

Master-Thesis

Verteilte Rekonstruktion von Informationskaskaden

vorgelegt von:

Michael Huber

17. November 2014

1. Betreuer: Prof. Dr. Peter Fischer
2. Betreuer: Io Taxidou



Albert-Ludwigs-Universität Freiburg
Technische Fakultät
Studiengang Applied Computer Science

Kurzfassung

Das Ziel dieser Arbeit besteht in der Realisierung und Evaluation einer verteilten Rekonstruktion von Informationskaskaden. Das verteilte System wird dabei auf ein bisher bestehendes System zur zentralen Rekonstruktion von Informationskaskaden aufgebaut.

Eine Informationskaskade stellt die Ausbreitung einer Information durch die Interaktion von Benutzern dar. Dabei gibt eine Informationskaskade in korrekter zeitlicher Abfolge wieder, welcher Benutzer von wem beeinflusst wurde. Bei dieser Arbeit werden Informationskaskaden bestehend aus Retweets rekonstruiert. Um eine Rekonstruktion durchzuführen wird auch ein sozialer Graph benötigt der die Beziehungen der Benutzer untereinander angibt. Ein sozialer Graph wird in dieser Arbeit durch Benutzer von Twitter und deren Follower repräsentiert.

Eine verteilte Rekonstruktion soll es ermöglichen, sehr große soziale Graphen in den Hauptspeicher von mehreren Rechnern zu laden und dabei trotzdem eine schnelle Rekonstruktion der Kaskaden zu erhalten.

Um diese Ziele zu erreichen wird bei dieser Arbeit der soziale Graph partitioniert. Anschließend werden die Graphpartitionen auf eine gegebene Anzahl Rechner allokiert. Dabei werden die Kanten zwischen den Allokationen minimiert um den Nachrichtenfluss im verteilten System möglichst gering zu halten und so später eine gute Laufzeit zu ermöglichen. Schließlich wird das verteilte System selbst entworfen und implementiert. Im Anschluss wird die verteilte Rekonstruktion von Informationskaskaden durch Versuche analysiert und bewertet.

Die Umsetzung der Ziele ist in dieser Arbeit gelungen. Es war möglich einen größeren sozialen Graphen zu partitionieren dann zu allozieren und innerhalb einer guten Laufzeit eine verteilte Rekonstruktion von Informationskaskaden durchzuführen. Allerdings könnten, um bestimmte Teilaspekte des Systems noch besser zu prüfen, mehr Versuchsreihen durchgeführt werden.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Freiburg, den 17. November 2014

Inhaltsverzeichnis

Abbildungsverzeichnis	III
Tabellenverzeichnis	IV
Listingverzeichnis	V
1. Einleitung	1
1.1. Motivation und Ziele	2
1.2. Aufbau	4
2. Hintergrund	5
2.1. Twitter	5
2.2. Apache Storm	6
2.3. GraphChi	8
2.4. Bisherige Arbeiten am Lehrstuhl	8
2.4.1. Zentrale Kaskadenrekonstruktion	8
2.4.2. Aktivitätsgraph	9
2.4.3. Speicherkonzept Cassandra	10
3. Methodologie	12
3.1. Gemeinschaftsbasierte Graphpartitionierung	12
3.1.1. Auswahl des Algorithmus	13
3.1.2. Eingabograph	14
3.1.3. Vor der Partitionierung	14
3.1.4. Algorithmus	14
3.1.5. Ergebnisse und Auswertung	18
3.2. Allokation	23
3.2.1. Auswahl des Algorithmus	23
3.2.2. Voraussetzungen	24
3.2.3. Algorithmus	24
3.2.4. Ergebnisse und Auswertung	30

3.3. Verteilte Rekonstruktion	32
3.3.1. Ausgangspunkt	32
3.3.2. Modelle zu verteilter Rekonstruktion	33
3.3.3. Entwicklung des verteilten Modells	35
4. Versuche	50
4.1. Untersuchte Kaskaden	51
4.2. Laufzeiten	52
4.2.1. Einzelne Kaskaden	53
4.2.2. 60.000 Retweets	55
4.2.3. Gemixte Kaskaden	56
4.2.4. Abschließende Analyse	57
4.3. Entfernte Aufrufe	58
4.4. Ladezeiten und Speicherverbrauch Sozialer Graph	60
5. Fazit und Ausblick	64
5.1. Fazit	64
5.2. Ausblick	65
5.2.1. Weitergehende Versuche	65
5.2.2. Verbesserungen	65
A. Infrastrukturaufbau Rechnernetzwerk	67
B. Zusätzliche Abbildungen	71
C. Untersuchte Kaskaden	74
Literaturverzeichnis	77

Abbildungsverzeichnis

2.1.	Auszug Twitter-Account Katy Perry	6
2.2.	Netzwerk aus Spouts und Bolts	7
2.3.	Zwei Kaskaden	10
2.4.	Resultierender Aktivitätsgraph	10
2.5.	Laufzeitvergleich Cassandra zu Hauptspeicher	11
3.1.	Getrennte Partitionen	17
3.2.	Einzelne Graphpartition	18
3.3.	Allokation Beispiel 1a	28
3.4.	Allokation Beispiel 1b	29
3.5.	Zentrale Topologie	32
3.6.	Modell Übersicht	40
3.7.	Nachrichtenströme für verteilte Rekonstruktion	41
3.8.	Nachrichtenaustausch nur lokal	42
3.9.	Nachrichtenaustausch entfernt	44
3.10.	Nachrichtenströme für verteilte Rekonstruktion (erweitert)	46
3.11.	Nachrichtenaustausch Twitterstrom-Spout zu Rekonstruktions-Bolts	48
3.12.	Erweitertes Nachrichtenmodell	49
4.1.	Laufzeit einzelne Kaskaden	53
4.2.	Laufzeit 60.000 Retweets	55
4.3.	Laufzeit gemixte Kaskaden	57
A.1.	Infrastrukturaufbau Rechnernetzwerk	68

Tabellenverzeichnis

3.1.	Kantenmetriken aus der Partitionierung des Aktivitätsgraphen	20
3.2.	Anzahl der Followers und Friends und deren Speicherverbrauch per Partition	21
3.3.	Übergänge Allokation nach Zufallsverteilung	29
3.4.	Übergänge Allokation nach 1.Iteration	29
3.5.	Vergleich der Kanten von Zufallsallokation zu Hill climbing	30
3.6.	Eingesparte Kanten	31
3.7.	Errechneter Speicherverbrauch der Knoten bei 16 Bytes pro Follower . . .	31
4.1.	Laufzeit einzelner Kaskaden in Millisekunden	54
4.2.	Rekonstruktionszeit vs Nachrichtenzeit für <i>Mittel</i> in Millisekunden	54
4.3.	Rekonstruktionszeit vs Nachrichtenzeit für <i>Größte</i> in Millisekunden	54
4.4.	Laufzeit 60.000 Retweets in Millisekunden	55
4.5.	Rekonstruktionszeit vs Nachrichtenzeit für <i>10Kask</i> in Millisekunden	56
4.6.	Laufzeit gemixte Kaskaden in Millisekunden	57
4.7.	Tabellen entfernte Aufrufe für 8, 4 und 2 Knoten	62
4.8.	Speicherverbrauch sozialer Graph in Gigabyte	63
4.9.	Ladezeiten sozialer Graph	63

Listingverzeichnis

3.1.	Pseudocode Graphpartitionierungs-Algorithmus	16
3.2.	Pseudocode Allokationsalgorithmus	25
3.3.	Pseudocode Funktion berechneErgebnis	27
A.1.	Nimbus Konfiguration	68
A.2.	Spout-Supervisor Konfiguration	68
A.3.	Node-Supervisor Konfiguration	69
A.4.	Config von Topologie für Scheduler	70

1. Einleitung

Heutzutage sind soziale Netzwerke nicht mehr weg zu denken. Zeiten in welchen selbst Oma Mustermann ihren Facebook Beziehungsstatus auf Single setzt weil Opa Mustermann gerade auf Twitter verkündet, dass er sich frisch verliebt hat, sind näher an der Realität als mancher glaubt. Doch Opa und Oma Mustermann sind nicht allein wenn es um die Nutzung sozialer Netzwerke geht. Rund 1.35 Milliarden monatlich aktive Nutzer weltweit [2] kann Facebook vermerken. Bei Twitter sind es immerhin noch 248 Millionen monatlich aktive Nutzer weltweit, welche zusammen rund 500 Millionen mal am Tag "zwitschern" [10]. Aber auch Google+, Tumblr, Instagram und andere Netzwerke erfreuen sich immer größerer Beliebtheit.

Die Folge dieser hohen Nutzerzahlen ist die Erzeugung einer unglaublich großen Menge an persönlichen Daten, die Nutzer dieser Netzwerke bereitwillig, zum Teil frei zugänglich, ins Netz stellen. An der Verarbeitung solch großer Datenmengen, auch oft *BigData* genannt, besteht heutzutage ein großes Interesse. Das Hauptziel der Verarbeitung und Auswertung solcher Daten ist es Erkenntnisse über Benutzer zu sammeln. Diese können dann zum Beispiel genutzt werden um gezielt Werbung zu senden. Ein weiteres Beispiel stellt die Sammelwut der CIA in den USA dar. Diese hat den Nutzen solcher Daten schon längst erkannt und nutzt Analysewerkzeuge um potentiellen Terrorismus zu bekämpfen.

Diese Arbeit beschäftigt sich insbesondere mit der Informationsausbreitung solcher Daten. Durch eine Analyse der Informationsausbreitung kann bestimmt werden, welche Benutzer besonders viel Einfluss auf die Ausbreitung einer Information genommen haben, wie schnell diese sich verbreitet hat und welche Wege sie dabei genommen hat. Aber auch welche Art von Information sich über welche Benutzer besonders gut ausbreitet. Folgend wird ein mögliches Szenario zum Nutzen von Informationsausbreitung beschrieben.

In einem stark besiedelten Gebiet kommt es zu einer Naturkatastrophe und aufgrund der Schwere der Katastrophe sind viele Menschen ohne Obdach und Nahrung. Eine extrem schnelle Hilfe, in Form von Spenden, wird benötigt. Über Twitter verbreiten sich Informationen extrem schnell und so entscheidet sich eine Hilfsorganisation einen Prominenten zu bitten, mit einem Tweet, zum Spenden aufzurufen. Aber welchen Prominenten sollte sie am ehesten zur Hilfe bitten. Vielleicht Justin Bieber, welcher aktuell die zweitmeisten Follower auf Twitter hat? Oder doch lieber einen Prominenten der sich selbst sozial

engagiert und weniger Follower, hat aber dessen Follower diese Information vermutlich schneller ausbreiten? Hier könnte eine Analyse vorhergehender Informationsausbreitung die Antwort bringen.

Eine Informationsausbreitung wird durch sogenannte *Informationskaskaden* dargestellt. Eine Informationskaskade erfasst die Ausbreitung einer Information durch die Interaktion von Benutzern. Dabei gibt eine Informationskaskade in zeitlich korrekter Abfolge wieder, welcher Benutzer von wem beeinflusst wurde. Bei dieser Arbeit werden Informationskaskaden bestehend aus Retweets rekonstruiert. Um eine Rekonstruktion durchzuführen wird auch ein sozialer Graph benötigt, der die Beziehungen der Benutzer untereinander angibt. Ein sozialer Graph wird in dieser Arbeit durch Benutzer von Twitter und deren Follower repräsentiert.

Twitter bietet eine API die es erlaubt Informationskaskaden, sowie den sozialen Graphen herunter zu laden. Die Informationskaskaden bestehen bei Twitter aus einem Tweet und dessen Retweets (siehe Abschnitt 2.1 für nähere Informationen) und werden im Laufe der Arbeit auch *Retweet-Kaskaden* oder einfach nur *Kaskaden* genannt.

Der Lehrstuhl Web-Science an der Technischen Fakultät der Universität Freiburg nutzt die Twitter API, um Teile des sozialen Graphen und Retweet-Kaskaden herunter zu laden. Diese Arbeit greift auf diese gesammelten Daten zurück. Dabei werden insbesondere Retweet-Kaskaden von Olympia 2012 in London verwendet.

1.1. Motivation und Ziele

Eine Rekonstruktion einer Retweet-Kaskade ist erforderlich, da Retweets keinerlei Informationen beinhalten bezüglich wer Retweetet wurde. In einem Retweet ist lediglich der Retweeter selbst bekannt und der ursprüngliche Tweeter einer Kaskade. Es kann also rein aus den Retweets selbst kein kompletter Pfad der Informationsausbreitung bestimmt werden.

In der Arbeit *Social Media Analysis* von Lukas Sättler und Simon Ebner [9], welche im Rahmen eines Masterprojekts an der Universität Freiburg geschrieben wurde, werden Retweet-Kaskaden mittels eines zentralen Algorithmus rekonstruiert. Die Rekonstruktion erfolgt mit Hilfe eines sozialen Graphen der die Beziehungen der Retweeter untereinander darstellt. Durch den Abgleich des sozialen Graphen mit den Retweetern einer Kaskade wird ein Verbreitungsgraph erzeugt. Dieser Verbreitungsgraph ist eine Approximation einer Retweet-Kaskade, welche die Beziehungen der Retweeter untereinander bei seiner Erstellung miteinbezieht.

Das Problem hierbei ist, dass ein sozialer Graph potentiell extrem groß sein kann. Barack Obama zum Beispiel hat ca. 50 Millionen Follower. Würde man den kompletten

Graphen von Barack Obama herunterladen hätte dieser eine geschätzte Größe von 760 Gigabyte. Das ist der soziale Graph eines einzelnen Benutzers. Aus einem Benutzer lässt sich aber kein Informationsausbreitungsgraph erstellen. Dafür benötigt man mindestens zwei Benutzer. Daher laden wir noch den sozialen Graphen von Oprah Winfrey herunter, welche Obama beim Wahlkampf unterstützt hat. Oprah Winfrey hat ca. 26 Millionen Follower und ihr Graph damit eine Größe von ungefähr 400 Gigabyte. Der soziale Graph hat also zusammengenommen eine Größe von über einem Terabyte für zwei Benutzer. Der soziale Graph, welcher für diese Arbeit verwendet wird, hat insgesamt 1.346.925 Benutzer.

Selbstverständlich hat nicht jeder Benutzer 50 Millionen Follower, das sind natürlich Ausnahmen. Aber dieses Beispiel macht die Dimensionen klar, in welchen sich ein sozialer Graph bewegen kann.

Man sieht also, dass eine zentrale Rekonstruktion bei diesen Datenmengen, welche potentiell in den Hauptspeicher geladen werden sollen, an harte Grenzen stößt. Durch eine verteilte Rekonstruktion könnte der soziale Graph auf mehrere Rechner aufgeteilt, sowie die Ladezeiten des Graphen durch die Verteilung reduziert werden. Eine verteilte Lösung eröffnet aber auch die Möglichkeit einer parallelen Rekonstruktion mehrerer Retweet-Kaskaden und könnte so die gesamte Laufzeit der Kaskadenrekonstruktion vermindern.

Das Leitziel dieser Arbeit ist folglich *die Realisierung und Evaluation einer verteilten Rekonstruktion von Informationskaskaden*. Dies lässt sich in folgende Teilziele aufteilen:

- Durch eine Aufteilung des sozialen Graphen auf mehrere Rechner sollen auch sehr große soziale Graphen im Hauptspeicher gehalten werden können. Durch eine Rekonstruktion im Hauptspeicher kann die schnelle Laufzeit des Algorithmus der zentralen Kaskadenrekonstruktion beibehalten werden. Da es starke Hinweise darauf gibt, dass eine Informationsausbreitung in Gemeinschaften stattfindet, wird eine gemeinschaftsbasierte Graphpartitionierung des sozialen Graphen durchgeführt. Gruppen von Benutzern, welche oft miteinander interagieren, sollen dabei den gleichen Partitionen zugeteilt werden. So kann eine spätere Rekonstruktion möglichst lokal erfolgen.
- Nach der Graphpartitionierung sollen die einzelnen Partitionen auf eine gegebene Anzahl Knoten eines Rechnernetzwerks allokiert werden. Dabei soll die Anzahl der Kanten (die Beziehungen zwischen den Benutzern) zwischen den Knoten so klein wie möglich gehalten werden. Dadurch wird später die Kommunikation des verteilten Systems im Netzwerk gering gehalten und die Einbußen der Laufzeit durch einen Nachrichtenaustausch werden reduziert.
- Schließlich soll der partitionierte und allokierte soziale Graph von einer verteilten Version der Kaskadenrekonstruktion verwendet werden um Informationskaskaden zu

rekonstruieren. Die verteilte Kaskadenrekonstruktion muss hierfür zuerst entworfen und dann implementiert werden.

- Das Ergebnis der Arbeit soll durch Versuche analysiert und bewertet werden.

Letztendlich ist eine verteilte Rekonstruktion der Retweet-Kaskaden auch ein weiterer Schritt in Richtung der Echtzeitanalyse von Kaskaden. Momentan werden Kaskaden noch im Batch-Modus vom Lehrstuhl Web-Science analysiert. Die Retweeter einer vollendeten Kaskade werden herausgefiltert. Dann wird für diese Retweeter der soziale Graph geladen und anschließend die Kaskade rekonstruiert. Ein zukünftiges Ziel ist es jedoch einen einkommenden Strom von Retweets live zu analysieren. Dazu muss aber der soziale Graph in einem verteilten Rechnernetzwerk bereits vorgehalten werden. Eine erfolgreiche verteilte Rekonstruktion von Informationskaskaden würde einen Teil zu diesem Ziel beitragen.

1.2. Aufbau

Nach der *Einleitung* ist diese Arbeit in folgende weitere Kapitel aufgeteilt:

- In Kapitel 2 *Hintergrund* werden die Technologien und deren Terminologie erläutert, welche bei dieser Arbeit benutzt werden. Zudem wird in *Bisherige Arbeiten am Lehrstuhl* auf andere Arbeiten hingewiesen, welche in dem selben Forschungsgebiet wie diese Arbeit angesiedelt sind und entsprechenden Einfluss auf diese Arbeit genommen haben.
- Kapitel 3 *Methodologie* stellt den Hauptteil dieser Arbeit dar und zeigt wie eine verteilte Rekonstruktion von Informationskaskaden erreicht werden kann. Es ist in die Unterkapitel *Gemeinschaftsbasierte Graphpartitionierung*, *Allokation* und *Verteilte Rekonstruktion* aufgeteilt.
- In Kapitel 4 *Versuche* werden die gemachten Versuche vorgestellt und deren Ergebnisse analysiert und bewertet.
- Kapitel 5 *Fazit und Ausblick* gibt ein Fazit über die gesamte Arbeit ab. Die Ziele dieser Arbeit werden mit den erreichten Ergebnissen abgeglichen. Zum Schluss wird ein Ausblick auf mögliche Erweiterungen dieser Arbeit sowie daraus resultierende mögliche neue Forschungsziele gegeben.

2. Hintergrund

Dieses Kapitel umfasst hauptsächlich Hintergrundwissen zu den benutzten Technologien. Die Funktionsweise und Anwendung der Technologien wird grob dargestellt und erklärt. Zugehörige Terminologien werden eingeführt und im Einzelnen erläutert. Zudem wird in Abschnitt 2.4 *Bisherige Arbeiten am Lehrstuhl* auf aktuelle Werke hingewiesen, welche im Zusammenhang mit dieser Arbeit stehen und auf deren Ergebnisse sich diese Arbeit zum Teil gründet.

2.1. Twitter

Twitter ist ein Online-Dienst für Mikroblogging und wird auch als soziales Netzwerk bezeichnet. Zu finden ist Twitter unter der URL: <https://twitter.com/>

Twitter zeichnet sich vor allem durch das Posten von Kurznachrichten aus. Diese dürfen nicht länger als 140 Zeichen sein. Eine Kurznachricht wird als *Tweet* bezeichnet. Benutzer die einem Twitter-Account folgen sind sogenannte *Follower* dieses Twitter-Accounts. Wird Benutzer A von Benutzer B gefolgt dann ist Benutzer A ein sogenannter *Friend* von Benutzer B. Verfasst ein Benutzer einen Tweet wird dieser automatisch an seine Follower weitergeleitet. Diese haben dann wiederum die Möglichkeit diesen Tweet an ihre Follower weiterzuleiten. Das wird dann *retweeten* genannt.

Der Twitter-Account mit den meisten Followern ist (Stand 25.10.2014) Katy Perry. Sie bringt es auf insgesamt 59,5 Millionen Follower [11]. Abbildung 2.1 ist ein Auszug aus dem Twitter-Account von Katy Perry und zeigt Beispielhaft zwei ihrer Tweets und einen Retweet.

Nachstehend werden Begriffe welche in dieser Arbeit verwendet werden noch einmal im Einzelnen erläutert.

Tweet Ein Tweet ist eine Kurznachricht auf Twitter. Sie besteht maximal aus 140 Zeichen. Tweets werden den Followern eines Twitter-Accounts automatisch angezeigt.

Retweet Leitet ein Benutzer einen Tweet an seine Follower weiter, nennt man das einen Retweet.



Abbildung 2.1.: Auszug Twitter-Account Katy Perry aus [7]

Retweet-Kaskade Eine Retweet-Kaskade ist eine Folge aus einem Tweet und dessen Retweets. Eine Retweet-Kaskade enthält mehrere Informationen. Unter anderem auch die Twitter-Id des jeweiligen Verfassers. Bei Retweets aber auch die Twitter-Id des ursprünglichen Verfassers.

Follower Ein Benutzer welcher einem anderen auf Twitter folgt ist ein Follower. Dieser erhält automatisch alle Nachrichten des gefolgten Benutzers.

Follower-Liste Eine Follower-Liste ist einem einzelnen Twitter-Account zugeordnet. Sie enthält alle Twitter-Account-Ids der Follower des jeweiligen Twitter-Accounts.

Friend Der von einem Benutzer gefolgten Benutzer wird als Friend bezeichnet. Er ist sozusagen das Gegenstück eines Followers.

2.2. Apache Storm

Apache Storm ist ein kostenloses Open-Source-Framework für verteilte Berechnungen in Echtzeit. Zu finden ist Apache Storm unter der URL: <https://storm.apache.org/>.

Storm macht es einfach kontinuierliche Ströme von Daten auf verlässliche Weise zu verarbeiten [3]. In Storm werden Berechnungen in sogenannten *Topologien* durchgeführt. Jede Topologie besitzt eine gewisse Anzahl an *Spouts* und *Bolts*. Diese liefern sich durch definierte Ströme gegenseitig Daten und führen Berechnungen aus. Storm macht es möglich fehlgeschlagene Berechnungen von Datentupeln nochmals auszuführen. Somit kann Storm garantieren, dass jedes Datentupel verarbeitet wird. Abbildung 2.2 zeigt ein Beispiel einer Topologie. Sie besteht in dem Beispiel aus zwei Spouts, welche die Eingangsströme weitergeben und fünf Bolts, welche Ströme empfangen und weitergeben können.

Storm wird in dieser Arbeit verwendet, um aus einer Retweet-Kaskade und einem sozialen Graph, nach einer verteilten Rekonstruktion, einen Verbreitungsgraph zu erstellen.

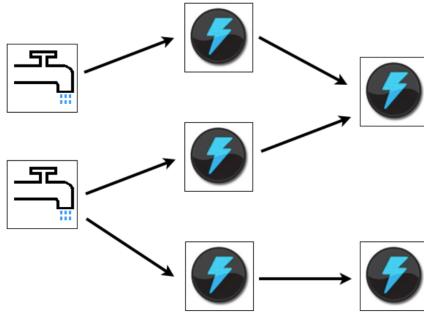


Abbildung 2.2.: Netzwerk aus Spouts und Bolts aus [3]

Nachstehend werden Begriffe, welche in dieser Arbeit verwendet werden nochmals im Einzelnen erläutert.

Spout Ein Spout ist in Abbildung 2.2 als Symbol eines Wasserhahns zu sehen. Er hat die Aufgabe eingehende Datenströme an die Bolts weiter zu geben. Ein Spout kann keine eingehenden Ströme von einem anderen Spout oder einem Bolt empfangen.

Bolt Ein Bolt ist in Abbildung 2.2 als Symbol eines Blitzen zu sehen. Er kann Datenströme von einem Spout oder einem anderen Bolt empfangen. Nach der Berechnung kann ein Bolt wiederum Daten an andere Bolts senden.

Topologie Eine Topologie ist ein Netzwerk bestehend aus Spouts und Bolts. Sie ist ein gerichteter azyklischer Graph. Eine Topologie läuft unendlich lange. Sie kann nur manuell (per *kill* Kommando) gestoppt werden.

Nimbus Der Nimbus ist der Masterserver in einem Storm Cluster. Auf diesen muss die Topologie hochgeladen werden. Der Nimbus ist dafür zuständig den Code der Topologien an die Supervisor zu verteilen. Er weist einzelne Tasks den Supervisoren zu und überwacht diese.

Supervisor Ein Supervisor ist ein Arbeiterknoten in einem Storm Cluster. Sie führen die ihnen zugewiesenen Spouts und Bolts aus und machen so die eigentlichen Berechnungen einer Topologie.

Ack Ein Ack ist ein Signal welches von einem Bolt gesendet wird, wenn ein Datentupel erfolgreich verarbeitet wurde. Senden alle Bolts in einem Berechnungsgraphen ein Ack, dann weiß der Spout welcher das Tupel ursprünglich gesendet hat, dass das Datentupel erfolgreich verarbeitet wurde. Acks können nur für Tupel ausgegeben werden, welche beim Senden *verankert* wurden.

2.3. GraphChi

GraphChi ist ein Framework, welches es ermöglicht selbst sehr große Graphberechnungen auf einem einzigen Rechner auszuführen. Zu finden ist GraphChi unter der URL: <http://graphlab.org/projects/graphchi.html>

GraphChi wird in dieser Arbeit verwendet, um eine gemeinschaftsbasierte Graphpartitionierung eines sozialen Graphen durchzuführen. Da soziale Graphen ihre Struktur über die Zeit hinweg verändern und dies in einem sehr dynamischen Maß, ist es nötig Berechnungsmethoden zu wählen, welche auch extrem große Graphen in einer annehmbaren Zeit partitionieren können können. Somit können Berechnungen schnell wiederholt und der Dynamik von sozialen Graphen Rechnung getragen werden. GraphChi bietet diese Eigenschaften und ist somit ein idealer Kandidat für diese Arbeit.

Die Informationen des folgenden Abschnitts sind zum Teil aus [5] entnommen.

Mit GraphChi ist es möglich auf Knoten basierende Berechnungen asynchron und parallel auszuführen. Dabei sind Änderungen von Kanten für darauffolgende Berechnungen sofort sichtbar. Es ist ebenfalls möglich die Struktur von Graphen während der Laufzeit zu ändern. Einer der Vorteile von Graphchi ist das einfachere Debugging. Da GraphChi auf einem einzigen Rechner ausgeführt werden kann, im Gegensatz zu den vergleichbaren Frameworks für verteilte Graphberechnungen, ist das Debuggen einer Berechnung um ein Vielfaches einfacher.

2.4. Bisherige Arbeiten am Lehrstuhl

Am Lehrstuhl Web-Science, an der Technischen Fakultät der Universität Freiburg, wurden bereits mehrere Arbeiten rund um das Thema *Rekonstruktion von Informationskaskaden* erstellt. Dieser Abschnitt nennt die Arbeiten, die auf diese Master-Thesis den meisten Einfluss haben und auf deren Ergebnisse sich diese Master-Thesis zum Teil stützt.

2.4.1. Zentrale Kaskadenrekonstruktion

Das Masterprojekt von Lukas Sättler und Simon Ebner *Social Media Analysis* [9] implementiert, mit Hilfe von Apache Storm, verschiedene Algorithmen zur Kaskadenrekonstruktion.

Die Informationen des nächsten Abschnitts wurden zum Teil aus [9] Seite 1-3 entnommen.

Retweet-Kaskaden, welche mit Hilfe der Twitter API heruntergeladen werden können, sind leider meist unvollständig. Meist fehlen Retweets und es kann nicht eindeutig festgestellt werden, welcher Benutzer welchen retweeted hat. Das Hauptziel der Kaskadenre-

konstruktion ist das Erstellen einer Approximation der Retweet-Kaskade. Dazu wird ein sozialer Graph benötigt, welcher die Beziehungen der Benutzer untereinander darstellt. Dieser kann ebenfalls mit der Twitter API heruntergeladen werden. Die Approximation, welche durch die Rekonstruktion einer Retweet-Kaskade entsteht, wird Verbreitungsgraph genannt.

Ein Verbreitungsgraph kann unter zwei Annahmen erstellt werden.

1. Benutzer von Twitter retweeten nur ihre Friends.
2. Der soziale Graph bleibt über einen Zeitraum unverändert.

Sättler und Ebner implementieren einen *blockierenden Algorithmus*, bei welchem die komplette Retweet-Kaskade beim Start bekannt sein muss. Außerdem wird auch ein *iterativer Algorithmus* implementiert, bei welchem schon beim ersten Retweet mit der Rekonstruktion gestartet werden kann. Diese beiden Algorithmen werden verwendet, um eine Konstruktion des Verbreitungsgraphen auszuführen. Sättler und Ebner stellen aber auch Modelle vor die einen Verbreitungsgraphen unter der Annahme von verschiedenen Einflussmodellen konstruieren. Ein Einflussmodell wäre zum Beispiel, dass nur der Friend mit den meisten Followern retweeted wird.

Diese Arbeit stützt sich hauptsächlich auf eine Konstruktion des Verbreitungsgraphen mit Hilfe des *iterativen Hash-Algorithmus*. Beim iterativen Hash-Algorithmus werden alle Retweets in zeitlich geordneter Abfolge rekonstruiert. Für einen neu zu rekonstruierenden Retweet einer Kaskade werden alle vorherigen Retweets auf mögliche Friends untersucht. Der Algorithmus hat also eine Komplexität von $O(|\text{Retweets}|^2)$. Die Datenstruktur, um den sozialen Graph zu halten, besteht hierbei aus einer Hashmap. Für eine detaillierte Beschreibung wird auf [9] Seite 33 verwiesen.

2.4.2. Aktivitätsgraph

Die Bachelorarbeit von Joachim Wolff *Verteilung und Partitionierung von sozialen Graphen anhand von Verbreitungsmustern von Informationen* [12], benutzt sogenannte *Aktivitätsgraphen* um Interaktionen zwischen Benutzern festzuhalten.

Der Ausgangspunkt für einen Aktivitätsgraphen ist die Annahme, dass nicht alle Benutzer in einem sozialen Netzwerk aktiv sind. Für die Rekonstruktion von Informationskaskaden sind aber hauptsächlich die aktiven Nutzer interessant. Im Falle von Twitter sind das jene Benutzer, welche schon einmal einen anderen Benutzer retweeted haben. Ein Aktivitätsgraph stellt also eine Untermenge eines sozialen Graphen dar. Um den Aktivitätsgraphen zu erstellen werden mehrere Retweet-Kaskaden analysiert. Ein Eintrag im Aktivitäts Graph beinhaltet Nutzer A, Nutzer B und die Anzahl der Aktivitäten zwischen

diesen beiden Nutzern. Der Vorteil eines Aktivitätsgraphen ist dessen geringe Größe im Vergleich zu dem kompletten sozialen Graphen.

Abbildung 2.3 zeigt ein Beispiel für zwei Kaskaden und deren Benutzer mit Ids. Aus diesen zwei Kaskaden kann dann der Aktivitätsgraph in Abbildung 2.4 erzeugt werden. Man sieht, dass die Aktivitätszahl zwischen Nutzer 1 und Nutzer 3 insgesamt zwei beträgt, da dieses Nutzerpaar in beiden Kaskaden vorhanden war.

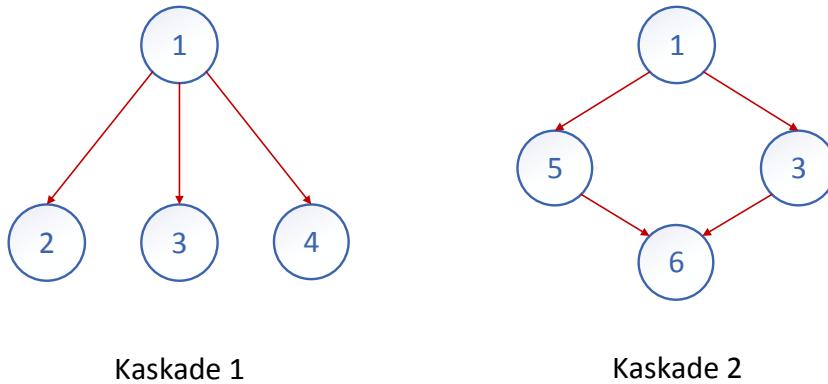


Abbildung 2.3.: Zwei Kaskaden

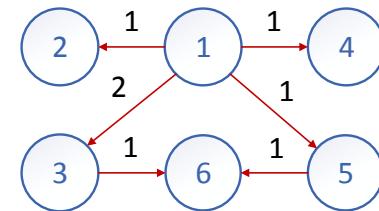


Abbildung 2.4.: Resultierender Aktivitätsgraph

2.4.3. Speicherkonzept Cassandra

Im Rahmen des Masterprojekts *Distributed Graph Storage from Cascade Reconstruction* von Roy Ramos [8] wurde untersucht wie sich ein alternatives Speicherkonzept, in Form von Apache Cassandra, auswirkt. Apache Cassandra ist ein verteiltes Datenbankverwaltungssystem. Es ist für große strukturierte Datenbanken geeignet und bietet eine hohe Skalierbarkeit sowie Ausfallsicherheit.

Statt wie bisher im Hauptspeicher soll der soziale Graph bei dem Masterprojekt von Ramos in Cassandra gespeichert werden. Dies soll ermöglichen große soziale Graphen zu laden, welche nicht in den Hauptspeicher einer einzelnen Maschine passen. Untersucht wurden hauptsächlich die Laufzeiten der Rekonstruktion von Kaskaden verschiedener Größe.

Abbildung 2.5 zeigt die Laufzeiten der Rekonstruktion von Hauptspeicher (blaue Linie) und Cassandra (orangene Linie). Es zeigte sich, dass Cassandra im Bereich der Rekonstruktion kleiner Kaskaden noch als annehmbares Speicherkonzept, im Vergleich zu Hauptspeicher, betrachtet werden kann. Ab einer Kaskadengröße von 203 nimmt die benötigte Laufzeit allerdings stetig zu. Angesichts von Kaskadengrößen die durchaus mehrere tausend Retweets beinhalten können (die größte Retweet-Kaskade im Olympia 2012 Datensatz beträgt 60.451), zeigt sich Cassandra als keine sinnvolle Alternative zu einer rein Hauptspeicher basierten Lösung.

Durch die Arbeit von Ramos wird klar, dass eine verteilte Rekonstruktion von Informationskaskaden sinnvoll erscheint. Durch eine Aufteilung des sozialen Graphen auf ein Netzwerk von Rechnern, könnte dieser auch im Hauptspeicher vorgehalten werden und so eine schnelle Rekonstruktion ermöglichen.

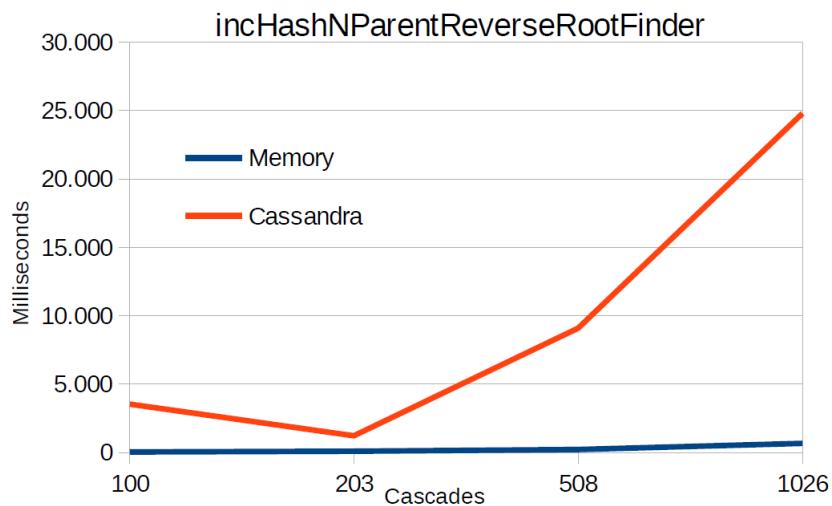


Abbildung 2.5.: Laufzeitvergleich Cassandra zu Hauptspeicher aus [8] Seite 12

3. Methodologie

Dieses Kapitel umfasst den Hauptteil dieser Arbeit und beschreibt wie eine verteilte Rekonstruktion von Informationskaskaden umgesetzt werden kann.

Dazu wird zuerst eine gemeinschaftsbasierte Graphpartitionierung des Aktivitätsgraphen (der Aktivitätsgraph wurde in Unterabschnitt 2.4.2 beschrieben) vorgenommen. Dabei wird berücksichtigt wie oft Benutzer im Graph aktiv sind. Danach werden die einzelnen Graphpartitionen durch eine Allokation auf eine gewünschte Zahl von Knoten aufgeteilt. Anhand dieser Allokation können später die Benutzer des Aktivitätsgraphen auf verschiedene Rechner in einem Netzwerk aufgeteilt werden. Auf diesen kann dann zum Schluss eine verteilte Rekonstruktion der Informationskaskaden erfolgen.

In dieser Arbeit wurden bei der verteilten Rekonstruktion und bei der Allokation nur Follower und keine Friends berücksichtigt. Der Umfang der, zum Zeitpunkt der Erstellung der Arbeit, vorhandenen Follower-Listen übersteigt den der Friends-Listen. Um aussagekräftigere Versuche durchzuführen wurden daher die Follower als Basis gewählt. Die Arbeit könnte aber, mit einigen Änderungen, auch für Friends verwendet werden.

3.1. Gemeinschaftsbasierte Graphpartitionierung

Für eine effiziente verteilte Rekonstruktion von Retweet-Kaskaden muss der soziale Graph im Hauptspeicher gehalten werden. Um dies zu erreichen müssen die potentiell sehr großen sozialen Graphen aufgeteilt werden. Eine Aufteilung der sozialen Graphen kann durch Partitionierungsverfahren erfolgen.

Um Graphen zu partitionieren bestehen in der Literatur bereits verschiedene Verfahren. Viele davon sind für die Partitionierung eines sozialen Graphen aber ungeeignet. Soziale Graphen unterscheiden sich in ihrem Aufbau von regulären Graphen durch das Auftreten von populären Knoten. Diese Knoten können sehr viele ausgehende Kanten haben (beim Knoten Barack Obama in Twitter wären es zum Beispiel 50 Millionen). Während normale Graphpartitionierungsverfahren meist versuchen ausgeglichene Graphpartitionen zu erstellen, ist dies für diese Arbeit in primärer Linie aber kein gewünschter Ansatz. Zudem muss bei der normalen Graphpartitionierung die Anzahl der Partitionen angegeben werden, was der Bildung von Gemeinschaften ebenfalls abträglich ist.

Es wird also eine gemeinschaftsbasierte Graphpartitionierung benötigt, die Benutzer, welche miteinander interagieren, möglichst den gleichen Graphpartitionen zuteilt. Die Anzahl der Partitionen kann dabei vorher nicht bestimmt werden. Da davon ausgegangen wird, dass Informationsausbreitung in Gemeinschaften stattfindet, kann durch eine gemeinschaftsbasierte Graphpartitionierung die spätere Rekonstruktion zum Großteil lokal stattfinden.

Demzufolge ist es nötig einen entsprechenden gemeinschaftsbasierten Algorithmus zur Graphpartitionierung auszuwählen. In Unterabschnitt 3.1.1 *Auswahl des Algorithmus* wird also zunächst die Auswahl des Algorithmus erläutert. Es folgen Erklärungen zum Eingabegraph und was vor einer Partitionierung zu beachten ist. In Unterabschnitt 3.1.4 *Algorithmus* wird dann schließlich der implementierte Algorithmus gezeigt, welcher mit Hilfe von GraphChi (siehe auch Abschnitt 2.3 für mehr Informationen) umgesetzt wird. Die erzielten Ergebnisse werden am Ende in Unterabschnitt 3.1.5 *Ergebnisse und Auswertung* vorgestellt.

3.1.1. Auswahl des Algorithmus

Um soziale Graphen zu partitionieren gibt es in der aktuellen Literatur ebenfalls schon Ansätze. In [1] wird für Powergraph ein Verfahren beschrieben, welches *Vertex-Cutting* genannt wird. Dort werden große Knoten in mehrere Teile gespalten, um so ähnlich große Graphpartitionen zu erhalten. Dieser Ansatz ist aber wie zuvor schon beschrieben für diese Arbeit nicht wünschenswert. Eine Folge der Nichtaufteilung großer Knoten ist allerdings, dass die Graphpartitionierung dieser Arbeit ungleich große Partitionen erzeugt.

Für die gemeinschaftsbasierte Graphpartitionierung dieser Arbeit wird schließlich ein *Label-Propagation-Algorithmus* gewählt. Bei *Label-Propagation* werden sogenannte *Labels* von Knoten zu Knoten weitergegeben. Letztendlich wird der Wert eines Knotens aus den erhaltenen *Labels* seiner Nachbarn bestimmt. Somit werden Nachbarn welche gut miteinander verbunden sind in die gleiche Partition eingeteilt. Gemeinschaften aus Benutzern bleiben somit bestehen. Zudem muss vor der Ausführung des Algorithmus keine Anzahl der zu erstellenden Partitionen genannt werden.

Als Basis für die Implementierung des Algorithmus wird [6] gewählt. Der Algorithmus wird auf die Ansprüche dieser Arbeit angepasst und erweitert. Insbesondere kann der Algorithmus ursprünglich nur mit einem Kantengewicht von 1 umgehen. Dies wird auf ein beliebiges Kantengewicht erweitert. Außerdem muss die Ausgabe des Algorithmus geändert werden. So besteht die momentane Ausgabe nur aus der Anzahl der Knoten pro Partition und deren Id. Um nach der Graphpartitionierung eine Allokation durchzuführen, müssen aber zum Beispiel auch die ausgehenden Kanten jeder einzelnen Graphpartition bekannt sein. Dies wurde ebenfalls hinzugefügt. Außerdem wurden mehrere Metriken zur

Bewertung der Partitionierung hinzugefügt, welche während des Algorithmus mitberechnet werden.

3.1.2. Eingabegraph

Der Eingabegraph ist der Graph für welchen eine Partitionierung durchgeführt werden soll. GraphChi benötigt einen Eingabegraph, welcher entweder in einem *Kantenlistenformat* oder einem *Nachbarlistenformat* vorliegen muss. Für diese Arbeit wird das *Kantenlistenformat* gewählt, da der Aktivitätsgraph bereits in einem *Kantenlistenformat* vorliegt.

Beim *Kantenlistenformat* besteht der Eingabegraph aus Zeilen, welche in drei Einträge aufgeteilt sind. Der erste Eintrag einer Zeile stellt den *Ausgangsknoten* dar. Der zweite Eintrag einer Zeile stellt den *Zielknoten* dar. Der dritte Eintrag einer Zeile stellt dann schließlich das *Gewicht* der Kante zwischen den beiden Knoten dar. Eine Zeile sieht demnach so aus: *Ausgangsknoten Zielknoten Gewicht*.

Der zu partitionierende Aktivitätsgraph wurde aus den Kaskaden der Olympischen Spiele 2012 London erstellt. Er hat insgesamt 3.675.581 Kanten. Diese Kanten haben je nach Interaktion der Benutzer untereinander entsprechende Gewichte. Die 3.675.581 Kanten haben 1.346.925 disjunkte Knoten. Jeder Knoten im Aktivitätsgraph stellt eine Twitter-Account-Id eines Benutzers dar.

3.1.3. Vor der Partitionierung

Bevor eine Partitionierung mit Hilfe von GraphChi durchgeführt werden kann, muss zuerst ein Mapping der Twitter-Account-Ids erfolgen. Der Grund dafür ist, dass GraphChi anhand der Ids der Knoten bestimmt, wieviele Knoten der Eingabegraph insgesamt hat. Da Twitter-Account-Ids durchaus im Bereich neunstelliger Zahlen und mehr liegen können, würde GraphChi beim Sharding (Zielknoten werden in Intervalle zur späteren Berechnung aufgeteilt) annehmen, dass es bis zu 1.000.000.000 Knoten im Graph gibt. Entsprechend viel Speicherplatz wird dann für die Knoten reserviert.

Um dieses Problem zu umgehen werden zunächst alle Knoten des Graphen remapped. Im Falle des Aktivitätsgraphen werden also entsprechend Ids von 0 bis 1.346.925 an die Knoten vergeben. Diese werden in einer Datei zwischengespeichert und nach der Graphpartitionierung verwendet um die ursprünglichen Twitter-Ids zu erhalten.

3.1.4. Algorithmus

Um den Algorithmus besser zu verstehen muss zunächst erklärt werden wie GraphChi seine Graphberechnungen ausführt. GraphChi implementiert eine *Update-Funktion*. Jeder Knoten des Graphen wird pro Berechnungsrounde einmal ausgeführt. D.h. für jeden

Knoten wird die erwähnte Update-Funktion einmal pro Runde aufgerufen. Der Benutzer des Frameworks schreibt seinen Code folglich innerhalb der Update-Funktion. Eine Berechnung endet wenn kein Knoten mehr aktualisiert werden kann oder wenn eine gewisse Anzahl an Berechnungsrunden erreicht ist. GraphChi kann Berechnungen von Knoten asynchron ausführen, garantiert aber, dass Knoten die voneinander abhängig, sind entsprechend behandelt werden. Besteht folglich eine Kante von Knoten A zu Knoten B und die Id von Knoten A ist kleiner als die Id von Knoten B, dann wird A immer vor B ausgeführt. Außerdem stellt GraphChi sicher, dass Knoten B die veränderte Kante noch in der gleichen Runde erhält. Haben Knoten A und B keine gemeinsame Kante, ist die Abfolge der Ausführung nicht definiert.

Die einzelnen Schritte des Algorithmus werden nun anhand von Pseudocode erklärt und dann in zwei anschließenden Beispielen verdeutlicht.

Schritte

Die einzelnen Schritte werden aus der Sicht eines Knoten erklärt. Die Zeilenangaben in den folgenden Schritten beziehen sich auf Listing 3.1.

1. Im ersten Schritt wird in Zeile 3 die eigene Id eines Knotens als dessen aktuelles Label gesetzt. Dann muss sich der Knoten selbst auf die Scheduling-Liste setzen damit er in der nächsten Iteration wieder auf der Liste der zu berechnenden Knoten steht. Diese beiden Zeilen werden nur in der ersten Iteration des Algorithmus ausgeführt. Sie sind als Setup für den Algorithmus zu sehen.
2. Im zweiten Schritt werden in Zeile 15 alle Nachbarn über den neuen Wert des Knoten informiert. In der ersten Iteration ist dies die Id des Knoten selbst.
3. Im dritten Schritt wird in Zeile 21 die Iteration erhöht und in Zeile 22 überprüft ob es noch Knoten gibt die berechnet werden müssen. Es wird also überprüft ob diese auf der Scheduling-Liste stehen. Dieser Schritt wird von Graphchi ausgeführt, ist hier aber zur Verdeutlichung des Algorithmus mit angegeben. Ist kein Knoten mehr auf der Scheduling-Liste wird der Algorithmus abgebrochen.
4. Im vierten Schritt wird zunächst in Zeile 6 ein Label *populärstesLabel* definiert. Von Zeile 7-9 sucht der Algorithmus, für den aktuellen Knoten entlang aller seiner Kanten, das populärste Label. Dabei wird jedes gefundene Label mit dem Wert der zugehörigen Kante gespeichert. Wird ein Label mehrmals gefunden, werden dessen Werte aufaddiert. Das Label, welches nach Durchlauf aller Kanten den höchsten Wert hat, wird als *poplärstesLabel* gesetzt. Findet der Algorithmus mehrere höchste Werte, wird das Label mit der höheren Id als *populärstesLabel* gesetzt.

5. Im fünften Schritt wird nun in Zeile 13 überprüft ob der aktuelle Knoten ein neues Label bekommen soll. Ist dies der Fall wird in Zeile 14 das neue Label für den Knoten gesetzt. Danach werden in Zeile 15 alle Nachbarn informiert und in Zeile 18 werden schließlich alle Nachbarn des Knotens wieder auf die Scheduling-Liste gesetzt.
6. Nun wiederholen sich die Schritte 3-5.

```

1 do {
2     if (Iteration == 0) {
3         Setze eigene Knoten-ID als neuesLabel;
4         Setze mich auf die Scheduling-Liste
5     } else {
6         populärstesLabel = 0;
7         for all(Nachbarn) {
8             Finde und setze populärstesLabel der Nachbarn;
9         }
10        neuesLabel = populärstesLabel;
11    }
12
13    if (neuesLabel != aktuellesLabel || Iteration == 0) {
14        Setze mein neuesLabel gleich mein aktuellesLabel;
15        Sende mein neues Label an alle Nachbarn;
16
17        if(Iteration > 0) {
18            Setze alle meine Nachbarn auf die Scheduling-Liste;
19        }
20    }
21    Iteration++
22} while(Scheduling-Liste > 0);

```

Listing 3.1: Pseudocode Graphpartitionierungs-Algorithmus

Beispiel 1

Abbildung 3.1 zeigt ein Beispiel in welchem der Graph am Schluss in zwei verschiedene Graphpartitionen aufgeteilt wird. Alle Kantengewichte in diesem Beispiel sind 1. Die Knoten Ids sind über den Knoten angegeben. Deren aktuelles Label ist die Zahl innerhalb des Knotens. Da der Graph zusammenhängend ist, wird GraphChi die einzelnen Knoten nach der Reihenfolge ihrer Ids bearbeiten.

1. Nach Iteration 0 setzen alle Knoten ihre eigenen Ids als Labels.
2. Nach Iteration 1 nimmt Knoten 0 das Label von Knoten 1 an. Alle anderen Knoten auf der linken Seite nehmen Label 3 an. Da Label 4 das höchste auf der linken Seite

ist, könnte angenommen werden, dass dieses sich ausbreitet. Allerdings muss hier auf die Reihenfolge der Ausführung geachtet werden. Da Knoten 2 und 3 vor Knoten 4 ausgeführt werden, hat Knoten 4 bereits zwei Nachbarn mit Label 3, wenn dieser ausgeführt wird. Die rechte Seite nimmt das höchste Label mit der Id 8 an.

3. Der Knoten mit dem Label 1 bekommt nun das höhere Label 3 zugeordnet. Man beachte nun, dass der ursprüngliche Knoten 3 sein Label nicht auf 8 ändert. Der Grund ist, dass dieser drei Nachbarn mit dem Label 3 hat und nur einen Nachbarn mit dem Label 8. Er ändert sein Label also nicht.
4. Es wird eine weitere Iteration ausgeführt in der sich jedoch nichts ändert. Nach dieser Iteration wird der Algorithmus abgebrochen.

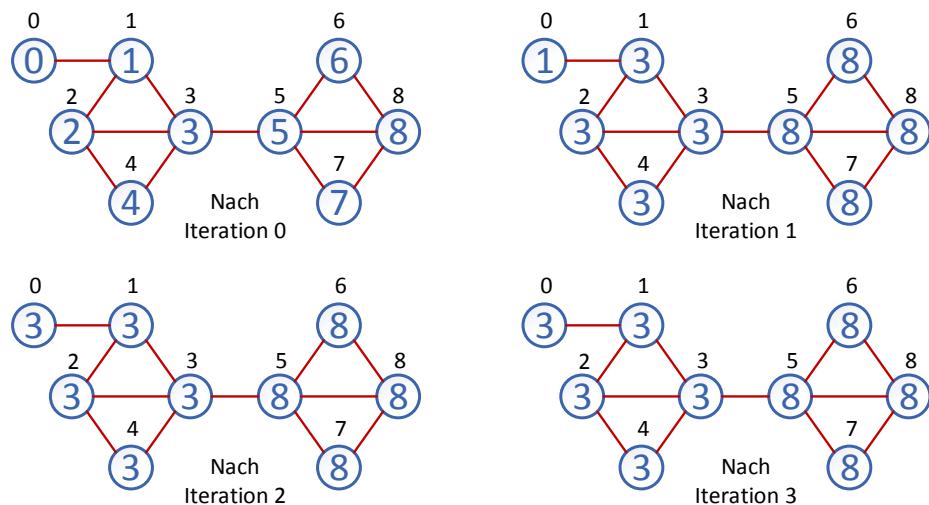


Abbildung 3.1.: Getrennte Partitionen

Beispiel 2

Abbildung 3.2 zeigt ein weiteres, aber etwas komplexeres Beispiel. In diesem Beispiel erhält der gesamte Graph das gleiche Label und ist somit eine einzige zusammenhängende Graphpartition. Die einzigen Unterschiede zum vorherigen Graph sind die Kantengewichte zwischen Knoten 3 und 5 und Knoten 5 und 8. Die Kantengewichte sind in grün angegeben. Iteration 0 ist diesmal nicht in der Abbildung enthalten. Diese setzt wie immer ihre eigenen Knoten IDs als Labels. Auch die letzte Iteration wurde weggelassen. Es werden nur die wichtigsten Änderungen erläutert.

1. Nach Iteration 1 nehmen Knoten 3, 4 und 5 diesmal, im Gegensatz zum vorherigen Beispiel, das Label 5 an. Das liegt daran, dass Knoten 3 diesmal das Label 5 statt 3, aufgrund des Kantengewichts von 4 zwischen Knoten 3 und Knoten 5, annimt.

Dadurch bekommt auch Knoten 4 das Label 5. Knoten 5 bleibt auf Label 5, da er als höchsten Nachbarn Knoten 3, hat welcher bereits Label 5 angenommen hat.

2. Nach Iteration 2 nimmt Knoten 5 nun das Label mit Id 8 an. Er findet das Label 5 vier mal vor und das Label 8 ebenfalls vier mal. Da 8 eine höhere Id als 5 ist, nimmt er die Id 8 an.
3. Nach Iteration 3 nimmt nun auch Knoten 3 und danach Knoten 4 das Label mit Id 8 an.
4. Nach den Iterationen 4,5 und 6 hat der gesamte Graph das Label 8 angenommen und ist nun eine zusammenhängende Graphpartition.

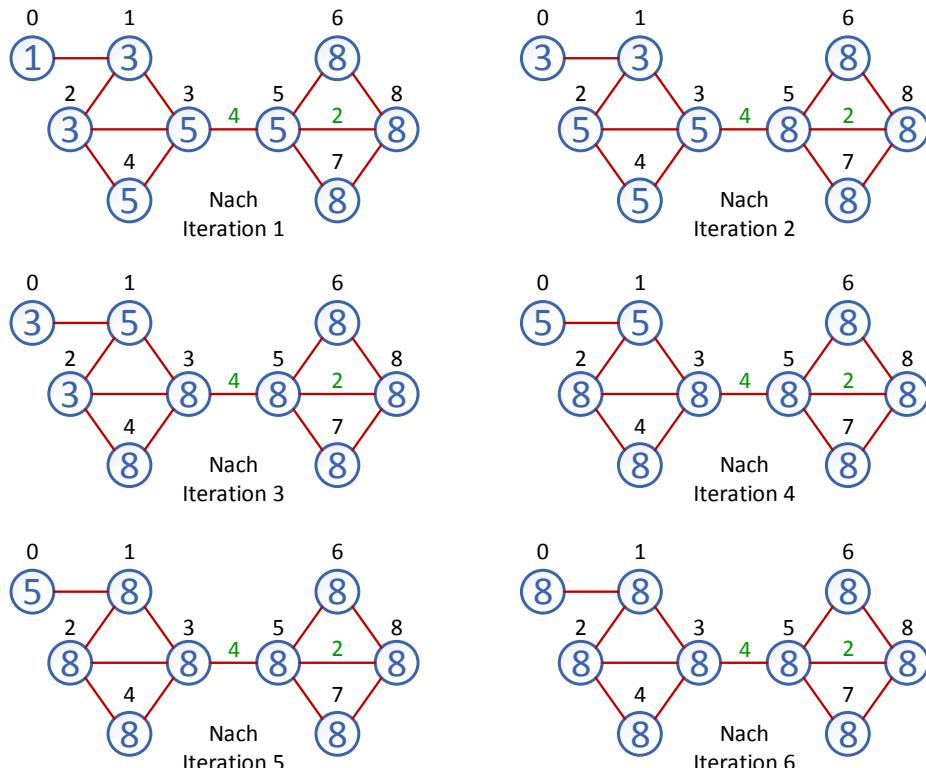


Abbildung 3.2.: Einzelne Graphpartition

3.1.5. Ergebnisse und Auswertung

In diesem Abschnitt werden die Ergebnisse aufgezeigt und bewertet, welche bei der Graphpartitionierung des Aktivitätsgraphen erreicht wurden. Zuerst werden die Ergebnisse für die gesamte gemeinschaftsbasierte Graphpartitionierung genannt. Diese sollen zeigen, ob die Graphpartitionierung insgesamt gut funktioniert hat. Dies kann vor allem aus dem Verhältnis der gewichteten Kanten zwischen den Partitionen und der Gesamtzahl der

gewichteten Kanten im Graph bestimmt werden. Zum Schluss werden exemplarisch die sechs größten Graphpartitionen, gemessen an der Zahl der Knoten, gezeigt. Dies soll einen Einblick verschaffen, ob alle Partitionen ähnliche Werte bei den Metriken aufweisen oder sich stark unterscheiden.

Der gesamte Graph hat insgesamt 1.346.925 Knoten. Vor der Ausführung der Partitionierung wurde der Graph auf *unabhängige Komponenten* untersucht. Es ergab sich eine Anzahl von 25.409 *unabhängigen Komponenten*. Die größte Komponente hat 1.281.490 Knoten und die zweitgrößte hat nur 118 Knoten. Das bedeutet, dass der Aktivitätsgraph insgesamt gut miteinander verbunden ist. Somit ist dessen Graphpartitionierung nicht trivial.

Insgesamt wurde der Graph in 80.042 *Graphpartitionen* unterteilt. Hierbei beträgt die *Gesamtzahl der gewichteten Kanten* im Graph 4.287.107 und die *Gesamtzahl der gewichteten Kanten zwischen den Partitionen* beträgt 502.213. Das bedeutet, dass 11,7 % der gewichteten Kanten des Graphen zwischen den Graphpartitionen liegen. Es werden später also geschätzt 11,7% der Aufrufe nicht lokal erfolgen können. Dieser Wert ist nicht so hoch, dass ein anderer Graphpartitionsalgorithmus herangezogen werden muss, aber auch nicht so gering, dass eine gute Laufzeit bei der verteilten Rekonstruktion garantiert werden kann. Es bleibt daher abzuwarten wie sich die Laufzeiten entwickeln.

Die ungewichteten Kanten des gesamten Graphs betragen 3.675.581 und die ungewichteten Kanten zwischen den Graphpartitionen 479.031. Es sind also 13% der ungewichteten Kanten zwischen den Graphpartitionen. Vergleicht man nun die 13 % der ungewichteten mit den 11,7 % der gewichteten Kanten ist zu erkennen, dass die Graphpartitionierung die Kantengewichte gut berücksichtigt hat. Das bedeutet, dass Kanten mit hohen Gewichten eher innerhalb von Graphpartitionen platziert sind als zwischen den Graphpartitionen.

Zum Schluss werden noch die Anzahl der Follower und Friends des kompletten Graphen gezeigt und dazu der Speicher berechnet, welchen diese voraussichtlich brauchen werden. Dabei wird angenommen, dass ein Follower oder Friend rund 16 Byte auf der Festplatte benötigen. Dieser Wert wurde aus der Anzahl Follower, im Verhältnis zu verbrauchtem Festplattenplatz, von großen Follower-Listen (Barack Obama) errechnet. Ebenfalls wird angenommen, dass ein Laden der Listen in den Hauptspeicher eine ähnliche Anzahl Bytes benötigt. 8 Bytes für einen Long, welche die Id des Followers repräsentiert und 8 weitere Bytes für Schlüssel und andere Metadaten des Containers in welchem eine Follower-Liste für den jeweiligen Benutzer gehalten wird. Die zweiten 8 Bytes könnten aber eventuell zu hoch gegriffen sein.

Es zeigte sich, dass die Anzahl der *Follower* für den kompletten Graph und dessen Knoten 2.325.988.959 beträgt. Der errechnete *Speicherbedarf für Follower* liegt also bei 37.215,8 MB. Weiterhin beträgt die Zahl der *Friends* im Graphen 610.547.659. Der errechnete

Speicherbedarf für Friends liegt also bei 9.768,76 MB. Der Unterschied zwischen Follower und Friends ist enorm. Insgesamt würde es sich lohnen die komplette Rekonstruktion auf Friends statt auf Follower auszulegen. Dies ist aber wie in diesem Kapitel zu Beginn erwähnt keine Option, da die Datengrundlage für Friends zu gering ausfällt, um eine Rekonstruktion auszuführen. Für zukünftige Projekte sollte dies aber beachtet werden.

Es lässt sich also feststellen, dass ein Speichervolumen von 37.215,8 MB für diese Arbeit auf mehrere Knoten aufgeteilt werden muss. Dies scheint im Bereich des Machbaren zu liegen.

Im Folgenden werden einzelne Graphpartitionen nun genauer betrachtet. Die Tabelle 3.1 zeigt Metriken der sechs größten Partitionen (Partitionen mit den meisten Knoten) des Graphen. Es werden Metriken der Kanten gezeigt. Die folgenden Spalten sind in der Tabelle aufgelistet:

- **GID:** Die Id der jeweiligen Graphpartition.
- **Knoten:** Anzahl der Knoten innerhalb der Graphpartition.
- **InterK:** Anzahl der Kanten die von dieser Graphpartition in andere Graphpartitionen führen. (Inter-Kanten).
- **InterWK:** Anzahl der gewichteten Kanten die von dieser Graphpartition in andere Graphpartitionen führen. (Inter-Kanten gewichtet).
- **IntraK:** Anzahl der Kanten die innerhalb dieser Graphpartitionen liegen. (Intra-Kanten).
- **IntraWK:** Anzahl der gewichteten Kanten die innerhalb dieser Graphpartition liegen. (Intra-Kanten gewichtet).
- **InterWK/IntraWK:** Verhältnis der gewichteten Inter-Kanten zu Intra-Kanten. Je geringer dieser Wert ist desto unabhängiger ist eine Graphpartition von anderen und desto besser ist die Partition gelungen.

Tabelle 3.1.: Kantenmetriken aus der Partitionierung des Aktivitätsgraphen

GID	Knoten	InterK	InterWK	IntraK	IntraWK	InterWK/IntraWK
0	184.329	69.334	74.545	1.634.808	2.018.954	0.0369226
1	27.522	32.904	36.513	36.709	52.758	0.692085
2	16.725	8.077	8.358	20.540	22.496	0.371533
3	14.759	16.701	17.947	16.193	22.813	0.786701
4	13.367	4.874	4.974	13.607	14.367	0.34621
5	11.559	802	802	60.312	71.015	0.0112934

Mit 184.329 Knoten hebt sich die größte Graphpartition deutlich von den anderen ab. Diese ist auch sehr unabhängig von den anderen Partitionen, was das Verhältnis *InterWK/IntraWK* mit einem Wert von nur 0.0369226 deutlich zeigt. Nach den sechs gezeigten Partitionen nimmt die Zahl der Knoten pro Partition rapide ab. Schon ab der 162 größten Partition sinkt die Knotenzahl unter 1.000 und dies bei insgesamt 80.042 Partitionen. Insgesamt ist zu erkennen, dass die Partitionen deutliche Unterschiede aufweisen. Den extrem guten Werten der ersten Partition stehen zum Beispiel schlechte Werte der dritten Partition gegenüber. Wie sich dies in der späteren verteilten Rekonstruktion auswirkt bleibt abzuwarten. Es wird aber erwartet, dass dadurch vereinzelte Kaskaden eine schlechte Laufzeit aufweisen, während die meisten aber eine eher gute Laufzeit aufweisen werden.

Die Tabelle 3.2 zeigt zusätzlich Metriken der Follower und Friends sowie deren voraussichtlichen Speicherverbrauch. Die folgenden Spalten sind zusätzlich zu *GID* und *Knoten* in der Tabelle aufgelistet:

- **Followers:** Die gesamte Anzahl der Follower dieser Graphpartition.
- **Friends:** Die gesamte Anzahl der Friends dieser Graphpartition.
- **Friends (MB):** Der geschätzte Speicherverbrauch (MB) für die Graphpartition anhand der Follower.
- **Followers (MB):** Der geschätzte Speicherverbraucht (MB) für diese Graphpartition anhand der Friends.

Tabelle 3.2.: Anzahl der Followers und Friends und deren Speicherverbrauch per Partition

GID	Knoten	Followers	Friends	Followers (MB)	Friends (MB)
0	184.329	270.658.845	125.740.449	4.330,54	2.011,85
1	27.522	29.698.189	9.720.360	475,171	155,526
2	16.725	10.991.830	4.179.240	175,869	668,678
3	14.759	5.287.092	4.650.074	845,935	744,012
4	13.367	4.575.498	2.691.351	73,208	430,616
5	11.559	1.769.152	22.751.446	283,064	364,023

Es zeigt sich, dass die größte Partition eine voraussichtliche Größe von 4.330, 54 MB für Follower benötigt. Dies ist in einer späteren Allokation wichtig. Da die Graphpartitionen nicht mehr weiter geteilt werden sollen, muss also mindestens ein Rechner eine ausreichende Größe für diese Graphpartition im Hauptspeicher bieten. Alle folgenden Partitionen sind deutlich kleiner und sollten daher kein Grund für weitere Bedenken sein.

Es wurden außerdem fünf Versuche durchgeführt, um die Laufzeiten der Graphpartitionierung zu ermitteln. Die maximale Laufzeit betrug 12,14 Sekunden. Die minimale Laufzeit betrug 10,71 Sekunden. Die durchschnittliche Laufzeit betrug 11,12 Sekunden. Die Anzahl der benötigten Iterationen lag bei 14. Man erkennt, dass GraphChi die 1.346.925 Knoten relativ schnell und nur innerhalb von 14 Iterationen partitionieren konnte. Dies lässt darauf schließen, dass die Struktur des Aktivitätsgraph sich gut für diese Art der gemeinschaftsbasierten Partitionierung eignet.

3.2. Allokation

Nach dem Aufteilen des Aktivitätsgraphen in einzelne Graphpartitionen, kann nun eine Allokation der Partitionen auf eine vorgegebene Anzahl an Knoten (zum Beispiel Rechner in einem Netzwerk) erfolgen. Um die einzelnen Partitionen, meist ungleicher Größe, zu verteilen, wird eine lokale Suche als Algorithmus verwendet. Das Ziel ist die Anzahl der Kanten zwischen den Knoten zu minimieren. Dabei darf ein vorgegebene Speicherkapazität pro Knoten nicht überschritten werden. Es wird also sichergestellt, dass die Rechner den aufgeteilten sozialen Graphen später auch in den Hauptspeicher laden können. Wird die Allokation später bei der verteilten Rekonstruktion angewandt, wird durch die reduzierte Anzahl an Kanten sichergestellt, dass möglichst viele Zugriffe auf den sozialen Graphen lokal erfolgen können. Das wiederum reduziert die Anzahl der Nachrichten die zwischen den Rechnern ausgetauscht werden müssen und spart somit Zeit und Netzwerkverkehr.

Zunächst wird in Unterabschnitt 3.2.1 *Auswahl des Algorithmus* der Algorithmus bestimmt mit welchem die Allokation erfolgen soll. Dann werden in Unterabschnitt 3.2.2 *Voraussetzungen* die Vorbedingungen genannt, welche erfüllt sein müssen um eine Allokation durchzuführen. In Unterabschnitt 3.2.3 *Algorithmus* wird schließlich der eigentliche Algorithmus gezeigt. Zum Ende werden dann in Unterabschnitt 3.2.4 *Ergebnisse und Auswertung* die Ergebnisse präsentiert und ausgewertet.

3.2.1. Auswahl des Algorithmus

Die Allokation von Graphpartitionen auf eine gegebene Anzahl Knoten, unter der Reduzierung der Kanten, ist ein Optimierungsproblem.

Um dieses Optimierungsproblem zu lösen, wird in dieser Arbeit das heuristische Optimierungsverfahren *Hill climbing* verwendet. *Hill climbing* gehört zur Gruppe der lokalen Suche. Es garantiert keine besten Lösungen, sondern verbessert eine schon vorhandene Lösung. Somit kann *Hill climbing* kein globales Optimum der Lösung erreichen, sondern nur ein lokales Optimum. Der Vorteil des Algorithmus liegt bei seiner guten Laufzeit, sowie seiner relativ einfachen Umsetzung.

Da bei der Allokierung der Graphpartitionen in dieser Arbeit kein globales Optimum erreicht werden muss, sondern gute lokale Optima ausreichen, ist *Hill climbing* eine geeignete Wahl.

Sollte später bei der Auswertung festgestellt werden, dass *Hill climbing* keine guten Lösungen liefert, können andere Algorithmen wie zum Beispiel *Tabu Suche* oder *Simulated Annealing* in Betracht gezogen werden.

3.2.2. Voraussetzungen

Um eine Allokation durchzuführen, müssen bestimmte Voraussetzungen erfüllt sein. Genauer gesagt, müssen bestimmte Informationen bereits im Vorfeld bekannt sein. Hierfür wurden bereits während der Partitionierung des Graphen Informationen bezüglich der einzelnen Graphpartitionen gesammelt. Für die Allokation müssen gegeben sein:

- Die Kanten der Partitionen untereinander. Es muss also für jede Partition bekannt sein wieviele Kanten sie jeweils zu jeder anderen Partition hat.
- Wieviele Follower sind innerhalb von jeder Partitionen enthalten. Dies ist wichtig um die benötigte Speicherkapazität der allokierten Partitionen bestimmen zu können.
- Wieviel Bytes werden pro Follower angenommen. Um den Speicherplatz jeder Partition zu bestimmen, muss eine Byte Anzahl pro Follower angenommen werden, welche dann jeweils mit der Anzahl der Follower jeder Partition multipliziert wird.

Sind diese Informationen nicht gegeben kann keine Allokation der Graphpartitionen stattfinden.

3.2.3. Algorithmus

Der implementierte Algorithmus ist im Kern ein sogenannter *Hill-Climbing*-Algorithmus. Dieser optimiert die *Anzahl der gewichteten Kanten* zwischen den Knoten *im Gesamten*. Er wird mit den einschränkenden Parametern, *Anzahl der Knoten* und *Maximale Speicherkapazität der Knoten* versehen.

Die einzelnen Schritte des Algorithmus werden aus Sicht der *höchsten Ebene* aufgeführt und anhand des Pseudocodes in Listing 3.2 erklärt. Danach wird die Funktion *berechnere-Ergebnis* noch einmal getrennt von der höchsten Ebene genauer, anhand von Listing 3.3, erklärt. Darauf folgt ein Beispiel einer Allokation.

Schritte höchste Ebene

Der Algorithmus wird in die folgenden sieben Schritte unterteilt und bezieht sich auf Listing 3.2:

1. Vor dem eigentlichen Algorithmus werden die Graphpartitionen zufällig auf den vorgegebenen Knoten verteilt. Dies ist nötig, da eine lokale Optimierung wie Hill climbing bereits einen Zustand braucht der optimiert werden kann. Bei der zufälligen Verteilung werden die maximalen Speichergrenzen der Knoten eingehalten. Der Schritt ist im Pseudocode, der Überschaubarkeit halber, lediglich als Einzeiler in Zeile 1 dargestellt.

```

1 Zufällige Allokation der Graphpartitionen;
2 Berechnung der gesamten Zahl aller gewichteten Kanten zwischen
   den Knoten;
3 Speichern nach gesamteZahlGewKanten;

4

5 while (Abbruchkriterium) {
6     bestesErgebnis = gesamteZahlGewKanten;
7     for all (Partitionen) {
8         for all (Knoten) {
9             if (Platz für Partition auf Knoten ausreichend) {
10                tempErgebnis = berechneErgebnis(
11                    gesamteZahlGewKanten, Partition, Knoten);
12                if(tempErgebnis < bestesErgebnis) {
13                    bestesErgebnis = tempErgebnis;
14                    Merke Partition;
15                    Merke Knoten;
16                }
17            }
18        }
19        if(Abbruch) {
20            Setze Abbruchkriterium;
21        }
22        gesamteZahlGewKanten = bestesErgebnis;
23        Verschiebe Partition auf Knoten entsprechend bestem
           Ergebnis;
24    }

```

Listing 3.2: Pseudocode Allokationsalgorithmus

2. Dieser Schritt wird ebenfalls vor dem Algorithmus ausgeführt. Er berechnet die gesamte Anzahl der gewichteten Kanten zwischen den Knoten (Zeile 2-3).
3. Dieser Schritt startet den eigentlichen Algorithmus. Sein Ziel ist es die gesamte Anzahl der gewichteten Kanten zwischen den Knoten zu minimieren. Er verläuft solange in einer Schleife bis das Abbruchkriterium erfüllt ist (Zeile 5, 24).
4. Speichere das gesamte Gewicht aller Kanten zwischen den Knoten in *bestesErgebnis* (Zeile 6). Dann wird über alle Graphpartitionen und danach über alle Knoten iteriert (Zeile 7-8, 16-17). Es wird überprüft ob für die Graphpartition noch Platz auf dem Knoten ist (Zeile 9). Ist dies der Fall wird das temporäre Ergebnis, welches aus

der Platzierung der Partition auf dem neuen Knoten resultiert, in *tempErgebnis* zwischengespeichert (Zeile 10). Danach wird überprüft, ob das temporäre Ergebnis besser ist als das bisherige beste Ergebnis (Zeile 11). Ist dies der Fall, wird das beste Ergebnis entsprechend ersetzt und die aktuelle Partition und der Knoten wird vom Algorithmus vorgemerkt (Zeile 12-14).

5. Es wird überprüft ob ein Abbruch stattfinden soll (Zeile 19). Das normale Abbruchkriterium ist erfüllt, wenn kein besseres Ergebnis mehr gefunden werden kann. Der Algorithmus kann aber auch vorzeitig abgebrochen werden. Ein Abbruch kann zum Beispiel stattfinden, wenn eine vorgegebene Anzahl an Iterationen erreicht wurde oder die Ergebnisverbesserung kleiner ist, als ein vorgegebener Schwellwert. So kann eine zu lange Laufzeit des Algorithmus, bei minimaler Verbesserung des Ergebnisses, verhindert werden. Soll ein Abbruch stattfinden, wird ein entsprechendes Flag gesetzt (Zeile 20).
6. Setze die gesamte Zahl der gewichteten Kanten gleich dem besten Ergebnis des aktuellen Durchlaufs (Zeile 22). Danach wird der neue Zustand gesetzt, indem die vorher gemerkte Graphpartition (Zeile 13) auf den vorher gemerkten Knoten (Zeile 14) verschoben wird (Zeile 23).
7. Nun wiederholen sich die Schritte 3 bis 6.

Schritte Funktion *berechneErgebnis*

Die Funktion wird in die folgenden sechs Schritte unterteilt und bezieht sich auf Listing 3.3:

1. Der Knoten auf welchem die aktuell zu berechnende Graphpartition vorher war, wird in *vorherigerKnoten* gespeichert (Zeile 2).
2. Für alle Kanten die die aktuelle zu berechnende Partition zu anderen Partitionen hat werden die Schritte 3 bis 5 ausgeführt (Zeile 3, 11).
3. Hole den Knoten der Zielpartition der momentan durchlaufenden Kante und speichere ihn in *zielKnoten*. (Zeile 4). Dies ist sozusagen der Knoten der am anderen Ende der Kante, der aktuell zu berechnenden Partition, liegt.
4. Ist der Knoten auf welchem die aktuell zu berechnende Graphpartition zuvor lag ein anderer als der Knoten der Zielpartition, dann ziehe das *Gewicht der Kante**2 von der gesamten Anzahl der gewichteten Kanten ab (Zeile 5-7). Das bedeutet, waren die beiden Partitionen vorher nicht auf dem gleichen Knoten, ist dieses Gewicht momentan in der Gesamtzahl aller Kantengewichte enthalten. Es muss also

vorerst abgezogen werden. Das Gewicht der Kante muss außerdem doppelt abgezogen werden, da dieses zuvor auch für jeden der beteiligten Knoten hinzugefügt wurde. Waren die Partitionen zuvor auf dem gleichen Knoten, muss nichts abgezogen werden, da folglich auch kein Gewicht zu der Gesamtzahl aller Kantengewichte hinzugefügt wurde.

5. Ist der Knoten auf welchem die aktuell zu berechnende Partition nun liegt ein anderer als der Knoten der Zielpartition, dann füge das *Gewicht der Kante**2 der gesamten Anzahl der Kanten zu (Zeile 8-10). Sind also die beiden Partitionen jetzt auf dem gleichen Knoten muss kein Gewicht hinzugefügt werden. Sind diese nun aber auf unterschiedlichen Knoten wird das *Gewicht**2 hinzugefügt. Das Gewicht muss für jeden der zwei Knoten zu der Gesamtzahl der Kantengewichte hinzugefügt werden. Daher wird es doppelt genommen.
6. Das berechnete Ergebnis für den aktuellen Knoten und die aktuell zu berechnende Partition wird zurück gegeben (Zeile 12).

```

1 berechneErgebnis(gesamteZahlGewKanten , aktuellePartition ,
2   aktuellerKnoten) {
3   vorherigerKnoten = Hole vorherigen Knoten aus aktuellePartition ;
4   for all (Kanten zu anderen Partitionen) {
5     zielKnoten = Hole von Kante die Zielpartition und davon den
6       Zielknoten .
7     if (vorherigerKnoten != zielKnoten) {
8       Ziehe von gesamteZahlGewKanten das Gewicht*2 der
9         aktuellen Kante ab ;
10    }
11  }
12  Gebe gesamteZahlGewKanten zurück ;
13 }
```

Listing 3.3: Pseudocode Funktion berechneErgebnis

Beispiel

Abbildung 3.3 und Abbildung 3.4 zeigen ein Beispiel einer simplen Allokation. Im Folgenden werden detailliert die Schritte genannt, welche auf den vier Bildern ausgeführt werden.

1. Das erste Bild auf der linken Seite von Abbildung 3.3 zeigt den Zustand vor der Graphpartitionsverteilung auf die drei Knoten. In der Wolke sind die einzelnen Partitionen zu sehen. Die Zahl innerhalb der Partitionen stellt in diesem Beispiel ihre Id und gleichzeitig auch die Anzahl der Follower dar, welche in der Partition vorhanden sind. Die Kanten zwischen den Partitionen sind mit zugehörigen Kantengewichten versehen. Den maximalen Speicher die ein Knoten fassen kann beträgt 12.
2. Das zweite Bild auf der rechten Seite von Abbildung 3.3 zeigt, dass die Partitionen nun zufällig auf die einzelnen Knoten verteilt wurden. Die getrichelten Linien geben die Kanten zwischen den Partitionen an. Die normalen Linien geben die addierte Kantenzahl zwischen den Knoten an. Die gesamte Anzahl der Kanten zwischen den Knoten beträgt 22. Die Zufallsverteilung achtet darauf keine Partitionen auf Knoten zu verteilen, wenn deren Speicher überschritten würde.

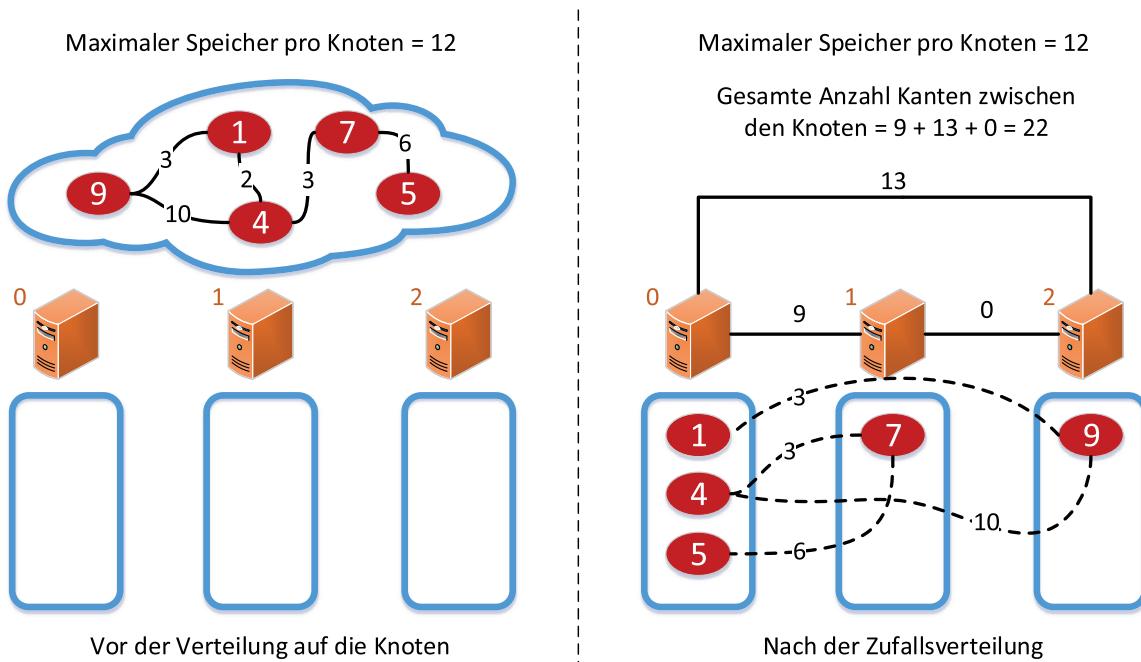


Abbildung 3.3.: Allokation Beispiel 1a

3. Das dritte Bild auf der linken Seite von Abbildung 3.4 zeigt die Allokation nach der 1. Iteration. Die Partition mit der Id 5 wurde von Knoten 0 nach Knoten 1 transferiert. Tabelle 3.3 zeigt die vor der 1. Iteration möglichen Übergänge. Es werden nur Übergänge angezeigt, welche den Speicher der Knoten nicht überschreiten. Man könnte zum Beispiel annehmen, dass Partition 4 nach Knoten 2 transferiert wird, da dies ein Kantengewicht von 8 (10-2) einsparen würde. Dies ist aber nicht möglich da der Speicher von Knoten 2 dann auf 13 ansteigen würde.

Tabelle 3.3.: Übergänge Allokation nach Zufallsverteilung

Partitions-Id	Von Knoten	Nach Knoten	Ergebnis
1	0	1	$22+2=24$
1	0	2	$22+2-3=21$
4	0	1	$22+2-3=21$
5	0	1	$22-6=16$

4. Das vierte Bild auf der rechten Seite von Abbildung 3.4 zeigt die Allokation nach der 2. Iteration. Die Partition mit der Id 1 wurde von Knoten 0 nach Knoten 2 transferiert. Tabelle 3.4 zeigt die vor der 2. Iteration möglichen Übergänge.

Tabelle 3.4.: Übergänge Allokation nach 1. Iteration

Partitions-Id	Von Knoten	Nach Knoten	Ergebnis
1	0	2	$16+2-3=15$
5	1	0	$16+6=22$

5. Die Allokation ist nun beendet, da von diesem Zustand aus keine Verbesserung der Gesamtzahl der Kanten mehr erreicht werden kann.

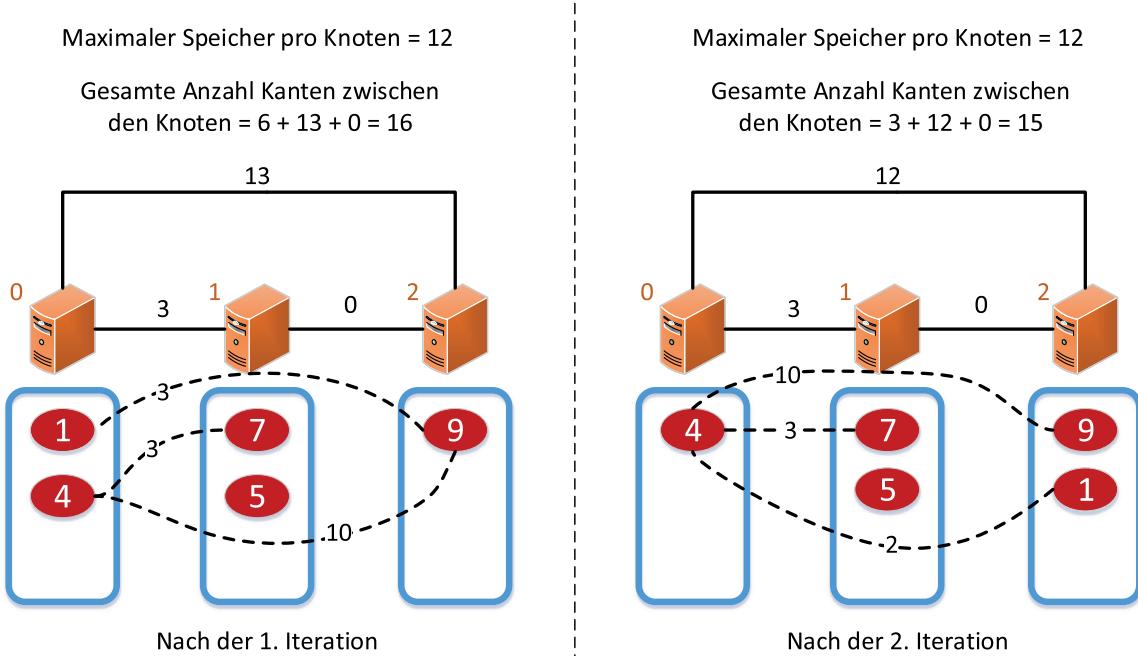


Abbildung 3.4.: Allokation Beispiel 1b

3.2.4. Ergebnisse und Auswertung

Bei den Ergebnissen wird erwartet, dass die Allokation die Anzahl der gewichteten Kanten, im Vergleich zur Graphpartitionierung, zwischen den Knoten deutlich verringert hat. Im Speziellen sollte dabei das verbesserte Ergebnis der Zufallsallokation nochmals merklich durch den Hill-Climbing-Algorithmus verbessert werden. Ist dies nicht der Fall sollte ein anderer Algorithmus für das Optimierungsproblem gewählt werden. Weiterhin soll das endgültige Verhältnis *der Anzahl gewichteter Kanten zwischen den Knoten, zu der Anzahl der gewichteten Kanten im sozialen Graph*, bestimmt werden. So kann geschätzt werden wieviele Aufrufe, der später erfolgenden Rekonstruktion von Informationskaskaden, nicht lokal sein werden. Schließlich wird für die allokierten Partitionen noch der erwartete Speicherverbrauch angegeben.

Die Allokation wurde für zwei, vier und acht Knoten ausgeführt. Tabelle 3.5 zeigt die Kantenzahlen der Allokationen und die Verbesserung des Ergebnisses der Zufallsallokation durch *Hill climbing*. Es wurde aus drei Versuchen je Knotenzahl die jeweils beste Allokation gewählt. Dabei wird für jeden Allokationstyp, welcher die Anzahl der Knoten angibt, gezeigt wie hoch die Kantenzahlen nach der Zufallsallokation und nach dem *Hill climbing* sind. Schließlich wird noch die Verbesserung der Kantenzahl durch *Hill climbing* angegeben.

Tabelle 3.5.: Vergleich der Kanten von Zufallsallokation zu Hill climbing

Allok.-typ	Nach Zufallsallokation	Nach Hill climbing	Verbesserung
2 Knoten	247.633	114.586	133.047
4 Knoten	376.491	179.918	196.573
8 Knoten	441.023	267.554	173.469

Es zeigt sich, dass durch eine Ausführung von *Hill climbing* die gewichtete Kantenzahl der Zufallsallokation, bei nahzu allen Allokationstypen, um ca. 50 % reduziert werden kann. Dies wird als ein gutes Ergebnis angesehen. Der Hill-Climbing-Algorithmus scheint zur Allokation der Graphpartitionen geeignet zu sein und kann als Algorithmus beibehalten werden.

In der Tabelle 3.6 ist zu sehen um wieviel die Allokation das Ergebnis der Graphpartition verbessern konnte. Außerdem wird die Anzahl der Kanten nach der Allokation, zur Gesamtzahl der gewichteten Kanten im Aktivitätsgraph, im Verhältnis dargestellt. Die gesamte Anzahl der gewichteten Kanten im Aktivitätsgraph beträgt 4.287.107. Dies wurde in Unterabschnitt 3.1.5 festgestellt.

Es ist zu sehen, dass bei allen Allokationstypen das Verhältnis *der Anzahl der gewichteten Kanten nach Hill climbing zu der Anzahl der gesamten gewichteten Kanten* im Graph sehr gering ist. Dies ist insbesondere später bei der verteilten Rekonstruktion

Tabelle 3.6.: Eingesparte Kanten

Allok.-typ	Vor Allokation	Nach Allokation	Verbesserung	Verhältnis Allok/Ges.
2 Knoten	502.213	114.586	387.627	2,7 %
4 Knoten	502.213	179.918	322.295	4,2 %
8 Knoten	502.213	267.554	234.659	6,2 %

wichtig, da abgeschätzt werden kann wieviel Prozent der Retweets vermutlich nicht direkt auf den lokalen Rechnern rekonstruiert werden können, sondern von entfernten Rechnern Follower-Daten benötigen.

Tabelle 3.7 zeigt den errechneten Speicherverbrauch der einzelnen Knoten unter Einhaltung des maximal erlaubten Speicherverbrauchs. Es zeigt sich, dass die Allokation es vorzieht Knoten komplett zu befüllen und nur einen bis zwei Knoten nicht voll auszulasten. Dies ist logisch, da nur die Anzahl der gewichteten Kanten als zu verbesserndes Merkmal ausgewählt wurde und nicht die proportionale Auslastung der Knoten.

Tabelle 3.7.: Errechneter Speicherverbrauch der Knoten bei 16 Bytes pro Follower

Allokationstyp	Knoten Nr.	Speicherverbr. (MB)	Max. erlaubt (MB)
2 Knoten	0	18.016	19.200
2 Knoten	1	19.200	19.200
4 Knoten	0	8.416	9.600
4 Knoten	1	9.600	9.600
4 Knoten	2	9.600	9.600
4 Knoten	3	9.600	9.600
8 Knoten	0	3.878	5.120
8 Knoten	1	5.120	5.120
8 Knoten	2	5.120	5.120
8 Knoten	3	5.120	5.120
8 Knoten	4	5.120	5.120
8 Knoten	5	5.120	5.120
8 Knoten	6	2.618	5.120
8 Knoten	7	5.120	5.120

3.3. Verteilte Rekonstruktion

Nach erfolgreicher gemeinschaftsbasierter Graphpartitionierung und Allokation, die es ermöglichen den sozialen Graphen auf Knoten in einem Rechnernetzwerk zu verteilen, kann nun ein System entwickelt werden um eine verteilte Rekonstruktion von Retweet-Kaskaden durchzuführen. Dieses wird mit dem Framework Apache Storm [3] umgesetzt (siehe auch Abschnitt 2.2 für mehr Informationen).

Dabei wird in Unterabschnitt 3.3.1 *Ausgangspunkt* zuerst der Ausgangspunkt des Systems gezeigt, welches die zentrale Kaskadenrekonstruktion darstellt. Danach werden in Unterabschnitt 3.3.2 *Modelle zu verteilter Rekonstruktion* verschiedene Modelle erläutert, welche eine verteilte Rekonstruktion ermöglichen. Daraus wird eines der Modelle gewählt. In Unterabschnitt 3.3.3 *Entwicklung des verteilten Modells* wird dann schließlich die komplette Entwicklung des verteilten Modells gezeigt.

3.3.1. Ausgangspunkt

Um den Ausgangspunkt der Entwicklung deutlich zu machen wird in Abbildung 3.5 die Topologie der zentralen Kaskadenrekonstruktion dargestellt. Grundsätzlich zeigt die Abbildung den Spout als Quadrat und die Bolts als Kreise. Der soziale Graph wird als Datenbanksymbol dargestellt. Im Folgenden werden die einzelnen Komponenten der Topologie, welche auch auf der Abbildung zu sehen sind, erklärt.

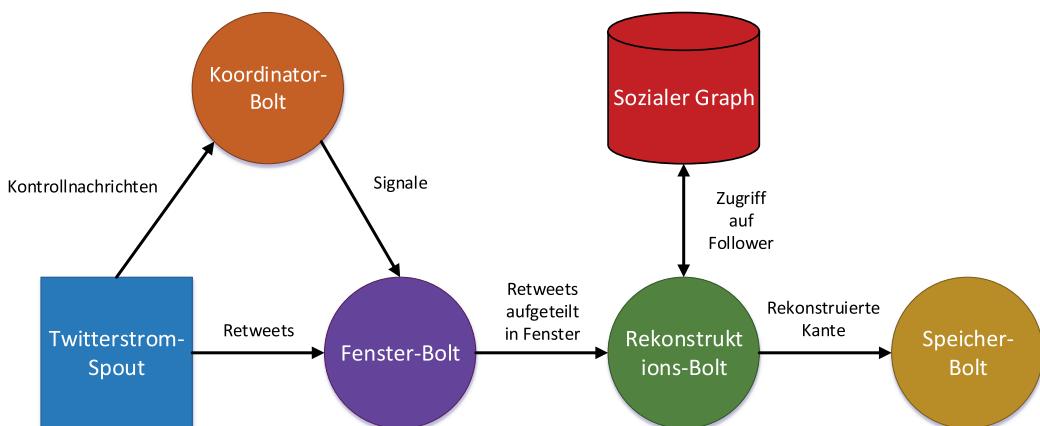


Abbildung 3.5.: Zentrale Topologie

- Der *Twitterstrom-Spout* hat die Aufgabe die Retweets einer Retweet-Kaskade zu lesen und weiter zu senden. Dies könnte zum Beispiel ein eingehender Retweet-Strom von der Twitter-API sein. In der momentanen Rekonstruktion wird der *Twitterstrom-Spout* allerdings eher in einer Art *Batch-Prozess* benutzt. Das bedeutet, dass die Retweet-Kaskaden zuvor auf der Festplatte gespeichert wurden und

nun von dort aus, vom *Twitterstrom-Spout*, geladen werden. Die Retweets versendet der Spout dann an den *Fenster-Bolt*. Außerdem werden über einen Kontrollstrom Nachrichten an den *Koordinator-Bolt* gesendet. Diese Nachricht wird zum Beispiel gesendet wenn keine neuen Retweet-Kaskaden mehr gelesen werden können.

- Der *Koordinator-Bolt* hat lediglich die Aufgabe Signale an den *Fenster-Bolt* zu senden. Bekommt der Koordinator-Bolt eine entsprechende Nachricht vom *Twitterstrom-Spout*, wandelt er dies in ein Signal um und sendet es an den *Fenster-Bolt*. Bekommt er zum Beispiel die Nachricht, dass alle Retweet-Kaskaden gelesen wurden, sendet er ein *Flush-Signal* an den *Fenster-Bolt*.
- Der *Fenster-Bolt* hat die Aufgabe die erhaltenen Retweets in einzelne Fenster aufzuteilen und die Fenster mit einem Status zu versehen. Er versendet Nachrichten in der Form *[Fenster-Id, Fensterstatus, Nutzlast]*. Für jede Retweet-Kaskade die er erhält wird eine separate Fenster-Id angelegt. Der Fensterstatus bestimmt sich je nach erhaltener Kaskade. Die wichtigsten Fensterstati sind *Opening*, *Consuming* und *Closing*. Erhält der *Fenster-Bolt* zum ersten mal eine Kaskade wird der Status *Opening* verwendet. Ist die Kaskade bekannt wird *Consuming* verwendet und ist dies der letzte Tweet einer Kaskade wird *Closing* verwendet. Ein Beispiel einer Nachricht für den ersten erhaltenen Retweet der ersten Kaskade wäre: *[1, Opening, Retweet]*. Diese Nachrichten werden an den *Rekonstruktions-Bolt* weitergegeben.
- Der *Rekonstruktions-Bolt* hat die Aufgabe eine Rekonstruktion der Retweet-Kaskaden auszuführen. Dazu benötigt er Zugriff auf den sozialen Graphen, welcher die Follower-Listen der Benutzer hält. Ein wichtiger Punkt hierbei ist, dass bei der zentralen Rekonstruktion, der Graph vor dem Bauen der Topologie geladen wird und dann als Objekt dem *Rekonstruktions-Bolt* mitgegeben wird. Dies ist bei großen Graphen in einer verteilten Rekonstruktion später nicht möglich. Einen kurzen Überblick über die eigentliche Rekonstruktion wurde in Unterabschnitt 2.4.1 gegeben (für umfassende Informationen wird hierzu auf [9] verwiesen). Die rekonstruierten Kanten werden dann schließlich an den *Speicher-Bolt* verschickt.
- Der *Speicher-Bolt* hat lediglich die Aufgabe die vom *Rekonstruktions-Bolt* erhaltenen rekonstruierten Kanten in Dateien auf der Festplatte zu schreiben. Dabei erhält jede Kaskade eine separate Datei. Diese Datei stellt dann den Verbreitungsgraphen dar.

3.3.2. Modelle zu verteilter Rekonstruktion

Um nun eine verteilte Rekonstruktion, der vorherig gezeigten zentralen Rekonstruktion, umzusetzen wurden verschiedene Modelle in Betracht gezogen.

Die ursprüngliche Idee war es durch Remote Procedure Calls (RPC) von einem Rekonstruktions-Bolt auf die sozialen Daten eines anderen Rekonstruktions-Bolt zuzugreifen. Bei der Nutzung dieses Modells könnte die Struktur der bisherigen zentralen Kaskadenrekonstruktion beibehalten werden. Dieses Modell wurde teilimplementiert, dann aber wieder verworfen. Es stellten sich Schwierigkeiten beim Benutzen mehrerer Threads ein. Um einerseits den RPCs antworten zu können und andererseits die normale Rekonstruktion weiterzuführen müssen innerhalb der Rekonstruktions-Bolts mehrere Threads laufen. Das Problem ist, dass das Senden von Informationen bei *Storm* immer nur einem Thread erlaubt ist. Will ein zweiter Thread ebenfalls Senden werden Fehler geworfen.

Zudem muss bei Multithreading auch auf eine threadsichere Umgebung des Systems geachtet werden. Ein bereits existierendes System threadsicher zu machen ist nicht immer trivial. Außerdem müssen um RPCs nutzen zu können die IPs der Server bekannt sein und an die Clients übergeben werden. Die IPs müssten per Liste statisch an die Bolts weitergegeben werden. Eine Nutzung verschiedener Server zöge auch immer eine Änderung der IPs mit sich. RPCs stellten sich daher als ungeeignet für diese Arbeit heraus.

Eine andere Möglichkeit hätte darin bestanden, neben der aktuellen Storm-Topologie eine zweite einzuführen, die nur auf Daten des sozialen Graphen zugreift und diese an die erste Topologie weiterliefert. Eine direkte Kommunikation zwischen Storm-Topologien ist allerdings nicht möglich und so müsste eine Nachrichtenkommunikation außerhalb der Topologien durch externe Speicher stattfinden. Externe Speichermöglichkeiten wurden bei einer Arbeit des Lehrstuhls anhand von Cassandra aber schon evaluiert und als zu langsam eingestuft (siehe Unterabschnitt 2.4.3). Dieses Modell wurde daher ebenfalls nicht gewählt.

Eine weitere Methode ist es schließlich eine verteilte Rekonstruktion innerhalb der bestehenden Topologie zu implementieren. Hierzu muss allerdings die bisherige Struktur der zentralen Kaskadenrekonstruktion geändert werden. Es müssen eventuell neue Bolts eingeführt und zwischen den Bolts muss durch ein Nachrichtenmodell deren Art der Kommunikation bestimmt werden.

Die Vorteile sind, dass die komplette Rekonstruktion innerhalb einer Topologie vollzogen werden kann, ohne externe Technologien, zu verwenden. Dies erleichtert eine Analyse des Systems zur Laufzeit. Außerdem können Nachrichten Topologieintern gesendet und müssen nicht durch externe Nachrichtenpuffer zwischengespeichert werden. Dies ist unter dem Gesichtspunkt einer guten Laufzeit erfreulich.

Schließlich wurde das letztgenannte Modell gewählt um eine verteilte Rekonstruktion zu realisieren.

3.3.3. Entwicklung des verteilten Modells

In diesem Abschnitt wird das Modell der verteilten Rekonstruktion entwickelt und erklärt. Dabei wird zuerst ein generelles Modell gefolgt von einem Nachrichtenmodell entwickelt. Anschließend werden Probleme der Modelle aufgezeigt und schließlich wird ein neues Nachrichtenmodell entwickelt in welches verschiedene Verbesserungen einfließen um die Probleme zu beheben.

3.3.3.1. Generelles Modell

Nachdem der Ausgangspunkt der zentralen Kaskadenrekonstruktion bekannt ist, kann ein generelles Modell für eine verteilte Rekonstruktion entworfen werden.

Fragestellung

Bei einem verteilten Modell der Rekonstruktion einer Retweet-Kaskade müssen folgende Fragen adressiert werden:

1. Wie sollen die Bolts und Spouts auf den Knoten (Rechnern) in einem Netzwerk aufgeteilt werden?
2. Wie erreichen die Retweet-Kaskaden die Knoten auf welchen die meisten Benutzer (mit den geladenen Follower-Listen) der Kaskade liegen?
3. Was ist zu tun wenn ein Benutzer aus einer Retweet-Kaskade nicht auf dem Knoten liegt auf den die Kaskade geleitet wurde?

1. Frage

Wie sollen die Bolts und Spouts auf den Knoten (Rechnern) in einem Netzwerk aufgeteilt werden?

Die Antwort auf diese Frage ist relativ einfach. Sie sollten so aufgeteilt werden, dass der Austausch an Nachrichten im Netzwerk minimal ausfällt. Zunächst ist relativ klar, dass auf jedem Knoten ein *Rekonstruktions-Bolt* laufen muss. Da dieser die Rekonstruktion ausführt und den sozialen Graph lädt, kann eine verteilte Rekonstruktion nur erfolgen wenn jeder Knoten einen *Rekonstruktions-Bolt* besitzt.

Der angeschlossene *Speicher-Bolt* schreibt im Prinzip nur die Ergebnisse der Rekonstruktion auf die Festplatte. Es macht also Sinn auch diesen Bolt auf jedem Knoten auszuführen. Bei nur einem Speicher-Bolt müssten alle Ergebnisse der *Rekonstruktions-Bolts* über das Netzwerk gesendet werden. Zur Handhabung wäre allerdings ein einzelner *Speicher-Bolt* einfacher, da alle Rekonstruktionsergebnisse auf einem einzelnen Knoten liegen würden. Für diese Arbeit wird der *Speicher-Bolt* aber für jeden Knoten repliziert.

Der *Twitterstrom-Spout*, *Koordinator-Bolt* und *Fenster-Bolt* könnten auf getrennten Knoten laufen, allerdings bietet sich auch hier an diese auf dem selben Knoten einzurichten um die Anzahl der Nachrichten im Netzwerk gering zu halten.

Zum Schluss sei noch angemerkt, dass Apache Storm es von Haus aus nicht erlaubt Bolts und Spouts auf bestimmten Knoten zu fixieren. Dies muss durch einen eigenen Scheduler realisiert werden. Für den Moment reicht es jedoch zu wissen, dass dies möglich ist.

2. Frage

Wie erreichen die Retweet-Kaskaden die Knoten auf denen die meisten Benutzer (mit den geladenen Follower-Listen) der Kaskade liegen?

Um diese Frage zu beantworten benötigt die Topologie einen zusätzlichen Bolt. Die ursprüngliche Topologie kann lediglich vom *Fenster-Bolt* aus, an den *Rekonstruktions-Bolt*, Nachrichten verschicken. Es muss also ein Bolt zwischengeschaltet werden, welcher entscheidet wo hin die *Retweet-Kaskaden* geschickt werden, wenn es mehrere *Rekonstruktions-Bolts* gibt. Es wird also ein Bolt mit dem Namen *Routing-Bolt* in die Topologie eingefügt. Dieser muss über die Verteilung des sozialen Graphen auf den Knoten im Rechnernetzwerk Bescheid wissen.

Die Entscheidung wohin eine Kaskade geleitet wird, wird anhand des *urprünglichen Tweeters* einer Kaskade bestimmt. Der ursprüngliche Tweeter ist in jedem Retweet vorhanden und daher ideal um zu bestimmen an welchen *Rekonstruktions-Bolt* die Kaskade geleitet werden soll. Da der *Routing-Bolt* über die Verteilung des sozialen Graphen Bescheid wissen muss, kann er bestimmen auf welchem Knoten sich der ursprüngliche Tweeter befindet. Zudem sollten sich bei einer gelungenen Allokation des sozialen Graphen die meisten anderen Benutzer der Kaskade ebenfalls auf dem selben Knoten befinden.

3. Frage

Was ist zu tun wenn ein Benutzer aus einer Retweet-Kaskade nicht auf dem Knoten liegt auf den die Kaskade geleitet wurde?

Auch eine noch so gute Allokation kann nicht verhindern, dass sich manche Benutzer einer Retweet-Kaskade auf einem anderen Knoten, als dem Knoten des ursprünglichen Tweeters, befinden. Es muss also eine Möglichkeit geschaffen werden wie die Daten des sozialen Graphen (Follower-Listen) bei Bedarf auch ausgetauscht werden können.

1. Eine Möglichkeit wäre es diese Daten über den *Routing-Bolt* zu verschicken. Ein *Rekonstruktions-Bolt* frägt beim *Routing-Bolt* einen bestimmten Benutzer an, welchen er selbst nicht in seinem sozialen Graph hat. Der *Routing-Bolt* schickt die Anfrage an den korrekten Knoten weiter. Dieser antwortet dem *Routing-Bolt*, welcher wiederum dem ursprünglichen Anfrager antwortet.

- Vorteile:
 - Sehr einfach zu implementieren, es müsste lediglich ein Nachrichtenstrom von den *Rekonstruktions-Bolts* zum *Routing-Bolt* hin erzeugt werden.
 - Nachteile:
 - Für eine erfolgreiche Anfrage müssen insgesamt vier Nachrichten geschickt werden.
 - Der ursprüngliche Anfrager weiss nicht ob der Benutzer den er anfragt, sich überhaupt auf einem Knoten im Rechnernetzwerk befindet. Er muss also für *jeden* Benutzer den er nicht in seinem sozialen Graph findet anfragen. Bei jeder nicht erfolgreichen Anfrage müssten ebenfalls wieder mindestens zwei Nachrichten geschickt werden. Die Anfrage und die Absage. Die Absage könnte auch durch einen Timeout erfolgen. Dann wäre es aber möglich, dass Antworten bei einer zu hohen Antwortzeit verloren gehen.
 - Frägt ein *Rekonstruktions-Bolt* während der Rekonstruktion der Retweet-Kaskade einen Benutzer von einem anderen Bolt an und dieser ist momentan selbst mit der Rekonstruktion beschäftigt, wird der Anfrager blockiert. Entscheidet man nicht zu blockieren und nach einem Timeout weiter zu machen können Ergebnisse verloren gehen. Es wäre ebenfalls möglich nicht während der Rekonstruktion die Benutzer anzufragen, sondern davor und die Rekonstruktion für die jeweilige Kaskade erst zu starten, wenn alle Anfragen bearbeitet sind. Dadurch ließe sich dieser Nachteil beheben. Allerdings muss dann das Ende einer Retweet-Kaskade bekannt sein oder nach einem Timeout gestartet werden. Dann können aber Ergebnisse verloren gehen.
2. Eine andere Möglichkeit wäre es jeden *Rekonstruktions-Bolt* über die Verteilung des sozialen Graphen Bescheid wissen zu lassen. Diese könnten dann untereinander Anfragen und Antworten kommunizieren.

- Vorteile:
 - Es müssen maximal zwei Nachrichten für eine erfolgreiche Anfrage versendet werden.
 - Die *Rekonstruktions-Bolts* können gezielt nur Benutzer nachfragen, welche auch im Rechnernetzwerk vorhanden sind.
 - Der Sendeaufwand für Anfragen wird auf mehrere Knoten verteilt.
- Nachteile:

- Durch die redundante Haltung einer Map der Benutzerverteilung des sozialen Graphen, muss in jedem Knoten mehr Hauptspeicher beansprucht werden.
 - Während einer Rekonstruktion können keine Anfragen für andere einkommende Retweets gemacht werden.
 - Der gleiche Nachteil mit dem blockierten Anfrager wie bei erstens.
3. Die letzte Möglichkeit wäre es eine Teilimplementierung des Rekonstruktionsalgorithmus im *Routing-Bolt* auszuführen. Dieser weiß dadurch, welche Anfragen es für eine Retweet-Kaskade von einem *Rekonstruktions-Bolt* geben wird. Er kann diese Anfragen dann direkt an den betroffenen Knoten schicken. Dieser kann dann dem entsprechenden Knoten, welcher die Retweet-Kaskade rekonstruiert, antworten. Der Retweet selbst wird weiterhin ganz normal an den entsprechenden Knoten geroutet.
- Vorteile:
 - Es müssen maximal zwei Nachrichten für eine erfolgreiche Anfrage versendet werden.
 - Bei einer Anfrage wird niemand blockiert.
 - Es können auch Anfragen für einen *Rekonstruktions-Bolt* verteilt werden, der gerade rekonstruiert. Hier sollte erwähnt werden, dass Bolts und Spouts Empfangspuffer und Sendepuffer haben. Dies wird bei Apache Storm über das Protokoll *Netty* realisiert. Ein Bolt der gerade Rekonstruiert kann die Antworten für Anfragen also im Puffer zwischenspeichern.
 - Keine redundante Haltung von Maps, bezüglich Benutzerverteilung, in den Knoten.
 - Als Bonus wäre es sogar theoretisch möglich die Anfragen per Piggyback mit den normalen Retweets zu senden.
 - Nachteile:
 - Eine Rekonstruktion kann erst gestartet werden, wenn das Ende einer Retweet-Kaskade bekannt ist. Alternativ könnte auch nach einem Timeout gestartet werden, dann können aber Ergebnisse verloren gehen.
 - Der *Routing-Bolt* hat einen hohen Sendeaufwand von Nachrichten.

Nun muss eine Entscheidung bezüglich der zu wählenden Variante getroffen werden. Alle Varianten haben gemeinsam, dass sie die Rekonstruktion erst starten können wenn das Ende der Kaskade bekannt ist oder ein Timeout erfolgt.

Die erste Variante hat kaum Vorteile und dazu einen sehr hohen Nachrichtenaufwand. Diese Variante kann daher ausgeschlossen werden.

Die zweite Variante ist interessant in Bezug auf die Lastenverteilung bei Anfragen, kann dafür aber auch keine Anfragen starten wenn rekonstruiert wird.

Die dritte Variante hat im Vergleich zu Variante zwei nur den Nachteil des hohen Sendeaufwands. Da die Auswirkungen des Sendeaufwands schlecht abzuschätzen sind und ansonsten keine Nachteile bestehen, wird die Entscheidung getroffen die dritte Variante mit dem *Routing-Bolt* als Teilimplementierung des Rekonstruktionsalgorithmuses zu verwenden.

Modell

Abbildung 3.6 zeigt nun schließlich ein komplettes Modell der Topologie für die verteilten Rekonstruktion. Ein größere Variante des Modells ist in *Anhang B* zu finden. Das Modell geht von einem Rechnernetzwerk von mindestens drei Knoten aus. Auf Knoten 0 und Knoten 1 sind die duplizierten *Rekonstruktions-* und *Speicher-Bolts*. Es wurde ein *Routing-Bolt* eingefügt, welcher anhand des ursprünglichen Tweeters entscheidet wohin ein Retweet gesendet wird. Dieser hält zusätzlich eine Routing-Tabelle im Hauptspeicher und kennt somit die Verteilung der Benutzer auf den Knoten. Diese stammt aus einer vorherigen Allokation (siehe Abschnitt 3.2 *Allokation*). Zwischen den *Rekonstruktions-Bolts* kann nun ein Austausch von Nachrichten (nach Variante 3 nur Antworten, keine Anfragen) stattfinden. Ein dritter Knoten der nicht explizit gekennzeichnet ist hält den *Twitterstrom-Spout*, *Koordinator-Bolt*, *Fenster-Bolt* und schließlich den *Routing-Bolt*.

3.3.3.2. Erstes Nachrichtenmodell

Da nun das generelle Modell entwickelt ist wird folgend das konkrete Nachrichtenmodell aufgezeigt. Dieses definiert den Inhalt der ausgetauschten Nachrichten und deren Verlauf. Zur besseren Überschaubarkeit werden nur der *Routing-Bolt* und die *Rekonstruktions-Bolts* gezeigt, da der Austausch der verteilten Nachrichten nur zwischen diesen stattfindet. Die *Rekonstruktions-Bolts* befinden sich auf verschiedenen Knoten im Rechnernetzwerk. Sie haben Zugriff auf die Follower-Listen auch wenn dies nicht explizit dargestellt ist. Die *Rekonstruktions-Bolts* beginnen damit den sozialen Graphen zu laden sobald die Topologie startet. Der *Twitterstrom-Spout* beginnt ebenfalls beim Start der Topologie damit Retweets zu versenden.

Die folgenden Bilder und Erklärungen sind beispielhaft für zwei Knoten gezeigt. Das Nachrichtenmodell kann selbstverständlich auf eine beliebige Anzahl Knoten erweitert werden.

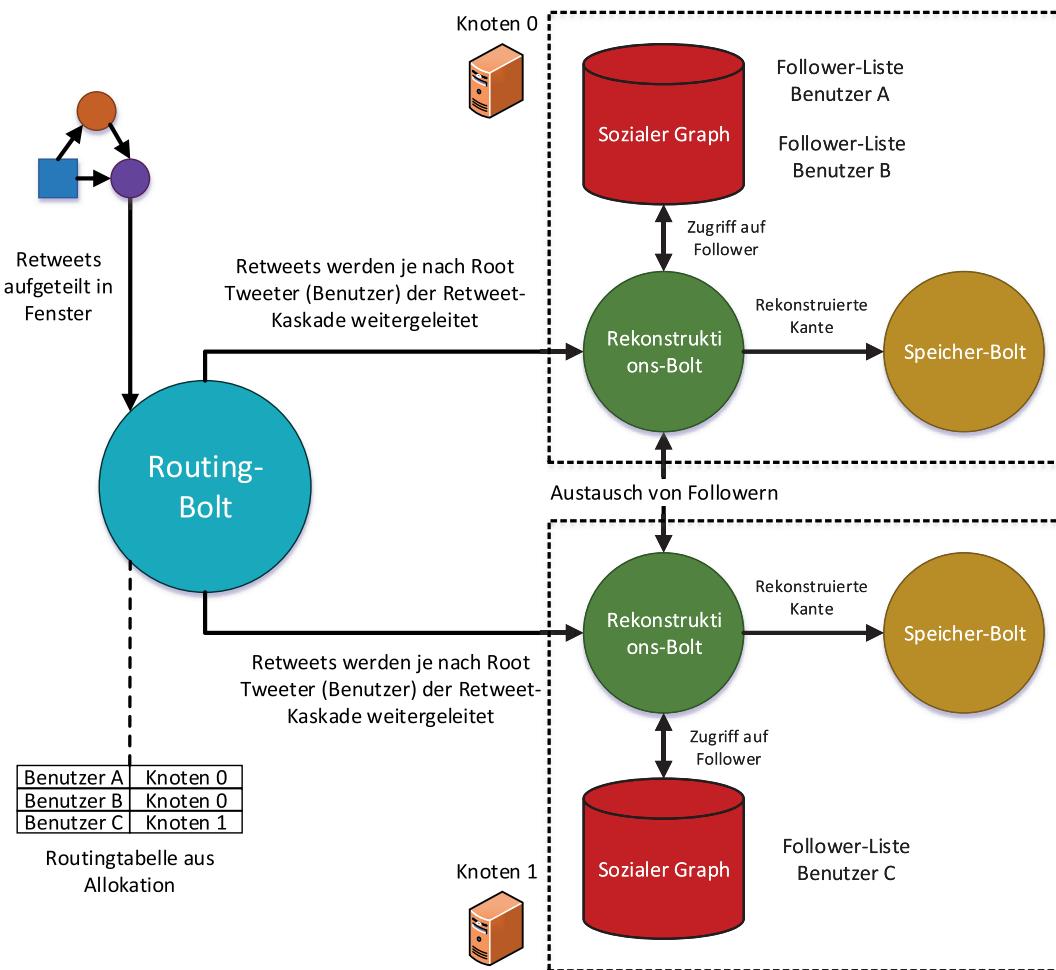


Abbildung 3.6.: Modell Übersicht

Nachrichtenströme

In *Apache Storm* werden Nachrichten entlang von Nachrichtenströmen geschickt. Ein Bolt kann sich für einen bestimmten Nachrichtenstrom einschreiben. Wird kein Nachrichtenstrom angegeben, dann erstellt *Storm* automatisch einen *default* Nachrichtenstrom.

Die Abbildung 3.7 zeigt welche Nachrichtenströme für die verteilte Rekonstruktion neu eingeführt wurden. Auf der Abbildung sind die Namen der Ströme und deren Richtung zu sehen. Die Ströme mit der Kennung 0 und 1 sind die Ströme auf welchen die normalen Retweet-Nachrichten gesendet werden. Die Ströme *Request0* und *Request1* sind Ströme auf welchen die Anfragen nach Followern von Benutzern gesendet werden. Die Ströme *Response0* und *Response1* sind Ströme, auf welchen die jeweiligen Antworten an die anderen Knoten gesendet werden.

Nachrichtentypen

Beim Nachrichtenaustausch gilt es drei neue Typen von Nachrichten zu definieren:

1. **Entfernte Anfrage:** Sie wird vom *Routing-Bolt* zu einem *Rekonstruktions-Bolt* ver-

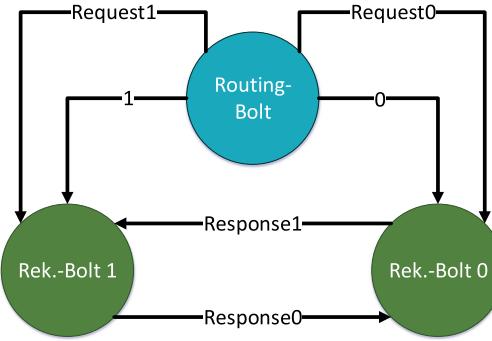


Abbildung 3.7.: Nachrichtenströme für verteilte Rekonstruktion

sendet. Sie hat den Inhalt [**Fenster-Id**, **Remote_Request**, [**Ursprungsstrom**, **Benutzer-Id**, **Entfernte Benutzer-Id**, **Entfernter Benutzer Name**, **Zeitstempel**]]. Die Nachricht hat als Fensterstatus immer den Status *Remote_Request*. Durch das schicken des *Ursprungsstroms* kann der *Rekonstruktions-Bolt* die Antwort später an den richtigen Strom senden. Die *Benutzer-Id* ist die Id des Benutzers, nach welchem die Follower-Liste durchsucht werden soll. Die *entfernte Benutzer-Id* und der *entfernte Benutzer Name* stellen den Benutzer dar dessen Follower-Liste durchsucht werden soll. Es ist also der Benutzer, welcher sich im sozialen Graph des *Rekonstruktions-Bolt* befindet. Der Zeitstempel zeigt an wann die Nachricht vom *Routing-Bolt* abgesendet wurde.

2. **Entfernte Antwort:** Sie wird von einem *Rekonstruktions-Bolt* zu einem anderen *Rekonstruktions-Bolt* versendet. Sie hat den Inhalt [**Fenster-Id**, **Remote_Response**, [**Benutzer-Id**, **Entfernte Benutzer-Id**, **Entfernter Benutzer Name**, **Benutzer enthalten**, **Zeitstempel**]]. Die Nachricht hat als Fensterstatus immer den Status *Remote_Response*. Der empfangende *Rekonstruktions-Bolt* weiß durch den boolschen Wert in „*Benutzer enthalten*“, ob die *Benutzer-Id* in der Follower-Liste des *entfernten Benutzer* gefunden wurde. Ist dies zutreffend fügt der empfangende *Rekonstruktions-Bolt* die *Benutzer-Id* seinem sozialen Graphen hinzu. Sie wird in der Follower-Liste des *entfernten Benutzers* gespeichert.
3. **Entfernte Schließung:** Sie wird vom *Routing-Bolt* zu einem *Rekonstruktions-Bolt* versendet. Sie hat den Inhalt [**Fenster-Id**, **Remote_Close**, [**Gesamtzahl entfernter Nachrichten**]]. Die Nachricht hat als Fensterstatus immer den Status *Remote_Close*. Durch diese Nachricht weiß der *Rekonstruktions-Bolt*, dass eine Kaskade beendet ist und er auf entfernte Antworten warten soll. Er wartet solange bis er soviele Nachrichten erhält, dass sie der *Gesamtzahl entfernter Nachrichten* entsprechen. Dann kann er die Rekonstruktion beginnen.

Die Nachrichten welche vom *Fenster-Bolt* aus an den *Routing-Bolt* geschickt werden bleiben unverändert. Sie haben das gleiche Format wie in der zentralen Kaskadenrekonstruktion.

Erklärung Funktionsweise

Da nun die Nachrichtenströme und Nachrichtentypen definiert sind, können die Nachrichten ausgetauscht werden. Die gezeigten Nachrichten werden in der Form “Nachrichtenfolge. [Nachrichtinhalt]” gezeigt. Die Nachrichtenfolge sagt aus in welcher Reihenfolge die Nachrichten auftreten.

Abbildung 3.8 zeigt den Nachrichtenaustausch wenn alle Retweeter einer Kaskade auf dem selben Knoten liegen. In der ersten Runde wird eine Nachricht mit der Fenster-Id 1, Fensterstatus *Opening* und mit Nutzlast *Retweet1* vom Routing-Bolt empfangen. Das bedeutet, dass die Kaskade mit Fenster-Id 1 neu ist und gerade erstellt wurde. Diese wird im zweiten Schritt an *Rekonstruktions-Bolt 1* weitergeleitet. In der zweiten Runde wird eine Nachricht mit der Fenster-Id 1, Fensterstatus *Consuming* und mit Nutzlast *Retweet2* vom Routing-Bolt empfangen. Dieser Retweet gehört zur Kaskade mit Fenster-Id 1. Er wird nach Empfang direkt weitergeleitet. In der dritten Runde wird eine Nachricht mit der Fenster-Id 1, Fensterstatus *Closing* und mit Nutzlast *Empty* vom Routing-Bolt empfangen. Die Nachricht wird weitergeleitet und *Rekonstruktions-Bolt 1* weiß nun, dass die Kaskade mit der Fenster-Id 1 beendet ist und für diese keine weiteren Retweets mehr eintreffen. Es werden also alle drei Nachrichten einfach über den Strom 1 weitergeleitet.

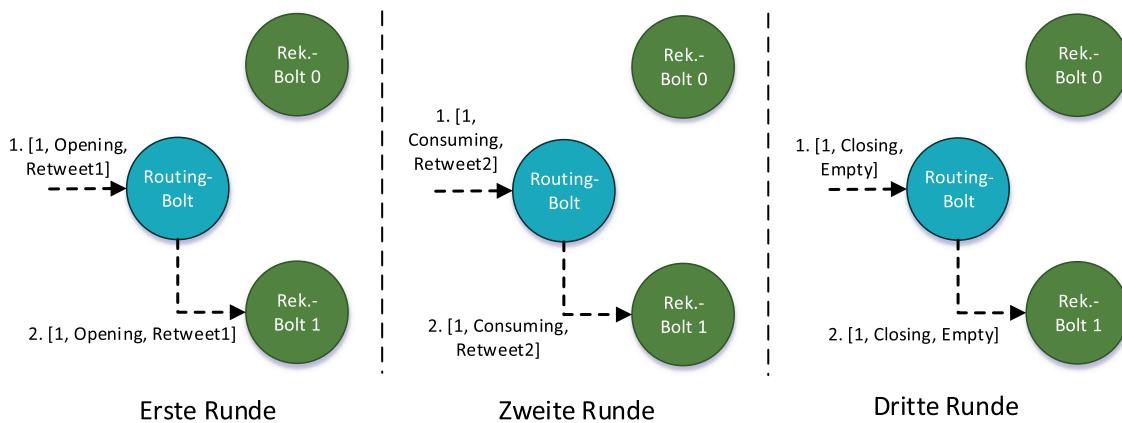


Abbildung 3.8.: Nachrichtenaustausch nur lokal

Nach dem Beispiel für den lokalen Nachrichtenaustausch, zeigt Abbildung 3.9 den entfernten Nachrichtenaustausch. Bei diesem liegt *Retweeter2* im sozialen Graphen von *Rekostruktions-Bolt 0*.

In der ersten Runde passiert nichts besonderes und der Retweet wird weitergeleitet. In der zweiten Runde merkt sich der *Routing-Bolt Retweeter2* vor, da er weiß, dass

Rekonstruktions-Bolt 1 diesen bei seiner Rekonstruktion benötigt aber nicht in seinem sozialen Graphen vorliegen hat.

Er muss nun für **alle** kommenden Retweets der gleichen Kaskade eine Nachricht an *Rekonstruktions-Bolt 0* schicken, welche den jeweils aktuellen Retweeter in der Follower-Liste von *Retweeter2* sucht (dies ist die Teilimplementierung des iterativen Algorithmus).

In der dritten Runde wird eine Anfrage an *Rekonstruktions-Bolt 0* geschickt, welche unter anderem die Ids von *Retweeter2* und *Retweeter3* enthält. Somit kann *Rekonstruktions-Bolt 0* nun prüfen ob *Retweeter3* in der Follower-Liste von *Retweeter2* enthalten ist. Die normalen Retweet-Nachrichten werden nach wie vor an *Rekonstruktions-Bolt 1* weitergeleitet.

In der vierten und letzten Runde schickt *Rekonstruktions-Bolt 0* an *Rekonstruktions-Bolt 1* schließlich eine Antwort. In diesem Fall ist *Retweeter3* in der Follower-Liste von *Retweeter2* enthalten. Daher wird in der Nachricht der boolsche Wert *true* gesendet. Die Nachrichtenabfolge kann für die Antwort innerhalb der vierten Runde nicht bestimmt werden. Sie findet unabhängig von Nachricht 1 und Nachricht 2 statt. Der *Routing-Bolt* schickt in dieser Runde an *Rekonstruktions-Bolt 1* eine schließende Nachricht. Diese enthält die einen Wert der die Gesamtzahl der zuvor geschickten entfernten Anfragen darstellt. Dieser beträgt in dem vorliegen Fall “1”. *Rekonstruktions-Bolt 1* weiß nun, dass er noch auf eine Antwort warten muss. Nach dessen Erhalt kann er mit der Rekonstruktion beginnen.

3.3.3.3. Aufgetretene Probleme

Beim Ausführen größerer Tests des aktuellen Nachrichtenmodells zeigt sich, dass der Hauptspeicher der Knoten überläuft. Zuerst wurde angenommen dies läge hauptsächlich an der Größe des zu ladenden sozialen Graphen. Wie sich aber zeigte lief der Hauptspeicher auch ohne Laden des sozialen Graphen voll. Das Problem scheint zu sein, dass die Senden- und Empfangsnachrichtenpuffer bei einer zu hohen Anzahl an Nachrichten überlaufen und den Hauptspeicher nach und nach füllen. Das unterliegende Nachrichtenprotokoll für Apache Storm ist *Netty*. Normalerweise sollte Netty, wenn der Nachrichtenpuffer voll läuft weitere Nachrichten verwerfen. Warum genau dies nicht passiert konnte im Rahmen dieser Arbeit nicht festgestellt werden. Es wird aber versucht die hohe Anzahl an Nachrichten zu reduzieren.

Die hohe Anzahl an Nachrichten im aktuellen Nachrichtenmodell hat folgende Ursachen:

- Der *Twitterstrom-Spout* wartet nicht darauf, dass der soziale Graph in den Rekonstruktions-Bolts geladen ist, sondern fängt sofort an das Netzwerk mit Nachrichten zu fluten. Die Rekonstruktions-Bolts können die Nachrichten aber erst verarbeiten wenn der soziale Graph geladen ist.

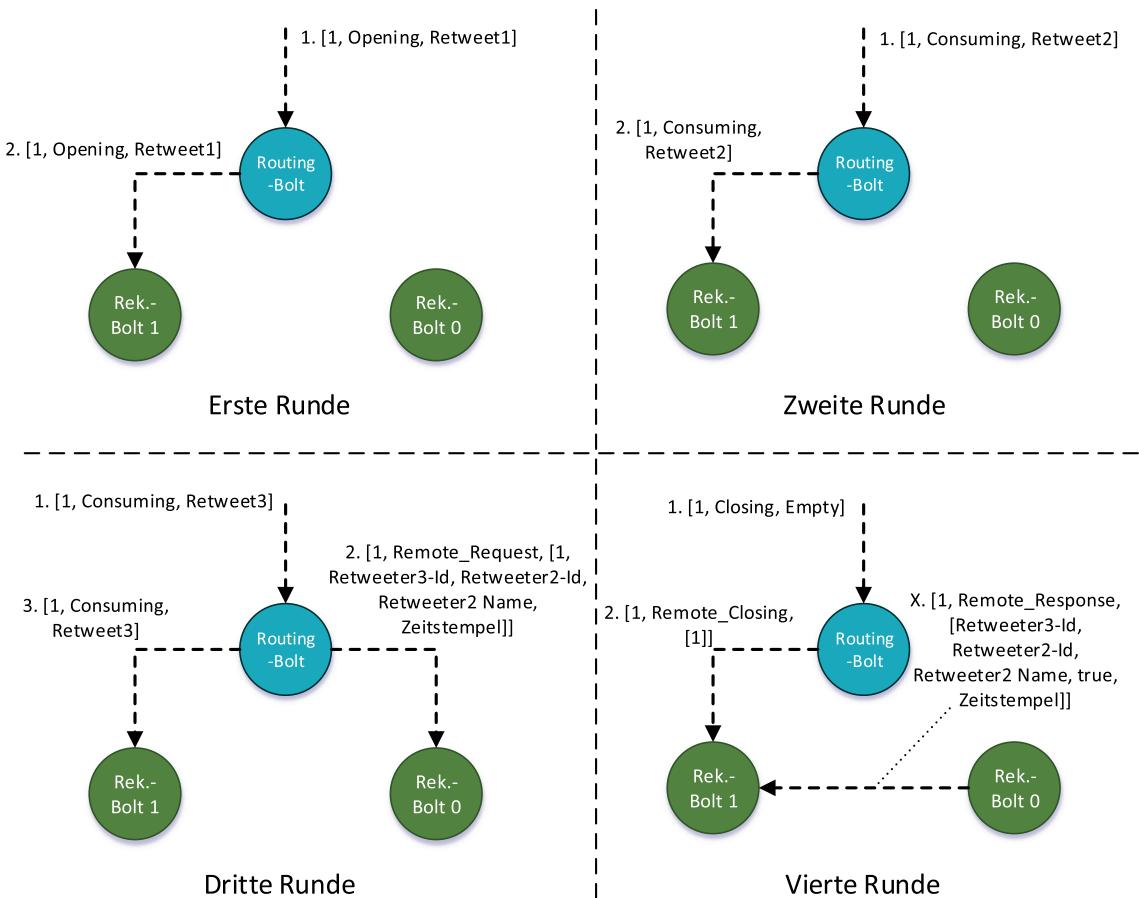


Abbildung 3.9.: Nachrichtenaustausch entfernt

- Der *Routing-Bolt* muss aufgrund der Teilimplementierung des iterativen Algorithmus zuviele Anfragen versenden.
- Die *Rekonstruktions-Bolts* müssen für jede Anfrage eine Antwort mit Zusage oder Absage an den jeweiligen anderen Rekonstruktions-Bolt senden.

3.3.3.4. Verbesserungen

Um die Nachrichtenzahl zu verringern müssen Verbesserung am Nachrichtenmodell vorgenommen werden.

- Um das Fluten des Netzwerks, bevor der soziale Graph von den Rekonstruktions-Bolts geladen wurde, zu verhindern, muss der *Twitterstrom-Spout* geblockt werden. Die Problematik ist dabei, dass ein Spout keine Nachrichten von Strömen empfangen und so normalerweise nicht von den Bolts benachrichtigt werden kann. Durch einen kleinen Trick, mit der Nutzung von *Acks* (für Informationen zu *Ack* siehe Abschnitt 2.2), kann der Spout aber dennoch zuerst am Senden gehindert und nachdem die Rekonstruktions-Bolts bereit sind wieder zum Senden aktiviert werden. Im

Spout wird ein boolscher Wert gesetzt, welcher ihm erlaubt zu senden oder nicht. Der Spout schickt nun gleichzeitig eine Nachricht an alle Rekonstruktions-Bolts, diese ist mit einer bestimmten Id versehen. Danach wird der boolsche Wert auf *false* gesetzt um den Spout am Senden zu hindern. Ein Spout hat außerdem eine Funktion *ack()*, die immer nach dem Empfang von Ack's aufgerufen wird. Dort wird nun der boolsche Wert wieder auf *true* gesetzt wenn von allen Rekonstruktions-Bolts das Ack mit der entsprechenden Id eingetroffen ist. Bekommt der Spout ein *Fail* zurück weil für die Nachricht ein Timeout stattgefunden hat, wird in der Funktion *fail()* die Nachricht erneut geschickt. Ist jedoch das Ack eingetroffen kann der *Twitterstrom-Spout* seine Retweet Nachrichten versenden. Das Nachrichtenmodell muss dafür entsprechend angepasst werden.

- Der *Routing-Bolt* soll nun anstatt für jedes Paar “*Benutzer, entfernte Benutzer*” eine Anfrage zu senden, nur eine gesammelte Anfrage, mit mehreren entfernten Benutzern senden. Dadurch liegt die maximale Anzahl der versendeten Anfragen bei (Anzahl der Retweets) * (die Anzahl der Knoten). Das Nachrichtenmodell muss entsprechend angepasst werden.
- Statt des Sendens jeder Antwort, sollen die *Rekonstruktions-Bolts* nur Nachrichten senden, wenn die Antwort eine Zusage ist. Dies muss ebenfalls im erweiterten Nachrichtenmodell berücksichtigt werden.
- Es besteht zusätzlich die Möglichkeit die Nachrichtenzahl insgesamt zu drosseln. Dazu bietet Storm einen Parameter *max.spout.pending* an. Mit diesem Parameter kann bestimmt werden wieviele Nachrichten der Spout unquittiert versenden darf. Wäre zum Beispiel *max.spout.pending* auf 1 gesetzt, muss der Spout auf das *Ack* aller beteiligten Bolts warten, bevor er eine neue Nachricht senden kann. Um dies zu realisieren werden ab jetzt alle Nachrichten, welche der Spout ausgibt *verankert*. Es zeigt sich aber das selbst bei *max.spout.pending* 1 die Speicher noch voll laufen können, jedoch können insgesamt wesentlich mehr Nachrichten verarbeitet werden bevor dies geschieht. Diese Verbesserung zieht keine Veränderung des eigentlichen Nachrichtenmodells nach sich, sondern muss lediglich bei der Implementierung verbessert werden.

3.3.3.5. Erweitertes Nachrichtenmodell

Nachdem die Verbesserungsmöglichkeiten aufgezeigt wurden, werden diese nun in einem erweiterten Nachrichtenmodell umgesetzt. Es werden weiterhin nur die relevanten Bolts und Spouts in den Abbildungen gezeigt. Es werden wie zuvor nur zwei Knoten mit den jeweiligen *Rekonstruktions-Bolts* zur Erklärung benutzt.

Nachrichtenströme

Um die Verbesserung eines blockierenden *Twitterstrom-Spouts* umzusetzen muss ein zusätzlicher Strom in die Topologie eingefügt werden. Abbildung 3.10 zeigt nun einen Strom *readyRequest*, welcher neu hinzugefügt wurde. Dieser Strom geht vom *Twitterstrom-Spout* direkt auf die *Rekonstruktions-Bolts*.

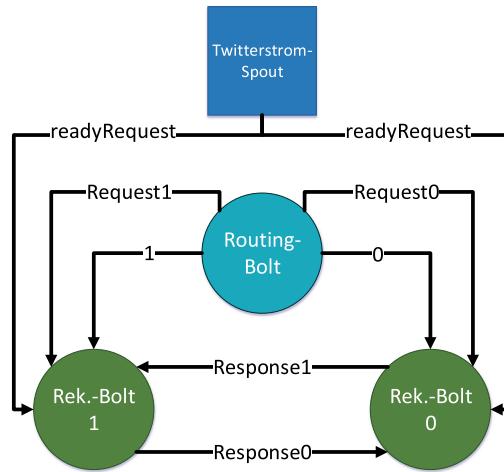


Abbildung 3.10.: Nachrichtenströme für verteilte Rekonstruktion (erweitert)

Nachrichtentypen

Die Nachrichtentypen müssen entsprechend verändert werden. Es muss ebenfalls ein neuer Nachrichtentyp eingeführt werden

1. **Entfernte Anfrage:** Anstatt nur ein entfernter Benutzer wird nun eine Liste der entfernten Benutzer mitgeschickt. Der Inhalt der Nachricht ändert sich nun auf [**Fenster-ID**, **Remote_Request**, [**Ursprungsstrom**, **Benutzer-ID**, **Liste der entfernten Benutzer-IDs**, **Liste der entfernten Benutzer Namen**, **Zeitstempel**]]. Dies ermöglicht es die Anzahl der Anfragen, welche vom *Routing-Bolt* versendet werden, zu reduzieren.
2. **Entfernte Antwort:** Es gibt jetzt ein neues Feld Nachrichtenzähler in der Antwort. Die Nachricht hat nun den Inhalt [**Fenster-ID**, **Remote_Response**, [**Benutzer-ID**, **entfernte Benutzer-ID**, **entfernter Benutzer Name**, **Nachrichtenzähler**, **Benutzer enthalten**, **Zeitstempel**]]. Der Nachrichtenzähler wird benötigt um auch Antworten welche nicht gesendet werden mit zu zählen. Bei der nächsten erfolgreichen Anfrage wird dieser dann mitgesendet. Dies ermöglicht es die Anzahl der Antworten, bei einer guten Allokation, *erheblich* zu reduzieren
3. **Entfernte Schließung:** Sie bleibt im Vergleich zum vorherigen Nachrichtenmodell unverändert.

4. **Entfernter Flush:** Diese Nachricht wird vom *Routing-Bolt* an alle *Rekonstruktions-Bolts* gesendet, welche sich an der verteilten Rekonstruktion einer bestimmten Kaskade beteiligt haben. Ist die letzte Anfrage, innerhalb einer Kaskade, an einen *Rekonstruktions-Bolt* negativ, gibt dieser seinen aktuellen Nachrichtenzähler nie an den empfangenden *Rekonstruktions-Bolt* weiter. Die letzte Antwort muss in so einem Fall erzwungen werden. Dies wird durch einen *entfernten Flush* realisiert. Die Nachricht hat den Inhalt [**Fenster-Id**, **Remote_Flush**, [**Ursprungsstrom**]]. Die Nachricht hat als Fensterstatus immer den Status *Remote_Flush*. Das Feld *Ursprungsstrom* wird benötigt um die Antwort an den korrekten Strom zu versenden.

Erklärung Funktionsweise

Da nun die erweiterten Nachrichtenströme und Nachrichtentypen definiert sind, können die Nachrichten ausgetauscht werden.

Zum Start der Topologie wird zuerst geprüft wann die *Rekonstruktions-Bolts* bereit sind, d.h. den sozialen Graphen geladen haben. Dieser Nachrichtenaustausch zwischen *Rekonstruktions-Bolts* und dem *Twitterstrom-Spout* wird in Abbildung 3.11 dargestellt. Beim Start der Topologie wird eine Nachricht mit der Id *NodesRdyMessage* geschickt. Sind die Bolts bereit so senden diese ein *Ack*. Wenn die Nachricht von einem Bolt nicht "geackt" wird erhält der Spout durch einen Timeout automatisch ein *Fail*. Er verschickt die Nachricht dann nochmals an alle *Rekonstruktions-Bolts*. Haben alle Bolts den sozialen Graphen geladen und *Acks* geschickt, kann der *Twitterstrom-Spout* damit beginnen die Retweet-Kaskaden zu senden.

Abbildung 3.12 zeigt den Nachrichtenaustausch wenn die *Retweeter 2 und 3* im sozialen Graphen des *Rekonstruktions-Bolt 0* liegen. Die Abbildung startet bei Runde vier. Die Runden eins bis drei sind die selben wie bei dem vorherigen Beispiel in Abbildung 3.9. Es wurden also bereits drei normale Retweets versendet und es ging eine Anfrage an *Rekonstruktions-Bolt 0*.

In der vierten Runde werden in der Anfrage nun die entfernten Retweeter 2 und 3 zusammen gesendet. Für diese beiden wird überprüft ob der Retweeter 4 in deren Follower-Liste liegt. Es werden die Namen der Retweeter, wie auch die Ids, zusammen gesendet. In der Antwort von *Rekonstruktions-Bolt 0* an *Rekonstruktions-Bolt 1* ist nun das zusätzliche Feld *Nachrichtenzähler* vorhanden. Dieses steht in dem Beispiel auf "1", da die Antwort direkt nach der Anfrage (die aus der dritten Runde) gegeben wird und somit nur eine Nachricht hochgezählt werden muss. Der normale Retweet wird wie immer weitergeleitet.

Nun wird einen Sprung zur zehnten Runde gemacht. In den dazwischenliegenden Runden wurden Anfragen für die Retweeter 5 - 9 an *Rekonstruktions-Bolt 0* geschickt. Diese sind aber nicht in den Follower-Listen von Retweeter 2 oder 3 enthalten. Der *Routing-Bolt* erhält nun ein *Closing* Fensterstatus für die Kaskade mit Fenster-Id 1. Er sendet

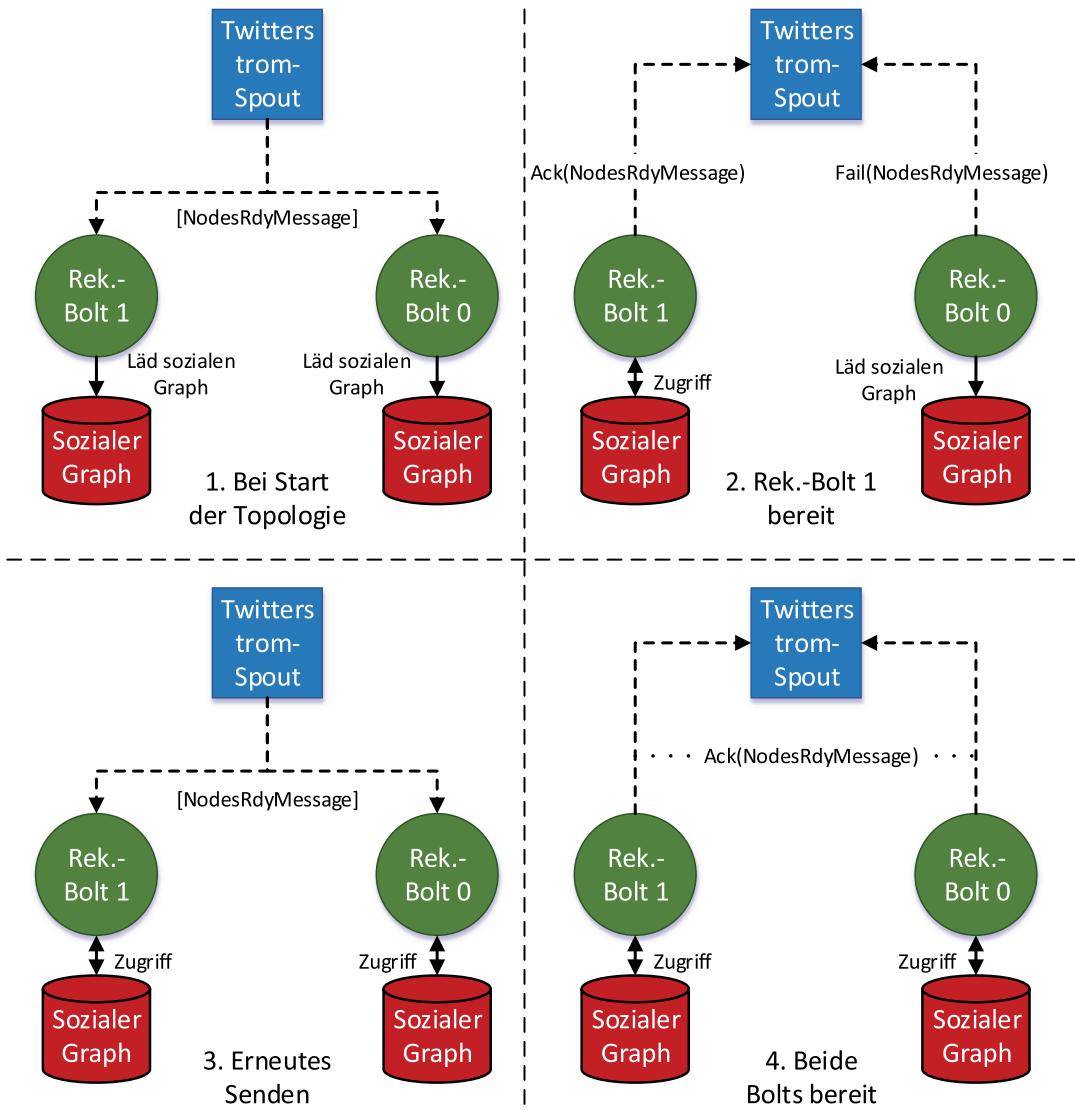


Abbildung 3.11.: Nachrichtenaustausch Twitterstrom-Spout zu Rekonstruktions-Bolts

eine Nachricht mit dem Fensterstatus *Remote_Close*, wie zuvor auch und gibt dieser die Anzahl der zu erwartenden Antworten, von entfernten Anfragen, mit. Diese beträgt hier “7”. Außerdem wird nun, die in diesem Modell neue Nachricht, mit dem Fensterstatus *Remote_Flush* an *Rekonstruktions-Bolt 0* gesendet.

In der elften Runde schickt *Rekonstruktions-Bolt 0* eine Antwort an *Rekonstruktions-Bolt 1*. Die Antwort wurde aufgrund des entfernten Flushs aus Runde zehn veranlasst. Der Wert des Nachrichtenzählers steht auf “6” (es wurden die negativen *entfernten Anfragen* für Retweeter 4 - 9 hochgezählt) und der boolsche Wert steht auf *false*. Alle anderen Werte sind *default* Werte, welche gefüllt werden müssen da die Nachricht eine *entfernte Antwort* ist.

Der *Rekonstruktions-Bolt 1* addiert nun den Nachrichtenzähler zu seinem internen

Nachrichtenzählerstand hinzu. Er kommt auf den Wert “7”, welcher die gesamte Anzahl an entfernten Anfragen darstellt. Dies wurde ihm in der *Remote_Close* Nachricht mitgeteilt. Nun kann er mit der Rekonstruktion der Kaskade, mit der Fenster-Id 1, beginnen.

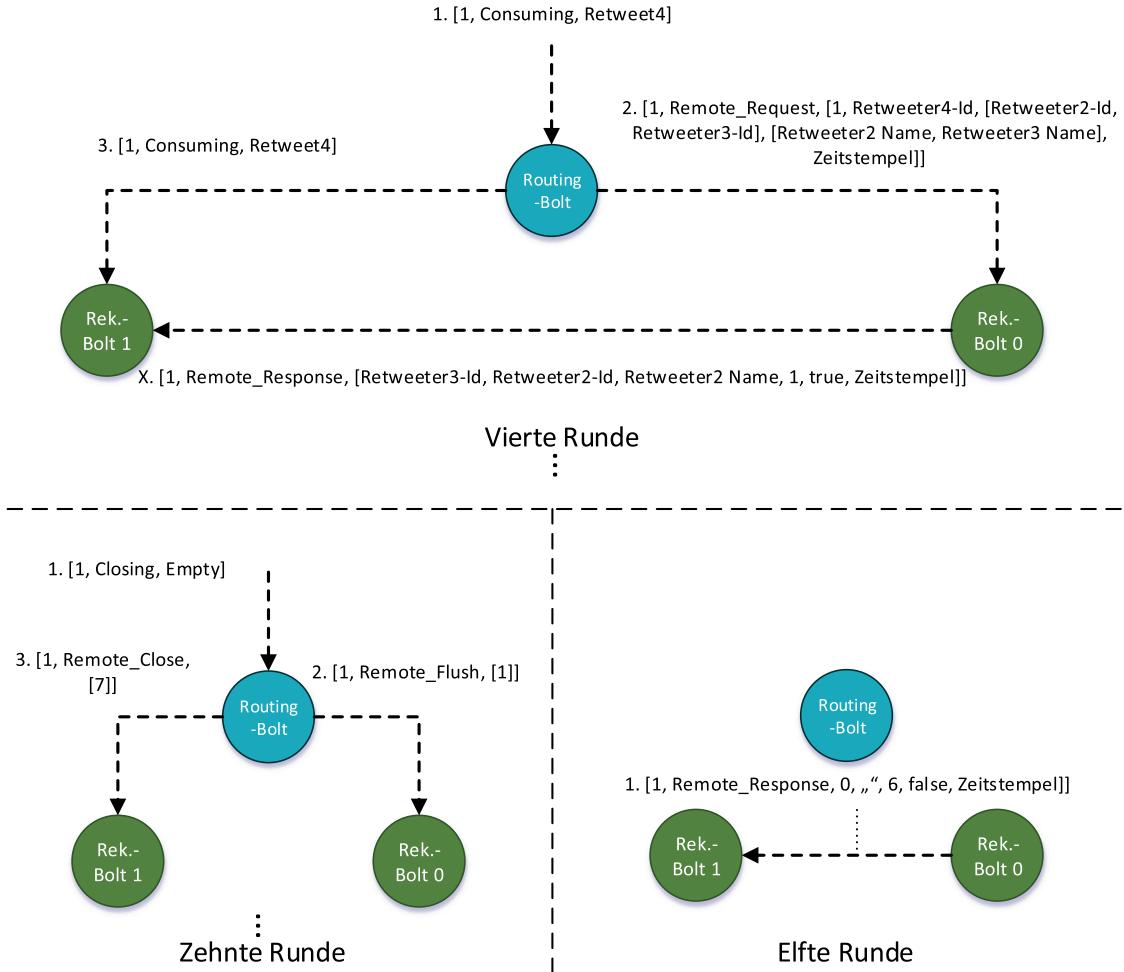


Abbildung 3.12.: Erweitertes Nachrichtenmodell

4. Versuche

In diesem Kapitel werden die Ergebnisse der Versuche gezeigt, welche zur verteilten Rekonstruktion von Informationskaskaden, im Rahmen der Arbeit gemacht wurden. Dabei werden in Abschnitt 4.1 *Untersuchte Kaskaden* zuerst die verwendeten Datensätze (Retweet-Kaskaden) beschrieben. Danach werden in Abschnitt 4.2 *Laufzeiten* die Laufzeiten der Kaskaden bei einer verteilten Rekonstruktion untersucht. Folgend werden in Abschnitt 4.3 *Entfernte Aufrufe* die einzelnen Kaskaden auf die Anzahl ihrer entfernten Aufrufe überprüft und mit den zu Erwartenden verglichen. Zum Schluss werden in Abschnitt 4.4 *Ladezeiten und Speicherverbrauch Sozialer Graph* noch die Ladezeiten und der Speicherverbrauch des sozialen Graphen überprüft und mit den Annahmen aus der Allokation verglichen.

Die Basis für die Verteilung des sozialen Graphen sind die Ergebnisse der Allokation aus Unterabschnitt 3.2.4. Die Versuche werden in einem Rechnernetzwerk ausgeführt, welches sich in einem lokalen Netz befindet. Die Rechner auf welchen die verteilte Rekonstruktion ausgeführt wird hat dabei folgende Spezifikationen:

- CPU: 1.86 GHz mit 12 Kernen
- Hauptspeicher (RAM): 32 Gigabyte
- Betriebssystem: Linux 3.13.0-39-generic (x86_64)

Der Aufbau der Infrastrukturaufbau des Rechnernetzwerks wird in *Anhang A* genauer gezeigt und kann bei einem Versuchsnachbau helfen.

Die Versuche wurden für einen, zwei, vier und acht Knoten durchgeführt, auf welchen *Rekonstruktions-Bolts* und *Speicher-Bolts* liefen. Auf einem separaten Knoten liefen dabei immer der *Twitterstrom-Spout*, der *Koordinator-Bolt*, der *Fenster-Bolt* und der *Routing-Bolt*. Die Versuche mit einem Knoten sind in etwa vergleichbar mit einer zentralen Kaskadenrekonstruktion, da dort nur ein minimaler Nachrichtenaustausch (Weiterleitung der Retweets) stattfindet. Außerdem wurden alle Versuche zweimal durchgeführt. Für alle folgenden Auswertungen wurden die jeweiligen Durchschnittswerte der Versuche verwendet.

4.1. Untersuchte Kaskaden

Alle untersuchten Kaskaden stammen aus dem Olympia 2012 Datensatz. Die folgenden Retweet-Kaskaden wurden für die Versuche herangezogen:

- Einzelne Kaskaden verschiedener Größe:
 - Eine Kaskade mit 3.500 Retweets sie wird mit dem Namen “*Kleiner*” identifiziert.
 - Eine Kaskade mit 7.226 Retweets sie wird mit dem Namen “*Klein*” identifiziert.
 - Eine Kaskade mit 14.847 Retweets sie wird mit dem Namen “*Mittel*” identifiziert.
 - Eine Kaskade mit 29.783 Retweets sie wird mit dem Namen “*Groß*” identifiziert.
 - Eine Kaskade mit 60.451 Retweets sie wird mit dem Namen “*Größte*” identifiziert. Dies ist die größte Kaskade, welche für den Olympia 2012 Datensatz, zum Zeitpunkt der Anfertigung der Arbeit, zur Verfügung steht.
- Mehrere Kaskaden mit einer ungefähren Größe von 60.000 Retweets:
 - 500 Kaskaden mit einer Anzahl Retweets pro Kaskade zwischen 113 und 128. Diese werden mit dem Namen “*500Kask*” identifiziert. Die gesamte Anzahl Retweets beträgt 60.050.
 - 100 Kaskaden mit einer Anzahl Retweets pro Kaskade zwischen 569 und 643. Diese werden mit dem Namen “*100Kask*” identifiziert. Die gesamte Anzahl Retweets beträgt 60.140
 - 10 Kaskaden mit einer Anzahl Retweets pro Kaskade zwischen 5356 und 6971. Diese werden mit dem Namen “*10Kask*” identifiziert. Die gesamte Anzahl Retweets beträgt 59.893.
- Gemixte Kaskaden bestehend aus *500Kask*, *100Kask*, *10Kask* und *Größte*. Diese haben insgesamt 611 Kaskaden und eine gesamte Anzahl Retweets von 240.534. Sie werden mit dem Namen “*Gemixt*” identifiziert.

Eine Auflistung der Kaskaden-Ids, zur Referenz, findet sich in *Anhang C*. Es wurde auf die Auflistung von *500Kask* verzichtet.

4.2. Laufzeiten

In diesem Abschnitt werden die Kaskaden auf ihre Laufzeiten überprüft. Generell ist zu erwarten, dass die Laufzeit bei den Versuchen mit einzelnen Kaskaden mit einem Knoten besser ist als mit mehreren Knoten. Dies ist zu erwarten da ein Knoten alle Follower-Listen (sozialer Graph) vorliegen hat, während mehrere Knoten die entsprechenden Follower erst über Nachrichten kommunizieren müssen. Dabei ist vor allem die Höhe der Differenz in den Laufzeiten interessant.

Bei Versuchen mit mehreren Kaskaden insbesondere bei *Gemixt* wird erhofft, dass eine parallele Rekonstruktion der Retweet-Kaskaden auf mehreren Knoten eine Laufzeitverbesserung gegenüber der Rekonstruktion auf einem einzelnen Knoten erbringt.

Für die Versuche werden folgende Zeiten angegeben:

- **Gesamtzeit:** Sie stellt die komplette Zeit dar, welche die verteilte Rekonstruktion benötigt. Der Startzeitpunkt stellt dabei den Empfang des ersten Retweets auf einem der Knoten dar. Der Endzeitpunkt stellt dabei das Ende der Rekonstruktion der letzten Kaskade auf einem der Knoten dar.
- **Rekonstruktionszeit:** Für jede Kaskade wird separat die Zeit der benötigten Rekonstruktion gemessen. Die Rekonstruktionszeit stellt die Summe aller dieser Zeiten, über alle Knoten, dar. Die Rekonstruktionszeit würde also bei einer komplett parallel verlaufenden Rekonstruktion der Kaskaden eine Zeit von (*Gesamtzeit mal Anzahl der Knoten*) ergeben.
- **Nachrichtenzeiten:** Diese Zeit wurde nicht gemessen sondern wird aus (*Gesamtzeit minus Rekonstruktionszeit*) errechnet. Die Grundlage hierfür ist, dass Knoten in der Zeit welche nicht von der Rekonstruktion beansprucht wird, Nachrichten austauschen. Wird diese Zeit negativ ist es ein Hinweis darauf, dass die Zeiteinsparung einer parallelen Rekonstruktion den Zeitverbrauch zum Senden von Nachrichten übersteigt.

Um die Rekonstruktion der Kaskaden selbst durchzuführen wird auf den Rekonstruktions-Bolts nach wie vor der iterative Hash-Algorithmus der zentralen Rekonstruktion verwendet (siehe Unterabschnitt 2.4.1). Dabei wurde der Füllfaktor der Hashmaps auf 0.5 von 0.8 erhöht (dieser Faktor bestimmt wie voll eine Hashmap sein kann bevor sie zusätzlichen Speicher allokiert). Dies war nötig um einen größeren sozialen Graphen in den Hauptspeicher laden zu können. Insbesondere da die ersten Tests nur auf einem kleineren Rechnernetzwerk, bei dem jeder Rechner nur 8 Gigabyte Hauptspeicher hatte, ausgeführt wurden. Der Füllfaktor wurde dann über die Arbeit hinweg beibehalten.

Bei den Versuchen für einen und zwei Knoten passt der soziale Graph nicht in den Hauptspeicher. Daher werden aus den Retweet-Kaskaden die Ids der Retweeter extrahiert und dann nur die Retweeter in den sozialen Graph geladen, welche bei der Rekonstruktion benötigt werden. Um einen gleichen Versuchsaufbau zu erhalten wird dies schließlich auch bei vier und acht Knoten beibehalten.

4.2.1. Einzelne Kaskaden

Die Abbildung 4.1 zeigt die Gesamtaufzeiten der Kaskaden *Kleiner*, *Klein*, *Mittel*, *Groß* und *Größte* im Vergleich. Die Datengrundlage der Abbildung ist in Tabelle 4.1 zu finden. Jede Linie in der Abbildung steht für eine andere Anzahl an Knoten. Man sieht sofort, dass die Linien sehr eng zusammen liegen. Die Laufzeiten scheinen sich also nach Anzahl der Knoten nur gering zu unterscheiden und dies bei jeder Kaskadengröße. Eine Ausnahme bildet dabei *Mittel*, welche für acht und vier Knoten deutlich mehr Laufzeit benötigt als für zwei und einen Knoten.

Die genauen Werte in Tabelle 4.1 zeigen also, dass die Laufzeit der verteilte Rekonstruktion einzelner Kaskaden, bei abnehmender Knotenzahl, nur moderat ansteigt. Selbst bei der größten Kaskade beträgt der Unterschied zwischen einem und acht Knoten nur 27.214 Millisekunden. Dies ist bei einer Gesamtaufzeit von 586.096 Millisekunden noch hinnehmbar. Die verteilte Rekonstruktion scheint hier also gut zu funktionieren.

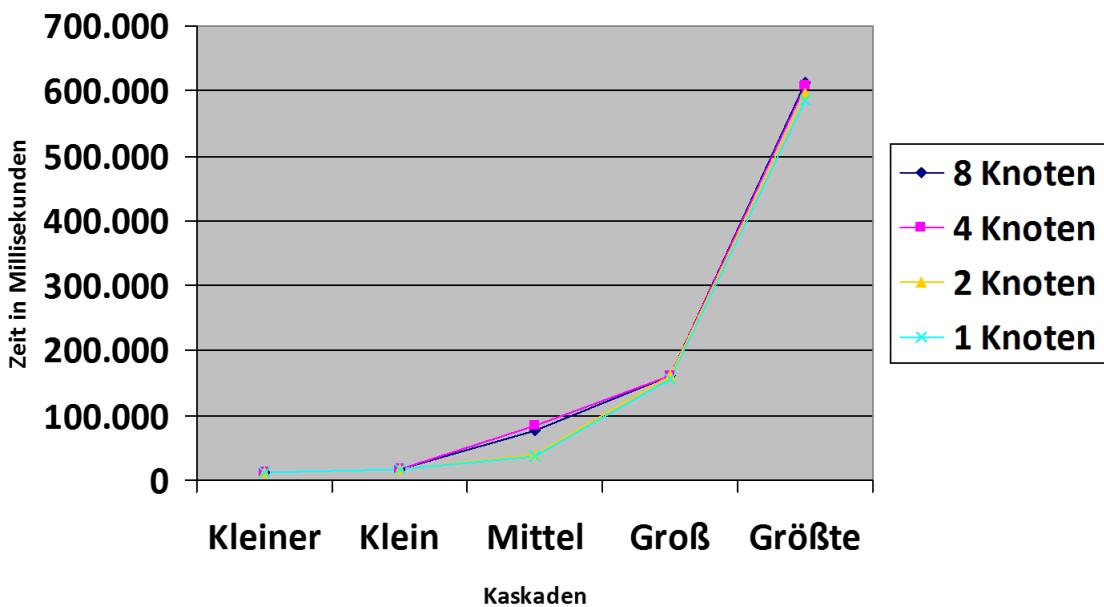


Abbildung 4.1.: Laufzeit einzelne Kaskaden

Um die Abweichungen von *Mittel* zu verstehen wird in Tabelle 4.2 neben der Gesamtzeit auch die Rekonstruktions- und Nachrichtenzeit gezeigt. Dort zeigt sich, dass die Nachrichtenzeit bei acht Knoten deutlich höher ist als bei den anderen Knotenzahlen.

Tabelle 4.1.: Laufzeit einzelner Kaskaden in Millisekunden

Knoten	Kleiner	Klein	Mittel	Groß	Größte
1	11.866	17.424	38.113	156.488	586.096
2	12.288	17.615	40.117	160.973	595.505
4	12.293	17.719	83.461	160.328	607.089
8	12.469	18.295	83.623	162.099	613.310

tenzeiten für vier und acht Knoten sich von einem und zwei Knoten um einen Faktor von ca. 5 unterscheiden. Es scheint, dass hier die Partitionierung des sozialen Graphen nicht funktioniert hat und darum viele Nachrichten versendet werden mussten. Diese These wird später in Abschnitt 4.3 überprüft.

Interessant ist auch, dass die Rekonstruktionszeit von vier und acht Knoten um ca. 13.000 Millisekunden besser ist als bei einem und zwei. Dies ist überraschend da vermutet wurde, dass die Rekonstruktionszeiten des iterativen Hash Algorithmus auch bei unterschiedlicher Knotenzahl gleich bleiben. Eine Analyse zu diesem Thema wird später in Unterabschnitt 4.2.4 *Abschließende Analyse* gegeben.

Tabelle 4.2.: Rekonstruktionszeit vs Nachrichtenzeit für *Mittel* in Millisekunden

Knoten	Gesamt	Rekonstruktion	Nachrichten
1	38.113	24.161	13.952
2	40.117	25.445	14.672
4	83.461	11.937	71.524
8	83.623	12.071	71.552

Zum Vergleich zu *Mittel* wird auch *Größte* auf Nachrichten- und Rekonstruktionszeit analysiert. Die Zahlen dazu sind in Tabelle 4.3 zu finden. Es zeigt sich dort, dass die normalen Erwartungen alle erfüllt werden. Die Nachrichtenzeit steigt mit steigender Knotenzahl und die Rekonstruktionszeit ist ungefähr gleichbleibend. Aber auch hier kann bei acht Knoten die niedrigste Rekonstruktionszeit verzeichnet werden.

Tabelle 4.3.: Rekonstruktionszeit vs Nachrichtenzeit für *Größte* in Millisekunden

Knoten	Gesamt	Rekonstruktion	Nachrichten
1	586.096	569.233	16.863
2	595.505	570.356	25.149
4	607.089	569.576	37.513
8	613.310	557.543	55.767

4.2.2. 60.000 Retweets

Bei diesem Versuch wird verglichen wie sich die verteilte Rekonstruktion bei einer gleichbleibenden Zahl an Retweets aber einer unterschiedlichen Zahl an Kaskaden verhält.

Die Abbildung 4.2 zeigt die Gesamtlaufzeiten der Datensätze *500Kask*, *100Kask* und *10Kask*. Die Datengrundlage der Abbildung ist in Tabelle 4.4 zu finden. Jede Linie in der Abbildung steht für eine andere Anzahl an Knoten. Man sieht in der Abbildung, dass die Laufzeiten für verschiedene Knotenzahlen bei *500Kask* und *100Kask* sehr nah zusammen liegen. Die Laufzeiten bei einem und zwei Knoten sind aber besser als bei vier und acht. Dies lässt darauf schließen, dass eine parallele Konstruktion zumindest bei dieser Anzahl an Kaskaden und Retweets nicht stattgefunden hat oder keine Vorteile brachte. Bei *10Kask* zeigt sich dann ein anderes Bild. Dort unterscheidet sich die Gesamtlaufzeit für einen Knoten doch merklich von den Gesamtlaufzeiten der anderen Knoten.

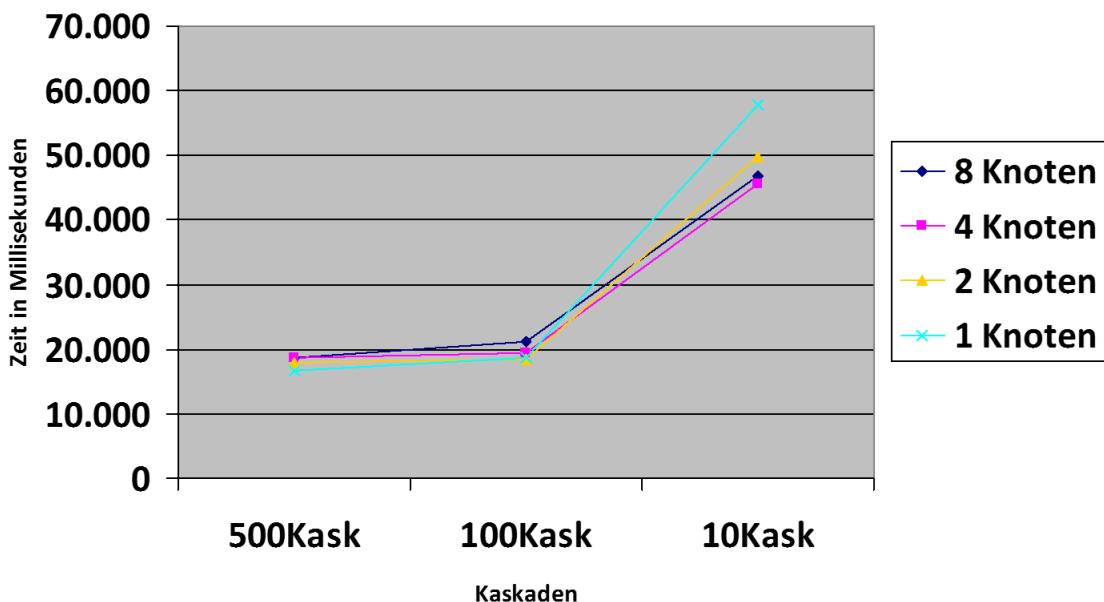


Abbildung 4.2.: Laufzeit 60.000 Retweets

Tabelle 4.4.: Laufzeit 60.000 Retweets in Millisekunden

Knoten	500Kask	100Kask	10Kask
1	16.678	18.606	57.738
2	18.038	18.381	49.760
4	18.638	19.358	45.491
8	18.696	21.220	46.711

Der Datensatz *10Kask* wird daher genauer auf die Rekonstruktions- und Nachrichtenzeit analysiert. Die Zahlen dazu sind in Tabelle 4.5 zu sehen. Es zeigt sich, dass die

Rekonstruktionszeit ein Hauptfaktor für die erhöhte Gesamtaufzeit von einem Knoten ist. Eine Analyse zu diesem Thema wird aber später in Unterabschnitt 4.2.4 *Abschließende Analyse* gegeben. Es kann aber auch eine erhöhte Nachrichtenzeit im Vergleich zu den anderen Knotenzahlen vermerkt werden. Dies ist durchaus verwunderlich, da hier für einen Knoten normalerweise die geringste Zeit erwartet wird. Leider konnte auch nach einer eingehenden Untersuchung der Zeiten der einzelnen Kaskaden innerhalb des Datensatzes *10Kask* keine befriedigende Antwort darauf gefunden werden. Es fällt aber auch auf, dass dieser Datensatz generell unregelmäßige Ergebnisse liefert. Möglicherweise war das gesamte Netzwerk des Versuchsaufbaus bei diesem Test unregelmäßig belastet.

Tabelle 4.5.: Rekonstruktionszeit vs Nachrichtenzeit für *10Kask* in Millisekunden

Knoten	Gesamt	Rekonstruktion	Nachrichten
1	57.738	39.579	18.159
2	49.760	35.813	13.947
4	45.491	36.111	9.380
8	46.711	34.074	12.637

4.2.3. Gemixte Kaskaden

Bei diesem Versuch wird das Verhalten der verteilten Rekonstruktion unter einer hohen Anzahl verschiedener Kaskaden und einer hohen Anzahl gesamter Retweets gezeigt.

Die Abbildung 4.3 zeigt die Gesamtaufzeit und die Rekonstruktionszeit des Datensatzes *Gemixt*. Die Datengrundlage der Abbildung ist in Tabelle 4.6 zu finden. Auf der X-Achse befinden sich hierbei diesmal die jeweilige Anzahl der Knoten. Es ist gut zu sehen, dass die Gesamtaufzeit von einem Knoten bis hin zu vier Knoten abnimmt. Von Knoten vier auf Knoten acht steigt die Gesamtaufzeit dann wieder minimal.

Der Grund für die anfängliche Abnahme der Gesamtaufzeit ist die Rekonstruktionszeit, welche ebenfalls, je Knotenzahl, abnimmt. Für die steigende Tendenz zwischen vier und acht Knoten ist eine erhöhte Nachrichtenzeit verantwortlich. Dies ist daran zu erkennen, dass die Rekonstruktionszeit auch bei acht Knoten im Vergleich zu vier Knoten abnimmt, aber die Gesamtaufzeit trotzdem steigt. Die Fläche zwischen den beiden Linien kann sozusagen als Nachrichtenzeit gesehen werden.

Das Ergebnis scheint vorerst durchaus erfreulich. Allerdings ist die sinkende Gesamtaufzeit bei mehreren Knoten, der sinkenden Rekonstruktionszeit zuzuschreiben und nicht einer erwarteten parallel laufenden Rekonstruktion. Wieso dies so sein könnte wird im nächsten Abschnitt erläutert.

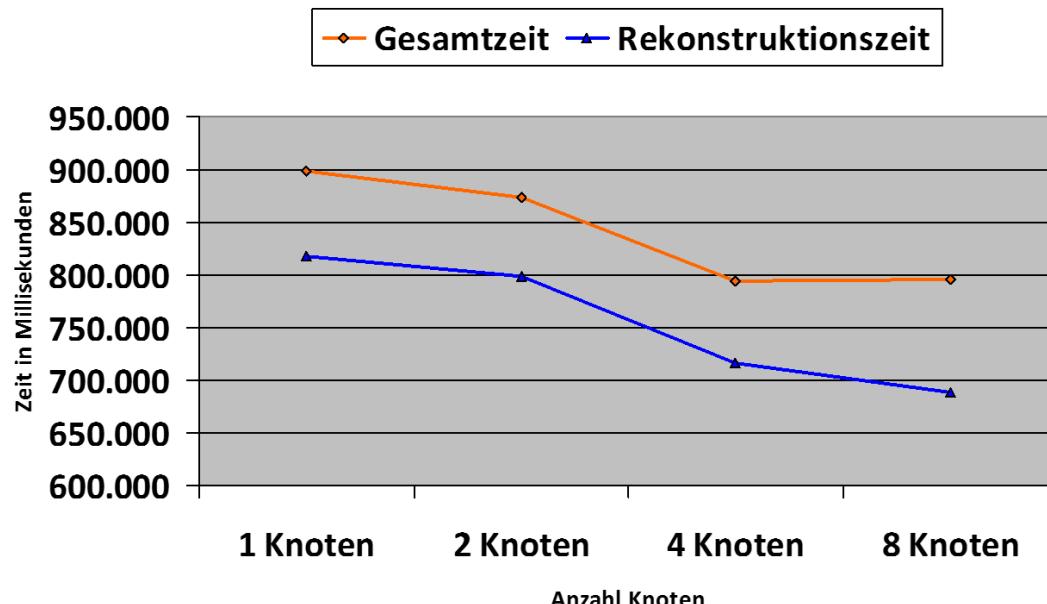


Abbildung 4.3.: Laufzeit gemixte Kaskaden

Tabelle 4.6.: Laufzeit gemixte Kaskaden in Millisekunden

Knoten	Gemixt	Rekonstruktion	Nachrichten
1	898.775	818.192	80.583
2	873.085	797.988	75.097
4	794.692	716.907	77.785
8	795.968	688.311	107.657

4.2.4. Abschließende Analyse

Bei den *einzelnen Kaskaden* wurden die Erwartungen an die Laufzeiten größtenteils erfüllt. Mit steigender Knotenzahl stieg auch die Laufzeit. Dies aber in einem sehr moderaten Maß.

Eine parallele Rekonstruktion der Kaskaden konnte nicht festgestellt werden. Es wird allerdings davon ausgegangen, dass dieser Vorteil erst bei einer noch höheren Anzahl an Retweets und Kaskaden zu sehen ist. Jedoch zeigte sich über alle Versuche hinweg, dass eine Erhöhung der Knotenzahl mit einer Senkung der Rekonstruktionszeit dahergeht. Dies ist vorerst verwunderlich, da davon ausgegangen wurde, dass der iterative Algorithmus über die Knoten hinweg eine gleibleibende Rekonstruktionszeit zeigt. Es scheint aber als sei der Füllfaktor der Hashmap von 0,8 welche für die Versuche verwendet wurde dafür ausschlaggebend.

Ein Füllfaktor von 0,8 liegt am oberen Rand der für ein Open-Addressing-Verfahren (siehe [4]) verwendet werden sollte. Die Zeit um einen Eintrag in einer Follower-Liste zu finden könnte also potentiell hoch sein (es muss oft sondiert werden bis der Eintrag gefunden ist).

den ist). Ist nun der soziale Graph verteilt, werden nur einzelne Einträge für bestimmte Retweeter im sozialen Graphen des Rekonstruktions-Bolts abgelegt. Für diese Retweeter muss der iterative Hash-Algorithmus meist nur eine Follower-Liste in einstelliger Größe durchsuchen. Sind dagegen die Follower-Listen aller Retweeter komplett gefüllt, geht das finden des Eintrags entsprechend länger. Durch die Verteilung des sozialen Graphen wurde das Sondieren für den iterativen Hash-Algorithmus also einfacher. Die Vermutung ist nun, dass dies der Hauptgrund für die sinkenden Rekonstruktionszeiten ist. Dies kann aber im Rahmen der Arbeit leider nicht mehr validiert werden.

Interessant wäre es zu sehen wie sich der ursprüngliche Loadfaktor von 0,5 auswirkt. Es wird vermutet, dass selbst dann noch die Rekonstruktionszeiten bei steigender Knotenzahl sinkt, aber vermutlich in einem moderateren Maß. Vor allem die Laufzeit eines Knoten mit einem Füllfaktor von 0,5 im Vergleich zu acht Knoten mit einem Füllfaktor von 0,8, bei einem Versuch mit *Gemixt* scheint interessant. Läge die Laufzeit bei acht Knoten dann immernoch tiefer als die Laufzeit bei einem Knoten, kann durch die verteilte Rekonstruktion eine bessere Laufzeit mit einem höheren Füllfaktor und so mehr geladenen Follower-Listen pro Knoten erreicht werden.

In jedem Fall zeigt die verteilte Rekonstruktion bei der vorliegenden Partitionierung des sozialen Graphs nur geringe Nachteile bei den Laufzeiten. Durch sinkende Rekonstruktionszeiten konnten für mehrere Knoten, z.B. bei *Gemixt*, sogar bessere Laufzeiten als für einen Knoten verzeichnet werden. Wie stark dieser Faktor bei einem geringeren Füllfaktor der Hashmap noch zu sehen ist, kann aber nicht abgeschätzt werden.

4.3. Entfernte Aufrufe

Hier werden die Versuchsergebnisse bezüglich entfernter Aufrufe überprüft. Als ein *entfernter Aufruf* wird jeder Retweeter einer Kaskade gezählt, dessen sozialer Graph nicht auf dem Knoten liegt, auf welchem die Kaskade rekonstruiert wird.

In Unterabschnitt 3.2.4 wurden die Ergebnisse der Allokation präsentiert. Unter anderem wurde auch das Verhältnis der Kanten zwischen den Knoten zu den gesamten Kanten des Graphs gezeigt. In diesem Versuch ist es nun interessant zu sehen ob sich dieses Verhältnis auch in der Anzahl entfernter Aufrufe, zur Anzahl lokaler Aufrufe, wiederfindet. Es wird erwartet das dies der Fall ist.

Folglich zeigt Tabelle 4.7 auf Seite 62 drei Untertabellen, in welchen für jeweils 8, 4 und 2 Knoten die entfernten Aufrufe untersucht werden. Die Untertabellen haben dabei die folgenden Spalten:

- **Kaskaden:** Name des untersuchten Datensatzes.

- **Gesamt:** Die gesamte Anzahl der Aufrufe von Retweetern (lokal und entfernt). Dies entspricht der Anzahl an Retweets.
- **Entfernt:** Die Anzahl der Aufrufe, welche auf entfernte Knoten gemacht werden.
- **Verhältnis:** Verhältnis der entfernten Aufrufen zur gesamten Anzahl an Aufrufen.
- **Vergl. zu Allokation (x %):** Der Vergleich vom Verhältnis (*Anzahl entfernte Aufrufe/Anzahl Aufrufe gesamt*) bei der Rekonstruktion, zum Verhältnis (*Anzahl gewichteter Kanten zwischen Knoten / Anzahl gesamter gewichteter Kanten im Graph*) in der Allokation. Der Wert x ist dabei die Basis, welche bei der Allokation, für die jeweilige Anzahl an Knoten, ermittelt wurde.

Es ist zu beachten, dass bei *100Kask*, fünf Kaskaden zufällig verteilt und bei *500Kask* 34 zufällig verteilt wurden. Das bedeutet, dass die ursprünglichen Tweeter der Kaskade nicht im Aktivitätsgraphen enthalten waren. Die Anzahl der entfernten Aufrufe dieser Kaskaden variierte demnach geringfügig um bis zu maximal 80 Aufrufen.

Es zeigt sich, dass das Verhältnis von *Entfernt* zu *Gesamt* für fast alle untersuchten Kaskaden bei allen Knotenzahlen höher liegt, als es aufgrund der Allokation erwartet wurde. Die einzigen Kaskaden welche durchgehend ein besseres Verhältnis haben sind *Groß* und *Größte*. Demnach scheint die Partitionierung des Graphen und die darauffolgende Allokation der Graphpartitionen, vor allem für sehr große Kaskaden, gut zu funktionieren. Für die kleineren Kaskaden liegen die Vergleichswerte bei acht Knoten zwischen 3,7 % und 18,6%, bei vier Knoten 7,6 % und 13,5 % und bei zwei Knoten zwischen 5,9 % und 9,2 %. Das Verhältnis wird also, im Vergleich zur Allokation, mit der abnehmenden Anzahl Knoten besser. Insgesamt sind die Verhältnisse aber vertretbar.

Eine Ausnahme der bisherigen Analyse der Ergebnisse ist die Kaskade *Mittel*. Sie zeigt bei acht und vier Knoten eine extrem hohe Anzahl an entfernten Aufrufen. Das Verhältnis zu lokalen Aufrufen liegt bei acht Knoten bei ganzen 91,1 %. Eine Analyse der Kaskade ergab, dass diese eine sehr komplexe Struktur aufweist und demnach keine sternförmige Kaskade (die meisten Retweeter sind zum Ursprungstweeter verbunden) ist. Dies zeigt sich zum Beispiel an der durchschnittlichen Anzahl von Friends, von welchen ein Retweeter potentiell beeinflusst werden könnte. Diese liegt bei der Kaskade *Mittel* bei 4.238. Die These aus der Laufzeitanalyse, dass die Partitionierung für *Mittel* nicht funktioniert hat, hat sich also bestätigt.

Die Kaskade *Kleiner* zeigt mit 22,1 % an entfernten Aufrufen, bei 8 Knoten, auch einen relativ hohen Wert. Bei der Analyse der Rekonstruktion der Kaskade zeigte sich, dass diese nur sehr schwach miteinander verbunden ist (hohe Zahl an Knoten aber wenige Kanten). Daraus ergibt sich ,dass der soziale Graph für diese Kaskade zu Beginn der

Graphpartition auch schon schwach verbunden war. Die Partitionierung oder Allokation sind hier also nicht der Verursacher der erhöhten Anzahl entfernter Aufrufe, sondern die Struktur des Aktivitätsgraph selbst.

Für die meisten Kaskaden wurden die Erwartungen an die Anzahl entfernter Aufrufe in einem Rahmen von +15% erfüllt. Insgesamt stimmt die Tendenz der entfernten Aufrufe mit den Erwartungen aus der Allokation überein. Allerdings scheint die Graphpartitionierung nach der vorherigen Analyse von *Mittel* für komplexe Strukturen ungeeignet. Dies müsste in einer weitergehenden Versuchsreihe mit zusätzlichen komplexen Kaskaden validiert werden, was aber zu umfangreich für den Rahmen dieser Arbeit ist.

4.4. Ladezeiten und Speicherverbrauch Sozialer Graph

Hier werden die Versuchsergebnisse bezüglich Ladezeiten und Speicherverbrauch des Hauptspeicher und benötiger Festplattenspeicher des sozialen Graphen gezeigt. Es gilt zu bestimmen ob der Speicherverbrauch des sozialen Graphen im Hauptspeicher und der physische Speicher auf der Festplatte mit den zuvor angenommenen Werten übereinstimmen. Zudem wird bei den Ladezeiten des sozialen Graphen erwartet, dass ein sozialer Graph schneller lädt wenn er über mehrere Knoten verteilt wurde.

Tabelle 4.8 auf Seite 63 zeigt für eine bestimmte Anzahl Knoten die folgenden Spalten:

- **K-Nr.:** Die jeweilige Knotennummer, für welchen die Speicherzahlen angegeben werden.
- **Erwartet:** Der erwartete Haupt- und Festplattenspeicherverbrauch, welcher bei der Allokation geschätzt wurde.
- **Festplatte:** Wieviel Platz die Dateien des sozialen Graphen (Follower-Listen) tatsächlich auf der Festplatte benötigten.
- **Hauptspeicher:** Wieviel Platz das Programm tatsächlich zur Laufzeit im Hauptspeicher eingenommen hat.
- **Haupt./Festpl.:** Das Verhältnis zwischen Hauptspeicher und benötigtem Festplattenspeicher.

Es ist zu erkennen, dass der benötigte Festplattenspeicher für die Follower-Listen sehr eng an den zu erwartenden Werten liegen. Die Annahme von 16 Bytes per Follower scheint für den benötigten Festplattenspeicher eine gute Größe zu sein. Beim Hauptspeicherverbrauch des kompletten Programms zeigt sich allerdings ein deutlicher Unterschied zwischen erwartetem und tatsächlichen Speicherverbrauch.

Bei den Messungen mit acht Knoten wurden 2,45 bis 3,1 mal mehr Hauptspeicher verbraucht als physikalischer Speicher auf der Festplatte, für den Graphen, benötigt wurde. Bei vier Knoten lagen diese Werte immerhin noch im Bereich von 1,72 bis 1,88. Bei zwei Knoten konnten keine konkreten Werte ermittelt werden, da der Hauptspeicher voll lief.

Natürlich kann vom physikalisch benötigten Festplattenspeicher nicht einfach *direkt* auf den Hauptspeicher des kompletten Programms geschlossen werden, da zum Beispiel die Laufzeitumgebung der JVM auch Speicher benötigt. Es zeigt sich gerade beim Vergleich von acht zu vier Knoten, dass der Overhead der JVM signifikant scheint. Möglicherweise benötigt der *Garbagecollector* für die hohe Menge an allokiertem Heap selbst deutlich mehr Hauptspeicher. Wieso genau die Zahlen so hoch liegen, konnte aber im Rahmen dieser Arbeit nicht mehr festgestellt werden.

Insgesamt zeigt sich jedoch, dass ein großer sozialer Graph erfolgreich auf mehrere Knoten verteilt werden kann. Die Effizienz der Speichernutzung sinkt allerdings bei steigender Knotenzahl.

Tabelle 4.9 auf Seite 63 zeigt die durchschnittlichen Ladezeiten des sozialen Graphen, sowie für die einzelnen Knoten die längste und kürzeste Ladezeit. Die Zeiten wurden nur für vier Knoten und acht Knoten gemessen. Beim Laden des sozialen Graphen für einen und für zwei Knoten wurde festgestellt, dass die jeweiligen Graphen nicht in den Hauptspeicher der Versuchsrechner passten. Außerdem konnte für vier Knoten kein Vorteil aus einem vorherigen Cachen des sozialen Graphen gezogen werden, da der Hauptspeicher zu voll wurde um den Cache zu halten.

Die Tabelle zeigt, dass ein Verteilen des sozialen Graphen auf mehrere Knoten signifikante Vorteile in den Ladezeiten bietet. Insbesondere da die Verteilung des Graphen auf mehrere Knoten, durch die geringere Größe, je nachdem Caching ermöglichen kann. Es zeigt sich aber auch, dass die Unterschiede der Ladezeiten zwischen den Knoten sehr groß sein können. Dies unterstreicht die Notwendigkeit des Blockierens des Spout bis alle Knoten bereit sind, da ansonsten das Netzwerk mit Nachrichten geflutet würde.

Die Ladezeiten sind interessant wenn mehrere Versuche, möglicherweise auch mit unterschiedlichen sozialen Graphen, ausgeführt werden. In einer finalen Umgebung allerdings, in welcher die Topologie die einkommenden Retweets in Echtzeit rekonstruiert, ist die Ladezeit vernachlässigbar, da die Topologie einmal geladen wird und dann durchgehend läuft.

Tabelle 4.7.: Tabellen entfernte Aufrufe für 8, 4 und 2 Knoten

(a) Anzahl Aufrufe für 8 Knoten

Kaskaden	Gesamt	Entfernt	Verhältnis	Vergl. zu Allokation (6,2 %)
Kleiner	3.500	774	22,1 %	+ 15,9 %
Klein	7.226	718	9,9 %	+ 3,7 %
Mittel	14.847	13.529	91,1 %	+ 84,9 %
Groß	29.783	517	1,7 %	- 4,5 %
Größe	60.451	2.303	3,8 %	- 2,4 %
500Kask	60.050	13.119	21,8 %	+ 15,6 %
100Kask	60.140	14.926	24,8 %	+ 18,6 %
10Kask	59.893	13.226	22,1 %	+ 15,9 %
Gemixt	240.534	43.657	18,2 %	+ 12 %

(b) Anzahl Aufrufe für 4 Knoten

Kaskaden	Gesamt	Entfernt	Verhältnis	Vergl. zu Allokation (4,2 %)
Kleiner	3.500	615	17,6 %	+ 13,4 %
Klein	7.226	854	11,8 %	+ 7,6 %
Mittel	14.847	13.439	90,5 %	+ 86,3 %
Groß	29.783	341	1,1 %	- 3,1 %
Größe	60.451	1.405	2,3 %	- 1,9 %
500Kask	60.050	9.681	16,1 %	+ 11,9 %
100Kask	60.140	10.643	17,7 %	+ 13,5 %
10Kask	59.893	9.419	15,7 %	+ 11,5 %
Gemixt	240.534	31.424	13,1 %	+ 8,9 %

(c) Anzahl Aufrufe für 2 Knoten

Kaskaden	Gesamt	Entfernt	Verhältnis	Vergl. zu Allokation (2,7 %)
Kleiner	3.500	407	11,6 %	+ 8,9 %
Klein	7.226	724	10 %	+ 7,3 %
Mittel	14.847	147	1 %	- 1,7 %
Groß	29.783	143	0,5 %	- 2,2 %
Größe	60.451	614	1 %	- 1,7 %
500Kask	60.050	5.991	10 %	+ 7,3 %
100Kask	60.140	7.171	11,9 %	+ 9,2 %
10Kask	59.893	6.973	11,6 %	+ 8,9 %
Gemixt	240.534	20.580	8,6 %	+ 5,9 %

Tabelle 4.8.: Speicherverbrauch sozialer Graph in Gigabyte

Knoten	K-Nr.	Erwartet	Festplatte	Hauptspeicher	Haupt./Festpl.
2	0	18	19	> 32	> 1,68
2	1	19,2	21	> 32	> 1,52
4	0	8,4	9,2	17	1,85
4	1	9,6	9,6	16.5	1,72
4	2	9,6	9,6	18	1,88
4	3	9,6	11	20	1,82
8	0	3,8	3.8	10,5	2,76
8	1	5,1	4.8	14	2,92
8	2	5,1	5.5	15	2,73
8	3	5,1	5.3	13	2,45
8	4	5,1	5.0	13	2,6
8	5	5,1	5.7	15,5	2,72
8	6	2,6	2.9	9	3,1
8	7	5,1	5.8	16	2,76

Tabelle 4.9.: Ladezeiten sozialer Graph

Knoten	Mit/Ohne Cache	Durchschnitt	Längste	Kürzeste
4	Ohne	37 min 50 s	48 min 48 s	21 min 10 s
8	Ohne	10 min 59 s	21 min 5 s	6 min 4 s
8	Mit	2 min 59 s	3 min 43 s	1 min 42 s

5. Fazit und Ausblick

In der Einleitung dieser Arbeit wurde als Ziel angegeben *eine Realisierung und Evaluation einer verteilten Rekonstruktion von Informationskaskaden* durchzuführen. Dieses Ziel ist in dieser Arbeit erreicht worden. Es wurde ein System entworfen, welches die gleichen funktionalen Ergebnisse wie die zentrale Kaskadenrekonstruktion, in einer verteilten Umgebung, liefert. Die Laufzeiten sind dabei vergleichbar oder besser und der soziale Graph kann auf mehrere Knoten aufgeteilt werden. Im *Fazit* wird nun genauer darauf eingegangen welche Teilziele vollständig oder teilweise umgesetzt werden konnten und wie deren Umsetzung zu bewerten ist. Der *Ausblick* zeigt auf, welche weitergehenden Versuche und welche Verbesserungen möglich wären.

5.1. Fazit

Durch eine erfolgreiche gemeinschaftsbasierte Partitionierung des sozialen Graphs und einer anschließenden Allokation der Graphpartitionen konnte gezeigt werden, dass der soziale Graph sich auf mehrere Knoten aufteilen lässt. Diese Aufteilung ermöglicht es auch sehr große soziale Graphen für eine Informationskaskadenrekonstruktion im Hauptspeicher zu halten. Dabei kann die Anzahl der Knoten, auf welchen das System läuft, beliebig erhöht werden um entsprechend viel Hauptspeicher bereit zu stellen.

Ebenfalls kann auch für das verteilte System, die gute Laufzeit der bisherigen zentralen Kaskadenrekonstruktion, aufrecht erhalten werden. Die Umsetzung des Ziels soziale Graphen aufzuteilen um so auch sehr große soziale Graphen im Hauptspeicher zu halten ist also gelungen.

Die Zeitersparnis einer parallel laufenden Kaskadenrekonstruktion konnte bei den Versuchen allerdings nicht aufgezeigt werden. Bei Versuchen mit noch größeren Datensätzen sollte sich dieser Aspekt aber aufzeigen lassen. Hier ist nur eine teilweise Umsetzung der Ziele gelungen. Es wurde jedoch beobachtet, dass der iterative Hash-Algorithmus der zentralen Kaskadenrekonstruktion bei einem verteilten sozialen Graphen, im Punkt Rekonstruktionszeit profitiert.

Eine komplett umfassende Versuchsreihe um das ganze verteilte System für alle Arten von Kaskaden abzudecken konnte nicht vollständig durchgeführt werden. Eine Versuchs-

reihe mit noch größeren Datensätze oder eine Versuchsreihe hauptsächlich bestehend aus komplexen Kaskaden konnte nicht umgesetzt werden. Bei den Versuchen konnten die Ziele also nur teilweise erreicht werden.

Abschließend kann noch gesagt werden, dass durch die Umsetzung dieser Arbeit eine zukünftige Echtzeitrekonstruktion von Informationskaskaden absolut machbar erscheint.

5.2. Ausblick

Hier werden Anregungen für zukünftige Arbeiten auf Basis der verteilten Rekonstruktion von Informationskaskaden gegeben. Dabei werden zuerst weitergehende Versuche genannt, welche noch mehr Aufschluss zum aktuellen System geben könnten und danach werden mögliche Verbesserungen aufgezeigt.

5.2.1. Weitergehende Versuche

Der wohl vielversprechendste Versuch wäre es, große Datensätze (viele Retweets von vielen verschiedenen Kaskaden) mit einem geringeren Füllfaktor der Hashmaps, des sozialen Graphen, zu testen. Dabei könnten deutlich andere Laufzeiten der verteilten Rekonstruktion entstehen. Dabei wäre es durchaus möglich, dass ein Füllfaktor von 0,5 bei einem Knoten trotzdem noch größere Laufzeit hat als ein Füllfaktor von 0,8 bei acht Knoten.

Weiterhin sollten komplexe Kaskaden und deren Auswirkung auf die Graphpartition untersucht werden. In dieser Arbeit zeigte sich beim Versuch mit komplexen Kaskaden, dass die Partitionierung dort sehr schlecht funktionierte, was zu einem enormen Anstieg der Zeit führte, welche die Knoten zum Austausch von Nachrichten benötigen. In dieser Arbeit konnte dies bei einer Kaskade beobachtet werden, ob dies bei allen komplexen Kaskaden der Fall ist, ist zu untersuchen.

Zudem könnte eine Verbesserung des Allokationsalgorithmus zu einem noch geringeren Austausch an Nachrichten zwischen den Knoten führen. Die gefundenen Lösungen, in dieser Arbeit, durch den gierigen Hill-Climbing-Algorithmus waren gut, jedoch könnte eine noch bessere Allokationsstrategie bessere Lösungen bringen. Zum Beispiel könnte eine Implementierung von *Simulated Annealing* eine Verbesserungen der bisher erzielten Ergebnisse bringen.

5.2.2. Verbesserungen

Bei der Implementierung des Systems wurden mögliche Verbesserungen erkannt. Die Implementierung war aber im Rahmen dieser Arbeit nicht mehr möglich.

Das Nachrichtenmodell in Unterunterabschnitt 3.3.3.5 wurde schon soweit verbessert, dass nicht alle Nachrichten einzeln gesendet werden, sondern Informationen gebündelt und dann als eine einzelne Nachricht verschickt werden. Dies kann aber noch weiter verbessert werden, indem nicht für jeden Retweet eine Nachricht, sondern nur eine Nachricht für die Informationen mehrerer Retweets gesendet wird. Dies ist eine besonders interessante Verbesserung, da beim Versenden sehr vieler Nachrichten der Speicherpuffer der Bolts immer noch vollzulaufen scheint. Dabei könnte auch untersucht werden ob dies am Nachrichtenprotokoll *Netty* liegt oder andere Gründe hat.

Schließlich könnten entfernte Anfragen per Piggyback, mit den normalen Retweets, gesendet werden. Dies ist allerdings nur bei Knoten möglich, welche selbst eine Rekonstruktion durchführen und demnach auch normale Retweets empfangen.

A. Infrastrukturaufbau

Rechnernetzwerk

In diesem Abschnitt wird der Aufbau der Infrastruktur des Rechnernetzwerks gezeigt, auf welchem mit Hilfe von *Storm Apache*, die Topologie zur *verteilten Rekonstruktion von Informationskaskaden* laufen soll.

Jeder Knoten auf dem Storm Apache laufen soll, benötigt zuerst eine Storm Installation. In dieser Installation findet sich eine Konfigurationsdatei welche Storm beim Start eines Nimbus oder eines Supervisors lädt (siehe Abschnitt 2.2 für Hintergrundinformationen). Diese Konfigurationsdatei heißt *storm.yaml*. Sie muss für den Nimbus und für die Supervisor speziell konfiguriert werden, damit die verteilte Rekonstruktion funktioniert.

Außerdem muss entschieden werden wie die Dateien im Rechnernetzwerk verteilt werden. Verteilt werden müssen: Die *Retweet-Kaskaden*, die *Allokationsdateien* und die nach der Allokation *aufgeteilten Dateien des sozialen Graphen* (Follower-Dateien). Dabei müssen sich die *Retweet-Kaskaden* auf dem gleichen Knoten wie der *Twitterstrom-Spout* befinden. Die *Allokationsdateien* müssen auf dem gleichen Knoten wie der *Routing-Bolt* sein. Die *Teile des sozialen Graphen* müssen auf dem von der Allokation vorgesehenen Knoten sein und auf diesen Knoten müssen die *Rekonstruktions-Bolts* laufen.

Die Spouts und Bolts können mit Hilfe eines selbsterstellten Schedulers auf bestimmte Supervisor mit einem identifizierbaren Namen fixiert werden. Ein solcher Scheduler wurde für diese Arbeit entwickelt. Ein Überblick über die gesamte Infrastruktur und das Dateisystem gibt Abbildung A.1

Setup des Nimbus

Die Konfigurationsdatei des Nimbus muss mindestens die Einträge in Listing A.1 enthalten. Zeile 1 gibt die Verbindung zu Zookeeper (Nimbus und Supervisor kommunizieren mit Hilfe von Zookeeper) an. Zeile 2 gibt die IP des Nimbus selbst an. Die wichtigste Zeile ist Zeile 3. Sie gibt den Namen des selbsterstellten Schedulers an. Dieser muss dem Nimbus als *Jar-Datei* im *Storm_Home/lib* Verzeichnis bereitgestellt werden. Ohne den Scheduler kann die Topologie für eine *verteilte Rekonstruktion von Informationskaskaden* nicht laufen.

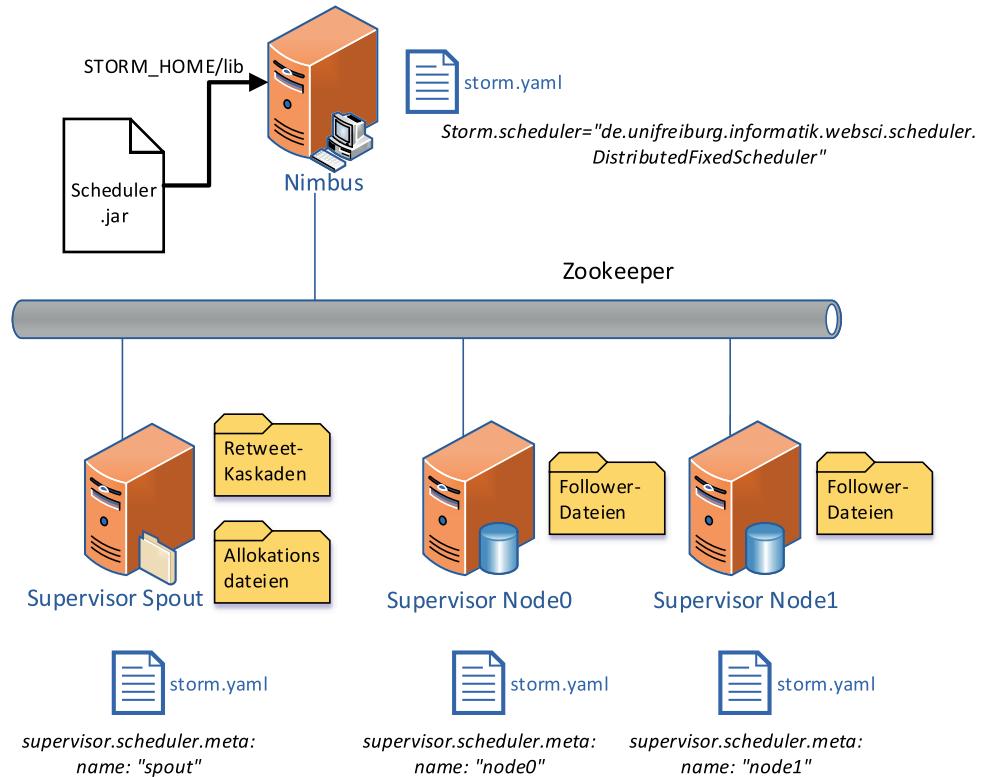


Abbildung A.1.: Infrastrukturaufbau Rechnernetzwerk

```

1 storm.zookeeper.servers: "127.0.0.1"
2 nimbus.host: "127.0.0.1"
3 storm.scheduler: "de.unifreiburg.informatik.websci.scheduler.
    DistributedFixedScheduler"

```

Listing A.1: Nimbus Konfiguration

Setup der Supervisor

Der Supervisor auf welchem der Twitterstrom-Spout, der Koordinator-Bolt, der Fenster-Bolt und der Routing-Bolt laufen sollen muss mindestens die Einträge in Listing A.2 in seiner *storm.yaml* Konfigurationsdatei haben. Zeile 1 und 2 sind notwendig um die IPs von Nimbus und Zookeeper zu kennen, damit überhaupt eine Kommunikation stattfinden kann. Bei Zeile 3 muss ein lokales Verzeichnis angegeben werden in welchem Storm seine Daten speichern kann. Der letzte und wichtigste Eintrag befindet sich in Zeile 4 und 5. Dieser Eintrag weist dem Supervisor den Namen *spout* zu. Nur mit diesem Namen können die speziellen Bolts auf diesen Supervisor fixiert werden.

```

1 storm.zookeeper.servers: "127.0.0.1"
2 nimbus.host: "127.0.0.1"

```

```
3 storm.local.dir: "/mein/storm/daten/verzeichnis"  
4 supervisor.scheduler.meta:  
5   name: "spout"
```

Listing A.2: Spout-Supervisor Konfiguration

Die Supervisor auf welchem die Rekonstruktions- und Speicher-Bolts laufen sollen, müssen den Namen nodeX bekommen. Wobei das X für eine gerade positive Zahl steht. Welcher Supervisor welche Zahl hat wird nach der ihm zugeteilten Allokation entschieden. Das heißt, werden die Dateien des sozialen Graphen aus Allokationsdatei node0.csv auf den Knoten des Supervisors kopiert, muss dieser auch den Namen *node0* bekommen. Zeile 1 bis 3 sind gleich dem Spout-Supervisor. Listing A.3 zeigt die *storm.yaml* eines Node-Supervisor.

```
1 storm.zookeeper.servers: "127.0.0.1"  
2 nimbus.host: "127.0.0.1"  
3 storm.local.dir: "/mein/storm/daten/verzeichnis"  
4 supervisor.scheduler.meta:  
5   name: "nodeX"
```

Listing A.3: Node-Supervisor Konfiguration

Es sei noch erwähnt, dass in der *storm.yaml* des Nimbus natürlich auch ein Name eines Supervisors vergeben werden kann. Auf einem Knoten können Nimbus und Supervisor gleichzeitig laufen.

Fixed Scheduler

Die Entwicklung des *Fixed Scheduler* basiert auf dem Code aus [13]. Der Scheduler muss dem Nimbus als *Jar-Datei* im *Storm_Home/lib* Verzeichnis bereitgestellt werden. Der Scheduler braucht um die Topologie erfolgreich zu schedulen folgende Informationen:

- Den Name der Topologie.
- Die Namen der Spouts und Bolts. Bzw. das Prefix für die Rekonstruktions- und Speicher-Bolts.
- Den Namen des Supervisor auf welchem der Twitterstrom-Spout, der Koordinator-Bolt, der Fenster-Bolt und der Routing-Bolt laufen sollen.
- Das Prefix für den Namen des Supervisor auf welchem die Rekonstruktions- und Speicher-Bolts laufen sollen.
- Die Anzahl der Knoten auf welchen die Rekonstruktions-Bolts und Speicher-Bolts laufen sollen.

Um den Scheduler dynamischer zu machen können ihm diese Werte per Config von der Topologie mitgeteilt werden. So muss zum Beispiel bei einer Namensänderung eines Supervisors nicht der komplette Scheduler neu compiliert und dem Nimbus erneut bereitgestellt werden. Alle dem Scheduler übergebenen Werte sind in Listing A.4 zu sehen. Der erste Parameter der `put()` Funktion stellt den Schlüssel dar und der zweite den zugeteilten Wert.

```
1 conf.put("twitteranalysis.topologyName", deployName);
2 conf.put("twitteranalysis.spoutSupervisorName", "spout");
3 conf.put("twitteranalysis.reconstSupervisorNamePrefix", "node");
4 conf.put("twitteranalysis.spoutName", SPOUT_NAME);
5 conf.put("twitteranalysis.incoordinatorBoltName",
    INCOORDINATOR_BOLT_NAME);
6 conf.put("twitteranalysis.cascWindowBoltName", CASC_WINDOW_BOLT_NAME)
    ;
7 conf.put("twitteranalysis.routingBoltName", ROUTING_BOLT_NAME);
8 conf.put("twitteranalysis.reconstBoltNamePrefix",
    RECONST_BOLT_NAME_PREFIX);
9 conf.put("twitteranalysis.storageBoltNamePrefix",
    STORAGE_BOLT_NAME_PREFIX);
10 conf.put("twitteranalysis.nodeCount", String.valueOf(nodeCount));
```

Listing A.4: Config von Topologie für Scheduler

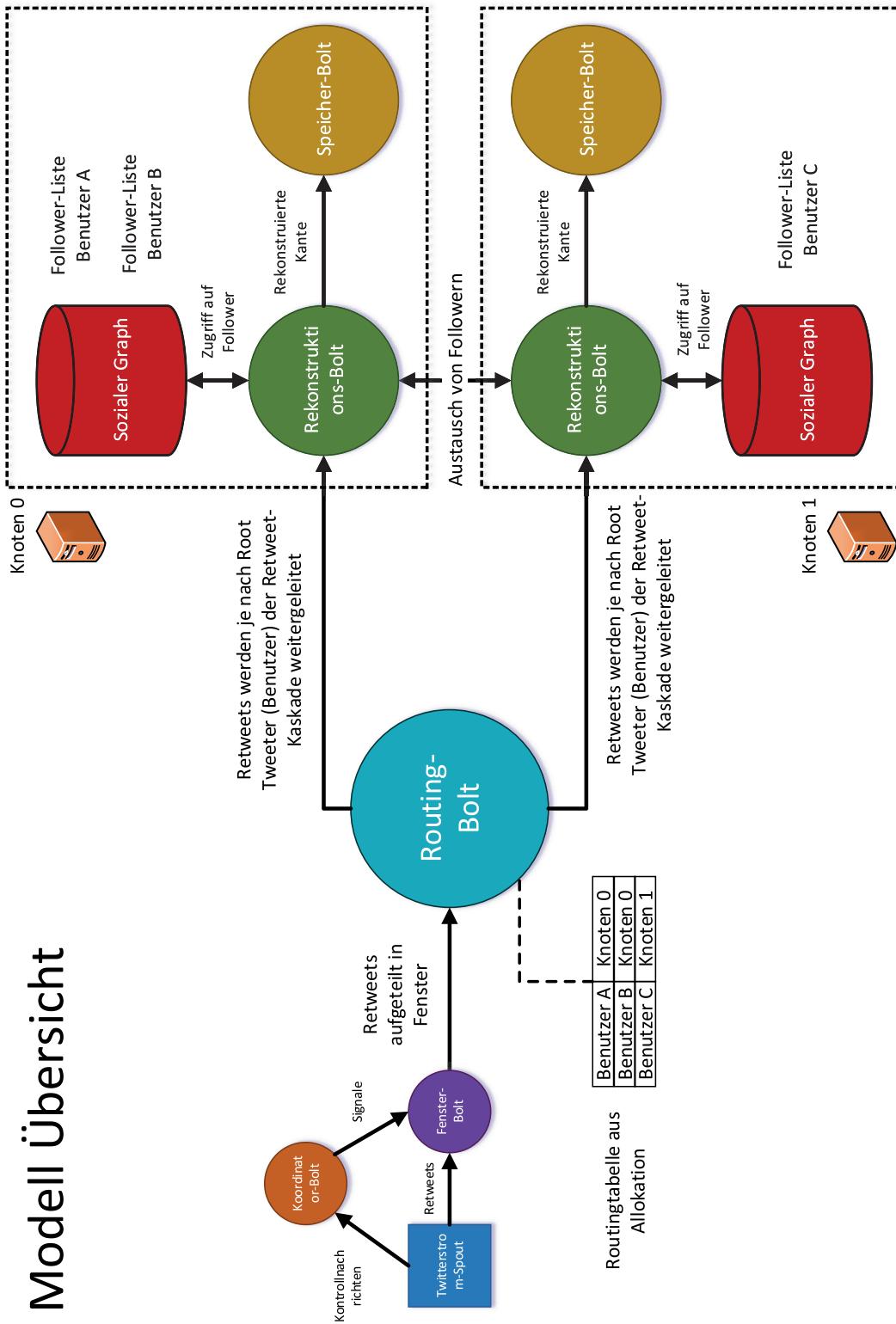
Storm UI

Am Rande sei hier noch kurz die Storm UI erwähnt. Sie ist ein nützliches Tool um eine laufende Topologie zu überwachen und wurde während dem Entwickeln des praktischen Teil der Arbeit häufig verwendet.

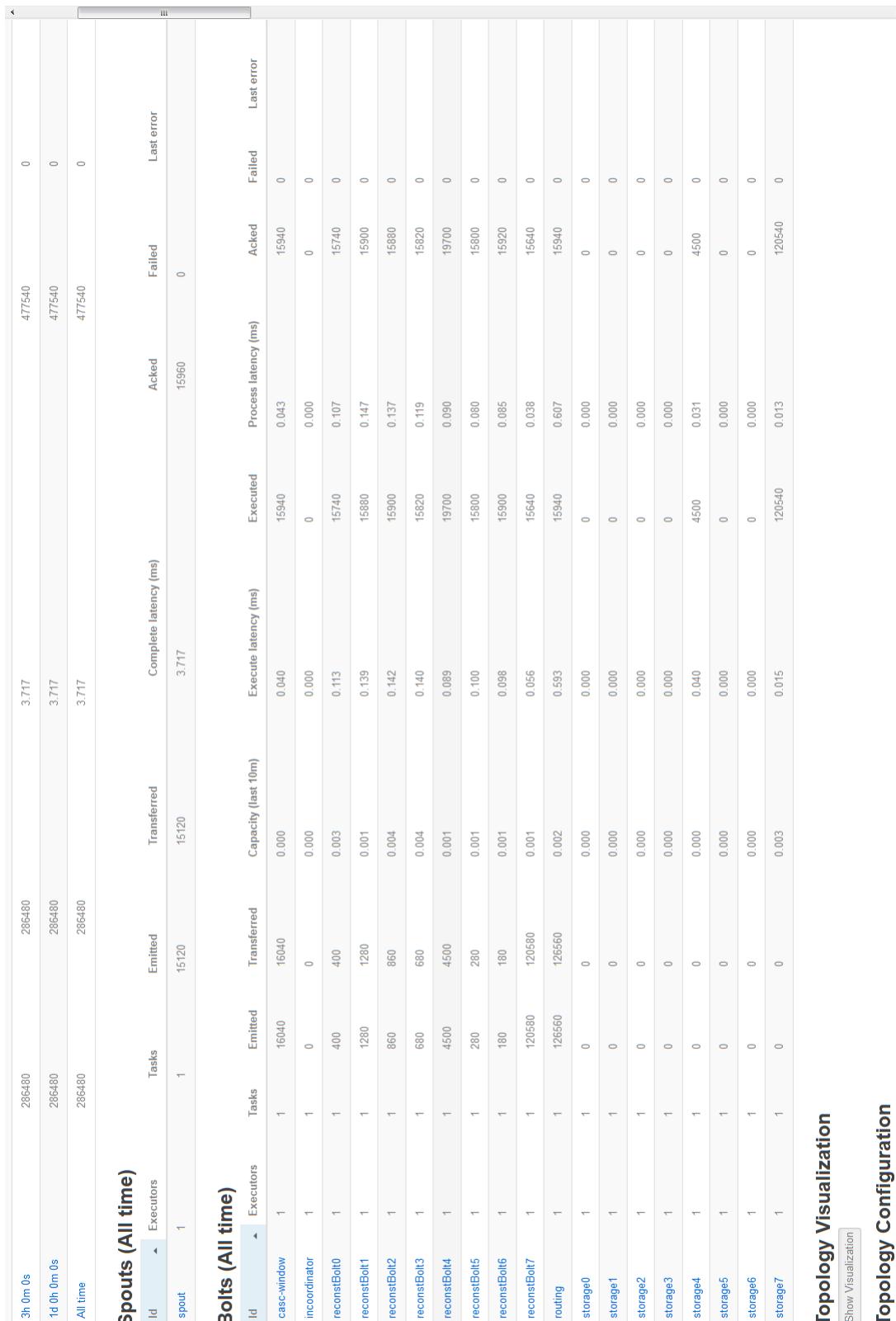
Sie zeigt live wieviele Nachrichten die Topologie bearbeitet. Für jeden Spout und Bolt wird die Anzahl der bearbeiteten Nachrichten, deren Ack's, deren Fail's und vieles mehr angezeigt. Ein Beispiel einer ausgeführten Topologie mit einer Retweet-Kaskade die 15.120 Retweets enthält ist in *Anhang B* zu finden.

B. Zusätzliche Abbildungen

Modell Übersicht



Storm UI



C. Untersuchte Kaskaden

Untersuchte Kaskaden mit deren Retweet-Größe und Dateinamen.

Kleiner:

3500 ./casc-232154412867530752.json

Klein:

7226 ./casc-232940326766137344.json

Mittel:

14847 ./casc-234745788608172033.json

Groß:

29783 ./casc-234747959626039296.json

Größe:

60451 ./casc-234954143758946304.json

10Kask:

6971 ./casc-231810937479380992.json

6576 ./casc-243956016511873024.json

6406 ./casc-232587608771948544.json

6039 ./casc-231449129992220672.json

5951 ./casc-232051416846454784.json

5829 ./casc-234681064751247361.json

5820 ./casc-234797997341437952.json

5503 ./casc-232927877279145984.json

5442 ./casc-234336858480467968.json

5356 ./casc-231925016621621249.json

100Kask:

643 ./casc-243501632262922240.json 643 ./casc-234839911818211329.json

643 ./casc-232878358596296705.json 643 ./casc-229508069644709888.json

642 ./casc-244551795081351168.json 640 ./casc-237204858619428865.json

640 ./casc-235111041661542402.json 640 ./casc-231778836583444480.json

639 ./casc-235658195484438528.json 639 ./casc-234403121349148672.json
639 ./casc-233629479182954496.json 636 ./casc-238375004079792128.json
632 ./casc-231928267916075008.json 631 ./casc-234317547401539584.json
629 ./casc-233502918937042944.json 629 ./casc-231848925072609281.json
626 ./casc-229679766322675712.json 625 ./casc-234378705496047616.json
625 ./casc-233685817820864513.json 625 ./casc-233566299870097408.json
625 ./casc-231931839936352256.json 623 ./casc-232891338398523392.json
622 ./casc-233948275303120897.json 619 ./casc-236402297196929025.json
619 ./casc-234768164016230401.json 618 ./casc-235560275498131456.json
615 ./casc-232588688670982146.json 615 ./casc-232062737776250881.json
614 ./casc-234771673876934656.json 614 ./casc-234234626422480896.json
613 ./casc-232664893227163648.json 612 ./casc-248317780124909569.json
612 ./casc-246124110944563200.json 612 ./casc-231451079739596800.json
611 ./casc-234844256190218241.json 611 ./casc-234003142965088256.json
610 ./casc-233729617398214656.json 608 ./casc-233976388565213184.json
608 ./casc-232134460009754625.json 607 ./casc-231788919073673216.json
606 ./casc-230742565111095296.json 605 ./casc-232257365930541056.json
604 ./casc-234758084071944193.json 604 ./casc-233653592731578368.json
599 ./casc-235476803911815168.json 599 ./casc-235422255360901120.json
599 ./casc-234173822624989185.json 597 ./casc-231708434393870336.json
595 ./casc-236629694865358848.json 595 ./casc-236576504732057602.json
595 ./casc-233219011461128192.json 594 ./casc-235969171966550016.json
594 ./casc-234816874561536000.json 594 ./casc-233297114715791362.json
593 ./casc-233716059914510339.json 592 ./casc-245059996574953474.json
592 ./casc-232575191513960448.json 591 ./casc-237234793971081216.json
591 ./casc-234789558414356482.json 591 ./casc-233305086116376576.json
590 ./casc-234798392369356800.json 590 ./casc-233785681938370560.json
590 ./casc-232141244099354624.json 590 ./casc-231101521524293632.json
589 ./casc-247902767849549825.json 589 ./casc-232116449353801729.json
588 ./casc-233681246641856512.json 588 ./casc-231881794155335680.json
587 ./casc-233504246077739010.json 586 ./casc-234524351309938688.json
586 ./casc-234069714790404097.json 586 ./casc-231869417410146304.json
585 ./casc-233378167140790273.json 584 ./casc-235119471981899776.json
584 ./casc-232506553251536897.json 582 ./casc-240903588015124480.json
582 ./casc-234884500902010881.json 582 ./casc-233662949363953664.json
581 ./casc-233524808086867968.json 580 ./casc-231998125114544128.json
579 ./casc-234316639179534336.json 579 ./casc-231575926914293760.json

578 ./casc-233587653868658689.json 578 ./casc-233048145804800000.json
578 ./casc-231946912260059136.json 577 ./casc-234691184939900928.json
577 ./casc-234617675349909504.json 577 ./casc-231864581465714688.json
576 ./casc-234685534872682496.json 576 ./casc-232944694764646400.json
576 ./casc-231586271171051520.json 576 ./casc-231434873083142145.json
574 ./casc-233424385652965377.json 572 ./casc-234877042431250432.json
572 ./casc-232894629580791808.json 571 ./casc-231799169357062144.json
571 ./casc-231670586391269376.json 569 ./casc-249228279586897923.json
569 ./casc-232642458348048384.json 569 ./casc-231403971158355970.json

Literaturverzeichnis

- [1] Joseph E Gonzalez et al. *PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.*, volume 12. OSDI, 2012.
- [2] Facebook. *Facebook Company Info.* 2014. Verfügbar unter <https://newsroom.fb.com/company-info/>, letzter Zugriff 20.10.2014.
- [3] Apache Software Foundation. *Apache Storm.* 2014. Verfügbar unter <http://storm.apache.org/>, letzter Zugriff 25.10.2014.
- [4] Wikipedia Foundation. *Open addressing.* 2014. Verfügbar unter http://en.wikipedia.org/wiki/Open_addressing, letzter Zugriff 14.11.2014.
- [5] GraphLab. *GraphChi.* 2014. Verfügbar unter <http://graphlab.org/projects/graphchi.html>, letzter Zugriff 25.10.2014.
- [6] Aapo Kyrola. *Communitydetection.* 2012. Verfügbar unter https://github.com/GraphChi/graphchi-cpp/blob/master/example_apps/communitydetection.cpp, letzter Zugriff 02.11.2014.
- [7] Katy Perry. *Katy Perry Twitter-Account.* 2014. Verfügbar unter <https://twitter.com/katyperry>, letzter Zugriff 25.10.2014.
- [8] Roy Ramos. *Distributed Graph Storage from Cascade Reconstruction.* Universität Freiburg, Institut für Informatik, 2013. Team Projekt.
- [9] Lukas Sättler and Simon Ebner. *Social Media Analysis.* Universität Freiburg, Institut für Informatik, 2013. Team Projekt.
- [10] Twitter. *Twitter About.* 2014. Verfügbar unter <https://about.twitter.com/company/>, letzter Zugriff 20.10.2014.
- [11] Twittercounter. *Twitter Top 100 Most Followers.* 2014. Verfügbar unter <http://twittercounter.com/pages/100>, letzter Zugriff 25.10.2014.

- [12] Joachim Wolff. *Verteilung und Partitionierung von sozialen Graphen anhand von Verbreitungsmustern von Informationen*. Universität Freiburg, Institut für Informatik, 2014. Bachelor Thesis.
- [13] xumingming. *DemoScheduler*. Mai 2012. Verfügbare unter <https://github.com/xumingming/storm-lib/blob/master/src/jvm/storm/DemoScheduler.java>, letzter Zugriff 07.11.2014.