Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Jakob Smedegaard Andersen

Funktionale Erweiterung des GraphX Frameworks

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Jakob Smedegaard Andersen

Funktionale Erweiterung des GraphX Frameworks

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Wirtschaftsinformatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Zukunft
Zweitgutachter: Prof. Dr. Steffens

Eingereicht am: 18.12.2015

Jakob Smedegaard Andersen

Thema der Arbeit
Funktionale Erweiterung des GraphX Frameworks

Stichworte
GraphX, Graphverarbeitung, Implementierung, Semi-Clustering, kollaboratives Filtern

Kurzzusammenfassung
Die Arbeit untersucht die beiden Fragen, ob GraphX eine geeignete Umgebung für die Realisierung von Graphalgorithmen darstellt und wie die Berechnung der Graphalgorithmen skaliert. GraphX ist ein Framework für die verteilte Verarbeitung von Graphen. Es wird der Implementierungsprozess beginnend mit einer abstrakten Formulierung bis hin zu einer lauffähigen Implementierung veranschaulicht. Experimentelle Untersuchungen weisen darauf hin, dass verschiedene Graphalgorithmen in GraphX abgebildet werden können und diese fast linear skalieren.
Jakob Smedegaard Andersen

Title of the paper
A Functional Extension of the GraphX Framework

Keywords
GraphX, graph processing, implementation, Semi-Clustering, collaborative filtering

Abstract
The work investigates the two questions, whether GraphX is a suitable environment for the implementation of graph algorithms and how the computation of graph algorithms scales. GraphX is a distributed graph processing framework. The work illustrates the implementation process beginning with a conceptual formulation up to an executable program. Experimental investigations provide evidence that different graph algorithms can be imaged in GraphX and that these scale nearly linear.

Inhaltsverzeichnis

Symbolverzeichnis

E	Kantenmenge
G	Graph
$G(P)$	Property-Graph
M	Typ einer Nachricht
$N(v)$	Menge der Nachbarknoten von v
$N_{in}(v)$	Menge der Nachbarknoten, die v üben eingehende Kanten erreicht
$N_{out}(v)$	Menge der Nachbarknoten, die v über ausgehende Kanten erreicht
$P_E(e)$	Property der Kante e
$P_V(v)$	Property des Knotens v
$Q(v)$	Vertexprogramm-Instanz, die auf dem Knoten v ausgeführt wird
S_n	n-te Superstep
V	Knotenmenge
$\mathcal{P}(M)$	Potenzmenge der Menge M
$\mathcal{P}_2(M)$	2-elementige Potenzmenge der Menge M
e	Kante
u, v	Knoten

Abbildungsverzeichnis

Listings

1. Einleitung

Immer mehr reale Problemstellungen basieren auf der Analyse von netzwerkartigen Strukturen. Die Bedeutung dieser Probleme ist insbesondere in aktuellen Fortschritten im *maschinellen Lernen* und *Data-Mining* groß (vgl. Xin u. a., 2013). Um die entstehenden Aufgaben zu lösen, müssen spezielle Algorithmen entwickelt werden. Der Beitrag diese Arbeit liegt darin, genau solche zu implementieren. Die steigende Relevanz der Problemfelder und die steigenden Datenmengen haben den Bedarf an einfachen Werkzeugen für die Anwendung und Entwicklung von Algorithmen auf netzwerkartigen Strukturen erhöht (vgl. Xin u. a., 2013). Dieses Interesse hat zu der Entwicklung einer Vielzahl von Verarbeitungs-Frameworks geführt. Mit diesen wird die Möglichkeit verfolgt, eine effiziente parallele Verarbeitung zu ermöglichen. Die Systeme sind in der Lage iterative Algorithmen auszuführen (vgl. Xin u. a., 2014). Die Vielfältigkeit der zu lösenden Problemstellungen fordern von den einzelnen Frameworks eine vielseitige Anwendbarkeit. Betrachtungspunkt dieser Ausarbeitung ist das GraphX Framework, da es sich bei diesem um ein relativ neues Forschungsfeld handelt.

Ziel dieser Arbeit ist es, eine auf dem GraphX Framework aufbauende Implementierung von Algorithmen durchzuführen und die Laufzeiten anhand von Experimenten zu bewerten. Dabei soll der Weg von einer abstrakten Formulierung über ein grobes Realisierungskonzept bis hin zu einer lauffähigen Implementierung veranschaulicht werden. Die Angemessenheit und Eignung der zur Verfügung gestellten Operationen stellt eine bedeutende Rolle dar. Eventuelle Problematiken sollen aufgedeckt werden.

Die Arbeit ist in drei Bereiche unterteilt. In Kapitel 2 werden grundlegende Aspekte eingeführt, die in späteren Kapiteln eine fundamentale Rolle einnehmen. Hierzu gehören eine Einführung in graphentheoretische Grundlagen und ein Überblick über Spark, einem Framework für parallele Datenverarbeitung.

Kapitel 3 befasst sich mit GraphX, einem Framework für die parallele Verarbeitung von Graphstrukturen. Im Mittelpunkt stehen hier die Darstellung eines Graphen und des damit verbunden Verarbeitungskonzeptes sowie die Verteilung der zugrundeliegenden Daten. Auch wird auf die von dem GraphX Framework bereitgestellten Operationen eingegangen.

Kapitel 4 widmet sich anschließend der Realisierung zweier Graphalgorithmen, aufbauend auf dem GraphX Framework. Zunächst wird sich umfassend mit dem jeweiligen Konzept beschäftigt. Es folgt die Entwicklung und Implementierung einer Umsetzung mit anschließender experimenteller Laufzeitbetrachtung.

Letztlich in Kapitel 5 eine Zusammenfassung und ein Ausblick.

2. Grundlagen für GraphX

In diesem Kapitel werden grundlegende Aspekte, auf denen das GraphX Framework aufbaut, vorgestellt. Es werden Graphen als Strukturierung von Daten eingeführt. Es folgt ein Einblick über *Spark*, einem Framework für parallele Datenverarbeitung, auf dessen Schnittstellen GraphX implementiert wurde.

2.1. Graphen

Das GraphX Framework operiert auf der Struktur eines Graphen. Mit diesen ist eine entsprechend große Relevanz verbunden. In Abschnitt 2.1.1 wird das mathematische Konzept eines Graphen verdeutlicht. Abschnitt 2.1.2 betrachtet daraufhin Graphen aus der Sicht der Verarbeitung von Informationen.

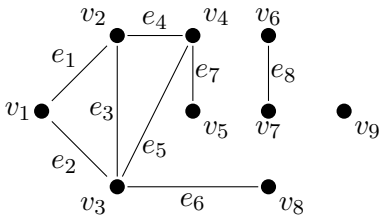
2.1.1. Begrifflichkeiten

In diesem Absatz wird die abstrakte Struktur eines Graphen grundlegend veranschaulicht. Auch auf häufig vorkommende Begrifflichkeiten im Kontext von Graphen wird eingegangen. Diese kurze Darstellung aus der Graphentheorie beschränkt sich lediglich auf die Mittel, die unmittelbar im Rahmen dieser Ausarbeitung relevant sind. Für weiterführende Inhalte sei auf Diestel (2010) verwiesen. Diestel (2010) wurde auch als Grundlage für die formale Darstellung herangezogen. Ein Graph ist eine abstrakte Struktur, die Objekte und deren Verbindungen zueinander repräsentiert. Für Graphen gibt es verschiedene Formen, die je nach Anwendungsgebiet Verwendung finden.

Ein **(ungerichteter) Graph** $G = (V, E)$ wird durch ein geordnetes Paar disjunkter, endlicher Mengen V und E beschrieben. Die Elemente von V werden als *Knoten*, die von E als *Kanten* bezeichnet. Eine Kante $e \in E$ ist eine Verbindung zweier Knoten und wird als 2-elementige Teilmenge von V beschrieben. Entsprechend ist $E \subseteq \mathcal{P}_2(V) \coloneqq \{M \subseteq V || M| = 2\}$.

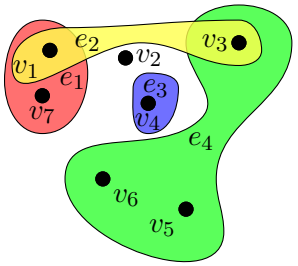
Ein **Hypergraph** ist ein generalisierter Graph $G = (V, E)$ mit der endlichen Knotenmenge V und Kantenmenge E. Zudem gilt für die Kantenmenge E, dass diese eine Teilmenge der Potenzmenge $\mathcal{P}(V) \coloneqq \{M | M \subseteq V\}$ ist. Eine Kante kann entsprechend beliebig viele Knoten

umfassen. Der Unterschied zwischen einem (ungerichteten) Graph und einem Hypergraph wird in Abbildung 2.1 verdeutlicht.



(a)

$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$
$e_1 = \{v_1, v_2\}, e_2 = \{v_1, v_3\},$
$e_3 = \{v_2, v_3\}, e_4 = \{v_2, v_4\},$
$e_5 = \{v_3, v_4\}, e_6 = \{v_3, v_8\},$
$e_7 = \{v_4, v_5\}, e_8 = \{v_6, v_7\}$



(b)

$E = \{e_1, e_2, e_3, e_4\}$
$e_1 = \{v_1, v_7\}, e_2 = \{v_1, v_3\},$
$e_3 = \{v_4\}, e_4 = \{v_4, v_5, v_6\}$

Abbildung 2.1.: Ein (ungerichteter) Graph (a) und ein Hypergraph (b) inklusive einer formalen Darstellung derer Kanten

Für die Darstellungen netzwerkartiger Strukturen sind diese Modelle nur bedingt geeignet. Entitäten lassen sich als Knoten und ihre Beziehungen zueinander als Kanten ausdrücken. Die Grenzen dieser Modelle sind jedoch erreicht, sobald Beziehungen gerichtet dargestellt werden sollen. Wie sie beispielsweise bei einer Repräsentation der Hyperlink-Beziehungen zwischen Webseiten oder Transportrouten im Straßenverkehr vorkommen. Einige Formen von Graphen sehen auf Grund dieser Notwendigkeit auch eine Modellierung von gerichteten Kanten vor.

Ein **(gerichteter) Graph** wird durch ein geordnetes Paar (V, E) disjunkter, endlicher Mengen V von Knoten und gerichteten Kanten $E \subseteq V \times V$ beschrieben. Für eine gerichtete Kante $e = (u, v) \in E$ bezeichnet man $u \in V$ als Anfangs- und $v \in V$ als Endknoten. Grafisch werden gerichtete Kanten durch einen Pfeil in Richtung des Endknotens dargestellt. Mit $N_{in}(v) \coloneqq \{u \in V | (u, v) \in E\}$ wird die Menge aller Nachbarn von v beschrieben, die v über eine eingehende Kante erreichen kann. Analog kommen $N_{out}(v) \coloneqq \{u \in V | (v, u) \in E\}$ und $N(v) \coloneqq N_{in} \cup N_{out}$ hinzu. Als *Grad* eines Knotens $d(v) \coloneqq |N(v)|$ wird die Anzahl der mit $v \in V$ in Verbindung stehenden Kanten bezeichnet. Dieser lässt sich in Eingangsgrad $d_{in}(v) \coloneqq |N_{in}(v)|$ und Ausgangsgrad $d_{out}(v) \coloneqq |N_{out}(v)|$ untergliedern.

Zudem sei auf einen **Multigraph** verwiesen, ein Quadrupel $(V, E, init, ter)$ bestehend aus den zwei disjunkten Mengen von Knoten V und Kanten E und zwei Funktionen $init : E \to V$ und $ter : E \to V$. Jeder Kante $e \in E$ wird durch $init(e)$ seinen Anfangs- und durch $ter(e)$ seinen Endknoten zugewiesen. Im Vergleich zu einem gerichteten Graph kann ein Multigraph

auch mehrere gleich gerichtete Kanten zwischen zwei Knoten besitzen. Wenn für eine Kante e gilt $init(e) = ter(e)$, bildet e eine Schlinge.

2.1.2. Verarbeitung

Die Verarbeitung und Analyse von Graphen hat sich in vielen Bereichen als nützliches Instrument erwiesen, um versteckte Muster und Beziehungen in netzwerkartigen Strukturen zu ermitteln (vgl. Lim u. a., 2015). Graphen bieten sich als Struktur an, wenn Entitäten in Beziehungen zueinander gesetzt werden. Dieses können Personen, Gegenstände oder Produktionsabläufe sein, um nur einige Bereiche anzuschneiden. Die Graphentheorie ist ein weit ausgearbeitetes Teilgebiet der Mathematik und liefert Methodiken, die unmittelbar genutzt werden können. Enorme Wachstumsraten der zugrundeliegenden Daten, mehrere Milliarden Knoten und Kanten (vgl. Xin u. a., 2014) sowie die zunehmende Komplexität führten zu neuen Herausforderungen bei der Verarbeitung. Zu diesen zählt auch die Verteilung der vernetzten Datensätze, um eine handhabbare Verarbeitungszeit zu ermöglichen. Um diese Probleme zu bewältigen, entstanden eine Vielzahl von Graphverarbeitungs-Frameworks, die auf verschiedene Verarbeitungsparadigmen aufbauen (vgl. Lim u. a., 2015). Diese haben das Ziel, dem Anwender eine Abstraktion zu bieten, um eine komfortable Analyse von Graphstrukturen zu erreichen.

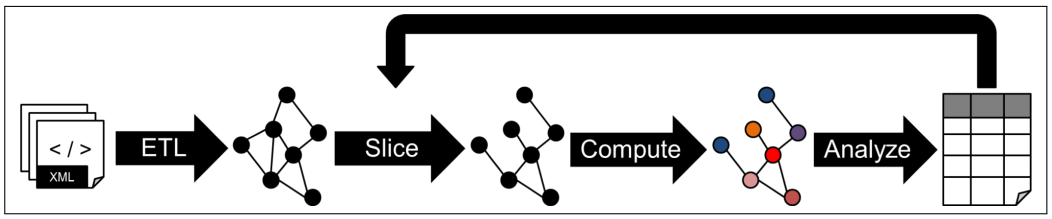


Abbildung 2.2.: Der Analyseprozess von Graphen (angelehnt an Xin u. a., 2014)

Algorithmen kompakt beschreiben zu können, bildet dabei nur einen Teil des Aufgabenbereichs. Vorbereitende Schritte, wie das Konstruieren von Graphen aus oftmals unstrukturierten Daten oder die Ergebnisextraktion, nehmen eine maßgebliche Rolle ein. Der typische Analyseprozess von Graphen umfasst somit Tätigkeiten von der Erzeugung eines Graphen aus rohen Daten über die Verarbeitung bis hin zur Analyse der Ergebnisse (vgl. Xin u. a., 2014). Der ursprüngliche Graph kann durchaus für mehrere unterschiedliche Verarbeitungen herangezogen werden. Viele Frameworks ermöglichen es dem Anwender, sich auf die Entwicklung zu fokussieren. Administrative Aufgaben, wie die optimale Verteilung der Daten oder Datenverluste, werden oftmals automatisch vom Framework behandelt. GraphX stellt ein solches

Graphverarbeitungs-Framework dar. Dieses hat sich zum Ziel genommen, den kompletten in Abbildung 2.2 dargestellten Analyseprozess abzubilden (vgl. Xin u. a., 2014).

2.2. Spark

Spark ist ein Framework für die verteilte Verarbeitung von Daten. Zu Beginn werden in Abschnitt 2.2.1 die Bestandteile und Besonderheiten von Spark veranschaulicht. Die Hauptabstraktion von Spark stellt das *Resilient Distributed Dataset* dar. Dieses wird in Abschnitt 2.2.2 beschrieben. Die Ausführungsstrategie, der von der *Resilient Distributed Dataset* bereitgestellten Operationen, ist Bestandteil von Abschnitt 2.2.3.

2.2.1. Spark Analyse Stack

Apache Spark[1] ist ein Open-Source-Framework, das von dem AMPLab der Universität Berkeley Kalifornien entwickelt wurde. Es dient der Datenanalyse innerhalb eines Clusters. Spark wurde konzipiert, weil bisherige clusterorientierte Verarbeitungs-Frameworks, nicht für alle Arten von Anwendungsgebieten eine optimale Lösung darstellen (vgl. Zaharia u. a., 2012). Die jedoch durchaus für die Herausforderungen von skalierbaren und datenintensiven Anwendungen gewachsen sind. Beispielsweise ist hier Apache Hadoop[2] zu nennen, eine Open-Source-Implementierung von MapReduce. Die Motivation von Spark liegt in der Verarbeitung von iterativen Algorithmen, die mehrere Berechnungen auf gleichen Zwischenergebnissen ausführen. Oft kommen solche im Gebiet des maschinellen Lernens oder bei Graphalgorithmen vor. Zusätzlich beinhaltet Spark eine In-Memory-Datenverarbeitung. Daten und Zwischenergebnisse müssen auf Grund dieser nicht nach einer Berechnung aufwendig in einem Dateisystem gespeichert und serialisiert werden, wie es bei Hadoop der Fall ist. Durch die Verwendung von Spark verzichtet man zudem nicht auf die Fehlertoleranz und Skalierbarkeit einer klassischen MapReduce-Anwendung (vgl. Lu u. a., 2014).

Spark setzt, sich wie in Abbildung 2.3 dargestellt, aus mehreren Komponenten zusammen. *Spark Core* bildet die zentrale Komponente, welche grundlegende Funktionalitäten wie Scheduling, Verteilung der Daten und Monitoring liefert (vgl. Karau u. a., 2015). Zudem umfasst *Spark Core* auch das *Resilient Distributed Dataset*, eine im Cluster partitionierte Datenstruktur (vgl. Zaharia u. a., 2012). Diese bildet das Kernkonzept hinter Spark. Hierauf wird in Abschnitt 2.2.2 näher eingegangen. Aufbauend auf *Spark Core* wurden mehrere spezifische Module entwickelt, die jeweils für unterschiedliche Anwendungsgebiete geschaffen wurden. Diese Arbeit wird

[1] https://spark.apache.org/
[2] https://hadoop.apache.org/

Spark SQL	Spark Streaming	MLip	GraphX
Spark Core			
YARN	Mesos	Stand-Alone Scheduler	

Abbildung 2.3.: Spark Analyse Stack (vgl. Karau u. a., 2015)

sich mit der Modul GraphX beschäftigen. Für einen Einblick in Spark SQL, Spark Streaming oder MLip sei auf Karau u. a. (2015) verwiesen.

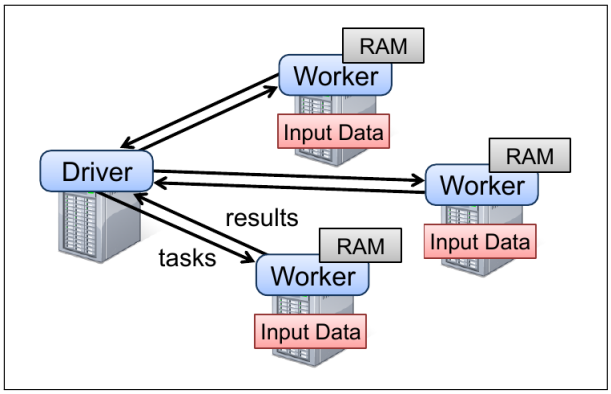


Abbildung 2.4.: Spark im Cluster

Eine Spark-Anwendung wird durch einen *SparkContext*, eine von Nutzern konfigurierbares Objekt, gesteuert (siehe Abb. 2.4). Es repräsentiert eine Verbindung zum Cluster und liefert alle wesentlichen Funktionalitäten von Spark. Der *SparkContext* kann mit verschiedenen Clustermanagementsystemen, wie Apache Mesos, Hadoop YARN oder einem Stand-Alone-Modus verbunden werden (vgl. Lu u. a., 2014). Das Cluster besteht aus einem Master-Knoten, auf dem die Anwendung läuft *(Driver)* und vielen Arbeiter-Knoten *(Worker)*. Diese können, müssen sich jedoch nicht zwangsläufig auf unterschiedlichen Maschinen befinden. Die Arbeiter-Knoten sind dafür zuständig vom *SparkContext* verschickte Aufgaben *(Tasks)* auszuführen. Dies sind Funktionen, die auf den Datensätzen angewandt werden sollen. Die Resultate werden der Anwendung zur Verfügung gestellt. Spark ist in der funktionalen, objektorientierten, statisch

getypten Programmiersprache Scala[3] geschrieben und auch zum Scala-Interpreter kompatibel, welche optimal für interaktives Datamining genutzt werden kann.

2.2.2. Resilient Distributed Dataset

Das *Resilient Distributed Dataset*, im Folgenden nur noch mit RDD bezeichnet, stellt eine *immutable*, parametrisierte *Collection* dar, die nach bestimmten Regeln automatisch im Cluster partitioniert wird (vgl. Xin u. a., 2013). RDD's können nur durch die von Spark bereitgestellten deterministischen Operationen über einen *SparkContext* erzeugt werden (vgl. Xin u. a., 2013). Um eine RDD zu erzeugen, stehen dem Programmierer eine Vielzahl von Möglichkeiten zur Verfügung. Zum einen kann eine RDD aus einer Datei aus einem Dateisystem eingelesen werden, zum anderen können auch andere Scala Collections durch eine *parallelize*-Operation in eine RDD umgewandelt werden. Diese Operation partitioniert die Collection und verteilt diese Partitionen anschließend auf verschiedene Worker-Instanzen (vgl. Zaharia u. a., 2010). Da Scala Code auf der Java virtuellen Maschine ausgeführt wird, können RDD's beliebige Java Objekte beinhalten. Auf RDD's sind mehrere parallele Operationen ausführbar. Man unterscheidet dabei zwischen Transformationen und Aktionen.

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Abbildung 2.5.: Transformationen und Aktionen auf RDD's

Transformationen sind Operationen, die eine RDD auf eine andere RDD abbilden. Das Ergebnis einer Transformation ist immer eine RDD.

Die *map*-Transformation erwartet beispielsweise eine beliebige Funktion $f : T \rightarrow U$ als Argument, welche für jedes Element der RDD parallel ausgeführt wird. Das Ergebnis ist eine RDD mit den je nach Abbildungsvorschrift resultierenden Ergebnissen als Elemente. Um

[3] http://www.scala-lang.org/

Elemente einer RDD auszusortieren, kann man die Transformation *filter* mit einem Prädikat als Argument aufrufen. Das Ergebnis umfasst danach lediglich Elemente, die das Prädikat erfüllen.

Da RDD's *immutable* sind, sobald sie erstellt wurden, erzeugt jede Transformation auf einer RDD eine neue RDD (vgl. Zaharia u. a., 2012). Zudem sind Transformationen *lazy*. Das bedeutet, dass Transformationen nicht sofort bei Aufruf ausgeführt werden, sondern erst, wenn die RDD erstmals durch eine Aktion verwendet wird. Es ist nicht notwendig jede RDD einmal zu materialisieren, da mehrere nacheinander folgende Transformationen als ein Datenstrom verarbeitet werden können (vgl. Zaharia u. a., 2010). Ein weiterer Vorteil ist gerade bei sehr großen Datenmengen, dass nicht alle RDD's komplett kalkuliert werden müssen, sondern nur der Anteil der tatsächlich benötigten Daten.

Aktionen hingegen aggregieren eine RDD zu einem Wert, der dem Client zur weiteren Verwendung zur Verfügung steht oder für späteren Gebrauch persistent gespeichert wird. Dieser wird parallel aus den einzelnen Partitionen berechnet. Bei dem Ergebnis handelt es sich, im Vergleich zu Transformationen, nicht mehr um eine RDD. Ein prominentes Beispiel hierfür ist *reduce*, eine Aktion, die als Argument eine assoziative Funktion $f : (T, T) \rightarrow U$ erhält, um einen einzigen Wert zu bestimmen. Abbildung 2.5 zeigt eine Übersicht einiger Transformationen und Aktionen auf RDD's in Spark.

Wenn eine RDD als Zwischenergebnis für mehrere Berechnungen verwendet wird, ist es möglich, diese manuell durch den Aufruf der *cache*-Operation im Arbeitsspeicher für spätere Verwendungen zu halten (vgl. Zaharia u. a., 2010). Damit werden aufwendige und redundante Neuberechnungen besonders bei iterativen Algorithmen vermieden. Bei *cache* handelt es sich jedoch lediglich um einen Hinweis, dass die RDD für spätere Berechnungen gespeichert werden soll, wenn genügend Arbeitsspeicher zur Verfügung steht (vgl. Zaharia u. a., 2010).

Eine RDD ist fehlertolerant gegenüber dem Verlust von einzelnen Partitionen (vgl. Zaharia u. a., 2012). Die Fehlertoleranz wird nicht durch eine Replikation der Datensätze realisiert. Verlorene Partitionen werden hingegen neu berechnet. Jede RDD loggt die ausgeführten Transformationen mit, aus der sie entstanden ist und kennt entsprechend die Berechnungen. Mit diesen Abstammungsinformationen können einzelne Partitionen auch über mehrere Transformationen hinweg berechnet werden (vgl. Xin u. a., 2013). Deshalb ist es nicht zwingend notwendig, alle Elemente einer RDD jederzeit im Speicher zu halten (vgl. Zaharia u. a., 2010).

2.2.3. Verarbeitung von Aktionen

Durch den Aufruf einer Aktion wird anhand der Abstammungsinformation aller beteiligten RDD's ein azyklischer gerichteter Graph erstellt. Wie in Abbildung 2.6 veranschaulicht wird,

repräsentieren die Kanten verschiedene Transformationen. Die gefüllten Rechtecke stellen Partitionen dar. Mehrere Partitionen bilden eine RDD, hier die RDD's A bis G. Die Abhängigkeiten zwischen den einzelnen Partitionen lassen sich in zwei Kategorien aufteilen (vgl. Zaharia u. a., 2012).

- Bei **engen** Abhängigkeiten wird für die Transformation einer Partition nur eine einzige Partition als Quelle verwendet. Eine Sequenz von Transformationen mit engen Abhängigkeiten lassen sich in einem Datenfluss verarbeiten, da sie unabhängig voneinander ausgeführt werden können. Mehrere folgende *enge* Abhängigkeiten können in einem Schritt (Stage) verarbeitet werden.
- **Breite** Abhängigkeiten hingegen nutzen bei der Transformation mehrere Partitionen als Quelle und verursachen viel Netzwerkkommunikation. Das Ergebnis RDD wird jeweils neu partitioniert und diese Transformationen sind entsprechend aufwendiger in der Verarbeitung (vgl. Lu u. a., 2014).

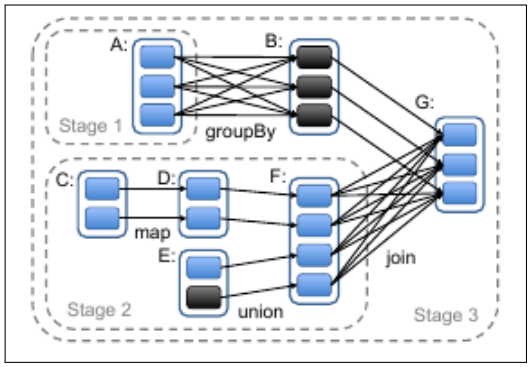


Abbildung 2.6.: Verarbeitungsgraph einer RDD

Wird auf G eine Aktion ausgeführt, führt dies zur Ausführung der Transformationen. Wie in Abbildung 2.4 dargelegt, werden die Berechnungen als Aufgaben an die einzelne Partitionen verschickt. Die Verarbeitungsschritte werden dort nacheinander ausgeführt. Wenn das Ergebnis *RDD* eines Verarbeitungsschrittes bereits im Arbeitsspeicher vorliegt (durch ein schwarzes Rechteck verdeutlicht), wird dies nicht wiederholt berechnet. In Abbildung 2.6 wird *Stage 1* beispielsweise nicht ausgeführt, da die Ergebnisse bereits vorliegen. Das Ergebnis der Aktion wird nun der Anwendung zur weiteren Verarbeitung zur Verfügung gestellt.

3. GraphX

Nachdem in Kapitel 2 grundlegende Aspekte eingeführt wurden, geht es in diesem Kapitel um das GraphX Framework. Dabei liegt der Fokus der Darstellung von GraphX auf den Bereichen, dessen Kenntnisse für die Entwicklung eines Algorithmus notwendig sind. Im Folgenden wird dabei auf das Modell hinter GraphX eingegangen. Es folgt ein Abschnitt über die Verteilung der zugrundeliegenden Daten und eine Übersicht über einige von der GraphX Abstraktion bereitgestellten Operationen.

3.1. Modell

GraphX wurde aufbauend auf Spark entwickelt und ermöglicht eine parallele Verarbeitung von Graphstrukturen. Da Spark an sich nur eine Verarbeitung von listenartigen Strukturen vorsieht, wird sich zu Beginn in Abschnitt 3.1.1 mit dem Konzept der Graphverarbeitung in GraphX beschäftigt. Die in Definition 2.1.1 präsentierte Darstellung eines Graphen reicht für die Repräsentation eines Graphen im GraphX Framework nicht aus und wird im Anschnitt 3.1.2 zu einem mächtigeren Modell erweitert.

3.1.1. Graph Verarbeitungsparadigma

Spark ist zwar den Anforderungen der parallelen Verarbeitung großer Datenmengen gewachsen, jedoch gilt dies nicht unbedingt für strukturierte Daten, wie einen Graphen. Das datenparallele Modell hinter Spark sieht eine partitionsorientierte Sicht auf die Daten vor. Transformationen werden auf RDD's mit einer Vielzahl von Elementen angewandt. Die Parallelität wird hier dadurch erreicht, dass unabhängige Daten auf unabhängigen Servern berechnet werden können. Der Anwender implementiert eine Funktion, die auf sämtlichen Elementen einer listenartigen Struktur ausgeführt wird. Bei der Graphverarbeitung liegt der Fokus jedoch auf der Transformation einzelner Elemente im Kontext anderer (vgl. Xin u. a., 2013). Ein Graph ist eine rekursive Datenstruktur. Viele Algorithmen auf Graphen arbeiten wiederholt auf mehreren Knoten und verarbeiten diese im Kontext seiner Kanten und deren Endknoten. Diese vielen sich resümierenden, lokalen Berechnungen stellen das Hauptmerkmal graphparalleler Verarbeitung

dar (vgl. Xin u. a., 2014). Der Graph wird bei der Parallelisierung auf viele Maschinen partitioniert. Bei jeder lokalen Transformation müssen Abhängigkeiten zwischen Knoten entlang der Kanten aufgelöst werden (vgl. Xin u. a., 2014). Diese Transformationen lassen sich parallel ausführen. Ausgangspunkt ist auch hier eine vom Anwender implementierte Funktion, die auf den einzelnen Knoten angewandt wird. Durch das Lokalitätsverhalten der Verarbeitung kann die Struktur des Graphen für eine effiziente und optimale Partitionierung herangezogen werden (vgl. Xin u. a., 2014). Eine Graphverarbeitung direkt auf Spark kann leicht zu einer komplexen Aneinanderreihung von *join*-Operationen und massiger Netzwerklast durch Datenaustausch führen (vgl. Xin u. a., 2013).

GraphX baut auf Spark auf und wurde entworfen, um diese Probleme zu lösen. Wie auch Spark wird GraphX an der Universität Berkeley Kalifornien entwickelt und erstmals von Xin u. a. (2013) beschrieben. Weitere Veröffentlichungen, die sich intensiv mit GraphX beschäftigen, sind Xin u. a. (2014) und Gonzalez u. a. (2014).

3.1.2. Graph Repräsentation

GraphX erweitert das RDD-Konzept und baut somit auf deren Fehlertoleranz und Skalierbarkeit auf. GraphX arbeitet auf der Struktur eines gerichteten Graphen und ermöglicht es zudem jedem Knoten und jeder Kante ein vom Benutzer definiertes Attribut (*Property*) zuzuweisen. Ein sogenannter *Property-Graph* wird folgendermaßen definiert:

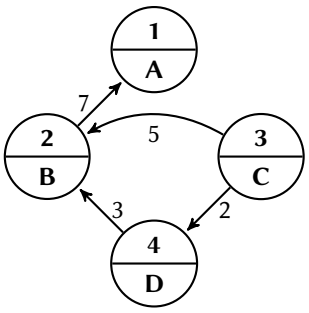
Definition 1 (Property-Graph) *Ein Property-Graph $G(P) = (V, E, P)$ ist ein gerichteter Graph[1] nach Abschnitt 2.1.1 und einer Kollektion von Properties $P = (P_V, P_E)$. Jeder Knoten und jede Kante besitzt eine Property. Dieses kann eine Gewichtung, ein Zustand oder andere Metadaten sein. Eine Property eines Knotens $v \in V$ wird durch $P_V(v)$ dargestellt. Die Property einer Kante $e \in E$ durch $P_E(e)$. Handelt es sich bei $P_E(e)$ um einen numerischen Wert, wird $P_E(e)$ vereinfacht auch als Kantengewichtung von e bezeichnet. Für einen Property-Graph $G(P)$ kann sich die Menge der Properties durch eine Transformation $f(P) \rightarrow P'$ ändern. Die Struktur des neuen Property-Graph $G(P')$ bleibt dabei unverändert (angelehnt an Xin u. a. (2014)).*

Die Entwicklung von GraphX ist inspiriert durch die Idee, dass man einen Property-Graphen effizient durch Tabellen von Knoten und Kanten abbilden kann und die Verarbeitung eines Graphen sich relationsalgebraisch formulieren lässt (vgl. Xin u. a., 2014). Zur Darstellung eines Property-Graphen wurde für GraphX eine neue Abstraktionsebene, der *Resilient Distributed Graph* (RDG), geschaffen. Ein Property-Graph $G(P)$ wird in GraphX intern als RDG durch ein

[1] Um auch Mehrfachkanten darstellen zu können, handelt es sich bei der Kantenmenge um eine Multimenge.

RDD-Paar dargestellt (vgl. Xin u. a., 2014). Eine RDD repräsentiert die Knoten und die andere die Kanten von $G(P)$. Logisch lassen sich beide RDD's als Mengen von den Wertepaaren $(i, P_V(i))$ und $((i,j), P_E((i,j)))$ für alle Knoten $i \in V$ und Kanten $(i,j) \in E$ darstellen. Dabei wird für die Identifizierung eines Knotens ein 64-Bit Integer Wert verwendet. Dieser kann vom Nutzer selbst bestimmt oder automatisch durch eine Hashfunktion, die auf der Property $P_V(i)$ des jeweiligen Knotens angewandt wird, berechnet werden. Ein RDG kennt somit die Adjazenzstruktur und die mit den Knoten und Kanten assoziierten Attribute und stellt eine graphorientierte Ansicht auf die Daten dar. Wie auch RDD's sind RDG's *immutable*. Die Graphverarbeitung in GraphX ist entsprechend eine Aneinanderreihung von Transformationen von einem statischen Graphen zu einem anderen auf Ebene der Transformation von einzelnen Knoten und Kanten (vgl. Xin u. a., 2013).

Die Elemente der RDD's in GraphX bestehen aus ungeordneten Schlüssel-Wertepaaren und repräsentieren unstrukturierte Daten (vgl. Xin u. a., 2014). Schlüssel brauchen keinen Wert oder können mehrfach vorkommen. Dies ist essentiell, um rohe Daten einfach verarbeiten zu können. Zur Erhaltung der Konsistenz werden innerhalb eines RDG's aus der Menge aller Knoten mit gleicher Identifizierung ein willkürlich gewählter Knoten in die Struktur aufgenommen. Nicht vorhandene Knoten für existierende Kanten werden erzeugt und erhalten einen definierbaren *Default*-Zustand.



Knoten

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

Kanten

(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)

Abbildung 3.1.: Repräsentation eines Property-Graphen durch Tabellen

Die Aufteilung der Knoten- und Kantendaten in separate Datenmengen verfolgt dabei gewisse Vorteile. Aus Perspektive der Verarbeitung ist die Strukturierung von Daten in Form

eines Graphen nur ein Teil des Analyseprozesses (vgl. Xin u. a., 2014). Es werden neue Daten erhoben und weitere Erkenntnisse aus anderen Verarbeitungen gewonnen. GraphX ermöglicht es, diese externen Daten mit der internen Graphstruktur dynamisch zusammen zu bringen. So können die Properties aller Knoten mit anderen Daten erweitert oder die gleichen Knoten- oder Kantenlisten für verschiedene Graphen genutzt werden. Ein Graph lässt sich zu jeder Zeit in seine listenartige Darstellung dekonstruieren. Verschiedene Graphen können so sehr einfach verglichen werden. GraphX bietet dadurch neben der graphorientierten auch eine listenartige Sicht auf die zugrundeliegenden Daten. Dies sind die eingangs erwähnten Listen aller Knoten und Kanten des Graphen oder einem Hybriden, dem sogenannten *Triple*.

Das Triple ist eine RDD, dessen Elemente eine Kante und deren Anfangs- und Endknoten, sowie alle beteiligten Attribute umfasst. Veranschaulicht wird die Nachbarschaftsbeziehung zwischen Knoten (vgl. Gonzalez u. a., 2014). Diese Ansicht auf die Daten ermöglicht eine Vielzahl von Operationen auf die in Abschnitt 3.3 eingegangen wird. Logisch lässt sich das Triple als Menge von Wertepaaren $((i, j), (P_V(i), P_E((i, j)), P_V(j))$ darstellen. Die Repräsentation eines Property-Graphen anhand von Tabellen ist in Abbildung 3.1 exemplarisch verdeutlicht.

3.2. Verteilung

Das GraphX Framework baut auf RDD's auf und nutzt somit deren Möglichkeiten für eine Partitionierung der Daten. Mit der parallelen Verarbeitung von Graphen sind jedoch andere Herausforderungen mit der Verteilung der Daten verbunden. Auf diese wird in Abschnitt 3.2.1 eingegangen. Aufbauend darauf wird in Abschnitt 3.2.2 die interne und somit verteilte Darstellung eines Graphen thematisiert. Mit der Triple-Ansicht als zentrales Glied der graphparallelen Verarbeitung befasst sich Abschnitt 3.2.3.

3.2.1. Herausforderung natürlicher Graphen

Wachsende Datenmengen führen dazu, dass eine Maschine der Last der Verarbeitung nicht mehr gewachsen ist. Die Erwartungen der Nutzer liegen auf skalierbaren Systemen. Deshalb ist es wichtig, Datensätze und deren Verarbeitung auf mehrere Maschinen zu verteilen, um damit eine effizientere parallele Verarbeitung zu ermöglichen. Um auch Graphen als stark vernetzte Struktur auf mehrere Maschinen verteilen zu können, sind spezielle Mechanismen der Indizierung und Strukturierung erforderlich (vgl. Xin u. a., 2013).

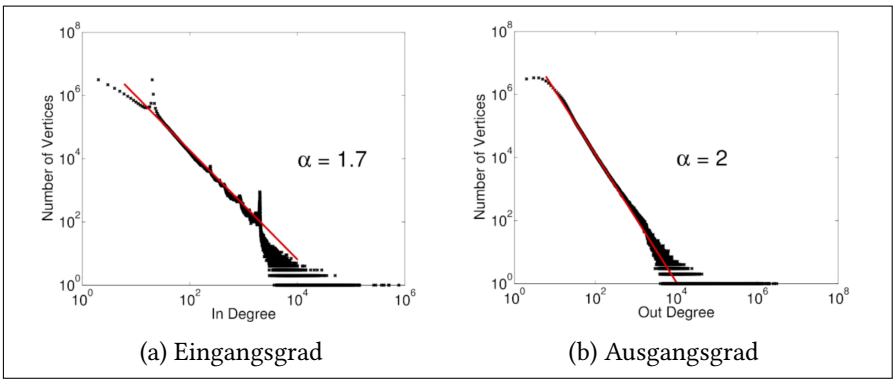
Gegenstand der Analyse von Graphen sind häufig Strukturen aus der realen Welt. Diese weisen gewisse Eigenschaften auf, die zu Herausforderungen bei der parallelen Verarbeitung führen. Werfe man beispielsweise einen Blick auf Strukturen, wie sie von sozialen Netzwerken

wie Twitter[2] oder Facebook[3] generiert werden. In den hier zugrundeliegenden Graphstrukturen werden verschiedene Personen durch soziale Interaktionen miteinander in Verbindung gesetzt. Diese Beziehung kann eine Freundschaft oder ein im Twitter Kontext gebräuchliche "Folgen"-Beziehung sein. Besonderheit ist, dass viele Personen in Verbindung mit nur wenigen anderen stehen. Jedoch lediglich ein Prozent der Personen ist für fast die Hälfte aller bestehenden Beziehungen verantwortlich (vgl. Gonzalez u. a., 2012). Die Wahrscheinlichkeit, dass ein Knoten u einen Grad k besitzt, steht im Verhältnis:

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

Der Exponent α regelt die Schiefe der Verteilung. Je größer α, desto mehr Knoten besitzen einen niedrigeren Grad. Bei der Verteilung für natürliche Graphen wird $\alpha \approx 2$ unterstellt (vgl. Gonzalez u. a., 2012).

Dieses Verhalten lässt sich durch die globale Reichweite von medienpräsenten Personen begründen. Abbildung 3.2 zeigt die Verteilung der Knoten nach Eingangs- und Ausgangsgrad von Twitter. Die Kernherausforderung liegt in der unausgewogenen Verteilung der Anzahl der Nachbarn zwischen den Personen. Zudem bestehen auf sozialen Strukturen basierende Graphen über signifikant mehr Beziehungen als agierende Entitäten.



(a) Eingangsgrad (b) Ausgangsgrad

Abbildung 3.2.: Verteilung des Eingangsgrad (a) und Ausgangsgrad (b) im Twitternetzwerk (entnommen Gonzalez u. a., 2012).

Graphverarbeitung verlangt die Verarbeitung von Knoten und Kanten im Kontext seiner Nachbarn. Der Graph muss so partitioniert werden, dass die Auflösung dieser Beziehungen mit geringer Netzwerklast verbunden ist. Verfügt ein Knoten über eine sehr große Anzahl von Nachbarn, hat dieser verhältnismäßig mehr Arbeit zu verrichten. Dieses führt zu variablen

[2] https://twitter.com/
[3] https://www.facebook.com/

Ausführungszeiten und einer unterschiedlichen Menge an Speicherplatzbedarf für die Daten adjazenter Elemente (vgl. Gonzalez u. a., 2012). Im folgenden Absatz wird darauf eingegangen, wie GraphX die zugrundeliegenden Daten partitioniert, um diese Herausforderungen zu lösen.

3.2.2. Interne Darstellung

Die Aufteilung der Daten auf verschiedene Maschinen spielt eine zentrale Rolle in der Verarbeitung. Für eine effektive Verarbeitung gilt es, die Netzwerklast so gering wie nur möglich zu gestalten und für eine ausgeglichene Balance zwischen den Maschinen zu sorgen (vgl. Gonzalez u. a., 2012).

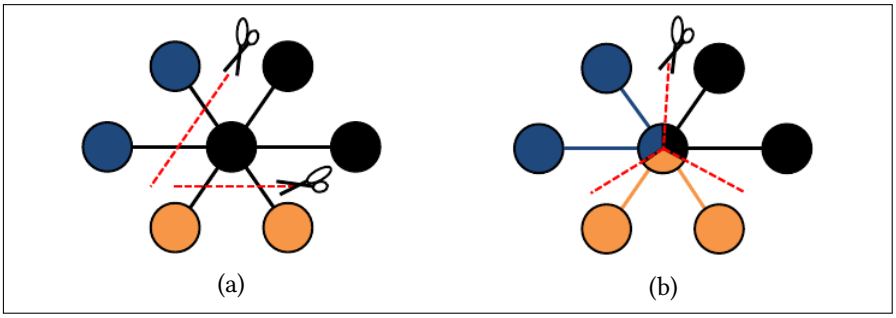


Abbildung 3.3.: Ein Kanten-Schnitt (a) und ein Knoten-Schnitt (b) für eine Partitionierung eines Graphen in drei Teile (entnommen Xin u. a., 2013)

.

Um einen Graphen zu partitionieren, existieren zwei grundlegende Verfahren:

- Eine Möglichkeit ist die horizontale Partitionierung des Graphen anhand eines **Kanten-Schnitts** (siehe Abb. 3.3a). Dies sieht vor, jeden Knoten genau einer Maschine zuzuweisen. Eine geschnittene Kante wird hingegen auf mehrere Maschinen repliziert. Änderungen die eine geschnittene Kante betreffen, wie beispielsweise die Modifikation der Kanten-Property, müssen synchronisiert werden (vgl. Gonzalez u. a., 2012). Der Speicher und Kommunikationsaufwand dieses Verfahrens wächst proportional mit der Anzahl geschnittener Kanten (vgl. Xin u. a., 2013).

- Eine andere Möglichkeit sieht vor, jede Kante einer Maschine zuzuordnen und Knoten auf mehrere Maschinen zu verteilen. Abbildung 3.3b zeigt einen sogenannten **Knoten-Schnitt**. Änderungen an Kanten müssen hier nicht kommuniziert werden, da sie nicht repliziert wurden. Lediglich Änderungen an Knoten müssen zu sämtlichen Maschinen, auf denen sich Replikate befinden, kommuniziert werden (vgl. Gonzalez u. a., 2012).

Der Aufwand der Speicherung und Kommunikation wächst hierbei proportional zu der Anzahl von Maschinen, auf welche je ein Knoten repliziert wird.

GraphX setzt für die Partitionierung eines Graphen einen Knoten-Schnitt um. Es hat sich gezeigt, dass ein Kanten-Schnitt für natürliche Graphen, wie sie in Abschnitt 3.2.1 dargestellt sind, im Vergleich zu einem Knoten-Schnitt schlechte Laufzeiten erzielen (vgl. Xin u. a., 2013). In GraphX lässt sich die Partitionierung durch ein vom Nutzer auswählbares Hashverfahren steuern. Nach diesem werden Kanten auf verschiedene Partitionen verteilt. Auf jede Kante wird hierzu ein Hashverfahren angewandt, welches in Abhängigkeit zur Anfangs- und Endknoten-Identifizierung und der maximalen Anzahl an Partitionen eine *PartitionsId* berechnet. Nach dieser wird eine Verteilung in Gang gesetzt. Neben den von GraphX bereitgestellten Verfahren können auch eigene Varianten für eine Partitionierung herangezogen werden. Die einfachste Variante eine Partitionierung durchzuführen, ist die Anwendung eines zufälligen Knoten-Schnitts. Dieser verteilt Kanten zufällig auf die Partitionen.

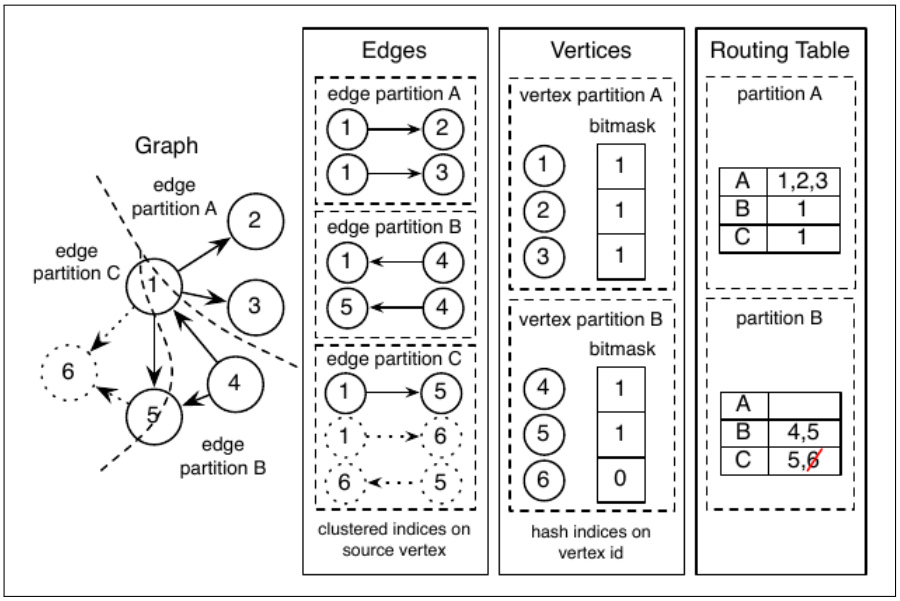


Abbildung 3.4.: Interne Repräsentation eines in drei Teile partitionierten Graphen (entnommen Gonzalez u. a., 2014)

Abbildung 3.4 zeigt die interne Darstellung eines Graphen, der durch einen Knoten-Schnitt partitioniert wurde. Die drei horizontal partitionierten Tabellen werden in GraphX durch jeweils eine RDD realisiert. Mit ihnen sind folgende Aufgaben verbunden:

- Die **Edges** Tabelle stellt die adjazente Struktur dar. Jede Kante wird anhand ihrer *PartitionsId* einer Partition zugeordnet. Für eine Kante werden die Anfangs- und Endknoten-Identifizierung, sowie die Property der Kante gespeichert.
- Die nach Knoten-Identifizierung partitionierte **Vertices** Tabelle speichert für jeden Knoten seine Identifizierung und deren Property. Zudem wird die Verfügbarkeit eines Knotens über eine *Bitmask* gespeichert. Markierte Knoten werden bei Transformationen nicht mehr berücksichtigt und können so logisch aus der Struktur entfernt werden. Dies ist beispielsweise bei der Teilgraphbildung erforderlich (siehe Abschnitt 3.3.2).
- Die Tabelle **Routing Table** fungiert wie ein assoziatives Datenfeld. Es wird für eine Knoten-Identifizierung die jeweilige *PartitionsId* der adjazenten Kanten geliefert. Diese spielt eine zentrale Rolle in der Erzeugung der Triple-Ansicht (siehe Abschnitt 3.2.3) Diese Tabelle wird mit **Vertices** co-partitioniert.

Die hier vorgenommene Strukturierung der Knoten- und Kantendaten durch Indizierung kann über mehrere Transformationen hinweg erhalten bleiben. Graphen in GraphX sind wie eingangs erwähnt *immutable*. Nach jeder Transformation wird ein neues Objekt erzeugt, welches die interne Strukturierung wiederverwenden kann (vgl. Xin u. a., 2014). Dies hängt jedoch davon ab, ob die ausgeführte Transformation die Struktur verändert oder nur auf der Menge der Properties operiert.

3.2.3. Triple-Ansicht

Die bereits in Abschnitt 3.1.2 kurz thematisierte Triple-Ansicht wird bei Bedarf aus der Knoten- und Kanten-RDD extrahiert. Hierzu sind *join*-Operationen zwischen der Knoten- und Kanten-RDD notwendig, da eine Kante jeweils Zugang zu den Properties seines Anfangs- und Endknotens benötigt. Auf Grund dessen, dass diese beiden RDD's unabhängig voneinander partitioniert sind, kommt es dadurch zu Datenaustausch zwischen den Partitionen. Diese gilt es so gering wie möglich zu halten. GraphX sieht für die Erzeugung der Triple-Ansicht vor, die Knoten-Properties zu den Kanten-Partitionen zu verschicken. Dadurch wird die Performanz auf zweierlei Weise verbessert (vgl. Gonzalez u. a., 2014). Zum einen wird eine Knoten-Property durchaus von mehreren Kanten innerhalb einer Partition verwendet und kann entsprechend wiederholt für diese genutzt werden. Zum anderen liegt der Betrachtungspunkt der Analyse häufig auf sozialen Strukturen, wie sie in Abschnitt 2.1.2 aufgezeigt wurden. Diese verfügen über signifikant mehr Kanten als Knoten.

Durch eine Hinzunahme der *Routing Table* können Knoten-Properties lediglich zu den Kanten-Partitionen geschickt werden, welche die Information auch verwenden. Ein Broadcast

wird so vermieden. Dadurch spielt es auch keine Rolle auf welcher Partition die Kanten liegen. Die Partitionierung unterliegt entsprechend einer sehr großen Flexibilität.

Die Verteilung des Graphen auf mehrere Partitionen wird von dem GraphX Framework übernommen. Der Anwender muss sich damit primär nicht beschäftigen. Bei Bedarf ist es jedoch jederzeit möglich, die Partitionierungs-Strategie und die Anzahl an Partitionen zu ändern.

3.3. Logische Abstraktion

Im Folgenden wird ein Blick auf die GraphX-API geworfen. Diese stellte eine unabdingbare Grundlage für eine spätere Algorithmen-Implementierung dar. Der aktuelle Source Code ist im Apache Spark GitHub Repository[4] frei verfügbar. GraphX bietet eine Programmierschnittstelle, die sowohl graph- als auch datenparallele Operationen ermöglicht. Ein RDG wird in GraphX, durch die abstrakte Klasse *Graph* dargestellt, welche eine Vielzahl von Operationen bereitstellt. In Abschnitt 3.3.1 werden deren Bestandteile beleuchtet. Daraufhin beschäftigt sich Abschnitt 3.3.2 und 3.3.3 mit den von der Graph-Abstraktion bereitgestellten Operationen. Dabei wird eine Unterteilung in strukturerhaltende und strukturverändernde Operationen vorgenommen. Zuletzt wird in Abschnitt 3.3.4 ein iteratives Verarbeitungsmodell vorgestellt.

3.3.1. Bestandteile der Graph-Abstraktion

Die Graph-Abstraktion baut, wie schon in vorherigen Kapiteln erläutert, auf drei listenartige Strukturen auf. Es handelt sich dabei um eine Knoten-, Kanten- und Triple-Ansicht der zugrundeliegenden Daten. Diese Bestandteile sind im Listing 3.1 dargestellt.

```
1 abstract class Graph[VD, ED] {
2   val vertices: VertexRDD[VD]
3   val edges: EdgeRDD[ED]
4   val triplets: RDD[EdgeTriplet[VD, ED]]
5   ...
6 }
```

Listing 3.1: Bestandteile der Graph-Abstraktion

Die listenartigen Knoten- und Kantenansichten werden jeweils durch die abstrakten Klassen *VertexRDD* und *EdgeRDD* repräsentiert. Sie erweitern RDD's um spezifische Operationen. Die Triple-Ansicht wird, wie in Abschnitt 3.2.3 beschrieben, aus den Mengen der Knoten

[4] https://github.com/apache/spark/tree/master/graphx

und Kanten dynamisch erzeugt. Repräsentiert wird sie durch eine einfache RDD. Ein Graph lässt sich jederzeit in diese öffentlichen Bestandteile zerlegen, um beispielsweise einfache datenparallele Operationen auf ihnen auszuführen. Diese werden häufig für Aggregationen und der Analyse des Graphen am Ende der Berechnung verwendet (vgl. Xin u. a., 2014). Da es sich um übliche Operationen für die parallele Verarbeitung von Listen handelt, wird an dieser Stelle nicht näher auf sie eingegangen.

Das Grundkonzept von GraphX sieht vor, eine graphparallele Verarbeitung durch die Transformation der zugrundeliegenden RDD's zu realisieren. Zentrale Rolle spielt hier die Triple-Ansicht, da sie ein Zusammenspiel von den Knoten- und Kantendaten ermöglicht.

In folgenden Absätzen soll aufgezeigt werden, welche Möglichkeiten das GraphX Framework für die Verarbeitung von Graphen zur Verfügung stellt. Dazu soll kurz dargestellt werden, wie ein RDG-Instanz erzeugt werden kann. Die Klasse *Graph* stellt dazu eine *apply*-Methode zur Vergnügung. Diese erwartet zwei RDD's, von denen eine die Kanten und die andere die Knoten repräsentiert. In Abbildung 3.2 wird dieses verdeutlicht.

```
1  val sc: SparkContext = ...
2
3  val users: RDD[(VertexId, String)] = sc.parallelize(
4     Array((1l, "A"), (2l, "B"), (3l, "C")))
5
6  val relationships: RDD[Edge[Int]] = sc.parallelize(
7     Array(Edge(1l, 2l, 3), Edge(1l, 3l, 4),
8        Edge(2l, 4l, 6), Edge(5l, 4l, 9)))
9
10 val rawGraph: Graph[String, Int] = Graph(users, relationships)
```

Listing 3.2: Konstruktion einer RDG

3.3.2. Strukturverändernde Operationen

Bei der Entwicklung der *Graph*-Schnittstelle war ein Hauptkriterium die Sparsamkeit. Die daraus resultierende schmalere API ist mit einer besseren Möglichkeit verbunden, eine tiefgehende Optimierung der einzelnen Operationen zu erreichen (vgl. Xin u. a., 2013). Alle bereitgestellten Operationen können flexibel in einer beliebigen Reihenfolge ausgeführt werden. GraphX ist darauf ausgelegt worden eine aktuelle Aufgabe zu lösen ohne sich damit beschäftigen zu müssen, ob eine darauf folgende Verarbeitung inperformant ist.

Die Abstraktion *Graph* liefert eine graphorientierte Sicht auf die Daten. Entsprechend bieten die Operationen der Klasse eine graphparallele Verarbeitung an. Diese lässt sich als Aneinan-

derreihung von *join*, *group-by* und *map*-Operationen realisieren. Diesem Konzept entsprechend wurde die Graph API entworfen. Die Operationen lassen sich dabei in unterschiedliche Kategorien einordnen. Die hier getroffene Unterteilung unterscheidet hierbei, ob die Operation nur die Menge der Properties oder die Knoten- oder Kantenmenge verändern kann. Letztere zählen zu den strukturverändernden Operationen, welche nun Betrachtungspunkte sind. Eine Auswahl diese soll nun der Reihe nach vorgestellt werden. Strukturerhaltende Operationen werden in Abschnitt 3.3.3 thematisiert.

```
def subgraph(
   epred: EdgeTriplet[VD, ED] => Boolean = (x => true),
   vpred: (VertexId, VD) => Boolean = ((v, d) => true))
   : Graph[VD, ED]
```

Listing 3.3: *subgraph*-Operation

Die *subgraph*-Operation kann einzelne Knoten und Kanten aus der Graphstruktur entfernen. Dies geschieht über Prädikate, die auf den jeweiligen Elementen angewandt werden. Die Operation liefert einen Teilgraphen, der folgendermaßen definiert ist. Für zwei Property-Graphen $G(P) = (V, E, P)$ und $G'(P) = (V', E', P)$ heißt G' Teilgraph von G falls $V' \subseteq V$ und $E' \subseteq E$ ist. Zudem bleiben die Knoten- und Kanten-Properties erhalten. Es gilt entsprechend $\forall v \in V' : P'_V(v) = P_V(v)$ und $\forall e \in E' : P'_E(e) = P_E(e)$. Um die Konsistenz sicher zu stellen, müssen alle verbleibenden Kanten für ihre Anfangs- und Endknoten das Knotenprädikat erfüllen. Außerdem muss auch das Kantenprädikat für jede Kante gelten, andernfalls wird diese entfernt (vgl. Xin u. a., 2014).

```
def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]

def reverse: Graph[VD, ED]
```

Listing 3.4: *groupEdges* und *reverse*-Operation

Es ist durchaus möglich, dass sich innerhalb eines Graphen mehrere gleich gerichtete Kanten zwischen zwei Knoten befinden, welche sich lediglich in der Property voneinander unterscheiden. Viele Algorithmen sehen solche Mehrfachkanten allerdings nicht vor und diese stören somit bei der Berechnung. Für eine solche Situation aggregiert die *groupEdges*-Operation die Mehrfachkanten zu einer einzigen Kanten mithilfe einer vom Nutzer definierten *merge*-Funktion. Diese wird auf den Properties gleichartiger Mehrfachkanten angewandt. Auch die Richtung einer Kante kann geändert werden. Die Operation *reverse* ändert beispielsweise die Richtung sämtlicher Kanten im Graphen. Für einen Property-Graph $G = (V, E, P)$ sei G' das Resultat von *reverse*. Dann gilt für den resultierenden Graph $G' = (V', E', P)$, dass sich im

Vergleich zur Eingabe der Anfangs- und Endknoten jeder Kante vertauscht hat. Entsprechend gilt $E' = \{(u, v)|(v, u) \in E\}$. Die Menge der Knoten verändert sich hingegen nicht. Es bleibt somit $V' = V$.

3.3.3. Strukturerhaltende Operationen

In diesem Abschnitt werden einige der Operationen vorgestellt, die lediglich auf der Menge der Knoten- oder Kanten-Properties operieren. Die Konsistenz des Graphen bleibt nach jeder dieser Transformation erhalten.

```
def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]

def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]

def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
def mapTriplets[ED2](
   map: EdgeTriplet[VD, ED] => ED2,
   tripletFields: TripletFields): Graph[VD, ED2]
```

Listing 3.5: Transformationen

Die *mapVertices* und *mapEdges*-Operationen transformieren die Knoten bzw. Kanten-Properties. Der Anwender implementiert dazu eine *map*-Funktion. Diese lässt sich auch als Abbildung darstellen. Sei $G(P) = (V, E, P)$ ein Property-Graph. Die Abbildung $map : P \to P$ transformiert die Menge der Properties von G. Der resultierende Property-Graph $(V, E, map(P))$ behält die ursprüngliche Struktur von G bei. Analog ist dieses auch auf der Triple-Ansicht möglich.

```
def joinVertices[U](table: RDD[(VertexId, U)])(
   map: (VertexId, VD, U) => VD): Graph[VD, ED]

def outerJoinVertices[U, VD2](other: RDD[(VertexId, U)])(
   map: (VertexId, VD, Option[U]) => VD2): Graph[VD2, ED]
```

Listing 3.6: *join*-Operationen

Mit der *joinVertices*-Operation ist es möglich, eine separate RDD von Knoten-Properties mit einem Graphen zu vermengen. Dieses ist notwendig, wenn beispielsweise Zusatzinformationen aus anderen Berechnungen für jeden Knoten hinterlegt werden sollen. Der Unterschied zwischen einem *join* und *outer join* bezieht sich darauf, dass bei einem *outer join* ein alternativer Wert angegeben werden kann, sollte der Knoten über keinen verfügen. Zudem kann sich der

Typ der Property ändern. Ansonsten sind beide Operationen logisch äquivalent (vgl. Xin u. a., 2013).

```
def aggregateMessages[M](
   sendMsg: EdgeContext[VD, ED, M] => Unit,
   mergeMsg: (M, M) => M,
   tripletFields: TripletFields = TripletFields.All)
   : VertexRDD[M]
```

Listing 3.7: Aggregatoren

Die für die Graphverarbeitung wichtigste Operation ist *aggregateMessages*. Diese stellt eine zweistufige Verarbeitung dar. Logisch handelt es sich dabei um die Anwendung einer *map*- gefolgt vom *group by*-Operation auf der Triple-Ansicht (vgl. Gonzalez u. a., 2014). *aggregateMessages* ermöglicht es, Knoten im Kontext seiner Nachbarn zu betrachten. Dieses Verhalten wurde in Abschnitt 3.1.1 bereits als wichtiges Kriterium für eine Graphverarbeitung identifiziert. Bei der Umsetzung wird auf Anwendung eines nachrichtenorientierten Konzeptes zurückgegriffen. In diesem ist es vorgesehen, dass sich Knoten untereinander Daten in Form von Nachrichten versenden und auf eingehende reagieren können. Dieses ist ein wesentliches Konzept von GraphX und wird auch im weiteren Verlauf eine große Bedeutung zugeschrieben.

Der Anwender implementiert zwei Funktionen *sendMsg* und *mergeMsg*. Die *sendMsg*-Funktion wird auf jedem Element der Triple-Ansicht ausgeführt und hat als Aufgabe einen Wert zu ermitteln, der an einen anderen Knoten geschickt werden soll. Ein Triple-Element wird durch die Klasse *EdgeContext* repräsentiert, welche über einen Methodenaufruf direkt den Wert an den Zielknoten senden kann. Die Methode braucht entsprechend keinen Rückgabewert. Der zu übertragende Wert kann auch als Nachricht vom Typ M aufgefasst werden, welche an einen Knoten adressiert ist. Dies ist der Anfangs- oder Endknoten jedes Triple-Elements. Alle Nachrichten mit gleichem Zielknoten werden schließlich durch die vom Anwender definierte *mergeMsg* Funktion aggregiert und dem jeweiligen Knoten als neuen Property-Wert zugewiesen. Optional lässt sich aus Sicht der Performanz die Verarbeitung auch auf einer Untermenge aller Kanten ausführen. Dies ist sinnvoll, wenn vor dem Aufruf schon ersichtlich ist, dass nicht für alle Knoten eine Property-Modifikation stattfinden wird.

Mit den bisher vorgestellten Operationen bietet GraphX die Möglichkeit, eine parallele Verarbeitung auf Graphen zu realisieren. Jedoch wird keine direkte Schnittstelle für die Implementierung von iterativen Algorithmen geliefert. Das RDG Interface wurde entworfen, um die Entwicklung von eigenständigen Verarbeitungsmodellen zu ermöglichen (vgl. Xin u. a., 2013). Bisher handelt es sich bei *Pregel* um das einzige in der aktuellen Version realisierte Modell. Dieses wird im Abschnitt 3.3.4 genauestens beschrieben. GraphX bietet gegenwärtig

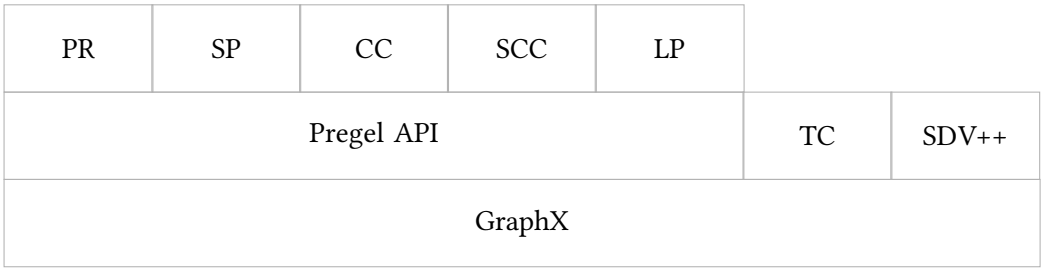


Abbildung 3.5.: GraphX stellt bereits einige Algorithmen bereit.

eine Reihe von Algorithmen, die aufbauend auf den Basisoperationen oder unter Nutzung des Pregel-Verarbeitungsmodells implementiert wurden (siehe Abb. 3.5). Hierzu zählen:

Page Rank (PR)
Ein Algorithmus zur Ermittlung der Relevanz von Knoten in Bezug auf die Netzwerkstruktur. Ursprünglich von Google für die Bewertung von Webseiten erdacht.

Shortest Path (SP)
Ermittelt alle kürzesten Strecken in Bezug auf die Anzahl der dazwischen liegenden Kanten für eine beliebige Anzahl von Knoten.

Connected Component (CC)
Ein Algorithmus zur Ermittlung von zusammenhängenden Komponenten. Dieses sind Teilgraphen, in denen alle Knotenpaare sich über beliebige Kanten erreichen können.

Strongly Connected Components (SCC)
Ermittelt starke Zusammenhangskomponenten. Diese sind Teilgraphen in dem jeder Knoten über gerichtete Kanten jeden anderen erreichen kann.

Label Propagation (LP)
Ein Algorithmus zur Ermittlung von Gemeinschaften.

Triangel Count (TC)
Berechnet die Anzahl an Dreiecken, die sich durch Kanten zwischen Knoten aufspannen. Für einen Graphen $G = (V, E)$ ist $\Delta = (V_\Delta, E_\Delta)$ ein Dreieck von G, falls $V_\Delta = \{u, v, w\} \subset V$ und $E_\Delta = \{(u, v), (v, w), (w, u)\} \subset E$ gilt. Die Anzahl der Dreiecke gibt Aufschluss über die Struktur des Graphen.

SDV++
Ein Algorithmus zur Singulärwertzerlegung.

3.3.4. Pregel

Pregel stellt ein knotenorientiertes, iteratives Verarbeitungsmodell von Graphen dar und wurde ursprünglich von Google entwickelt und von Malewicz u. a. (2010) erstmals vorgestellt. Im Folgenden wird auf die leicht vom Ursprung abweichende Pregel-Implementierung von GraphX eingegangen.

Ein auf Pregel aufbauender Algorithmus wird auf einem Property-Graphen $G(P)$ ausgeführt. Bei der Ausführung durchläuft der Algorithmus sequenziell mehrere Iterationen. Diese werden im Pregel Kontext als *Supersteps* bezeichnet. Zu Beginn implementiert der Anwender ein *Vertexprogramm* Q. Dieses ist eine Funktion, die auf sämtlichen Knoten von $G(P)$ angewandt wird. Das Vertexprogramm bildet den Hauptteil der Programmlogik und wird nur ein einziges Mal definiert. Seien v und u Knoten von $G(P)$, dann kann die Vertexprogramm-Instanz $Q(v)$ eines Knotens durch das Versenden von Nachrichten mit einer benachbarten Instanz $Q(u)$ kommunizieren. Voraussetzung dazu ist, dass dem Senderknoten die Identifikation des Empfängers bekannt ist. In Pregel kennen sich adjazente Knoten immer, entsprechend sind Kantenverbindungen zwischen den Knoten die üblichen Transportwege der Nachrichten. Ob nur auf ein- oder ausgehende Kanten oder beiden Nachrichten versandt werden sollen, kann frei definiert werden. Vertexprogramm-Instanzen arbeiten unabhängig voneinander und können in beliebiger Reihenfolge parallel verarbeitet werden (vgl. Xin u. a., 2014). Innerhalb eines Supersteps S_n erhält eine Vertexprogramm-Instanz $Q(v)$ alle Nachrichten, die ihr im Superstep S_{n-1} geschickt wurden. Entsprechend werden im Superstep S_n verschickte Nachrichten im Superstep S_{n+1} empfangen.

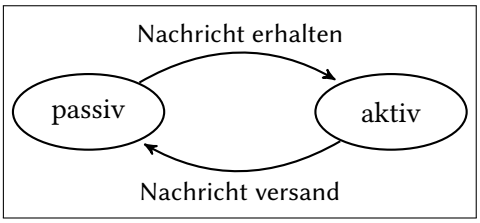


Abbildung 3.6.: Zustandswechsel in Pregel

Bei Pregel handelt es sich zudem um ein zustandsorientiertes Verfahren (vgl. Malewicz u. a., 2010). Jeder Knoten nimmt dabei zu jeder Zeit den Zustand *aktiv* oder *passiv* an. Ein *aktiver* Knoten führt das Vertexprogramm aus und ist in der Lage Nachrichten zu versenden. Das Versenden einer Nachricht führt wiederum zum Zustand *passiv*. Ein solcher *passiver* Knoten hat keine Arbeit mehr zu verrichten, kann jedoch durch das Erhalten einer Nachricht zum Zustand *aktiv* zurückkehren. Abbildung 3.6 verdeutlicht dieses. Initial befinden sich alle

Knoten im Zustand *passiv*. Um die Ausführung anzustoßen wird in der Initialisierungsphase zum Zeitpunkt S_0 eine initiale Nachricht an sämtliche Knoten geschickt. Dadurch gehen alle Knoten in den Zustand *aktiv* über, was jeweils zur Ausführung des Vertexprogrammes führt. Von Knoten verschickte Nachrichten enthalten immer die Identifikation des Zielknotens und einen Wert der übermittelt wird. Es ist auch möglich, dass ein *aktiver* Knoten eine leere Nachricht verschickt. Somit geht dieser Knoten in den Zustand *passiv* über ohne, dass andere Knoten eine Nachricht erhalten. Es wird dadurch ermöglicht, dass sich die Anzahl der neu verschickten Nachrichten innerhalb eines Supersteps reduzieren kann. Ein Knoten ist somit in der Lage keine (eine leere), eine oder mehrere Nachrichten zu versenden. Werden am Ende eines Supersteps keine Nachrichten versandt, terminiert der Algorithmus. Da dieses nicht immer in einem für den Anwender akzeptablen Zeitraum garantiert ist, lässt sich die maximale Anzahl an Iterationen auch begrenzen.

Ein Superstep lässt sich auch im GAS-Modell darstellen (vgl. Gonzalez u. a., 2012). Diese verallgemeinerte Darstellung unterteilt einen Superstep in drei hintereinander durchlaufende Phasen und gibt einen guten Überblick über den genauen Ablauf. Diese sind *Gather*, *Apply* und *Scatter*. Die einzelnen Phasen sind bei der Ausführung am Beispiel eines festen Knotens v mit folgenden Funktionalitäten verbunden[5]:

> In der **Gather**-Phase werden alle Nachrichten, die an den Knoten v adressiert sind, durch einen *Message Combiner* $\otimes$ zu einer einzigen Nachricht aggregiert. Die Operation $\otimes$ ist sowohl assoziativ als auch kommutativ. Für eine Nachricht $M_{u,v}$ ist u der Sender- und v der Empfängerknoten.
>
> $$M_\Sigma \leftarrow \bigotimes_{u \in N_{in}(v)} m(M_{u,v})$$
>
> Die **Apply**-Phase bestimmt in ihrer Ausführung eine neue Property für den Knoten v. Dabei kann auf die bisherige Property von v und die in der *Gather*-Phase berechnete Nachricht M_Σ zugegriffen werden. Die Änderung wird mit Beginn des nächsten Supersteps wirksam.
>
> $$P_V^{new}(v) \leftarrow a(P_V(v), M_\Sigma)$$

[5] Es wird davon ausgegangen, dass Nachrichten nur in Kantenrichtung verschickt werden.

In der **Scatter**-Phase werden Nachrichten an Nachbarknoten von v verschickt. Diese werden aus der neuen Property von v, der Property der jeweiligen Kante und der Property des Zielknotens bestimmt.

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

Das GAS-Modell wird in Pregel durch drei vom Nutzer zu implementierenden Funktionen umgesetzt. Diese haben gleiches Verhalten, werden jedoch als *messageCombiner-*, *vertexProgram-* und *sendMessage*-Funktion bezeichnet.

Pegel setzt das *Bulk Synchronous Parallel (BSP)* Modell (vgl. Valiant, 1990) um. BSP ist ein Verarbeitungsmodell für parallele Berechnungen. Dieses sieht eine Unterteilung des Programmes in eine Sequenz von Iterationen vor. Bei Pregel sind diese die Supersteps. Jede Vertexprogramm-Instanz innerhalb eines Supersteps kann auch als nebenläufiger Prozess aufgefasst werden (siehe Abb. 3.7). Das BSP-Modell sieht vor, alle Prozesse am Ende jeder Iteration durch eine Barriere aufzuhalten. Die Barriere wird aufgelöst, wenn alle Prozesse terminieren. Durch diese Synchronisation ist eine genaue Trennung der Supersteps möglich. Somit ist garantiert, dass sämtliche Properties aktualisiert und alle Nachrichten zugestellt sind.

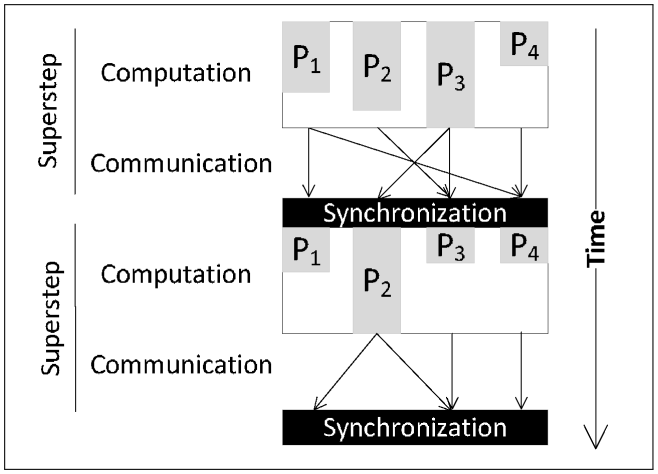


Abbildung 3.7.: Darstellung des BSP Modells. Jeder Prozesse P_i wird am Ende eines Supersteps synchronisiert (Entnommen Nisar u. a., 2013)

Diese Synchronisation wird jedoch schnell zum Flaschenhals, da in jeder Iteration sämtliche Prozesse auf die Terminierung des langsamsten Prozesses warten müssen (vgl. Nisar u. a., 2013). Dem gegenüber stehen jedoch nach Xin u. a. (2014) deutliche Vorteile. Begründet dadurch, dass eine deterministische Verarbeitung das Debuggen und die Fehlertoleranz erheblich vereinfacht.

Zudem werden am Ende eines Supersteps sämtliche Zustände gespeichert. Im Fehlerfall muss nur der aktuelle Superstep für betroffene Daten wiederholt werden. In einem asymmetrischen Verarbeitungsmodell werden vergleichsweise Prozesse erst angestoßen, sobald Rechenkapazität zur Verfügung steht und diese in Abhängigkeit zu anderen Prozessen parallel ausgeführt werden können. GraphX unterstützt neben dem synchronen auch die Entwicklung von asynchronen Verarbeitungsmodellen (vgl. Shang und Yu, 2014).

Pregel unterliegt auch gewissen Einschränkungen, denn es lässt sich nicht jeder beliebige Graphalgorithmus als Pregel-Algorithmus formulieren. Die Kommunikation ist beispielsweise nur zwischen adjazenten Knoten vorgesehen. Besteht keine direkte Verbindung in Form einer Kante, ist eine Kommunikation nur schwer realisierbar. Zudem beschreiben Pregel-Algorithmen mehrere Iterationen auf statischen Graphstrukturen. Es ist nicht möglich, Knoten oder Kanten zu entfernen oder hinzuzufügen.

Die in Kapitel 3.3 vorgestellten Operationen sind dabei mächtig genug, das Konzept von Pregel in weniger als 20 Zeilen zu implementieren. Die Implementierung ist in der gleichnamigen Klasse *Pregel* verfügbar. Deren *apply*-Methode erwartet bei dem Aufruf sieben Argumente:

- Einen Graphen auf dem operiert wird.
- Eine initiale Nachricht, die zu Beginn an alle Knoten geschickt wird.
- Die Obergrenze der Anzahl an durchlaufenden Supersteps, da nicht jeder Pregel-Algorithmus besonders bei Optimierungsproblemen ohne diese Einschränkung von sich aus terminieren muss.
- Die Spezifizierung über welche Kanten Nachrichten versandt werden sollen.
- Eine Funktion, die das Vertexprogramm spezifiziert.
- Eine Funktion, die spezifiziert an welche Knoten Nachrichten verschickt werden sollen und was diese beinhalten.
- Eine Funktion die mehrere Nachrichten, die innerhalb eines Supersteps den gleichen Zielknoten besitzen, zu einer einzigen Nachricht akkumuliert.

Pregel wird im Kapitel 4 unter anderem als Grundlage für eigene Implementierungen herangezogen.

4. Erweiterung des GraphX Frameworks

Nachdem ein ausführlicher Überblick über das GraphX Framework gegeben wurde, sollen nun diese Mittel für eine eigenständige Entwicklung von Algorithmen genutzt werde. Dabei liegen die Algorithmen bereits in Form einer groben Beschreibung durch eine Veröffentlichung der jeweiligen Autoren vor. In diesem Abschnitt werden zwei verschiedene Algorithmen vorgestellt und anschließend implementiert und experimentell auf ihre Skalierung getestet.

Zu Beginn wird in Abschnitt 4.1 der Auswahlprozess der gewählten Algorithmen erläutert. Daraufhin wird in Abschnitt 4.2 und 4.3 der Prozess der Implementierung dieser, von einem ausgehenden Konzept, über die Beschreibung der Umsetzung, bis hin zu einer experimentellen Laufzeitbetrachtung beschrieben. Abschnitt 4.4 beinhaltet ein abschließendes Fazit.

4.1. Auswahl der Algorithmen

Die Motivation hinter einer Implementierung von Algorithmen im GraphX Framework liegt darin, die Möglichkeiten des Frameworks darzustellen und etwaige Schwächen zu identifizieren. Hierzu zählen sowohl die knotenorientierte Sicht auf die zugrundeliegenden Daten, als auch die zur Verfügung gestellten Operationen. Um dieses Ziel zu erreichen erhält die Auswahl der Algorithmen eine bedeutsame Rolle.

Ausgangspunkt einer Implementierung stellen aktuelle Probleme der Web 2.0 Bewegung dar, welche auch intensiv zur Datenerhebung genutzt werden. Der hier gebildete Schwerpunkt liegt auf der Analyse sozialer Netzwerke und des Güterhandels aus Händlersicht. Mit Blick auf das GraphX Framework ist die Situation gegeben, dass lediglich Pregel als einziges iteratives Verarbeitungsmodell zur Verfügung gestellt wird. Dieses bildet ein unmittelbar nutzbares Konstrukt, das häufig die erste Anlaufstelle für ein nachrichtenorientiertes Modell darstellt. Ein häufiges Anwendungsgebiet für die Nutzung von Pregel sind Verfahren der Clusteranalyse. Ein solches wurde bisher auch nicht in GraphX implementiert und stellt somit den ersten Kandidaten dar. Aufgrund dessen soll das Verfahren *Semi-Clustering* implementiert werden. Dieses Problem behandelt die iterative Verarbeitung eines statischen Graphen.

Im Gegensatz dazu fordern viele andere Algorithmen eine Manipulation der Graphstruktur. Hierzu muss auf die sonstigen Operationen von GraphX zurückgegriffen werden. Als zweites

wird ein Verfahren für *kollaboratives Filtern* implementiert. Dieses Problem siedelt sich im Bereich des Güterhandels an. Aufgabe ist es, Produktvorschläge auf Basis des Nutzerverhaltens zu bestimmen. Diese sind vergleichbar mit neuen Kanten zwischen Personen und Gütern, die in ein bestehendes Netzwerk integriert werden.

4.2. Semi-Clustering

Semi-Clustering ist ein agglomeratives Cluster-Verfahren. Zudem ist vorgesehen, dass Knoten auch Bestandteile mehrerer verschiedener Cluster sein können. Das Verfahren wird von Malewicz u. a. (2010) beschrieben. Die dortige Beschreibung dient als Anhaltspunkt für eine Implementierung aufbauend auf GraphX. Entwickelt wurde das Verfahren für die Analyse von sozialen Netzwerkstrukturen. Ziel ist es, mehrere Gruppierungen zu ermitteln, die untereinander in einer verhältnismäßig ausgeprägten Interaktionen zueinander stehen.

Abschnitt 4.2.1 legt das Konzept des Algorithmus dar. Daraufhin werden in Abschnitt 4.2.2 erste Überlegungen bezüglich der Umsetzung in GraphX diskutiert. Es folgt in Abschnitt 4.2.3 die Darstellung der gewählten Architektur. In Abschnitt 4.2.4 wird auf die Nutzung des Pregel Modelles für die Implementierung eingegangen. Abschließend wird in Abschnitt 4.2.5 die Laufzeit der Implementierung experimentell aufgezeigt.

4.2.1. Konzept

Ausgangspunkt ist ein ungerichteter Graph, der eine soziale Struktur vertritt. Die Knoten des Graphen repräsentieren Personen, die Kanten stellen die Beziehungen zwischen ihnen dar. Eine Kante kann beispielsweise für eine Aktion, wie das Hinzufügen als Freund oder das Verhalten in Form von Nachrichtenaustausch stehen (vgl. Malewicz u. a., 2010). Die Intensität einer Kante kann durch eine Gewichtung ausgedrückt werden. Ein hoher Wert steht für eine intensivere Kommunikation. Alternativ werden alle Kanten gleich gewichtet, sollte eine Gewichtung irrelevant sein und nur die Verbindungen selbst im Vordergrund stehen. Ziel ist es, Gruppierungen (Cluster) von Personen zu finden, die untereinander verhältnismäßig viel interagieren und weniger mit Außenstehenden. Personen können durchaus zu mehreren verschiedenen Gruppierungen gehören.

Beim Semi-Clustering ist ein Cluster eine Ansammlung von Knoten und einem zusätzlichen Wert S, der sie untereinander vergleichbar macht. Dieser ist für ein Cluster c folgendermaßen definiert:

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

Wobei I_c die Summe aller Gewichtungen der Kanten ist, die zwei Knoten verbinden, welche beide Bestandteile des Clusters c sind. B_c ist hingegen die Summe aller Gewichtungen der Kanten, die aus dem Cluster c herausragen. Dieses liegt vor, wenn entweder Anfangs- oder Endknoten einer Kante nicht Bestandteile des Clusters sind. Die Bedeutsamkeit von B_c kann durch den Faktor f_B bestimmt werden. Dieser kann vom Anwender definiert werden und sollte im Intervall $[0, 1]$ liegen. V_c ist die Anzahl der Knoten im Cluster c.

Der Wert eines Clusters S_c ist durch die Anzahl der Knoten innerhalb einer Clique der Größe V_c, normalisiert. Damit Cluster aus vielen Knoten nicht eine zu hohe Bewertung erhalten und eine Expansion nicht überwiegend zu einer Verbesserung führt (vgl. Malewicz u. a., 2010). Sei $G = (V, E)$ ein ungerichteter Graph. Eine Clique $C = (V_C, E_C)$ mit $n = |V_C|$ ist ein Teilgraph von G für den gilt, dass je zwei Knoten aus V_C benachbart sind (vgl. Diestel, 2010). Es gilt also $E_C = \mathcal{P}_2(V_C)$. Die Anzahl der 2-elementigen Teilmengen einer n-elementigen Menge lässt sich durch den Binomialkoeffizient ausdrücken. Es gilt für den Divisor von S also:

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

Semi-Clustering wurde basierend auf dem Pregel-Verarbeitungsmodell entwickelt und durchläuft somit eine Sequenz von Supersteps. Operiert wird auf einem Property-Graphen. Alle Knoten haben als Property eine Liste von Clustern, die anfangs leer ist. Die Property der Kanten ist eine Gewichtung. Zu Beginn fügt sich jeder Knoten in seine Liste als Cluster der Größe eins hinzu. Der Wert S jedes der anfangs $|V|$ Cluster beträgt initial eins. Anschließend verschickt jeder Knoten seine Clusterliste an alle seine Nachbarknoten und stößt so eine Sequenz von Supersteps an, die folgende Schritte durchlaufen (vgl. Malewicz u. a., 2010).

(1) Jeder Knoten v iteriert über die eingehenden Cluster $c_1, ..., c_k$. Wenn v noch nicht Bestandteil eines Clusters c ist, wird ein neues Cluster c' erzeugt, das c um v erweitert.

(2) Alle Cluster $c_1, ..., c_k, c'_1, ..., c'_k$ werden nach Wert absteigend sortiert. Nur die m besten Cluster werden an $N(v)$ verschickt, wobei m eine vom Nutzer definierte Zahl ist.

(3) Jeder Knoten v aktualisiert seine Liste von Clustern zu denen aus $c_1, ..., c_k, c'_1, ..., c'_k$, die v beinhalten.

Der Algorithmus terminiert wenn sich kein Cluster mehr verändert oder die maximale Anzahl an vorgegebenen Supersteps erreicht wurde. Ein Knoten geht demzufolge in den Zustand inaktiv, wenn sich in Phase (1) eines Supersteps keine hinzukommenden Cluster ergeben. Sämtliche Cluster können am Ende der Berechnung in eine globale Liste aggregiert werden. Diese stellt das Resultat des Algorithmus dar. Es besteht die Möglichkeit, über gewisse Parameter die Verarbeitung zu beeinflussen.

- C_{max} definiert die maximale Anzahl an Clustern, die in die Ausgabe übernommen werden.
- V_{max} definiert die maximale Anzahl an Knoten, die sich innerhalb eines Clusters befinden können.

Semi-Clustering gehört zu der Klasse der gierigen Algorithmen. Es wird lediglich zum Zeitpunkt jedes Supersteps das dortige Maximum aus der Sicht eines Knotens berechnet. Das Ergebnis ist eine korrekte Lösung. Es muss sich jedoch nicht zwangsläufig um die beste existierende Lösung handeln. Eine grafische Einteilung eines Graphen in Cluster ist in Abbildung 4.1 vorgenommen worden.

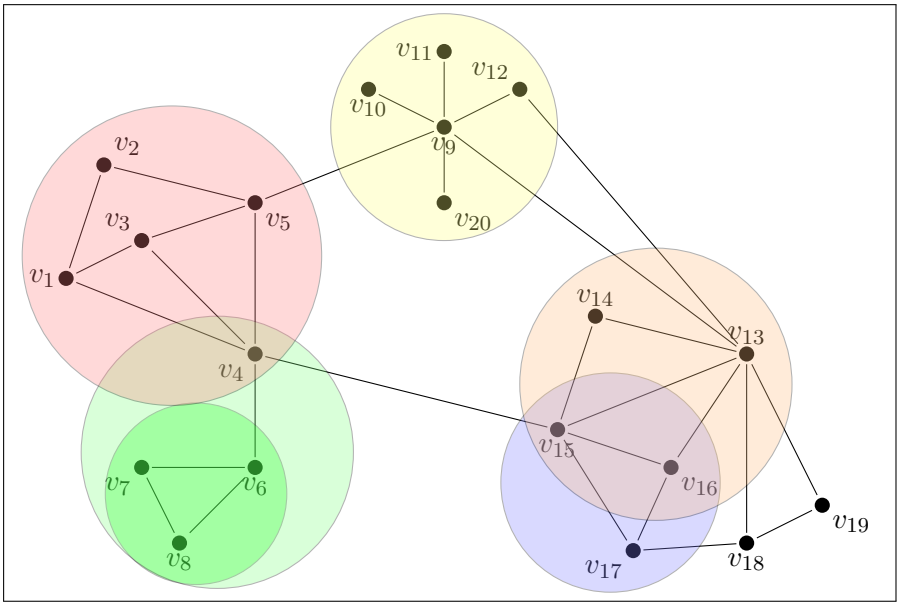


Abbildung 4.1.: Eine graphische Darstellung von Clustern durch das Verfahren des Semi-Clustering

4.2.2. Vorbereitende Maßnahmen

Die kurze Beschreibung des Algorithmus von Malewicz u. a. (2010) wird dort aufgeführt, um zu zeigen, welche häufig auftretenden Probleme durch das Pregel-Verarbeitungsmodell hervorragend gelöst werden können. Im Folgenden soll eine Implementierung dieses Problems aufbauend auf GraphX vorgestellt werden. Es wird dabei die GraphX Pregel-Schnittstelle herangezogen.

Der Algorithmus operiert auf der Struktur eines ungerichteten Graphen. Zudem sind die Kanten gewichtet. Dieses bedeutet, dass mit ihnen jeweils ein numerischer Wert assoziiert wird. Um einen solchen Graphen in GraphX zu simulieren, müssen Vorarbeiten geleistet werden. Eine direkte Umsetzung von ungerichteten Kanten ist in GraphX nicht möglich. Dies folgt daraus, dass auf der Struktur eines Property-Graphen gearbeitet wird und dieser nach Definition nur gerichtete Kanten enthält. Von Malewicz u. a. (2010) wird vorgeschlagen, eine ungerichtete Kante durch exakt zwei gerichtete Kanten zu simulieren. Dieser Ansatz wird hier nicht verfolgt, da in der Pregel-Implementierung von GraphX eine Nachricht auch entgegen der Kantenrichtung verschickt werden kann. Eine Unterteilung in zwei Kanten ist so nicht notwendig und die Anzahl der Kanten wird zudem halbiert. Für die Strukturierung des zugrundeliegenden Graphen müssen entsprechend alle Gewichtungen der Kanten zwischen je zwei Knoten summiert und durch eine einzige Kante mit beliebiger Richtung und dieser Summe als neue Gewichtung ersetzt werden. Um dieses Umzusetzen verfügt ein RDG über die Methode *convertToCanonicalEdges*. Diese tauscht die Anfangs- und Endknoten der Kanten, wenn die Identifizierung des Anfangsknotens größer als die des Endknotens ist. Die dadurch entstehenden Mehrfachkanten werden durch eine vom Nutzer definierte Funktion durch ein *reduceByKey* aggregiert. Des Weiteren werden auch Schlingen vernachlässigt. Das Augenmerk liegt nur auf der Beziehung verschiedener Entitäten. Der Strukturierungsprozess ist in Abbildung 4.2 dargestellt.

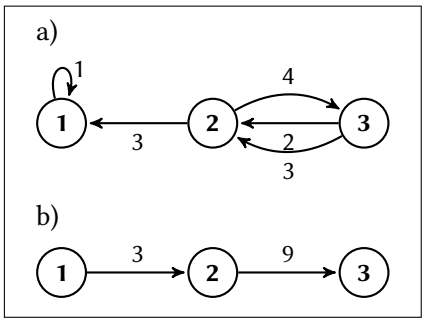


Abbildung 4.2.: Ein unstrukturierter Graph (a) und ein strukturierter Graph (b)

Ausgehend von einem nun strukturierten Property-Graph $G = (V, E, (P_V, P_E))$ mit $P_E : E \rightarrow Double$ als Kantengewichtung, geht es im Folgenden um die Entwurfsentscheidungen, die im Laufe der Realisierung getroffen wurden.

Für den Gebrauch der Pregel-Schnittstelle müssen drei Funktionen implementiert werden. Dies sind die Funktionen: *vprog*, *sendMsg* und *mergeMsg*. Die sequenzielle Ausführung dieser Funktionen stellt aus Anwendersicht ein Superstep dar. Die Ausführungsreihenfolge ist für sämtliche Supersteps identisch.

4.2.3. Architektur

Bevor mit der Entwicklung der *vprog* Funktion begonnen wird, sind grundlegende Überlegungen zu treffen. Der Algorithmenbeschreibung ist zu entnehmen, dass jeder Knoten über eine Liste von Clustern verfügt. Ein Cluster wiederum besteht aus einer Liste von Knoten und seiner Gewichtung S. Für die Repräsentation eines Clusters wurde die Klasse *SemiCluster* erstellt.

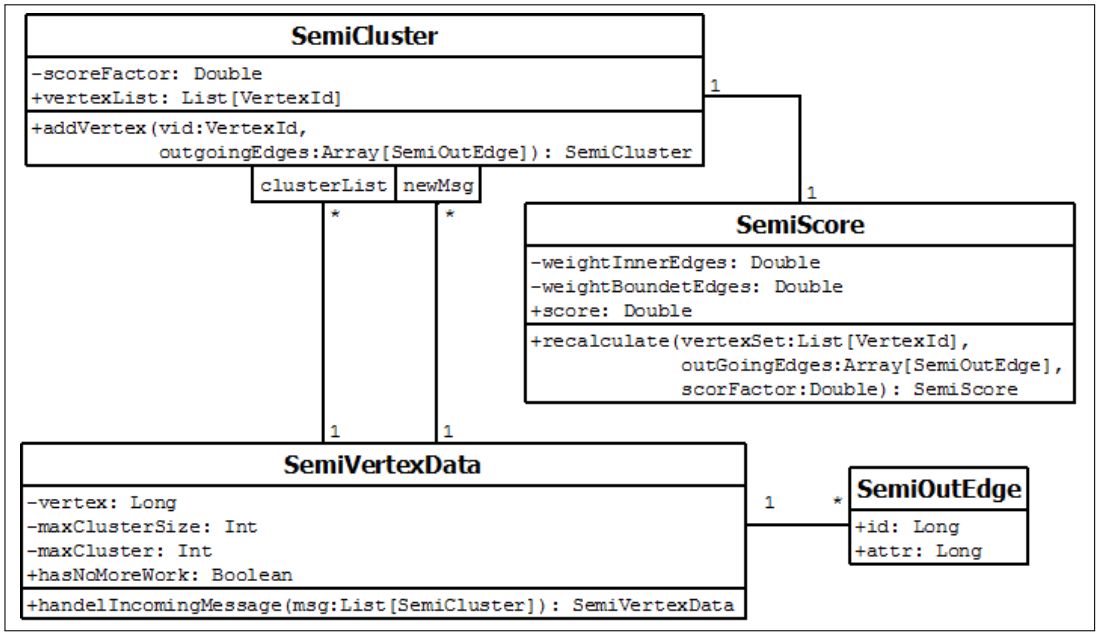


Abbildung 4.3.: Klassendiagramm: Semi-Clustering

Ein *SemiCluster* kapselt eine Liste von Knoten (*vertexList*), die sich innerhalb dieses Clusters befinden. Da in GraphX keine eigene Abstraktionsebene für einen Knoten existiert, wird ihre eindeutige Identifikation[1] als Repräsentation des Knotens verwendet. Im Laufe der Berechnung wird sich die Anzahl der Knoten nur erweitern. Eine Verkleinerung ist nicht möglich.

[1] Eine 64-Bit Integer Variable

Deshalb verfügt die Klasse lediglich über eine öffentliche *addVertex* Methode zur Aufnahme von Knoten. Das Attribut $semiScore$ hält den Wert S dieses Clusters. Um diesen nicht nach jeder Hinzunahme eines Knotens komplett neu berechnen zu müssen, werden die Bestandteile der Formel für spätere Berechnungen gespeichert. Für ein Cluster c entspricht I_c der Variable *weightInnerEdges*, B_c ist *weightBoundetEdges* und bei *scoreFactor* handelt es sich um f_B.

```
case class SemiScore(
   private val weightInnerEdges: Double = 0,
   private val weightBoundetEdges: Double = 0,
   val score: Double = 0) {

   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 4.1: Berechnung der Clustergewichtung

Die Berechnung wird folgendermaßen durchgeführt: Nach Definition ist der Wert S eines Clusters der Größe eins gleich eins. An dieser Stelle wird bereits die Summe aller ausgehenden Kanten des einen Knotens, der Variable *weightBoundetEdges* zugewiesen. Wird ein weiterer

Knoten in das Cluster aufgenommen, muss geprüft werden, ob Kanten, die vorher aus dem Cluster ragten, dieses immer noch tun. Hierzu kommt die Kategorisierung der, mit dem neu hinzugefügten Knoten assoziierten Kanten, in innere und herausragende Kanten. Sei v der zuletzt hinzugefügte Knoten des Clusters c. Für jeden adjazenten Knoten $u \in N(v)$ muss geprüft werden, ob u Bestandteil von c ist. Wenn ja, handelt es sich bei der Kante zwischen v und u, diese sei e, um eine in c liegende Kante. Die Variable *weightInnerEdges* wird um $P_E(e)$ erhöht. Da die Kante e bereits in früheren Berechnungen, frühestens bei der Hinzunahme des ersten Knotens, als aus c herausragend klassifiziert wurde, wird dieses durch die Verminderung von *weightInnerEdges* um $P_E(e)$ korrigiert. Ist u nicht in c enthalten, handelt es sich bei e um eine herausragende Kante und es wird entsprechend verfahren.

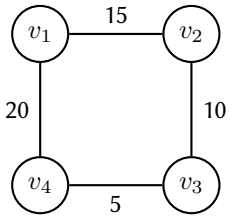
Bei der Berechnung wurde vorausgesetzt, dass jeder Knoten die Menge seiner Nachbarn inklusive der beteiligten Kanten kennt (hier: *outGoingEdges*). Aufgrund einer fehlenden Knoten-Abstraktion ist dieses nicht unmittelbar gegeben. Deshalb müssen diese Informationen auch ein Teil der Knoten-Property werden, um auf sie in der Berechnung zugreifen zu können.

Jede Knoten-Property soll genau durch ein Objekt der Klasse *SemiVertexData* repräsentiert werden. Die Klasse enthält alle Informationen auf welche im *vprog* zugegriffen werden soll. Dazu gehören, die bereits erwähnte Liste des Typs *SemiCluster*, die Identifizierung des Knotens, die Menge aller Nachbarknoten und die Nachricht, die im nächsten Superstep verschickt werden soll. Dass die zu versandten Nachrichten ein Attribut einnehmen und sich nicht aus der Liste der SemiCluster bei Bedarf nachträglich berechnet werden können, liegt an der Reihenfolge der Verarbeitungsschritte von Pregel. Die Funktion *vprog* endet mit der Aktualisierung der Property des betreffenden Knotens. Nachrichten werden anschließend durch den internen Aufruf von *sendMsg* getätigt. Da die Nachricht nach Beschreibung eine Obermenge der Cluster-Liste ist und auch nicht aus ihr ableitbar ist, muss sie bereits vorher innerhalb von *vprog* bestimmt werden. Die Beschreibung von Malewicz u. a. (2010) basiert auf einer anderen Pregel-Architektur. Diese sieht lediglich eine *compute*-Funktion vor, welche das Verhalten von *vprog*, *sendMsg* und *mergeMsg* in einer Funktion vereint und somit andere Möglichkeiten bietet. Die Menge der Nachbarn eines Knotens lässt sich für alle Knoten gleichzeitig durch die Methode *collectEdges* der RDG Abstraktion berechnen.

4.2.4. Umsetzung

Der Ablauf lässt sich mit Bezug auf die Implementierung folgendermaßen beschreiben. Die Berechnung wird durch eine initiale Nachricht, hier eine leere Liste, angestoßen. Es wird daraufhin die Funktion *vprog* für jeden Knoten ausgeführt. Innerhalb von *vprog* besteht nun Zugriff auf die dem jeweiligen Knoten zugestellten Nachrichten. Die Nachricht ist eine weitere

Liste von Clustern. Über diese wird nun iteriert und wenn möglich wird der Knoten auf welchem das *vprog* ausgeführt wird als weiteres Bestandteil hinzugefügt. Die dadurch neu erzeugten Cluster werden für einen Graphen in Abbildung 4.4 tabellarisch dargestellt. Diese werden dort dem jeweiligen Superstep für jeden Knoten gegenübergestellt. Die neu berechneten Cluster bilden mit den Clustern aus allen vorherigen Supersteps die neue Liste von Clustern.



	v_1	v_2	v_3	v_4
S_0	$(v_1\|1)$	$(v_2\|1)$	$(v_3\|1)$	$(v_4\|1)$
S_1	$(v_1, v_4\|18)$ $(v_1, v_2\|12)$	$(v_1, v_2\|12)$ $(v_2, v_3\|8)$	$(v_2, 3\|8)$ $(v_3, v_4\|2)$	$(v_1, v_4\|18)$ $(v_3, v_4\|2)$
S_2	$(v_1, v_2, v_3\|7.5)$ $(v_1, v_3, v_4\|7.5)$ $(v_1, v_3\|\text{-}5)$	$(v_1, v_2, v_4\|11.2)$ $(v_2, v_3, v_4\|3.8)$ $(v_2, v_4\|\text{-}5)$	$(v_1, v_3, v_4\|11.2)$ $(v_1, v_2, v_3\|7.5)$ $(v_1, v_3\|\text{-}5)$	$(v_1, v_2, v_4\|11.2)$ $(v_2, v_3, v_4\|3.8)$ $(v_2, v_4\|\text{-}5)$
S_3	$(v_1, v_2, v_4\|11.2)$ $(v_1, v_2, v_3, v_4\|8.3)$	$(v_1, v_2, v_3\|7.5)$ $(v_1, v_2, v_3, v_4\|8.3)$	$(v_2, v_3, v_4\|3.8)$ $(v_1, v_2, v_3, v_4\|8.3)$	$(v_1, v_3, v_4\|7.5)$ $(v_1, v_2, v_3, v_4\|8.3)$

Abbildung 4.4.: Semi-Clustering Verfahren. Dargestellt wird, wie sich die Menge der Cluster nach jedem Superstep für einen exemplarischen Graphen erweitert. Jedes $(list|S)$ Tupel ist ein Cluster, wobei $list$ die Menge der Knoten im Cluster und S den Wert des Clusters mit $f_B = 0.1$ entspricht.

Bei der in diesem Superstep zu verschickenden Nachricht handelte es sich um die Vereinigung, der in diesem Superstep eingegangenen Nachricht und den neu berechneten Clustern. Ausgehend von einem Knoten werden Nachrichten unabhängig von der Kantenrichtung zu seinen Nachbarn geschickt. Dies wird unter anderem dadurch ermöglicht, dass die *sendMsg*-Funktion auf die Triple-Repräsentation der jeweiligen Kante zugreifen kann. Damit besteht sowohl Zugriff auf die Identifizierung des Anfangs- und Endknotens, die für eine Nachrichtenzustellung vorliegen müssen. Nachrichten werden in Form eines Iterators vom Typ $[(VertexId, M)]$ verschickt. Wobei $VertexId$ der Typ der Identifikation des Knotens ist, an den diese Nachricht adressiert ist und M der Typ der Nachricht. Dieses entspräche hier $List[SemiCluster]$. Der Iterator ist der Rückgabewert der *sendMsg* Funktion. Die Rückgabe einer Iterator-Instanz ohne Elemente würde dem Versenden keiner Nachricht entsprechen.

Es werden von einem Knoten auch Cluster verschickt, von denen er jedoch kein Bestandteil ist. Dies führt auch zu der Möglichkeit, dass ein Cluster aus mindestens einem Knoten besteht, der sich nicht in einer Nachbarschaftsrelation zu mindestens einem Knoten des Clusters befindet. In Abbildung 4.4 wird zum Zeitpunkt S_2 das Cluster (v_1, v_3) erzeugt, obwohl sich v_1 und v_3 nicht über eine Kante erreichen können. Dieses liegt zum einen an der Ausbreitungsgeschwindigkeit von einer Kante pro Superstep, zum anderen daran, dass sämtliche auf diesem Pfad erzeugten Cluster wiederholt verschickt werden. Aus Sicht der Performanz ist die Begrenzung der sich innerhalb einer Nachricht befindenden Cluster notwendig. Sollten innerhalb eines Supersteps mehrere Nachrichten an denselben Knoten adressiert worden sein, werden die einzelnen Nachrichten zusammengefasst und auf die höchst gewichteten Cluster beschränkt.

4.2.5. Experimentelle Laufzeituntersuchung

Das Laufzeitverhalten der Implementierung des *Semi-Clustering* Algorithmus soll nun experimentell dargelegt werden. Im Mittelpunkt der Analyse steht zum einen die Skalierbarkeit des Algorithmus durch Hinzunahme von *Worker*-Instanzen, zum anderen die Laufzeitentwicklung bei Hinzunahme von Knoten und Kanten in die Graphstruktur.

Die Messungen werden zum einen in einem Spark-Cluster unter Verwendung des *Stand-Alone Cluster Managers* durchgeführt. Es wird die Spark Version 1.5.1 genutzt. Das Cluster besteht aus bis zu vier Worker-Instanzen mit je 1GB RAM und mit Zugriff auf je einen Kern. Diese werden auf unterschiedlichen Maschinen gestartet. Der Graph wird durch einen zufälligen Knotenschnitt auf sämtliche *Worker* partitioniert. Zum anderen wird, wenn vorab nicht explizit darauf hingewiesen, der lokaler Modus von Spark verwendet. Dieser läuft auf einer Maschine unter der Verwendung von 3 Kernen und bis zu maximal 8GB RAM. Die abgebildeten Laufzeiten beziehen sich lediglich auf die Ausführung des Algorithmus. Die Zeit für das Erstellen des Clusters oder die Generierung der Testdaten wird nicht berücksichtigt. Alle Messungen wurden in fünffacher Ausführung durchgeführt. Die jeweils abgebildete Laufzeit entspricht dem arithmetischen Mittel der fünf Durchläufe. In Balkendiagrammen ist diese explizit angegeben. Das Intervall der Streuung wird ebenso abgebildet. Dazu zeigt das Symbol $\top$ die maximale, bzw. $\bot$ die minimale Abweichung zum arithmetischen Mittel an. Die einzelnen Messwerte zu den Abbildungen werden im Anhang B tabellarisch aufgeführt.

Für die Generierung der Testdaten wird auf die eigens von GraphX bereitgestellten Möglichkeiten zurückgegriffen. GraphX liefert verschiedene Graphgeneratoren, die zufällige Testgraphen erzeugen können. Die Anzahl der Kanten kann als absoluter Wert angegeben oder aus einer Verteilung ermittelt werden. Für die folgenden Experimente wird eine log-normal

Verteilung des Ausgangsgrads der Knoten, mit der nachfolgenden Dichtfunktion verwendet (vgl. Malewicz u. a., 2010).

$$f(d|\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma d} e^{-(\ln d - \mu)^2/2\sigma^2}$$

Der Erwartungswert $\mu = 4$ und die Standardabweichung $\sigma = 1,3$ werden als *Default*-Angaben des Generators übernommen. Die Verteilung ist dem natürlichen Graphen nachempfunden und eignet sich als realistisches Testobjekt. Mehrfachkanten wurden zudem bereits aggregiert.

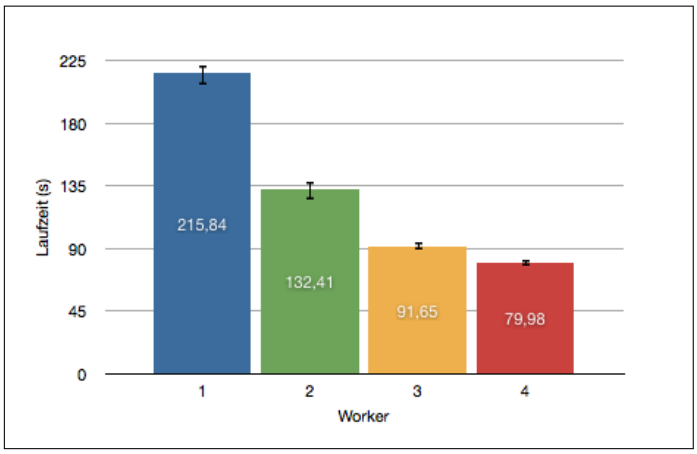


Abbildung 4.5.: Semi-Clustering: Skalierung durch Worker bei einer Graphgröße von 3.000 Knoten

Abbildung 4.5 zeigt die Skalierbarkeit des Semi-Clustering Algorithmus durch Hinzunahme von Worker-Instanzen. Es wird jeweils auf einer gleichbleibenden Graphstruktur mit einer variablen Anzahl von Worker-Instanzen operiert. Dargestellt wird die durch Spark verbundene Möglichkeit, die Last durch die Ausführung im Cluster auf mehrere Maschinen zu verteilen. Zu erkennen ist das erwartete Szenario, dass die Erhöhung der Rechnerkapazitäten in Form von Worker-Instanzen zu einer Verringerung der Ausführungszeit führt. Eine Parallelisierung wirkt sich somit durchaus positiv auf die Laufzeit aus. Die Laufzeiten verhalten sich durch die Zunahme weiterer Worker-Instanzen nicht linear. Es ist zu erkennen, dass die Laufzeiteinsparung durch Verwendung von mehreren Worker-Instanzen sich zunehmend sättigt. Dies zeigt, dass eine Verteilung mit zusätzlicher Last verbunden ist, welche mit zunehmender Parallelität zunimmt. Unter der Annahme, es gäbe keine Kosten für die Verteilung und keine Kommunikation zwischen den Partitionen, wäre eine lineare Verbesserung durchaus denkbar.

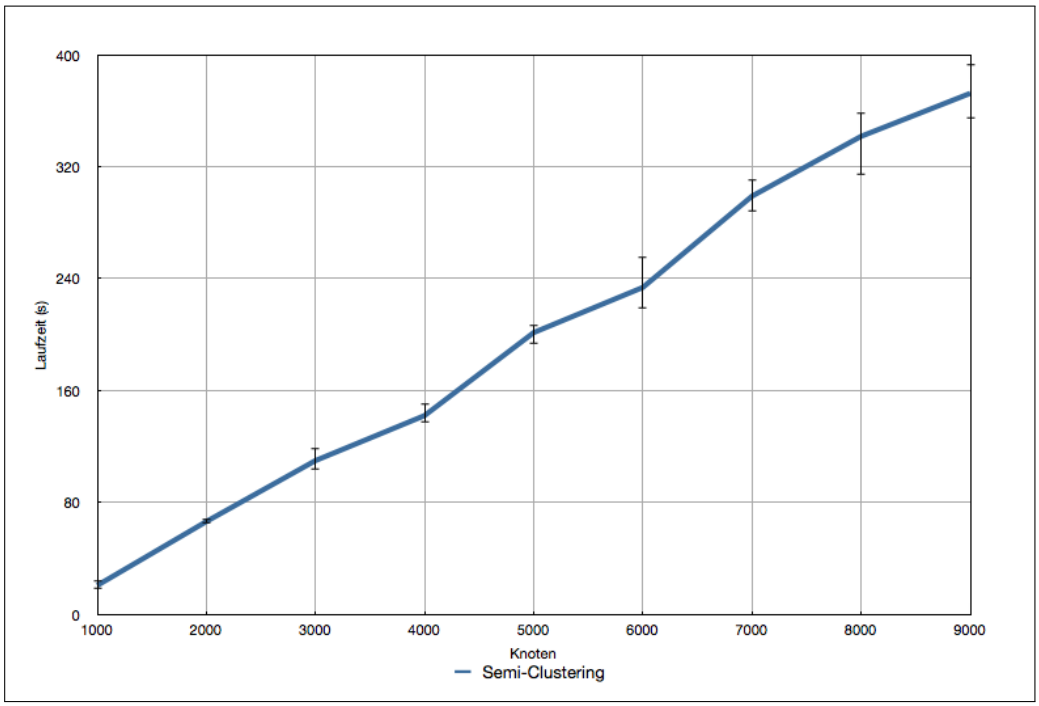


Abbildung 4.6.: Semi-Clustering: Skalierung durch Graphgröße

Um die Skalierung des Algorithmus durch die Vergrößerung des zugrundeliegenden Graphen zu demonstrieren, werden in Abbildung 4.6 Laufzeiten auf verschiedenen Graphen dargestellt. Diese Tests wurden im lokalen Modus ausgeführt. Die Anzahl der Knoten variiert von 1000 bis 9000. Es ist zu sehen, dass die Laufzeit nahezu linear in Abhängigkeit zu der Anzahl der Knoten wächst. Eine genaue Auflistung der Struktur von den verschiedenen Testdaten ist Abbildung 4.7 zu entnehmen.

$\|V\|$	$\|E\|$	$\|E\|/\|V\|$
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

Abbildung 4.7.: Struktur der Testdaten

4.3. Kollaboratives Filtern

Kollaboratives Filtern (vgl. Huang u. a., 2005) beschäftigt sich mit der Ermittlung von Nutzerpräferenzen auf Basis des vergangenen Nutzerverhaltens. Ziel ist es, Produktempfehlungen für potenzielle Käufer zu generieren. Dazu werden Graphen, deren Knoten Personen und Kanten Güterinteraktionen darstellen, analysiert. Das im Folgenden vorgestellte Verfahren basiert zudem auf dem *Link Prediction Problem* (vgl. Liben-Nowell und Kleinberg, 2003).

Abschnitt 4.3.1 wird sich zu Anfang mit dem Konzept des kollaborativen Filterns annehmen. Daraufhin werden in Abschnitt 4.3.2 verschiedene Metriken vorgestellt, welche die Qualität einer Voraussage quantifizieren. Die Umsetzung des Algorithmus in GraphX wird in Abschnitt 4.3.3 thematisiert. Abschließend wird in Abschnitt 4.3.4 auch für diesen Algorithmus eine experimentelle Laufzeitbetrachtung durchgeführt.

4.3.1. Konzept

Ausgangspunkt des hier betrachteten Problems sind Interaktionen zwischen Personen und Gütern. Diese können beispielsweise den Erwerb eines Produktes, Informationen oder Service darstellen (vg. Huang u. a., 2005). Die Relation von Personen und Gütern lässt sich durch einen *bipartiten* Graphen abbilden. Ein solcher Graph zeichnet sich dadurch aus, dass seine Knotenmenge in zwei disjunkte Mengen teilbar ist. Für einen *bipartiten* Graphen G ist seine Knotenmenge V in eine Menge von Personen U und I als Menge der Güter zweigeteilt (siehe Abb. 4.8). Zudem gilt: $V = U \cup I$ und $\emptyset = U \cap I$. Eine Kante von G verbindet somit immer eine Person $u \in U$ mit einem Gut $i \in I$.

Es gilt nun, zukünftige Personen/Güter-Paare auf Basis der Topologie des Graphen zu bestimmen. Diese Problematik lässt sich als *Link Prediction* Problem darstellen (vgl. Huang u. a., 2005). Als *Link Prediction* Problem bezeichnet man die Fragestellung, welche Kanten innerhalb eines Graphen in naher Zukunft eher entstehen werden (vgl. Liben-Nowell und Kleinberg, 2003). Dazu wird eine Momentaufnahme eines Graphen $G = (V, E_{old})$ zu einem Zeitpunkt t betrachtet. Gesucht ist eine möglichst genaue Vorhersage über die zu einem gegebenen Zeitpunkt t' mit $t < t'$ neu hinzukommenden Kanten $E_{new} \subseteq V \times V$ mit $E_{old} \cap E_{new} = \emptyset$. Für eine Prognose wird die Topologie des bestehenden Graphen herangezogen.

Durch die Darstellung der Personen/Güter-Beziehungen durch einen *bipartiten* Graphen entspricht die Ermittlung möglicher neuer Beziehungen dem *Link Prediction* Problem (vgl. Huang u. a., 2005). Entsprechend können deren Verfahren genutzt werden. Dazu zählt die Gewichtung der potenziellen Kanten durch verschiedene Metriken, die im Abschnitt 4.3.2 vorgestellt werden. Hoch gewichteten Kanten stellen erfolgversprechende Empfehlungen dar.

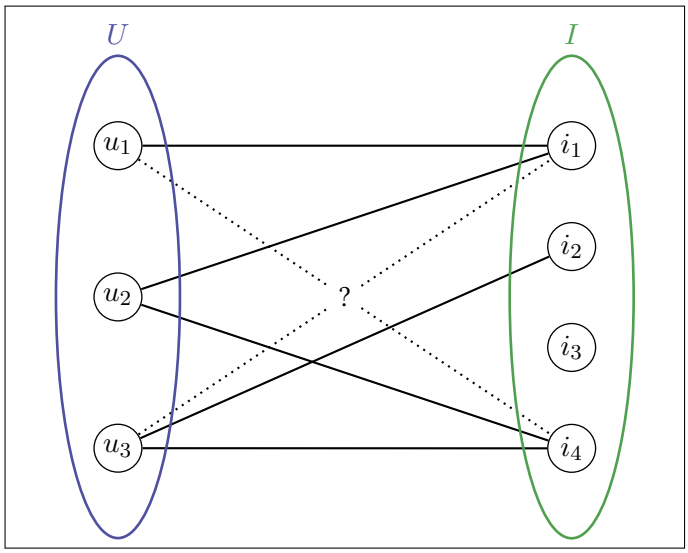


Abbildung 4.8.: Ein *bipartiter* Graph stellt die Relation von Personen (U) und Gütern (I) dar. Kollaboratives Filtern hat die Aufgabe Interessen zu prognostizieren.

4.3.2. Metriken

Für einen *bipartiten* Graphen $G = (U \cup I, E)$ mit $U \cap I = \emptyset$ werden alle noch nicht existierenden Kanten durch Metriken gewichtet. Diese $|U| * |I| - |E|$ Kanten werden durch eine Funktion $score : U \times I \rightarrow Double$ einen Wert zugewiesen. Eine Gewichtung ermöglicht es, mehrere Kanten untereinander zu vergleichen. Es wird jedoch nicht die direkte Wahrscheinlichkeit des Auftretens einer Kante in naher Zukunft berechnet, sondern nur ein Prognosemaß, welches eine Abwägung der Qualität einzelner Empfehlungen ermöglicht. Je höher die Gewichtung ausfällt, desto erfolgversprechender kann die Empfehlung betrachtet werden.

Um zu entscheiden, ob eine Kante für eine Person eine erfolgversprechende Empfehlungen darstellt, lässt sich anhand der Ähnlichkeit der Nutzerinteressen ermittelt. Für ein Personen/Gut Paar (u, i) kann das Nutzerverhalten der Person u und das Verhalten aller Käufer des Gutes i analysiert werden. Für eine Analyse wird folgende Annahme herangezogen: Je ähnlicher das bisherige Nutzerverhalten von Personen ist, desto wahrscheinlicher ist es, dass sie auch zukünftig die gleichen Güter kaufen. Nutzerpräferenzen lassen sich so auf andere Nutzer übertragen.

Im Folgenden werden einige Möglichkeiten der Gewichtung dargestellt (vgl. Huang u. a., 2005). Hinter den einzelnen Metriken verbirgt sich eine Variante dieser, die lediglich auf die Anwendung auf *bipartiten* Graphen angepasst wurde. Dabei sei für ein Gut i durch $\hat{N}(i) = \bigcup_{c \in N(i)} N(c)$ die Menge der Güter beschrieben, die von den Erwerbern von i zusätz-

lich erworben wurden[2]. Allgemein handelt es sich um eine vereinfachte Darstellung der Menge der Nachbarn von den Nachbarn eines Knotens.

Common Neighbours

Gemessen wird die Überlappung der Interessen von Person u mit denen Personen, die Gut i auch erworben haben. Die Gewichtung spiegelt die Anzahl der gemeinsamen Nutzerinteressen wider.

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

Jaccard's Coefficient

Bei der Gewichtung wird die durch die **Common Neighbours**-Metrik berechnete Kennzahl durch die Anzahl an betrachteten Elementen geteilt. Hierdurch wird das Ergebnis auf das Intervall $[0, 1]$ beschränkt.

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

Preferential Attachment

Die Gewichtung setzt sich aus der Aktivität der Personen und der Popularität des Gutes zusammen.

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

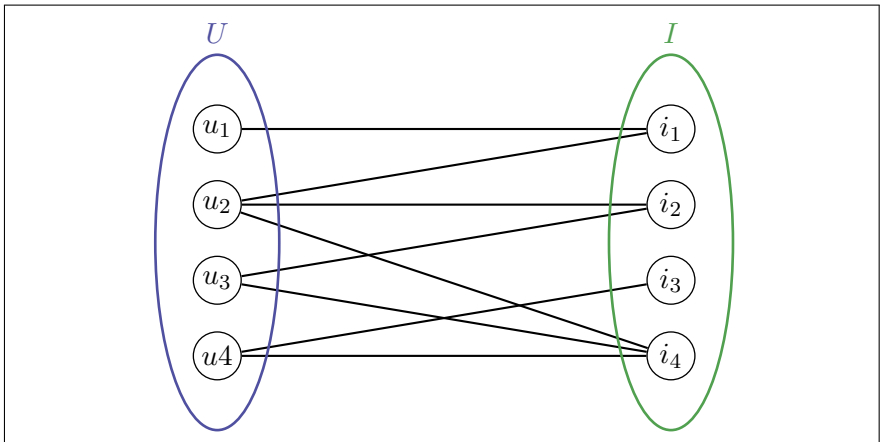
Adamic/Adar

Unterscheidet sich zur **Common Neighbours**-Metrik dahingehend, dass nicht die Elemente gezählt werden, sondern für jedes Element die Anzahl der Nachbarn im Mittelpunkt stehen.

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

Abbildung 4.9 zeigt die Anwendung des *kollaborativen Filterns* auf einem beispielhaft gewählten *bipartiten* Graphen. Die Kanten wurden jeweils einmal mit jeder der vorgestellten

[2] In Huang u. a. (2005) wird die Menge der Nachbarn von den Nachbarn eines Knotens x mit $\hat{N}(x) = \bigcap_{c \in N(x)} N(c)$ definiert. Die Modifikation der Definition ist notwendig, da bei den hier verwendeten Datensätzen nicht garantiert ist, dass sämtliche Nachbarn mit gleichen Elementen in Verbindung stehen. Eine Durchschnittsbildung würde bei schon einem von der Gesamtheit abweichenden Nachbarn keine Elemente mehr umfassen.



	Common Neighbours	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

Abbildung 4.9.: Gegenüberstellung der Gewichtungen verschiedener Metriken durch Anwendung des kollaborativen Filterns

Metriken gewichtet. Es wird für jede Metrik eine Liste aller Gewichtungen für die Kanten erstellt. Es obliegt dem Nutzer, welche n-besten Kanten in die Ergebnismenge der Vorschläge übernommen werden sollen. Im Beispiel wurde auf eine Einschränkung verzichtet. Zudem ist zusehen, wie sich die unterschiedlichen Perspektiven der Metriken auf das Resultat auswirken.

4.3.3. Umsetzung

Der Algorithmus operiert auf einem vom Nutzer bereitgestellten Property-Graphen $G = (V, E_{old}, P)$. Für eine Umsetzung wird zunächst davon ausgegangen, dass es sich bei diesem zugrundeliegenden Graphen um einen *bipartiten* Graphen handelt. Der Anwender hat hierfür Sorge zu tragen. Für den Algorithmus ist eine klare Unterteilung in Personen- und Güterknoten notwendig. Der Anwender hat hier wiederum die Aufgabe zuvor jede Knoten-

Property auf den Typ *Boolean* abzubilden. Für einen Knoten $v \in V$ ist diese folgendermaßen definiert:

$$P_V(v) = \begin{cases} true & \textit{für „v repräsentiert eine Person“} \\ false & \textit{für „v repräsentiert ein Gut“} \end{cases}$$

Somit ist eine Unterteilung in Personen (U) und Güter (I) für die Anwendung sichtbar. Diese Unterteilung ist für die Verwendung von verschiedenen Metriken notwendig.

Im nächsten Schritt gilt es die Menge aller noch nicht existierenden Kanten in G zu bestimmen. GraphX sieht keine Erweiterung der Kantenmenge eines Graphen vor und liefert keine entsprechenden Operationen. Es muss auf eine manuelle Erzeugung der Kanten auf der Ebene von einfachen RDD's zurückgegriffen werden. Aus dieser lässt sich anschließend ein neuer Graph erzeugen. Da RDD's Mengenoperationen unterstützen, lässt sich die Berechnung der Menge der potenziellen Kanten $E_{new} := (U \times I) \setminus E_{old}$ direkt auf Transformationen von RDD's übertragen.

Sämtliche Elemente von E_{new} werden durch eine Metrik gewichtet. Die Gewichtung selbst wird jeden dieser Kanten als Kanten-Property zugewiesen. Aus Sicht der Implementierung lassen sich diese in verschiedenen Kategorien einordnet. Diese beziehen sich darauf, auf welche Daten ein jeweiliger Knoten für die Berechnung Zugriff haben muss.

(i) Jeder Knoten kennt lediglich die Anzahl seiner Nachbarn. Ob der Knoten eine Person oder Gut repräsentiert ist irrelevant.

(ii) Jedem Personen-Knoten sind die Identifikationen seiner Nachbarn bekannt. Jeder Gut-Knoten kennt die Identifikationen der Nachbarn seiner Nachbarn. Sämtliche Knoten haben somit Kenntnisse über jeweils eine Menge von Gütern.

(iii) Letztere basiert auf der Anzahl der Nachbarn der Schnittmenge, der in (ii) beschriebenen Güter.

Kategorie (i) der Metriken lässt sich durch eine *join*-Operation auf der Menge der Knoten mit der jeweiligen Anzahl von Nachbarn realisieren. Jede RDG-Instanz hat auf diese Informationen anhand des *degrees*-Attributes der Klasse *GraphOps* direkten Zugriff. Die *Preferential Attachment*-Metrik verwendet genau dieses Vorgehen.

Kategorie (ii) nutzt die Unterteilung der Knoten in Personen und Güter. Die benötigten Kenntnisse der Personen über die Nachbarn lassen sich durch einen Aufruf von *collectNeighborIds* auf dem Graph (V, E_{new}) beschaffen. Diese weist jedem Knoten als Property ein Array mit den Identifikationen seiner Nachbarn zu. Für die Ermittlung der Güter-Knoten-Properties hingegen wird auf das Modell des Nachrichtenaustausches über Kanten zurückgegriffen. Es

wird die Idee verfolgt, dass jeder Personen-Knoten nun eine Nachricht mit seiner eben ermittelten Property an alle seine erworbenen Güter schickt. Die Vereinigung aller für ein Gut eintreffender Nachrichten entspricht der Nachbar-Identifikationen seiner Nachbarn. Dieses Verfahren wird von den Metriken *Common Neighbours* und *Jaccard's Coefficient* genutzt.

Die **Kategorie (iii)** nutzt die Eigenschaft, dass die gleiche RDG-Instanz für mehrere unterschiedliche Berechnungen verwendet werden kann. Zu Beginn wird wie in Kategorie ii verfahren. Es wird für jeden Knoten die Menge der Identifikationen seiner Nachbarn gesammelt. Die Property jedes Knotens ist initial von dem Typ $Array[VertexId]$. Auf dieser Struktur folgen zwei unterschiedliche Berechnungen. Hierbei werden die Knoten-Properties der Personen und der Güter in getrennten Berechnungen, aber auf gleicher Struktur ausgeführt. Diese werden daraufhin einer weiteren Verarbeitung zur Gewichtung einer Kante unterzogen.

Die Property der Kanten stellt sich aus einer Menge von Wertepaaren zusammen. Diese beinhalten für jeden Nachbarn seine Identifikation und Anzahl seiner Nachbarn. Dieses entspricht wiederholt einem einfachen Nachrichtenaustausch und lässt sich durch die Methode *aggregateMessages* formulieren. Die Properties der Güter setzen sich aus der Menge der Nachbar-Identifikationen zusammen. Somit ist die Situation gegeben, dass sowohl die Personen, als auch die Güter, die Menge seiner Nachbarn kennt. Es lässt sich nun die Schnittmenge beider Mengen berechnen. Für die Ergebnismenge liegen damit die Anzahl der Nachbarn vor und können in einer Berechnung verwendet werden. Beispielsweise in der Anwendung der *Adamic/Adar* Metrik.

Sämtliche hier implementierten Metriken basieren auf den in Kategorie (i - iii) dargestellten Daten. Da verschiedene Metriken auf gleichen Datenbeständen arbeitet, kann deren Implementierung wiederholt genutzt werden. Die genaue Berechnung kann als Funktion an den jeweiligen strukturierten Graphen geschickt werden. So lassen sich leicht weitere Metriken ergänzen.

4.3.4. Experimentelle Laufzeituntersuchung

Um auch die Laufzeit der Implementierung für das *kollaborative Filtern* in seinen Varianten experimentell zu untersuchen, sollen auch hier die Möglichkeiten der Testdatengenerierung von GraphX genutzt werden. Der Bedarf liegt auf der Erzeugung eines *bipartiten*-Graphen. GraphX bietet nicht die Möglichkeit einen solchen Graphen unmittelbar zu generieren. Es wird jedoch die Möglichkeit geboten, einzelne Kanten zwischen verschiedenen Mengen von Knoten-Identifikationen zu erstellen. Diese Eigenschaft eignet sich hervorragend für die Erzeugung von *bipartiten* Graphen. Aus diesen Kanten lässt sich auch ohne explizite Angabe von Knoten ein Graph erzeugen, da aufgrund der Konsistenz für jede Kante, Anfangs- und Endknoten

automatisch erzeugt werden. Diese sind lediglich nur noch in Personen und Güter zu unterteilen, welches Aufgrund der Zugehörigkeit der Identifizierung leicht möglich ist.

Für die Laufzeitexperimente wird ein *bipartiter*-Graph mit einer festen Anzahl an Güter-Knoten und wenn nicht anderes definiert, einer variablen Anzahl an Personen-Knoten generiert. Dies folgt der Annahme, dass innerhalb eines Web Shops die Anzahl der Nutzer über die Zeit hinweg einem stärkeren Wachstum unterliegt als das Sortiment. Es soll hiermit ein realitätsnäheres Verhalten abgebildet werden. In sämtlichen Experimenten wird eine Gütermenge von 750 Stück verwendet. Die Nachfrage regelt die Anzahl der Kanten und einspricht hier einer Normalverteilung mit einem Mittelwert von $\mu = 45$ und Standardabweichung von $\sigma = 15$ Einheiten.

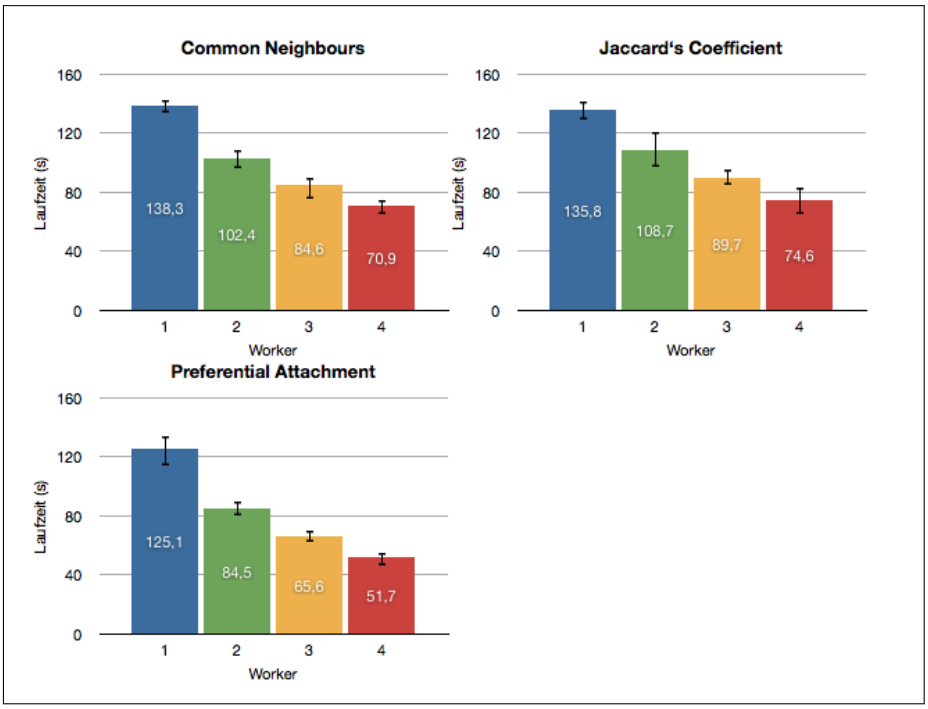


Abbildung 4.10.: Kollaboratives Filtern: Skalierung der Metriken durch Worker

Zu Beginn soll die Skalierung der einzelnen Metriken durch Hinzunahme von Worker-Instanzen verdeutlicht werden. Dazu wird die Anzahl der Personen auf 9.000 festgelegt. Für das Experiment wird das, bereits im vorherigen Experimenten (Abschnitt 4.2.5) genutzte Cluster verwendet. Die Ergebnisse sind in Abbildung 4.10 dargestellt. Es wiederholt sich auch hier das Bild, dass durch die Hinzunahme weiterer Worker-Instanzen keine lineare Verbesserung erzielt wird. Aber im Vergleich zum Semi-Clustering eine Sättigung der Laufzeiteinsparung früher stattfindet. Dies lässt auf eine bessere Parallelisierung des Semi-Clustering schließen.

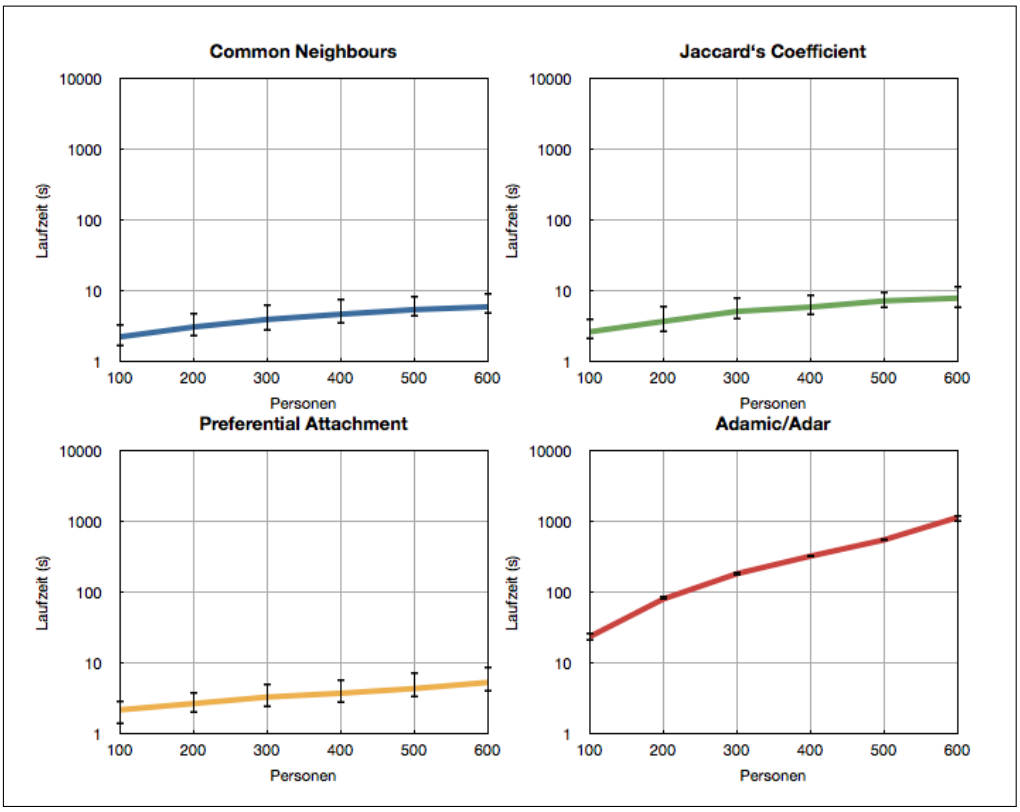


Abbildung 4.11.: Kollaboratives Filtern: Vergleich der Laufzeitentwicklungen der Metriken durch Hinzunahme von Personenknoten

Des Weiteren ist zu entnehmen, dass die Metriken *Common Neighbours* und *Jaccard's Coefficient* ein nahezu gleiches Verhalten zeigen. Dies ist aus der Sicht der Implementierung zu erwarten, da beide auf gleichen Ausgangsdaten basieren und sich nur in der endgültigen Berechnung unterscheiden, wie in Abschnitt 4.3.3 dargelegt. Wird die *Preferential Attachment*-Metrik in den Vergleich der beiden Vorherigen einbezogen, so zeigt sich hier eine deutlich geringere Laufzeit ab. Die Metrik setzt lediglich die Kenntnis des Grades für jeden Knoten voraus. Dies lässt sich durch eine einfache *join*-Operation abbilden und ist so von der Komplexität wesentlich einfacher zu realisieren.

Die *Adamic/Adar*-Metrik stellt sich in seiner Laufzeitbetrachtung als vergleichsweise aufwendig heraus. Das verwendete Cluster verfügte nicht über ausreichende Arbeitsspeicher, um eine Gewichtung mit der *Adamic/Adar*-Metrik durchzuführen. Aufgrund dieser Tatsache zeigt Abbildung 4.11 einen Überblick über die Laufzeit aller vier Gewichtungen. Diese nun in der ebenfalls in Abschnitt 4.2.5 genutzten lokalen Konstellation von Spark. Es zeichnet

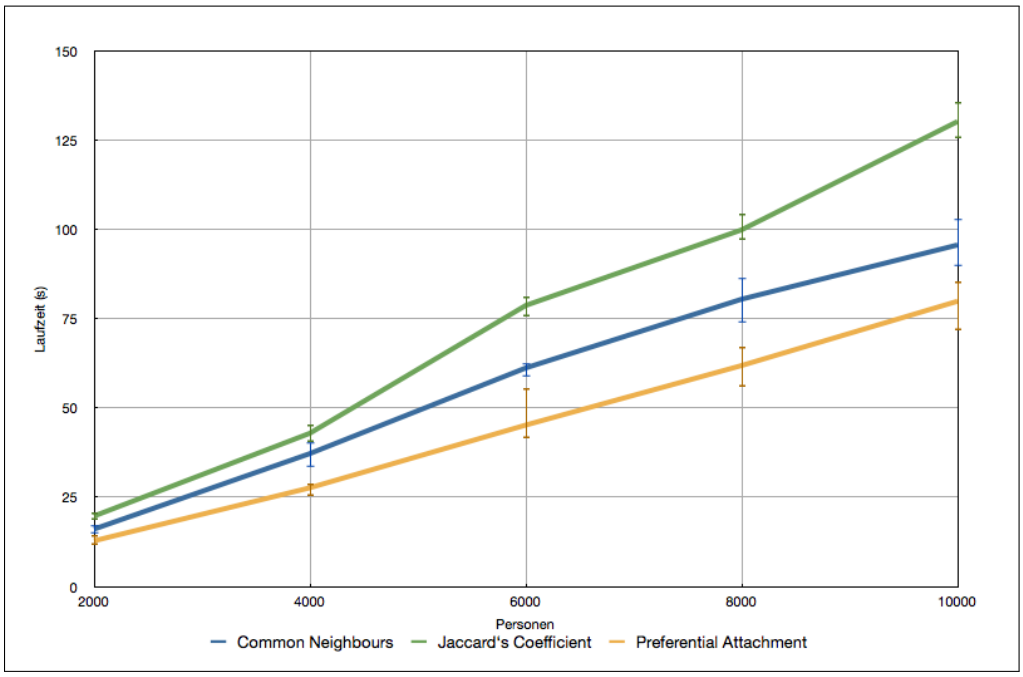


Abbildung 4.12.: Kollaboratives Filtern: Die Messwerte zeigen ein lineares Wachstum der *Common Neighbours*-, *Jaccard's Coefficient*- und *Preferential Attachment*-Metrik

sich ab, dass die *Adamic/Adar*-Metrik über ein exponentielles Wachstum verfügt und bereits bei geringer Anzahl von Personen sich erheblich von den anderen Metriken unterscheidet. An dieser Stelle wird deutlich, dass sich weitere Bemühungen mit der Entwicklung einer effizienteren Implementierung auseinander setzen sollten. Die Anwendung *Common Neighbours*-, *Jaccard's Coefficient*- und *Preferential Attachment*-Metrik weist hingegen ein lineares Wachstumsverhalten bei steigender Anzahl an Personen auf. Dies ist insbesondere Abbildung 4.12 zu entnehmen.

4.4. Fazit

Dieses Kapitel beschäftigte sich mit der Erweiterung des GraphX Frameworks. Dazu wurden zu Beginn zwei Algorithmen für eine Implementierung ausgewählt. Das Kriterium der Praxisrelevanz spielte dabei eine übergeordnete Rolle. Für eine Implementierung lagen beide Algorithmen in einer von den jeweiligen Autoren angefertigten Beschreibung vor. Dies sind die Verfahren des Semi-Clustering und des kollaborativen Filterns. Auf die Konzepte jedes Algorithmus wurde eingegangen. Somit wurde der Weg für eine eigenständige Umsetzung geebnet. Da die Algorithmen in einer sehr allgemeinen Form vorlagen, wurden Maßnahmen

getroffen, um die Problemstellungen auf das GraphX Framework anzupassen. Dazu zählen Maßnahmen zur Simulierung einer geforderten Graphstrukturierung, sowie die Umsetzung eines theoretischen Modells durch die Verwendung gegebener Schnittstellen.

Die Implementierungen wurden anschließend experimentell in Hinblick auf das Laufzeitverhalten untersucht. Hierzu wurde sowohl Spark im Cluster-Betrieb, als auch im lokalen Modus aufgesetzt. Im Fokus der Untersuchung standen zum einen die Skalierbarkeit der Algorithmen im Clusterbetrieb durch die Hinzunahme weiterer Worker-Instanzen. Zum anderen wurde das Wachstumsverhalten der Laufzeit auf einer variierenden und wachsenden Graphstruktur gemessen. Dies bezieht sich insbesondere auf die Knoten und Kanten des für den zu untersuchenden Graphen.

Bei der Umsetzung des Semi-Clustering und des kollaborativen Filterns hat sich gezeigt, dass die vom Framework bereitgestellten Operationen mächtig genug sind diese Algorithmen umzusetzen. Die Auswahl von lediglich einem iterativen Verarbeitungsmodell hat keine Einschränkung dargestellt. Pregel hat sich als Grundlage für das Semi-Clustering als geeignet identifiziert. Es hat sich jedoch auch gezeigt, dass sich in dem nachrichtenorientierten Modell von Pregel sehr viele redundante Nachrichten ergeben haben. Oftmals werden wiederholt Nachrichten versandt, die zu keiner Änderung von Zuständen führten, da diese bereits in vorherigen Iterationen erfolgreich zugestellt wurden. Auch wenn sich das Problem nicht in Pregel formulieren ließ, konnte man dennoch mit den sonstigen mächtigen Operationen von GraphX eine bequeme Lösung finden.

Aufbauend auf den Messwerten der einzelnen Untersuchungen, lässt sich erkennen, dass im Vergleich das Semi-Clustering durch die Hinzunahme einer weiteren Worker-Instanzen eine höhere Laufzeiteinsparung erreicht. Dies lässt eine bessere Parallelisierung des Algorithmus vermuten. Eine höherer Parallelität ist jedoch auch mit zunehmender Last verbunden. Das lineare Wachstumsverhalten deutet für die hier betrachteten Umstände auf ein durchaus in der Praxis anwendbares Lösungskonzept hin. Es fällt jedoch auf, dass dies nicht unmittelbar für die Anwendung der *Adamic/Adar*-Metrik als Gewichtung für das kollaborative Filtern gilt. An dieser Stelle sollten sich weitere Forschungsbestrebungen mit der Entwicklung einer alternativen Implementierung auseinandersetzen.

5. Zusammenfassung und Ausblick

Diese Arbeit hat sich mit der Implementierung von Algorithmen in dem GraphX Framework beschäftigt. Dazu wurden Konzepte für eine Realisierung erarbeitet und umgesetzt. Es wurden dabei die Möglichkeiten von GraphX aufgezeigt.

5.1. Zusammenfassung

Die Arbeit beginnt mit der Frage, wie sich das GraphX Framework für eine Algorithmen-Implementierung eignet. Um dem Ziel eigener Implementierungen näher zu kommen, werden zu Beginn die dafür notwendigen Grundlagen geschaffen. Hierzu werden auf relevante Begrifflichkeiten der Graphentheorie und auch Spark als Framework, auf dessen Schnittstellen GraphX aufbaut, eingeführt. Das *Resilient Distributed Dataset* (RDD) als verteilte und listenartige Datenstruktur nimmt als Hauptabstraktion von Spark eine wesentliche Rolle ein.

Im nächsten Schritt wird das GraphX Framework in der Tiefe so dargestellt, dass es für eine auf sich aufbauende Implementierung effektiv genutzt werden kann. GraphX ist ein Framework für parallele Verarbeitungen auf Graphstrukturen. Mit GraphX ist ein Paradigmenwechsel verbunden, da nun neben einer listenartigen auch eine graphorientierte Verarbeitung ermöglicht wird. Eine neue Abstraktionsebene, den *Resilient Distributed Graph* (RDG) als Repräsentant von Graphstrukturen, wird von GraphX geliefert. *RDG's* werden aus mehreren *RDD* konstruiert. Dessen theoretisches Modell wird als Property-Graph formalisiert. Spark sieht eine verteilte Verarbeitung von listenartigen Datensätzen vor. Mit der Verteilung von Graphen, als stark vernetzte Struktur sind neue Herausforderungen der Indizierung und Strukturierung verbunden. Die daraus folgende verteilte Darstellung wird thematisiert. Es folgt ein Überblick über die von GraphX bereitgestellten Operationen. Die durch die Operationen ermöglichte graphorientierte Verarbeitung bildet das Fundament einer eigenen Implementierung. Eine besondere Bedeutung wird *Pregel* zugeteilt, da es sich bei diesem um ein direkt nutzbares iteratives Verarbeitungsmodell handelt.

Im Fokus der Implementierung stehen zwei Algorithmen. Einerseits wird das Verfahren des Semi-Clustering implementiert. Es handelt sich um ein Clusterverfahren für soziale Netzwerke. Ziel des Verfahrens ist es, soziale Gruppierungen zu identifizieren, welche untereinander

in einem intensiven Kontakt zueinander stehen. Andererseits wird auch das Verfahren des kollaborativen Filterns betrachtet. Dieses findet Anwendung bei dem Prognostizieren von Nutzerinteressen auf Basis des vergangenen Kaufverhaltens. Eine Implementierung ist mit dem Ziel verbunden aufzuzeigen, wie sich die von GraphX bereitgestellten Operationen für die jeweilige Problemstellung eigenen. Für jeden dieser Algorithmen werden zu Beginn deren Konzepte beschrieben. Anschließend wird sich der Umsetzung in GraphX angenommen. Diese werden im Rahmen der Ausarbeitung auch implementiert. Das Laufzeitverhalten der Umsetzung wird daraufhin experimentell untersucht.

Die von dem GraphX Framework gelieferten Operationen haben sich als mächtig genug erwiesen sowohl das Semi-Clustering, als auch das kollaborative Filtern umzusetzen. Es wird die Möglichkeit geboten verschiedene Algorithmen implementieren zu können. Die Implementierungen skalieren nahezu linear und sind somit auch für große Datenmengen geeignet. Demnach kann das Ziel der Arbeit als erreicht angesehen werden.

5.2. Ausblick

Dem GraphX Framework wird in aktueller Literatur zum Thema Spark noch wenig Aufmerksamkeit zugeteilt. Trotz dessen ist zu erwarten, dass die wachsende Bedeutsamkeit von Spark sich auch positiv auf das GraphX Framework auswirken wird. Werden auch weiterhin die Datenmengen wachsen, wird auch die Nachfrage an einer Möglichkeit zur verteilten Verarbeitung stetig zunehmen. Dies gilt auch insbesondere für die Verarbeitung von Graphstrukturen. Auf der einen Seite liefert das Framework nur eine geringe Auswahl an Algorithmen. Auf der anderen Seite wird einem Entwickler jedoch die Möglichkeit geboten, eigene Algorithmen mit einer primär darauf ausgelegten Abstraktion zu entwickeln.

GraphX bietet die Möglichkeit eine Vielzahl von Algorithmen zu implementieren. Zukünftige Forschungen könnten sich weitergehend mit diesem Gebiet befassen, da es sich um ein relativ neues Forschungsfeld handelt. Weitere Arbeiten können sich außerdem mit der Umsetzung zusätzlicher Verarbeitungsmodelle in GraphX beschäftigen und diese mit dem bestehenden *Pregel* Modell vergleichen, da mit *Pregel* auch gewisse Einschränkungen verbunden sind. Zudem könnten nachfolgende Studien das genauere Laufzeitverhalten verschiedener Operationen untersuchen. Aus der Sicht des Anwenders spielen die zur Verfügung stehenden Operationen eine wesentliche Rolle. In beiden Fällen wäre ein Vergleich mit anderen Graphverarbeitungs-Frameworks denkbar, um mögliche Vorteile dieser zu identifizieren.

A. Inhalt der CD

- BA_Andersen.pdf
- SourceCode.zip

  Die CD beinhaltet eine digitale Version der Arbeit, den Programmcode und die Testdaten. Diese liegen als Scala Maven Projekt vor, welches die Scala Version 2.10 voraussetzt.

B. Messwerte

Sämtliche Messwerte sind in Sekunden angegeben.

B.1. Semi-Clustering

Durchlauf	1 W	2 W	3 W	4 W
1.	212,71	137,91	95,08	82,07
2.	207,73	124,87	89,48	80,01
3.	221,76	129,12	92,45	78,35
4.	215,55	133,01	89,77	80,28
5.	221,46	137,15	91,49	79,21
AVG	**215,84**	**132,41**	**91,65**	**79,98**

Abbildung B.1.: Messwerte zu Abbildung 4.5 - Skalierung durch Worker W

Durchlauf	1000 K	2000 K	3000 K	4000 K	5000 K
1.	24,62	68,51	120,42	150,83	200,43
2.	22,90	67,25	111,56	145,83	207,50
3.	18,74	66,01	107,97	137,85	201,53
4.	18,66	64,90	105,55	139,10	204,25
5.	17,79	64,56	103,61	136,53	193,08
AVG	**20,54**	**66,25**	**109,82**	**142,03**	**201,36**

Durchlauf	6000 K	7000 K	8000 K	9000 K
1.	256,50	310,31	343,63	393,97
2.	235,77	294,50	314,21	383,93
3.	226,92	290,41	349,10	354,25
4.	230,60	311,42	359,67	363,50
5.	218,42	288,19	341,71	366,45
AVG	**233,64**	**298,97**	**341,66**	**372,42**

Abbildung B.2.: Messwerte zu Abbildung 4.6 - Skalierung durch Knoten K

B.2. Kollaboratives Filtern

CN

Durchlauf	1 W	2 W	3 W	4 W
1.	133,92	104,34	90,23	74,83
2.	141,59	97,42	86,01	72,11
3.	142,45	105,67	80,56	64,67
4.	136,45	108,43	75,53	73,43
5.	137,29	95,99	90,45	69,25
AVG	**138,34**	**102,37**	**84,55**	**70,86**

JC

Durchlauf	1 W	2 W	3 W	4 W
1.	129,41	104,13	84,84	73,23
2.	139,88	121,56	92,34	83,45
3.	135,02	111,11	85,43	65,45
4.	141,95	109,34	95,67	72,45
5.	132,77	97,34	90,23	78,43
AVG	**135,81**	**108,70**	**89,70**	**74,60**

PA

Durchlauf	1 W	2 W	3 W	4 W
1.	134,23	84,99	67,26	55,67
2.	114,01	85,32	65,61	49,59
3.	129,18	79,97	69,93	54,71
4.	117,97	82,57	63,23	46,36
5.	130,31	89,43	61,97	52,30
AVG	**125,14**	**84,46**	**65,60**	**51,73**

Abbildung B.3.: Messwerte zu Abbildung 4.10 - Skalierung durch Worker W

CN

Durchlauf	100 P	200 P	300 P	400 P	500 P	600 P
1.	3,42	4,98	6,75	8,15	8,69	9,51
2.	1,96	3,18	4,16	4,56	5,01	5,96
3.	2,49	2,62	3,06	3,53	4,33	4,78
4.	1,66	2,37	2,75	3,58	4,71	4,70
5.	1,59	2,19	2,88	3,41	4,23	4,65
AVG	**2,22**	**3,07**	**3,92**	**4,65**	**5,40**	**5,91**

JC

Durchlauf	100 P	200 P	300 P	400 P	500 P	600 P
1.	4,13	6,14	8,42	9,22	9,97	12,03
2.	2,62	4,02	5,10	5,55	7,73	8,02
3.	2,09	2,95	4,01	5,08	6,63	7,08
4.	2,04	2,77	3,94	4,47	5,85	6,49
5.	2,23	2,54	3,97	4,98	5,66	5,59
AVG	**2,62**	**3,68**	**5,09**	**5,86**	**7,17**	**7,84**

PA

Durchlauf	100 P	200 P	300 P	400 P	500 P	600 P
1.	3,06	4,11	5,24	5,93	7,45	9,27
2.	2,01	2,83	3,56	3,94	4,18	4,37
3.	2,74	2,28	3,02	3,22	3,38	3,95
4.	1,68	1,99	2,31	2,66	3,43	4,48
5.	1,35	2,08	2,32	2,85	3,22	4,37
AVG	**2,17**	**2,66**	**3,28**	**3,72**	**4,33**	**5,29**

AA

Durchlauf	100 P	200 P	300 P	400 P	500 P	600 P
1.	27,22	90,49	205,23	346,05	549,11	1289,09
2.	20,59	78,79	178,68	316,64	536,62	1197,55
3.	20,99	76,72	177,73	313,69	534,05	1161,79
4.	23,96	77,97	176,12	318,21	543,39	997,45
5.	24,61	76,42	173,50	319,09	575,67	1045,67
AVG	**23,47**	**80,08**	**182,25**	**322,74**	**547,77**	**1138,31**

Abbildung B.4.: Messwerte zu Abbildung 4.11 - Skalierung durch Personenknoten P

CN

Durchlauf	2000 P	4000 P	6000 P	8000 P	10000 P
1.	17,31	40,46	61,36	86,59	103,15
2.	16,78	38,93	60,62	83,20	95,61
3.	16,09	37,01	62,36	83,81	99,33
4.	15,54	36,43	62,72	74,73	90,18
5.	14,62	33,25	58,84	73,79	89,70
AVG	**16,07**	**37,22**	**61,18**	**80,42**	**95,59**

JC

Durchlauf	2000 P	4000 P	6000 P	8000 P	10000 P
1.	21,02	45,54	81,39	104,74	135,72
2.	18,99	43,44	76,41	100,49	130,56
3.	20,56	43,88	81,08	99,02	131,25
4.	18,59	40,45	75,45	97,03	125,29
5.	19,45	41,22	79,01	97,88	127,86
AVG	**19,72**	**42,91**	**78,67**	**99,83**	**130,14**

PA

Durchlauf	2000 P	4000 P	6000 P	8000 P	10000 P
1.	14,42	29,19	55,69	67,34	85,13
2.	14,14	27,91	44,49	66,17	80,26
3.	11,52	28,96	42,60	58,62	85,32
4.	11,55	25,42	41,38	55,76	71,60
5.	12,33	26,74	41,79	61,38	76,90
AVG	**12,79**	**27,64**	**45,19**	**61,85**	**79,84**

Abbildung B.5.: Messwerte zu Abbildung 4.12 - Skalierung durch Personenknoten P

Literaturverzeichnis

[Diestel 2010] Diestel, Reinhard: *Graphentheorie*. Bd. 4. Heildelberg, NY, USA : Springer Berlin, 2010

[Gonzalez u. a. 2012] Gonzalez, Joseph E. ; Low, Yucheng ; Gu, Haijie ; Bickson, Danny ; Guestrin, Carlos: PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA : USENIX Association, 2012 (OSDI'12), S. 17–30

[Gonzalez u. a. 2014] Gonzalez, Joseph E. ; Xin, Reynold S. ; Dave, Ankur ; Crankshaw, Daniel ; Franklin, Michael J. ; Stoica, Ion: GraphX: Graph Processing in a Distributed Dataflow Framework. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA : USENIX Association, 2014 (OSDI'14), S. 599–613

[Huang u. a. 2005] Huang, Zan ; Li, Xin ; Chen, Hsinchun: Link Prediction Approach to Collaborative Filtering. In: *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries*. New York, NY, USA : ACM, 2005 (JCDL'05), S. 141–142

[Karau u. a. 2015] Karau, Holden ; Konwinski, Andy ; Wendell, Patrick ; Zaharia, Matei: *Learning Spark: Lightning-Fast Big Data Analysis*. Sebastopol, CA, USA : O'Reilly Media, Inc., 2015

[Liben-Nowell und Kleinberg 2003] Liben-Nowell, David ; Kleinberg, Jon: The Link Prediction Problem for Social Networks. In: *Proceedings of the Twelfth International Conference on Information and Knowledge Management*. New York, NY, USA : ACM, 2003 (CIKM'03), S. 556–559

[Lim u. a. 2015] Lim, Seung-Hwan ; Lee, Sangkeun ; Ganesh, G. ; Brown, Tyler C. ; Sukumar, Sreenivas R.: Graph Processing Platforms at Scale: Practices and Experiences. In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software*. Philadelphia, PA, USA : IEEE Computer Society, 2015 (ISPASS), S. 42–51

[Lu u. a. 2014] Lu, Xiaoyi ; Rahman, Md. Wasi U. ; Islam, Nusrat ; Shankar, Dipti ; Panda, Dhabaleswar K.: Accelerating Spark with RDMA for Big Data Processing: Early Experiences. In: *Proceedings of 2014 IEEE 22nd Annual Symposium on High-Performance Interconnects*, IEEE Computer Society, 2014 (HOTI), S. 9–16

[Malewicz u. a. 2010] Malewicz, Grzegorz ; Austern, Matthew H. ; Bik, Aart J. ; Dehnert, James C. ; Horn, Ilan ; Leiser, Naty ; Czajkowski, Grzegorz: Pregel: A System for Large-scale Graph Processing. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2010 (SIGMOD'10), S. 135–146

[Nisar u. a. 2013] Nisar, M. U. ; Fard, Arash ; Miller, John A.: Techniques for Graph Analytics on Big Data. In: *Proceedings of the 2013 IEEE International Congress on Big Data*. Athens, GA, USA : IEEE Computer Society, 2013 (BIGDATACONGRESS'13), S. 255–262

[Shang und Yu 2014] Shang, Zechao ; Yu, Jeffrey X.: Auto-approximation of Graph Computing. In: *Proc. VLDB Endow.* 7 (2014), Nr. 14, S. 1833–1844

[Valiant 1990] Valiant, Leslie G.: A Bridging Model for Parallel Computation. In: *Commun. ACM* 33 (1990), Nr. 8, S. 103–111

[Xin u. a. 2014] Xin, Reynold S. ; Crankshaw, Daniel ; Dave, Ankur ; Gonzalez, Joseph E. ; Franklin, Michael J. ; Stoica, Ion: GraphX: Unifying Data-Parallel and Graph-Parallel Analytics. In: *ArXiv e-prints* (2014)

[Xin u. a. 2013] Xin, Reynold S. ; Gonzalez, Joseph E. ; Franklin, Michael J. ; Stoica, Ion: GraphX: A Resilient Distributed Graph System on Spark. In: *First International Workshop on Graph Data Management Experiences and Systems*. New York, NY, USA : ACM, 2013 (GRADES'13), S. 2:1–2:6

[Zaharia u. a. 2012] Zaharia, Matei ; Chowdhury, Mosharaf ; Das, Tathagata ; Dave, Ankur ; Ma, Justin ; McCauley, Murphy ; Franklin, Michael J. ; Shenker, Scott ; Stoica, Ion: 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*. Berkeley, CA, USA : USENIX Association, 2012 (NSDI'12), S. 2–2

[Zaharia u. a. 2010] Zaharia, Matei ; Chowdhury, Mosharaf ; Franklin, Michael J. ; Shenker, Scott ; Stoica, Ion: Spark: Cluster Computing with Working Sets. In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. Berkeley, CA, USA : USENIX Association, 2010 (HotCloud'10), S. 10–10

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 18.12.2015 Jakob Smedegaard Andersen