# Visualizing Project Relations on GitHub

Hendrik van Antwerpen
H.vanAntwerpen@student.tudelft.nl

Niels van Kaam
N.vanKaam@student.tudelft.nl

Software Engineering Research Group, EECMS
Delft University of Technology

**Abstract**

GitHub has emerged as an interesting object of study for software repository analysis because of the data available through its open API. Visualizing this data can be challenging, because of the lack of knowledge about its structure and because of the size of the dataset. For distributed processing of large data sets functional programming languages and techniques and models like MapReduce have become increasingly popular. In this report we describe a web based system that makes it possible to explore one type of relation in GitHub, the links between projects based on common committers. Users can explore the data by selecting time limits, link degrees and project language. This report describes the implementation of that system in the object-oriented function programming language Scala, using Sawzall style aggregators and other function programming techniques.

## 1 Introduction

In the field of software repository mining, GitHub has emerged as an interesting subject of study and tools have emerged to collect and publish the data it publishes (**?**). Making sense of the data and visualizing it in interesting ways is not trivial. Finding interesting correlations and metrics is hard. Interactive software that visualizes aspects of the data while the user can influence constraints like time period or project language allows easy exploration of the data and can be a starting point for more formal analysis.

We present a web based software system that allows a user to explore relations between projects on GitHub. A relation is defined by the existence of a common committer. The user can influence the time period wherein the link must exists, the main programming language of the project and the minimum number of committers before a link is shown.

We identify several challenges in the design of the software:

- Providing a general visualization that might reveal interesting properties of the data to the user.
- Dealing with the large size of the data that is being processed.
- Ensuring responsiveness and reduce waiting times of the software to encourage the user to explore.

The software was developed in the context of a functional programming course[1] at the Delft University of Technology. The course focused on functional programming techniques, application of these techniques in more imperative languages and using them for processing large data sets. The report therefore also describes how the functional programming and data processing techniques from the course were applied. The project was implemented using Scala and the code samples will be mostly in that language.

The report is structured as following:

- A description of the data and the data processing (section **??**).
- A description of a MapReduce implementation in Scala, created to implement the data processing (section **??**).
- A description of the distributed design of the backend, created to improve responsiveness of the software (section **??**).
- A description of the visualization used in the software (section **??**).
- Conclusions and ideas for further research and development (section **??**).

# 2 GitHub data analysis

The data we used was a list of all projects with their main programming language, a list of all users and a list of all commits on GitHub. A commit provides the basic link between a user and a project at a certain time. To give you an impression here are some statistics about the data:

| | |
|---|---|
| #projects | 600k |
| #users | 400k |
| #commits | 14m |
| #(user,project)/week | 3m |
| #links | 400k |

Some more details on the date can be seen in figures **??** (commits over time), **??** (link degree) and **??** (projects per user).

We allow the user to vary the following parameters:

- Project language (limits projects within one ecosystem).
- Time period in which the links are counted.
- Minimum degree of a link before it's shown.

The desired end result is a list of links between projects, a weight on each link of the amount of common committers and a weight on each project of the amount of connected projects. This last property can identify central projects in the graph.

Processing data can be elegantly expressed as a stream of filter and transform operations. For our purpose they are:

1. filter projects for selected language

[1] `http://swerl.tudelft.nl/bin/view/Main/FunctionalProgrammingCourse`

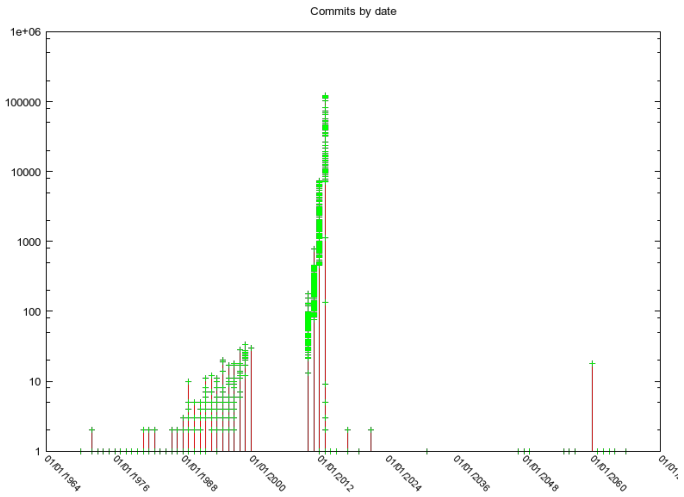


Figure 1: Commits by date

2. filter commits and exclude commits outside of the selected time period or not in the filtered project list
3. create a set of projects per user
4. per user, create links between all those projects
5. merge all project links, assigning them weight based on their occurrence
6. transform the list of links to a format understood by the visualization code

# 3 MapReduce with Scala collections

An increasingly popular model for processing big data sets is MapReduce (**?**). This model lends itself well for distribution and parallel data processing. One descendant of this model is found in Google's Sawzall (**?**), using aggregators as a key abstraction. In **?** a rigorous description of these two models is provided. The Sawzall concept of aggregators is identified as a generalized form of monoids. Their properties allow easy combination of parallel computed partial results, possible in a hierarchical way. Because monoids for common data types like lists, maps and tuples are readily available, developers need think less about the reduction of intermediate results and can focus on the map part of the process.

Sawzall is very liberal in the output it accepts from map functions. The output can be a monoid, a collection of the monoid or an element if the monoid is a collection itself. To capture all these Lämmel defines aggregators in terms of generalized monoids:

```
1  class Monoid m => Aggregator e m | e -> m
2    where
3      minsert :: e -> m -> m
```

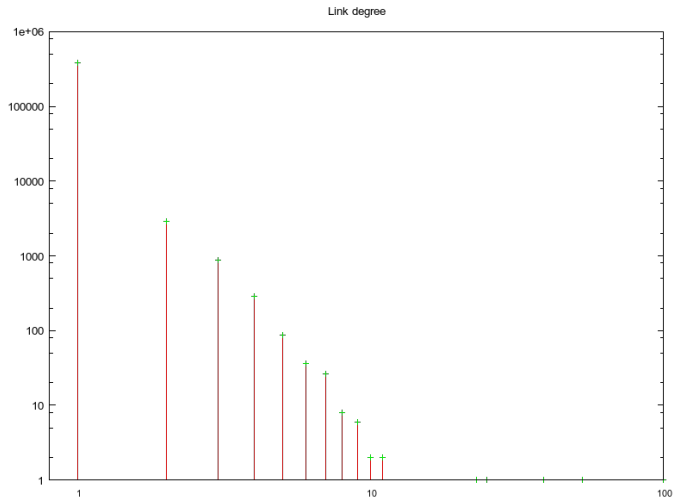


Figure 2: Link degree histogram

To our knowledge no implementation of this approach is available in Scala. Therefore we implemented a version of aggregators for Scala and used those to implement a map-reduce function that integrates seamless with the Scala collection library.

## 3.1 Aggregators

An aggregator is defined by the following trait:

```
1  trait Aggregator[A,B] {
2    def zero: A
3    def insert(a: A, b: B): A
4  }
```

Although Lämmel requires the type `A` to be a monoid, we dropped this requirement for reasons that will become clear later. An `Aggregator` where `A` and `B` are the same type behaves the same as a monoid of that type.

We defined the `minsert` function in Scala as `a |<| b`. Whenever this operator is used, the compiler infers a fitting Aggregator. All Scala **`implicit`** rules apply, so if no Aggregator is found there's compile time error, if multiple are found the programmer has to specify programmatically which one to use.

Assuming an appending aggregator for `List` and a summing one for `Int`, we now can do things like the following:

```
1  val r:Int = i:Int |<| j:Int // sum ints
2  val r:List[Int] = l:List[Int] |<| m:List[Int] // concat lists
3  val r:List[Int] = l:List[Int] |<| i:Int // append int element to list
```

Combining our aggregators with type inference and the very generic Scala collections library (see (**?**) for more details), we were able to provide aggregators for all Scala collections (including parallel ones) by implementing only seven aggregators. For each one of `Set`, `Seq` and `Map`, we implemented an aggregator for monoid behaviour and one for element insertion. Because we don't require our aggregator collection to be a strict monoid, we could make the monoid

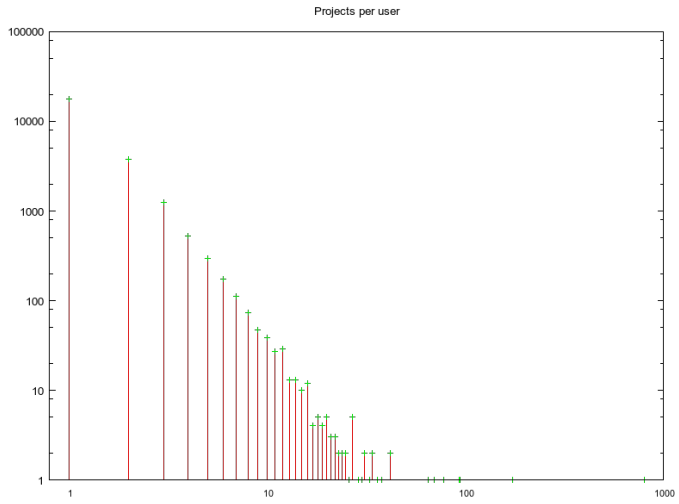


Figure 3: Projects per user histogram

aggregator very liberal in its accepted input. The collections library defines every collection type to be a subtype of `GenTraversableOnce[A]`. Instead of only accepting collections of the same type to be inserted, we can now accept any collection as long as the element type matches. Here are some examples:

```
1  val r:List[Int] = l:List[Int] |<| s:Set[Int]
2  val r:TreeMap[Int,Int] = tm:TreeMap[Int,Int] |<| sm:SortedMap[Int,Int]
3  //val r:Map[Int,Int] = m:Map[Int,Int] |<| l:List[(Int,Int)] // doesn't work?
```

The last example is something that is expected to work (`Map[K,V] <:Traversable[(K,V)]`, but is somehow not inferred by the compiler. Discussions online suggest that the compiler doesn't consider the complete type hierarchy when inferring, but only a few levels. The seventh aggregator was implemented to handle this special case.

A nice property of monoids is that they can be structurally combined. For example a tuple with two monoid values is itself a monoid. In our aggregator world, we happen to have similar properties. Aggregators for `TupleN` and `Map` where implemented to require their tuple values and map value to be aggregators themselves. The fact that we only have aggregators and don't differentiate between monoid append and element insertion really pays of here. When creating deep structures, on every level we have the choice to insert either a value or a collection. Some examples should show the flexibility this gives us:

```
1  // inserted tuples contain collections or elements. values are independent
2  val r:(Set[Int],Set[String]) = t1:(Set[Int],Set[String]) |<| t2:(Set[Int],Set[String])
3  val r:(Set[Int],Set[String]) = t1:(Set[Int],Set[String]) |<| t2:(Int,String)
4  val r:(Set[Int],Set[String]) = t1:(Set[Int],Set[String]) |<| t2:(Set[Int],String)
5
6  // three level deep structures
7  val m:Map[Int,Set[String]] = Map[Int,Set[String]] |<| Map[Int,Set[String]]
8  val m:Map[Int,Set[String]] = Map[Int,Set[String]] |<| Map[Int,String]
9  val m:Map[Int,Set[String]] = Map[Int,Set[String]] |<| (Int,Set[String])
10 val m:Map[Int,Set[String]] = Map[Int,Set[String]] |<| (Int,String)
```

Our design allows a lot of freedom in the shape of the elements inserted into a collection. What type of elements can be inserted is dictated by the defined aggregators and fully inferred by the compiler. The aggregators for the Scala collections are very generic and in most cases the developer doesn't have to care about how to merge collections. For opaque values aggregators are implemented for string concatenation and integer summing, but others are very easy to implement; one implicit function and an implementation of the `Aggregator` trait is enough.

## 3.2 MapReduce

Using the aggregators we now have, a map-reduce library was implemented. We wanted the map-reduce functionality to be as easy to use as the standard collection functions, like `map` or `filter`. To a high degree this can be achieved by enriching libraries, a process similar to defining type classes in Haskell (see **?** for details).

Using again the generic design of the collections library and type inference allows us to write map-reduce by only specifying the expected result type. A word count example similar to one Lämmel gives is:

```
1  val docs: List[String]
2  def wordcount(doc: String): List[(String,Int)] = doc.split(" ").toList.map( w => (w,1) )
3  val wc = docs.mapReduce[Map[String,Int]]( wordcount )
4  // wc has type Map[String,Int] here
```

The performance of our map-reduce in the same order and range as hand written code using e.g. `foldLeft`. Most of the work is done through inference by the compiler. At runtime the aggregators use folds and collection operations just as you would in handwritten code. There a little overhead of some function calls, but all the reduction details are abstracted away, resulting in less repetition and simpler data transformation functions.

With this result we can write our processing steps from section **??** in Scala. You can see how they mix with the existing collection library functions.

```
1  val commits: List[Commit] = ...
2  def projectsToPairs(ps: List[Project]): List[(Project,Project)] = ...
3  val linkMap =
4    commits.filter( c => c.timestamp >= from && c.timestamp <= until )
5           .mapReduce[Map[User,Set[Project]]]( c => (c.user,c.project) )
6           .values
7           .mapReduce[Map[(Project,Project),Int]](projectsToParis)
8           .filter( _._2 >= minDegree )
```

It is possible that multiple aggregators for the same type exist, for example a sum and a product on numbers. In such a case it is very easy to specify which one to use. Here is a small example:

```
1  def SumMonoid: Aggregator[Int,Int] = ...
2  def ProductMonoid: Aggregator[Int,Int] = ...
3
4  val words: List[String] = ...
5  val sumOfWordLengths = words.mapReduce( w => w.size )(SumMonoid)
6  val prodOfWordLengths = words.mapReduce( w => w.size )(ProductMonoid)
```

As you can see the result type is not required, because it is inferred from the aggregator.

One aspect of the Sawzall model we described in the beginning is not addressed yet. The possibility to insert a collection. This case is handled by the `flatMapReduce` function, which is the map-reduce version of `flatMap`. This does nothing more then iterating over a `Traversable` and inserting every element individually. Because every Scala collection is a `Traversable` of its element type, reductions of collection monoids can be written both with `mapReduce` as with `flatMapReduce`. The implementation in the former could be more efficient, taking the more specific type into account.

# 4 Distributed data-processing with Akka

Even tough the computation of the project links has been optimised, still a lot of data needs to be processed. Our dataset consisted of 21 million tweets . Because of this a request on a set of project links for a given time-interval and degree could take multiple minutes to compute. Such long waiting times are not acceptable, so we decided to distribute the computation over different machines. Because the system was built in Scala, the most logical choice for this is using the akka toolkit.

valideren of dit klopt

- Short explanation about akka with references to current uses goes here
- after that, a short explanationabout actors and how akka sees actors

In order to reduce the total computation time by distributing the data over multiple actors, the data needs to be cut into multiple parts. Simply cutting the dataset in multiple pieces is not possible in this case. A link between a project is defined as a "common committer". By splitting the dataset and computing one for example one half on one actor and the other half on a second actor could cause these common commits.

Therefore we came up with the following better solution. We initially give each actor system access to the entire dataset. Then, when an actor is created, it receives a set of users for whose commits it is responsible. This set of users we defined by taking the remainder of a modulo operation on the userid. With a set of user's the actor can now compute the entire process towards the project links, except for the filtering on the degree of the link. For that last operation an actor needs to be sure that it has all links between a project in order to be sure the result is correct, for which it would also the commits from the other users.

A disadvantage is that booting up the system does not receive any speedup, because every actor needs to read all commits in order to select the commits based on the users it is responsible for. This however does not influence the response time, because the system is only booted once after which the user's commits are kept in the memory of the actors. Finally once the actors respond to the request, they send all the project links they computed. The actor that requested the project links receives them and a reduction is done, after which the filter operation over the degree can be done.

The resulting actor system is shown in the graph below
graph met webservice-actor en een aantal computation actoren

- diepere uitleg implementatie met futures

- uitleg schaalbaarheid van systeem, met diagram van uitgebreid systeem
- hypothese over snelheidsoptimalisatie

  Content die hendrik wil:What did we want to address How did we design the actors, where was the caching, where which processing steps What was the performance (did we speed up response time?) A design diagram

# 5 Visualization in the browser with D3

* What did we want to show?
* How did we implement it (D3, what graph parameters)
* How did we emphasize characteristics (e.g. central projects)
* An example result

# 6 Conclusions and further research

* Did it work
* Is it responsive?
* Did we show interesting things
* What more interesting things could fit in this framework?

We believe the given approach can be easily adapted to include other parameters or show other relations in the data.

* Optimization by only considering weeks

# Appendix

# A Scala collection Aggregator example