Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Marc Kaepke

Graphen im Big Data Umfeld - Experimenteller Vergleich von Apache Flink und Apache Spark

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Marc Kaepke

Graphen im Big Data Umfeld - Experimenteller Vergleich von Apache Flink und Apache Spark

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

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

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

Eingereicht am: 11. September 2017

Marc Kaepke

Thema der Arbeit
Graphen im Big Data Umfeld - Experimenteller Vergleich von Apache Flink und Apache Spark

Stichworte
Apache Flink, Apache Spark, Graphen, verteilte Graphverarbeitung, Gelly, GraphX, Big Data, Experimente, Semi-Clustering, PageRank, Implementierung

Kurzzusammenfassung
Graphen eignen sich ideal, um Relationen zwischen Objekten abzubilden. Allerdings benötigen die Big Graphs spezielle Systeme, um diese zu verarbeiten und zu analysieren. Die Open Source Frameworks Apache Flink und Apache Spark stellen Bibliotheken für verteilte Graphverarbeitungen bereit. Im Rahmen dieser Thesis soll verglichen werden, welches Framework sich besser für diesen Zweck eignet. Mehrere Experimente zeigen, dass Gelly (Flink) leichte Vorteile gegenüber Graph (Spark) bietet und dass die Systeme unterschiedlich skalieren.
Marc Kaepke

Title of the paper
Graphs in Big Data - experimental comparison of Apache Flink and Apache Spark

Keywords
Apache Flink, Apache Spark, graphs, distributed graph processing, Gelly, GraphX, Big Data, experiments, Semi-Clustering, PageRank, implementation

Abstract
Graphs are ideal for mapping relations between objects. However, the Big Graphs require special systems to process and analyze them. The open source frameworks Apache Flink and Apache Spark provide libraries for distributed graph processing. Within the framework of this bachelor's thesis it is intended to compare which framework is more convenient for this purpose. Several experiments show that Gelly (Apache Flink) offers slight advantages over GraphX (Apache Spark) and that the systems scale differently.

Inhaltsverzeichnis

1 Einleitung **1**

2 Apache Spark **3**
2.1 Ökosystem . . . 3
2.1.1 Bibliotheken und APIs . . . 3
2.1.2 Cluster-Manager und Datenquellen . . . 5
2.2 Resilient Distributed Dataset . . . 5
2.3 GraphX . . . 6
2.3.1 Graph-Partitionierung . . . 7
2.3.2 Pregel API . . . 8

3 Apache Flink **10**
3.1 Ökosystem . . . 10
3.2 Streaming Dataflow Modell . . . 13
3.2.1 Dataflow Graph . . . 14
3.2.2 Spezialfall: DataSet API . . . 15
3.3 Batch Iterationen . . . 16

4 Gelly **18**
4.1 Repräsentation von Graphen . . . 18
4.2 Modifikation, Transformation und Nachbarschaft . . . 19
4.3 Verteilte Graphverarbeitung . . . 20
4.3.1 vertex-centric . . . 21
4.3.2 scatter-gather . . . 22
4.3.3 gather-sum-apply . . . 24
4.3.4 Gelly spezifische Modell-Konfiguration . . . 25
4.3.5 Gegenüberstellung der Modelle . . . 27
4.4 Bibliothek und Algorithmen . . . 28

5 Erweiterung der Gelly Bibliothek **30**
5.1 Allgemeines zum Semi-Clustering . . . 30
5.2 Konzept . . . 31
5.3 Umsetzung in Gelly . . . 33
5.3.1 Vorverarbeitung . . . 33
5.3.2 Architektur . . . 34

6 Experimente **36**
6.1 Aufbau der Experimente . . . 36
6.1.1 Hypothesen . . . 36
6.1.2 Cluster . . . 37
6.1.3 Daten . . . 38
6.1.4 Algorithmen . . . 39
6.2 Experimentelle Durchführung . . . 41
6.2.1 Hypothese 1 . . . 42
6.2.2 Hypothese 2 . . . 43
6.2.3 Hypothese 3 . . . 46
6.2.4 Hypothese 4 . . . 49
6.2.5 Hypothese 5 . . . 50
6.3 Korrektheit der Experimente . . . 52
6.4 Auswertung der Experimente . . . 52

7 Zusammenfassung und Ausblick **55**
7.1 Flink vs. Spark - Gegenüberstellung . . . 55
7.1.1 Iteration . . . 56
7.1.2 Speichermanagement . . . 57
7.1.3 Graph-Partitionierung . . . 57
7.1.4 Graph Programmiermodelle . . . 57
7.2 Fazit und Ausblick . . . 57

Literaturverzeichnis **59**

Abbildungsverzeichnis

2.1 Apache Spark Komponentenübersicht 4
2.2 edge-cut vs. vertex-cut 7
2.3 GraphX - tabellarische Repräsentation vom vertex-cut 8
2.4 BSP-Ausführung mit synchronisiertem Superstep 9

3.1 Apache Flink Komponetenübersicht 11
3.2 Apache Flink Architektur 12
3.3 Streaming Dataflow Programm unter Flink 14
3.4 Schemenhafte Darstellung eines Directed Acyclic Graph 15
3.5 Iterations-Operatoren in Apache Flink 17

4.1 Beispiel einer vertex-centric Iteration 22
4.2 Beispiel einer scatter-gather Iteration 23
4.3 Beispiel einer gather-sum-apply Iteration 24

5.1 Beispielergebnis vom Semi-Clustering 31
5.2 Vorbereitungsschritte beim Semi-Clustering 34
5.3 Ausschnitt vom Semi-Clustering Klassendiagramm 35

6.1 Hypothese 1 - Ergebnisvisualisierung 42
6.2 Hypothese 2 - Ergebnisvisualisierung von PageRank (10x Graphskalierung) . . 45
6.3 Hypothese 2 - Ergebnisvisualisierung von PageRank (2x Graphskalierung) . . 46
6.4 Hypothese 3 - Ergebnisvisualisierung vom PageRank 47
6.5 Hypothese 3 - Ergebnisvisualisierung vom Semi-Clustering 48
6.6 Hypothese 4 - Visualisierung des Speicherverbrauchs 50
6.7 Hypothese 5 - Ergebnisvisualisierung 51

7.1 Flink und Spark im Kontext 55
7.2 Schema der Iterationen unter Apache Flink und Apache Spark 56

Listings

4.1 Gelly - Knoten erzeugen 18
4.2 Gelly - Kante erzeugen 19
4.3 Gelly - Ausschnitt einer Graphanalyse 28

Tabellenverzeichnis

4.1 GraphX und Gelly im Überblick 21
4.2 Gelly Konfigurationsparameter der Graph Programmiermodelle 25
4.3 Vergleich von vertex-centric, scatter-gather und GSA Modell 27

6.1 Angepasste Konfigurationsparameter für Apache Flink und Apache Spark . . 41
6.2 Hypothese 1 - Laufzeiten 43
6.3 Hypothese 2 - Laufzeiten 44
6.4 Laufzeitverbesserung bei unterschiedlichen Datentypen 53

1 Einleitung

Graphen, auch Netzwerke genannt, gewinnen fortlaufend an Wichtigkeit im Big Data Umfeld und sind allgegenwärtig. Jedes soziale Netzwerk, wie Facebook oder Instagram, weist umfangreiche Graphstrukturen auf. Im e-Commerce Bereich lassen sich Kaufempfehlungen auf Basis von Graphen berechnen. Ebenso sind Graphen in der Offline-Welt beim Analysieren oder Koordinieren von Energie- und Transportnetzwerken vorzufinden. Aber auch in der Medizin zur Erforschung von Krebstumoren wird auf Graphstrukturen zurückgegriffen, um Teilnetze mit einer erheblichen Anzahl an mutierten Patienten aufzuspüren [vgl. Ste17].

Graphen repräsentieren dabei Informationen aus einer spezifischen Domäne. Die Knoten spiegeln beliebige Objekte, wie Personen oder Flughäfen, wider und die Kanten symbolisieren Relationen zwischen Objekten, wie Freundschaften oder Flugverbindungen. Durch Attribute an den Kanten lassen sich zusätzliche Angaben zur Relation festhalten, wie beispielsweise der Beginn einer Freundschaft oder die Dauer der Flugreise.

Um große Graphen möglichst effizient verarbeiten und anschließend analysieren zu können, sind Cluster-Systeme notwendig. Apache Flink und Apache Spark sind moderne Open-Source Frameworks, die konzipiert wurden, um große und verteilte Datenmengen zu verarbeiten. Die Schwerpunkte beider Frameworks liegen im klassischen Batch-Processing und dem neueren Stream-Processing. Zusätzlich beinhalten beide Systeme spezielle Bibliotheken und APIs zur Verarbeitung und Analyse von Graphen.

Ziel dieser Thesis ist, festzustellen, inwieweit sich Apache Flink besser oder schlechter zur Graphverarbeitung eignet als Apache Spark. Die Bewertungsgrundlage hierfür bilden Experimente, in denen Graph-Algorithmen mit unterschiedlichen Daten ausgeführt werden. In den Experimenten wird die Laufzeit der Algorithmen ermittelt, die diese benötigen, um unterschiedliche Graphausdehnungen zu verarbeiten. Zusätzlich werden die Laufzeiten bei einer horizontalen Skalierung des Clusters gemessen.

In den beiden nachfolgenden Kapiteln werden zunächst die Frameworks Apache Spark (2) und Apache Flink (3) vorgestellt und auf einige Besonderheiten eingegangen. Das zweite Kapitel behandelt zusätzlich die Graph Bibliothek von Apache Spark.

Im 4. Kapitel liegt der Fokus auf Gelly und dessen Operatoren zur Graphverarbeitung. Ein Abschnitt befasst sich unter anderem mit den Funktionsweisen der unterschiedlichen Programmiermodelle für Graphen.

An einem praktischen Beispiel wird die Gelly Bibliothek im 5. Kapitel um einen Algorithmus erweitert. Der implementierte Semi-Clustering Algorithmus dient der Vergleichbarkeit beider Frameworks. Das Kapitel beschreibt das Konzept des Algorithmus und wie dieser mit Hilfe von Apache Flink umgesetzt wird.

Im 6. Kapitel finden die Experimente statt. Hierfür werden zunächst der Aufbau und die Durchführung beschrieben, um anschließend ein Resümee der Ergebnisse zu ziehen. Die Experimente orientieren sich dabei an aufgestellten Hypothesen, die auf Grundlage der vorherigen Theoriebasis definiert sind.

Das letzte Kapitel (7) bietet einen kompakten Überblick der beiden Frameworks bezüglich einiger Aspekte zur Graphverarbeitung. Auf Basis der Ergebnisse aus den Experimenten erfolgt zum Schluss ein Fazit, außerdem wird es einen Ausblick für weiterführende Experimente in diesem Bereich geben.

2 Apache Spark

Apache Spark [vgl. ASF17f] ist ein Open-Source Framework für die parallele Datenverarbeitung und wurde 2009 an der University of California, Berkeley entwickelt.

Spark bietet Entwicklern eine leistungsfähigere und flexiblere Alternative zu Hadoop's MapReduce. Als Framework verbindet es eine In-Memory Verarbeitung mit einer abgestimmten Engine für verteilte Berechnungen. Daten und Zwischenergebnisse einzelner Operationen können dadurch temporär im Arbeitsspeicher gehalten werden. Speziell bei iterativen Algorithmen bedeutet dies einen enormen Geschwindigkeitsunterschied und ermöglicht eine bis zu 100x schnellere Verarbeitung verglichen mit Hadoop [vgl. Sal u. a.16]. Spark verwendet die Daten-Abstraktion *Resilient Distributed Dataset* (Abschnitt 2.2) und reduziert somit zusätzlich die Abhängigkeit zum unterliegenden Speichersystem. Spark besitzt kein eigenes Speichersystem. Es ist in der Lage, auf ein bestehendes *Hadoop Distributed File System* (kurz HDFS) und anderen Speichersystemen adaptiert zu werden. Neben der klassischen Batch-Verarbeitung stellt das Framework weitere Bibliotheken bereit. Diese werden zusammen mit den Speichersystemen im Abschnitt 2.1 erläutert und beinhalten beispielsweise den *Spark Core* oder die *MLlib* für maschinelles Lernen. Die Graph-Library *GraphX* wird in Abschnitt 2.3 behandelt.

2.1 Ökosystem

Das Spark Ökosystem setzt sich aus mehreren, teils unabhängigen, Komponenten (Abb. 2.1) zusammen und bietet unterschiedliche Werkzeuge für die Verarbeitung und Analyse an. Die beiden nachfolgenden Abschnitte behandeln die Bibliotheken in Spark (2.1.1) und die Möglichkeiten zur Anbindung von externen Datenquellen (2.1.2).

2.1.1 Bibliotheken und APIs

Der *Spark Core* ist die zugrundeliegende Engine und stellt elementare Funktionalitäten wie Scheduling, Verteilung der Daten, Fehlertoleranz und Monitoring bereit [vgl. And15]. Ein wesentlicher Vorteil von Spark, gegenüber Hadoop, ist die Vielfalt von verfügbaren Libraries,

die auf dem Core aufsetzen und leistungsfähige APIs in Scala, Java, Python und R bereitstellen [vgl. Sal u. a.16].

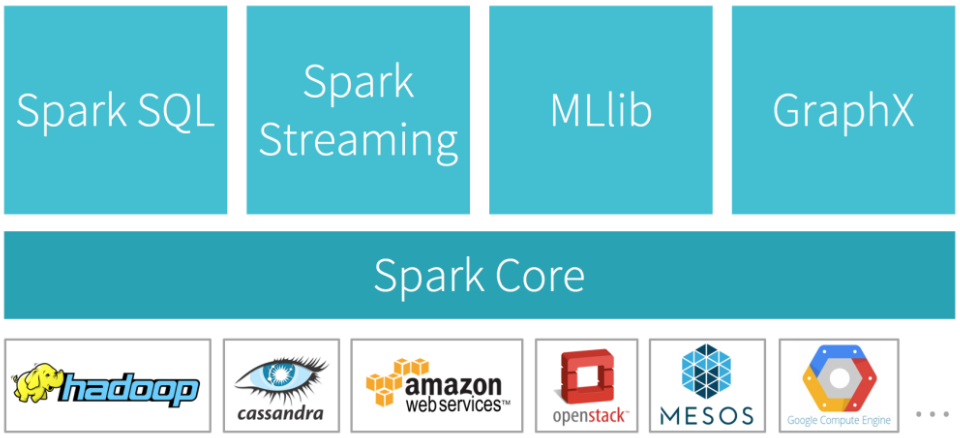


Abbildung 2.1: Apache Spark Komponentenübersicht
(Quelle: https:// databricks.com)

- **Spark SQL**

  Spark SQL ist das Modul für (semi-) strukturierte Datenverarbeitungen. Es bietet eine Abstraktion namens *DataFrame* und funktioniert zudem als Engine für verteilte SQL-Abfragen. Ein DataFrame ist konzeptionell eine Tabelle einer relationalen Datenbank und kann aus einer Vielzahl von Datenquellen (strukturierte Dateien, Tabellen in Hive oder existierenden RDDs) erzeugt werden.

- **Spark Streaming**

  Mittels *Spark Streaming* können leichtgewichtige Streaming-Analyse-Programme entwickelt werden. Unterscheiden tut sich diese Komponente durch eine on-the-fly Datenstromverarbeitung, welche mit der konventionellen Batch-Verarbeitung kombinierbar ist. Ein Anwendungsbeispiel ist die Echtzeitanalyse von Twitter Tweets mit persistenten Daten aus der Datenbank. Intern zerlegt Spark die eingehenden Datenströme in kleine Micro-Batches, so genannte DStreams. Diese bestehen wiederum aus RDDs und bringen eine integrierte Fehlertoleranz mit.

- **MLlib**

  MLlib ist die Spark-Erweiterung für verteiltes maschinelles Lernen (ML), um beispielsweise Prognosen (predictive analytics) zu berechnen. Des Weiteren lassen sich Texte clustern oder kategorisieren. In so genannten Pipelines werden unterschiedliche Verarbeitungsschritte und ML-Algorithmen kombiniert, um große Datenmengen zu analysieren. Hierbei kommt die In-Memory Technologie und simultane Verarbeitung von Spark zum Einsatz. Die Library stellt für das Pipelining eine Reihe von ML-Algorithmen zur Verfügung, darunter K-means und naives Bayes.

- **GraphX**

  GraphX ist eine Erweiterung für Spark, die verteilte und flexible Graphen-Programme in einem Spark-Cluster ermöglicht. Im Abschnitt 2.3 wird diese Library detaillierter betrachtet.

2.1.2 Cluster-Manager und Datenquellen

Im Cluster-Modus verwendet Spark eine Master-Slave Architektur. Ein Master-Knoten verteilt dabei die Aufgaben an endlich viele Slave-Knoten (auch Worker genannt) und kann dadurch flexibel skaliert werden [vgl. And15].
Spark unterstützt folgende vier Cluster-Manager:

- *Apache YARN* (dem Ressourcen Manager aus Hadoop 2) [vgl. ASF16a]
- *Apache Mesos* (ein allgemeiner Cluster-Manager) [vgl. ASF17e]
- *Amazon EC2* (Serververbund von Amazon) [vgl. Ama17]
- Standalone Modus

Das Framework stellt eine Vielzahl von Datenquellen-APIs zur Verfügung. Es können unter anderem HDFS, Cassandra oder HBase Systeme integriert werden [vgl. Sal u. a.16].

2.2 Resilient Distributed Dataset

Ein Resilient Distributed Dataset (kurz RDD) ist eine Datenabstraktion im Spark Core und ermöglicht eine effiziente Wiederverwendung von Daten. RDDs sind fehlertolerante, parallele Datenstrukturen und bieten Entwicklern eine gezielte Datenhaltung im Arbeitsspeicher [vgl. Zah u. a.12].

Ein RDD Objekt ist *immutable* und kann aus Daten einer externen Quelle oder durch Transformationen bestehender RDDs erzeugt werden. Eine Transformation (bspw. *map*, *filter* und *join*) ist eine Operation auf einem RDD. Aufgrund der immutable Eigenschaft resultiert dadurch ein neues RDD [vgl. And15].

Als fehlertolerante Abstraktion werden Datenreplikationen vermieden. Jede Transformation wird dabei in einem Log protokolliert. Mittels des Logs lässt sich feststellen, durch welche Operationen und Reihenfolge ein spezifisches RDD entstanden ist [vgl. Sal u. a.16]. Im Falle eines Datenverlusts oder Fehlers besitzen RDDs somit ausreichend Informationen, um die fehlenden Partitionen mit Hilfe anderer RDDs zu rekonstruieren. Diese Wiederherstellungsstrategie leistet eine hohe Toleranz im Cluster und kommt ohne kostspieliger Replikationen aus [vgl. Zah u. a.12].

Entwickler sind in der Lage, zwei zusätzliche Aspekte eines RDDs zu beeinflussen. Es lässt sich definieren, welche Datenhaltungsvariante verwendet wird. So können RDDs bei einem iterativen Algorithmus im Cache gehalten werden und vermeiden eine redundante Neuberechnungen [vgl. And15]. Der zweite Aspekt ist die manuelle Festlegung, wie RDDs partitioniert werden. Die Aufteilung basiert auf einem Schlüssel und ist in jedem RDD vorhanden.

Sofern ausreichend Arbeitsspeicher vorhanden ist, werden RDDs nicht materialisiert. Zudem sind Transformationen *lazy*. Spark führt eine Transformation nur durch, wenn am RDD-Objekt eine *Action* aufgerufen wird [vgl. Zah u. a.12].

Die RDD Collection stellt in Spark's Programmiermodell die Basisabstraktion dar und ist Grundlage für die Batch-Verarbeitung. Datenströme werden über eine eigene DStream Collection realisiert, welche intern jedoch mit RDDs umgesetzt sind.

2.3 GraphX

GraphX ist eine Komponente von Spark und realisiert die parallele Verarbeitung von Graphstrukturen. Unter Erweiterung des RDD-Konzepts wird eine zusätzliche Abstraktion verwendet: Ein gerichteter Multigraph mit Attributen (Property) an den Knoten und Kanten. Das erweiterte RDD Konzept wird *Resilient Distributed Graph* (kurz RDG) genannt und intern durch ein RDD-Paar dargestellt. Die Knoten bzw. Kanten eines Graphen werden durch einzelne RDD abgebildet [vgl. And15]. In einer GraphX Anwendung spezifizieren Transformationen den Übergang von einem Graphen in einen Neuen. Durch diese Operationen verändern sich Knoten oder auch Kanten. Da Graphen rekursive Strukturen sind, haben Änderungen meistens zudem eine Auswirkung auf benachbarte Knoten und Kanten [vgl. Xin u. a.13].

2.3.1 Graph-Partitionierung

Beim Daten-Parallelismus erfolgen die Berechnungen parallel und unabhängig voneinander. In einem Graph-parallelem System, wie GraphX, stehen Änderungen im direkten Zusammenhang zu benachbarten Knoten bzw. Kanten [vgl. Xin u. a.13]. Jede Transformation hängt von den Ergebnissen vieler verteilten *joins* ab. Hierfür gibt es unterschiedliche Partitionierungsansätze wie *(random) edge-cut* oder analog dazu *(random) vertex-cut* [vgl. Xin u. a.13].

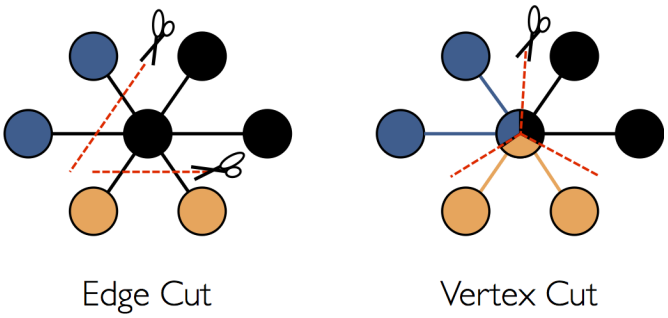


Abbildung 2.2: edge-cut vs. vertex-cut
(Quelle: https:// spark.apache.org)

GraphX verwendet zum Partitionieren den vertex-cut (dt. *Knoten-Schnitt*), da dieser im Gegensatz zum edge-cut kürzere Algorithmen-Laufzeiten erzielt [vgl. Xin u. a.13]. Um einen großen Graphen in mehrere Komponente zu zerlegen, werden bei diesem Verfahren sog. *Gelenkknoten* identifiziert [vgl. And15]. Durch das Entfernen eines Gelenkknotens ist ein Graph anschießend nicht mehr zusammenhängend. Die einzelnen Komponenten lassen sich auf unterschiedlichen Spark-Workern verteilen und mit geringem Overhead verarbeiten. In Abbildung 2.2 stellt beispielsweise der mittlere Knoten einen solchen Gelenkknoten dar.

Die Abbildung 2.3 zeigt die interne Repräsentationsstruktur eines Graphen durch GraphX. Unter anderem lassen sich hier die drei Komponenten (engl. *partition*) ausmachen. GraphX repräsentiert diesen durch drei horizontal partitionierte Tabellen, die folgende Eigenschaften besitzen:

- **EdgeTable (*pid, src, dst, data*)**

  In dieser Tabelle befinden sich die Kantendaten und dessen Adjazenzstruktur. Jede Kante wird als Tupel repräsentiert und besteht aus einer Quellknoten-ID *(src)* bzw. Zielknoten-ID *(dst)*, den individuellen Attributen *(data)* und einem virtuellen Partitionsidentifier *(pid)* [vgl. Xin u. a.13]. In dieser Tabelle werden lediglich Knoten-IDs gespeichert, die Daten selbst sind in der *Vertex Data Table* hinterlegt.

- **VertexDataTable (*id*, *data*)**

  Die VertexDataTable speichert für jeden Knoten eine ID und dessen Attribut ab. Eine *Bitmask* zeigt zusätzlich die Verfügbarkeit eines Knotens an, da markierte Knoten bei einer Transformation nicht berücksichtigt werden [vgl. And15].

- **VertexMap (*id*, *pid*)**

  Diese *Map* wird auch *Routing Table* genannt. Aus der Map ($id \rightarrow pid$) kann entnommen werden, in welchen Partitionen ein Knoten enthalten ist.

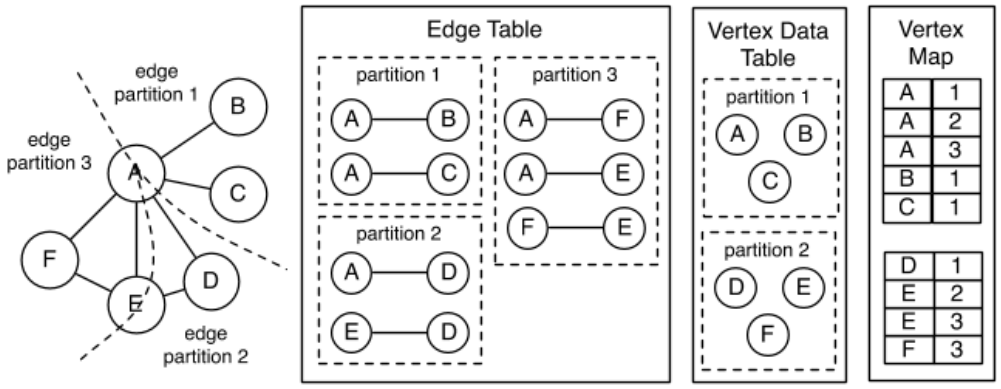


Abbildung 2.3: GraphX - tabellarische Repräsentation vom vertex-cut
(Quelle: Entnommen aus [Xin u. a.13])

2.3.2 Pregel API

Über GraphX stellt Spark unterschiedliche Graph-Transformationen sowie Filter- und Abfrage-Optionen bereit. Für die Implementierung von iterativen Graph-Algorithmen (bspw. *Semi-Clustering*) fehlt es jedoch an einer geeigneten Schnittstelle [vgl. Xin u. a.13]. Der *Pregel Operator* basiert auf dem Pregel Konzept und befindet sich oberhalb von GraphX. Durch diese Abstraktion wird GraphX um die fehlende Schnittstelle erweitert.

Das von Google entwickelte Pregel Konzept ist ein knotenorientiertes, iteratives Verarbeitungsmodell von Graphen und verfolgt den *„think like a vertex“* (kurz TLAV) Ansatz. Entlang der Kanten übermitteln die Knoten Nachrichten zu anderen Knoten. Pregel organisiert diesen Nachrichtenaustausch in einer Abfolge von Iterationen, den *Supersteps*. Supersteps sind nach dem *gather-apply-scatter* (kurz GAS) Paradigma aufgebaut [vgl. And15] und lassen sich infolge dessen in drei Schritte aufteilen: Alle eingehenden Nachrichten einlesen (*gather*),

interne Knotenstatus (bspw. Knotenwert) aktualisieren (*apply*) und Nachrichten an Nachbarknoten versenden (*scatter*). Die Ausführung der benutzerdefinierten Knotenfunktion erfolgt in synchronisierten *Supersteps* [vgl. Mal u. a.10]. Im Superstep S_n empfangen alle Knoten die Nachrichten aus dem vorherigen Superstep S_{n-1}. Wenn jeder Knoten die Knotenfunktion erfolgreich durchgeführt hat, werden die neuen Nachrichten synchron zum nachfolgenden Superstep S_{n+1} übertragen. Aufgrund dieser Synchronisationsbarriere (Abb. 2.4) ist sichergestellt, dass Knoten lediglich Nachrichten vom vorherigen Superstep empfangen. Dieses Ausführungsmodell wird *Bulk Synchronous Parallel* (kurz BSP) Modell [vgl. Val90] genannt. Ein weiterer wichtiger Bestandteil von Pregel sind die Zustände: *aktiv* und *inaktiv*. Jeder Knoten befindet sich zu jedem Zeitpunkt in genau einem dieser beiden Zustände. Empfängt ein Knoten eine Nachricht, wechselt bzw. verbleibt dieser im Zustand „aktiv“. Beim Versenden wird der Knoten als „inaktiv“ markiert. Während der Durchführung eines Supersteps führen nur „aktiv“ markierte Knoten die Knotenfunktion aus. Der Algorithmus terminiert, sobald sich am Ende eines Supersteps alle Knoten im Zustand „inaktiv“ befinden [vgl. Mal u. a.10].

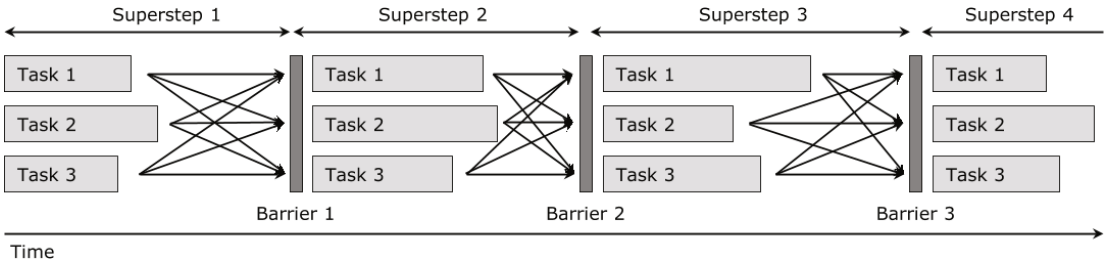


Abbildung 2.4: Beispiel einer BSP-Ausführung mit mehreren synchronisierten Supersteps *(Quelle: Entnommen aus [MWM15])*

Graphen sind rekursive Datenstrukturen. Dies bedeutet, dass ein Knoten in Zusammenhang mit seinen benachbarten Knoten steht und diese wiederum mit dessen Nachbarknoten. Als Konsequenz folgt eine häufige Neuberechnung von Knoten und die Laufzeitverschlechterung eines Algorithmus [vgl. Xin u. a.13]. Der *Pregel-Operator* in GraphX reguliert daher das Caching und Uncaching von Zwischenergebnissen, um Neuberechnungen möglichst gering zu halten [vgl. Apa14].

Die GraphX-Implementierung von Pregel sieht vor, dass Nachrichten nur an adjazente Knoten gesendet werden können. Zudem definiert der Anwender eine Funktion, die spezifiziert, an welche Knoten Nachrichten geschickt werden [vgl. And15]. Diese Anpassungen stellen, gegenüber anderen Pregel-Implementierungen, eine Optimierung von GraphX dar [vgl. Apa14].

3 Apache Flink

Apache Flink [vgl. ASF17d] ist ein Open-Source Stream Processing Framework. Der Ursprung liegt im Berliner Forschungsprojekt *Stratosphere* (2010) [vgl. Mar12]. Im Jahr 2014 übernahm die Apache Software Foundation (kurz ASF) die Plattform und entwickelt es seitdem als Top-Level Projekt unter dem Namen *Apache Flink* weiter.

Das Konzept hinter Flink verfolgt die Philosophie der nativen Datenstromverarbeitung. Datenverarbeitende Anwendungen lassen sich als fehlertolerante Datenströme innerhalb einer Pipeline ausdrücken und ausführen [vgl. Car u. a.15a]. Dazu zählen Echtzeit-Analysen, kontinuierliche Datenströme, konventionelle Batch-Verarbeitungen und iterative Algorithmen (bspw. maschinelles Lernen, Graph-Analyse).

Einen Überblick über das Ökosystem mit den Bibliotheken von Flink ist im Abschnitt 3.1 gegeben. Die grundlegende Dataflow Architektur wird im Abschnitt 3.2 zusammen mit der DataSet API behandelt. Der letzte Abschnitt (3.3) bezieht sich auf die native Unterstützung der Iteration und bildet einen Eckpfeiler für die Graphverarbeitung im Kapitel 4.

3.1 Ökosystem

Analog zu Spark besitzt Flink kein eigenes Speichersystem, sondern stellt Adapter bereit, um unterschiedliche Systeme zu integrieren. Dazu zählen das Hadoop *DFS*, *MongoDB* und klassische *SQL-Datenbanken*. Zudem gibt es Adaptionen zu Datenstrom-Quellen wie *Kafka* [vgl. ASF16b]. In der Abbildung 3.1 befinden sich die Speichersysteme unterhalb der *Deploy* Ebene und bilden das Datenfundament.

Für die Ausführung eines Flink Programms gibt es drei Kategorien. Im *lokalen-Modus* wird die Flink Anwendung innerhalb einer einzelnen Java VM ausgeführt (bspw. in der Entwickler-IDE). Bei großen Anwendungen, mit diversen verteilten Maschinen, kann auf den *Cluster-Modus* und *Cloud-Modus* zurückgegriffen werden. Die Cluster-Systeme, wie *Hadoop YARN* und *Mesos*, sind nativ verfügbar. Im Cloud-Modus kann die *Google Cloud* und *Amazon Cloud* eingebunden werden.

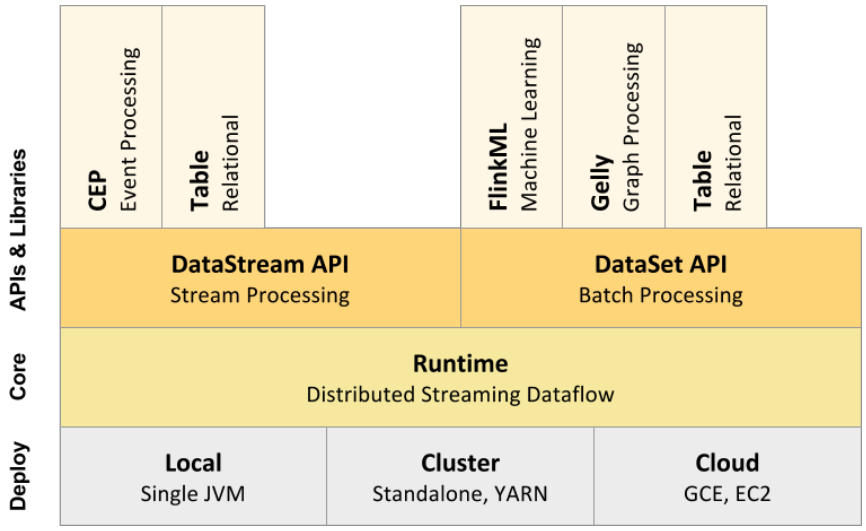


Abbildung 3.1: Apache Flink Komponentenübersicht
(Quelle: https://flink.apache.org)

Den Kern von Flink bildet die *Runtime*, auch *Kernel* genannt. Die Runtime ist eine verteilte Streaming Dataflow Engine und sorgt unter anderem für die parallele Ausführung. Zusätzlich fungiert diese Schicht als zuverlässige Basis für diverse Systemanforderungen, wie Ressourcenmanagement und Fehlertoleranz.

Oberhalb der Dataflow Engine befinden sich die *DataStream* und *DataSet* API. Die DataStream API verarbeitet potentiell unbegrenzte Datenströme (*Stream Processing*) und die DataSet API arbeitet mit begrenzten Datensätzen (*Batch Processing*). Beide APIs erzeugen intern Runtime Programme, welche durch die Dataflow Engine ausgeführt werden. Außerdem schaffen die APIs eine Plattform und Abstraktion für darauf aufbauende Bibliotheken. Flink integriert folgende Domain-spezifische Bibliotheken:

- **Flink ML**

  In der *Flink ML* werden Werkzeuge aus dem Bereich des maschinellen Lernens bereitgestellt. Die Bibliothek verwendet die DataSet API und die damit verfügbare native Iteration. Ein Schlüsselkonzept von Flink ML ist der Pipeline Mechanismus und orientiert sich hierbei an der Python ML-Bibliothek *scikit-learn* [vgl. Ped u. a.11].

- **Gelly**

  Gelly ist die Graph API von Flink. Sie unterstützt eine Vielzahl von Operationen zum Erstellen, Transformieren und Verarbeiten von Graphen. Im Kapitel 4 wird die Bibliothek detaillierter betrachtet.

- **Table**

  Die *Table* API ist eine SQL-ähnliche Abfragesprache und kann über die DataSet oder DataStream API angewendet werden. Die relationale Abstraktion *Table* lässt sich aus externen Datenquellen oder bestehenden DataStreams und DataSets erzeugen.

- **Flink CEP**

  Flink CEP ist die Bibliothek für *Complex Event Processing* und erkennt komplexe Events innerhalb eines kontinuierlichen Datenstroms. Über eine integrierte Pattern API lassen sich eigene Patterns anlegen und verwalten.

Jede Bibliothek besitzt eine breite Sammlung an Operationen und Algorithmen zur jeweiligen Domain. Zudem können über Java und Scala APIs eigene Erweiterungen umgesetzt werden. Im Falle der DataSet Abstraktion wird zusätzlich eine Python API angeboten.

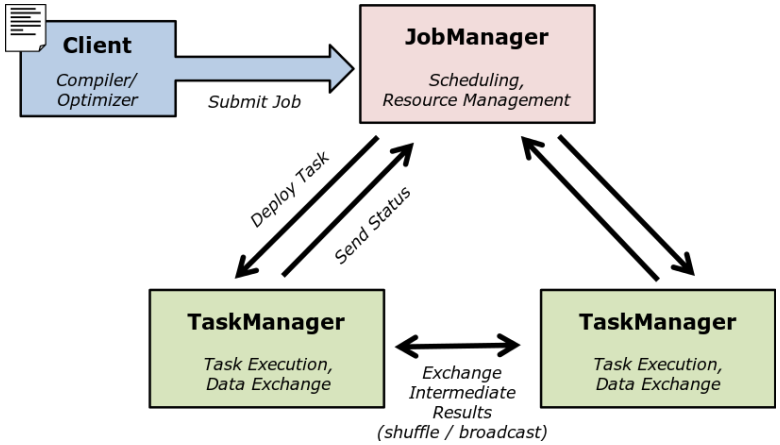


Abbildung 3.2: verteilte Ausführung eines Flink Programms; der JobManager ist der Koordinator im System und die TaskManager führen die einzelnen Programmteile parallel aus
(Quelle: https://flink.apache.org)

Flink Programme werden nach dem *Master-Slave* Prinzip ausgeführt. Das vom Benutzer geschriebene Flink Programm wird hierbei durch einen *JobGraph* abgebildet. Ein JobGraph ist

ein generischer, paralleler Dataflow mit beliebig kombinierbaren Funktionen (bspw. *map*, *reduce*, *join*). Dieser wird von der DataSet bzw. DataStream API erzeugt und von der Core Schicht (Abbildung 3.1) verarbeitet. Der *JobManager* (auch Master oder Koordinator) verteilt einzelne Teilberechnungen des JobGraphs an die *TaskManager* (auch Worker), verwaltet die Ressourcen und koordiniert Checkpoints sowie Wiederherstellungen. Die Bearbeitung des JobGraphs läuft parallel ab und kann beliebig horizontal skaliert werden. Die TaskManager tauschen benötigte Zwischenergebnisse selbständig aus und liefern in regelmäßigen Zeitabständen einen Status an den JobManager. Sobald ein TaskManager seine Berechnung erfolgreich absolviert hat, bekommt dieser eine neue unabhängige Aufgabe zugeteilt. Sollte ein TaskManager während seiner Ausführung ausfallen oder nicht mehr erreichbar sein, kann dies mittels *Checkpointing* ausgeglichen werden. Der von Flink verwendete Snapshot-Mechanismus ist konsistent und sorgt für die Fehlertoleranz [vgl. Car u. a.15b]. Im Falle eines Fehlers (bspw. Maschinen-, Netzwerk- oder Softwareausfalls) stoppt Flink diesen Bereich und startet die Operatoren vom letzten erfolgreichen Checkpoint neu.

3.2 Streaming Dataflow Modell

Die Basiskomponenten eines Flink Programms sind *Streams* und *Transformationen*. Konzeptionell ist ein Stream (dt. *Datenstrom*) eine nicht endende Abfolge von Datensätzen. Eine Transformation ist eine Operation, welche einen oder mehrere Streams als Eingabe bekommt und einen oder mehrere Streams als Ergebnis produziert. Bei der Ausführung wird ein Programm als *Streaming Dataflow* abgebildet, wie es exemplarisch in der Abbildung 3.3 zu sehen ist. Flink orientiert sich bei diesem Modell an Googles Dataflow Modell [vgl. Aki u. a.15]. Meistens gibt es eine 1:1 Übereinstimmung zwischen den Transformationen im Programm und den Operatoren im Dataflow. Es existieren jedoch Konstellationen, wo eine Transformation aus mehreren Transformation-Operatoren zusammengesetzt ist. Jeder Dataflow beginnt mit mindestens einer Quelle und endet in mindestens einer Senke. Ein Dataflow wird allgemein als gerichteter Graph präsentiert und von der Flink Runtime ausgeführt.

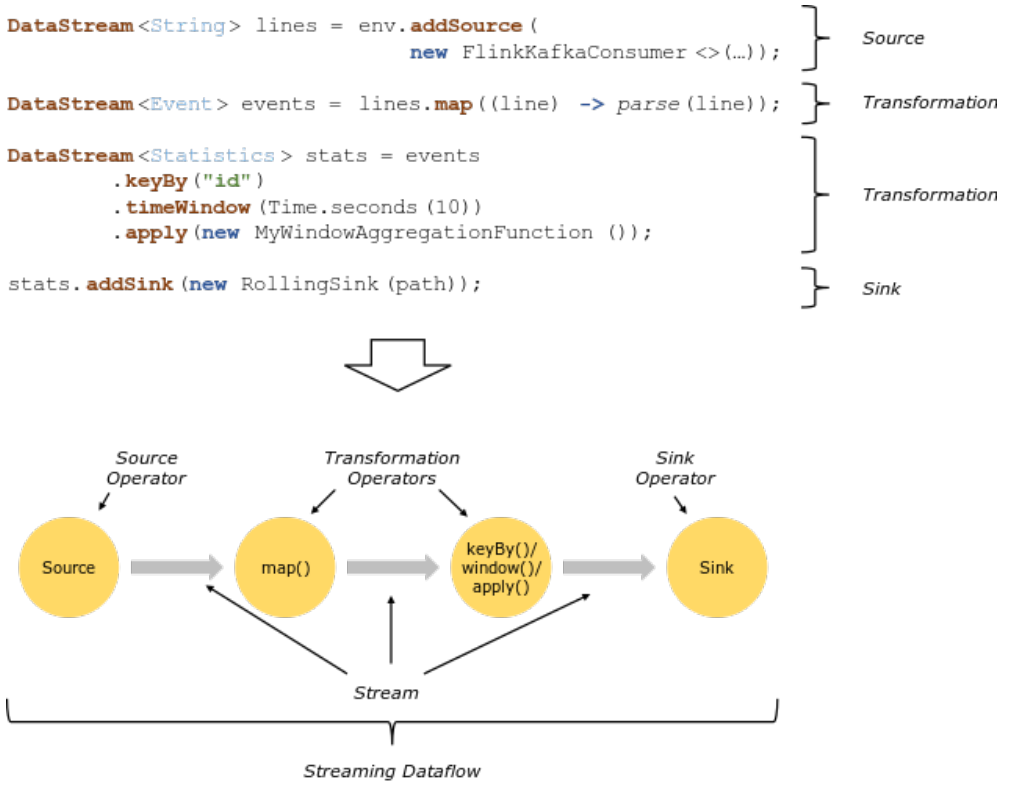


Abbildung 3.3: ein Flink DataStream Programm abgebildet auf die interne Streaming Dataflow Architektur
(Quelle: Entnommen aus [ASF17a])

3.2.1 Dataflow Graph

Ein Dataflow Graph ist ein *Directed Acyclic Graph* (kurz DAG, Abb. 3.4) und besteht aus zustandsbehafteten Operatoren und Streams [vgl. Car u. a.15a]. Ein Flink Programm ist von sich aus parallel und verteilt [vgl. ASF17a]. Bei der Ausführung wird ein Input-Stream in mehrere Stream Partitionen gespalten und jeder Operator teilt sich in Teilaufgaben auf. Die Verarbeitung der einzelnen Teilaufgaben verläuft unabhängig voneinander und wird in unterschiedlichen lokalen Threads aufgeführt. Ebenso können die Teilaufgaben und Partitionen auf verschiedenen Instanzen bzw. Containern ausgeübt werden.

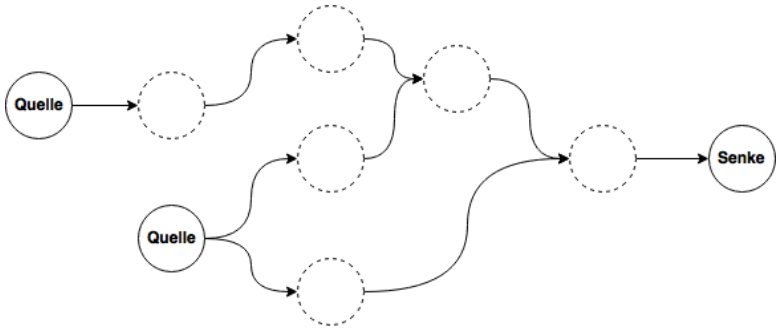


Abbildung 3.4: Ausführungsschema eines *Directed Acyclic Graph*; die leeren Knoten repräsentieren beliebig verknüpfte Operatoren; die Kanten entsprechen den Streams *(Quelle: Eigene Darstellung)*

Streams transportieren Daten zwischen zwei Operatoren nach folgenden Pattern: *1:1-Stream* oder *Weiterverteilung-Stream*. Bei 1:1-Streams wird die Reihenfolge der Partitionierung und der Elemente innerhalb einer Partition beibehalten. Beim zweiten Pattern ändert sich unter Umständen die Partitionierung. Jede Unteraufgabe sendet in diesem Fall Daten an verschiedene Ziel-Unteraufgaben, abhängig von der aktuellen Transformation. Beispielsweise findet eine Umverteilung beim *keyBy()* Operator statt, wo die Partitionen anhand eines Hash neu angeordnet werden. Die Operatoren sind in einigen Sonderfällen zustandslos und implementieren die Verarbeitungslogik (bspw. Filter, Hash-Join oder Stream Windowing Funktionen) [vgl. Car u. a.15a].

3.2.2 Spezialfall: DataSet API

Die DataSet API (Batch) bildet zusammen mit der DataStream API das Bindeglied zwischen der Flink Runtime und den Domain-spezifischen Bibliotheken. Technisch gesehen handelt es sich bei der DataSet API um einen Stream [vgl. VTK16]. Die Runtime stellt alle Flink Programme als Streaming Dataflows dar. Um die konventionelle Batch-Verarbeitung zu integrieren, bedient sich Flink einer einfachen Manipulation. Da ein Stream potenziell unendlich ist, wird dieser künstlich begrenzt und beinhaltet somit nur endlich viele Elemente. Der Vorteil besteht darin, dass Flink mit der selben Engine beide Verarbeitungstypen bewerkstelligen kann [vgl. ASF17a].

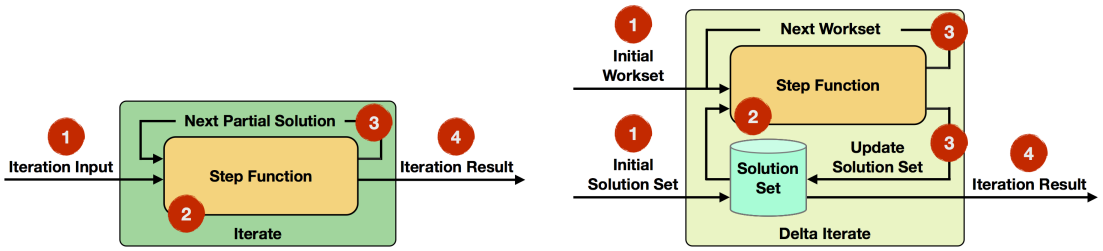
Des Weiteren kommt bei DataSet Programmen kein Checkpointing zum Einsatz. Die Fehlertoleranz wird mittels Wiederholung der fehlerhaften oder abgebrochenen Stream-Partition realisiert. Dies ist möglich, da die Eingabe begrenzt ist und Wiederholungen dort setzen an, wo das jüngste Zwischenergebnis erfolgreich materialisiert wurde.

Die Runtime von Flink unterstützt eine Vielzahl von Ausführungsstrategien, um den Aufwand gering zu halten. Dazu gehören die Neuzuweisung und die Übertragung des Datentransfers sowie verschiedene Verfahren zum Sortieren, Hashing, Join und Gruppieren [vgl. Car u. a.15a]. Die Runtime analysiert den Aufwand anhand diverser Faktoren wie Netzwerk- und Festplattenauslastung, aber auch die CPU-Auslastung und Datenbeschaffenheit fließen in die optimale Strategie mit ein [vgl. Mon u. a.16].

3.3 Batch Iterationen

Die *Directed Acyclic Graph* Repräsentation (Abschnitt 3.2.1) ist nicht immer eine praktikable Strategie. Iterative oder rekursive Graph-Algorithmen benötigen beispielsweise einen zyklischen Ausführungsgraphen [vgl. Jun u. a.17]. Hierfür stellt Flink zwei spezialisierte Operatoren für Iterationen zur Verfügung: *Bulk-* und *Delta Iteration*. Beide Varianten benutzen die DataSet API, da eine synchronisierte Iteration nur mit begrenzten Streams umsetzbar ist [vgl. ASF17a]. Der Flink eigene *Optimizer* erkennt Daten in einer Schleifen-Invariante und speichert diese automatisch im Zwischenspeicher (*caching*).

Hinter der *Bulk Iteration* befindet sich die klassische Iterationsform. Jede Iteration erzeugt ein neues Ergebnis, basierend auf dem Ergebnis der vorherigen Iteration. Das Iterationskonzept in Flink lässt sich in vier Phasen (Abbildung 3.5a) zerlegen: Der *Iteration Input* (1) ist das initiale DataSet für die erste Iteration. Die *Step Function* (2) nimmt die Ausgabe der vorangegangen Iteration als Eingabe und führt beliebige Transformationen (*map*, *reduce*, *join*, etc.) darauf aus. Das resultierende DataSet ist die nächste Teillösung (3) und wird für die folgende Iteration verwendet. Das *Iteration Result* (4) ist das erzeugte DataSet der letzten Iteration und wird abschließend materialisiert oder für weitere Operationen verwendet. Bei einer Bulk Iteration ist das *Konvergenzkriterium* eine maximale Anzahl von Iterationen oder ein benutzerdefiniertes Kriterium [vgl. Jun u. a.17].



(a) Bulk Iteration (b) Delta Iteration

Abbildung 3.5: Iterations-Operatoren in Apache Flink
(Quelle: https://flink.apache.org)

Der *Delta Iteration Operator* bezeichnet die Umsetzung einer inkrementellen Iteration. Bei der Delta Iteration werden Berechnungen nur auf der Teilmenge der Daten angewendet, die sich verändert haben. Die Motivation für diese Vorgehensweise sind Algorithmen, bei denen die Aktualisierung eines Elements nur wenige andere Elemente direkt beeinflusst [vgl. Ewe u. a.13].

Der grobe Aufbau dieser Iterationsform ähnelt der *Bulk Iteration*. In der Abbildung 3.5b lassen sich erneut vier Phasen erkennen. Im Unterschied zur Bulk Iteration gibt es in der ersten Phase zwei Eingabe-DataSets: *Initial Workset* (1) und *Initial Solution Set* (1). Das Solution Set entwickelt sich bei jeder Iteration weiter. Die *Step Function* (2) besteht aus einem beliebigen azyklischen Ablauf von Transformationen und wird sowohl auf dem Workset, als auch auf dem Solution Set ausgeführt. Analog zur Eingabe, besteht die Ausgabe der Step Function aus zwei DataSets: *Next Workset* (3) und *Update Solution Set* (3). Das neue Workset dient unmittelbar als neue Eingabe der Step Function. Das Update Solution Set beinhaltet die inkrementellen Aktualisierungen für das Initial Solution Set [vgl. Ewe u. a.13]. Nach der letzten Iteration stellt das Solution Set das Ergebnis der gesamten Iteration dar und kann auch hier in weiterführenden Operatoren benutzt werden. Beide Konvergenzkriterien der Bulk Iteration sind ebenso bei der Delta Iteration anwendbar. Zusätzlich zum benutzerdefinierten Kriterium und dem Erreichen einer maximalen Iterationsanzahl, terminiert eine Delta Iteration sobald das Workset leer ist [vgl. Jun u. a.17].

4 Gelly

Nachdem im Kapitel 3 grundlegende Konzepte und Mechanismen von Flink erläutert wurden, behandelt dieses Kapitel die *Gelly* Bibliothek.

Bei Gelly handelt es sich um eine Graph-spezifische Bibliothek und ist mit Hilfe der DataSet API von Flink implementiert. Neben unterschiedlichen Graph-Verarbeitungsmodellen, bietet Gelly eine eigene API mit zusätzlichen Operatoren und eine Vielzahl von Algorithmen an. Das zugrundeliegende Datenmodell ist ein gerichteter Multigraph mit frei definierbaren Attributen an Knoten und Kanten [vgl. Jun u. a.17].

Gelly erlaubt Anwendern das Ausüben einer *end-to-end* Datenanalyse (Vorverarbeitung, Graph-Erzeugung, Graph-Analyse und Nachbearbeitung) innerhalb des Flink Systems. Möglich ist dies durch eine nahtlose Anbindung zur DataSet API. Die Gelly Bibliothek und API sind in Java und Scala verfügbar. Bei den Scala Methoden handelt es sich um Wrapper Klassen der Java Methoden [vgl. ASF15].

Nachfolgend wird auf die Graph Repräsentation (Abs. 4.1) und die Optionen zur Modifikation sowie Transformation (Abs. 4.2) eingegangen. Im Abschnitt 4.3 werden anschließend die drei unterstützten Programmiermodelle für Graphen veranschaulicht.

4.1 Repräsentation von Graphen

In Gelly wird ein Graph intern über ein Knoten-DataSet und ein weiteres DataSet für die Kanten repräsentiert. Ein Knoten (Listing 4.1) beinhaltet einen eindeutigen und vergleichbaren Identifier (*ID*) sowie einen Wert. Knoten ohne einen Wert lassen sich mit dem Typ *NullValue* darstellen.

```
1 Vertex<Integer,String> v = new Vertex<Integer,String>(42,"Zukunft");
```

Listing 4.1: Erzeugung eines neuen Knotens mit Gelly; Java

Eine Graphkante (Listing 4.2) definiert sich durch eine ID vom Quellknoten, eine ID vom Zielknoten und einem optionalen Wert. Die IDs müssen dabei vom selben Typ sein, wie der Knotentyp. Um eine Kante ohne Wert zu erzeugen, ist der NullValue Typ zu verwenden. In Gelly sind Kanten immer vom Quell- zum Zielknoten gerichtet.

```
1 Edge<Integer,Integer,String> e = new Edge<Integer,Integer,String>
    (13,37,"BigData");
```

Listing 4.2: Erzeugung einer neuen Kante mit Gelly; Java

Alle IDs und Werte sind generisch und müssen zum Zeitpunkt der Grapherzeugung deklariert sein [vgl. Jun u. a.17]. Durch passende Gelly Methoden lassen sich Graphobjekte als DataSet-Objekte zurückgeben und ermöglichen ein nahtloses Kombinieren mit anderen Bibliotheken (bspw. FlinkML) innerhalb des Flink Systems. Ein Gelly-Graph verfügt über konventionelle Methoden zum Abrufen von Eigenschaften und Metriken. So können unter anderem Informationen zum Knotengrad (ein- und ausgehend) oder die Knoten bzw. Kantenanzahl abgefragt werden.

4.2 Modifikation, Transformation und Nachbarschaft

Gelly stellt eine Reihe von Funktionen zur Modifikation, Transformation und Nachbarschaft bereit. Diese Operationen lassen sich dabei flexibel und in unterschiedlicher Reihenfolge kombinieren und ausführen.

Durch eine Modifikation am Graphen lassen sich nachträglich Knoten bzw. Kanten hinzufügen und löschen. Das Ergebnis einer Modifikation ist ein neuer Graph mit aktualisierten DataSets gegenüber dem Ursprungsgraphen.

Transformationen werden auf einem Eingabegraphen ausgeführt und aufgrund der immutable Eigenschaft von DataSets führt dies zur Erzeugung eines neuen Graphen. Intern übersetzt Gelly eine einzelne Graphtransformation in eine Folge von Transformationen der Knoten und Kanten [vgl. ASF15]. Auf einem Graphen lassen sich folgende Transformationen ausführen:

- **Map** erlaubt die Modifikation von Knoten- und Kantenwerten durch eine benutzerdefinierte Map-Funktion. Das Ergebnis ist ein neuer Graph, dessen IDs unverändert bleiben. Die Funktion kann zudem benutzt werden, um den Wertetyp zu verändern.

- **Translate** sind spezialisierte Map-Funktionen, um a) die Knotenwerte (*translateVertexValues*), b) die Kantenwerte (*translateEdgeValues*) oder c) die Werte und/ oder Typen der IDs nachträglich anzupassen.

- **Filter** lassen sich auf Graphen anwenden und erzeugen einen neuen Teilgraph. Knoten bzw. Kanten, die eine benutzerdefinierte logische Aussage erfüllen, werden aus dem Eingabegraphen extrahiert und bilden zum Schluss einen neuen Graphen. Beispielsweise kann ein Teilgraph erzeugt werden, in dem alle Kantenwerte größer gleich 10 sind.

- **Join** erlaubt die Kombination von Knoten- und Kanten-DataSets mit zusätzlichen Eingabe-DataSets. Über diese Transformation können bspw. externe Daten einer relationalen Datenbank einem bestehenden Gelly-Graphen hinzugefügt werden.
- **Reverse** erzeugt einen neuen Graphen, wo die Richtung aller Kanten umgekehrt ist.
- **Undirected** ist eine schnelle Möglichkeit, um aus einem gerichteten einen ungerichteten Graphen zu erzeugen. In Gelly sind Kanten immer gerichtet. Für eine ungerichtete Kante muss diese zunächst dupliziert und mit *reverse()* umgedreht werden.
- **Union/ Difference/ Intersect** erlaubt das Zusammenführen von zwei Eingabegraphen in einen neuen Graphen. Diese Transformationen basieren auf der Theorie der Mengenlehre (Vereinigungsmenge/ Differenz/ Schnittmenge).

Neben den Modifikationen und Transformationen bietet Gelly Methoden für Nachbarschaftsbeziehungen von Knoten an. Die Methoden lassen sich auf den inzidenten Kanten und adjazenten Knoten eines jeden Knotens anwenden [vgl. ASF15]. Eine Anwendungsmöglichkeit ist die Aggregation von benachbarten Knotenwerten, um dessen Durchschnitt oder Maximum zu ermitteln. Gelly stellt zwei Varianten für Nachbarschaften zur Verfügung: *reduceOnEdges/ -Neighbors* und *groupReduceOnEdges/ -Neighbors*. Die erste Variante erlaubt die Aggregation von Knoten und Kanten durch eine benutzerdefinierte assoziative und kommutative Funktion [vgl. Jun u. a.17]. Sofern die Aggregationsfunktion nicht assoziativ und auch nicht kommutativ ist, lässt sich die erste Variante nicht anwenden. Dafür ist die zweite Variante zuständig, welche es außerdem erlaubt, mehrere Werte pro Knoten zurückzugeben.

4.3 Verteilte Graphverarbeitung

Dieser Abschnitt behandelt die von Gelly umgesetzten high-level Programmiermodelle für eine verteilte Graphverarbeitung. Ein solches Modell ist eine Abstraktion der unterliegenden Berechnungsinfrastruktur und erlaubt die Definition von Graphdatenstrukturen und Graphalgorithmen. Ein Anwendungsentwickler kann sich auf die Logik im Algorithmus konzentrieren, da beispielsweise von der Datenpartitionierung, den Kommunikationsmechanismen und der Systemarchitektur abstrahiert wird.

Neben dem *vertex-centric* Modell (4.3.1) gibt es eine Implementierung vom *scatter-gather* (4.3.2) und *gather-sum-apply* (4.3.3) Modell. Gelly greift dazu auf die Delta Iteration von Flink zurück, um diese Modelle umzusetzen.

Die verschiedenen Abstraktionen basieren auf der allgemeinen Architektur für verteilte Graphverarbeitung-Frameworks: Dem *Master-Worker* Prinzip. In dieser Architektur dient der Master als Koordinator und die Worker führen die verteilten Verarbeitungen durch. Der Eingabegraph ist dabei partitioniert und auf den Worker-Einheiten verteilt [vgl. Jun u. a.17]. Bei der Partitionierung des Graphen verwendet Gelly derzeit eine random/ hashing-Kombination, die auf der Knoten-ID basiert.

System	Jahr	Programmiermodell	Ausführung	Kommunik.
GraphX	2014	GAS	synchr. mit BSP	Dataflow
Gelly	2015	vertex-centric, scatter-gather, GSA	synchr. mit BSP	Dataflow

Tabelle 4.1: Gegenüberstellung der verteilten Graphverarbeitungssysteme GraphX und Gelly im Bezug auf unterstützte Programmiermodelle, Ausführungsmodell und Kommunikationsmechanismus
(Quelle: Entnommen aus [KVH16])

4.3.1 vertex-centric

Das vertex-centric Modell äußert sich durch die perspektivische Sicht eines Knotens und ist auch als „Pregel-Modell“ bekannt. Eine vertex-centric Anwendung bekommt als Eingabe einen gerichteten Graphen und eine Knotenfunktion. Die Knoten kommunizieren mit Nachrichten entlang der Kanten mit anderen Knoten. Der Nachrichtenaustausch setzt ein Wissen über die ID des Zielknotens voraus.

Der Ablauf einer vereinfachten Ausführung ist in der Abbildung 4.1 visualisiert. Die Berechnungen verlaufen innerhalb von synchronisierten Supersteps (Iterationsschritte). Bei jedem Superstep führen die gepunkteten Kästen parallel die benutzerdefinierte Knotenfunktion aus. Der Nachrichtenverlauf zwischen den Supersteps ist in der Abbildung mit den gepunkteten Linien dargestellt. Unter Gelly definiert der Anwender die Knotenfunktion und legt die maximale Anzahl der Iterationsschritte fest. Die optionale Funktion *MessageCombiner* ermöglicht, durch akkumulierte Nachrichten, eine Reduktion der Übertragungskosten [vgl. ASF17b]. Auch GraphX setzt mit dem Pregel Operator auf dieses Iterationsmodell.

Das Modell eignet sich bei Algorithmen, wo die Knotenfunktion lokal formuliert werden kann und nur Zugriff auf adjazente Knoten benötigt [vgl. KVH16]. Ein iterativer Algorithmus passt somit in das Schema des vertex-centric Modells. Der Vorteil liegt in der Einfachheit und der linearen Skalierung von iterativen Graphalgorithmen. Nicht-iterative Algorithmen lassen sich aufgrund des Superstep-Konzept nur schwer mit dem Modell umsetzen.

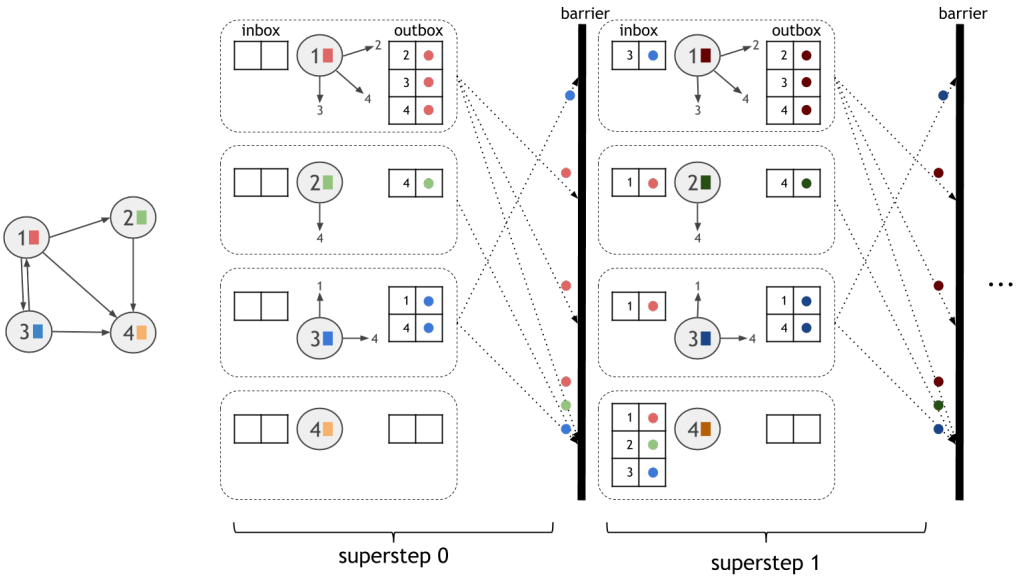


Abbildung 4.1: Superstep Ausführung und Nachrichtenverlauf (mit BSP Barriere) im vertex-centric Modell [vereinfachte Darstellung]
(Quelle: Entnommen aus [ASF17b])

In einem Nachrichtenübermittlungsmodell wird die Kommunikation häufig zum Flaschenhals [vgl. And15]. Es existieren unterschiedliche Vorgehensweisen, um diese Einschränkungen zu verringern. Ein Ansatz verfolgt die Reduktion der zu übertragenen Nachrichten. In einem anderen Ansatz werden Knoten mit einem hohem Knotengrad dupliziert und auf weiteren Workern verteilt. Den benachbarten Knoten stehen somit mehrere Instanzen zur Verfügung [vgl. KVH16].

4.3.2 scatter-gather

Das Scatter-Gather Modell (auch *Signal-Collect* [vgl. SBC10]) teilt die selbe „think like a vertex“ Philosophie wie das vorherige vertex-centric Modell. Konzeptionell unterscheiden sich beide Modelle kaum voneinander. Die Iterationen funktionieren als synchronisierte Supersteps und Nachrichten dienen als Kommunikationsmechanismus zwischen den Knoten. Der Hauptunterschied stellt die geteilte Berechnungsphase dar. Muss im vertex-centric Modell nur eine Knotenfunktion definiert werden, sind es bei diesem Modell zwei benutzerdefinierte Funktionen: *scatter* und *gather*.

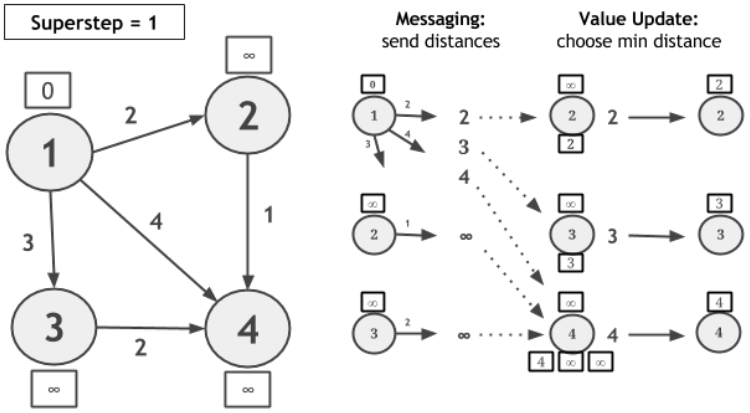


Abbildung 4.2: Scatter (*Messaging*) und Gather (*Value Update*) Phase anhand des Single-Source-Shortest-Paths Algorithmus [Ausschnitt]
(Quelle: Entnommen aus [ASF17b])

Während der Scatter Phase (*Messaging* in Abb. 4.2) führt jeder Knoten die *ScatterFunction* aus und übermittelt Nachrichten entlang der ausgehenden Kanten. In der Gather Phase (*Value Update* in Abb. 4.2) wird die *GatherFunction* auf Basis der empfangenen Nachrichten ausgeführt und der Knotenstatus aktualisiert [vgl. Jun u. a.17]. Eine Besonderheit ist, dass empfangende Nachrichten im selben Superstep versendet wurden [vgl. KVH16]. Analog zum vertex-centric Modell befinden sich Knoten entweder im aktiven oder im inaktiven Zustand. Sofern ein Knoten v seinen Wert nicht in der Gather Phase aktualisiert hat, führt der Knoten v die Scatter Funktion im nachfolgendem Superstep nicht aus. Seine Nachbarn verfügen bereits über seine aktuellen Informationen. Der Knoten befindet sich somit im Zustand „inaktiv“.

Es ist zu beachten, dass die Gültigkeitsbereiche beider Funktionen getrennt sind und sich die Interfaces unterscheiden. Über das Scatter Interface kann der aktuelle Knotenwert und der Status benachbarter Kanten abgerufen werden. Zusätzlich lassen sich Nachrichten an angrenzende Knoten schicken [vgl. KVH16]. Das Gather Interface besitzt Zugriff auf empfangene Nachrichten und kann den Knotenwert auslesen und manipulieren.

Das Scatter-Gather Modell eignet sich für knappe Algorithmen. Durch die Entkoppelung der zwei Phasen, sind Algorithmen ggf. besser nachvollziehbar und leichter zu optimieren [vgl. KVH16]. Typischerweise besitzen Scatter-Gather Algorithmen eine geringe Speicheranforderung. Ein Grund dafür ist der nicht simultane Zugriff auf den Nachrichteneingang und -ausgang. Eine negative Auswirkung durch die Entkopplung ist, dass Knoten in der selben Phase keine Nachrichten erzeugen und den Wert nicht aktualisieren können. Bei Graphen

mit stark zusammenhängenden Komponenten benötigen Algorithmen häufig diese beiden Eigenschaften. Das Scatter-Gather Modell ist eingeschränkter als das vertex-centric Modell, bietet bei bestimmten Algorithmen jedoch Vorteile [vgl. KVH16].

4.3.3 gather-sum-apply

Das Gather-Sum-Apply (kurz GSA) ist eine Variante des *Gather-Apply-Scatter* (kurz GAS) Modells. Das GAS Modell wurde erstmalig von *PowerGraph* [vgl. GLG12] eingeführt und setzt bei den Performance Problemen von *power-law Graphen* an. Bei solchen Graphen besitzen die meisten Knoten einen relativ geringen Knotengrad, während eine kleine Knotenteilmenge mit einem Großteil der Graphknoten verbunden ist. In einem Superstep müssen die wenigen Knoten mit einem hohen Grad mehr Informationen verarbeiten, als die restlichen Knoten. Aufgrund der Barriere des BSP-Mechanismus verzögern sich die Supersteps [vgl. KVH16]. Beim GSA und GAS Modell erfolgt die Berechnung in der Gather-Phase parallel auf den Kanten [vgl. Jun u. a.17; KVH16].

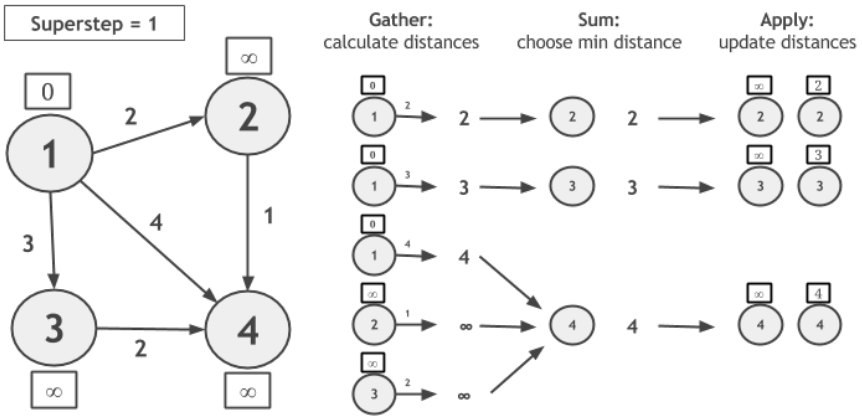


Abbildung 4.3: Die drei Phasen im gather-sum-apply Modell anhand des Single-Source-Shortest-Paths Algorithmus [Ausschnitt]
(Quelle: Entnommen aus [ASF17b])

Das GSA Modell unterteilt einen Superstep in drei Phasen mit je einer benutzerdefinierten Funktion: *gather-*, *sum-* und *apply*-Phase. In der Gather Phase wird die *GatherFunction* in der direkten Nachbarschaft jedes Knotens angewendet. Das Ergebnis ist ein Teilwert pro Knoten, bestehend aus den adjazenten Knoten und den inzidenten Kantenpaaren. In der Sum-Phase aggregiert die *SumFunction* alle Teilwerte aus der Nachbarschaft zu einem Gesamtwert. In der letzten Apply Phase berechnet die *ApplyFunction* aus der Aggregation und dem eigenen

Knotenwert schlussendlich den neuen Knotenwert. Um die Berechnungszeit und den Netzwerkverkehr zu reduzieren, werden Ergebnisse in der Sum-Phase intern gebündelt, bevor diese übermittelt werden. Im Gegensatz zu Scatter-Gather rufen die Knoten die Informationen bei den Nachbarn ab und bekommen diese nicht von ihnen zugesandt [vgl. Jun u. a.17].

4.3.4 Gelly spezifische Modell-Konfiguration

Die drei Modelle vertex-centric (4.3.1), scatter-gather (4.3.2) und gather-sum-apply (4.3.3) lassen sich zusätzlich spezifizieren. Die jeweiligen Konfigurations-Objekte verfügen über allgemeine und spezielle Parameter. In der nachfolgenden Tabelle (4.2) sind diese aufgelistet und werden anschließend kurz beschrieben.

Parameter	vertex-centric	scatter-gather	gather-sum-apply
Bezeichner	✓	✓	✓
Parallelität	✓	✓	✓
Speicherverwaltung	✓	✓	✓
Aggregatoren	✓	✓	✓
Broadcast Variablen	✓	✓	✓
Knotenanzahl		✓	✓
Knotengrad		✓	
Nachrichtenrichtung		✓	
Nachbar Richtung			✓

Tabelle 4.2: verfügbare Parameter, die in den jeweiligen Programmiermodellen konfiguriert werden können
Quelle: Eigene Darstellung

- **Bezeichner**

  Der Bezeichner für eine Iteration wird in den Logs und in den Status-Meldungen benutzt. Der Wert wird über die *setName()* Methode gesetzt.

- **Parallelität**

  Mit dem Parameter kann definiert werden, über wie viele parallele Instanzen die Iteration maximal ausgeführt wird. Die Methode *setParallelism()* legt den Integer Wert fest.

- **Speicherverwaltung**

  Flink behält Objekte intern standardmäßig in einer serialisierten Form. Über die Methode *setSolutionSetUnmanagedMemory()* wird das Ergebnis als Heap verwaltet.

- **Aggregatoren**

  Aggregatoren können über die *registerAggregator()* Methode registriert werden. Ein Iterationsaggregator kombiniert alle Aggregate in einem Superstep und ermöglicht einen externen Zugriff. In der scatter-gather bzw. gather-sum-apply Iteration kann innerhalb der benutzerdefinierten Funktionen auf die registrierten Aggregatoren zugegriffen werden.

- **Broadcast Variablen**

  DataSets können als Broadcast Variablen freigegeben werden und sind dadurch für andere parallele Instanzen einer Operation verfügbar.

- **Knotenanzahl**

  Durch das Setzen der *setOptNumVertices()* Methode kann während einer Iteration auf die Gesamtknotenzahl zugegriffen werden. Mit einer weiteren Methode *getNumberOfVertices()* wird der Integer-Wert abgerufen. Sofern diese Eigenschaft nicht gesetzt ist, gibt die zweite Funktion den Wert -1 zurück.

- **Knotengrad**

  In einer scatter-gather Iteration besteht die Option zum Abfragen der Knotengrade und wird mit der *setOptDegrees()* Methode aktiviert. Über die Methoden *getInDegree()* bzw. *getOutDegree()* wird der Knotengrad für ein- bzw. ausgehende Kanten zurückgegeben. Sofern diese Eigenschaft nicht gesetzt ist, geben die Funktionen den Wert -1 zurück.

- **Nachrichtenrichtung**

  Standardmäßig versendet ein Knoten neue Nachrichten entlang ausgehender Kanten und aktualisiert den eigenen Wert basierend auf Nachrichten von eingehenden Kanten. Durch das Setzen von *EdgeDirection.IN*, *EdgeDirection.OUT* oder *EdgeDirection.ALL* wird dieses Verhalten beeinflusst. Diese Eigenschaft wird mit der *setDirection()* Methode manipuliert und steht nur in der scatter-gather Iteration zur Verfügung.

- **Nachbarkanten Richtung**

  Bei der GSA Iteration ruft ein Knoten v die Informationen bei den Nachbarknoten selbst ab. Standardmäßig werden nur Nachbarknoten abgefragt, die eine ausgehende Kante zu v haben. Durch das Anpassen der *setDirection()* Methode mit *EdgeDirection.IN*, *EdgeDirection.OUT* oder *EdgeDirection.ALL* kann dies geändert werden.

4.3.5 Gegenüberstellung der Modelle

Obwohl sich die drei zuvor vorgestellten Modelle ähnlich sehen, weisen sie im Detail einige Unterschiede auf. In diesem Abschnitt sollen die Unterschiedlichkeiten hervorgehoben und in der Tabelle 4.3 zusammengefasst werden.

Das vertex-centric Modell ist die allgemeinste Abstraktion und unterstützt eine frei wählbare Berechnung sowie den Nachrichtenaustausch zu jedem Knoten. Im scatter-gather Modell ist die Logik der Nachrichtenerzeugung von der Knotenwertaktualisierung entkoppelt. In Folge dessen kann ein Knoten innerhalb einer Phase keine Nachrichten empfangen und seinen Wert berechnen. Algorithmen werden jedoch verständlicher und bringen meistens eine bessere Performance mit sich. Wenn ein Algorithmus konzeptionell voraussetzt, dass ein Knoten nebenläufig Nachrichten entsenden und den Knotenwert anpassen kann, ist das scatter-gather Modell problematisch. Entsprechende Workarounds führen meistens zu einer höheren Speicherauslastung und verkomplizieren den Algorithmus [vgl. KVH16].

Modell	**Update Funktion**	**Update Logik**	**Kommunik.-bereich**	**Kommunik.-logik**
vertex-centric	beliebig	beliebig	jeder Knoten	beliebig
scatter-gather	beliebig	basiert auf empfangene Nachrichten	jeder Knoten	basiert auf Knotenstatus
GSA	assoziativ & kommutativ	basiert auf Nachbarwerte	Nachbarschaft	basiert auf Knotenstatus

Tabelle 4.3: Vergleich des vertex-centric, scatter-gather und GSA Modells

Das GSA Modell weist starke Ähnlichkeiten zum scatter-gather Modell auf. Tatsächlich lässt sich jeder GSA-Algorithmus als eine scatter-gather Implementierung ausdrücken. Der Hauptunterschied liegt in der Parallelisierung der Gather Phase im GSA Modell. Die Berechnung erfolgt dabei über die Kanten und nicht über die Knoten wie beim scatter-gather Modell. In der Abbildung 4.2 (scatter-gather) erzeugen die Knoten 1, 2 und 3 ihre Nachrichten parallel. Der erste Knoten generiert drei und die anderen beiden jeweils eine Nachricht. Verglichen mit der Abbildung 4.3 (GSA) lässt sich hier erkennen, dass die drei Nachrichten beim ersten Knoten parallel generiert werden. Wenn in der Gather Phase größere Berechnungen beinhaltet sind, bietet sich die verteilte Ausführung des GSA Modells an. Im scatther-gather Modell haben Knoten mit vielen Nachbarn (power-law Graph) eine größere Berechnungslast zu bewältigen und dies müsste beispielsweise durch leistungsstärkere Hardware ausgeglichen werden.

Ein weiterer Unterschied zwischen den beiden Modellen ist, dass Gelly bei der Implementierung des scatter-gather Modells den *coGroup* Operator intern verwendet und bei GSA hingegen den *reduce* Operator.

Das vertex-centric und scatter-gather Modell sind technisch in der Lage, an jeden Knoten eine Nachricht zu senden. Unabhängig ob sich dieser Knoten in der Nachbarschaft befindet, wie es beim GSA Modell notwendig ist.

4.4 Bibliothek und Algorithmen

Neben den Operationen zur Strukturveränderung (4.2), verfügt Gelly über eine vielseitige Sammlung von Algorithmen. Über den Methodenaufruf *run()* können statische Graphanalysen durchgeführt werden [vgl. ASF17c]. Die Abstraktion *Graph* liefert eine graphorientierte Sicht auf die Daten und bildet die Grundlage der aufzurufenden Algorithmen.

```
1 ExecutionEnvironment env = ExecutionEnvironment.
    getExecutionEnvironment();
2
3 Graph<Long,Long,NullValue> graph = Graph.fromTupleDataSet(vertices,
    edges, env);
4
5 DataSet<Vertex<Long, Long>> verticesWithCommunity = graph.run(new
    LabelPropagation<Long>(30));
```

Listing 4.3: [Ausschnitt] Ausführung einer Label Propagation mit 30 Iterationen, um Zusammenhänge im Eingabegraphen festzustellen; Java

Eine Auswahl dieser iterativen Algorithmen soll nachfolgend kurz vorgestellt werden. Einige sind in der Bibliothek mehrfach vorzufinden, da der Algorithmus mit Hilfe mehrerer Modelle (4.3) implementiert ist. Für den Anwender resultiert dadurch eine flexiblere Wahl des Algorithmus, je nach der Strukturgegebenheit des Eingabegraphen.

Ein Anwendungsfall in der Analyse von Graphen ist es, (zusammenhängende) Komponente zu identifizieren und somit die Knoten-Objekte zu klassifizieren [vgl. Leu u. a.09]. Gelly unterstützt dies unter anderem mit dem *Community Detection* oder dem *Label Propagation* Algorithmus. Beide sind mittels des scatter-gather Modells umgesetzt. Der *Connected Components* implementiert den *Weakly Connected Components* Algorithmus und liegt in zwei Modell-Varianten vor. Zum einen in der scatter-gather Version und zusätzlich als gather-sum-apply.

Speziell im Big Data Umfeld sind Graphen häufig zu groß, um diese vollständig zu visualisieren [vgl. ASF17c]. Der *Summarization* Algorithmus berechnet einen komprimierten Graphen, indem Knoten und Kanten basierend auf ihren Werten gruppiert werden. Im Ergebnisgraphen repräsentiert jeder Knoten eine Gruppe von Knoten mit dem selben Wert. Eine Kante, die einen Knoten mit sich selbst verbindet (*self-loop*), spiegelt alle Kanten wieder, die Knoten innerhalb der gleichen Knotengruppe verknüpfen und den identischem Kantenwert vorweisen. Wenn eine Kante hingegen im Ergebnisgraphen ungleiche Knoten in Relation stellt, handelt es sich im Eingabegraphen um alle Kanten mit einheitlichem Kantenwert, jedoch unterschiedlichem Knotenwert [vgl. ASF17c]. Gelly verwendet beim Summarization Algorithmus keine Modellabstraktion, sondern arbeitet unmittelbar mit der DataSet API, um u.a Knoten und Kanten nach den Werten zu gruppieren (*GroupReduceOperator*).

Der *PageRank* Algorithmus ist ein Verfahren, um die Wichtigkeit einer Website anhand der eingehenden Links zu bewerten. Erfunden und patentiert wurde er von den Google Gründern Larry Page und Sergei Brin im Jahr 1997. Je mehr eingehende Links (gerichtete Kanten) eine Website (Knoten) besitzt, desto höher wird sie bewertet und desto besser ist ihr PageRank-Wert. In der Kategorie der Link-Analyse Algorithmen befindet sich ebenfalls der *Hyperlink-Induced Topic Search*.

Für Clustering-Analysen stehen dem Anwender unterschiedliche Algorithmen zur Koeffizientenbestimmung, Triadic Census und Triangle Listing zur Verfügung. Letzterer benutzt Optimierungen des *Schrank-Algorithmus* [vgl. Sch u. a.01], um die Performance bei sehr hohen Knotengraden zu verbessern.

Um die Ähnlichkeit zwischen mehreren Knoten zu ermitteln, kann auf den *Adamic-Adar* oder dem *Jaccard Index* zurückgegriffen werden.

Neben den komplexeren Algorithmen stellt Gelly einfachere und fundamentale Metriken bereit. Diese unterteilen sich zunächst in Knoten- bzw. Kantenmetriken und bieten Statistiken für gerichtete und ungerichtete Graphen. Es kann beispielsweise die Knoten- und Kantenanzahl oder der minimale/ maximale/ durchschnittliche Knotengrad abgefragt werden.

5 Erweiterung der Gelly Bibliothek

In diesem Kapitel wird die Gelly Bibliothek um den *Semi-Clustering* Algorithmus erweitert. In der Thesis „Funktionale Erweiterung des GraphX Frameworks“ [vgl. And15] beschäftigte sich der Autor mit der Implementierung des Semi-Clustering Verfahrens und nutzte dafür die Pregel API von Apache Spark. Beschrieben wurde das Verfahren erstmalig von Malewicz, Austern, Bik u. a. [vgl. Mal u. a.10]. In den nachfolgenden Experimenten (Kapitel 6) sollen beide Implementierungen als Anhaltspunkte für eine Evaluierung dienen.

Die Grundlagen für die Erweiterung der Gelly Bibliothek sind in den vorherigen Kapiteln beschrieben und finden in diesem Kapitel ihre Anwendung. Zunächst wird der Algorithmus in den Big Data Kontext eingeordnet (5.1) und das Konzept erläutert (5.2). Im letzten Abschnitt wird die Umsetzung durch Gelly (5.3) behandelt.

5.1 Allgemeines zum Semi-Clustering

Die nachrichtenorientierte Funktionsweise von Pregel findet häufig Anwendung in der Clusteranalyse [vgl. And15]. Das Semi-Clustering Verfahren ist eine Variante zum Gruppieren und stammt aus der Graph-Analyse von sozialen Netzwerken. In sog. *social graphs* repräsentieren die Knoten typischerweise Personen und eine Kante verdeutlicht die Relation zwischen zwei Personen. Relationen können dabei auf expliziten Aktionen basieren, wie dem Hinzufügen eines neuen Freundes oder lassen sich durch das Verhalten der Person ableiten (bspw. Kommunikation via Chat). Zusätzlich zur Kantenrichtung können Kanten über eine Gewichtung verfügen, um eine Kommunikationsfrequenz oder eine Intensität auszudrücken [vgl. Mal u. a.10].

Das agglomerative Semi-Clustering Verfahren erzeugt iterativ eine benutzerdefinierte Anzahl von Semi-Clustern. Ein solches Cluster entspricht im social graph einer Gruppe von Personen, die eine hohe Kommunikationsfrequenz untereinander und eine geringe Interaktion mit anderen aufweisen. Bei diesem Verfahren ist zudem vorgesehen, dass sich eine Person in mehreren Clustern gleichzeitig befinden darf.

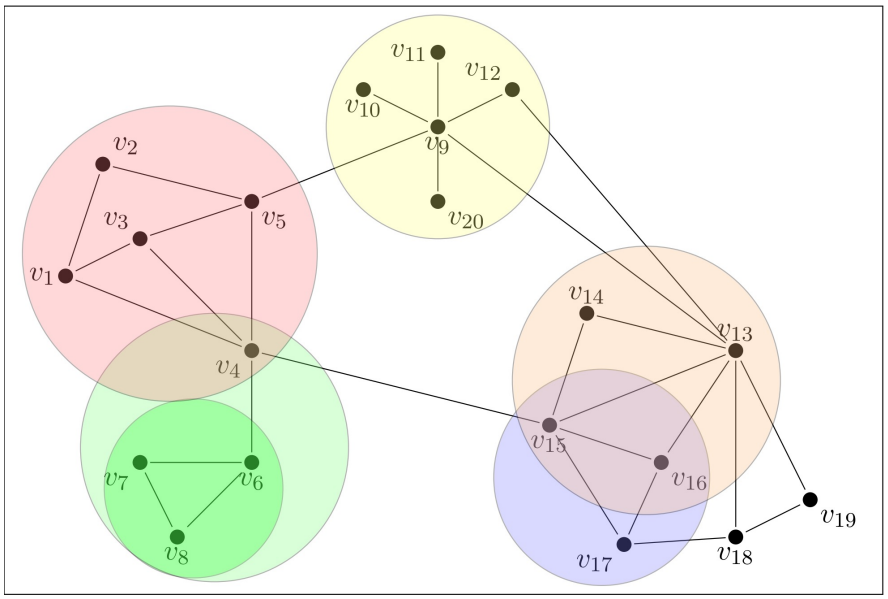


Abbildung 5.1: Grafische Darstellung der sechs besten Semi-Cluster als Ergebnis des Semi-Clustering Verfahren
(Quelle: Entnommen aus [And15])

5.2 Konzept

Als Ausgangspunkt dient ein ungerichteter Graph mit einer Gewichtung an den Kanten. Ein hohes Gewicht entspricht einer intensiveren Kommunikation. Alternativ werden alle Kanten mit dem selbem Gewicht versehen, wenn die Intensität irrelevant ist und lediglich die Verbindungen selbst im Vordergrund stehen [vgl. And15]. Das Ziel ist es, Semi-Cluster (Gruppierungen) von Personen zu finden, die untereinander viel interagieren.

Ein Cluster wird nach dem greedy Prinzip aufgebaut und verfügt neben den enthaltenen Knoten einen zusätzlich Wert S, um die Semi-Cluster vergleichen zu können. Der Wert S_c entspricht einer Kennzahl und 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 Gewichte der inneren Kanten ist. Eine innere Kante verbindet dabei zwei Knoten des selben Clusters. B_c ist demgegenüber die Summe aller Gewichtungen der Kanten, die aus dem Cluster c herausragen. Die Startknoten dieser Kanten befinden sich dementsprechend nicht im gleichen Cluster wie dessen Zielknoten. Der Faktor f_B reguliert

die Bedeutsamkeit von B_c. Dabei handelt es sich um einen benutzerdefinierten Parameter und befindet sich üblicherweise im Intervall von [0, 1]. Im Nenner der Formel steht die Anzahl aller Kanten, die innerhalb des Clusters c liegen. Durch diese Normalisierung erreichen sehr große Cluster keine künstlich hohe Kennzahl [vgl. Mal u. a.10]. Die Variable V_c ist die Anzahl der Knoten im Cluster c.

Das Semi-Clustering basiert auf dem Pregel Modell und durchläuft somit eine definierte Sequenz von Supersteps. Jeder Knoten des Eingabegraphen verwaltet eine sortierte Liste von Clustern, die anfänglich leer ist. Die Ordnung der Sortierung ist dabei entweder ab- oder aufsteigend und verwendet die Kennzahl S eines Clusters.

Als Vorbereitung fügt sich jeder Knoten v in seine eigene Liste als Cluster hinzu und setzt S auf eins. Nach diesem Schritt beträgt sowohl die Cluster-Größe, als auch die Kennzahl S den initialen Wert eins. Anschließend verschickt jeder Knoten seine Cluster-Liste als Nachricht an alle seine Nachbarknoten und leitet so eine Sequenz von Supersteps ein. Jeder Superstep durchläuft dabei folgende drei Schritte:

1. Der Knoten v iteriert über die eingehenden Cluster c_1, ..., c_k, die aus dem vorherigen Superstep stammen. Wenn v noch nicht in einem Cluster c enthalten ist, wird ein neues dupliziertes Cluster $c´$ erzeugt und dieses um v erweitert.

2. Die Semi-Cluster c_1, ..., c_k und $c´_1$, ..., $c´_k$ werden nach ihrer jeweiligen Kennzahl S sortiert. Über eine vom Benutzer definierte Zahl V_{max} wird reguliert, dass nur die V_{max} besten Cluster an die Nachbarknoten von v geschickt werden.

3. Der Knoten v aktualisiert seine Liste von Semi-Clustern mit den Semi-Clustern aus c_1, ..., c_k und $c´_1$, ..., $c´_k$, die v beinhalten.

Der Algorithmus terminiert, sobald eins der Kriterien zutrifft: Die Semi-Cluster verändern sich nicht mehr oder die maximale Anzahl an vorgegebenen Supersteps wird erreicht. Das zweite Kriterium definiert der Benutzer und legt dadurch eine obere Schranke fest. Durch das erste Kriterium soll die Performance deutlich verbessert werden [vgl. Mal u. a.10]. Wenn der Knoten v im ersten Schritt eines Supersteps keine eingehenden Nachrichten empfängt, wechselt dieser in den Zustand inaktiv [vgl. And15]. Laut Definition von Pregel terminiert ein Algorithmus, sofern sich alle Knoten gleichzeitig im inaktiv-Zustand befinden. An diesem Punkt wird eine globale Liste aus den besten Semi-Clustern aller Knoten aggregiert und stellt das Endergebnis des Algorithmus dar. Die Kennzahlen der Semi-Cluster ermöglichen eine Ordnung, die entweder aufsteigend oder absteigend ist.

Der Semi-Clustering Algorithmus selbst kann über folgende benutzerdefinierte Parameter in seiner Verarbeitung beeinflusst werden:

- f_B ist Bestandteil der Kennzahl-Ermittlung und definiert die Bedeutsamkeit der herausragenden Kantengewichte aus einem Cluster.
- V_{max} gibt an, wie viele Knoten sich maximal in einem Cluster befinden dürfen.
- C_{max} definiert die maximale Anzahl an Clustern, die zum Schluss als Ergebnis übernommen werden.

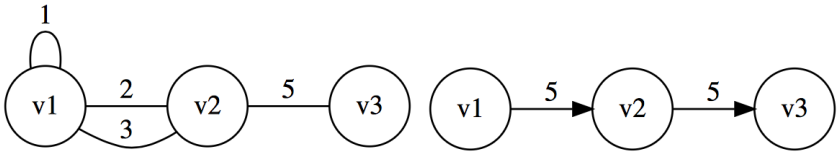
5.3 Umsetzung in Gelly

In diesem Abschnitt wird das Konzept des Semi-Clustering Algorithmus implementiert. Hierfür wird die Gelly Bibliothek und die DataSet API von Flink herangezogen. Die gesamte Graph-Abstraktion und das vertex-centric Modell sind Bestandteil der Gelly Bibliothek. Aus der DataSet API werden zusätzliche Funktionen verwendet, um unmittelbar auf *Tuple* und *DataSet* Strukturen operieren zu können.

Hierfür sind zunächst einige Vorverarbeitungsschritte (5.3.1) notwendig, bevor der Algorithmus die Semi-Cluster ermitteln kann. Anschließend folgt die Architektur (5.3.2) der Implementation in einer vereinfachten Form.

5.3.1 Vorverarbeitung

Die Eingabe ist beim Semi-Clustering ein ungerichteter Graph und zudem an den Kanten gewichtet. Eine direkte Umsetzung von ungerichteten Graphen ist nicht möglich, da Gelly im Kern auf gerichteten Graphen operiert. [Mal u. a.10] schlägt vor, eine ungerichtete Kante e durch zwei gerichtete Kanten (e_1, e_2) abzubilden. Die gerichtete Kante e_2 ist eine gespiegelte Version von e_1. Nachteilig bei diesem Vorgehen ist die Verdopplung der zu speichernden Kanten und ein resultierender Mehraufwand in der Berechnung. Im Gegensatz zu den anderen Programmiermodellen kann bei der vertex-centric Iteration nicht parametrisiert werden, in welche Richtung eine Nachricht verschickt wird. Um eine Nachricht in beide Richtungen zu senden, muss der eben genannte Vorschlag umgesetzt werden. Dies erfolgt über den Operator *getUndirected()*.



(a) unstrukturierter Eingabegraph (b) strukturierter Graph

Abbildung 5.2: Semi-Clustering: ein unstrukturierter Eingabegraph (a) und ein strukturierter Graph (b)
(Quelle: Eigene Darstellung)

Für die Strukturierung des zugrundeliegenden Graphen ist es notwendig, alle Mehrfachkanten zwischen zwei Knoten zu vereinfachen. Dieser Übergangsschritt ist in den Abbildungen 5.2a zu 5.2b veranschaulicht. Im Ursprungsgraphen verbinden zwei Kanten die Knoten *v1* und *v2* und können durch eine einzelne gerichtete Kante ersetzt werden. Das Gewicht beider Kanten wird hierfür aufsummiert und an die neue Kante geschrieben. In Gelly gibt es keine Transformation, die diese Reduzierung ausführt. Stattdessen wird dies durch eine Kombination mehrerer DataSet-Operationen umgesetzt. Zunächst sorgt eine benutzerdefinierte *map()* oder *flatMap()* Funktion dafür, dass bei jeder Kante die nummerische ID des Startknotens kleiner gleich die des Zielknotens ist. Da die Operationen auf einem Tuple ausgeführt werden, können anschließend die ersten beiden Tuple-Felder mittels *groupBy(0, 1)* gruppiert und mit *sum(2)* das dritte Feld summiert werden. Im letzten Schritt der Strukturierung sind zusätzlich die Schlingen zu vernachlässigen. Über eine benutzerdefinierte *filter()* Funktion lassen sich alle Kanten entfernen, die einen identischen Start- und Zielknoten besitzen.

5.3.2 Architektur

Aus der Beschreibung des Algorithmus lässt sich entnehmen, dass jeder Knoten über eine Liste mit Semi-Clustern verfügt. Ein einzelnes Cluster verwaltet alle enthaltenen Knoten und die Kennzahl S, die als Gewichtung dient. Anhand dieser Kennzahl lassen sich Semi-Cluster vergleichen und eine Rangfolge feststellen. Die Java-Klasse *SemiCluster* bildet einen Semi-Cluster ab und verfügt über Instanzvariablen für die Gewichtung S, die Kantengewichte, dem Faktor f_B und den Knoten. Um die Errechnung von S zu optimieren, existieren zwei Variablen, die jeweils die Kantengewichtssumme der Knoten speichern. Hierfür steht je eine Variable für innen liegende und herausragende Kanten zur Verfügung. Die Überprüfung und Anpassung erfolgt ausschließlich beim Hinzufügen eines neuen Knotens, da der Algorithmus das Entfernen eines Knotens nicht vorsieht. Sorgt ein neuer Knoten dafür, dass eine zuvor herausragende

Kante nun eine innen liegende Kante ist, wird das Kantengewicht von *weightBoundedEdges* subtrahiert und zu *weightInnerEdges* addiert.

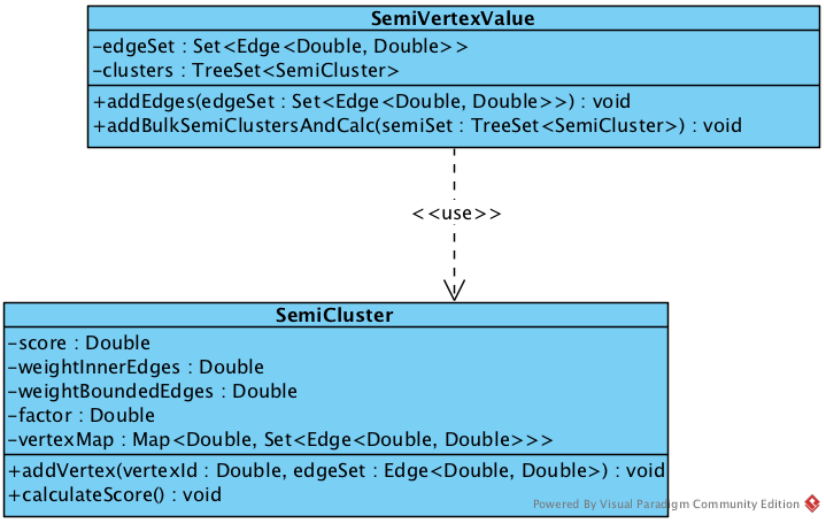


Abbildung 5.3: [Ausschnitt] Klassendiagramm vom Semi-Clustering Algorithmus *(Quelle: Eigene Darstellung)*

Die Knoten-Abstraktion von Gelly bietet keine Möglichkeit, auf die inzidenten Kanten eines spezifischen Knotenobjekts zuzugreifen. Als Lösung fungiert die *SemiVertexValue* Klasse, welche neben einer Menge aller anliegenden Kanten auch die Semi-Cluster beinhaltet. Die Erstellung der Kantenmenge erfolgt initial für jeden Knoten mittels einer *groupReduceOnEdges()* Operation. Während der Algorithmus-Durchführung kann jeder Knoten die benötigten Kanteninformationen bereitstellen.

Den Kern des Algorithmus bildet die benutzerdefinierte *SemiCompute* Funktion, die Bestandteil des vertex-centric Modells ist. Innerhalb der Funktion führen die aktiven Knoten die drei Superstep Schritte (Abschn. 5.2) aus. Die Koordination der Supersteps und die Zustände der Knoten übernimmt die Modellabstraktion. Beim Semi-Clustering Algorithmus wird die *setNewVertexValue()* Methode benötigt, um die angepassten Semi-Cluster in die *SemiVertexValue* Klasse zu überführen. Am Ende der *SemiCompute* Funktion werden die besten Semi-Cluster mit Hilfe der *sendMessageToAllNeighbors()* Methode an die Nachbarknoten gesendet. Die Anzahl der besten Cluster ist vom Anwender definierbar.

Nachdem der Algorithmus terminiert ist, werden alle Semi-Cluster aggregiert und stellen das Endergebnis des Semi-Clusterings dar.

6 Experimente

In diesem Kapitel wird die Performance der beiden Frameworks Apache Flink und Apache Spark in Form von Experimenten getestet. Der Fokus liegt dabei auf die End-zu-End-Laufzeit eines Jobs. Im Folgenden werden zuerst der Aufbau (6.1) und die Ergebnisse der Experimente (6.2) präsentiert. Am Ende des Kapitels wird auf die Korrektheit der Experimente (6.3) und auf die gewonnenen Erkenntnisse (6.4) eingegangen.

6.1 Aufbau der Experimente

Mit Hilfe der Experimente soll aufgezeigt werden, wo die Stärken und Schwächen beider Frameworks im Graphenumfeld liegen. Hierfür werden zunächst Hypothesen (6.1.1) aufgestellt, die sowohl allgemeine, als auch spezielle Disziplinen abdecken. Das zugrundeliegende Cluster (6.1.2) und die angewendeten Daten (6.1.3) sollen dabei im Rahmen dieser Arbeit einen Praxisbezug haben.

6.1.1 Hypothesen

Die Experimente umfassen fünf Hypothesen, um Apache Flink mit Apache Spark in Bezug auf die Performance zu vergleichen. Die fünf Hypothesen sind:

1. Vertex-centric Algorithmen sind unter Apache Spark und Apache Flink gleich schnell.
2. Apache Flink und Apache Spark brauchen für die x-fache Datenmenge, eine x-mal so lange Zeit für die Verarbeitung (lineares Wachstum).
3. Apache Spark ist schneller als Apache Flink, wenn das Cluster horizontal skaliert wird.
4. Apache Flink hat einen geringeren Arbeitsspeicher-Verbrauch als Apache Spark.
5. Apache Flink ist bei power-law Graphen schneller als Apache Spark.

Begründung für die Wahl der Hypothesen

Sowohl Spark als auch Flink sind moderne Frameworks zur Datenverarbeitung. Jedoch wurde Spark seit Entwicklungsbeginn auf die Batch-Verarbeitung optimiert und besitzt mit der RDD-Abstraktion eine gute Voraussetzung für ein performantes und verteiltes Verarbeiten. Dem gegenüber steht Flink mit seiner nativen iterativen Verarbeitungsmöglichkeit, die bei der Graphverarbeitung zum Tragen kommt. Im Experiment müsste daher eine vergleichbar schnelle Laufzeit beider Frameworks sichtbar werden und ist unter Hypothese 1 festgehalten.

Bei einer Verdopplung der Kanten im Eingabegraphen, lässt sich eine proportionale Laufzeitentwicklung vermuten, da die Ressourcen der Frameworks unverändert bleiben. In der 2. Hypothese wird die Annahme widergespiegelt, dass sich die Ausführungszeit linear zur Eingabemenge verhält.

Spark verwendet zur Partitionierung eines Graphen den vertex-cut. Durch diese Eigenschaft wird ein Eingabegraph mit geringem Overhead auf mehrere Worker-Knoten partitioniert und adjazente Knoten befinden sich größtenteils in der gleichen Komponente. Der vertex-cut ist bis zu 70% besser als eine zufällige Partitionierung [vgl. Rah u. a.14], wie sie bei Flink vorzufinden ist. Die 3. Hypothese greift diese Aussage auf.

Beide Frameworks führen ihre Jobs in der Java VM aus. Im Gegensatz zu Spark verwaltet Flink seinen Speicher aktiv selbst und dies lässt vermuten (Hypothese 4), dass der Arbeitsspeicherverbrauch geringer ausfällt als bei Spark.

Flink verfügt über mehrere Modelle zur Graphverarbeitung. Verglichen mit dem vertex-centric Modell ist das GSA Modell deutlich eingeschränkter. Jedoch wurde dieses ursprünglich speziell für power-law Graphen konzipiert und ist in Flink implementiert. Die 5. Hypothese bezieht sich auf diesen theoretischen Vorteil gegenüber der Standalone Implementierung in Spark.

6.1.2 Cluster

Die Experimente werden in einem Cluster aus 4 dedizierten Maschinen ausgeführt. Die Frameworks befinden sich ohne Virtualisierung direkt auf dem Betriebssystem Ubuntu 16.04 LTS (64-Bit) und verwenden Java in der Version 8 von Oracle. Apache Flink wird in der Release Nummer 1.3.2 und Apache Spark in Version 2.2.0 herangezogen. Zum Zeitpunkt der Erstellung dieser Arbeit stellten diese die neusten stabilen Veröffentlichungen dar.

Innerhalb des Clusters kommt der *TICK* Stack [vgl. Inf17] für das Monitoring der System-Ressourcen zum Einsatz. Über den Service (*Telegraf*) werden die Metriken ausgelesen und in der *InfluxDB* persistiert. *Chronograf* benutzt diese Daten zum Visualisieren und stellt eine

explorative Durchsuchung bereit. Auf die Integration vom *Kapacitor* wird in diesem Szenario verzichtet. Dieser ermöglicht eine Alarmierung, sobald ein definierter Schwellenwert über- bzw. unterschritten wird.

Um äußere Einflüsse im Netzwerkverkehr zu minimieren, befindet sich das Cluster in einem eigenen Subnetzwerk. Die einzelnen Rechner sind via Gigabit-LAN verbunden. Ein Rechner ist als Master-Knoten definiert und die restlichen drei fungieren als Worker-Knoten. Die Bezeichnung Worker wird im weiteren Verlauf als Synonym für einen TaskManager von Flink bzw. für einen Worker von Spark verwendet. Die Rechner haben folgende Ressourcen-Konfiguration:

- Master-Knoten mit 32 Gigabyte RAM, 8 Prozessorkernen (Quad-Core mit Hyper-Threading) und 2 Terabyte Festplattenspeicher
- Worker-Knoten mit jeweils 32 Gigabyte RAM, 8 Prozessorkernen (Quad-Core mit Hyper-Threading) und 2 Terabyte Festplattenspeicher

Der Arbeitsspeicher wird im Rahmen der Experimente auf 8 GB pro Endgerät beschränkt. Dadurch lassen sich unter anderem beide Frameworks mit weniger großen Graphen an mögliche Ressourcengrenzen bringen. Zum anderen wird sichergestellt, dass den Frameworks zu jedem Zeitpunkt die 8 GB Arbeitsspeicher zur Verfügung stehen.

6.1.3 Daten

Die Datengrundlage bildet eine Vielzahl unterschiedlicher Eingabegraphen, die allesamt als Textdatei vorliegen. Neben selbst generierten Graphen, werden zusätzlich weitere aus der *Stanford Large Network Dataset Collection* [vgl. LK14] verwendet. Die Dateigröße der einzelnen Dateien variiert zwischen 2 Kilobyte bis 7,95 Gigabyte und ist zeilenweise abgespeichert. Eine Zeile entspricht hierbei genau einer gerichteten Kante mit dem Start- und Zielknoten und ggf. der Kantengewichtung. Diese Repräsentation nennt sich *edge-list* und die Knoten werden von den Frameworks automatisch aus den vorliegenden Kanteninformationen abgeleitet.

Für die Experimente werden die nachfolgenden Daten verwendet:

- Die Datensätze *edges-2048* (34 Kilobyte), *edges-3276800* (66 Megabyte) und *edges-327680000* (7,95 Gigabyte) sind künstlich generiert. Der kleinste Graph hat 2048 Kanten und 569 Knoten. Der mittlere verfügt über 3.276.000 Kanten, die 55.291 Knoten verbinden. Im größten Graphen sind 327.680.000 Kanten und 4.699.071 Knoten. Alle Kanten der drei Datensätze haben eine zufällig erzeugte Gewichtung.

- Aus der *SNAP* Datenbank kommen folgende Graphen:
  - *com-orkut* (1,7 Gigabyte) mit 117.185.083 Kanten und 3.072.441 Knoten ist ein Datensatz aus einem sozialen Netzwerk.
  - Der Datensatz *wiki-topcats* (422 Megabyte) mit 28.511.807 Kanten und 1.791.489 Knoten stammt von Wikipedia (2011).
  - *soc-sinaweibo* (4,05 Gigabyte) mit 261.321.072 Kanten und 58.655.849 Knoten ist ein Datensatz eines chinesischen Online-Dienstes.
- Zwei weitere Datenreihen sind geniert und haben keine Kantengewichtung. Die Graphen wachsen jeweils um den Faktor 10 (erste Reihe) bzw. Faktor 2 (zweite Reihe) an. Die beiden Reihen beinhalten:
  - *edges300*, *edges3000*, *edges30000*, *edges300000* und *edges3000000*. Die Zahl im Bezeichner entspricht der Kantenanzahl im Graph. Der erste Datensatz hat 63 Knoten und der letzte 510.065 Knoten.
  - *edges900000*, *edges1800000*, *edges3600000* und *edges7200000*. Die Kantenanzahl ist analog zur Datensatzbezeichnung. Der kleinste Datensatz hat 245.935 Knoten und der größte 1.936.705 Knoten.

Begründung für die Wahl der Daten

Die Daten aus der Stanford Sammlung stammen aus real existierenden Systemen wie sozialen Netzwerken aus aller Welt und stellen eine authentische Basis dar. Zudem sind die künstlichen Daten in unterschiedlichen Ausmaßen vorhanden und lassen sich somit auf Geschwindigkeit testen. Des Weiteren können mögliche Unterschiede zwischen künstlich generierten und realen Daten festgestellt werden.

6.1.4 Algorithmen

Für die Experimente kommen folgende Graph-Algorithmen zum Einsatz:

- Das **Semi-Clustering** bildet Cluster basierend auf der Kommunikationsintensität zwischen zwei Personen. Mit Hilfe einer Kantengewichtung ist der Algorithmus in der Lage, Personengruppen zu identifizieren, die untereinander eine hohe Kommunikation aufweisen.

- Das **PageRank** Verfahren bewertet Webseiten auf Basis der eingehenden Verlinkungen, die durch gerichtete Kanten repräsentiert werden. Das Prinzip hinter dem Algorithmus lautet: „je mehr Links, desto wichtiger die Seite“ und wurde von den Google Gründern an der Stanford Universität entwickelt. Eine Seite mit vielen eingehenden Links wird grundsätzlich höher bewertet, als eine Seite, auf die nur wenige andere Seiten verweisen.

Unter Benutzung des vertex-centric Modells wurde im Rahmen dieser Bachelorthesis der Semi-Clustering Algorithmus für Flink implementiert. Details zur Architektur und Umsetzung sind im Kapitel 5 zu finden. Die Implementierung für Spark stammt aus der Arbeit von Herrn [And15].

Das weitverbreitete PageRank Verfahren ist in beiden Frameworks in mindestens einer Implementierung vorhanden. Die Frameworks verwenden allerdings verschiedene Formeln, um den PageRank eines Knotens zu bestimmen. Bei Spark ist der Algorithmus basierend auf der ursprünglichen Definition umgesetzt worden. In dieser Definition entspricht die Summe aller PageRanks gleich der Knotenanzahl im Graphen. Flink wendet eine aktualisierte Definition an, die ebenfalls von den Google Gründern beschrieben wird [vgl. BP98]. Im Gegensatz zur ersten Formel sind die PageRanks normalisiert und bilden eine diskrete Wahrscheinlichkeitsverteilung über alle Knoten.

Begründung für die Wahl der Algorithmen

Die beschriebenen Algorithmen sind für die Experimente ausgewählt, da diese allgemeine Anwendungsfälle darstellen und eine gute Vergleichbarkeit der Ergebnisse erwartet wird. Flink stellt den PageRank in vier verschiedenen Varianten bereit und ist zugleich in Spark vorhanden. Neben den drei bekannten Modellen (vertex-centric, scatter-gather, gather-sum-apply) gibt es eine weitere Variante, die ohne Verwendung einer Modell-Abstraktion auskommt. In dieser wird nicht auf den Delta-Iterator, sondern auf den Bulk-Iterator zurückgegriffen und das Graph-Interface von Gelly benutzt. Bei Spark wird PageRank in der Standalone Implementierung verwendet, da nur diese eine benutzerdefinierte Konvergenzgrenze erlaubt. Der Algorithmus bedient sich hierfür an dem Graph-Interface der GraphX Bibliothek.

Der Semi-Clustering Algorithmus repräsentiert die Klasse der Clustering-Verfahren und liegt als vertex-centric Implementation vor.

6.2 Experimentelle Durchführung

Die Experimente werden anhand der fünf Hypothesen mit unterschiedlichen Daten durchgeführt. Für jede Hypothese werden die Daten genannt, die für das jeweilige Experiment zum Einsatz kommen. Nachfolgend sind die Ergebnisse der jeweiligen Hypothese vorgestellt. Alle benutzten Dateien sind für die Experimente auf den Workern lokal gespeichert und werden von dort aus vom jeweiligem Framework verarbeitet. Zur Verwaltung des Clusters benutzen beide Frameworks dessen Standalone Modus. Die Standardkonfigurationen der Frameworks werden größtenteils beibehalten und nur den Ressourcen im Cluster angepasst. Die Änderungen sind in der Tabelle 6.1 aufgeführt. Die Parallelität eines Jobs orientiert sich bei Flink an der Anzahl verfügbarer CPU Kerne. Spark verwaltet seine Ressourcen selbstständig und dynamisch, abhängig von der Rechenlast.

Apache Flink	
taskmanager.memory.off-heap	*true*
taskmanager.memory.preallocate	*true*
taskmanager.heap.mb	*8192*

Apache Spark	
spark.executor.cores	*8*
spark.executor.memory	*8g*

Tabelle 6.1: angepasste Parameter für Apache Flink und Apache Spark

Um eine Vergleichbarkeit zu erreichen, sind die Parameter der Algorithmen durchgehend identisch. Beim PageRank beträgt der Dämpfungsfaktor d gleich 0.85 und die Iterationskonvergenz liegt bei maximal zehn Durchläufen. Beim Semi-Clustering sind die Parameter wie folgt festgelegt: Der Berechnungsfaktor f_B ist 0.5, jedes Semi-Cluster hält höchstens fünf Knoten, es werden nur die drei besten Semi-Cluster übermittelt und der Algorithmus iteriert maximal zehn Mal über den Graphen.

Jede Messung wird drei Mal durchgeführt und die Ergebnisse arithmetisch gemittelt. Sowohl in den Ergebnistabellen, als auch in den Grafiken wird dieser gemittelte Werte angegeben. Um die Lesbarkeit zu verbessern, sind in den folgenden Hypothesen die Zahlen mit einem Punkt (bspw. 1.000) gegliedert.

6.2.1 Hypothese 1

„Vertex-centric Algorithmen sind unter Apache Spark und Apache Flink gleich schnell."

Für das Testen der ersten Hypothese eignet sich das Semi-Clustering Verfahren, da die vorliegenden Implementierungen das vertex-centric Modell anwenden. Um einen ausgewogenen Vergleich zu erzielen, bieten sich große und kleine Datensätze an, um die Laufzeiten bei unterschiedlichen Datengrößen zu messen.

Der große Datensatz *edges-3276800* verfügt über 3,2 Millionen Kanten. Als Gegenstück weist der Datensatz *edges-2048* knapp über 2.000 Kanten auf. Bei diesem Experiment werden alle drei Worker-Knoten eingebunden.

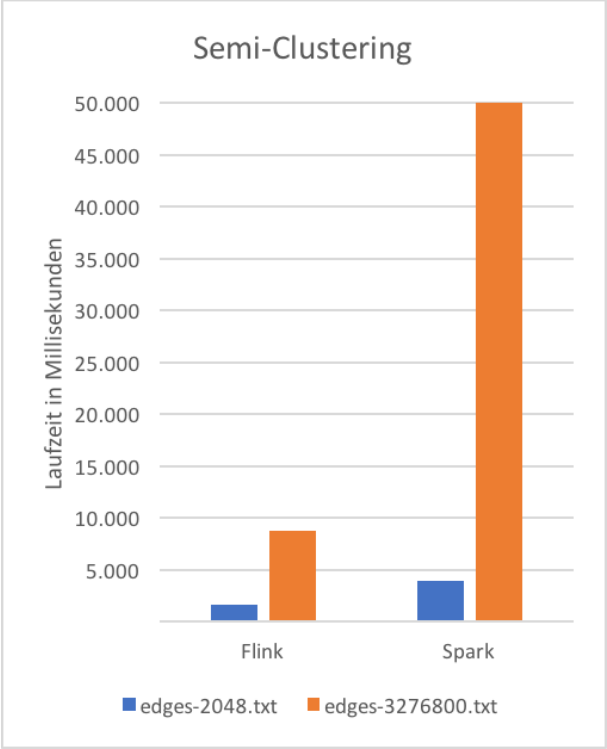


Abbildung 6.1: Hypothese 1 - Visualisierung der Laufzeitmessungen beim Semi-Clustering Algorithmus (vertex-centric)

In der Abbildung 6.1 sind die Laufzeiten für das Semi-Clustering visuell veranschaulicht. Die Zeitangabe an der y-Achse entspricht der Laufzeit des gesamten Jobs in Millisekunden. Die x-Achse unterteilt sich in Flink und Spark. Pro Framework sind zwei farbige Balken abgebildet, die jeweils einen Datensatz darstellen.

Entgegen der Annahme, dass beide Frameworks vergleichbar schnell sind, zeigen die Messergebnisse deutliche Unterschiede. Beim kleinen Graphen erzielt Flink eine bis zu 2,3-fache kürzere Berechnungslaufzeit als Spark. Der große Graph weist einen noch höheren Kontrast auf. Spark hat hier eine bis zu 5,9-fache längere Laufzeit gegenüber Flink. Die gemittelten Messwerte lassen sich aus der Tabelle 6.2 entnehmen. Die erste Hypothese konnte somit nicht verifiziert werden.

	Apache Flink	**Apache Spark**
kleiner Graph	1.681	3.910
großer Graph	8.773	51.074

Tabelle 6.2: arithmetisch gemittelte Laufzeiten in Millisekunden für Semi-Clustering beim kleinen und großen Eingabegraphen

In beiden Implementierungen wird das vertex-centric Modell angewendet und beide Graph Bibliotheken haben keine Abstraktion integriert, um von einem Knoten direkt auf seine ein- und ausgehenden Kanten zu schließen. Um dies zu kompensieren, muss eine Erweiterung der Knotenabstraktion erfolgen. Dadurch können in einem initialen Vorverarbeitungsschritt die notwendigen Kantendaten als Datenobjekt im Knotenwert gespeichert werden. In den genannten Aspekten unterscheiden sich beide Implementierungen nicht voneinander. Die Laufzeitdifferenzen lassen sich vermutlich auf einzelne Operationen zurückführen, die Bestandteil der Semi-Clustering Implementierung sind. Der native Delta-Iterator von Flink bringt zusätzlich Performancevorteile und überwiegt die theoretischen Vorteile der RDD-Abstraktion von Spark.

6.2.2 Hypothese 2

„Apache Flink und Apache Spark brauchen für die x-fache Datenmenge, eine x-mal so lange Zeit für die Verarbeitung (lineares Wachstum).“

Bei der zweiten Hypothese werden alle fünf Varianten des PageRank Algorithmus angewendet: Vier von Flink und einer von Spark. Für den PageRank Algorithmus werden zwei unterschiedlich skalierende Datensätze in zwei Messreihen verwendet, um verschiedene Ausmaße der Graphen zu evaluieren. Dadurch kann das Verhalten der Frameworks bei kleinen und großen Graphen verglichen werden.

Die erste Messreihe beginnt mit einem kleinen Graphen, der 300 Kanten (*edges300*) besitzt. Die vier weiteren Graphen in dieser Messreihe vergrößern sich jeweils um den Faktor 10.

Zwischen dem ersten und dem letzten Graphen, besteht eine 10.000-fache Skalierung in Bezug auf die Kanten- und Knotenanzahl, sowie der Dateigröße.

In einer zweiten Messreihe verfügt der erste Graph (*edges900000*) über 900.000 Kanten und ungefähr 250.000 Knoten. Die drei anderen Graphen sind in ihrer Knoten- und Kantenanzahl immer verdoppelt und erreichen bis zu 7,2 Millionen Kanten bei 2 Millionen Knoten.

Bei diesem Experiment werden alle drei Worker benutzt. Zur Visualisierung der Abbildungen 6.2 und 6.3 wurden die arithmetischen Mittel aus drei Messungen pro PageRank Implementierung und Eingabegraph gebildet.

Messreihe	**Datei**	**Flink**				**Spark**
		VC	SG	GSA	Bulk-Iter.	Standalone
1	*edges300*	0,90	0,40	0,81	0,45	4,05
	edges3000	0,88	0,51	0,92	0,67	4,57
	edges30000	0,95	0,49	0,76	0,73	4,56
	edges300000	1,80	1,08	1,17	1,25	6,12
	edges3000000	7,82	5,35	6,50	5,49	12,81
2	*edges900000*	2,46	1,67	1,92	2,18	8,13
	edges1800000	5,67	4,01	4,11	2,56	11,17
	edges3600000	8,89	6,31	7,77	6,78	16,44
	edges7200000	16,05	10,58	13,60	11,82	23,86

Tabelle 6.3: Das arithmetische Mittel aus drei Messwerten (in Sekunden) für den PageRank bei skalierenden Eingabegraphen. Die Zahl im Dateinamen entspricht der Kantenanzahl.

Die erste Messreihe beginnt mit einem sehr kleinen Graphen und erreicht bei den verschiedenen Implementierungen folgende Laufzeiten: 0,9 Sek. (Flink VC); 0,4 Sek. (Flink SG); 0,8 Sek. (Flink GSA); 0,5 Sek. (Flink Bulk-Iterator) und 4 Sek. (Spark). Aufgrund der Hypothese lassen sich ungefähre Laufzeiten zwischen einer und elf Stunden erwarten und spiegeln eine 10.000-fache Kantenskalierung wider. Die tatsächlich gemessenen Laufzeiten weisen in der Praxis eine immense Differenz auf. Der zeitliche Aufwand beim größten Graphen übersteigt bei keinem der fünf Varianten die 13 Sekundengrenze. Bei Flink tritt der größte Faktor bei der scatter-gather Variante auf und beträgt hier eine 13,3-fach längere Laufzeit. Spark weist das geringste Wachstum auf, mit einem Faktor von 3,1. Insgesamt betrachtet ist die scatter-gather Umsetzung die schnellste der fünf gemessenen Varianten und liegt kurz vor dem Bulk-Iterator. Zudem fällt auf, dass alle PageRank Implementierungen bei den Graphen mit 300, 3.000 und 30.000 Kanten jeweils nur marginal mehr Berechnungszeit benötigen. Erst beim Sprung von

300.000 auf 3 Millionen Kanten verdoppelt sich die Laufzeit bei Spark und auch Flink zeigt hier einen deutlichen Laufzeitsprung, der beim GSA Modell um das 5,5-fache ansteigt.

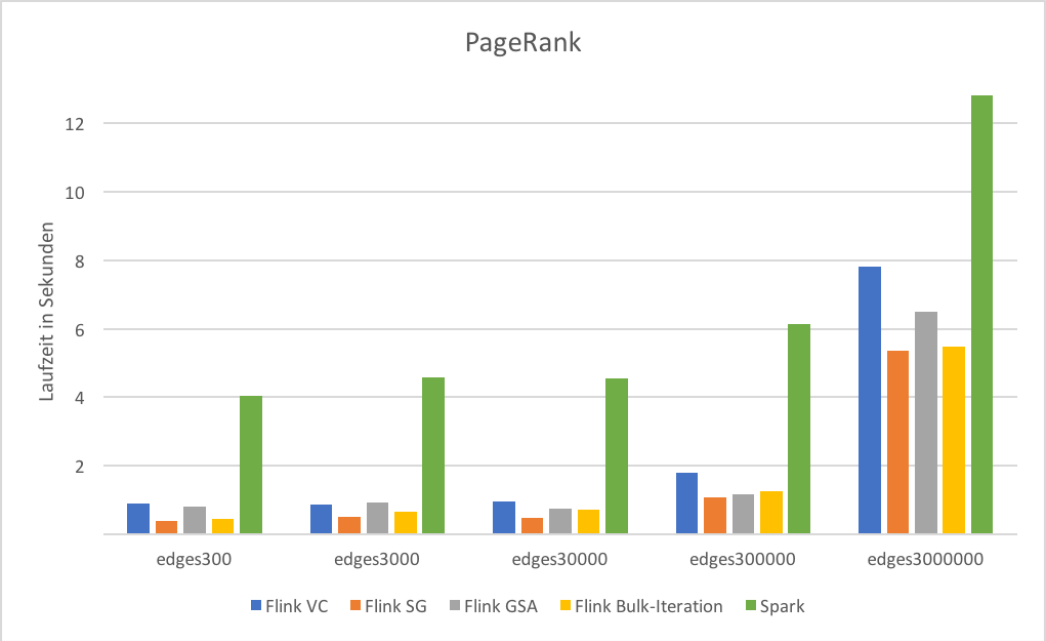


Abbildung 6.2: Hypothese 2 - Visualisierung der Laufzeitmessungen (in Sekunden) beim PageRank Algorithmus. VC steht für vertex-centric, SG für scatter-gather, und GSA für gather-sum-apply. Die Kantenanzahl skaliert jeweils um den Faktor 10.

Den Ausgangspunkt bildet in der zweiten Messreihe ein Graph mit 900.000 Kanten und vermisst folgende Laufzeiten: 2,5 Sek. (Flink VC); 1,7 Sek. (Flink SG); 1,9 Sek. (Flink GSA); 2,2 Sek. (Flink Bulk-Iterator) und 8,1 Sek. (Spark). In diesem Szenario erfolgt immer eine Verdopplung der Kanten- und Knotenanzahl.

Im Gegensatz zur vorherigen Messreihe, ist bei dieser Reihe größtenteils ein lineares Wachstum vorhanden. Die drei Modell-Varianten von Flink verdoppeln die Laufzeit zwischen dem ersten und dem zweiten Graphen. Bei Spark ist bei diesen Graphen lediglich ein Wachstum um 38% zu messen und der Bulk-Iterator hat einen Zuwachs von 18%. Ein durchgängig lineares Wachstum liegt beim vorletzten (*edges900000*) zum letzten Graphen (*edges7200000*) bei Flink vor. Analog zur ersten Messreihe, wächst Spark beim PageRank Algorithmus weniger schnell an. Der Zuwachs beträgt 37%, 47% und 45% bei einer jeweiligen Verdopplung des Graphen.

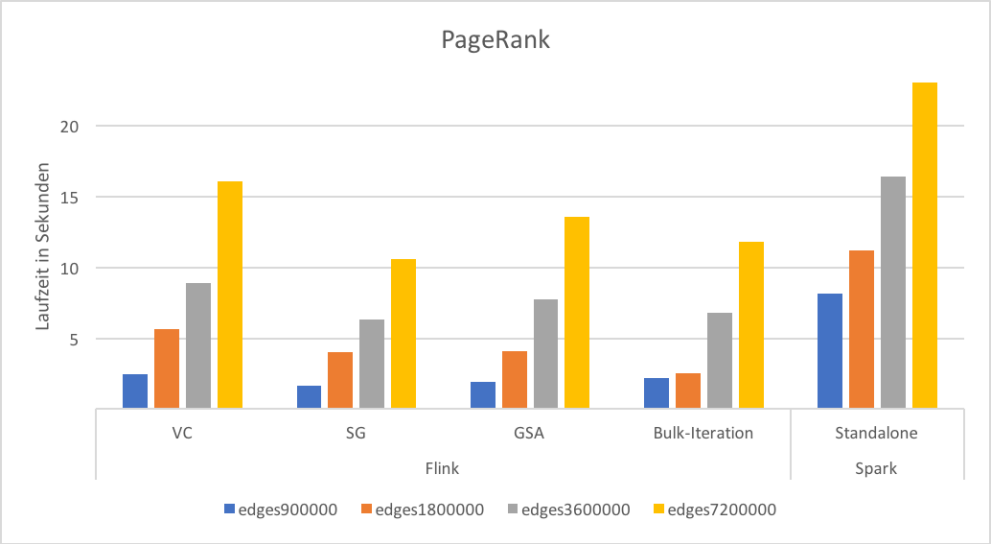


Abbildung 6.3: Hypothese 2 - Visualisierung der Laufzeitmessungen (in Sekunden) beim PageRank Algorithmus. Die Kantenanzahl skaliert jeweils um den Faktor 2.

Die Hypothese eines linearen Wachstums trifft bei der ersten Messreihe mit kleineren Eingabegraphen nicht zu. Die schlechteste Laufzeitskalierung weist Flink bei der scatter-gather Variante auf. In diesem Fall benötigt der Algorithmus die 13,3-fache Laufzeit bei einem Kantenwachstum um den Faktor 10.000. Beim Vergleichen der beiden Messreihen fällt auf, dass bei Flink ein lineares Wachstum erst bei größeren Graphen auftritt. Spark hat bei beiden Messreihen die längsten Laufzeiten bei gleichzeitig geringstem Wachstum.

6.2.3 Hypothese 3

„Apache Spark ist schneller als Apache Flink, wenn das Cluster horizontal skaliert wird."

Bei einer horizontalen Skalierung des Clusters (*scale-out*) lässt sich annehmen, dass die Laufzeit verbessert wird und gleichzeitig größere Datenmengen zwischen den Workern ausgetauscht werden. Die dritte Hypothese wird anhand der beiden Algorithmen (Semi-Clustering und PageRank) experimentell geprüft. Als Eingabe dienen zwei Graphen, die bei jedem Durchlauf mit unterschiedlicher Workeranzahl identisch bleiben. Der erste Graph (*com-orkut*) kommt beim PageRank zum Einsatz und verfügt über 117,1 Millionen Kanten. Der zweite Graph (*edges3276800*) ist für den Semi-Clustering Algorithmus und hat 3,2 Millionen Kanten.

In der Abbildung 6.4 sind die Laufzeiten bei der Skalierung des Clusters visualisiert. Jeder Algorithmus wird durch drei nebeneinander liegende Balken präsentiert. Die blauen Balken

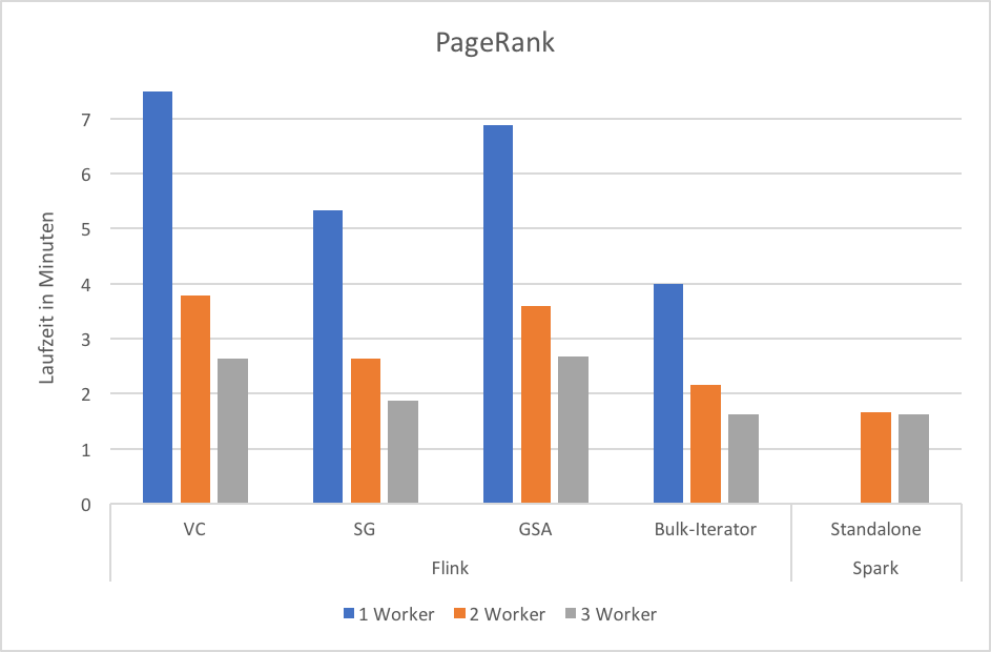


Abbildung 6.4: Hypothese 3 - Visualisierung der Laufzeit (in Minuten) beim scale-out des Clusters für den PageRank Algorithmus

entsprechen den Laufzeiten in der Konstellation: Ein Master und ein Worker. Die anderen beiden Balken zeigen die jeweiligen Laufzeiten bei zwei bzw. drei Workern. Bei Spark wurde der Algorithmus nach 20 Minuten manuell abgebrochen, als diesem nur ein Worker zur Verfügung stand. Das Framework war dort nicht in der Lage, den PageRank Algorithmus erfolgreich zu beenden. Während dieser Zeit las die ausführende Executor-Einheit über 200 Gigabyte ein. Zeitgleich versuchte der Java Garbage Collector vermeintlich alte RDD Partitionen zu entfernen und sorgte vermutlich für eine interne Schleife. In diesem *Deadlock* wurde der Executor permanent von Master durch einen neuen Executor ausgetauscht. Spark führte bei zwei bzw. drei parallelen Workern den Job in unter zwei Minuten durch.

Im Gegensatz zu Spark ist eine deutliche Verbesserung der Laufzeit erkennbar, wenn Flink um zusätzliche Worker erweitert wird. Insbesondere der Sprung von einem auf zwei Worker bedeutet bei Flink eine ungefähr 50%ige Laufzeitreduzierung bei allen vier Implementierungen. Durch das Hinzufügen eines weiteren Workers konnte die Laufzeit erneut um 30% gesenkt werden.

Beim Betrachten des Netzwerkverkehrs fällt auf, dass Spark insgesamt circa 3 Gigabyte Daten zwischen den einzelnen drei Workern und dem Master übermittelt. Bei Flink variiert

dieser Wert zwischen 6 Gigabyte beim Bulk-Iterator und 13,5 Gigabyte beim GSA Modell. Das deutlich erhöhte Datenaufkommen ist unter anderem auf die weniger optimale Partitionierung seitens Flink zurückzuführen. Im Gegensatz zu Spark wendet Flink nicht den vertex-cut an, um den Eingabegraphen auf den Workern zu verteilen. Flink nutzt eine Hash-Funktion und dadurch befinden sich benötigte Zwischenergebnisse häufiger auf unterschiedlichen Workern. Die Datenübertragung über das Netzwerk stellt aufgrund der geringeren Übertragungsgeschwindigkeit einen Flaschenhals dar, um erforderliche Daten zwischen den Workern zu transferieren.

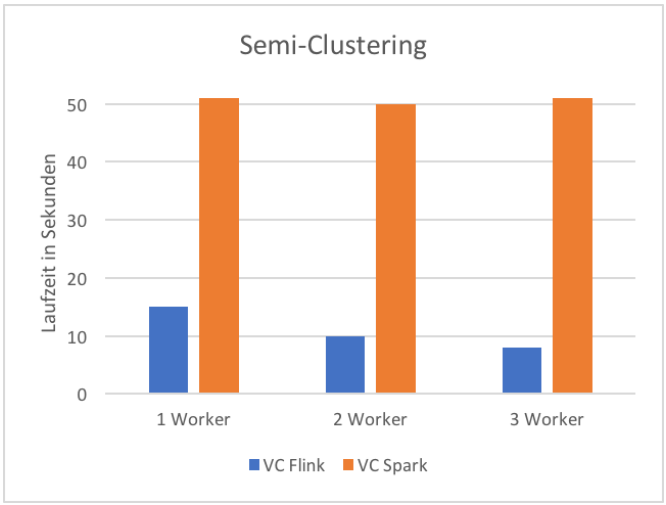


Abbildung 6.5: Hypothese 3 - Visualisierung der Laufzeit beim scale-out des Clusters für den Semi-Clustering Algorithmus

Wie in der Abbildung 6.5 zu erkennen, verhält sich der Semi-Clustering Algorithmus beim Skalieren vergleichbar zum PageRank Algorithmus. Bei Spark ist keine große Verkürzung der Laufzeit zu messen. Flink zieht die größten Vorteile durch das Erweitern von einem auf zwei Worker. Ein markanter Unterschied ist, dass Flink deutlich kürzere Laufzeiten aufweist, als es bei Spark der Fall ist. Dies konnte bereits in der ersten Hypothese festgestellt werden. Bei der Durchführung mit einem größeren Graphen (327 Millionen Kanten) verursachten beide Frameworks eine *out-of-memory Exception*. Dieser Fehler kann sowohl in der Konfiguration mit einem, als auch mit drei Workern festgestellt werden.

Unter der Prämisse, dass beide Frameworks mit einem Worker eine identische Laufzeit haben, ist die dritte Hypothese nicht zutreffend. Durch das scale-out auf drei Worker konnte die Laufzeit von Flink um 60% verringert werden. Bei Spark fällt der Gewinn zwischen zwei und drei Workern mit 2% deutlich schwächer aus. Im Gegensatz zu Flink sind bei Spark nicht immer zwangsläufig alle Worker parallel aktiv. Abhängig von den benötigten Ressourcen für eine auszuführende Aufgabe, werden weniger oder mehr Worker zugeteilt. Dieses Verhalten könnte eine mögliche Ursache sein, warum Spark kaum Unterschiede beim Semi-Clustering und PageRank aufzeigt.

6.2.4 Hypothese 4

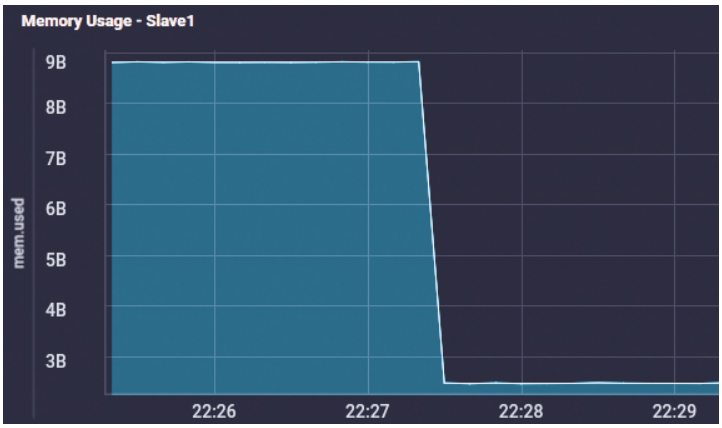
„Apache Flink hat einen geringeren Arbeitsspeicher-Verbrauch als Apache Spark."

Als Eingabemenge bietet sich für die vierte Hypothese ein sehr großer Graph (*soc-sinaweibo*) an, um den Arbeitsspeicherverbrauch möglichst stark ansteigen zu lassen.

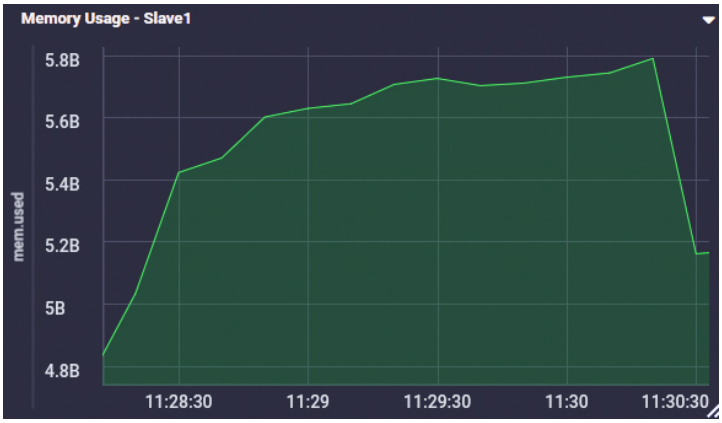
Die Abbildungen 6.6 zeigen auf der x-Achse die Zeitangaben (*hh:mm*) und auf der y-Achse den Speicherverbrauch in Gigabyte. Der Worker-Knoten „Slave1" dient hierbei repräsentativ für die einzelnen Worker.

Im Rahmen des experimentellen Umfeldes konnte allerdings keine Verbrauchsevaluierung unter Flink durchgeführt werden, da das Framework diesen aktiv selbst verwaltet. Der Monitoring Stack erlaubt zudem keine Überwachung einzelner Prozesse, sondern nur den gesamten Verbrauch aller Prozesse.

Im Gegensatz zu Flink lässt sich bei Spark ein dynamischer Verbrauch erkennen (Abb. 6.6b). Bei Flink beansprucht die Runtime zur Ausführungszeit den gesamten Speicher, der ihr zugewiesen ist und gibt diesen nach Beendigung wieder frei. Dieser Verlauf ist in der Abbildung 6.6a visualisiert und zeigt die Speicherfreigabe ungefähr zum Zeitpunkt 22:27:20. Der Flink Job benötigt hierbei nicht durchgehend den gesamten Speicher, sondern bekommt die notwendigen Ressourcen von der Flink Runtime zugeteilt.



(a) Flink



(b) Spark

Abbildung 6.6: Speicherverbrauch (in Gigabyte) von Flink und Spark während und nach einer Job-Ausführung

6.2.5 Hypothese 5

„Apache Flink ist bei power-law Graphen schneller als Apache Spark.“

Die Charakteristik eines power-law Graphen ist vor allem in sozialen Graphen vorzufinden. Für die fünfte Hypothese wird der PageRank Algorithmus angewendet, da dieser bei Flink in einer Implementierung vorhanden ist, die mit Hilfe des GSA Modells umgesetzt ist. Das GSA Modell findet primär Anwendung bei power-law Graphen. Der erste von den zwei Eingabegraphen bildet Relationen aus einem real existierenden Netzwerk - der Wikipedia - ab. Der zweite Graph wurde künstlich mit einem Graphgenerator aus Spark generiert. Für eine neutrale Evaluation wurde dieser anschließend als Datei exportiert und bietet keinen Vorteil für Spark.

Flink ist mit 7 Sek. beim künstlich generierten Graphen (*edges3600000*) mehr als doppelt so schnell wie Spark mit 16 Sek. Beim *wiki-topcats* Datensatz geben die beiden Algorithmen ein umgekehrtes Ergebnis ab. In diesem Fall liegt Spark vor dem GSA Modell von Flink mit einer Laufzeitdifferenz von 14 Sek. Entgegen der Theorie erreicht das GSA Modell bei größeren Graphen eine schlechtere Laufzeit als Spark. Beim Anwenden der Bulk-Iterator Implementierung zeigt sich zudem, dass das GSA Modell auch hier langsamer ist. Der Bulk-Iterator hat beim *wiki-topcats* Graphen ebenfalls eine Laufzeit von 24 Sek. und ist beim generierten Graphen unwesentlich schneller als das GSA Modell.

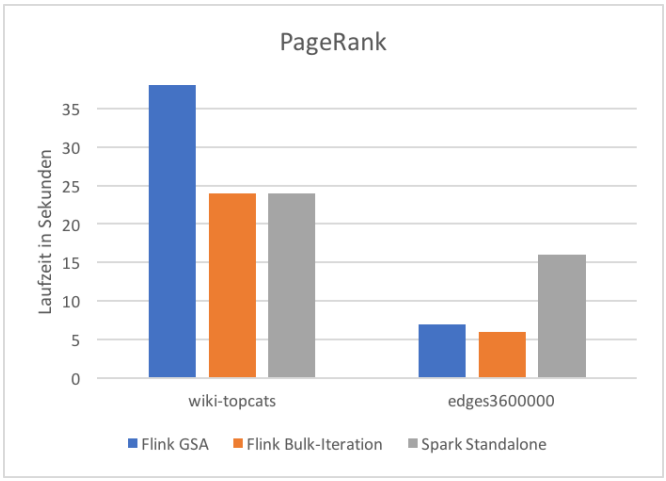


Abbildung 6.7: Hypothese 5 - Visualisierung der Laufzeit beim PageRank Algorithmus (Flink GSA, Flink Bulk-Iteration und Spark Standalone)

Das Experiment zeigt, dass die fünfte Hypothese nicht immer gültig ist. Beim real existierenden Datensatz (*wiki-topcats*) konnte Flink keine kürzeren Laufzeiten erreichen als Spark. Das spezielle GSA Modell ist in diesem Fall deutlich langsamer als die anderen beiden Implementierungen. Beim kleineren Graphen (*edges3600000*) sind die Flink Implementierungen sichtbar schneller und auch hier ist das GSA Modell langsamer als der Bulk-Iterator.

6.3 Korrektheit der Experimente

Es ist zu beachten, dass die Laufzeitmesswerte im Falle von Flink vom Framework selbst ermittelt wurden und bei Spark diente die Systemuhrzeit zur Erfassung des Start- und Endzeitpunkts. Bei Stichproben stimmten die Laufzeiten beider Frameworks mit einer manuellen Zeitmessung und den Log-Daten überein.

Bei den Flink Implementierungen des PageRank Verfahrens sind die Laufzeiten von unter einer bzw. knapp über einer Sekunde mit Vorsicht zu betrachten. Blockieren andere Prozesse die CPUs zur Ausführungszeit, kann die Gesamtlaufzeit des Jobs stark variieren. Bei einigen wenigen Messungen wurden Ausschläge von zusätzlichen 1-2 Sek. festgestellt. Da diese Sonderfälle nicht die Regel darstellen, wurden sie im arithmetischen Mittel nicht berücksichtigt.

Die verwendete Semi-Clustering Implementierung für Spark wurde ursprünglich mittels Apache Spark 1.5.1 umgesetzt. Im Rahmen dieser Thesis fand eine Migration auf Apache Spark 2.2.0 statt, um die neusten Optimierungen des Frameworks verwenden zu können. An der Implementierung wurden keine Anpassungen vorgenommen und bei einem Test konnte keine Laufzeitverschlechterung gemessen werden.

Bei beiden Frameworks kamen die PageRank Implementierungen zum Einsatz, die vom jeweiligen Framework bereitgestellt werden. Sowohl Spark, als auch Flink ermöglichen eine PageRank Berechnung für ein Graphobjekt als Eingabe. Allerdings benutzen die Frameworks sich leicht unterscheidende Definitionen zur Bestimmung des PageRank, weshalb die Ergebnisse nicht vollständig vergleichbar sind.

6.4 Auswertung der Experimente

Die Experimente (6.2) zeigen, dass Flink deutlich kürzere Ausführungszeiten als Spark ausweist, wenn Graphen mit bis zu 7,2 Millionen Kanten (Hypothese 2) verarbeitet werden. Beim *wiki-topcats* (28,5 Millionen Kanten) und beim *com-orkut* Datensatz (117,1 Millionen Kanten) unterscheidet sich die Verarbeitungsgeschwindigkeit beider Frameworks nur unwesentlich. Im Gegensatz zu Spark ist Flink in der Lage, einen Graphen mit 261,3 Millionen Kanten innerhalb von 6 Minuten zu verarbeiten. Das vertex-centric und das gather-sum-apply Modell sowie Spark sind an dieser Stelle mit den vorhandenen Ressourcen nicht ausgekommen. Die beiden Flink Modelle erzeugten beim Zusammenführen von Zwischenergebnissen eine *out-of-memory Exception* und Spark wurde nach 65 Minuten manuell abgebrochen. Dieses Verhalten von Spark konnte ebenfalls in der 3. Hypothese festgestellt werden, als nur ein Worker zur Verfügung stand.

Beim Delta-Iterator müssen sich die zu verarbeitenden Daten vollständig im Speicherbereich befinden, der von Flink aktiv selbst verwaltet wird. Aus diesem Grund eignen sich die Programmiermodelle von Flink nur bei Graphen, die inklusiv aller Zwischenergebnisse diese Eigenschaft erfüllen. Der technisch unterschiedlich funktionierende Bulk-Iterator hat diese Limitierung nicht und kann deswegen auch größere Graphen verarbeiten. Eine Ausnahme stellt die scatter-gather Implementierung des PageRank Algorithmus dar. Vermutlich sorgt hier ein geringerer Overhead dafür, dass dieser in der Lage ist, den PageRank für einen Graphen mit 261,3 Millionen Kanten erfolgreich zu berechnen.

In der Hypothese 3 wurde die horizontale Skalierung des Clusters betrachtet und die 2. Hypothese bezieht sich auf die Skalierung der Graphen. In beiden Aspekten weist Spark geringere Veränderungen in der Laufzeit auf als Flink. Durch das Hinzufügen zusätzlicher Worker konnten die Laufzeiten lediglich um 0% bis 2% verbessert werden. Bei Flink sind es hingegen zwischen 30% und 50%. Demgegenüber steht bei Spark das geringere Laufzeitwachstum bei ansteigender Graphgröße mit gleichbleibenden Ressourcen. Bei Spark fällt auf, dass nicht immer alle verfügbaren Executor gleichzeitig verwendet werden. Um ein ähnliches Verhalten bei Flink zu erreichen, muss die Parallelität individuell an den Operationen definiert sein. Standardmäßig wird jedoch die globale Job-Parallelität auf jeden Operator übertragen. Im Rahmen der Experimente entspricht die Parallelität gleich der Anzahl aller verfügbaren CPU-Kerne. Eine Erklärung für die geringe Skalierung bei Spark wäre die Anwendung des vertex-cut, um den Graphen zu partitionieren. Um die langsamere Netzwerkübertragung zu vermeiden, wird ein Graph vermutlich bis zu einem bestimmten Grad auf möglichst wenige Worker verteilt. Seitens des Frameworks findet eine vertikale Skalierung statt, wobei die maximal zugeteilten Ressourcen nicht überstiegen werden können.

	Double	**Integer**	**Differenz**
Beispiel 1	78,0	37,9	51,4%
Beispiel 2	out of memory	355,5	
Beispiel 3	48,2	22,6	53,1%

Tabelle 6.4: Beispiele für Verbesserung der Laufzeiten (in Sekunden) mit unterschiedlichen Datentypen für die Knoten-ID [Apache Flink]

Die Relevanz des passenden Datentyps zeigt die Tabelle 6.4. Die Werte in der ersten Spalte zeigen Laufzeiten unterschiedlicher PageRank Implementierungen unter Flink. In diesen Beispielen wurde bei der Erzeugung des Graphen der Datentyp *Double* für die Knoten-IDs deklariert. Da es sich bei den Datensätzen um Integer-Werte handelt, wurde der Datentyp angepasst und auf *Integer* geändert. Die Laufzeiten in der zweiten Spalte beziehen sich auf

die selbe PageRank Implementierung wie in der ersten Spalte. Im ersten und dritten Beispiel konnten Verbesserungen von 51,4% und 53,1% festgestellt werden. Beim zweiten Beispiel wurde vor der Anpassung noch eine Exception geworfen. Mit einem Integer als Datentyp konnte der Algorithmus den PageRank erfolgreich berechnen.

In den Experimenten ist aufgefallen, dass das allgemeine vertex-centric Modell (Flink) durchgängig schlechtere Laufzeiten erzielt, als die anderen Programmiermodelle unter Flink. Das in der Theorie eingeschränkteste GSA-Modell erreicht circa 7% bis 20% kürzere Ausführungszeiten als das vertex-centric Modell. Die besten Laufzeiten sind mit dem scatter-gather Modell möglich.

7 Zusammenfassung und Ausblick

In diesem Kapitel sind die Ergebnisse der Thesis zusammengefasst. Im Abschnitt 7.1 erfolgt eine gebündelte Gegenüberstellung von Apache Flink und Apache Spark mit Aspekten, die in der verteilten Graphverarbeitung relevant sind. Zum Schluss werden im Abschnitt 7.2 ein Fazit gezogen und Anhaltspunkte für fortführende Projekte aufgezeigt.

7.1 Flink vs. Spark - Gegenüberstellung

Apache Flink und Apache Spark haben ihren Ursprung jeweils in einem deutschen bzw. einem amerikanischen Forschungsprojekt. Mittlerweile sind es Top-Level Projekte der Apache Software Foundation und werden kontinuierlich weiterentwickelt.

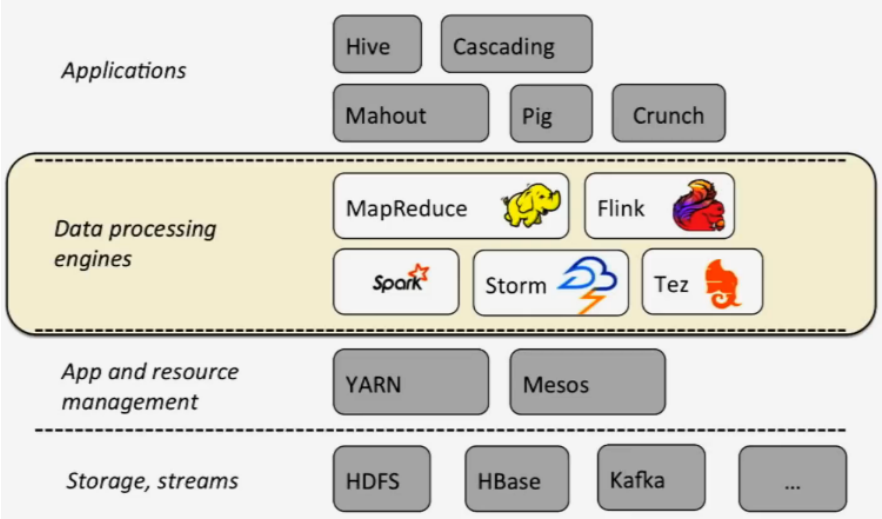


Abbildung 7.1: Flink und Spark im Kontext zueinander und zu weiteren Systemen (Quelle: https://nextplatform.com)

Wie aus der Abbildung 7.1 zu entnehmen ist, befinden sich Spark und Flink im selben Bereich wie Hadoop's MapReduce. Im Detail betrachtet unterscheiden sich beide Frameworks allerdings

grundlegend. Während bei Spark das Konzept der Batch Verarbeitung im Mittelpunkt steht, verfolgt Flink den Ansatz einer Stream-orientierten Verarbeitung. Trotz der entgegengesetzten Philosophien sind es hybride Systeme, mit entsprechenden Mechanismen für eine Batch- und Stream-Verarbeitung. Spark beherrscht technisch jedoch kein vollwertiges Streaming, sondern benutzt *Micro-Batching*. Für die Batch Verarbeitung begrenzt Flink einen (potentiell unendlichen) Stream künstlich auf viele endliche Streams.

Im Bezug auf die Graphverarbeitung stellen folgende Aspekte ein wichtiges Fundament dar: Iteration (7.1.1), Speichermanagement (7.1.2), Graph-Partitionierung (7.1.3) und -Modelle (7.1.4).

7.1.1 Iteration

Graph-Algorithmen benötigen meistens ein iteratives Verarbeitungsmodell. In Spark werden die Iterationen durch das *Driver-Program* koordiniert und mittels einzelner Batches über die Daten iteriert. Jede Iteration muss hierfür separat geplant und ausgeführt werden.

Flink bietet eine native Iterationsunterstützung und nutzt dafür die eigene Streaming Architektur. Der Delta-Iterator bearbeitet lediglich Daten, die sich seit der letzten Iteration geändert haben. Die Optimierungsebene ermöglicht zudem eine automatische Anpassung der DAG-Architektur.

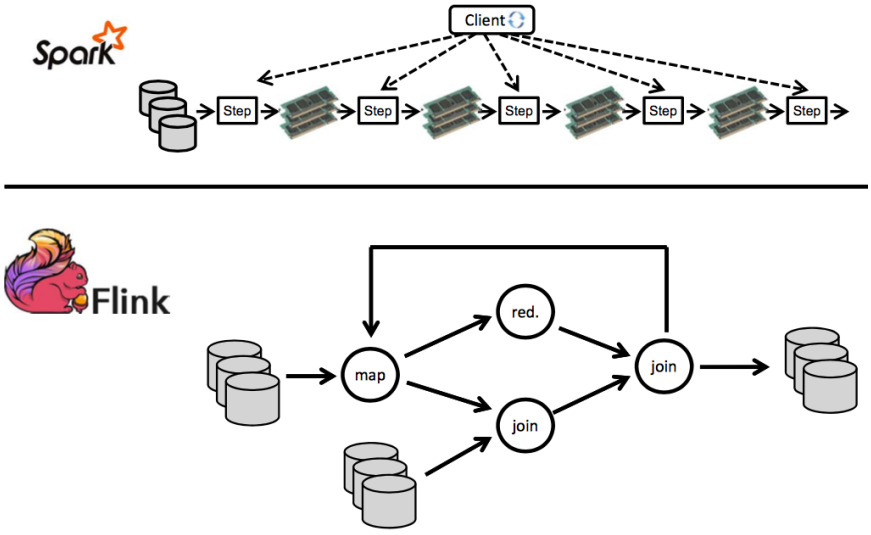


Abbildung 7.2: Apache Flink's native Iteration mit Rückführung über die Kanten; Apache Spark steuert Iterationen durch Wiederholungen von Batch Vorgängen (Quelle: http://digitale-technologien.de)

7.1.2 Speichermanagement

Die RDD Struktur stellt in Spark eine vollständige In-Memory Abstraktion dar. Die Verwaltung des Speichers überlässt Spark dabei der Java-VM.

Flink wird ebenfalls in Java-VMs ausgeführt. Im Unterschied zu Spark kommt hier eine aktive Speicherverwaltung zum Einsatz und wird von der Flink Runtime selbst geführt. Diese Vorgehensweise nennt Flink *„managed memory“* und wird bei der Batch-Verarbeitung angewendet.

7.1.3 Graph-Partitionierung

Die Partitionierung eines Graphen auf mehrere Worker-Instanzen spielt eine zentrale Rolle in der verteilten Graphverarbeitung. GraphX verwendet den weitverbreiteten Knoten-Schnitt (engl. *vertex-cut*) und bietet eine optimale Verteilung der RDGs.

Gelly nutzt eine Hash-Funktion, um einen Graphen zu partitionieren. Im Vergleich zu GraphX kann dies zu einer verstärkten Rechenlast führen, da benötigte Informationen zwischen den Instanzen ausgetauscht werden müssen.

7.1.4 Graph Programmiermodelle

GraphX selbst stellt kein Programmiermodell zur Verfügung, dies übernimmt die Pregel Implementierung oberhalb der Bibliothek. Pregel setzt das vertex-centric Modell um und synchronisiert seine Iterationen nach dem Bulk-Synchronous-Parallel Paradigma.

Analog zu GraphX implementiert auch Gelly das BSP Prinzip in seinen Programmiermodellen. Neben dem vertex-centric Modell wird das scatter-gather und das gather-sum-apply Modell unterstützt. Die drei Modelle unterscheiden sich in ihren Anwendungsbereichen, wobei GSA das eingeschränkteste und vertex-centic das allgemeingültige Modell darstellen.

7.2 Fazit und Ausblick

Die Experimente (6.2) haben gezeigt, dass Flink einen leichten Vorsprung in der Verarbeitung von Graphen hat. Zum einen ermöglichen die verschiedenen Programmiermodelle performancestarke Implementierungen für unterschiedliche Graph-Algorithmen. Zum anderen bildet die zeitgemäße Streaming Architektur mit den nativen Iterationen eine optimale Grundlage für iterative Algorithmen. Im Gegensatz zu Flink konnte Spark größere Graphen nicht verarbeiten und kam mit limitierten Ressourcen nicht zurecht. Die Vorteile von Spark liegen in der Verwendung des vertex-cut zur Graphpartitionierung und in einer dynamischen Koordination der

Teilaufgaben eines Jobs, die sich in einer automatischen, vertikalen Skalierung widerspiegeln. Das langsamere Anwachsen der Laufzeit bei skalierenden Eingabegraphen unterscheidet sich zu Flink. Demgegenüber steht eine schlechtere horizontale Skalierung seitens Spark.

In den Experimenten wurde ebenfalls ersichtlich, dass die Algorithmen durch die Abstraktion der Programmiermodelle an Performance verlieren. Lediglich das scatter-gather Modell erzielte vergleichbar gute Laufzeiten wie der Bulk-Iterator. Bei den kleinen und mittleren Graphen (7,2 Millionen Kanten) befinden sich die Laufzeiten des Modells sogar vor der PageRank Implementierung, die den Bulk-Iterator verwendet. Erst bei den großen Graphen, die mehr als 117,1 Millionen Kanten besitzen, konnte eine wachsende Laufzeitspanne beobachtet werden.

Das für die Untersuchung der Hypothesen verwendete Cluster besteht aus einem Koordinator, dem Master, und drei ausführenden Workern. In weiteren Experimenten könnte die Performance in einem deutlich größer skalierten Cluster evaluiert werden. Interessant wäre, wie sich die Laufzeit beider Frameworks verhält. In den Experimenten zeigte sich eine nur geringfügige Skalierung bei Spark und auch Flink profitierte vom dritten Worker weniger als vom zweiten. Vermutlich benötigen die Frameworks eine manuelle Optimierung der Konfigurationen, um die verfügbaren Ressourcen bestmöglich auf einen Algorithmus abzustimmen.

Echte Graphen sind heutzutage nicht statisch, sondern dynamischer Natur. In den Graphen von sozialen Netzwerken werden fortlaufend neue Relationen zwischen Objekten verknüpft und auch in anderen Bereichen verändern sich die Graphen permanent. Eine Entwicklerin der Gelly Bibliothek stellt eine experimentelle API bereit, um Graphen auf Stream-Basis mit Flink analysieren zu können [vgl. KC16; Kal17].

Literaturverzeichnis

[Aki u. a.15] Tyler Akidau, Robert Bradshaw, Craig Chambers u. a. “The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing”. In: *Vldb* 8.12 (2015), S. 1792–1803. URL: http://research.google.com/pubs/archive/43864.pdf.

[Ama17] Amazon. *Amazon EC2 - Elastic Compute Cloud*. 2017. URL: https://aws.amazon.com/de/ec2/ (besucht am 13. 04. 2017).

[And15] Jakob Smedegaard Andersen. “Funktionale Erweiterung des GraphX Frameworks”. Bachelorthesis. Hochschule für Angewandte Wissenschaften Hamburg, 2015.

[Apa14] Apache Spark. *GraphX Programming Guide Graph-Parallel*. 2014. URL: https://spark.apache.org/docs/2.1.0/graphx-programming-guide.html.

[ASF15] ASF. “Introducing Gelly : Graph Processing with Apache Flink”. 2015. URL: https://flink.apache.org/news/2015/08/24/introducing-flink-gelly.html.

[ASF16a] ASF. *Apache Hadoop Yarn*. 2016. URL: https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html (besucht am 13. 04. 2017).

[ASF16b] ASF. *Apache Kafka - A Distributed Streaming Platform*. 2016. URL: https://kafka.apache.org (besucht am 13. 04. 2017).

[ASF17a] ASF. “Apache Flink 1.3 Documentation: Dataflow Programming Model”. 2017. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.3/concepts/programming-model.html.

[ASF17b] ASF. “Apache Flink 1.3 Documentation: Iterative Graph Processing”. 2017. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/libs/gelly/iterative%7B%5C_%7Dgraph%7B%5C_%7Dprocessing.html.

[ASF17c] ASF. “Apache Flink 1.3 Documentation - Library Methods”. 2017. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/libs/gelly/library%7B%5C_%7Dmethods.html.

[ASF17d] ASF. *Apache Flink - Scalable Stream and Batch Data Processing*. 2017. URL: https://flink.apache.org (besucht am 13. 04. 2017).

[ASF17e] ASF. *Apache Mesos*. 2017. URL: https://mesos.apache.org (besucht am 13. 04. 2017).

[ASF17f] ASF. *Apache Spark - Lightning-Fast Cluster Computing*. 2017. URL: https://spark.apache.org (besucht am 13. 04. 2017).

[BP98] S Brin und L Page. "The anatomy of a large scale hypertextual Web search engine". In: *Computer Networks and ISDN Systems* 30.1/7 (1998), S. 107–17.

[Car u. a.15a] Paris Carbone, Stephan Ewen, Asterios Katsifodimos u. a. "Apache Flink : Stream and Batch Processing in a Single Engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36 (2015). URL: https://kth.diva-portal.org/smash/get/diva2:1059537/FULLTEXT01.pdf.

[Car u. a.15b] Paris Carbone, Gyula Fóra, Stephan Ewen u. a. "Lightweight Asynchronous Snapshots for Distributed Dataflows". In: *CoRR* (2015), S. 8. URL: http://arxiv.org/abs/1506.08603.

[Ewe u. a.13] Stephan Ewen, Sebastian Schelter, Kostas Tzoumas u. a. "Iterative Parallel Data Processing with Stratosphere : An Inside Look". In: *Proceedings of the 2013 International Conference on Management of Data (SIGMOD '13)* (2013), S. 1053–1056. URL: http://doi.acm.org/10.1145/2463676.2463693.

[GLG12] Je Gonzalez, Y Low und H Gu. "Powergraph: Distributed graph-parallel computation on natural graphs". In: *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (2012), S. 17–30. URL: https://www.usenix.org/system/files/conference/osdi12/osdi12-final-167.pdf.

[Inf17] InfluxData-Inc. *Components of the TICK Stack Telegraf In fl uxDB Chronograf Kapacitor*. 2017. URL: https://www.influxdata.com/time-series-platform/ (besucht am 13. 08. 2017).

[Jun u. a.17] Martin Junghanns, André Petermann, Martin Neumann und Erhard Rahm. "Management and Analysis of Big Graph Data: Current Systems and Open Challenges". In: *Big Data Handbook (eds.: S. Sakr, A. Zomaya), Springer* (2017), S. 457–505. URL: http://dx.doi.org/10.1007/978-3-319-49340-4%7B%5C_%7D14.

[Kal17] Vasia Kalavri. *An experimental Graph Streaming API for Apache Flink*. 2017. URL: https://github.com/vasia/gelly-streaming (besucht am 27. 08. 2017).

[KC16] Vasia Kalavri und Paris Carbone. *Single-pass Graph Stream Analytics with Apache Flink*. Techn. Ber. 2016. URL: https://archive.fosdem.org/2016/schedule/event/graph%7B%5C_%7Dprocessing%7B%5C_%7Dapache%7B%5C_%7Dflink/attachments/slides/1011/export/events/attachments/graph%7B%5C_%7Dprocessing%7B%5C_%7Dapache%7B%5C_%7Dflink/slides/1011/Gelly%7B%5C_%7DStream.pdf.

[KVH16] Vasiliki Kalavri, Vladimir Vlassov und Seif Haridi. "High-Level Programming Abstractions for Distributed Graph Processing". In: (2016), S. 1–19. URL: http://arxiv.org/abs/1607.02646.

[Leu u. a.09] Ian X Y Leung, Pan Hui, Pietro Liò und Jon Crowcroft. "Towards real-time community detection in large networks". In: *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics* 79.6 (2009), S. 1–10. URL: https://arxiv.org/pdf/0808.2633.pdf.

[LK14] Jure Leskovec und Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. 2014. URL: https://snap.stanford.edu/data/ (besucht am 15. 08. 2017).

[Mal u. a.10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik u. a. "Pregel: a system for large-scale graph processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (2010), S. 135–146. URL: http://dl.acm.org/citation.cfm?id=1807167.1807184.

[Mar12] Volker Markl. *Stratosphere - Next Generation Big Data Analytics Platform*. 2012. URL: https://stratosphere.eu (besucht am 13. 04. 2017).

[Mon u. a.16] G. Mon, M. Makkie, X. Li u. a. "Implementing Dictionary Learning in Apache Flink , Or : How I Learned to Relax and Love Iterations". In: *2016 IEEE International Conference on Big Data (Big Data)* (2016), S. 2363–2367. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7840869.

[MWM15] Robert Ryan McCune, Tim Weninger und Greg Madey. "Thinking Like a Vertex - A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing". In: *ACM Computing Surveys* 48.2 (2015), S. 1–39. URL: http://dl.acm.org/citation.cfm?id=2830539.2818185.

[Ped u. a.11] F. Pedregosa, G. Varoquaux, A. Gramfort u. a. *scikit-learn - machine learning in Python*. 2011. URL: https://scikit-learn.org/stable/ (besucht am 13. 04. 2017).

[Rah u. a.14] Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas und Seif Haridi. "Distributed Vertex-Cut Partitioning". In: *Distributed Applications and Interoperable Systems: 14th IFIP WG 6.1 International Conference, DAIS 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proce* (2014), S. 186–200. URL: https://doi.org/10.1007/978-3-662-43352-2%7B%5C_%7D15.

[Sal u. a.16] Salman Salloum, Ruslan Dautov, Xiaojun Chen u. a. "Big data analytics on Apache Spark". In: *International Journal of Data Science and Analytics* (2016), S. 145–164. URL: http://dx.doi.org/10.1007/s41060-016-0027-9.

[SBC10] Philip Stutz, Abraham Bernstein und William Cohen. "Signal/collect: Graph algorithms for the (semantic) web". In: *Proceedings of the 9th International Semantic Web Conference on The Semantic Web - Volume Part I* (2010), S. 764–780. URL: http://dl.acm.org/citation.cfm?id=1940281.1940330.

[Sch u. a.01] Thomas Schank, Thomas Schank, Dorothea Wagner u. a. "Finding, Counting and Listing all Triangles in Large Graphs, An Experimental Study". In: *Framework* 001907 (2001), S. 3–6. URL: http://i11www.iti.kit.edu/extra/publications/sw-fclt-05%7B%5C_%7Dt.pdf.

[Ste17] David Stephenson. *Big Data Storage and Graph-Based Analytics for Cancer*. 2017. URL: http://dsianalytics.com/big-data-storage-graph-based-analytics-cancer-research/ (besucht am 27. 08. 2017).

[Val90] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In: *Commun. ACM* 33 (1990), S. 103–111. URL: http://doi.acm.org/10.1145/79173.79181.

[VTK16] Ilya Verbitskiy, Lauritz Thamsen und Odej Kao. "When to Use a Distributed Dataflow Engine: Evaluating the Performance of Apache Flink". In: *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)* (2016), S. 698–705. URL: http://ieeexplore.ieee.org/document/7816910/.

[Xin u. a.13] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin u. a. "GraphX: A Resilient Distributed Graph System on Spark". In: *First International Workshop on Graph Data Management Experiences and Systems* (2013), 2:1–2:6. URL: http://doi.acm.org/10.1145/2484425.2484427.

[Zah u. a.12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das und Ankur Dave. “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing”. In: *NSDI* (2012). URL: http://dl.acm.org/citation.cfm?id=2228298.2228301.

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

Hamburg, 11. September 2017 Marc Kaepke