CloudVoting: Analyzing Preferences using Spark and GraphX

Theresa Csar
TU Wien, Austria
csar@dbai.tuwien.ac.at

Abstract

Due to the availability of very big amounts of data, cloud computing technologies have been highly developed and researched. The use of cloud computing technologies was always associated with a high effort for programming and for providing availability of infrastructure. In recent years this has significantly changed and the aim of this paper is to make cloud computing technologies available to work with large sets of preference data. In this paper we present a tool to read preferences in Spark (a cloud computing framework), to transform this preference data to the weighted majority graph or similar data representations, and to run basic analysis on those. Spark can be run locally and therefore no cloud computing infrastructure is needed to make use of this tool. The described tool written in Scala is called CloudVoting and is available open source on GitHub. We apply the tool of CloudVoting to a real world dataset and make some observations about performance.

1 Introduction

In the last decades the amount of data has been rapidly increasing and technologies and methods have been developed to cope with the growing amounts of data. The produced data is diverse and comes in lots of different structures. On the one hand there are big databases growing over time, for example companies store all their information about sales, costumers and products. On the other hand there are automatically generated datasets that come in very complex shapes and are often hard to reason about. Such generated datasets are created for example during the interaction of a user with a web interface like some online shopping site. In this paper we focus on preference data, created by a mechanism that ranks entities in some way. The most basic example is an election, where every voter ranks all (or a subset of) candidates. In more complex cases the preference data is automatically generated from user behaviour. For example the choice of a user clicking a certain link, or buying a product can be interpreted as a vote for one product over one or many other. When using streaming services also the choice of movies or songs can be interpreted as votes or even rankings can be calculated (e.g. the most watched movies per country and month). For instance look at a music streaming service, where billions of users stream music every day. For each day, each country, each genre etc. a preference list (ranking) can be extracted from this data. Such preference datasets can get very large very fast, which makes the analysis with classical winner determination methods on a local computer very time consuming or even impossible.

Huge amounts of data occur in many application areas and in the last decade the trend for managing these datasets goes towards cloud based systems. One of the ongoing challenges in this research area is to design efficient algorithms for those systems. There are two main programming paradigms for designing cloud based algorithms, those are MapReduce (Dean and Ghemawat 2008) and Pregel (Malewicz et al. 2010). MapReduce is a computation framework for batch-processing of big data sets in distributed systems. In the first step of a mapreduce procedure the data is splitted by mapping datavalues to key-value pairs. The key-value pairs are then distributed among the nodes grouped by their key and in the last step the data is processed on each node and saved to the file system. Hadoop is an open source implementation of MapReduce. Hadoop rapidly developed into a large ecosystem of cloud management tools and is now much more than "just" MapReduce. The concept of basic MapReduce is very static and has great disadvantages when it comes to dealing with incremental computations or streaming data. In order to be more flexible when it comes to complex distributed computations a new framework – Spark (Zaharia et al. 2010) – has been developed. Spark is an open source implementation combining the power of MapReduce with methods to speed up the computation and to make it more adaptive to the users need. Compared to Hadoops MapReduce it gains a lot of speedup by loading data into memory. For graph-based computations the concept of Pregel (Malewicz et al. 2010) has been developed. Pregel is a graph-based framework and the algorithms are formulated in a vertex centric manner, where vertices act as entities that send messages to their neighbours and process the received messages. With GraphX (Xin et al. 2013) the Pregel API is made available in Spark. GraphX comes together with a graph datastructure, many basic graph operators and algorithms.

When dealing with large preference datasets it can be a huge advantage to use cloud computing technologies. There

has already been some work on MapReduce algorithms for WinnerDetermination (Csar et al. 2017), but the use of Pregel-like technologies and Spark has not yet been discussed. The goal of this paper is to bridge this gap with proposing a tool to deal with preference data in Spark. The tool is implemented in Scala and is called *CloudVoting*. The source code is available on Github [1]. CloudVoting provides the tools for reading preference data from widely used dataformats as are the formats used by PrefLib (Mattei and Walsh 2013). CloudVoting includes methods for reading and transforming preference data into weighted majority graphs and similar graph datastructures. Basic winner determination rules are applied to the preference data using both the MapReduce and Pregel paradigms that are supported by Spark and GraphX.

Organisation and Main Results In this paper the basic tools for reading preference data into Spark and GraphX are explained and also the implementation and application of some basic winner determination rules are shown. The paper is structured as follows: Section 3 gives an introduction to cloud computing technologies, in particular MapReduce and Pregel and discusses some perfomance measures for cloud computing algorithms. The cloud computing framework Spark and the library GraphX are explained in Section 4. In Section 6 the parsing of preference data into GraphX, the used datastructure and supported data formats are explained. In Section 7 some basic social choice rules and their implementation in CloudVoting are documented. All functions have been tested and evaluated on real world data and the results can be found in Section 8. Also observations regarding performance and the underlying framework are made. These observations may be of use of future development of algorithms for dealing with preference data.

2 Related Work

Computational social choice is an active research topic as is the application of voting rules to preference data and there are already some tools available for handling preference data. Democratix (Charwat and Pfandler 2015) is a logic based implemenation of voting rules available as a python tool making use of ASP methods. Pnyx (Brandt, Chabin, and Geist 2015) is an online tool for aggregating and collecting preferences. Spliddit (Goldman and Procaccia 2015) is an online tool for fair division algorithms. None of these tools deal with big datasets or distributed algorithms.
Preference datasets have been made available and dataformats have been defined. In particular Preflib (Mattei and Walsh 2013) provides lots of preference datasets in different datatypes and conversion tools. CloudVoting supports the preflib data format as input.

3 Cloud Computing Techniques

There are two main programming paradigms used for designing algorithms for cloud systems. There is MapReduce that is used for batch processing and Pregel that is designed to deal with huge graphs or networks. Both have been highly researched in recent years. In this Section MapReduce and Pregel are explained and we summarize important recent work on performance considerations for algorithms using those paradigms.

3.1 MapReduce

MapReduce has been introduced by Google in 2008 (Dean and Ghemawat 2008). It is a computation paradigm to formulate distributed algorithms. The first phase is the map phase where the dataset is mapped to key-value pairs. Those key-values pairs are distributed among the cluster nodes in the shuffle phase and the computation happens in the last phase - the reduce phase. The most prominent example for MapReduce is word count. The input is a file containing text and in the map phase each word is mapped to a key-value pair with 1 as value (key=word, value=1). In the shuffle phase all key-value pairs are grouped by key and sent to the assigned node / reduce task. In the reduce phase all values are summed up for each word and as a result we get the word counts.

It is possible to chain several MapReduce jobs, but this requires the system to save the data after each job and read it in the next phase again. MapReduce is not very well suited for such iterative computations. There are already MapReduce adaptations that can deal with such iterative computations.

Performance Considerations In the map-phase first of all the **input data size** is of interest. If the values in the input data set are mapped to several key-value pairs we can get a high **replication rate** (Afrati et al. 2013). The replication rate is the average number of key-value pairs produced by one input value. The produced key-value pairs are then shuffled among the executors. Therefore a high replication rate also leads to high **communication cost** in the shuffle phase. If several MapReduce computations are chained, the **number of rounds** is also an important performance measure.

3.2 Pregel

Pregel has been introduced by (Malewicz et al. 2010) as a distributed graph processing framework. The main concept of the framework is that algorithms are formulated in a vertex-centric manner. The input graph consists of vertices and directed edges, where each edge connects exactly two vertices. In each round the vertices send messages to each other along the edges and each vertex processes the received data. If a vertex does not receive any messages in one round it is marked as inactive, but can be reactivated later. The whole computation stops as soon as all vertices are marked as inactive.

Performance Considerations As with MapReduce also for Pregel the communication cost and the space usage is very important. A pregel algorithm is called balanced and practical (BPPA) if it has **linear space usage**, **linear communication cost**, **linear computation cost** and uses at most a **logarithmic number of rounds** (Yan et al. 2014). For many problems it is hard to find algorithms that meet these performance guarantees. We propose a pregel algorithm for

[1] https://github.com/theresacsar/CloudVoting

computing the SchwartzSet in Section 7.4 that is based on a BPPA for computing Strongly Connected Components.

4 Cloud Computing Frameworks

4.1 Spark

Spark is a cloud computing framework and widely applied to many data-intensive tasks. It has first been introduced in 2010 by (Zaharia et al. 2010). The core concept of Spark is the Resilient Distributed Dataset (RDD). RDDs are distributed and read-only collections of objects. The data can be held in memory on the nodes in the cluster. With the basic method of MapReduce it was necessary to read and write to the filesystem after each MapReduce step. The ability of loading data into memory for reusing it in following steps gains enormous speed to iterative computations and makes interactive analysis possible. RDDs are distributed and replicated across the nodes in the cluster, this does not only speed up the computation but also provides a lot of data safety in case one partition of an RDD is lost. The partitioning can be configured, but also internal Spark processes optimize the process.
RDDs are evaluated in a lazy fashion. This means that the RDD is created only when the data is needed and not before. This provides Spark with all the information it needs for further optimization of the execution plan.
RDDs can be created by loading data from a file or by parallelizing an already existing collection of objects. The second approach is usually not applicable since this would require you to have the whole dataset in memory on only one machine. The first approach also has the advantage that you can use a file that is already stored in a distributed file system like hdfs (hadoop distributed file system).
Two types of RDD-methods are distinguished: actions and transformations. By transformations a new RDD is created, but due to the lazy-evaluation it is not yet computed, but a computation plan is created. The computation happens only when an action is called. This makes sure that there are no unnecessary objects in memory and no unnecessary computations are performed.

When starting a spark program a SparkContext has to be created. The SparkContext saves all information about the available resources and the workspace. In our code the SparkContext is saved in a variable called *sc*.

```
val conf = new SparkConf()
   conf.setMaster("local")
   conf.setAppName("CloudVoting")
val sc = new SparkContext(conf)
```

RDDs and MapReduce In Spark the advantage of lazy evaluation comes into play when performing MapReduce computations. The map phase corresponds to a transformation of an RDD whereas the reduce phase corresponds to an action. Since Spark won't start the computation before the action is called it already has a lot of information about the ongoing computation and can construct an efficient computation plan and optimize the partitioning. The basic MapReduce example wordcount is programmed in Spark using the programming language Scala as follows:

```
val file = sc.textFile(inputpath)
val words = textFile.flatMap(
         line => line.split( ))
val counts = words.map(word => (word, 1))
         .reduceByKey((v1,v2) => v1 + v2)
counts.saveAsTextFile(outputpath)
```

Each line of the file is read and split into words. The words are then mapped to key-value pairs of the structure (key=word, value=1) and in the reduce phase the key-value pairs are grouped by the keys (the words) and the sum of the values is computed. The file at the *inputpath* is not read before the action `.reduceByKey` is called.

4.2 GraphX

GraphX (Xin et al. 2013) is an extension to Spark, with the main contribution of the distributed Graph Datatype, which is an extention to the RDDs. GraphX also provides some basic graph operators and algorithms. The Graph Datatype is based upon RDDs and is therefore distributed in the same fashion. GraphX provides methods to optimize the partitioning of the Graph on the cluster in order to avoid huge communication costs and to speed up the computation time.

For a Graph `Graph[VD, ED]` the vertex datatype `VD` and the datatype of edge attributes `ED` are given. Graphs can be created from any set of `VertexRDDs` and `EdgeRDDs`. `VertexRDD` is a special type of `RDD` with the structure `RDD(VertexID, VD)` and the additional constraint that each `VertexID` can occur only once. An `EdgeRDD` can be created from any `RDD` of the structure `RDD(src=VertexID, dst=VertexID, attr=ED)`.

Pregel API The general mechanism of Pregel is explained in Section 3.2. The Pregel API of GraphX looks the following way:

```
def pregel (
  initialMsg: A,
  maxIter: Int,
  activeDir: EdgeDirection) (
    vprog: (VertexId, VD, A) => VD,
    sendMsg: EdgeTriplet[VD, ED] =>
        Iterator[(VertexId, A)],
    mergeMsg: (A, A) => A) : Graph[VD, ED]
```

The parameters of the function are the initial Message `intialMsg` of type `A` (sent to all vertices in the first step), the maximum number of iterations `maxIter` and the active direction of the edges. If `maxIter` is not set the computation stops when all vertices are inactive - i.e. no vertex received a message in the preceding step. The user has to provide a vertex program `vprog(VertexID, VD, A)=> VD`, which is a function every active vertex executes after all messages have been sent. The messages are of type `A` and the VertexDatatype is `VD`. The `sendMsg` function returns an

Iterator over the destination IDs and the corresponding messages. The `sendMsg` function is applied to all EdgeTriplets. EdgeTriplets contain the attribute of an edge including the vertex attributes of both connected vertices. All messages with the same destination vertex can be combined using the function `mergeMsg`. This allows for some precomputation before the actual vertex program `vprog` starts. The result is a graph `Graph[VD,ED]` with the same vertex and edge datatypes as the input graph (org.apache.spark 2017). [2]

5 Input: Preference Data

In this paper we consider elections with m candidates and n votes. Each vote is a strict linear partial order of candidates, that indicate the preferences of the respective voter or some other kind of ranking. Already prepared preference data can be found on the online platform PrefLib (Mattei and Walsh 2013) in several different data formats. Preferences can be generated from many other different types of input data. In our real world data example in Section 8 we use rankings created by the music-streaming service spotify [3]. The data is provided as .csv but we transform it to the Preflib dataformat .soi.

5.1 Preflib

Preflib is providing a wide range of preference data (Mattei and Walsh 2013) and CloudVoting allows you to use many of the provided dataformats as input. The election datasets are provided in different formats in PrefLib. The following formats are supported by CloudVoting:

- Strict Orders - Complete Lists (soc): The file contains several votes, where each vote is a list of strictly ordered preferences.
- Strict Orders - Incomplete lists (soi): allows a voter to leave some candidates unranked
- Majority Graph (mjg): a directed graph indicating which candidates are preferred over each other by the majority of votes.
- Weighted Majority Graph (wmg): a directed graph indicating which candidates are preferred over each other by the majority of votes, weighted by the difference between the number of votes that support the edge and the votes that support the edge in the other direction.
- Pairwise Graph (pwg): a directed graph, where each edge is weighted by the number of voters supporting it.

6 Data Structure: Preferences in CloudVoting

In CloudVoting the elections are represented as different types of weighted graphs. The edge datatype is `Long` and contains the number of votes supporting this edge, or a similar measure. The vertex datatype is `String` and contains the name of the candidate. The `PairwiseWeightedGraph` contains all votes in both directions. The `Strict-` and `WeakDominanceGraph` only contain an edge between vertices a and b if a dominates b, i.e. if more votes prefer a over b than the other way around. In the `Strict-` and `WeakDominanceGraph` the weight at the edge $a \rightarrow b$ is the the number of votes preferring a over b minus the number of votes preferring b over a. The `WeakDominanceGraph` also contains edges with weight 0, whereas in the `StrictDominanceGraph` these edges are deleted. All those voting graphs have the datatype `Graph[String,Long]`.

If we want to calculate scores, we save the scores directly at the vertex. For this case we use a different VertexDatatype `(String,Long)` and therefore a different GraphDatatype `Graph[(String,Long), Long]`. The available scored Graph Datatypes are: `ScoredPairwiseWeightedGraph`, `ScoredWeakDominanceGraph` and `ScoredStrictDominanceGraph`. The weights at the edges are calculated the same way as with the corresponding unscored Graph Datatypes.

Conversion functions between all Graph-datatypes are provided. For example the conversion from `PairwiseWeightedGraph` to `WeakDominanceGraph` can be very easily performed thanks to the API of GraphX.

```
def toWeak(pwg: PairwiseWeightedGraph) :
    WeakDominanceGraph
```

First the reverse graph is created and than the edges attributes are transformed such that the weight is the original weight multiplied by (-1). From the resulting reverse graph with negative weights we only select the edges.

```
var revedges = pwg.reverse
   .mapEdges(edge => (-1) * edge.attr)
   .edges
```

In the next step the attributes of edges connecting the same vertices are summed up and only the edges with positive value are kept. This is done by first creating a graph that contains both the original and the reverse edges (graph). Then the edges are grouped and new weights are calculated. The resulting edges are filtered and only the edges with positive weights are kept.

```
val graph = Graph(pwg.vertices,
  pwg.edges.union(revedges))

val edges = graph
  .groupEdges((i, j) => i + j)
  .edges.filter(e => e.attr >= 0))

Graph(pwg.vertices, edges)
```

[2] https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.graphx.Pregel

[3] https://www.kaggle.com/edumucelli/spotifys-worldwide-daily-song-ranking/data

7 Winner Determination

In this paper we consider elections of the following kind: There are m candidates and n votes. Each vote is a strict linear partial ordering of candidates.
In social choice we often use scoring rules. Some of those scoring rules are calculated directly from the votes by assigning each candidate a score relative to the position in the vote. The scores resulting from all votes are summed up to get the final score and the candidates with the highest scores are selected as winners (e.g. Borda-Score, Plurality, ...). There are also scoring rules based on the dominance graph or weighted majority graph of the election. A directed edge between candidates a and b in the dominance graph indicates that a dominates b. We say that a dominates b if there are more votes ranking a before b. In the weighted majority graph this edge has the number of votes preferring a over b minus the number of votes preferring b over a as weight. The Copeland Score is based on the Dominance Graph. The score of a is the number of candidates dominated by a minus the number of candidates that dominate a. (i.e. the difference of outgoing and incoming edges). Again the candidates with the highest scores are selected.

There are other methods for winner selection, that are not based on scoring (e.g. Smithset, Schwartzset, Simpson, Schulze, ...). We will briefly discuss the SchwartzSet, but leave further voting rules and more detailed discussion for future work. The selection of discussed voting rules in this paper comes hand in hand with the general work flow when analyzing preference data. The scoring rules based on the preference data can easily be computed while actually reading the data into the framework and creating the graph representation of the election. Calculating the graph representation is already a computationally expensive problem and makes full use of the spark infrastructure. We focus on some basic social choice rules, the design of algorithms for more complex problems is future work.

7.1 Borda Score

Borda scores assign a value to each candidate based on their position in the ranking (Brandt, Brill, and Harrenstein 2016). In the CloudVoting implementation Borda scores are calculated while reading from a file containing preference lists. Each vote is a list of preferences and each candidate receives points relative to its position in the ranking. In CloudVoting the following version of Borda Scores is implemented: For a preference list of length m the candidate ranked first receives $m - 1$ points, the candidate ranked second receives $m - 2$ points and so on. All points are then added up for all voters and the candidate with the highest score is selected as the winner (Brandt, Brill, and Harrenstein 2016).

7.2 Other Scoring Rules

When reading the preferences from a file the following scores can be computed in CloudVoting. The weights are assigned to each candidate in the preference lists at the corresponding position in the ranking. The final score is the sum over all weights. The weights for some other scoring rules are shown in table 1.

Rule	Weights
Plurality	(1, 0, 0,..., 0)
Anti-plurality	(1, 1, 1,..., 1, 0)
k-Approval	(1, 1, 1,..., 1, 0, 0,..., 0)
Borda	$(m-1, m-2, ..., 0)$

Table 1: Weights of Scoring Rules (Brandt, Brill, and Harrenstein 2016)

7.3 Copeland Scores

Copeland scores are computed based on the dominance graph and depend on the number of candidates a candidate is defeating and is defeated by. The Copeland Score is defined as: CopelandS(a) $= |\{b|a \succ b\}| - |\{b|b \succ a\}|$ (Brandt, Brill, and Harrenstein 2016). In other words: the Copeland-Score is the difference of the number of outgoing and incoming edges in the dominance graph.

```
def CopelandScores(graph: Graph[String,
    Long], sc: SparkContext): ScoredGraph
    = {
 val out = graph.outDegrees
 val in = graph.inDegrees
 val scores: RDD[(VertexId, Long)] =
    out.join(in).map(degrees =>
    (degrees._1,
    degrees._2._1-degrees._2._2))

 var resultgraph: Graph[(String, Long),
    VertexId] =
    graph.outerJoinVertices(scores) {
  case (id, name, Some(score)) => (name,
     score)
  case (id, name, None) => (name, 0)
 }

 resultgraph
}
```

The outdegree and indegree of each vertex can easily be computed using the provided methods of GraphX. The scores are computed from degrees by joining indegrees and outdegrees and calculating the difference. Then the scores have to be joined with the old vertices in the graph to create the new scored graph.

7.4 Schwartz Set

The Schwartz set is defined as the union of all minimal undominated sets of vertices in the strict dominance graph (Brandt, Fischer, and Harrenstein 2009). When all Strongly Connected Components are known the Schwartz Set is easily found by forming the union of all undominated SCCs. In (Yan et al. 2014) an algorithm for computing the SCCs in pregel – the min-label algorithm – is presented. The CloudVoting algorithm for the Schwartz Set is an adaptation of the min-label algorithm. Since the min-label algorithm is a balanced practical pregel algorithm (BPPA), it follows that

the SchwartzSet algorithm is BPPA too, i.e. it takes only a logarithmic number of rounds and space usage, communication cost and computation cost are linear.

As input the strict dominance graph DG is given. Each vertex has as vertex id v_{id} and (min_f, min_b) as value, where min_f is the forward min-label and min_b the backward min-label.

- Initialize each vertex with $(min_f = v_{id}, min_b = v_{id})$.
- Min-Label foward propagation (pregel procedure):
  - SendMsg: Each vertex sends the min_f value in forward direction to the adjacent vertices
  - vertexProg: min_f is set to the minimum value of all received values and the old min_f
- Min-Label backward propagation (Pregel procedure): like the forward propagation but with the reverse edge direction
- Postprocessing:
  - remove all edges between vertices with different (min_f, min_b) labels
  - mark all vertices with $min_f < min_b$ as *not* in Schwartz Set
  - save for every vertex with $min_f == min_b$, that its SCC is found and named min_f.
  - if $min_f > min_b$, the vertex is still a candidate for the SchwartzSet, but we mark all other vertices having min_b as forward label as *not* in SchwartzSet
- The program stops as soon as the number of edges stays unchanged, i.e. no new SCCs have been found.

8 Experimental Evaluation

For proof of concept and preliminary experimental evaluation we use the Spotify world wide ranking dataset [4]. The dataset contains the top 200 rankings for every country of the world for each day over six months (January to June 2017). The original input data file has 230.7 MB but after conversion to the preflib .soi dataformat the resulting file has 8.5 MB. It contains 12 460 songs (i.e. candidates) and ∼2 Mio. rankings. Each ranking has length of at most 200. The resulting pairwise weighted graph has 12 460 vertices and ∼3.6 Mio edges. Saving this graph to disc takes 66.2 MB for the edges and 450 KB for the vertices.

In this Section we show the computation of the discussed voting rules on the spotify ranking data and make observations regarding the performance of the execution in Spark. The aim is not only to show how CloudVoting can be applied but point out what properties of Spark are of interest for efficient design of algorithms. The observations of this Section are meant to be used for future research concerning the processing of preference data in cloud based systems.

[4] `https://www.kaggle.com/edumucelli/spotifys-worldwide-daily-song-ranking/data`

8.1 Borda Score

The borda scores can be computed in CloudVoting by writing the following lines of code:

```
val bordascores = BordaFromSoc(file, sc)
val winner = Winner(bordascores)
```

The concept of lazy evaluation causes Spark to start with the computation when the result is needed. This is the case in the second line of code, when we actually determine the winner over the scored vertices.

Spark does a lot of optimization in the background. In Figure 1 the computation graph created by Spark for computing the borda scores is shown.

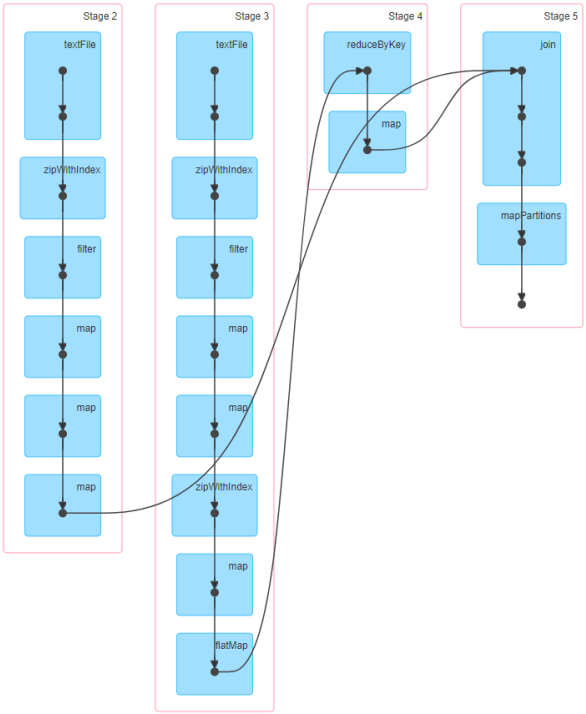

Figure 1: DAG - Execution Graph for Borda Scores

Spark splits the computation into different Stages at those points where shuffling is necessary. After each stage some of the data is saved to disc to be used later by a different stage. The amount of data saved during this Shuffle phase can be seen in Table 2 in the column `Write`. Respectively `Read` shows the size of the data that is read from preceding stages. This relationship between stages is also called ShuffleDependency (Karau and Warren 2017). Each stage can be executed without communication with other executors. It is usually advisable to design algorithms that don't take many stages.

Stages 2 and 3 both read the whole input file spotify.soi to transform (map) the read RDD. That the whole file is read can be observed in the column `Input` of Table 2.The output of Stage 3 is used as input to Stage 4. You can observe this in the DAG visualisation in Figure 1 and when looking at

ID	Descr.	Input	Read	Write
5	reduce		410.2 KB	
4	map		83.6 KB	99.7 KB
3	flatMap	8.5 MB		83.6 KB
2	map	8.5 MB		310.5 KB

Table 2: Stages of the Borda Score Computation

the size of Shuffle Read and Shuffle Write of the respective Stages in Table 2.

The borda winner of this dataset has a Borda score of 1 916 432 and is the song "Shape of You" by "Ed Sheeran".

The plurality score is computed in a very similar way as the borda score and the result is the same. The plurality score of our winner song is 3 396.

8.2 Copeland Score

To compute the copeland score, first the dominance graph has to be created from the preferences. The pairwise weighted graph is created while reading from the .soc file and then it is transformed to the weak dominance graph.

```
val pwg = readSoc("spotify.soi", sc)
val weakgraph = toWeak(pwg)
val copelandscores
    =CopelandScores(weakgraph, sc)
val winner = Winner(copelandscores)
```

The stages and the used data sizes are shown in Table 3. The input size of the file spotify.soi that contains all preferences is 8.5 MB, as you can see in the input column for Stages 3 and 5. Unfortunately the size does not stay at this level, since we have to compute all edges and their weights from the list of preferences. This large set of edges is created in Stage 6 and used as input to stage 7. In Table 3 it can be seen that this EdgeRDD is 161.2 MB large. This is huge in comparison to the original input data of size 8.5 MB. After stripping it of all edges that were produced several times by different votes, the remaining size of the EdgeRDD is 60.8 MB in memory.

ID	Descr.	Input	Read	Write
10	reduce	1455 KB	67.3 KB	
9	map	1455 KB	527.9 KB	67.3 KB
8	mapPart	60.8 MB		65.2 KB
7	mapPart	161 MB		50.1 KB
6	mapPart	81.1 MB	64.7 MB	64.9 KB
5	map	8.5 MB		297.7 KB
4	mapPart		64.7 MB	50.0 KB
3	flatMap	8.5 MB		64.7 MB
2	first	64.0 KB		
1	first	64.0 KB		
0	first	64.0 KB		

Table 3: Stages of the Copeland Score Computation

The Copeland Winner of this dataset is again "Shape of You" by "Ed Sheeran" with a Copeland Score of 11 876.

9 Conclusion

We reviewed existing cloud computing techniques and showed how they can be used to deal with large preference data. We introduced a tool called CloudVoting that can be used to work with preference datasets in a cloud computing environment. Some basic functionalities for dealing with preference data have been implemented and applied successfully. In Section 8 we applied basic winner determination rules to a real world dataset and made close observations of the behaviour of the system. We observed that a huge problem are potentially large intermediate results when computing e.g. the pairwise weighted graph from a set of preferences. We made observations about how the used memory increases during the computation and where the computation is split into stages. In the experimental section we gained insights to the underlying mechanisms of the system and can use this knowledge in future work to design distributed algorithms and implement more efficient methods for dealing with preference data.

We described how measures for efficient cloud computing algorithms are found and we gave some hints on where to tune the performance. The algorithm used to compute the Schwartzset has been designed to fit theoretical performance measures. In particular the SchwartzSet algorithm is BPPA (a balanced practical pregel algorithm), i.e. the number of rounds is logarithms and space usage, communication cost and computation cost are linear. It remains to apply such theoretical analysis to future social choice algorithms.

Future work includes designing high-performance algorithms for a larger variety of voting rules. Especially voting rules that are based on graphs are of great interest. Efficient distributed pregel algorithms for computing the smith set, schwartz set, schulze rule and other methods are yet to be found and investigated for their performance. Not only the design of distributed social choice algorithms is a future topic but also the enclosing of new datasets and application areas.

Acknowledgements This work was supported by the Vienna Science and Technology Fund (WWTF) through project ICT12-015, by the Austrian Science Fund projects (FWF):P25207-N23, (FWF):P25518-N23 and (FWF):Y698.

References

Afrati, F. N.; Sarma, A. D.; Salihoglu, S.; and Ullman, J. D. 2013. Upper and lower bounds on the cost of a map-reduce computation. *Proceedings of the VLDB Endowment* 6(4):277–288.

Brandt, F.; Brill, M.; and Harrenstein, P. 2016. Tournament solutions. In Brandt, F.; Conitzer, V.; Endriss, U.; Lang, J.; and Procaccia, A., eds., *Handbook of Computational Social Choice*. Cambridge University Press.

Brandt, F.; Chabin, G.; and Geist, C. 2015. Pnyx:: A powerful and user-friendly tool for preference aggregation. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, 1915–1916. In-

ternational Foundation for Autonomous Agents and Multiagent Systems.

Brandt, F.; Fischer, F.; and Harrenstein, P. 2009. The computational complexity of choice sets. *Mathematical Logic Quarterly* 55(4):444–459.

Charwat, G., and Pfandler, A. 2015. Democratix: A declarative approach to winner determination. In *International Conference on Algorithmic DecisionTheory*, 253–269. Springer.

Csar, T.; Lackner, M.; Pichler, R.; and Sallinger, E. 2017. Winner determination in huge elections with mapreduce. In *Proceedings of AAAI-17*. AAAI Press.

Dean, J., and Ghemawat, S. 2008. Mapreduce: simplified data processing on large clusters. *Communications of the ACM* 51(1):107–113.

Goldman, J., and Procaccia, A. D. 2015. Spliddit: Unleashing fair division algorithms. *ACM SIGecom Exchanges* 13(2):41–46.

Karau, H., and Warren, R. 2017. *HIgh Performance Spark*. " O'Reilly Media, Inc.".

Malewicz, G.; Austern, M. H.; Bik, A. J.; Dehnert, J. C.; Horn, I.; Leiser, N.; and Czajkowski, G. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 135–146. ACM.

Mattei, N., and Walsh, T. 2013. Preflib: A library for preferences http://www. preflib. org. In *International Conference on Algorithmic DecisionTheory*, 259–270. Springer.

org.apache.spark. 2017. Scala spark api documentation. `https://spark.apache.org/docs/latest/api/scala/index.html`.

Xin, R. S.; Gonzalez, J. E.; Franklin, M. J.; and Stoica, I. 2013. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, 2. ACM.

Yan, D.; Cheng, J.; Xing, K.; Lu, Y.; Ng, W.; and Bu, Y. 2014. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment* 7(14):1821–1832.

Zaharia, M.; Chowdhury, M.; Franklin, M. J.; Shenker, S.; and Stoica, I. 2010. Spark: Cluster computing with working sets. *HotCloud* 10(10-10):95.