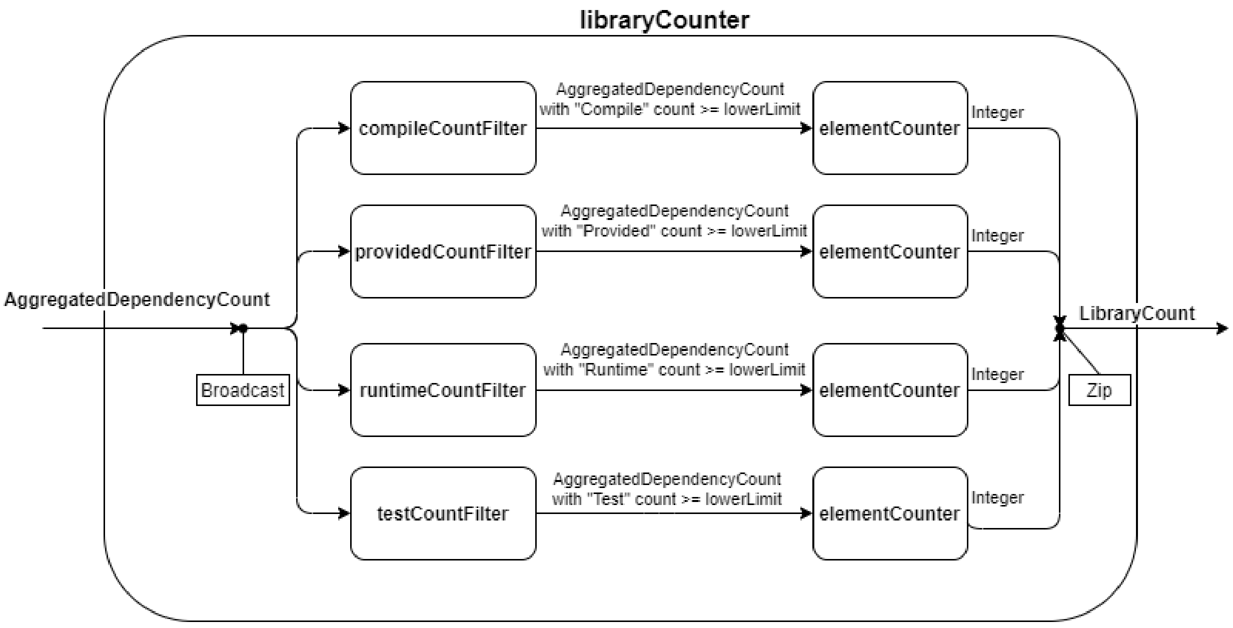
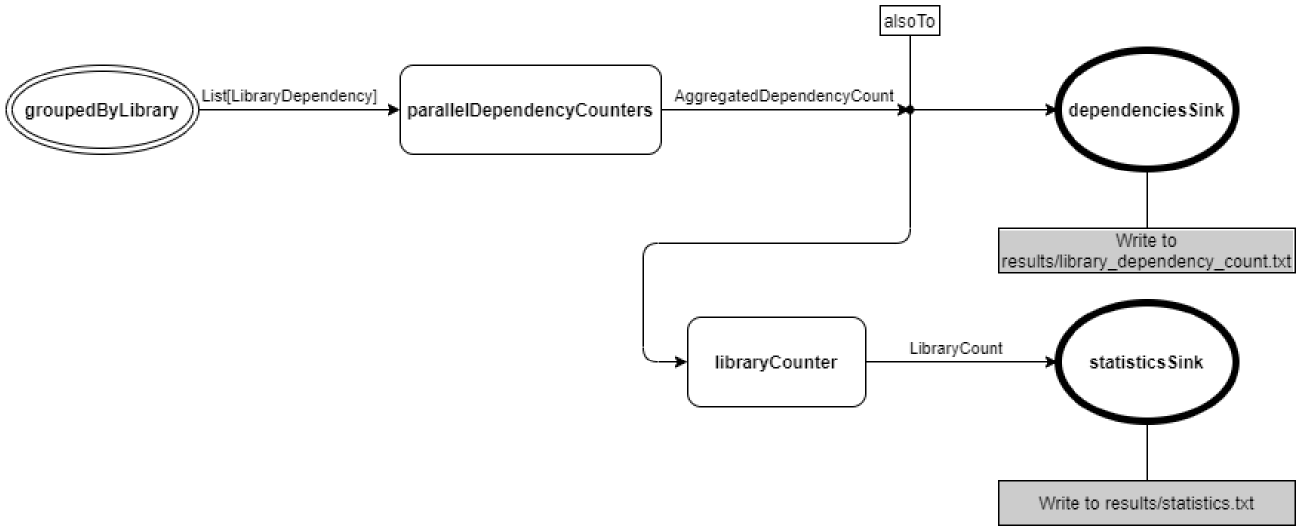

libraryCounter
AggregatedDependencyCount
Broadcast
compileCountFilter
providedCountFilter
runtimeCountFilter
testCountFilter
AggregatedDependencyCount with "Compile" count >= lowerLimit
AggregatedDependencyCount with "Provided" count >= lowerLimit
AggregatedDependencyCount with "Runtime" count >= lowerLimit
AggregatedDependencyCount with "Test" count >= lowerLimit
elementCounter
elementCounter
elementCounter
elementCounter
Integer
Integer
Integer
Integer
Zip
LibraryCount

Figure 5. Diagram showing the composition of the libraryCounter component

5. Full Pipeline


groupedByLibrary
List[LibraryDependency]
parallelDependencyCounters
AggregatedDependencyCount
alsoTo
dependenciesSink
Write to results/library_dependency_count.txt
libraryCounter
LibraryCount
statisticsSink
Write to results/statistics.txt