Evaluating the Scaling of Graph–Algorithms for Big Data using GraphX

Jakob Smedegaard Andersen
Department of Computer Science
HAW Hamburg
Hamburg, Germany
Email: jakob.smedegaardandersen@haw-hamburg.de

Olaf Zukunft
Department of Computer Science
HAW Hamburg
Hamburg, Germany
Email: zukunft@acm.org

***Abstract*—Graph processing has achieved a lot of attention in different big data scenarios. In this paper, we present the design, implementation, and experimental evaluation of graph processing algorithms in two different application areas. First, we use semi-clustering as an example of an algorithm typically used social network analysis. Then, we examine an algorithm for collaborative filtering as typically used in E-Commerce scenarios. For both algorithms, we make use of Apache GraphX as an existing distributed graph processing framework based on Apache Spark. As GraphX does not include these two algorithms, we describe how to implement them using a combination of GraphX and the underlying Spark Core. Based on our implementation, we perform experiments to test the scalability of both the algorithms and the GraphX processing framework. The experiments show that different kinds of graph algorithms can be supported within the Spark framework. Furthermore, we show that for our test data the algorithms scale almost linearly when properly designed.**

***Keywords*-GraphX, Graph Processing, Semi-Clustering, Collaborative Filtering, Parallel Computing**

I. Introduction

The area of big data research has recently achieved a lot of attention. Although there does not exist a single definition of big data, most authors agree that systems that process data of high volume, high velocity, and high variety are typical for big data scenarios [1]. Sometimes, high veracity and high value are also mentioned in this context. Another research area with a much longer history in computer science are algorithms for graph structured data [2]. Application areas like network routing, website modelling or the study of disease spreading can be modeled as graph-theoretical problems. For some application areas, the combination of graph structured data and big data is obviously useful. Digital social networks are an popular example of big data systems [3], but they are also graph based systems. In the E-Commerce domain, recommendation systems and prediction systems for online shops also operate on high volume and high velocity data. Some of this data can also efficiently be modeled using graph oriented data structures as shown in the following.

To process graph structured data, there exist numerous different frameworks. For big data scenarios, a distributed implementation is required in order to cope with the high volume of the data. Examples of such frameworks include Apache Hama [4], which is built on top of Hadoop, and Gelly, a graph processing library built on top of Apache Flink [5]. In this paper, we use the Spark Framework [6] that is also provided by Apache. While Spark itself does not provide graph-processing capabilities, GraphX [7] is a layer on top of Spark Core and is openly available. Both share the same infrastructure features like robust distributed computations on a set of inexpensive computers and easy monitoring/deploying. GraphX additionally provides an appropriate API for graph algorithms that is not available in Spark core.

In this paper, we investigate how GraphX can be used to support typical tasks for social network analysis and E–Commerce in a scaling environment. First, we will shortly introduce property–graphs as our modeling primitive, the graph processing framework Pregel and GraphX in section II. Then, we demonstrate how to implement social graph analysis using semi–clustering using GraphX in section III. We also perform experiments to investigate the scalability of our solution. Afterwards, we demonstrate how to implement collaborative filtering using GraphX in section IV. Opposed to the analyzing and graph–preserving character of semi–clustering, collaborative filtering modifies the underlying graph and is therefore a representative of a different kind of graph algorithms. Here, we also perform experiments with different variants of the algorithm to investigate the scaling to big data volumes. Finally, we describe related work, give our conclusions and illustrate future work.

II. Preliminaries

A. Graphs and Property-graphs

In this section, we shortly introduce the notion of graphs and property–graphs as used in this paper. For detailed information see for example [2].

An undirected graph $G = (V, E)$ is defined as an ordered set of two disjoint, finite sets V and E. The elements of V are called *vertices*, those of E *edges*. An edge $e \in E$ is a connection of exactly two vertices and can be described as a 2–element subset of V.

A directed graph (V, E) is also defined as an ordered set of two disjoint, finite sets of vertices V and ordered edges $E \subseteq V \times V$. For a directed edge $e = (u, v) \in E$ we call $u \in V$ the startnode and $v \in V$ the endnode. By $N_{in}(v) :=$

978-1-5090-4054-4/16 $31.00 © 2016 IEEE
DOI 10.1109/OBD.2016.8

$\{u \in V | (u,v) \in E\}$ we denote the set of all neighbors of v, which v can reach via an incoming edge. Corresponding, we use $N_{out}(v) := \{u \in V | (v,u) \in E\}$ and $N(v) := N_{in} \cup N_{out}$ for outgoing and all edges. The degree of a vertex $d(v) := |N(v)|$ in a graph is its number of incident edges $v \in V$. We distinguish between the in–degree $d_{in}(v) := |N_{in}(v)|$ and out–degree $d_{out}(v) := |N_{out}(v)|$.

A property–graph $G(P) = (V, E, P)$ is a directed graph[1] and a collection of properties $P = (P_V, P_E)$. A property is assigned to every vertex and every edge. We do not assume any format of this property, i.e. it may be a weight, a state or some meta-data. The property of a vertex $v \in V$ is denoted by $P_V(v)$. The property of an edge $e \in E$ is denoted by $P_E(e)$. For a given property–graph $G(P)$, the set of properties may be changed through a transformation $f(P) \to P'$. The structure of the new property–graph $G(P')$ does not change.

B. Pregel

To efficiently operate on any property–graph $G(P)$, Pregel [8] offers a vertex-oriented iterative programming model. This model provides the programmer to apply an algorithm on any property–graph. For this, the programmer defines a vertex program Q. This program Q will subsequently be applied to every vertex in $G(P)$. Hence, the algorithm is written in a vertex-centered manner. Each iteration of applying Q is called a super-step and there is typically a sequence of super-steps to implement an algorithm.

Each super-step is composed of three sequentially processed phases [9] called *Gather*, *Apply* and *Scatter*. The three phases are associated with different functionality while processing a vertex v:

In the **Gather**-phase, all messages which were directed to v are aggregated to a single message using a *Message Combiner* $\otimes$. The operation $\otimes$ is both associative and commutative. For a message $M_{u,v}$, u is the sending node and v the receiving node.

$$M_\Sigma \leftarrow \bigotimes_{u \in N_{in}(v)} m(M_{u,v})$$

The **Apply**-phase computes the new property for the vertex v. For this, we can access the existing property of v and the message M_Σ as processed in the former *Gather*-phase. Any modification of the property is performed ahead of the next super-step.

$$P_V^{new}(v) \leftarrow a(P_V(v), M_\Sigma)$$

During the **Scatter**-phase, messages are sent to the neighbor nodes of v. These are determined by the new property of v, the properties of the edges and the properties of the end node.

$$\forall w \in N_{out}(v) :$$
$$M_{v,w} \leftarrow s(P_V^{new}(v), P_E((v,w)), P_V(w))$$

The execution model of Pregel is based on the *Bulk Synchronous Parallel (BSP)* processing [10]. Each vertex program can be considered a concurrent process that computes the next super-step. All vertex programs are synchronized as shown in figure 1. By this, Pregel makes sure that all properties have been updated and all messages been delivered before the next super-step.

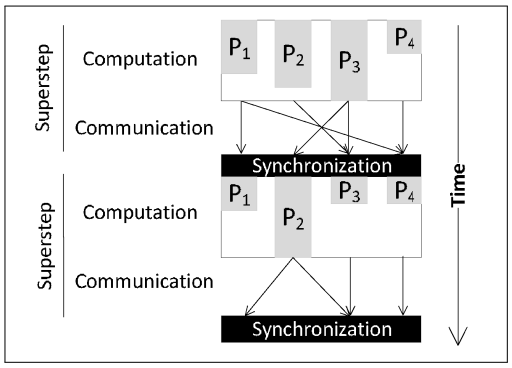


Figure 1: BSP Model: Each process P_i is synchronized at the end of each super-step [11]

C. GraphX

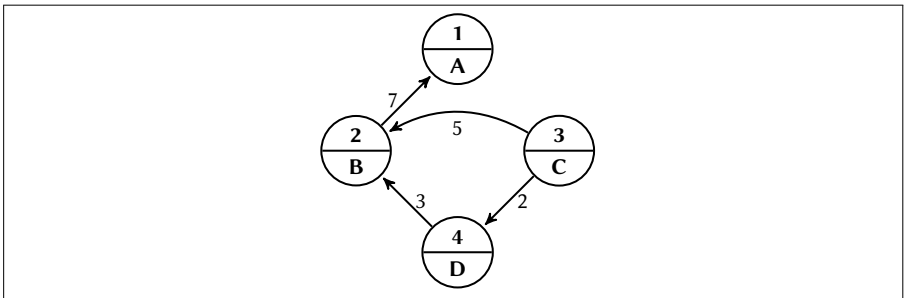
To use the Pregel approach of processing property–graphs, we either have to provide an implementation of our own or to use an existing implementation. The GraphX–System offers a platform to implement a graph-processing system using the underlying Spark [12] environment. It reuses the features of Spark Core, a component that offers basic functionality like scheduling, distribution of data and monitoring. The fundamental abstraction provided by Spark is the Resilient Distributed Dataset (RDD). An RDD ([13]) is a partitioned data structure that is to be processed by the computing primitives offered by Spark Core. Each Spark application can be deployed in a cluster. The application is running on a logically dedicated driver node and a typically high number of worker nodes that process tasks in parallel.

```
abstract class Graph[VD, ED] {
   val vertices: VertexRDD[VD]
   val edges: EdgeRDD[ED]
   val triplets: RDD[EdgeTriplet[VD, ED]]
   ...
}
```

Listing 1: Elements of graph abstraction

Inspired by and based upon the RDD–structure, GraphX ([14],[15]) offers a new layer of abstraction, the Resilient Distributed Graph (RDG). A property–graph can be represented by the RDG–abstraction. As each RDG is basically immutable, each transformation generates a new RDG. As GraphX is based upon Spark, it allows to process graph–structured data in a massive parallel way. Internally, an RDG is mapped onto a pair of RDD. One RDD represents the vertices and the other the edges of a given property–graph $G(P)$. On an abstract level, these two RDD can be considered as sets of value pairs $(i, P_V(i))$ and $((i,j), P_E((i,j)))$ for all vertices $i \in V$ and edges $(i,j) \in E$. Therefore, an RDG has direct access to the adjacency structure of a graph and to the attributes

[1] To denote multiple edges, we assume that E is a multiset.



Vertices

i	$P_V(i)$
1	A
2	B
3	C
4	D

Edges

(i,j)	$P_E((i,j))$
(2, 1)	7
(3, 2)	5
(3, 4)	2
(4, 2)	3

Triple

(i,j)	$(P_V(i), P_E((i,j)), P_V(j))$
(2, 1)	(B, 7, A)
(3, 2)	(C, 5, B)
(3, 4)	(C, 2, D)
(4, 2)	(D, 3, B)

Figure 2: Representation of property–graph using tables

associated with the edges and vertices as shown in figure 2. The abstraction offered to a programmer is shown in listing 1.

As a massive parallel execution environment, GraphX relies on a distribution of vertices and edges onto the different nodes in the computing cluster. For this distribution, different strategies can be used. Figure 3 shows the internal

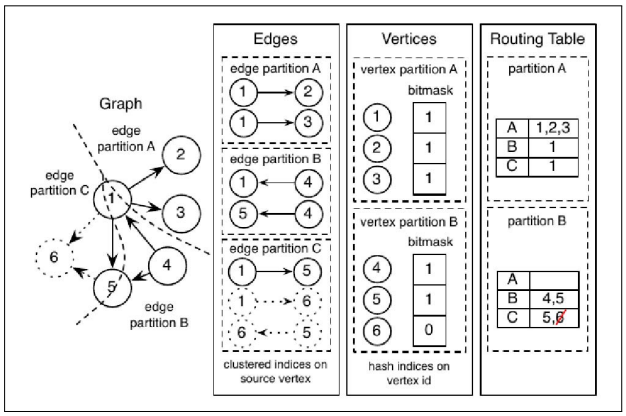


Figure 3: Internal representation of a three-partition graph [7]

representation of a graph that has been partitioned using a vertex-oriented cut.

- The **Edges** table implements the adjacency matrix. Each edge is assigned to a Partition using its PartitionsId. For an edge, the startnode, endnode and the properties of the edge are stored.
- The partitioned **Vertices** store the id and the properties of each vertex.
- The **Routing Table** is an associative data array. For the identification of vertices, the PartitionsId of the adjacent edges is stored. This table is co–partitioned with vertices.

III. Analysis of social graphs using semi–clustering

Digital social networks (DSN) like facebook can be modeled and analyzed by graph based algorithms. A typical analytic task is to find a group of individuals in a DSN that is highly related in some application specific sense. In the following, we give a graph–oriented description of this task. Then, we describe the design and implementation of our algorithm solving this task. Finally, we evaluate the scaling of our approach based on experimental work. An earlier version of this approach can be found in [16].

A. Graph model of semi-clustering

When modeling the data of DSN, we use vertices to denote persons and edges to denote any relationship between persons. Edges may be created by any interaction between persons like adding some person as a friend, subscribing to his/her channel or exchanging messages. Edges may have a weight describing the amount of interaction between persons. The graph is undirected.

We want to use this kind of graph to find a number of clusters, i.e. a group of persons that have a strong connection with the people in the same cluster and a weak connection to people outside of the cluster. This problem has been well studied ([17]). For our work, we adopt the semi-cluster approach introduced in [8].

Using this semi-clustering, a cluster is defined as a set of vertices and an additional property S, that allows to compare clusters. For a given cluster c, S_c is defined as:

$$S_c = \frac{I_c - f_B B_c}{V_c(V_c - 1)/2}$$

with I_c the sum of all weights of all edges with startnode and endnode in the cluster c. B_c is the sum of all weights of all edges that have either a startnode or endnode outside of the cluster. f_B assigns a weight to B_c and is assumed to be in the interval $[0, 1]$. Finally, V_c is the number of vertices in c.

The value S_c of a cluster is normalized through the number of vertices within a clique of size V_c. This is necessary to prevent clusters with a high number of vertices of raising their value. For an undirected graph $G = (V, E)$, a clique $C = (V_C, E_C)$ with $n = |V_C|$ is a subgraph of G so that every two vertices in V_C are neighbors [2]. Therefore, $E_C = \mathcal{P}_2(V_C)$. Hence, for the divisor of S we get:

$$|E_C| = \binom{n}{2} = \frac{n!}{2(n-2)!} = \frac{n(n-1)}{2}$$

A typical result of semi-clustering in shown in figure 4.

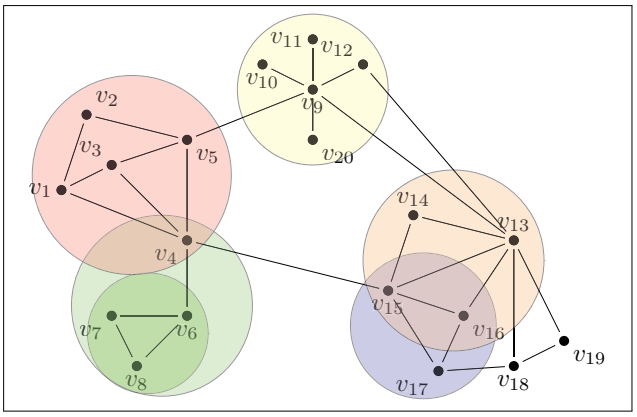


Figure 4: Example result of our semi-clustering algorithm [16]

The original algorithm as outlined in [8] cannot be used directly in GraphX. First, it operates on an undirected graph. Property–graphs as used in GraphX are always directed. One obvious approach would be to use two directed edges for each undirected graph. However, this is not used in our design due to performance concerns. Instead, we model each undirected edge through one directed edge and make use of the GraphX–feature to traverse each directed edge in both directions. This results in half the number of edges when compared to the original proposal. This reduction is largely significant for big data volumes.

Additionally, we restructure the original graph during preprocessing to reduce the number of edges between any given set of two vertices to at most one. For this, we add up the weights of all edges between their two vertices and replace these edges with a single new edge having the calculated sum as a new weight. This is of course an application specific optimization as the structure of the graph is changed, but for our application the result remains the same as without the optimization. We also eliminate all loops as they are not relevant for semi–clustering. An example of this preprocessing is shown in picture 5.

1) Design and Implementation of semi–clustering: Our scalable implementation of semi–clustering is based on Pregel and uses GraphX. It is currently implemented in Scala as

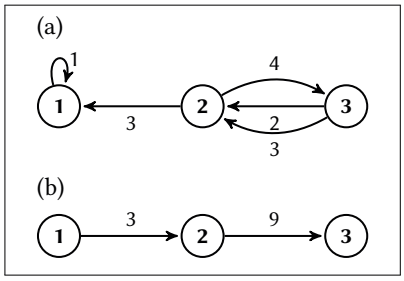


Figure 5: A social graph (a) and the result after our preprocessing step (b)

this is a straightforward choice for GraphX.[2] Our design is composed of four classes.

1) We have decided to create a dedicated class *SemiCluster* representing the list of clusters that is computed during each super-step. Each cluster encapsulates its list of contained vertices referenced by their ID and their weight S. As clusters only grow and never shrink, they only have a addVertex method.
2) The attribute $semiScores$ is stored in a separate class. It stores the score and is cached in each cluster to prevent the inefficient re-computation from scratch after adding a vertex.
3) The class SemiVertexData handles incoming messages as basic in the pregel approach.
4) The class SemiOutEdge represents the outgoing edges of one vertex and stores the value associated with that edge.

The algorithm for the re-computation of the SemiScore s is shown in listing 2. The computation starts with an initial message which contains the empty list. Afterwards, the vprog–function is executed on each vertex by access the messages directed to the vertex. This message contains the list of clusters which are iteratively processed by adding the vertex itself.

```
def recalculate(vertexSet: List[VertexId],
        outGoingEdges: Array[SemiOutEdge],
        scorFactor: Double): SemiScore = {
   if (vertexSet.size == 1) {
      SemiScore(weightInnerEdges,outGoingEdges.
          aggregate(0d)(_ + _.attr, _ + _), 1)
   } else {
     var wbe: Double = weightBoundetEdges
     var wie: Double = weightInnerEdges
     for (edge <- outGoingEdges) {
       if (vertexSet.contains(edge.id)) {
             wbe -= edge.attr
             wie += edge.attr
         } else {wbe += edge.attr} }
     SemiScore(wie, wbe,
         (wie - scorFactor * wbe) /
         ((vertexSet.size *
          (vertexSet.size - 1)) >> 1))
    } }
```

Listing 2: Function Recalculation of cluster weights

[2]The source code of our system is available upon request.

B. Experiment: Semi-Clustering

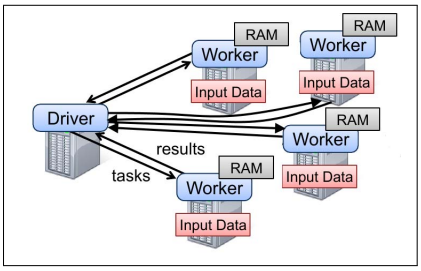


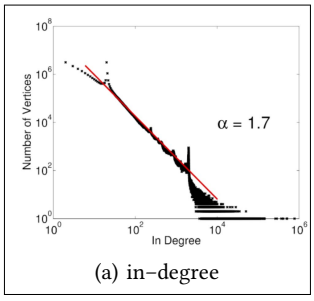
Figure 6: Spark Cluster

To validate the assumption that GraphX provides a scalable environment for appropriate graph algorithms we have performed some experiments. Our environment consists of a Spark/GraphX 1.5.1. cluster as depicted in figure 6 using the stand alone cluster manager and between one and four worker nodes. All worker nodes are dedicated single–core machines (1GB RAM) with an additional and three–core driver node (8 GB RAM). The graph is partitioned by a random cut among all worker nodes.

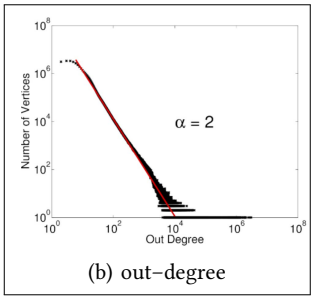
Regarding the graph to be processed, the probability of a vertex u to have a degree k is as follows:

$$Pr[d(u) = k] \sim k^{-\alpha}, \qquad \alpha > 0$$

The exponent α controls the asymmetry of the distribution. The bigger α, the more vertices have a lower degree. For graphs modeling social networks, a value of $\alpha \approx 2$ is assumed [9]. Figure 7 shows the distribution of vertices w.r.t. their in- and out–degree of twitter. Obviously, there is no symmetric distribution of neighboring vertices for different twitter users. With this distribution in mind, we have designed our test



(a) in–degree



(b) out–degree

Figure 7: Distribution of in–degree (a) and out–degree (b) for twitter [9] as an example for social networks

dataset as shown in figure I

For this test dataset, figure 8 shows the scaling of our algorithm for an increasing graph size (with constant number of workers). In this experiment, the number of vertices varies between 1000 and 9000. As it can be seen in this figure, the algorithm scales linearly with increasing graph sizes.

Table II shows the runtime for varying numbers of workers. The results have been achieved for a constant graph structure

$\|V\|$	$\|E\|$	$\|E\|/\|V\|$
1000	81.665	81,67
2000	337.366	168,68
3000	600.573	200,29
4000	813.018	203,25
5000	973.846	194,77
6000	1.099.045	183,17
7000	1.188.798	169,82
8000	1.256.259	155,78
9000	1.421.884	157,99

Table I: Size and characteristics of test graph used in experiments

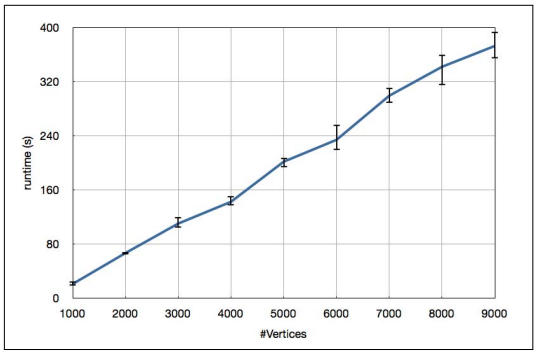


Figure 8: Experiment Semi-Clustering: Scaling the graph size

of 3000 vertices. As it can be seen, the observed runtime decreases with increasing numbers of workers as expected. However, we do not observe a linear scaling. For this, we would require the observed scaling factor (defined as 215,84 / observed runtime) to be equal to the number of workers. Even for four workers, we do not yet get a scaling factor of 3 which would be expected for three workers in an implementation that scales linearly. This behaviour is due to the overhead associated with the partitioned graph structure. As long as a distribution of messages between different workers is performed, no linear scaling can be expected.

Number of Workers	Observed runtime in s	Expected runtime with linear scaling	Observed scaling factor
1	215,84	215,84	1
2	132,41	107,92	1,63
3	91.65	71,95	2,355
4	79,98	53,96	2,699

Table II: Semi-Clustering: Scaling the number of Worker

IV. Collaborative Filtering

Our second case–study uses the concept of collaborative filtering. One motivation behind collaborative filtering is to predict product item purchases of specific users based on existing purchases of other users.

1) Graph–model of collaborative filtering: For collaborative filtering, we model users and product items as a bipartite graph G. For $G = (V, E)$, the set of vertices V is partitioned into users U and items I. We assume $V = U \cup I$ and $U \cap I = \emptyset$. An edge of G always connects a user $u \in U$ with an item $i \in I$. An example of such a graph is given in figure 9. For a given graph $G = (V, E_{old})$ at time t, we want to predict the structure of a modified $G = (V, E_{new})$ at t' with $t < t'$ assuming that $E_{new} \subseteq V \times V$ and $E_{old} \cap E_{new} = \emptyset$. This can be considered a variant of link–prediction [18].

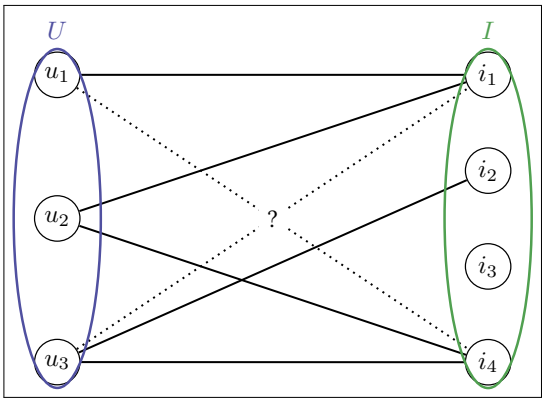


Figure 9: Example: Bipartite graph for collaborative filtering

In order to predict the new edges in G, we assign a weight to all non–existing edges in G. The higher the weight, the more probable is a future purchase. To compute the weight, several score–functions can be used. We implement four different approaches for this scoring functions that we adapted from [18] for bipartite graphs. For an item i, we define $\hat{N}(i) = \bigcup_{c \in N(i)} N(c)$ as the set of neighbors of neighbors of i (with $N(c)$ as the set of neighbors of c).[3]

Common Neighbors We define the score as the overlap of the interest of user u with other users who also purchased i.

$$score(u, i) := |N(u) \cap \hat{N}(i)|$$

Jaccard's Coefficient We define the score as the **Common Neighbors**-value divided by the number of all elements. This limits the results to a value in the interval $[0, 1]$.

$$score(u, i) := \frac{|N(u) \cap \hat{N}(i)|}{|N(u) \cup \hat{N}(i)|}$$

Preferential Attachment We define the score as the product of activity of users and attractivity of items.

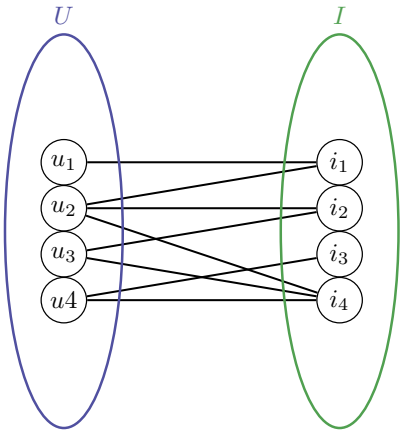
$$score(u, i) := d(u) \cdot d(i)$$

Adamic/Adar Opposed to **Common Neighbors**, we focus on the number of neighbors instead of the number of users/items.

$$score(u, i) := \sum_{c \in N(u) \cap \hat{N}(i)} \frac{1}{\log d(c)}$$

To illustrate these different functions, we demonstrate the different scores computed by the four functions in the example shown in figure 10 The bold elements show the most probable future purchases of an item i_x for a user u_y.

[3]This modifies the definition given in [18] in order to capture neighbors that are not connected to the same items.



	Common Neighbors	Jaccard's Coefficient	Preferential Attachment	Adamic Adar
(u_1, i_2)	1	0,33	2	3,32
(u_1, i_3)	0	0	1	0
(u_1, i_4)	1	0,25	3	3,32
(u_2, i_3)	1	0,25	3	2,10
(u_3, i_1)	**2**	**0,67**	**4**	**5,42**
(u_3, i_3)	1	0,33	2	2,10
(u_4, i_1)	1	0,25	**4**	2,10
(u_4, i_2)	1	0,25	**4**	2,10

Figure 10: Example: Different scores for a given graph

2) Design and Implementation of Collaborative Filtering: One major difference between Semi–Clustering and Collaborative Filtering is the fact that collaborative filtering is modifying the underlying graph G. GraphX does not provide direct support for the computation of missing edges between the two partitions in G. Also, we can not use Pregel for this. Nevertheless, since GraphX allows direct access to the underlying RDD, we decided to implement this step by direct access to RDD and to compute the candidate set of new edges $E_{new} := (U \times I) \setminus E_{old}$ using transformations on RDD. Afterwards, all elements $e_{new} \in E_{new}$ are given a score through each of the functions described above. The score is stored as a property of the edge.

From a design perspective, we distinguish between three categories describing what kind of access to graph elements is required for computing the score:

(i) Each vertex only knows the number of its neighbors. This kind of function can be implemented using the degree–attributes of RDG. The Preferential Attachment function is in this category.

(ii) Each user–vertex knows the ID of its neighbors. Each item–vertex knows the ID of all of the neighbors of its neighbors. To find the properties of items, we use the exchange of messages via edges. The functions Common Neighbors and Jaccard's Coefficient are in this category.

(iii) This category is based on the number of neighbors of the union set of the items described in (ii). We decided to compute both the properties of the users and of the items within the same RDG. Afterwards, we aggregate them within the connecting edge. The Adamic/Adar function is in this category.

A. Experiment: Collaborative Filtering

We have performed different experiments to test the scalability of collaborative filtering with different score–functions using GraphX. The infrastructure we used is the same as above; we deploy between one and four equally equipped worker nodes and additionally one dedicated driver. Our test data sets are bipartite graphs with a constant number of items and a variable number of users. For a start, we have created 750 items. The number of users varies between 100 and 600. The edges follow a Gaussian distribution with mean $\mu = 45$ and standard deviation of $\sigma = 15$.

The runtime of collaborative filtering with four scoring–functions are compared in Figure 11 for a variable number of users. From this figure, we can see that the Adamic/Adar scoring–function shows an exponential behavior and does not scale to big data scenarios. In contrast, the other three scoring–functions show a linear runtime behavior.

In figure 12, we show the effect of scaling by increasing the number of workers. The number of users is kept constant at 9000. All scaling functions show a nonlinear decrease in runtime after increasing the number of workers. The scaling functions Common Neighbors and Jaccard's Coefficient show a similar behavior. This can be expected as they are in the same category. The scaling function Preferential Attachment shows a lower runtime. This is due to the fact that the function only considers direct neighbors and this can be computed by a single join–operation in Spark. The Adamic/Adar–function has not been computable on the given data set due to memory constraints.

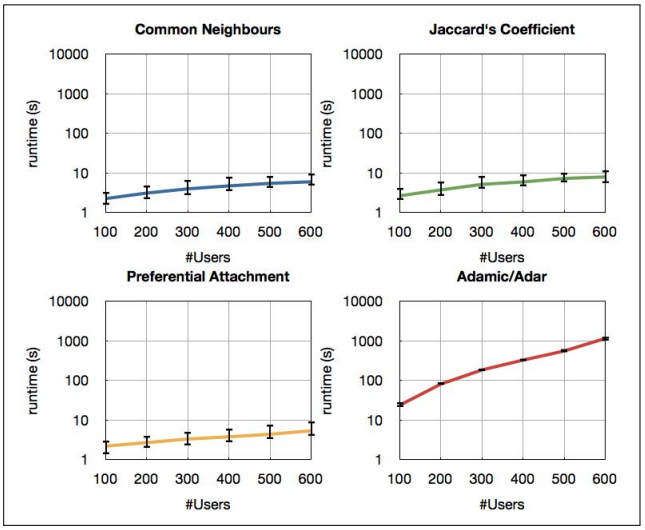


Figure 11: Experiment collaborative filtering: Scaling of users

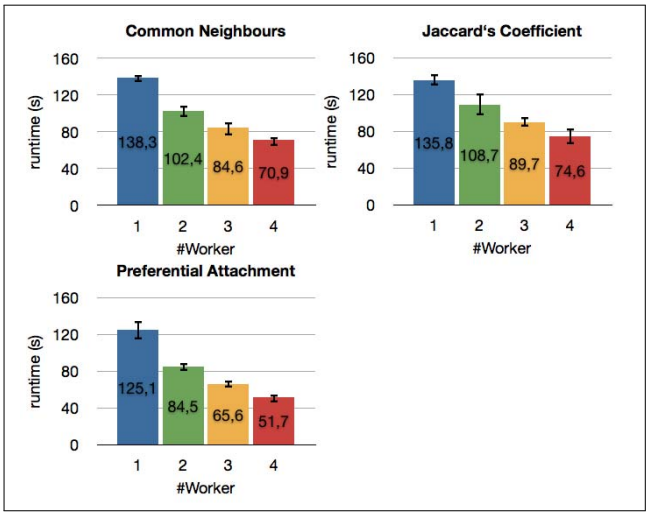


Figure 12: Experiment collaborative filtering: Scaling of Worker

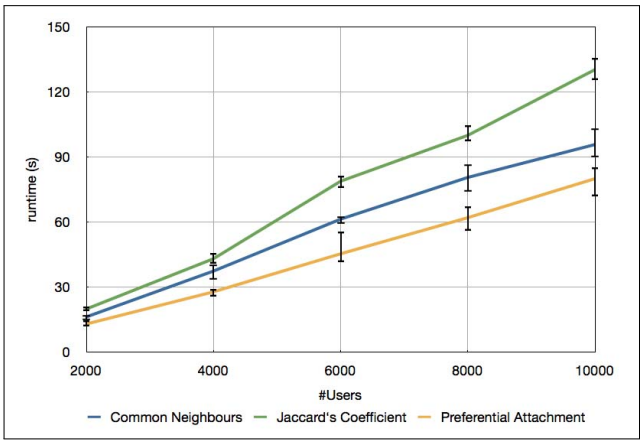


Figure 13: Experiment collaborative filtering: Linear Scaling for three different scoring functions

In figure 13, it can be seen that the three metrics *Common Neighbours*, *Jaccard's Coefficient*, and *Preferential Attachment* show a linear scaling for increasing numbers of users.

V. Related Work

The GraphX framework [7] provides a basis for our work and comes with a number of already implemented algorithms. However, none of the algorithms we introduced in this paper is available in GraphX. The Pregel system [8] provides a vertex–centric processing paradigm for graph algorithms. Nevertheless, the Pregel system itself is not openly available. Hence, a variant has been implemented in GraphX and is used in our implementation. Also, we adapt the semi–clustering algorithm from [8] with modifications to reduce the runtime. The algorithm for collaborative filtering on graph structured data follows the basic idea outlined in [18]. Opposed to the work presented there, we have proposed a

parallel implementation and performed a scalability–study. An alternative platform for running Pregel is Giraph [19], but it only provides an execution environment for Pregel and neither graph algorithms like semi–clustering nor access to the underlying graph structure as required for collaborative filtering.

In [11], the authors describe the scalability of graph–pattern matching algorithms. The idea to find scalable algorithms is related to our work, but they use different algorithms, graphs and execution platforms. The scalability–study described in [20] is quite similar to our work with regard to the basic idea. However, they use different algorithms, different graphs (not property–graphs) and do neither use Spark nor GraphX. Neo4j as a graph database also uses property–graphs, however as a database system it does not support the efficient execution of complex algorithms in parallel. Finally, the Spark MLLib [21] is also related to our work as they provide algorithms for machine learning like clustering on top of Spark. Yet, Spark MLlib does not operate on graphs as a data structure and neither provide semi–clustering nor collaborative filtering on graph structures.

VI. Conclusions

Our experiments have shown that GraphX is able to support both graph preserving and graph modifying algorithms. The offered API of Pregel is well suited for semi–clustering as an algorithm that preserves the processed graph. However, some modifications of the original algorithm have to be performed in order to use the specific Pregel–implementation of GraphX. The semi–clustering algorithm scales almost linearly with increasing graph sizes. As the runtime also decreases with the addition of workers it promises to be wells suited for big data scenarios.

The Pregel model of computation is not well suited for the realization of collaborative filtering as this algorithm modifies the graph during execution. However, using the lower level primitives of GraphX (RDG and RDD) we were able to implement this algorithm with different scoring–functions easier than on the Spark–core. We found that the scalability of collaborative filtering is dependent on the scoring–function. While three of them have shown acceptable (i.e. linear) scaling, the Adamic/Adar function has shown exponential runtime and is therefore not recommended for big data scenarios. This is the important when designing the algorithm.

For future work, we will focus on additional experiments. Currently, the number of workers is rather low. We will have to increase the computing power and memory of the underlying infrastructure to perform more realistic experiments. Here, we will especially use machines with more RAM as Spark core does rely on main memory to perform faster than for example Hadoop–based solutions. Also, we would like to increase not only the size of the test graph but also to increase the workload by performing a parallel set of tasks. Finally, we would like to compare the current GraphX–based solution to other frameworks for distributed graph processing and to compare our solution with a very well equipped single computer to determine a threshold for going distributed versus centralized computations.

References

[1] S. Sharma, U. S. Tim, J. Wong, S. Gadia, and S. Sharma, "A brief review on leading big data models," *Data Science Journal*, vol. 13, no. 0, pp. 138–157, 2014.

[2] R. Diestel, *Graph Theory*, 4th ed. Heidelberg: Springer, 2010.

[3] A. Silberstein, A. Machanavajjhala, and R. Ramakrishnan, "Feed following: The big data challenge in social applications," in *Databases and Social Networks*, ser. DBSocial '11. New York, NY, USA: ACM, 2011, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/1996413.1996414

[4] Apache, *Hama*. https://hama.apache.org, 2016.

[5] ——, *Flink*. https://flink.apache.org, 2016.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[7] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 599–613.

[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. of the 2010 ACM SIGMOD Intl. Conf. on Management of Data*, ser. SIGMOD'10. New York, NY, USA: ACM, 2010, pp. 135–146.

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 17–30.

[10] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[11] M. U. Nisar, A. Fard, and J. A. Miller, "Techniques for graph analytics on big data," in *Proc. of the 2013 IEEE Intl. Congress on Big Data*, ser. BIGDATACONGRESS'13. Athens, GA, USA: IEEE Computer Society, 2013, pp. 255–262.

[12] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark: Lightning-Fast Big Data Analysis*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2015.

[13] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2.

[14] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *First International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES'13. New York, NY, USA: ACM, 2013, pp. 2:1–2:6.

[15] R. S. Xin, D. Crankshaw, A. Dave, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: Unifying Data-Parallel and Graph-Parallel Analytics," *ArXiv e-prints*, 2014.

[16] J. S. Andersen and O. Zukunft, "Semi–clusering that scales: An empirical evaluation of graphx," in *Proceedings of the 2016 IEEE Big Data Congress*. San Francisco, USA: IEEE Computer Society, 2016.

[17] J. Scott, *Social network analysis*. Sage, 2012.

[18] Z. Huang, X. Li, and H. Chen, "Link prediction approach to collaborative filtering," in *Proc. of the 5th ACM/IEEE-CS Joint Conf. on Digital Libraries*, ser. JCDL'05. New York, NY, USA: ACM, 2005, pp. 141–142.

[19] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[20] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ser. ICDM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 229–238. [Online]. Available: http://dx.doi.org/10.1109/ICDM.2009.14

[21] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "Mllib: Machine learning in apache spark," *CoRR*, vol. abs/1505.06807, 2015. [Online]. Available: http://arxiv.org/abs/1505.06807