Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Heinrich Latreider

Konzeption und Entwicklung einer Plattform zur Echtzeitanalyse temporaler Graphen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Heinrich Latreider

Konzeption und Entwicklung einer Plattform zur Echtzeitanalyse temporaler Graphen

Masterarbeit eingereicht im Rahmen der Masterprüfung

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

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 07.12.2018

Heinrich Latreider

Thema der Arbeit
Konzeption und Entwicklung einer Plattform zur Echtzeitanalyse temporaler Graphen

Stichworte
Echtzeit-Graphverarbeitung, Datenstromverarbeitung, Big Data

Kurzzusammenfassung
Soziale Netzwerke wie Facebook oder Twitter zeichnen sich dadurch aus, dass sie zum einen sehr groß werden können und zum anderen häufigen Änderungen unterliegen. Um solche dynamischen Graphen effizient zu verarbeiten, beschäftigt sich die Forschung in diesem Bereich unter anderem mit der Graphverarbeitung als Datenstrom. In dieser Arbeit wird die Konzeption und Entwicklung einer Plattform beschrieben, die eine Verarbeitung dynamischer Graphen als Graphdatenstrom ermöglicht. Hierfür werden Metriken berechnet, mit denen Aussagen über die Evolution von Graphen und darauffolgend Aussagen über die Entwicklung sozialer Netzwerke über die Zeit möglich sind.

Heinrich Latreider

Title of the paper
Conception and development of a platform for real-time analysis of temporal graphs

Keywords
real-time graph processing, stream processing, big data

Abstract
Social networks such as Facebook or Twitter are characterized by the fact that, on the one hand, they can become very large and on the other hand are subject to frequent changes. In order to process these dynamic graphs, the research in this field is concerned with graph processing as data stream. In this thesis the conception and the development of a platform is described which allows processing of dynamic graphs as data stream. For this goal metrics are calculated which allow statements about graph evolution and therefore make statements about evolution of social networks over time possible.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Big Data Plattformen wie Apache Spark und Apache Flink bieten mit Schnittstellen wie GraphX und Gelly Möglichkeiten, Graphen verteilt im Cluster zu verarbeiten. Die Verarbeitung findet batchbasiert statt, indem der Graph im Cluster verteilt und anschließend durch Operationen weiterverarbeitet wird. Nachteile bei der Batchverarbeitung ergeben sich jedoch, wenn Graphen dynamisch sind, also häufigen Änderungen unterliegen. In diesem Fall kann die Batchverarbeitung nur auf Momentaufnahmen (Snapshots) des Graphen erfolgen. Diese Verarbeitung muss für jeden Snapshot wiederholt werden, was bei großen Datenmengen entsprechend lange dauern kann. Eine Alternative zur Batchverarbeitung bildet die Streamverarbeitung, bei der die Verarbeitung von Daten kontinuierlich und unmittelbar erfolgt. Die Verarbeitung schnell wachsender Daten resultiert letztendlich in einer geringeren Antwortzeit, da nicht zuerst auf den Abschluss eines Batch-Jobs gewartet werden muss. (vgl. Kalavri und Carbone (2016))

Mithilfe der Streamverarbeitung lassen sich somit auch dynamische Graphen wie soziale Netzwerke verarbeiten. In Gellys experimenteller Streaming-API (Kalavri u. a. (2017)) erfolgt dies über einen Strom von Kanten, der einen Graph abbildet. Sind Kanten des Graphen zusätzlich mit Zeitinformationen versehen, handelt es sich um einen temporalen Graphen[1]. Mit temporalen Graphen kann unter anderem abgebildet werden, seit wann eine Verbindung zwischen zwei Knoten besteht. Dies ist vor allem bei der Verarbeitung sozialer Graphen von Bedeutung, wenn dessen Evolution über die Zeit beschrieben werden soll.

1.1. Motivation

Soziale Netzwerke wie Facebook oder Twitter bestehen aus Millionen von Nutzern, die auf verschiedene Weisen miteinander agieren. Twitter verzeichnet nach Stand vom Oktober 2018 monatlich etwa 326 Millionen aktive Nutzer (Twitter (2018)). Twitter ist ein Microblogging-Service, der es erlaubt, in sogenannten *Tweets* kurze Nachrichten zu veröffentlichen. Diese Tweets können von anderen Nutzern retweetet (weitergeleitet) und beantwortet werden. Ebenfalls ist es möglich, andere Nutzer in seinen Tweets zu erwähnen (Mention) und ihnen zu

[1] https://en.wikipedia.org/wiki/Time-varying_network

folgen (Follow). Beziehungen zwischen den Nutzern lassen sich als Graph auffassen, indem Nutzer als Knoten und Interaktionen zwischen Nutzern als Kanten abgebildet werden. Facebook verzeichnet monatlich über 2 Milliarden aktive Nutzer, von denen etwa 1,4 Milliarden sogar täglich aktiv sind. Etwa eine Milliarde Nutzer verwenden den Facebook eigenen Messenger monatlich, mit dem täglich Milliarden Nachrichten versendet werden (Facebook (2016)). Auch in diesem Fall können Freundschaftsbeziehungen und Interaktionen zwischen Nutzern als Graph modelliert werden.

Soziale Graphen haben die Eigenschaft, dass sie dynamisch sind und somit häufigen Änderungen unterliegen. Wird zum Beispiel eine neue Freundschaftsbeziehung zwischen zwei Nutzern eingegangen oder ein Tweet kommentiert, äußert sich dies in einer neuen Kante zwischen beiden betroffenen Nutzern. Bei dynamischen Graphen spielt außerdem die zeitliche Komponente eine Rolle, da durch diese die Evolution eines Graphen nachvollzogen werden kann. Santoro u. a. (2011) setzten sich in ihrer Arbeit mit verschiedenen temporalen sowie nicht-temporalen Graph-Metriken auseinander, um die Evolution von dynamischen Graphen zu beschreiben. Beispiele hierfür sind Dichte, Clustering-Koeffizienten und Modularität. Die Interpretation dieser Metriken erlaubte letztendlich Aussagen über die Entwicklung sozialer Netzwerke.

1.2. Ziele der Arbeit

In dieser Arbeit soll der Einsatz von Big Data Technologien für die Echtzeitverarbeitung temporaler Graphen untersucht werden. Dafür wird prototypisch eine Plattform spezifiziert und entwickelt, die einen Graphdatenstrom entgegennimmt, kontinuierlich weiterverarbeitet und die berechneten Ergebnisse als Zeitreihe speichert. Diese Zeitreihe soll herangezogen werden, um Aussagen über die zeitliche Entwicklung eines Graphen treffen zu können. Da der Graph kontinuierlich weiterverarbeitet wird, ist zudem immer dessen aktueller Zustand einsehbar. Ein mögliches Einsatzszenario stellt Twitter mit ihrer Streaming-API dar. Mithilfe einer Evaluation der Plattform mit einem realen Graphdatenstrom soll die Eignung der Plattform zur Echtzeitverarbeitung dynamischer Graphen untersucht werden.

1.3. Aufbau der Arbeit

Die Arbeit ist in folgende Kapitel aufgeteilt. In Kapitel 2 werden die Grundlagen zu Big Data und Stream Processing sowie Grundlagen der Graphentheorie in Hinsicht auf sozialen und temporalen Graphen beschrieben. Ebenfalls findet eine Auseinandersetzung mit existierenden

Forschungsarbeiten in diesem Gebiet sowie eine Abgrenzung zur eigenen Arbeit statt. In Kapitel 3 werden vorhandene Technologien analysiert, die sich zur Echtzeitverarbeitung von Graphen eignen. Außerdem werden in diesem Kapitel die funktionalen und nichtfunktionalen Anforderungen an die Plattform formuliert. In Kapitel 4 findet der Entwurf der Plattform statt, indem aus den Anforderungen aus Kapitel 3 einzelne Komponenten sowie Schnittstellen abgeleitet werden. In Kapitel 5 werden besondere Aspekte und gemachte Erfahrungen bei der Realisierung der Plattform beschrieben. In Kapitel 6 findet anschließend die Evaluation der Plattform unter Durchführung von Auswertungen statt. Kapitel 7 fasst die zentralen Erkenntnisse dieser Arbeit zusammen und gibt einen Ausblick auf mögliche weiterführende Forschungsarbeiten.

2. Grundlagen und verwandte Arbeiten

In diesem Kapitel werden zunächst die Grundlagen zur Graphentheorie in Hinblick auf soziale und temporale Graphen sowie die Grundlagen von Big Data mit Fokus auf die Verarbeitung von Graphen beschrieben. Anschließend findet eine Auseinandersetzung mit verwandten Forschungsarbeiten sowie die Abgrenzung zur eigenen Arbeit statt.

2.1. Graphentheoretische Grundlagen

Die Anfänge der Graphentheorie reichen bis in das 18. Jahrhundert zurück, angefangen mit dem Königsberger Brückenproblem vom Mathematiker Leonhard Euler. Gegeben ist die Stadt Königsberg, die vom Fluss Pregel in vier Teile aufgeteilt wird, welche wiederum durch insgesamt sieben Brücken miteinander verbunden sind. Gesucht ist ein Pfad, bei dem alle sieben Brücken exakt ein einziges mal durchlaufen werden. Durch die graphentheorietische Modellierung (Eulerkreis) konnte Euler die Nichtexistenz eines solchen Pfades beweisen. Aber auch andere Probleme wie Kürzeste Pfade, optimale Matchings, das Traveling Salesman Problem oder modernere Probleme wie PageRank (Page u. a. (1999)) und Community Detection (Fortunato (2010)) lassen sich graphentheoretisch modellieren und lösen.

Ein Graph $G = (V, E)$ ist eine mathematische Struktur, gegeben durch eine Menge von Knoten $V = \{v_1, v_2, ..., v_n\}$ und eine Menge von Kanten $E = \{e_1, e_2, ..., e_n\}$, die jeweils zwei

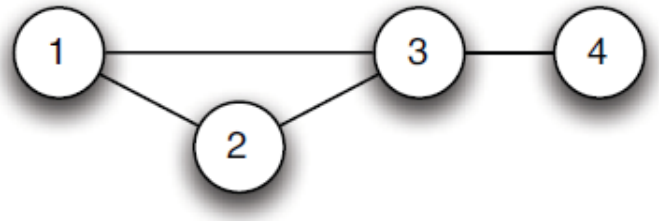


Abbildung 2.1.: Visuelle Darstellung eines Graphen mit vier Knoten und vier Kanten (Quelle: McGregor (2014))

Knoten $v_i, v_j \in V$ miteinander oder einen Knoten mit sich selbst verbinden. Durch Graphen lassen sich Objekte als Knoten und ihre Beziehungen untereinander als Kanten zwischen den Knoten darstellen. Beispiele hierfür sind Webseiten mit ihren Hyperlinks oder soziale Netzwerke mit Personen und ihren Freundschaftsbeziehungen untereinander (vgl. McGregor (2014)). Durch die Eigenschaft einer Kante, gerichtet oder ungerichtet zu sein, wird bestimmt, ob eine Beziehung zwischen zwei Knoten wechselseitig oder nur in eine Richtung gilt. Diese Eigenschaft gibt an, ob es sich um einen gerichteten oder ungerichteten Graphen handelt.

Im einfachsten Fall hat jede Kante eines Graphen ein Gewicht, mit dem sich etwa Kosten oder Distanzen zwischen zwei Knoten abbilden lassen. Es lassen sich aber auch andere Informationen in Knoten und Kanten abbilden. In vergangener Zeit hat sich hierfür das Property-Graph-Modell als Erweiterung des Graphmodells entwickelt. Dieses reichert den Graphen um eine Property-Menge an und ermöglicht so das Befüllen von Knoten und Kanten mit strukturierten oder unstrukturierten Daten (vgl. Edlich u. a. (2011)). Property-Graphen lassen sich heute in Graphdatenbanken wie Neo4j finden, sind aber auch als Basisstruktur in Graph Processing Frameworks wie GraphX (Gonzalez u. a. (2014)) und Gelly vertreten.

Der Grad (engl. *degree*) $deg(v)$ eines Knotens v definiert die Anzahl an Nachbarknoten, mit denen v verbunden ist. Wenn ein Knoten einen Grad von *zwei* hat, so ist er über zwei Kanten mit anderen Knoten verbunden. Bei einem gerichteten Graphen lässt sich der Grad zusätzlich in Eingangsgrad und Ausgangsgrad unterteilen, welche die Anzahl eingehender bzw. ausgehender Kanten angeben. Um Graphen zu verarbeiten, haben sich im Laufe der Zeit Graphalgorithmen entwickelt. Graphalgorithmen sind Algorithmen, die auf Graphen als Basisstruktur operieren. So können mithilfe des Dijkstra- oder Floyd-Warshall-Algorithmus kürzeste Pfade zwischen zwei Knoten oder mithilfe von Ameisenalgorithmen Lösungen für das Traveling Salesman Problem berechnet werden.

2.1.1. Soziale Graphen

Ein sozialer Graph ist definiert als ein Graph, der soziale Strukturen bzw. Netzwerke aus der realen Welt abbildet. Unter einem sozialen Netzwerk versteht man Personen und Dinge, die in unterschiedlicher Art miteinander in Beziehung stehen (vgl. Curtiss u. a. (2013)). Vertreter für soziale Netzwerke sind zum Beispiel Facebook, Twitter, Google Plus und LiveJournal (vgl. Leskovec und Krevl (2014)). Um diese verschiedenen Beziehungstypen abzubilden, hat jede Kante ein spezifisches Label. Während Facebook nach Curtiss u. a. (2013) Beziehungstypen wie *friend*, *likes* oder *tagged* in ihrem sozialen Graphen verwendet, lässt sich ein soziales Netzwerk in Twitter über Nutzer und Tweets sowie Beziehungstypen wie *follow*, *mention* oder *retweet*

abbilden (vgl. SNAP (2018)). Die meisten sozialen Graphen weisen Eigenschaften natürlicher Graphen auf.

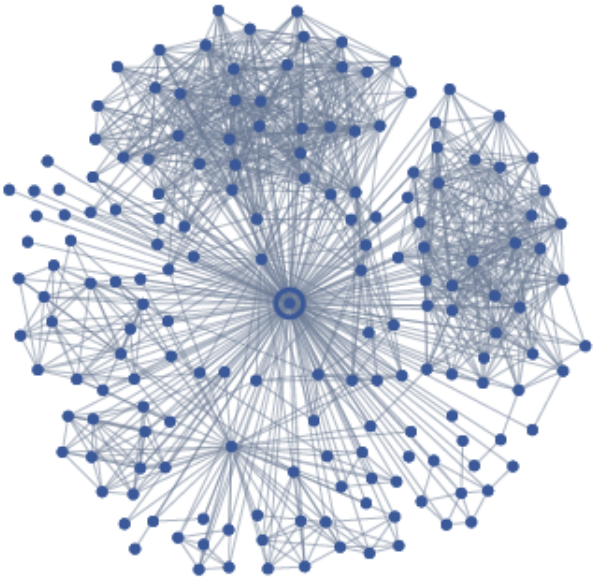


Abbildung 2.2.: Beispiel eines sozialen Graphen (Quelle: Curtiss u. a. (2013))

Natürlicher Graph

Ein Graph ist natürlich (skalenfrei), wenn er dem Potenzgesetz (engl. power law) folgt. Dem Potenzgesetz zufolge weist die Mehrheit der Knoten in einem natürlichen Graphen einen geringen Knotengrad auf, während einige wenige Knoten einen hohen Knotengrad besitzen (High-Degree Vertex) (vgl. Gonzalez u. a. (2012)). In einem sozialen Graphen bilden High-Degree Vertices in der Regel Nutzerkonten prominenter Personen wie Sänger, Schauspieler oder Politiker ab. Diese Personen kommen in vergleichsweise geringer Anzahl vor, die Anzahl ihrer Fans bzw. Anhänger liegt aber bei weitem höher als bei Durchschnittsnutzern. Bei Twitter liegt diese Anzahl im hohen zweistelligen bis dreistelligen Millionenbereich[1].

Mathematisch kann diese Eigenschaft durch folgende Formel beschrieben werden:

$$P(d) \propto d^{-\alpha} \tag{2.1}$$

α gibt die Schiefe der Wahrscheinlichkeitsverteilung an und liegt nach Santoro u. a. (2011) in der Regel im Intervall $[2, 3]$. Nach Gonzalez u. a. (2012) gilt für die meisten Graphen $\alpha \approx 2$. Je

[1] https://friendorfollow.com/twitter/most-followers (Zugriff: 12.06.2018)

höher α ist, desto höher ist die Wahrscheinlichkeit eines Knotens, einen niedrigen Grad und somit weniger Nachbarn zu haben. Dies resultiert wiederum in weniger Kanten und somit einer geringeren Dichte des Graphen.

Nach Gonzalez u. a. (2012) gehen im Twitter-Graph von einem Prozent der Knoten 50 Prozent der Kanten zu anderen Knoten aus. Dies bedeutet, dass die Hälfte aller insgesamt vorhandenen Beziehungen in Twitter durch ein Prozent der Nutzer entsteht. Diese Eigenschaft stelle besondere Anforderungen an die Verarbeitung durch Big Data Frameworks, da Knoten mit hohem Grad einen entsprechend höheren Bedarf an Speicherplatz und Rechenleistung aufweisen und einen höheren Kommunikationsaufwand erzeugen.

2.1.2. Temporale Graphen

Ein *temporaler* (auch *zeitveränderlicher*, *dynamischer* oder *sich entwickelnder*) Graph (vgl. Casteigts u. a. (2010)) ist eine Erweiterung des Graphmodells. Während ein statischer Graph, also ein Graph ohne temporale Eigenschaften, das Vorhandensein von Knoten und Kanten zu einem bestimmten Zeitpunkt modelliert und somit einen Schnappschuss der Problemdomäne abbildet, können durch temporale Graphen Entwicklungen wie das Hinzufügen und Entfernen von Knoten und Kanten und somit auch die Evolution eines Graphen abgebildet werden.

Einsatzzwecke für temporale Graphen sind nach Casteigts u. a. (2010) beispielsweise Transportnetzwerke wie die Luftfahrt, in denen Städte durch verschiedene Flüge erreicht werden können. Diese Flüge haben feste Abflugzeiten sowie eine bestimmte Flugdauer. Daraus folgt, dass Verbindungen zwischen Städten nur zu bestimmten Zeiten aktiv sind. Ein weiteres Beispiel sind Kommunikationsnetzwerke über Funk, in denen durch Kanten angeben werden kann, ob und zu welchem Zeitpunkt sich zwei Endpunkte in Reichweite voneinander befunden haben. Das letzte Beispiel, das auch in dieser Arbeit aufgegriffen wird, sind soziale Netzwerke. Soziale Netzwerke sind dynamisch aufgrund der Tatsache, dass Nutzer über die Zeit hinweg miteinander agieren. Durch temporale Graphen lässt sich zum Beispiel innerhalb sozialer Netzwerke abbilden, seit wann eine Freundschaft oder ein Kontakt zwischen zwei Personen besteht.

Formal lässt sich ein temporaler Graph nach Casteigts u. a. (2010) als ein TVG (time-varying graph) $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ modellieren. Ein TVG erweitert einen einfachen Graphen mit der Knotenmenge V und der Kantenmenge E um folgende Mengen:

- $\mathcal{T} \subseteq \mathbb{T}$ ist eine Zeitspanne und gibt die Lebenszeit eines Graphen an, wobei $\mathbb{T}$ die Zeitdomäne abbildet. Je nachdem, ob ein zeitdiskretes oder ein zeitkontinuierliches System modelliert wird, befindet sich $\mathbb{T}$ in der Domäne $\mathbb{N}$ für zeitdiskrete bzw. $\mathbb{R}^+$ für

zeitkontinuierliche Systeme. In realen Beispielen[2] wird für die Modellierung der Zeit oft der sogenannte Unix-Timestamp verwendet, der die vergangenen Sekunden seit dem 1. Januar 1970, 00:00 Uhr UTC angibt.

- $\rho : E \times \mathcal{T} \to \{0, 1\}$ ist eine Präsenzfunktion und bildet ab, ob eine Kante $e \in E$ zu einem Zeitpunkt $t \in \mathcal{T}$ aktiv (1) bzw. inaktiv (0) ist. Hierfür kennt jede Kante ihre Zeitintervalle, in denen sie aktiv ist. Der Fakt, dass eine Freundschaftsbeziehung zwischen zwei Personen seit dem 1. Januar 2018, 00:00 Uhr UTC existiert, kann zum Beispiel durch den Timestamp 1514764800 abgebildet werden.

- $\zeta : E \times \mathcal{T} \to \mathbb{T}$ ist eine Latenzfunktion und bestimmt die Dauer, die für die Überquerung einer Kante nötig ist. Innerhalb von Luftfahrtnetzwerken entspricht die Latenz zum Beispiel der Flugdauer, in Kommunikationsnetzwerken lässt sich damit die Übertragungsdauer eines Datenpakets abbilden. Die Latenz wird im weiteren Verlauf dieser Arbeit nicht berücksichtigt.

Soll ein bestimmter Ausschnitt aus dem temporalen Graphen $\mathcal{G}$ berücksichtigt werden, so lässt sich aus diesem ein statischer Graph G als *footprint* ableiten. Dieser Graph enthält alle Kanten, die innerhalb dieses Zeitraums existieren.

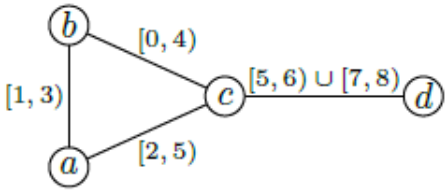


Abbildung 2.3.: Beispiel eines temporalen Graphen. Die Kanten enthalten Zeitintervalle, in denen sie aktiv sind. (Quelle: Santoro u. a. (2011))

Aus dem Graphen in Abb. 2.3 lassen sich zum Beispiel folgende statische Graphen ableiten.

- $G^{[1,2)}$: $V = \{a, b, c, d\}$, $E = \{(a, b), (b, c)\}$
- $G^{[2,3)}$: $V = \{a, b, c, d\}$, $E = \{(a, b), (b, c), (a, c)\}$
- $G^{[3,4)}$: $V = \{a, b, c, d\}$, $E = \{(a, c), (b, c)\}$
- $G^{[5,6)}$: $V = \{a, b, c, d\}$, $E = \{(c, d)\}$

[2] https://snap.stanford.edu/data/index.html#temporal (Zugriff: 20.06.2018)

2.2. Big Data und Graphverarbeitung

Als Big Data werden Daten bezeichnet, deren Menge (*Volume*), Wachstum (*Velocity*) und Vielfalt (*Variety*) so groß sind, dass konventionelle Hardware nicht mehr ausreicht, um diese Daten zu speichern und zu verarbeiten. Diese drei Eigenschaften werden auch unter dem 3-V-Modell zusammengefasst. Das Datenvolumen beginnt bei Big Data oft im Terabyte-Bereich (10^{12} Byte bzw. 1.000 Gigabyte) und kann bis in den Zettabyte-Bereich reichen (10^{21} Byte oder 1.000.000.000.000 Gigabyte). Durch das Wachstum der Datenmengen sind Technologien nötig, welche die Daten in Echtzeit als Datenströme verarbeiten können. Unter Datenvielfalt wird die Vielfalt der Datenformate bezeichnet. Daten können strukturiert, semistrukturiert oder unstrukturiert sein und kommen als Texte, Grafiken, Audio- und Videodateien vor. (Fasel und Meier (2016))

Um die genannten Probleme anzugehen, haben sich in der Vergangenheit Technologien entwickelt, mit denen sich große Datenmengen speichern und verarbeiten lassen. Ein bekanntes Beispiel ist die Plattform Apache Hadoop[3], die durch das verteilte Dateisystem HDFS eine verteilte Speicherung von Daten und durch den MapReduce-Algorithmus eine verteilte Verarbeitung dieser Daten auf Rechnerverbünden (engl. *Clustern*) ermöglicht.

MapReduce wurde erstmals von Google in Dean und Ghemawat (2008) vorgestellt und ist heute eines der zentralen Algorithmen für die Verarbeitung von Big Data. MapReduce basiert auf der Annahme, dass einzelne Datensätze unabhängig voneinander sind. Dadurch kann eine einfache Verteilung der Daten und eine nebenläufige Verarbeitung stattfinden. Da Graphen aufgrund ihrer Struktur jedoch zusammenhängende Daten enthalten, führt dies zu besonderen Herausforderungen bei der Verteilung und Verarbeitung von Graphen. In vergangener Zeit haben sich verschiedene Lösungsansätze entwickelt, die eine verteilte Verarbeitung von Graphen ermöglichen. Darunter befinden sich die Ansätze Pregel (Malewicz u. a. (2010)) und PowerGraph (Gonzalez u. a. (2012)). Pregel führt das Prinzip ”Think Like a Vertex” ein, durch das Graphalgorithmen aus ”Vertex-Programmen” zusammengesetzt sind, die untereinander Nachrichten austauschen und verarbeiten. PowerGraph fokussiert sich vor allem auf die Verarbeitung natürlicher Graphen.

Im Folgenden werden zunächst beide Ansätze vorgestellt, um danach tiefergehend auf die Verarbeitung von Datenströmen einzugehen, die im Verlauf dieser Arbeit relevant sein wird. Dabei wird speziell die Verarbeitung von Graphdatenströmen eine zentrale Rolle spielen.

[3] http://hadoop.apache.org

2.2.1. Batchverarbeitung

Pregel

Pregel ist ein Lösungsansatz von Google zur parallelen Verarbeitung von Graphen und wurde erstmals in Malewicz u. a. (2010) vorgestellt. Pregel wird sowohl als verteiltes Berechnungsmodell für Graphen nach dem Paradigma "Think like a Vertex" verstanden, aber auch als Plattform, die dieses Modell implementiert. Inspiriert ist der Ansatz vom Berechnungsmodell *Bulk Synchronous Parallel* (BSP) (Valiant (1990)).

Bulk Synchronous Parallel entstand bereits im Jahr 1990 als Vorschlag für ein "Bridging Model" zur Abstraktion paralleler Berechnungen vom darunterliegenden verteilten System und zum Entwurf und Implementation paralleler Algorithmen. Der Ablauf eines in BSP implementierten Algorithmus kann Abb. 2.4 entnommen werden.

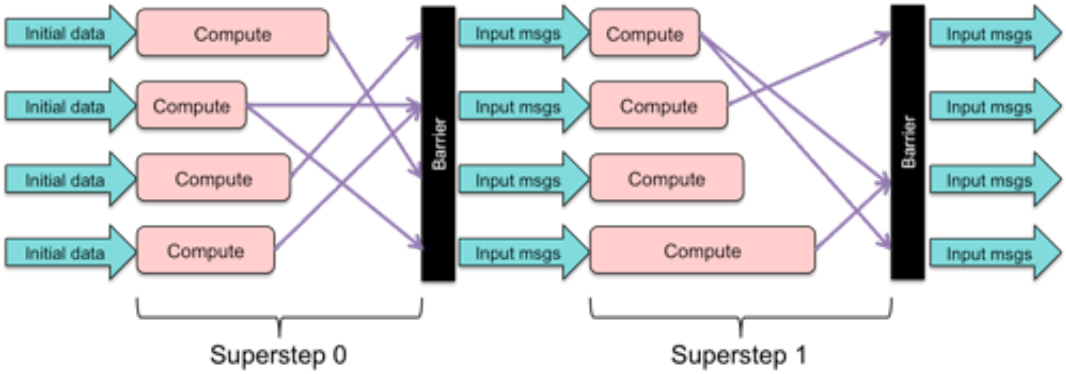


Abbildung 2.4.: BSP Berechnungsmodell (Quelle: Krzyzanowski (2012))

Auf der obersten Ebene einer BSP-Berechnung stehen Supersteps, die jeweils einen Berechnungsschritt abbilden. Ein Superstep besteht wiederum aus mehreren Einzelschritten, in denen die einzelnen BSP-Prozesse im ersten Schritt Eingabedaten erhalten. Im nächsten Schritt erfolgt die parallele Verarbeitung der Teildaten durch die einzelnen BSP-Prozesse, indem jeder Prozess lokal Berechnungen auf seinen Daten ausführt. Diese Verarbeitung kann je nach Prozess unterschiedlich lang dauern. Da es sich bei BSP um ein synchrones Berechnungsmodell handelt, findet nach jedem Superstep eine Synchronisation der BSP-Prozesse statt. Nach der lokalen Berechnung ist es jedem Prozess erlaubt, Nachrichten an andere Prozesse zu schicken. Erst nach der Synchronisation ist es möglich, in den nächsten Superstep überzugehen, wobei jedem Prozess alle Nachrichten zugestellt werden, die im vorherigen Superstep an sie geschickt wurden. Diese Nachrichten werden wiederum lokal durch die einzelnen BSP-Prozesse weiterverarbeitet, bis keine Nachrichten zwischen Prozessen mehr ausgetauscht werden.

Pregel macht sich diesen Ansatz zunutze, indem jeder Knoten im Graph als ein Prozess modelliert wird, der in der Lage ist, Nachrichten von anderen Knoten zu empfangen, diese zu verarbeiten und Nachrichten an andere Knoten zu senden. Gleichzeitig kann jeder Knoten die beiden Zustände $aktiv$ und $inaktiv$ einnehmen. Der Zustand eines Knotens hängt davon ab, ob der Knoten Berechnungen durchführen muss ($aktiv$) oder nicht ($inaktiv$). Der Wechsel zwischen den Zuständen erfolgt durch das Abschließen der lokalen Berechnung ($aktiv \rightarrow inaktiv$) bzw. durch den Empfang von Nachrichten ($inaktiv \rightarrow aktiv$). Haben alle Knoten den Zustand $inaktiv$ eingenommen, gilt die Berechnung als terminiert.

Da es sich bei Pregel um ein Berechnungsmodell handelt, müssen durch den Anwender der Pregel-Schnittstelle folgende Funktionen implementiert werden, um einen Algorithmus abzubilden.

- **Compute():** Pregel folgt dem Paradigma ”Think like a Vertex”, weshalb in der Compute-Funktion die Berechnungslogik des Algorithmus aus der Sicht eines Knotens spezifiziert wird. Die Funktion gibt an, wie ein Knoten empfangene Nachrichten verarbeitet, welchen Zustand er daraufhin einnimmt und welche Nachrichten verschickt werden. Die Ausführung dieser Funktion erfolgt parallel durch jeden Knoten.
- **Combine():** Da sich benachbarte Knoten auf verschiedenen Maschinen befinden können, ist eine Netzwerkkommunikation beim Senden von Nachrichten unumgänglich. Aus diesem Grund erfolgt wenn möglich eine Reduktion von Nachrichten mit dem gleichen Zielknoten, um die Netzwerklast möglichst gering zu halten. Diese Reduktion erfolgt durch eine Aufsummierung der Nachrichten, die in der Combine-Funktion spezifiziert wird. Der Zielknoten erhält somit eine einzelne akkumulierte Nachricht, auf dessen Basis der Knoten weitere Berechnungen vornimmt. Da über die Nachrichtenreihenfolge keine Annahmen getroffen werden, muss die Combine-Funktion zwingend assoziativ und kommutativ sein.

PowerGraph

Bei PowerGraph handelt es sich um einen Ansatz zur Verarbeitung natürlicher Graphen. Da sich Knotengrade in natürlichen Graphen stark unterscheiden, ist die Verarbeitung mit diversen Herausforderungen verbunden. So wird die Partitionierung des Graphen erschwert, da ein Knoten inklusive seiner Kanten bezüglich Speicher zu groß für eine Maschine sein kann. Auch die Rechenzeit pro Knoten ist ungleichmäßig verteilt, da Vertex-Programme bei Knoten mit einem hohen Grad (High-Degree Vertices) mehr Zeit benötigen als bei Knoten mit kleinem

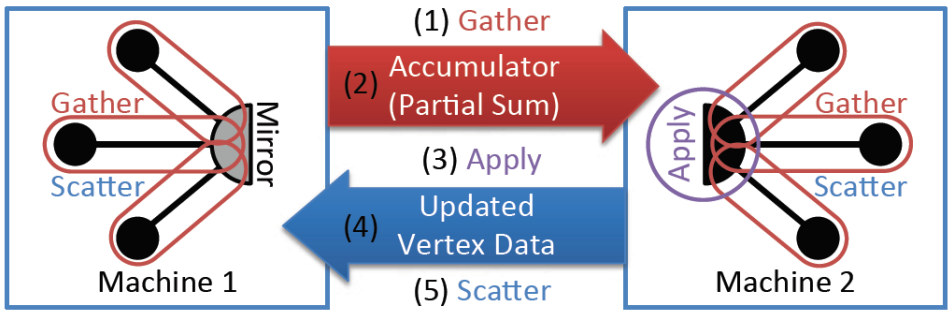


Abbildung 2.5.: Beispiel eines Vertex Cut in PowerGraph (Quelle: Gonzalez u. a. (2012))

Grad. Die Parallelität der Berechnung hängt somit maßgeblich von den High-Degree Vertices ab.

Zur Lösung dieses Problems führt PowerGraph eine neue Abstraktion ein, die eine Parallelisierbarkeit der Vertex-Programme ermöglichen soll und den Berechnungsaufwand gleichmäßiger verteilt. Um ein Vertex-Programm verteilen zu können, führt PowerGraph einen neuen Ansatz zur Graphpartitionierung ein. Diese findet nicht wie bei Pregel durch eine zufällige Verteilung von Knoten statt. Der Grund hierfür ist die Wahrscheinlichkeit, mit der sich zwei benachbarte Knoten auf verschiedenen Maschinen befinden. Diese beträgt nach Gonzalez u. a. (2012) $1 - \frac{1}{p}$, wobei p die Anzahl der Maschinen angibt. Im Beispiel $p = 20$ beträgt die Wahrscheinlichkeit bereits 95%, was in einem hohen Nachrichtenoverhead resultiert, wenn ein Knoten Informationen über seine Nachbarn benötigt. In realen Clustern ist die Anzahl der Maschinen beliebig nach oben skalierbar, womit auch die Wahrscheinlichkeit für den Schnitt einer Kante immer weiter ansteigt und sich dem Wert 1 annähert.

Statt eines Kantenschnitts findet in PowerGraph ein Schnitt über die Knoten statt (Vertex-Cut). In diesem Fall befinden sich die Kanten eines Graphen auf einer Maschine, ein Knoten kann hingegen auf mehrere Maschinen repliziert werden. Die Netzwerklast entsteht hier nicht mehr durch den Nachrichtenaustausch zwischen benachbarten Knoten, sondern zwischen den Replikaten eines Knotens. Abb. 2.5 zeigt den Vertex-Cut eines High-Degree Vertex mit sechs Nachbarn. Von diesem High-Degree Vertex werden zwei Replikate erzeugt und alle benachbarten Kanten gleichmäßig auf beide Maschinen verteilt. Ein Replikat wird dabei zum Master-Knoten ernannt.

In einem Berechnungsschritt berechnet sich jedes Replikat in der *Gather*-Phase aus seiner zugewiesenen Teilmenge benachbarter Kanten und Knoten nebenläufig seine Teilsumme gemäß einer Summierungsfunktion. Im nächsten Schritt übermitteln alle Replikate ihre Teilsumme an den Master-Knoten, der alle Teilsummen zu einer Gesamtsumme addiert. Abhängig

von der Gesamtsumme passt der Master-Knoten in der *Apply*-Phase seinen Zustand an und propagiert die Zustandsänderung wiederum an seine Replikate. In der *Scatter*-Phase wird der neue Zustand anschließend an die Nachbarn propagiert. Um den Kommunikationsaufwand bei der Graphverarbeitung möglichst gering zu halten, wird bei der Partitionierung ein möglichst geringer Replikationsfaktor der Knoten angestrebt.

2.2.2. Datenstromverarbeitung

Als Datenstrom wird eine lange bzw. ununterbrochene Folge von Datensätzen bezeichnet, dessen Speicherung aufgrund der Datenmenge und des Datenaufkommens nicht möglich ist (vgl. Bry u. a. (2004)). Ebenso ist das Ende eines Datenstroms im Voraus meist nicht abzusehen (vgl. Wikipedia (2018)). Aus diesem Grund müssen Datenströme fortlaufend verarbeitet werden. Dabei können Datenströme so groß werden, dass die Verarbeitung eines Datensatzes maximal einmal erfolgen kann (vgl. Aggarwal (2006)). Algorithmen, die diese Bedingung erfüllen, werden auch als *One-pass*-Algorithmen bezeichnet. Die Verarbeitung von Datenströmen findet entweder durch Programme statt, die speziell auf einen Anwendungsfall angepasst sind, oder in sogenannten Stream-Processing Frameworks, die eine Plattform für die Verarbeitung von Datenströmen zur Verfügung stellen. Datenströme lassen sich grob in die Kategorien Transaktionsdatenströme und Messdatenströme unterteilen. Während Messdatenströme durch Sensoren oder Messstationen generiert werden, sind Transaktionsdatenströme als Logdaten zu verstehen, die Ereignisse wie den Zugriff auf Webressourcen oder die Nutzung einer Kreditkarte protokollieren (Bry u. a. (2004)).

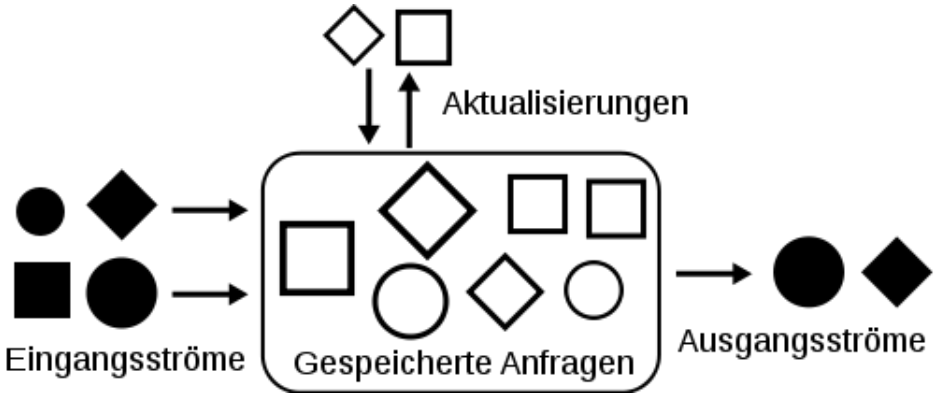


Abbildung 2.6.: Beispielhafte Verarbeitung von Datenströmen (Quelle: Wikipedia (2018))

Im Gegensatz zur Batch-Verarbeitung erfolgt die Verarbeitung von Datenströmen in ”quasi”-Echtzeit. Eingehende Datensätze werden durch Datenstromalgorithmen verarbeitet. Diese

aggregieren den Datenstrom, um etwa Metriken zu berechnen, oder transformieren ihn in andere Datenströme, z.B. durch Filtern oder Anreichern (vgl. Abb. 2.6). Auf diese Art lässt sich die fachliche Logik des Anwendungsgebietes realisieren. Im Folgenden werden Ansätze zur Verarbeitung von Graphdatenströmen vorgestellt.

Semi-Streaming

Das Semi-Streaming Modell ist ein Modell, welches speziell für die Verarbeitung von Graphen eingeführt wurde, dessen Kantenmenge zu groß ist, um in den Speicher zu passen (vgl. Feigenbaum u. a. (2005)). Die Verarbeitung muss somit in diesem Fall zwangsläufig als Datenstrom erfolgen. Ein Graph-Stream ist eine zufällige Folge von Kanten $e_1, e_2, \cdots, e_m$ mit $e_j \in E$ und $1 \leq j \leq m$. Die Verarbeitung des Datenstroms findet Kante für Kante statt. Das Semi-Streaming-Modell fokussiert sich vor allem auf Algorithmen, die eintreffende Daten nur einmal verarbeiten müssen. Es werden aber auch Algorithmen berücksichtigt, die für die Verarbeitung eine konstante oder maximal logarithmische Anzahl an Durchläufen benötigen. Ausgehend von dem Fakt, dass die Kantenmenge zu groß für den Speicher ist, steht einem Semi-Streaming-Algorithmus $O(n \cdot \text{polylog } n)$ Speicherplatz zur Verfügung, wobei $n = |V|$. Die Effizienz eines Semi-Streaming Algorithmus richtet sich insgesamt nach dem benötigten Speicherplatz, der Verarbeitungsdauer und der Anzahl an Durchläufen über den Datenstrom.

Nach McGregor (2014) können Graphdatenströme zudem in folgende Arten unterteilt werden.

Insert-Only Streams: Ein Datenstrom besteht aus einer Sequenz ungeordneter Kanten $e = (u, v)$, wobei $u, v \in V$. Dieser Datenstrom $S = \langle e_1, e_2, \cdots, e_m \rangle$ bildet die Kantenmenge E ab, sodass ein ungerichteter Graph $G = (V, E)$, mit $E = \{e_1, e_2, \cdots, e_m\}$ definiert wird. Kanten können in diesem Modell lediglich hinzugefügt, jedoch nicht entfernt werden.

Graph Sketches: Im Graph-Sketch Modell besteht ein Graphdatenstrom aus einer Sequenz von Kanten $S = \langle a_1, a_2, \cdots, a_m \rangle$, wobei $a_i = (e_i, \Delta_i)$ und $\Delta_i \in \{-1, 1\}$. Während e_i die Kante (u, v) definiert, bildet Δ_i ab, ob diese Kante hinzugefügt oder entfernt wird. Die Multiplizität einer Kante e wird durch $f_e = \Sigma_{i:e_i=e} \Delta_i$ angegeben. In einem einfachen Graphen gilt für alle Kanten $f_e \in \{0, 1\}$. Eine Unterkategorie von Datenstromalgorithmen bilden *Linear Sketches*. Diese Algorithmen verwalten intern eine lineare Projektion der Eingabedaten, aus der sich Eigenschaften der Eingabedaten ableiten lassen.

Sliding Window: Im Sliding Window Modell wird der Datenstrom ähnlich des Insert-Only

Streaming Modells als eine unendliche Folge von Kanten $\langle e_1, e_2, \cdots, e_m \rangle$ definiert. Der Unterschied besteht darin, dass im Sliding Window Modell zu einem Zeitpunkt t lediglich die letzten ω Kanten berücksichtigt werden, sodass ein Fenster $W = \{e_{t-\omega+1}, \cdots, e_t\}$ entsteht.

2.3. Verwandte Arbeiten

Santoro u. a. (2011) untersuchten in ihrer Arbeit soziale Graphen unter Berücksichtigung ihrer Dynamik. Hierfür legten sie das TVG-Modell zugrunde, das in Abschnitt 2.1.2 vorgestellt wurde. Auf Basis dieses Modells fand die Untersuchung eines Zitationsnetzwerks statt. Hierfür wurden temporale sowie atemporale Metriken identifiziert, deren Evolution Aussagen über das Netzwerks ermöglichen soll. Die Analyse fand jedoch nicht in Echtzeit statt. Auf die Metriken wird im folgenden detailliert eingegangen, da sie die Grundlage dieser Arbeit bilden.

Dichte: Die Dichte (engl. *density*) gibt das Verhältnis der Anzahl vorhandener Kanten in einem Graphen zur maximal möglichen Anzahl an. Die maximal mögliche Anzahl bei einem einfachen Graphen beträgt $|V| \cdot (|V| - 1)$. Für ungerichtete Graphen halbiert sich diese Anzahl nochmals. Die Berechnung der Dichte eines Graphen ermöglicht Aussagen über den Vernetzungsgrad des gesamten Graphen. Berechnet wird die Dichte folgendermaßen:

$$D = \frac{|E|}{|V| \cdot (|V| - 1)} \tag{2.2}$$

Ein Abnehmen der Dichte über die Zeit deute nach Santoro u. a. (2011) beispielsweise darauf hin, dass ein soziales Netzwerk zwar wächst, aber zwischen den Nutzern keine weiteren Verbindungen eingegangen werden.

Clustering Coefficient: Der *Clustering Coefficient* gibt an, wie nah die Nachbarschaftsmenge $N(v)$ eines Knotens v an eine Clique herankommt. Eine Clique ist ein Subgraph $U \subseteq V$, in welchem jedes Knotenpaar $u, v \in U$ durch eine Kante verbunden ist. U ist somit ein vollständiger Graph. Der Clustering Coefficient wird zunächst für jeden Knoten lokal berechnet (*Local Clustering Coefficient*):

$$C(x) = \frac{|\{(u, v) : u, v \in N(x)\}|}{deg(x)(deg(x) - 1)} \tag{2.3}$$

Der *Average Clustering Coefficient* gibt den Durchschnitt aller *Local Clustering Coefficient* im Graphen an und berechnet sich folgendermaßen:

$$AC = \frac{1}{|V|} \sum_{x \in V} C(x) \tag{2.4}$$

Über den zeitlichen Verlauf der Clustering Coefficients kann die Entwicklung von Gemeinschaften (Communitites) nachvollzogen werden. So deute ein Ansteigen des *Average Clustering Coefficient* darauf hin, dass sich innerhalb eines Graphen dichte Untergemeinschaften bilden.

Modularität: Die Modularität eines Graphen gibt an, inwieweit sich der Graph in Module unterteilen lässt. Eine hohe Modularität ist gegeben durch dichte *intramodulare* sowie lose *intermodulare* Verbindungen. Ein Anwendungsbeispiel ist etwa die Entdeckung von Communities (Community Detection). Die Modularität eines Knotenpaars $u, v \in V$ ist gegeben durch:

$$M(u, v) = \frac{deg(u) \cdot deg(v)}{2|E|} \tag{2.5}$$

Durch die Evolution der Modularität können Rückschlüsse auf das Aufteilen oder Zusammenwachsen von Communities gezogen werden. Nach Santoro u. a. (2011) weisen Netzwerke mit hohen Clustering Coefficients ebenfalls eine hohe Modularität auf.

Potenzgesetz: Die Verteilung der Knotengrade gibt Aufschluss darüber, inwieweit die Eigenschaft eines natürlichen Graphen erfüllt ist. Die Evolution des Potenzgesetzes über die Zeit kann Aufschluss über das Auftauchen oder Verschwinden von Knotenpunkten geben, die verschiedene Gruppen miteinander verbinden.

In Taxidou und Fischer (2013) wurden Methoden zur Echtzeitanalyse von sozialen Medien diskutiert. Der Fokus dieser Arbeit lag dabei in der Analyse der Informationsverteilung im sozialen Netzwerk Twitter. Diese Informationsverteilung findet zum Beispiel durch die *Retweet*-Funktion von Twitter statt, mit der es möglich ist, Tweets anderer Nutzer weiterzuleiten. Aus den Retweets können demnach Kaskaden entstehen, indem ein Tweet immer weiter retweetet wird. Die Autoren beschäftigten sich dabei vor allem mit den Fragen, auf welche Art sich Informationen in sozialen Netzwerken verbreiten, wie schnell dies vonstatten geht und wie lange der Prozess dauert. Ferner beschäftigten sie sich mit der Frage, welche Rolle die Nutzer bei der Informationsverteilung spielen. Mithilfe der Ergebnisse soll es Journalisten beispiels-

weise möglich sein, Informationen zurückzuverfolgen, die Verbreitung eigener Informationen nachzuverfolgen und eventuell selber Einfluss darauf zu nehmen.

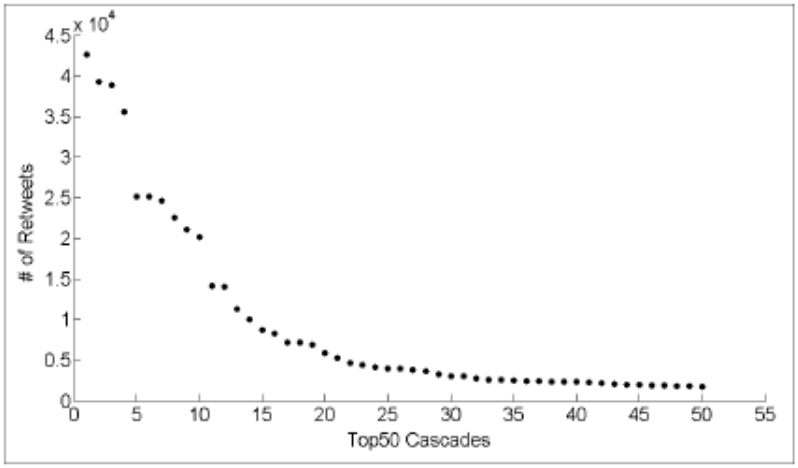


Abbildung 2.7.: Top 50 der größten Kaskaden über die Olympischen Spiele 2012 (Quelle: Taxidou und Fischer (2013))

Um die Analysen betreiben zu können, wurden zunächst Ansätze diskutiert, die eine Rekonstruktion von Informationskaskaden ermöglichen. Im nächsten Schritt wurden Metriken diskutiert, die aus den Kaskaden berechnet werden, etwa Pfadlängen, Anzahl involvierter Nutzer, Wachstumsraten, Latenzen zwischen Retweets oder die Lebensdauer einer Kaskade. Im letzten Schritt wurde eine Infrastruktur errichtet, welche die Echtzeitanalyse letztendlich implementiert. Als Stream Processing Framework kam hierbei Apache Storm (Storm (2018)) zum Einsatz. In Abb. 2.7 ist ein Ergebnis der Analyse zu sehen. Die Abbildung zeigt die Top-50 der Informationskaskaden mit dem Thema ”Olympische Spiele 2012”. Während zehn Tweets über 2.000 mal retweetet wurden, weisen die restlichen 40 Tweets eine Retweet-Rate von 1500 oder weniger auf. Dabei ist deutlich ein nichtlinearer Verlauf zu erkennen. Während sich die Autoren auf die Rekonstruktion von Informationskaskaden und dessen Auswertung konzentriert haben, liegt der Fokus dieser Arbeit auf Metriken, die Aussagen über die Evolution sozialer Netzwerke ermöglichen.

In Wickramaarachchi u. a. (2015) wurde ein Ansatz vorgestellt, der eine Echtzeitanalyse von Graphen ermöglichen soll, die sich in kurzen Abständen ändern bzw. weiterentwickeln. Der Fokus dieser Arbeit lag dabei auf verteilte Datenquellen sowie auf die Verteilbarkeit der Graphverarbeitung auf Cloud-Plattformen. Als Werkzeug zur Verarbeitung kam die Stream

Processing Engine Floe[4] zum Einsatz, die nach Angaben der Autoren eine Eigenentwicklung ist und Funktionen wie Fehlertoleranz und Skalierbarkeit bietet. Da Floe von Haus aus keine Graphschnittstelle anbot, entwickelten die Autoren aufbauend auf Floe eine Graphabstraktion, um eine Graphverarbeitung zu ermöglichen. Wichtige Ziele hierbei waren effiziente Graph-Updates sowie eine möglichst geringe Latenz bei der Verarbeitung.

Als Anwendungsfall kam die Ermittlung einer Menge von *Key-Nodes* zum Einsatz, die der Identifikation der wichtigsten Knoten innerhalb eines Graphen dient. Key-Nodes zeichnen sich dadurch aus, dass sie eine hohe Anzahl ausgehender Kanten aufweisen, die im Beispiel Twitter etwa durch häufiges Retweeten entstehen. Ebenfalls spiele die Nachbarschaft des Knotens eine wichtige Rolle. Potenzielle Einsatzgebiete für diesen Anwendungsfall seien etwa Advertising oder Disease Spread Monitoring. Aufbauend auf der Graphverarbeitung erfolgte die Entwicklung einer Visualisierungskomponente, mit der unter anderem eine geographische Visualisierung, aber auch eine Interaktion mit dem Graphen ermöglicht werden sollte.

In Marciani u. a. (2016) wurde das Big Data Framework Apache Flink[5] verwendet, um soziale Netzwerke zu analysieren. Ziel der Arbeit war zum einen die Implementation eines Algorithmus zur Ermittlung der k Posts mit dem meisten Einfluss innerhalb eines Zeitfensters und zum anderen die Identifikation der größten Communities bezogen auf ein Thema. Beides sollte in Echtzeit stattfinden. Neben der Implementation der Algorithmen spielte außerdem die Verarbeitungsgeschwindigkeit bzw. die Latenz eine wichtige Rolle. Der in der Arbeit beschriebene Experimentaufbau führte bei Experimenten letztendlich zu einer Verarbeitungsgeschwindigkeit von 400 Datensätzen pro Sekunde mit einer durchschnittlichen Latenz von 2,5 ms für den ersten Algorithmus, beim zweiten Algorithmus betrug die Rate 370 Datensätze pro Sekunde bei einer durchschnittlichen Latenz von 2,7 ms. Beide Werte beziehen sich darauf, dass die Hälfte des Testdatensatzes bereits verarbeitet wurde. Die Messung der Latenzen zeigte eine Korrelation mit der Anzahl verarbeiteter Datensätze, da diese mit wachsender Anzahl verarbeiteter Datensätze stetig anstieg.

[4] http://pgroup.usc.edu/wiki/Floe
[5] https://flink.apache.org

3. Analyse

In diesem Kapitel wird zunächst die Problemstellung beschrieben, die im Rahmen dieser Arbeit zu bearbeiten ist. Im nächsten Schritt werden sowohl funktionale als auch nichtfunktionale Anforderungen an die Plattform beschrieben. Daraufhin werden einige Algorithmen beschrieben, die in dieser Arbeit zum Einsatz kommen. Anschließend findet eine Auseinandersetzung mit existierenden Technologien und Werkzeugen statt, die als potenzielle Lösung für die Problemstellung infrage kommen.

3.1. Problembeschreibung

Im Rahmen dieser Arbeit soll eine Plattform entwickelt werden, die eine Echtzeitverarbeitung von temporalen Graphen ermöglicht. Der Fokus soll dabei auf soziale Graphen gelegt werden. Soziale Graphen unterliegen häufigen Änderungen und erfüllen somit die Eigenschaft temporaler bzw. dynamischer Graphen. In Kapitel 2 wurde bereits ein erster Überblick über Ansätze gegeben, die für die verteilte Verarbeitung von Graphen infrage kommen. Dabei wurde sowohl die Verarbeitung als Batch als auch die Streamverarbeitung angesprochen. Da die Verarbeitung der Graphen in Echtzeit stattfinden soll, kommt demnach nur die Streamverarbeitung infrage und somit eine Lösung, die auf Stream-Processing Technologien aufbaut.

Die Streamverarbeitung erfolgt wie in Kapitel 2 bereits erwähnt als Kantenstrom. Im Bereich sozialer Graphen wie Facebook entspricht dies einem Strom aus Ereignissen wie einer neuen Freundschaftsbeziehung oder den Beitritt in eine Gruppe. Im Falle Twitter entspräche dies dem Retweeten oder Kommentieren eines anderen Tweets. Ein Kante muss dabei mindestens einen Start- sowie einen Zielknoten aufweisen. Legt man die Definition von Property-Graphen zugrunde, können einer Kante zusätzlich weitere Attribute verliehen werden. Im Bereich temporaler Graphen wäre die naheliegendste Eigenschaft ein Zeitstempel, der die Zeit abbildet, an dem das Ereignis stattgefunden hat.

Da Datensätze sozialer Graphen aus verschiedenen Quellen stammen und somit verschiedene Strukturen aufweisen, muss zunächst eine Bereinigung bzw. Normalisierung der Rohdaten stattfinden. Diese Bereinigung ist von Quelle zu Quelle verschieden und sollte erfolgen, bevor die Datensätze weiterverarbeitet werden. Im nächsten Schritt soll der bereinigte Datenstrom

an eine Stream-Processing Engine weitergeleitet werden, welche den Datenstrom letztendlich auswertet. In Abschnitt 2.3 wurde auf Metriken eingegangen, die in Santoro u. a. (2011) untersucht wurden. Aus diesen Metriken soll eine Auswahl getroffen werden, die im Rahmen dieser Arbeit unter anderem implementiert wird. Die Ergebnisse sollen in regelmäßigen Zeitabständen zwischengespeichert werden, um die Evolution der Metriken abbilden zu können. Die Speicherung dieser Ergebnisse soll dabei in einer geeigneten Form erfolgen. Die Metriken werden im letzten Schritt visualisiert und ausgewertet, um entsprechende Aussagen über den untersuchten Graphen ableiten zu können.

Ferner ist für die Auswertung von Datenströmen die Fehlertoleranz des Gesamtsystems von großer Bedeutung. Die Plattform muss in der Lage sein, mit Ausfällen einzelner Komponenten oder ähnlichen Ereignissen umzugehen. In verteilten Systemen können solche Ereignisse nicht gänzlich ausgeschlossen werden. So sollte sichergestellt werden, dass der aktuelle Systemzustand nach dem Ausfall einer Komponente wiederhergestellt wird und Daten nicht verlorengehen.

3.2. Anforderungen

Um die Plattform zur Echtzeitverarbeitung von temporalen Graphen entwerfen und realisieren zu können, müssen zunächst alle funktionalen und nichtfunktionalen Anforderungen an die Plattform formuliert werden. Diese Anforderungen ergeben sich aus der oben genannten Problemstellung. Die Plattform wird im Folgenden als System bezeichnet.

3.2.1. Funktionale Anforderungen

Funktionale Anforderungen beschreiben die Funktionalität, die ein System gegenüber den Nutzern erfüllen muss.

F1 Das System kann Datenströme beliebig vieler Graphen entgegennehmen. Ein Graphdatenstrom besteht aus einer kontinuierlichen Sequenz von Kanten.

F2 Die Verarbeitung verschiedener Graphen erfolgt unabhängig voneinander.

F3 Eine Kante besteht aus einer eindeutigen Graph-ID, Start- und Zielknoten in Form von IDs sowie einer Zeitinformation in Form eines Unix-Timestamp.

F4 Die Graph-ID wird vom System vergeben, die IDs von Start und Zielknoten stammen aus der Datenquelle. Eine ID ist eine 64-Bit Ganzzahl.

F5 Die Zeitinformation stammt aus der Datenquelle, sodass auch vergangene Datensätze verarbeitet werden können. Es werden keine Annahmen über die aktuelle Zeit getroffen. Zudem wird dadurch das Testen erleichtert.

F6 Das System errechnet aus den Datenströmen in Echtzeit folgende Metriken: Anzahl der Knoten und Kanten, Dichte, Modularität, Average Clustering Coefficient sowie das Knoten und Kantenwachstum.

F7 Die Echtzeiteigenschaft kann durch ein konfigurierbares Zeitintervall beeinflusst werden. Der Standardwert dieses Intervalls beträgt 60 Sekunden.

F8 Die berechneten Metriken werden zusammen mit einem Zeitstempel abgespeichert. Der Zeitstempel ergibt sich aus dem aktuellsten Zeitstempel innerhalb des Intervalls.

F9 Das System soll die berechneten Metriken visualisieren. Die Visualisierung erfolgt als Liniendiagramm, um die zeitliche Komponente einer Metrik zu veranschaulichen. Die visualisierten Metriken werden wiederum in einem Dashboard zusammengefasst.

F10 Für jeden Graph existiert ein eigenes Dashboard.

3.2.2. Nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen beschreiben Anforderungen, die nicht die Funktionalität eines Systems definieren, aber dennoch wichtige bzw. wünschenswerte Eigenschaften bezüglich der Qualität des Systems beschreiben.

N1 Das System soll horizontal skalieren, d.h. bei Hinzufügen weiterer Ressourcen in Form von Workern soll der Durchsatz verarbeiteter Datensätze steigen.

N2 Das System soll fehlertolerant gegenüber Ausfällen einzelner Komponenten sein. Komponenten sollen automatisch wieder hochfahren und es sollen keine Daten verlorengehen.

3.3. Algorithmen

Dieser Abschnitt beschäftigt sich mit existierenden Algorithmen zur Berechnung der in Anforderung **F6** spezifizierten Metriken. Besonders die Berechnung des Average Clustering Coefficient und der Modularität ist nichttrivial. Aus diesem Grund werden im Folgenden Lösungsansätze für beide Probleme vorgestellt.

Modularität

Ziel der Ermittlung von Communities ist die Maximierung der Modularität. Die Berechnung einer Unterteilung, welche die Modularität maximiert, erweist sich nach Brandes u. a. (2006) als NP-vollständiges Problem. Das bedeutet, dass kein Algorithmus existiert, der das Problem in polynomieller Zeit löst. Stattdessen wird auf Näherungsverfahren zurückgegriffen, die keine exakte Lösungen berechnen, dafür jedoch in polynomieller Zeit akzeptable Lösungen finden. In Shang u. a. (2014) wurde ein Verfahren vorgestellt, das im Vergleich zu anderen Verfahren auch auf dynamischen Graphen operieren kann. Die ursprüngliche Definition der Modularität Q eines Graphen ergibt sich aus Gleichung (3.1).

$$Q = \frac{1}{2m} \sum_{i,j \in V} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j) \tag{3.1}$$

Während m die Summe der Kantengewichte innerhalb des gesamten Graphen repräsentiert, gibt k_i den Knotengrad und c_i die Community von i an. Die Funktion δ bildet ab, ob es sich bei c_i und c_j um dieselbe Community handelt und beträgt 1 für $c_i = c_j$, ansonsten 0. Eine andere Darstellung der Gleichung (3.1) ist in Gleichung (3.2) dargestellt.

$$Q = \frac{1}{2m} \sum_{c \in C} \left(\sum_{in}^{c} - \frac{\sum_{tot}^{c\ 2}}{2m} \right) \tag{3.2}$$

Während $\sum_{in}^{c}$ die Summe der inneren Kantengewichte in c angibt, stellt $\sum_{tot}^{c}$ die Summe aller Kantengewichte dar, die zu Knoten in c inzident sind. Ziel des Algorithmus ist die Maximierung von Q. Trifft eine neue Kante ein, so wird zuerst unterschieden, ob es sich um eine Kante (u, v) handelt, deren Knoten u und v derselben Community angehören (*inner community edge*), u und v in verschiedenen Communities liegen (*cross community edge*), nur ein Knoten neu ist (*half new egde*) oder beide Knoten neu sind (*new edge*). Aus diesen vier Kantentypen ergeben sich vier verschiedene Aktionen: 1. die Community-Struktur unverändert lassen, 2. zwei Communities miteinander verbinden, 3. Knoten einer existierenden Community zuweisen und 4. eine neue Community erstellen.

Handelt es sich bei der hinzugefügten Kante um eine *inner community edge*, so wird Aktion 1 angewandt, bei einer *half new edge* Aktion 3 und bei einer *new edge* Aktion 4. Die resultierende Aktion bei Hinzufügen einer *cross community edge* ergibt sich aus der Veränderung von Q unter Anwendung von Aktion 1 oder Aktion 2. Sollte das Verschmelzen zweier Communities zu einer höheren Modularität bzw. zu einem niedrigeren Verlust führen, so findet Aktion 2 Anwendung,

ansonsten Aktion 1. Der Vorteil dieses Ansatzes liegt darin, dass er auf feingranularer Ebene operiert und somit auch für dynamische Graphen infrage kommt.

Average Clustering Coefficient

Der Average Clustering Coefficient gibt den durchschnittlichen Grad innerhalb eines Graphen an, mit dem die Nachbarschaftsmenge $N(v)$ eines Knotens v an eine Clique heranreicht. Der Wert eines Clustering Coefficient liegt hierbei im Intervall $[0, 1]$. Bei einem Graphen ausschließlich bestehend aus Cliquen liegt der Wert des Average Clustering Coefficient beispielsweise nahe 1, während in Graphen mit weniger dichten Gruppen der Wert entsprechend geringer ist. Der Clusterkoeffizient eines Knotens v errechnet sich wie in Santoro u. a. (2011) beschrieben aus dem Quotienten der Anzahl der Kanten, die zwischen den Nachbarn von v verlaufen, und der maximal möglichen Kantenanzahl, die zwischen diesen maximal verlaufen können. Während sich hier der Divisor aus dem Knotengrad $deg(v)$ berechnen lässt, basiert die Berechnung des Dividenden auf dem Problem des Zählens von Dreiecken, in der Literatur als *Triangle Count* bezeichnet. Ein Dreieck beschreibt hierbei eine 3-Clique, in der drei Knoten jeweils durch Kanten miteinander verbunden sind. Die Anzahl der Dreiecke, in denen ein Knoten v enthalten ist, gibt gleichzeitig die Anzahl der Kanten an, die zwischen den Knoten aus $N(v)$ verlaufen.

In Gelly Stream liegt bereits eine Implementation des Triangle Count Algorithmus vor Kalavri u. a. (2016), die im Rahmen dieser Arbeit um eine temporale Komponente sowie um die Berechnung des Average Clustering Coefficient erweitert werden müsste.

3.4. Technologien

In diesem Abschnitt findet eine Auseinandersetzung mit Technologien und Werkzeugen statt, die als potenzielle Lösung für die zu entwickelnde Plattform infrage kommen. Die Technologien sind in die Kategorien Infrastruktur, Processing Frameworks und Softwarestacks unterteilt.

3.4.1. Docker

Docker[1] ist ein Werkzeug, das es möglich macht, Anwendungen und deren Abhängigkeiten zu anderen Ressourcen von der darunterliegenden Laufzeitumgebung (Betriebssystem) zu isolieren. Ermöglicht wird dies durch Containerisierung, bei der als Laufzeitumgebung einer Anwendung statt einer physischen oder virtuellen Maschine ein Container zum Einsatz kommt

[1] https://www.docker.com

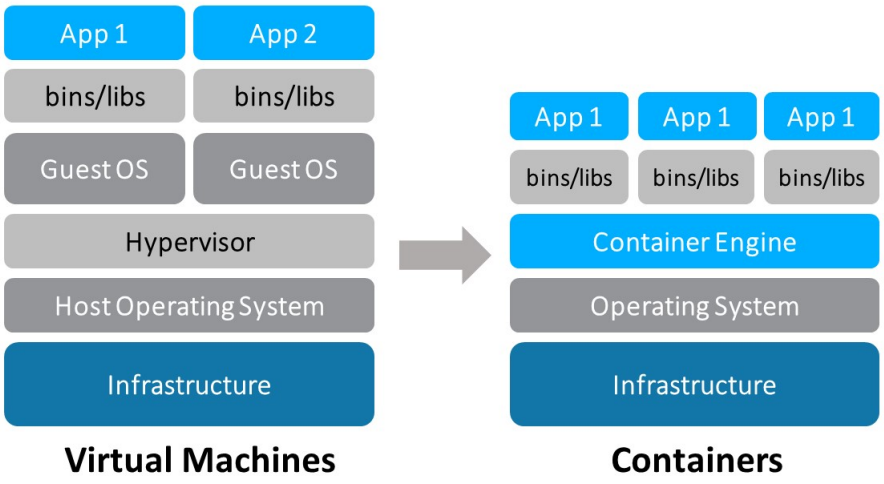


Abbildung 3.1.: Unterschied zwischen virtueller Maschine und Container (Quelle: Rancher (2018))

(vgl. Abb. 3.1). Während eine virtuelle Maschine ein eigenes Betriebssystem enthält, das auf einem Host-System läuft, wird ein Docker-Container durch die Docker Engine direkt auf dem Host-System ausgeführt, ist aber dennoch von diesem abgeschottet. Technisch betrachtet werden Anwendung sowie dessen Abhängigkeiten in einer Image-Datei zusammengefasst, aus denen im nächsten Schritt Container erzeugt werden. Anwendungen lassen sich als Image-Dateien auf andere Rechner übertragen und können dort mithilfe der Docker-Engine als Container ausgeführt werden. Das Deployment einer Anwendung beschränkt sich somit auf die Erstellung eines Image und der Erstellung eines Containers aus diesem Image auf dem Zielrechner. Die Plattform Docker Hub[2] bietet zahlreiche Image-Dateien, die von anderen Docker-Nutzern erstellt und veröffentlicht wurden und somit öffentlich zur Verfügung stehen.

Im Rahmen dieser Arbeit erscheint Docker als ein potenzielles Werkzeug zur Isolation einzelner Komponenten und somit zur Ermöglichung einer Microservice-Architektur.

3.4.2. Kubernetes

Kubernetes[3] ist ein Werkzeug zur Orchestrierung von Containeranwendungen. Ursprünglich von Google entwickelt handelt es sich bei Kubernetes heute um ein Open-Source Projekt,

[2] https://hub.docker.com
[3] https://kubernetes.io/

das von der Linux Foundation[4] betrieben und weiterentwickelt wird. Kubernetes übernimmt Aufgaben wie das automatisierte Deployment von Containern im Cluster, die horizontale Skalierung von Anwendungen oder das automatische Neustarten ausgefallener Komponenten. Besonders Letzteres spielt für die Fehlertoleranz einer verteilten Anwendung eine wichtige Rolle. Im Folgenden werden wichtige Konzepte und Komponenten von Kubernetes beschrieben sowie Begrifflichkeiten erläutert.

Pod

Ein Pod stellt die kleinste deploybare Einheit in einem Kubernetes Cluster dar und ist eines der wichtigsten Konzepte von Kubernetes. Ein Pod besteht aus einem oder mehreren Containern mit gemeinsamen Speicher und Netzwerkressourcen und entspricht somit einer Instanz einer Anwendung. Besteht ein Pod aus mehreren Containern, weisen diese meist eine hohe Kopplung auf. Als Containertechnologie kommt unter anderem Docker zum Einsatz, es ist jedoch auch die Nutzung anderer Containerumgebungen möglich. Pods werden durch Konfigurationsdateien spezifiziert, in denen unter anderem Metadaten wie Labels, aber auch Ressourcen wie das Docker-Image, Netzwerkkonfigurationen oder Umgebungsvariablen angegeben werden.

Ist für einen Anwendungsfall eine horizontale Skalierung vorgesehen, so kann dies in Kubernetes über Controller konfiguriert werden, die für die Erzeugung von Pods zuständig sind. Ein *Deployment* sorgt etwa dafür, dass eine spezifizierte Anzahl an Instanzen eines Pods läuft. Ein *StatefulSet* dagegen ist für zustandsbehaftete Anwendungen wie Datenbanken gedacht. Im Gegensatz zu *Deployments* weisen Pods hier eine Identität auf und werden beim Erstellen, Updaten und Löschen anders behandelt, um die Konsistenz des Dienstes sicherzustellen. Die Anzahl der Instanzen lässt sich im laufenden Betrieb aktualisieren, um so Ressourcen dynamisch anzupassen. Außerdem sind Rolling Updates möglich, in denen Pods mit einer neueren Version nach und nach alte Pods ersetzen und dabei die Verfügbarkeit des Dienstes beibehalten.

Master

Der Master ist eine Komponente, die für die Koordination des Kubernetes-Clusters zuständig ist. Dazu zählen Aufgaben wie Scheduling von Pods oder das Reagieren auf Ereignisse wie das Ausfallen eines Pods. Außerdem dient der Master als Schnittstelle des Clusters nach außen. Der Master kann auf jeder beliebigen Maschine im Cluster laufen, sollte jedoch nicht als Laufzeitumgebung für Benutzeranwendungen genutzt werden.

[4] https://www.linuxfoundation.org/projects/ Zugriff: 16.07.2018

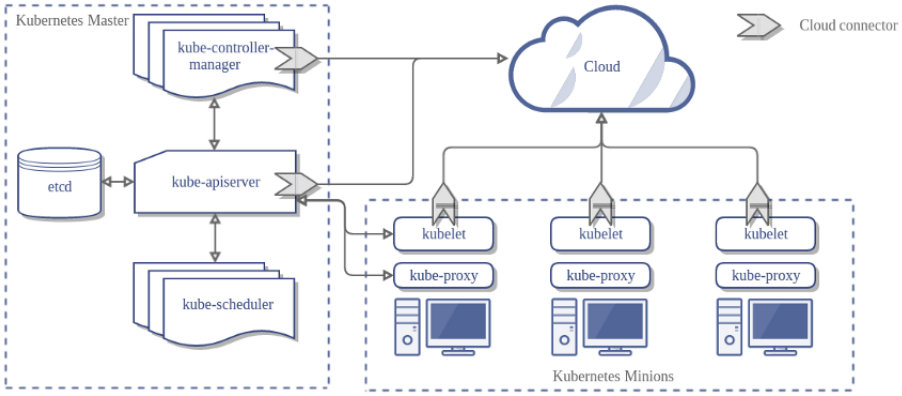


Abbildung 3.2.: Kubernetes Architektur (Quelle: Kubernetes (2018))

Node

Ein Node (*Minion*) ist ein Worker-Knoten und kann eine virtuelle oder physische Maschine sein. Ein Node enthält alle notwendigen Ressourcen, um Pods innerhalb des Clusters auszuführen. Ein Cluster kann aus beliebig vielen Nodes bestehen (vgl. Abb. 3.2).

Resource Requests

Resource Requests sind Ressourcenanfragen, die ein Pod an das Cluster stellen kann. Benötigt ein Pod eine Mindestmenge an Rechenpower oder Hauptspeicher, dann kann dies als Request innerhalb der Pod-Spezifikation angegeben werden. Kubernetes sorgt dafür, dass der Pod auf einem Node ausgeführt wird, der den entsprechenden Ressourcenbedarf decken kann.

Volumes

Da durch den Neustart eines Knotens oder durch Updates einer Anwendung ein neuer Pod erstellt wird, gehen alle Dateien verloren, die der alte Pod zu seiner Laufzeit gespeichert hat. Um diesen Datenverlust zu vermeiden, sollten Daten in einem persistenten Speicher abgelegt werden. In Kubernetes wird dies durch Volumes bewerkstelligt. Ein Volume ist eine Referenz auf persistenten Speicher und wird zur Laufzeit in das Filesystem des Pods gemountet. Innerhalb des gemounteten Verzeichnisses kann ein Pod Dateien lesen und schreiben.

Abbildung 3.3.: Streaming Modell in Spark (Quelle: Spark (2018b))

3.4.3. Apache Spark

Apache Spark[5] ist ein Processing Framework, das für die Verarbeitung von Daten in Clustern entwickelt wurde. Ursprünglich entwickelt am AMPLab der University of California (Berkley) gehört Spark heute zur Apache Software Foundation und wird als Open Source Projekt weiterentwickelt. Zentraler Bestandteil von Spark ist der Spark-Core, der für grundlegende Aufgaben wie Scheduling, I/O und Fehlertoleranz zuständig ist und grundlegende Schnittstellen zur Verarbeitung von Datensätzen anbietet. Auf dem Spark-Core sind wiederum weitere Schnittstellen aufgesetzt, die für verschiedene Szenarien bei der Big Data Verarbeitung entwickelt wurden. Neben *GraphX* für die Graphverarbeitung existieren außerdem die Schnittstellen *MLib* für den Bereich Maschinelles Lernen, *Spark SQL* für SQL-Abfragen auf Spark-Daten und *Spark Streaming* für die Verarbeitung von Datenströmen.

Die zentrale und wichtigste Datenstruktur in Spark ist das *Resilient Distributed Dataset (RDD)* (vgl. Zaharia u. a. (2012)), in welchem Daten in-Memory verteilt und fehlertolerant gespeichert werden. Auf *RDDs* können Transformationen (z.B. Map oder Filter) und Aktionen (z.B. Reduce oder Count) ausgeführt werden (vgl. Spark (2018a)). Die Fehlertoleranz wird dadurch ermöglicht, dass jede RDD durch einen *Directed Acyclic Graph (DAG)* weiß, wie sie berechnet wurde und im Falle eines Ausfalls die Daten der ausgefallenen Komponente neu berechnet werden können. RDDs kommen in Spark sowohl für Batch- als auch für Streamverarbeitung zum Einsatz. Um mithilfe von RDDs einen Stream abzubilden, teilt Spark den Datenstrom in sogenannte Micro-Batches, einer Sequenz von RDDs, auf (vgl. Spark (2018b) und Abb. 3.3). Die einzelnen Batches werden durch die Spark Engine weiterverarbeitet und können anschließend einer Datensenke zugeführt werden.

GraphX (Gonzalez u. a. (2014)) bietet innerhalb des Spark Frameworks eine Abstraktion für die Graphverarbeitung an. Ein Graph ist in Spark ein gerichteter Multigraph, dessen Knoten und Kanten Werte enthalten können und somit die Eigenschaft eines Property-Graphen erfüllen

[5] https://spark.apache.org/

(s. Abschnitt 2.1). Intern besteht dieser Graph aus einer *VertexRDD* für Knoten und einer *EdgeRDD* für Kanten sowie jeweils deren Attribute. Aufbauend auf diesem Graphen bietet GraphX Operationen etwa zur Transformation von Knoten und Kanten, aber auch Aktionen wie das Zählen von Knoten, Kanten oder Knotengraden. Durch eine Pregel-Schnittstelle können außerdem vertex-zentrierte Algorithmen implementiert werden. Einige Algorithmen wie Triangle Count, Connected Components und PageRank sind bereits in GraphX implementiert und können vom Benutzer von GraphX auf eigene Graphen angewendet werden. Für die Echtzeitverarbeitung von Graphen bietet GraphX zum aktuellen Zeitpunkt des Verfassens dieser Arbeit keine Schnittstelle.

3.4.4. Apache Flink

Apache Flink[6] ist ähnlich wie Apache Spark ein Open Source Framework für Big Data Verarbeitung in Clustern. Im Gegensatz zu Spark wird der Fokus jedoch auf Verarbeitung von Datenströmen gelegt, was sich darin äußert, dass Flinks Kern aus einer Stream Processing Engine besteht. Es ist aber grundsätzlich auch eine Batch-Verarbeitung von Daten möglich, wobei diese auf der Streaming Engine aufsetzt (vgl. Flink (2018b)). Ähnlich wie in Spark existiert in Flink für die Batchverarbeitung eine verteilte und fehlertolerante Datenstruktur, in Flink als *DataSet* bezeichnet. DataSets lassen sich ähnlich wie in Spark durch Transformationen und Aktionen weiterverarbeiten. Für die Verarbeitung durch iterative Algorithmen bietet Flink eine *Iteration-API* (vgl. Flink (2018a)). Diese API führt mit *Iterate* und *Delta Iterate* zwei Operatoren ein, mit denen sich DataSets verarbeiten lassen.

Gelly ist eine Graphabstraktion und bietet ähnlich wie GraphX diverse Funktionen zur Verarbeitung von Graphen in Flink. So können Graphen ähnlich wie in Spark durch Transformationen und Aktionen weiterverarbeitet werden. Für iterative Algorithmen bietet Gelly neben einer Pregel-Schnittstelle Iterationen nach den Modellen *Scatter-Gather* und *Gather-Sum-Apply*. Genauere Unterschiede zwischen diesen Schnittstellen sind unter Flink (2018c) aufgeführt. Ebenfalls implementiert Flink eine Reihe von Graphalgorithmen, darunter PageRank, Single-Source Shortest-Path, Triangle Count und Clustering (vgl. Flink (2018d)).

Mit Gelly-Streaming bieten Kalavri u. a. (2017) eine Flink-Schnittstelle zur Echtzeitverarbeitung von Graphen an. Gelly-Stream baut auf Flinks DataStream-API auf implementiert grundlegende Funktionen wie das Zählen von Knoten- und Kantengraden, aber auch Algorithmen wie Connected Components oder Triangle Count. Wie in Kapitel 2 beschrieben modelliert auch Gelly-Streaming einen Graph als Kantenstrom. Da es sich beim Kantenstrom intern um

[6] https://flink.apache.org/

ein DataStream handelt, ist ein Zugriff auf Flinks DataStream-API möglich. Somit ist auch eine Verteilung auf mehrere Rechner möglich und die Fehlertoleranz des Flink-Systems gegeben.

3.4.5. Apache Storm

Apache Storm[7] ist ein Open Source Processing Framework für die Echtzeitverarbeitung von Datenströmen. Im Vergleich zu Spark und Flink, die jeweils neben der Verarbeitung von Datenströmen auch eine Verarbeitung von Batches ermöglichen, ist Storm lediglich für die Verarbeitung von Datenströmen ausgelegt. Ähnlich wie Spark und Flink bewirbt auch Storm die horizontale Skalierbarkeit sowie die Fehlertoleranz des Systems. Storm verfügt über eine Stream-API, mit der die Verarbeitung von Datenströmen ermöglicht wird. Neben Transformationen wie Map und Filter und Aktionen wie Aggregate, Reduce und Group ist hier ebenfalls eine Unterteilung von Datenströmen in Fenster möglich, in denen bei der Verarbeitung jeweils nur die im Fenster enthaltenen Datensätze betrachtet werden.

Zum Zeitpunkt des Verfassens dieser Arbeit verfügt Storm über keine integrierte Graph-schnittstelle, dennoch zeigen Taxidou und Fischer (2013) durch ihre Arbeit eine prinzipielle Möglichkeit der Nutzung von Storm zur Graphverarbeitung.

3.4.6. SMACK

Bei SMACK handelt es sich um einen Software-Stack für die Verarbeitung von Big Data. Beim Begriff handelt es sich um ein Akronym, das die Namen der enthaltenen Technologien enthält (**S**park, **M**esos, **A**kka, **C**assandra und **K**afka).

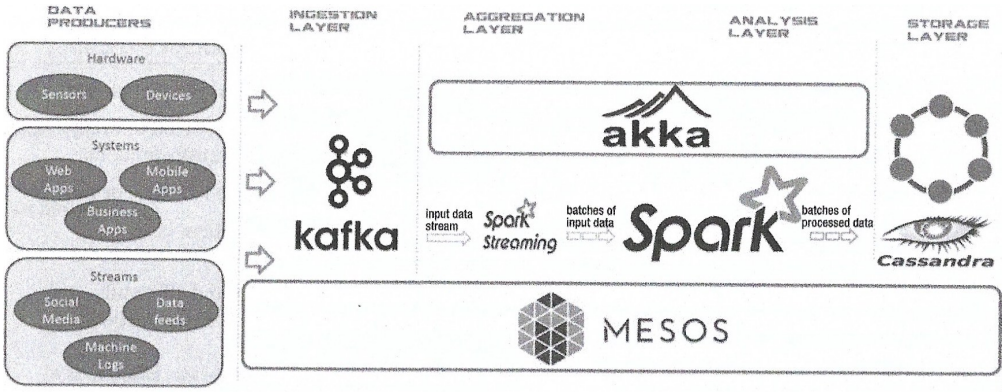


Abbildung 3.4.: Übersicht des SMACK-Stacks (Quelle: Estrada und Ruiz (2016))

[7] https://storm.apache.org/

Alle fünf Technologien sind Open Source und bis auf Akka handelt es sich um Projekte der Apache Software Foundation (vgl. Estrada und Ruiz (2016)). Eine grobe Übersicht über den Stack sowie das Zusammenspiel zwischen den Komponenten kann Abb. 3.4 entnommen werden. Während Mesos die Laufzeitumgebung für den Stack zur Verfügung stellt, dient Kafka als Datenquelle für die darauffolgende Verarbeitung durch Spark oder Akka. Die Datenspeicherung erfolgt in der verteilten Datenbank Cassandra.

Mesos

Apache Mesos ist ein Kernel für verteilte Systeme und abstrahiert Ressourcen wie CPU, RAM und Speicher einzelner Maschinen. Dadurch wird ermöglicht, verteilte Systeme skalierbar und fehlertolerant aufzusetzen. Innerhalb des SMACK-Stacks ist Mesos für die Orchestrierung sowie Skalierung der anderen Komponenten zuständig.

Akka

Akka ist ein Werkzeug zur Entwicklung verteilter, fehlertoleranter und nachrichtengetriebener Anwendungen. Akka nutzt als Laufzeitumgebung die Java Virtual Machine (JVM), was eine Entwicklung in den Programmiersprachen Java und Scala ermöglicht. Als Basis von Akka dient das Aktorenmodell, welches ein Programm in nebenläufige Aktoren unterteilt, die ausschließlich über Nachrichten Informationen miteinander austauschen.

Cassandra

Apache Cassandra ist eine verteilte, skalierbare und fehlertolerante Datenbank für große Datenmengen und zählt zu den NoSQL-Datenbanken. Die Datenbank ist im Gegensatz zu relationalen Datenbanken spaltenorientiert, was vor allem für Anwendungen günstig ist, die vor allem mit spaltenbasierten Anfragen wie Aggregationen einzelner Spalten arbeiten. Innerhalb des SMACK-Stacks dient Cassandra zur Speicherung operationaler Daten und kann als Datenquelle für die Präsentationsschicht zum Einsatz kommen.

Kafka

Apache Kafka ist ein verteilter Message-Broker, der ursprünglich für die Behandlung verteilter Logdaten entwickelt wurde. Der Hauptzweck eines Message-Brokers besteht darin, Daten oder Nachrichten aus verschiedenen Datenquellen effizient an Datenkonsumenten weiterzuleiten. Konsumenten können dabei Anwendungen, Stream-Processing Engines oder Datenbanken sein. Kafka funktioniert nach dem Publish/Subscribe-Prinzip, nach welchem Datenproduzenten ihre

Daten in eine Message-Queue schreiben (publish) und Datenkonsumenten Daten aus dieser Queue lesen (subscribe) können. Hierfür bildet Kafka verschiedene Datenströme als Topics ab und ermöglicht somit eine logische Trennung nach einzelnen Themen. Vorteile bei der Nutzung von Message-Brokern ergeben sich insofern, als dass Producer und Consumer sich nicht kennen müssen und somit voneinander entkoppelt sind. Lediglich eine Schnittstelle für übermittelte Nachrichten sollte für beide Seiten vereinbart werden. Durch Partitionieren von Datenströmen und Gruppieren von Konsumenten können außerdem Szenarien wie Lastverteilung realisiert werden.

Neben der Funktion als Message-Broker bietet Kafka mit Kafka Streams[8] eine API zur Verarbeitung von Datenströmen an. Die Schnittstelle arbeitet direkt auf den Topics in Kafka und ermöglicht damit eine Verarbeitung ohne eine separate Stream-Processing Engine wie Spark oder Flink. Kafka Streams arbeitet mit den Datenstrukturen *KStreams* und *KTable*. Während Erstere einen kontinuierlichen Datenstrom von Fakten liefert, bildet ein *KTable* einen Datenstrom von Updates einzelner Datensätzen ab. Beide Strukturen lassen sich durch Operationen weiterverarbeiten, um damit bestimmte Anwendungsfälle abzubilden.

3.4.7. Husky

Husky ist ein Technologie-Stack, der in Marović und Marić (2016) als Werkzeug zur Big Data Verarbeitung vorgestellt wurde. Als Datenquelle der Verarbeitungspipeline kommen Kafka bzw. Hadoop für Batch- bzw. Streamverarbeitung zum Einsatz. Als Processing Engine dient Apache Spark zur Verarbeitung von Batches bzw. Datenströmen, zur Speicherung von Ergebnissen wird ElasticSearch verwendet (vgl. Abb. 3.5 und Abb. 3.6). Im Folgenden soll detaillierter auf das Tool ElasticSearch eingegangen werden, da Spark und Kafka bereits in vorhergehenden Abschnitten behandelt wurden.

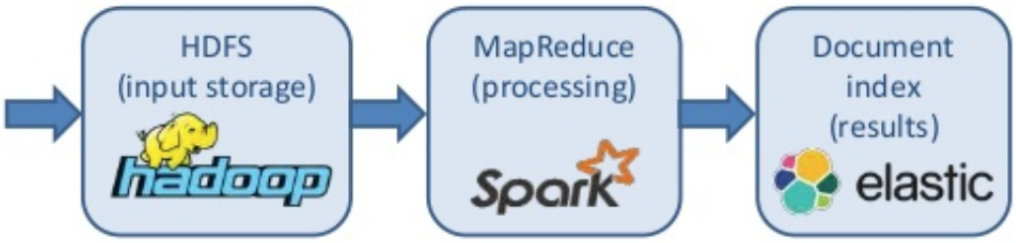


Abbildung 3.5.: Batch Verarbeitungspipeline Husky (Quelle: Marović und Marić (2016))

[8] https://kafka.apache.org/documentation/streams/

ElasticSearch

ElasticSearch[9] ist eine verteilte Such- und Analytik-Engine, die Daten verteilt in Form indizierter Dokumente speichert und Anfragen an diese Daten erlaubt. Hierfür wird eine REST-API angeboten, mit der Dokumente im JSON-Format in ElasticSearch abgelegt werden können. Durch eine Indizierung ermöglicht das Tool dann eine effiziente Suche nach vorhandenen Dokumenten. Aufbauend auf ElasticSearch ermöglicht das Tool Kibana[10] eine Visualisierung der gespeicherten Daten. Dadurch wäre im Rahmen dieser Arbeit die Möglichkeit gegeben, Ergebnisse anschaulich zu visualisieren.

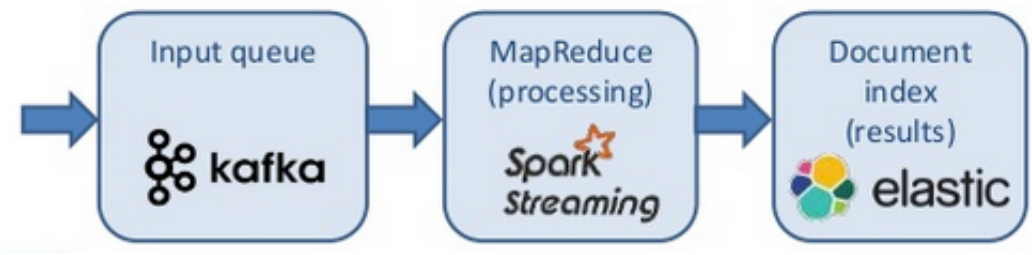


Abbildung 3.6.: Stream Verarbeitungspipeline Husky (Quelle: Marović und Marić (2016))

[9] https://www.elastic.co/de/products/elasticsearch
[10] https://www.elastic.co/de/products/kibana

4. Entwurf

Dieses Kapitel widmet sich dem Entwurf der Plattform, die in Kapitel 3 spezifiziert wurde. Im ersten Schritt findet die Auswahl der Architektur statt, nach der die Plattform realisiert wird. Daraufhin wird mithilfe der Kontextsicht die Umgebung festgelegt, in die das System eingebettet ist, sowie die beteiligten Akteure. Außerdem werden Komponenten spezifiziert sowie deren Schnittstellen untereinander festgelegt. Die Modellierung der verschiedenen Sichten erfolgt hierbei auf Grundlage von Starke (2015).

4.1. Architektur

Im ersten Schritt soll die Architektur des Systems festgelegt werden. Die Architektur einer Software legt Komponenten sowie deren Zusammenspiel innerhalb des Systems fest. Hierfür werden im folgenden zwei Architekturen vorgestellt, die sich an die Big Data Verarbeitung richten, die Lambda- und die Kappa-Architektur. Anschließend folgt eine begründete Auswahl einer der beiden Architekturen.

4.1.1. Lambda-Architektur

Bei der Lambda-Architektur handelt es sich um eine Architektur zur Verarbeitung von Big Data. Wichtige Komponenten in der Lambda-Architektur sind hierbei der Data-Ingestion-Layer, der Batch-Layer, der Speed-Layer und der Serving Layer. Die Lambda-Architektur geht ursprünglich aus Marz (2011) hervor.

Der Data-Ingestion-Layer dient als Eintrittspunkt in das System, von dem aus alle zu verarbeitenden Daten zentral gesammelt und sowohl an den Speed- als auch an den Batch-Layer gereicht werden. Der Batch-Layer ist für die Verwaltung des Master-Datasets zuständig, in dem alle eingetroffenen Daten persistent gespeichert werden. Die Datenverarbeitung durch den Batch-Layer erfolgt innerhalb festgelegter Zeitintervalle auf Daten des Master-Dataset. Sobald die Batchverarbeitung abgeschlossen ist, werden die berechneten Ergebnisse im Serving-Layer gespeichert, der als Austrittspunkt des Systems dient und dem Nutzer bzw. Analysten die berechneten Ergebnisse zur Verfügung stellt.

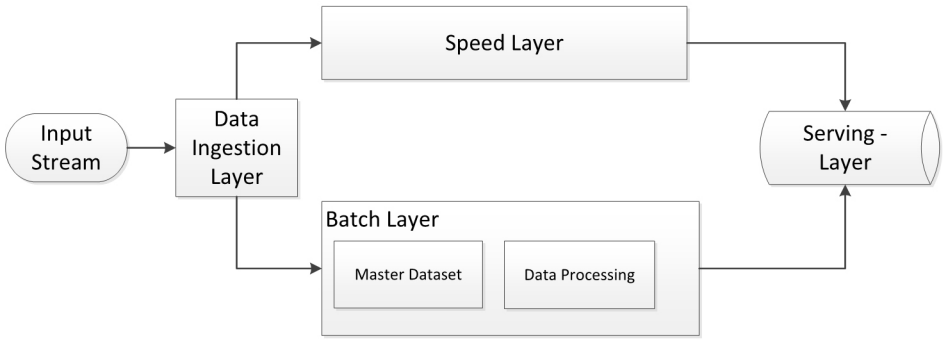


Abbildung 4.1.: Lambda-Architektur (Quelle: Berle (2017))

Der Speed-Layer ist dafür zuständig, ankommende Daten im Gegensatz zum Batch-Layer in Echtzeit zu verarbeiten. Dies ist beispielsweise dann nötig, wenn Ergebnisse sofort benötigt werden und an Wert verlieren, je später die Verarbeitung erfolgt. Der Speed-Layer bezieht seine Daten lediglich aus dem Input-Stream und verarbeitet sie meist In-Memory ohne persistente Speicherung. Die Speicherung der Ergebnisse erfolgt ebenfalls im Serving-Layer, welchem es durch die Ergebnisse beider Verarbeitungslayer möglich ist, verschiedene Anfragen zu beantworten.

4.1.2. Kappa-Architektur

Die Kappa-Architektur ist eine Sonderform der Lambda-Architektur und wurde ursprünglich von Jay Kreps entworfen (Berle (2017)). Im Vergleich zur Lambda-Architektur fällt hier der Batch-Layer weg, wodurch nur noch der Speed-Layer für die Datenverarbeitung sowie die Versorgung des Serving-Layers mit Ergebnissen zuständig ist. Dies ist dann von Vorteil, wenn die Batchverarbeitung nicht benötigt wird. Ebenfalls muss die Anwendungslogik nur noch im Speed-Layer implementiert werden, was wiederum Auswirkungen auf den Entwicklungsaufwand sowie die Wartbarkeit des Systems hat.

Der Master-Dataset wird in der Kappa-Architektur beibehalten. Im Gegensatz zur Lambda-Architektur dient dieser in der Kappa-Architektur dazu, bereits vergangene Daten zu einem späteren Zeitpunkt erneut zu verarbeiten, etwa wenn sich die Anwendungslogik ändert. Die erneute Verarbeitung wird dadurch ermöglicht, indem der Master-Dataset dem Speed-Layer in Form eines Datenstroms erneut zugeführt wird (vgl. Abb. 4.2).

Da für den in dieser Arbeit zu realisierenden Anwendungsfall keine Batchverarbeitung erforderlich ist, wird das System demzufolge in der Kappa-Architektur realisiert. Bei der Auswahl der Technologien für die Realisierung ist dabei darauf zu achten, dass die einzelnen Kompo-

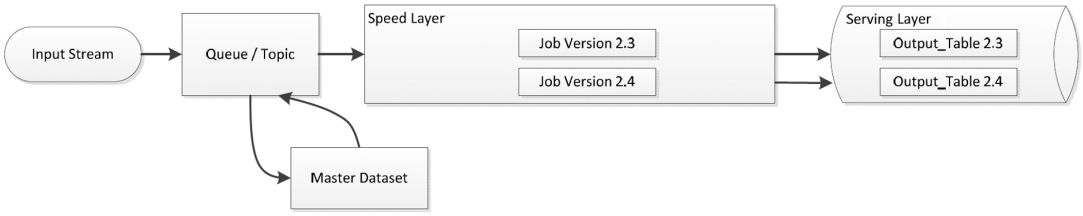


Abbildung 4.2.: Kappa-Architektur (Quelle: Berle (2017))

nenten ohne großen Aufwand zusammengeschaltet werden können, um den Entwicklungs- und Wartungsaufwand möglichst gering zu halten.

4.2. Kontextsicht

Die Kontextsicht stellt die Umgebung dar, in die das System eingebettet ist. Ist beispielsweise eine Software in einer Unternehmenslandschaft zu entwerfen, so muss bekannt sein, welche Rolle sie innerhalb dieser Landschaft annimmt und mit welchen Nachbarsystemen das System interagiert. Zusätzlich sind Nutzergruppen anzugeben, die in direkter oder indirekter Form mit dem System interagieren. Das System selber wird in dieser Sicht als Blackbox dargestellt, aus der keine weiteren Details über die Architektur hervorgehen.

Da die Hauptaufgabe der zu realisierenden Plattform die Echtzeitverarbeitung von Graphen ist, stellen auf der einen Seite beliebig viele Datenquellen Nachbarsysteme dar. Da sich Datenquellen durch die Struktur ihrer Daten unterscheiden, muss sichergestellt sein, dass der zu verarbeitende Datenstrom einem einheitlichen Format entspricht. Diese Datenbereinigung muss geschehen, bevor der Datenstrom dem System zugeführt wird, um die Unabhängigkeit der Plattform von der Datenquelle zu gewährleisten. Die Bereinigungskomponenten werden zwischen System und Datenquellen platziert. Die eigentliche Graph Processing Plattform ist damit nicht für die Datenbereinigung zuständig, sondern bietet lediglich eine Schnittstelle für eingehende Graphdatenströme.

Auf der anderen Seite stellen die Nutzer des Systems die Zielgruppe dar. In Falle dieser Arbeit wird diese Rolle von Data Analysts eingenommen, die aus der Visualisierung der berechneten Metriken entsprechende Aussagen über die Entwicklung eines Graphen ableiten können sollen. Die gesamte Kontextsicht kann Abb. 4.3 entnommen werden.

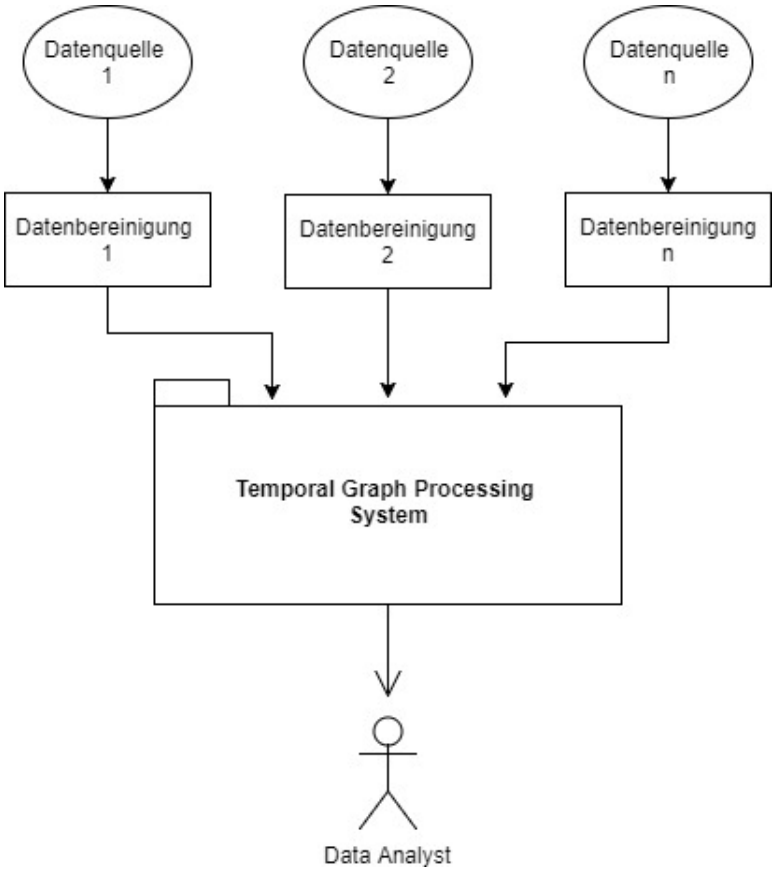


Abbildung 4.3.: Kontextsicht des zu realisierenden Systems

4.3. Bausteinsicht

Im Folgenden soll die Bausteinsicht des Systems beschrieben werden. Innerhalb der Bausteinsicht erfolgt die Modellierung des Systems als Whitebox, in der Komponenten des Systems sowie das Zusammenspiel untereinander abgebildet werden.

4.3.1. Datenbereinigung

Wie in vorherigen Abschnitten beschrieben wird die Datenaufbereitung nicht vom System übernommen, sondern muss bereits vorher erfolgen, um sicherzustellen, dass das System den Datenstrom verarbeiten kann. Um dem Nutzer des Systems die Anbindung der Datenquelle an das System zu vereinfachen, soll ein Konnektor angeboten werden. Wichtig ist dabei vor allem, dass dieser von der konkreten Implementierung des Data-Ingestion-Layer abstrahiert und sicherstellt, dass der Datenstrom in einem gültigen Format übergeben wird.

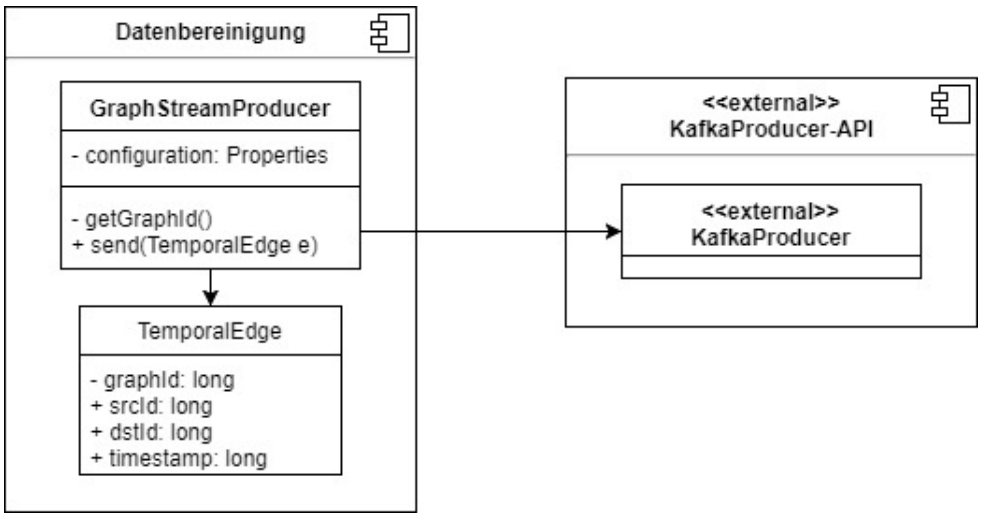


Abbildung 4.4.: Schnittstelle der Datenbereinigung

Abb. 4.4 stellt den Entwurf der Komponente Datenbereinigung am konkreten Fall von Kafka dar. Zentraler Bestandteil dieser Komponente ist die Klasse `GraphStreamProducer`. Diese abstrahiert vom Kafka-Producer und ermöglicht das Senden von temporalen Kanten an das System. Da Graphdatenströme außerdem durch eine ID voneinander unterschieden werden, erfolgt bereits hier die Anreicherung mit einer Graph-ID. Diese wird beim System angefragt, welches wiederum die Eindeutigkeit der ID sicherstellen muss.

4.3.2. Data Ingestion Layer

Als Technologie für den Eingangspunkt der Datenströme in das System sowie die Verwaltung der Daten im Master-Dataset erfolgte die Entscheidung zugunsten von Apache Kafka. Durch die Unterteilung der Datenströme in Topics kann eine logische Trennung nach verschiedenen Datenquellen durchgeführt werden (**F1**). Durch die persistente Speicherung der Datenströme eignet sich Kafka außerdem als Kandidat für den Master-Dataset. Durch Kafka erhöht sich die Fehlertoleranz des Systems (**N2**), da ein Topic und somit auch die Verarbeitung reproduziert werden kann und Kafka-Topics auf verschiedene Broker repliziert werden können. Durch die Zusammenfassung von Topic und Master-Dataset entfällt außerdem potenzieller Aufwand für die Wartung einer zusätzlichen Komponente für den Master-Dataset.

4.3.3. Data Processing Layer

Für die Verarbeitung der Datenströme kommt Apache Flink zum Einsatz. Dies erfolgt aus dem Grund, da Flink mit Gelly-Streaming bereits eine grundlegende Schnittstelle zur Verarbeitung von Graphdatenströmen anbietet. An diesem Punkt sollte jedoch erwähnt werden, dass es sich zum Zeitpunkt des Verfassens dieser Arbeit um eine experimentelle API handelt, die aus einem Forschungsprojekt der KTH Royal Institute of Technology Stockholm hervorging (Bali (2015)). Im Rahmen dieser Arbeit soll deshalb vorhandene Funktionalität genutzt und fehlende Funktionalität gegebenenfalls ergänzt werden.

Ein wichtiger Punkt in Flink ist die Fehlertoleranz des Systems. In Flink wird Fehlertoleranz durch Checkpointing realisiert. Voraussetzung hierfür ist eine Datenquelle, die Datensätze in einem bestimmten Rahmen erneut abspielen kann. Ebenfalls ist ein persistenter Speicher nötig, der Zustände von Flink-Berechnungen speichern kann (vgl. Flink (2018e)). Für den ersten Punkt liefert Kafka bereits eine hinreichende Möglichkeit. Der zweite Punkt soll im Rahmen dieser Architektur durch das verteilte Dateisystem HDFS realisiert werden. Checkpointing soll die Erfüllung der Anforderung **N2** unterstützen.

Der konkrete Entwurf der Processing-Komponente kann Abb. 4.5 entnommen werden. Für die Verarbeitung der Metriken ist die Klasse `GraphMetricProcessor` zuständig. In dieser werden alle zu berechnenden Metriken gesammelt. Dafür implementiert jede Metrik die abstrakte Klasse `GraphMetric`, die sowohl eine Funktion zur Verarbeitung als auch eine Funktion zur Speicherung von Ergebnissen vorgibt. Durch diesen Entwurf soll die Erweiterbarkeit durch weitere Metriken vereinfacht werden.

Da Flink mit Jobs arbeitet, die verschiedene Verarbeitungsaufträge abbilden, kann die unabhängige Verarbeitung verschiedener Graphen erfolgen (**F2**). Durch die Verteilung von Jobs

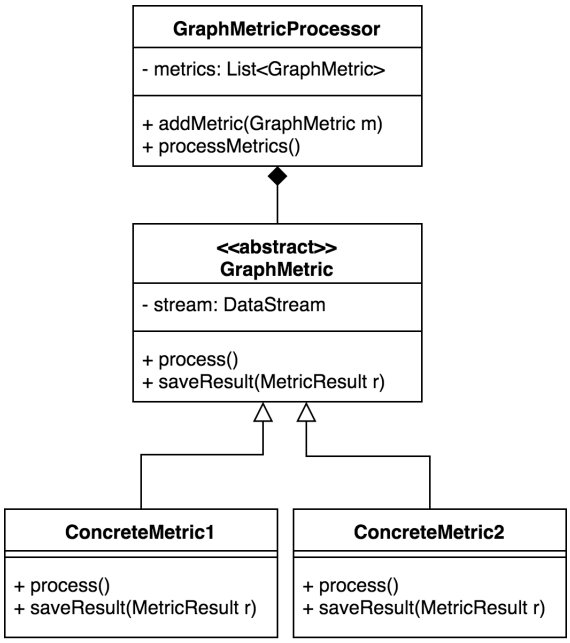


Abbildung 4.5.: Datenmodell der Data Processing-Komponente

auf mehrere Taskmanager wird eine horizontale Skalierbarkeit ermöglicht (**N1**). Durch die Möglichkeit der Verarbeitung von Datenströmen nach *EventTime* wird außerdem die Erfüllung der Anforderungen **F5** und **F6** unterstützt. Windowing ermöglicht die Unterteilung in konfigurierbare Zeitfenster, um Ergebnisdatenströme zu reduzieren (**F7**).

4.3.4. Serving Layer

Die Speicherung der Ergebnisse erfolgt in ElasticSearch. Dafür nimmt ElasticSearch die berechneten Daten in Form von JSON-Dokumenten über eine REST-API entgegen und speichert sie fehlertolerant und verteilt in einem Cluster. Die Indizierung ermöglicht ein effizientes Durchsuchen der Daten und dementsprechend eine effektive Visualisierung durch das Tool Kibana. Das Ziel hierbei ist die Visualisierung der Ergebnisse in Quasi-Echtzeit. Kibana verspricht durch Dashboards und diverse Diagramme eine Umsetzung der Anforderungen **F9** und **F10** mit vergleichsweise geringem Aufwand im Vergleich zu einer eigenen Umsetzung. Durch ein konfigurierbares Zeitintervall kann außerdem eine automatische Aktualisierung der Dashboards erfolgen. Das Datenmodell ist Abb. 4.6 zu entnehmen. Die Klasse `MetricResult` soll sicherstellen, dass die Ergebnisse im korrekten Format an den Serving-Layer übermittelt werden.

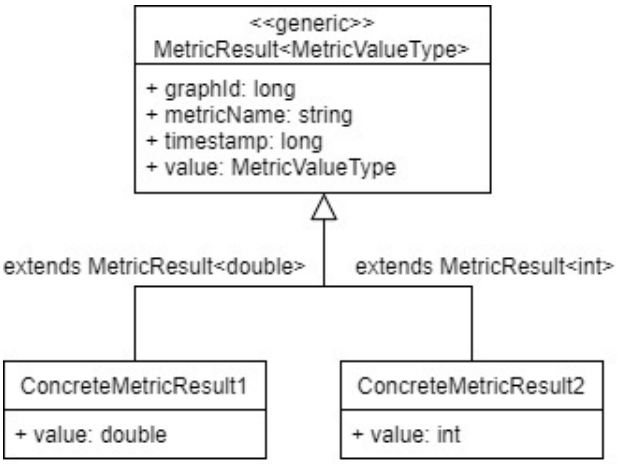


Abbildung 4.6.: Datenmodell der Ergebnisse

4.3.5. Schnittstellen

In einer komponentenbasierten Architektur sind einzelne Komponenten für einen fest definierten und abgegrenzten Ausgabenbereich zuständig. Damit ein System komponentenübergreifende Aufgaben erfüllen dann, müssen die verschiedenen Komponenten miteinander kommunizieren. Hierfür sind Schnittstellen notwendig, die zum einen technisch einen Datenaustausch zwischen den Komponenten ermöglichen, jedoch auch fest definierte Operationen vorgeben, die eine Komponente nach außen anbietet und von anderen Komponenten genutzt werden.

Die gesamte Bausteinsicht kann Abb. 4.7 entnommen werden. Diese Abbildung zeigt die verschiedenen Komponenten des Systems und ihre Beziehungen untereinander. Im Folgenden soll erläutert werden, welche Schnittstellen bzw. Konnektoren zum Einsatz kommen, um eine reibungslose Kommunikation zwischen den Komponenten sicherzustellen. Apache Kafka bietet als Message-Broker sowohl Konnektoren zum Produzieren als auch zum Konsumieren von Datenströmen an (Producer- bzw. Consumer-API). Erstere dient im Rahmen dieser Arbeit als Schnittstelle zwischen der Datenbereinigung und dem System. Im nächsten Schritt muss eine Kommunikation zwischen Kafka und Flink möglich sein. Flink bietet als Stream-Processing Engine diverse Konnektoren an, darunter auch für Kafka, HDFS und ElasticSearch (Flink (2018f)). Eine Implementation der technischen Aspekte der einzelnen Konnektoren entfällt dadurch. Lediglich das Datenformat der auszutauschenden Daten muss festgelegt werden, was bereits in vorherigen Abschnitten erfolgte.

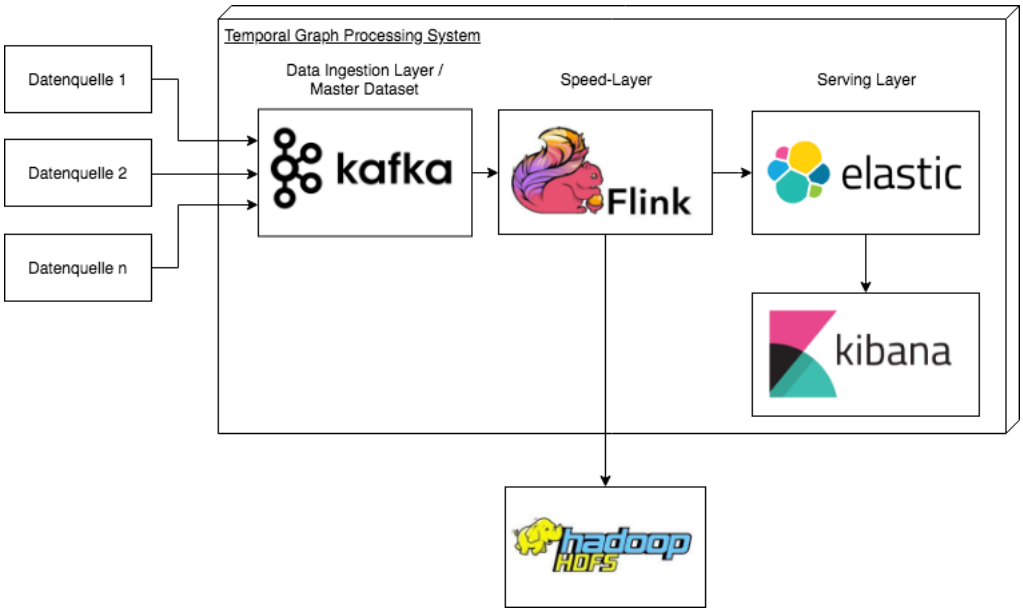


Abbildung 4.7.: Bausteinsicht des zu realisierenden Systems

4.4. Verteilungssicht

Die Verteilungssicht dient dazu, die Verteilung einzelner Komponenten auf die zugrundeliegende Infrastruktur abzubilden. Im Rahmen dieser Arbeit dient Kubernetes als Laufzeitumgebung der Plattform, womit Kubernetes für die Verteilung der einzelnen Komponenten zuständig ist. Die Modellierung der Verteilungssicht erfolgt aus diesem Grund auf logischer Ebene durch Pods anstatt auf physischer Ebene durch Rechner, da Pods die zentralen Einheiten in Kubernetes darstellen und aufgrund der Architektur von Kubernetes nicht zwangsläufig auf die physische Verteilung der Pods geschlossen werden kann.

Aus Abb. 4.8 kann die Verteilung der Kafka-Komponenten entnommen werden. Ein Kafka-Cluster besteht aus beliebig vielen Brokern, die durch den Konfigurations- und Nameservice Zookeeper[1] koordiniert werden. Durch diese Architektur soll innerhalb Kafkas die Ausfallsicherheit erhöht werden, indem eine Replikation der Datenströme auf verschiedene Knoten erfolgt. Ebenfalls kann dadurch eine Lastverteilung realisiert werden, indem ein Datenstrom in mehrere Partitionen unterteilt wird, die physisch auf mehrere Broker verteilt sind. Aus Abb. 4.9 kann die Verteilung der Komponenten eines Flink-Clusters entnommen werden. Ein Flink-Cluster besteht aus einem Jobmanager, der die Rolle des Masters einnimmt und für die Koordination des Clusters sowie der Flink-Jobs zuständig ist. Taskmanager übernehmen

[1] https://zookeeper.apache.org/

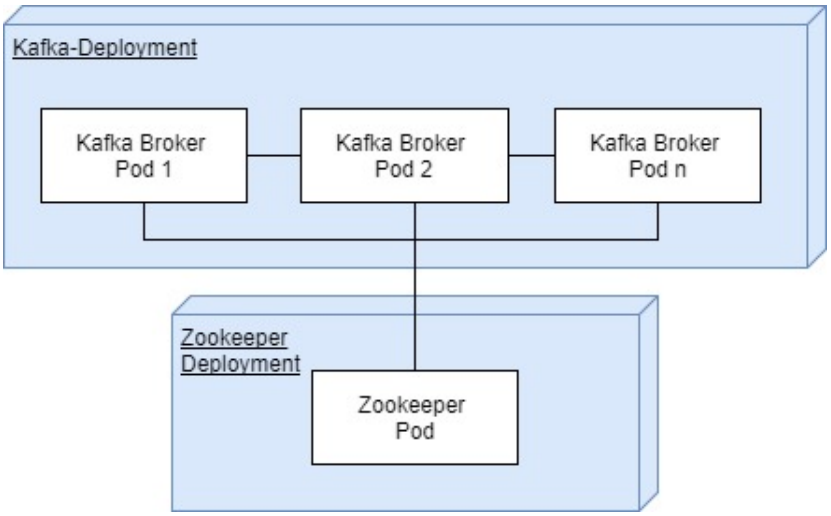


Abbildung 4.8.: Verteilungssicht des Kafka-Clusters

im Cluster die Aufgabe eines Workers und sind letztendlich für die Verarbeitung von Jobs zuständig.

Abb. 4.10 stellt die Architektur eines ElasticSearch Clusters dar. Ein Cluster besteht hier aus beliebig vielen Nodes, aus denen zu Beginn ein Knoten als Master-Node gewählt wird, welcher die Koordination des Clusters übernimmt. Es stellt sich jedoch die Frage, ob im Rahmen dieser Arbeit eine Skalierung von ElasticSearch notwendig ist, da der Fokus dieser Arbeit auf der Verarbeitung von Graphdatenströmen liegt.

4.5. Laufzeitsicht

Die Laufzeitsicht hat den Zweck, Anwendungsfälle des Systems abzubilden und dabei vor allem das Zusammenspiel der Komponenten untereinander in den Mittelpunkt zu stellen. Die Laufzeitsicht des Anwendungsfalls, welches innerhalb dieser Arbeit realisiert werden soll, kann Abb. 4.11 entnommen werden.

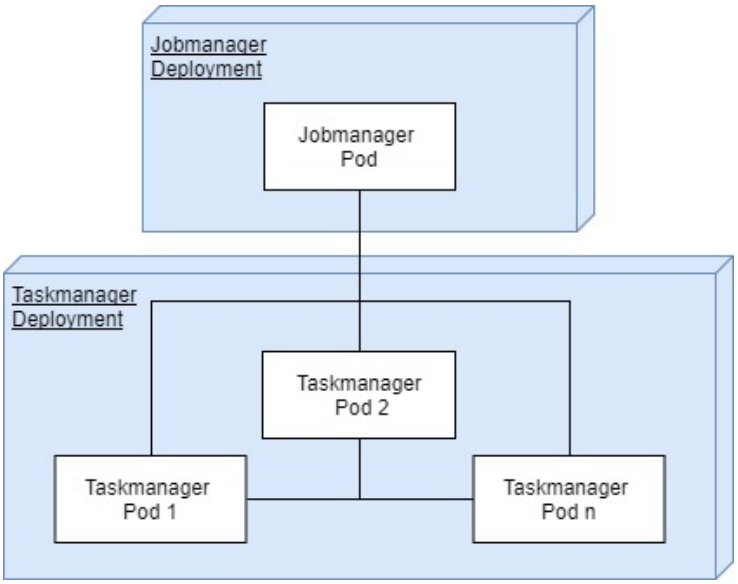


Abbildung 4.9.: Verteilungssicht des Flink-Clusters

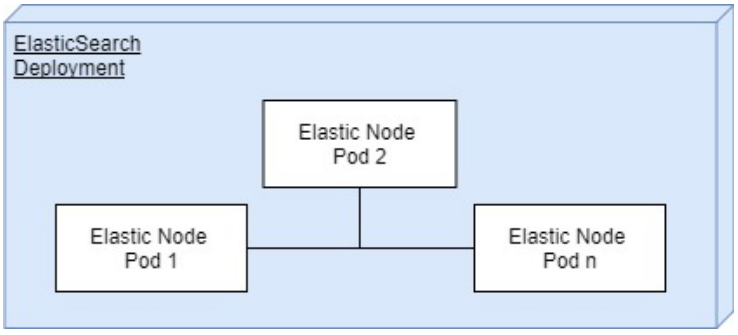


Abbildung 4.10.: Verteilungssicht des ElasticSearch-Clusters

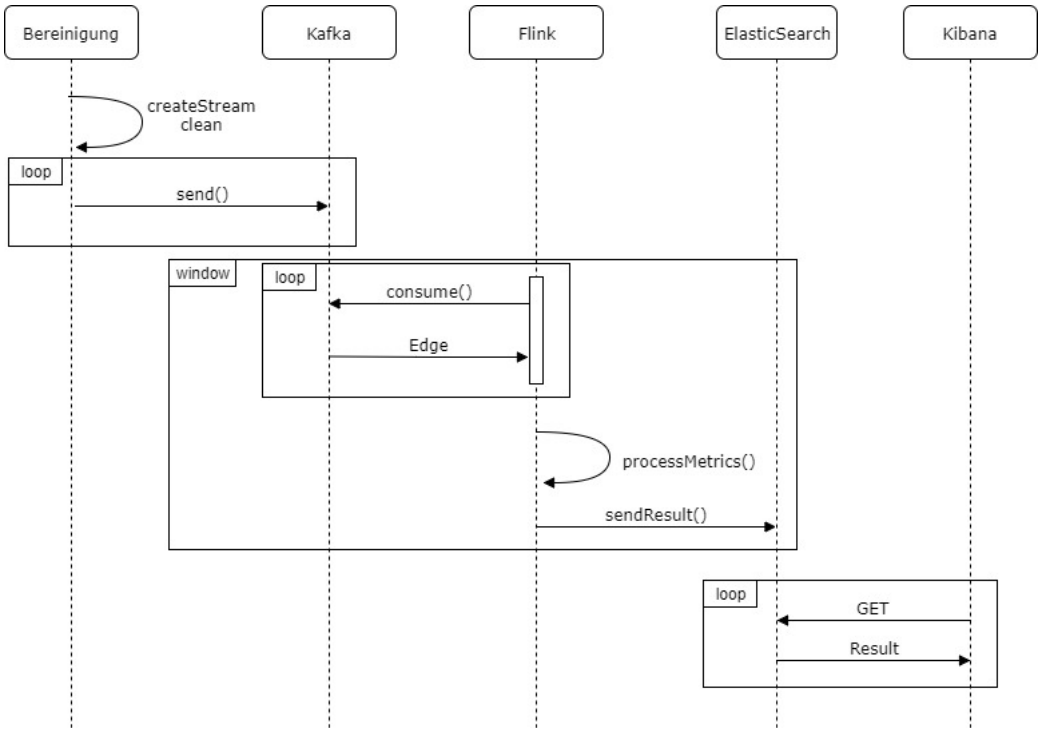


Abbildung 4.11.: Laufzeitsicht des Systems

Die Datenbereinigung ist innerhalb des Anwendungsfalls dafür zuständig, unabhängig von der konkreten Datenquelle einen (endlosen) Kantenstrom zu erzeugen, diesen für das System aufzubereiten und ihn letztendlich an das System zu senden. Bei einem Echtzeitdatenstrom geschieht dies im Prinzip in einer Endlosschleife. Flink ist für die Verarbeitung des Datenstroms zuständig und damit der Konsument des Datenstroms. Da Schnappschüsse der Metriken in bestimmten Zeitintervallen berechnet werden, werden innerhalb eines Intervalls alle Kanten gesammelt, die innerhalb dieser Zeit eintreffen. Sobald ein Intervall abgelaufen ist, erfolgt die inkrementelle Verarbeitung der Kanten und eine anschließende Speicherung der Ergebnisse in ElasticSearch. Dieser Schritt wird ebenfalls endlos wiederholt. Für die Visualisierung greift Kibana auf die REST-API von ElasticSearch zu und stellt die Ergebnisse in einer Weboberfläche dar. Dabei ist die Möglichkeit gegeben, Diagramme in bestimmten Zeitabständen zu aktualisieren. Solange die Weboberfläche aktiv ist, geschieht das Abrufen ebenfalls in einer Schleife.

5. Realisierung

Im diesen Kapitel soll die Realisierung des in Kapitel 3 spezifizierten und in Kapitel 4 entworfenen Systems thematisiert werden. Ziel dieses Kapitels ist, zentrale Punkte sowie Herausforderungen zu erläutern, die bei der Realisierung der Plattform entstanden sind. Hierfür wird auf die Realisierung der einzelnen Komponenten des Systems eingegangen. Im Abschnitt Data Processing wird auf die Implementation der Metriken und der dabei verwendeten Algorithmen eingegangen.

5.1. Datenbereinigung

Die Realisierung des Datenbereinigung-Konnektors verlief im Rahmen dieser Arbeit ohne große Schwierigkeiten. Wie im Entwurf vorgesehen ist die Komponente in der Lage, vom System eine ID anzufragen, um eine Trennung der einzelnen Datenströme zu ermöglichen. Im Rahmen der Implementation stellte sich jedoch heraus, dass eine explizite Codierung der Graph-ID in den Datenstrom nach Anforderung **F3** redundant ist, da Kafka bereits eine logische Trennung der Datenströme nach Topics ermöglicht und somit eine Codierung der ID in das Kafka-Topic möglich ist.

Zusätzlich zur Anfrage einer ID ist es möglich, manuell eine ID zu vergeben, etwa um eine Datenquelle an einen bereits existierenden Datenstrom anzuschließen. Dieses Szenario ist dann erforderlich, wenn der zu analysierende Graph mehrere Datenquellen besitzt. Außerdem wird dafür gesorgt, dass beim Neustart einer Datenquelle keine neue ID vergeben wird. Das Senden eines Datenstroms reduziert sich durch den Konnektor auf folgenden Code:

```
1 // Cleansing.java
2 import latreider.thesis.connector.GraphStreamProducer;
3 import latreider.thesis.connector.TemporalEdge;
4
5 private static GraphStreamProducer producer;
6
7 public static void main(String[] args){
```

```
8    producer = new GraphStreamProducer("broker-list");
9
10   Stream<String> stream = // erzeuge bereinigten Stream
11   stream.forEach(send);
12   stream.close();
13  }
14
15  private static void send(String edge) {
16    // src, trg und timestamp extrahieren
17    TemporalEdge e = new TemporalEdge(src, trg, timestamp);
18
19    // optional ID setzen
20    e.setGraphId(graphId);
21
22    producer.send(e);
23  }
```

Listing 5.1: Beispiel Main-Klasse der Datenbereinigung

Durch die Realisierung als Maven-Projekt lässt sich der Konnektor außerdem mit geringem Aufwand als *Dependency* in eigene Projekte einbinden.

5.2. Data Ingestion

Die Realisierung des Data-Ingestion Layers erfolgte wie bereits im Kapitel 4 spezifiziert als Kafka-Cluster. Da ein Kafka-Cluster aus beliebig vielen Brokern sowie einem Zookeeper-Service besteht, erfolgte der Aufbau des Kafka-Clusters in Kubernetes durch die Spezifikation zweier *StatefulSets*. Dies geschieht aus dem Grund, da es sich sowohl bei Kafka-Brokern als auch bei Zookeeper um zustandsbehaftete Anwendungen handelt. StatefulSets sorgen dafür, dass Kubernetes jedem darin enthaltenen Pod eine Identität verleiht und ihn somit unter einem eindeutigen Namen für andere Pods ansprechbar macht.

Zookeeper agiert im Kafka-Cluster als Nameservice und Konfigurationsmanager und ist für die Verwaltung der Kafka-Broker sowie der Topic-Metadaten verantwortlich. Aus diesem Grund ist es essenziell, die in Zookeeper anfallenden Daten persistent zu speichern. Kubernetes bietet hierfür wie in Abschnitt 3.4.2 beschrieben die Möglichkeit, durch Volumes Verzeichnisse in Pods einzubinden. Da Zookeeper seine Daten im Dateisystem speichert, war lediglich die Einbindung eines Volumes in das Data-Verzeichnis des Zookeeper-Pods erforderlich.

Kafka-Broker speichern Topics sowie dessen Inhalte ebenfalls im Dateisystem, wodurch es auch für Kafka-Pods ausreicht, Volumes in das Data-Verzeichnis des Brokers einzubinden. Bei einem Neustart des Pods gehen somit keine Daten verloren. Da in Kafka Topic-Inhalte jedoch standardmäßig nur für eine Woche (168 Stunden) gespeichert werden, war ebenfalls eine Anpassung der *Retention* in den *Broker-Properties* erforderlich. Kafka bietet mehrere Strategien zur Topic-Persistierung an, darunter die zeitbasierte und die größenbasierte Speicherung. Während die zeitbasierte Speicherung Topic-Inhalte nach Ablauf einer festgelegten Zeit löscht, sorgt die größenbasierte Speicherung dafür, dass eine Topic-Partition eine bestimmte Höchstgrenze nicht überschreitet und bei Überschreiten der Größe mit dem Überschreiben älterer Inhalte begonnen wird. Innerhalb dieser Arbeit wurde die Standardeinstellung von 168 Stunden durch die Option "*–override log.retention.hours = -1*" überschrieben, sodass Topics permanent persistiert werden.

5.3. Data Processing

In diesem Abschnitt wird die Realisierung der Graphverarbeitung durch Apache Flink behandelt. Hierfür wird auf die Implementation der einzelnen Metriken sowie die hierfür verwendeten Algorithmen eingegangen. Außerdem wird auf Flink-spezifische Besonderheiten eingegangen, die für die Realisierung relevant waren.

5.3.1. Edge und Vertex Count

Die erste Metrik bildet das Zählen von Knoten und Kanten eines Graphdatenstroms und ist bereits in einfacher Form in der Gelly-Streaming Bibliothek implementiert. Die Implementation eignete sich jedoch nicht für die Verarbeitung eines temporalen Graphdatenstroms, sondern gab lediglich einen Datenstrom von Zählern zurück, der für jeden eintreffenden Datensatz die Anzahl der Knoten bzw. Kanten zählt und keine Rückschlüsse auf den Zeitpunkt zulässt, an dem der Wert des Zählers gültig ist. Aus diesem Grund fand eine Erweiterung der Implementation zum Zählen von Knoten und Kanten statt, sodass zusätzlich jedem Zähler der zugehörige Zeitstempel zugeordnet wird, was wiederum Rückschlüsse auf die zeitliche Entwicklung beider Werte ermöglicht. Das Zählen erfolgte hierbei über eine Ganzzahlvariable für die Kantenanzahl bzw. über eine Knotenmenge für die Knotenanzahl.

Tabelle 5.1 demonstriert das Zählen von Knoten und Kanten anhand eines Beispiels mit vier eintreffenden Kanten. Während das erste Element des Tupels die jeweiligen Zähler repräsentiert, gibt das zweite Element den zugehörigen Zeitstempel an.

(srcId, trgId, ts)	(|E| , ts)	(|V| , ts)
(0, 1, 1000)	(1, 1000)	(2, 1000)
(1, 2, 2000)	(2, 2000)	(3, 2000)
(1, 3, 2000)	(3, 2000)	(4, 2000)
(2, 1, 3000)	(4, 3000)	(4, 3000)

Tabelle 5.1.: Beispiel für das Zählen von Knoten und Kanten

Im nächsten Schritt galt es, die ausgegebenen Zähler in Zeitfenster zu aggregieren, um die Ergebnismenge zu reduzieren und eine Visualisierung zu ermöglichen. Die Aggregation eines Zeitfensters ergibt sich aus dem Zähler mit dem aktuellsten Zeitstempel. Ein Beispiel kann Tabelle 5.2 entnommen werden.

```
1 public long extractTimestamp(Edge<Long, Long> element, long
     previousElementTimestamp) {
2   long timestamp = element.f1;
3   return timestamp;
4 }
```

Listing 5.2: Extrahieren des Zeitstempels aus den Quelldaten

Flink bietet für die Verarbeitung von Zeitfenstern diverse *TimeCharacteristic*-Modi an. Für diese Arbeit wurde die Charakteristik *EventTime* gewählt, die dafür sorgt, dass die Zeitinformation aus den eintreffenden Datensätzen berücksichtigt wird (vgl. Listing 5.2) statt des Zeitpunkt des Eintreffens im Flink-Cluster (*IngestionTime*) bzw. des Verarbeitungszeitpunkts (*ProcessingTime*). Dieser Modus ermöglicht eine Verarbeitung von Datensätzen mit älteren Zeitstempeln, was für die Analyse von Testdatensätzen von Vorteil ist und eine Wiederholung der Datenverarbeitung zulässt.

(srcId, trgId, ts)	(|E| , ts)	(|V| , ts)
(0, 1, 800)	(1, 800)	(2, 800)
(1, 2, 900)	**(2, 900)**	**(3, 900)**
(1, 3, 1500)	(3, 1500)	(4, 1500)
(2, 1, 1600)	**(4, 1600)**	**(4, 1600)**
(2, 3, 2900)	**(5, 2900)**	**(4, 2900)**

Tabelle 5.2.: Beispiel für Window Count von Knoten und Kanten. Die Aggregate eines Zeitfensters wurden hervorgehoben.

5.3.2. Edge und Vertex Growth

Verwandt und unmittelbar abhängig vom Zählen von Knoten und Kanten ist die Ermittlung des Knoten- und Kantenwachstums innerhalb eines definierten Zeitfensters. Das Wachstum der Knoten und Kanten ergibt sich aus der Differenz zwischen der aktuellen Anzahl und der Anzahl aus dem letzten Zeitfenster. Edge und Vertex Growth angewandt auf das Beispiel in Tabelle 5.2 kann Tabelle 5.3 entnommen werden.

(srcId, trgId, ts)	**(|E| , ts)**	**(|V| , ts)**
(0, 1, 800)	(1, 800)	(2, 800)
(1, 2, 900)	**(2, 900)**	**(3, 900)**
(1, 3, 1500)	(1, 1500)	(1, 1500)
(2, 1, 1600)	**(2, 1600)**	**(1, 1600)**
(2, 3, 2900)	**(1, 2900)**	**(0, 2900)**

Tabelle 5.3.: Beispiel für die Berechnung Des Knoten und Kantenwachstums.

Umgesetzt wurde die Berechnung des Knoten und Kantenwachstums durch das Zählen der Knoten und Kanten mit dem Zusatz, dass nach jeder Window-Aggregation die aktuellen Zähler gespeichert werden. Gleichzeitig wird die Differenz zum vorherigen Zähler aus dem letzten Zeitfenster berechnet, welche das Ergebnis bildet.

5.3.3. Density

Die Dichte eines Graphen ergibt sich aus dem Quotienten der Kantenanzahl $|E|$ und der maximal möglichen Anzahl an Kanten innerhalb eines einfachen Graphen ($|V| \cdot (|V| - 1)$). Bei ungerichteten Graphen wird letztere Anzahl nochmals halbiert, da sowohl zwei wechselseitig gerichtete Kanten als auch eine ungerichtete Kante eine wechselseitige Beziehung zwischen zwei Knoten abbilden. Die Berechnung der Dichte kann aus diesem Grund aufbauend auf das Zählen von Knoten und Kanten erfolgen.

Eine Einschränkung bei der Berechnung der Dichte ergibt sich daraus, dass in einem einfachen Graphen keine Mehrfachkanten erlaubt sind und somit vor der Zählung eine Filterung dieser Kanten erfolgen muss. Ebenfalls werden Schleifen nicht berücksichtigt, die als Kanten definiert sind, die einen Knoten mit sich selbst verbinden. Die Realisierung des Filters ist Listing 5.3 zu entnehmen. Der Filter kommt ebenfalls bei der Berechnung des Average Clustering Coefficient zum Einsatz. Für die Filterung verwaltet jeder Knoten v seine Nachbarschaftsmenge $N(v)$. Im Code wurde dies als Map realisiert, die eine Knoten-Id vom Typ `Long` auf eine Menge

von Knoten-Ids vom Typ `Set<Long>` abbildet. Diese Neighborhood-Struktur ermöglicht durch die `HashMap` eine effiziente Suche nach einem Knoten v in konstanter Zeit.

```
1  Map<Long, Set<Long>> neighborhoods = new HashMap<>();
2  Edge<Long, Long> e = // Kante mit source und target
3  // Schleife filtern
4  if (source == target){
5   return;
6  }
7
8  // wenn trgId in N(srcId) oder srcId in N(trgId) existiert,
      dann filtern
9  if (source in neighborhoods && N(source).contains(target)){
10  return;
11 }
12 if (target in neighborhoods && N(target).contains(source)){
13   return;
14 }
15
16 // fuege Kante der Nachbarschaftsmenge hinzu
17 N(source) += e.getTarget();
18 N(target) += e.getSource();
```

Listing 5.3: Filtern von Schleifen und Mehrfachkanten als vereinfachte Darstellung

Durch die `Set`-Struktur wird wiederum garantiert, dass $N(v)$ keine Duplikate enthält. Die Berechnung der Dichte erfolgt nach der oben angegebenen Formel. Ein Beispiel für die Dichteberechnung ist unter Tabelle 5.4 aufgeführt.

(srcId, trgId, ts)	**|E|**	**|V|**	**(density, ts)**
(0, 1, 800)	1	2	(1.0, 800)
(1, 2, 900)	**2**	**3**	**(0.666, 900)**
(1, 3, 1500)	3	4	(0.5, 1500)
(2, 1, 1600)	**3**	**4**	**(0.5, 1600)**
(2, 3, 2900)	**4**	**4**	**(0.666, 2900)**

Tabelle 5.4.: Beispiel für die Berechnung der Graphdichte.

5.3.4. Modularity

Zur Berechnung der Modularität kam wie in Abschnitt 3.3 beschrieben der Algorithmus aus Shang u. a. (2014) zum Einsatz. Der Algorithmus sieht je nach Kantentyp unterschiedliche Aktionen vor, deren Implementation in Algorithmus 1 abgebildet ist. Zur Verwaltung von Communities wurde eine Struktur eingeführt, die eine Community auf ihre Mitglieder sowie die Anzahl der Kanten zu anderen Communities abbildet. Letzteres spielt vor allem beim Mergen zweier Communities eine wichtige Rolle.

```
Data: EdgeStream S = < e_1, e_2... >
Result: OutputStream O = < (modularity, ts)... >
Communities := {(U, #inner, CrossEdges) | U ⊆ V und CrossEdges :=
  Community → ℕ }
for e = (src, trg, timestamp) in S = < e_1, e_2... > do
    if Typeof(e) is ICE then
        C = Communities.findCommunityOf(src, trg);
        C.inner++;
    end
    if Typeof(e) is CCE then
        C1 = Communities.findCommunityOf(src);
        C2 = Communities.findCommunityOf(trg);
        if modularity(C1.merge(C2) > modularity(addCrossEdge(C1, C2))) then
            C = C1.merge(C2);
            C.inner++;
        else
            addCrossEdge(C1, C2);
        end
    end
    if Typeof(e) is HNE then
        C = Communities.findCommunityOf(existingVertex);
        C.add(newVertex);
    end
    if Typeof(e) is NE then
        C = new Community();
        C.add(src, trg);
        Communities.add(C);
    end
    mod = calculateModularity();
    O.add(mod, timestamp);
end
```

Algorithm 1: Berechnung der Modularität basierend auf Shang u. a. (2014)

Um die Effizienz bei der Berechnung der Modularität zu erhöhen, wird intern mit Akkumulatoren gearbeitet, die teure Iterationen bei jeder neuen Kante vermeiden sollen. Um die Modularität effizienter zu berechnen, erfolgte eine Umformung der Gleichung (3.2) in folgende Gleichung:

$$Q = \frac{1}{2m} \left(sumInner - sumTotal \right) \tag{5.1}$$

Während $sumInner = \sum_{in}^{c}$ die Summe aller inneren Kanten im Graph entspricht und ebenfalls wie m in einer Zählervariablen gespeichert werden kann, gibt $sumTotal$ die Summe aller $\frac{1}{2m} \cdot \sum_{tot}^{c}{}^{2}$ Terme an. Unter der Voraussetzung, dass die Werte von $\sum_{tot}^{c}$ für mehrere Communities identisch sind, muss statt über alle Communities nur noch über alle verschiedenen Werte iteriert werden.

Das Mergen zweier Communities erfolgt durch das Verschmelzen beider Mitgliedsmengen. Die neuen Summen innerer Kanten bzw. insgesamt vorhandener Kanten berechnet sich dann folgendermaßen:

$$\sum_{in}^{c_{merged}} = \sum_{in}^{c_1} + \sum_{in}^{c_2} + \sum_{cross}^{c_1,c_2} \tag{5.2}$$

$$\sum_{tot}^{c_{merged}} = \sum_{tot}^{c_1} + \sum_{tot}^{c_2} - \sum_{cross}^{c_1,c_2} \tag{5.3}$$

$\sum_{cross}^{c_1,c_2}$ gibt hierbei die Summe aller Kanten an, die ehemals zwischen den beiden Communities c_1 und c_2 verlaufen sind.

5.3.5. Average Clustering Coefficient

In Gelly-Streaming lag wie in Abschnitt 3.3 bereits beschrieben eine Implementation des Triangle Count Algorithmus vor, die für diese Arbeit sowohl um eine temporale Komponente als auch um die Berechnung des Clustering Coefficients erweitert werden musste. Folgende Schritte wurden zur Berechnung des Average Clustering Coefficient umgesetzt:

- Filtern von Mehrfachkanten und Schleifen
- Berechnung des lokalen Triangle Count und des Knotengrads $deg(v)$
- Berechnung des Local Clustering Coefficient aus 2.
- Berechnung des Average Clustering Coefficient aus 3.

Im Folgenden wird detailliert auf die Implementation dieser Schritte eingegangen.

1. Das Filtern von Mehrfachkanten und Schleifen kam bereits bei der Berechnung der Graphdichte zum Einsatz und findet in diesem Algorithmus ebenfalls Verwendung. Um Mehrfachkanten zu erkennen, wird jeder Knoten v durch eine Map-Struktur auf seine Nachbarschaftsmenge $N(v)$ abgebildet und entsprechend gefiltert (s. Listing 5.3).

2. Sowohl die Berechnung der Dreiecksanzahl als auch die Berechnung des Knotengrades können auf $N(v)$ zurückgeführt werden. Algorithmus 2 beschreibt die Berechnung des Triangle Count und des Knotengrads innerhalb eines Kantenstroms. Als Neighborhood-Set kam die Struktur aus Schritt 1 zum Einsatz.

Data: EdgeStream S = $< e_1, e_2 ... >$
Result: OutputStream O = $< (v_1, \#triangles_{v1}, deg(v_1), ts) ... >$
NeighborhoodSet := $v \rightarrow N(v)$ mit $v \in V$;
for *Edge e in S* = $< e_1, e_2 ... >$ **do**
  src = source(e);
  trg = target(e);
  ts = timestamp(e);
  numTriangles = 0;
  for *Vertex v in* $|N(src) \cap N(trg)|$ **do**
    numTriangles++;
    O += $(v, 1, |N(v)|, ts)$; // v erhält ein zusätzliches Dreieck
  end
  if *numTriangles > 0* **then**
    O += $(src, numTriangles, 0, ts)$;
    O += $(trg, numTriangles, 0, ts)$;
  end
  O += $(src, 0, |N(src)|, ts)$;
  O += $(trg, 0, |N(trg)|, ts)$;
  CalculateCC();
end

Algorithm 2: Berechnung des Triangle Count und Knotengrads basierend auf Kalavri u. a. (2016)

Um den Triangle Count zu berechnen, werden die beiden Nachbarschaftsmengen $N(src)$ und $N(trg)$ geschnitten. Für jeden Knoten v innerhalb der Schnittmenge erhöht sich der Triangle Count um 1, da aus Sicht von Knoten v eine neue Kante zwischen seinen Nachbarn verläuft. Dies fließt im nächsten Schritt in die Berechnung des Local Clustering Coefficient ein. Für beide inzidenten Knoten der Kante erhöht sich der Triangle Count

entsprechend um die Anzahl der Knoten in der Schnittmenge. Der Knotengrad beträgt für alle betroffenen Knoten $|N(v)|$.

3. Der Local Clustering Coefficient errechnet sich aus den im vorherigen Schritt übermittelten Triangle Count und Knotengrad. Während der Triangle Count eines Knotens ein akkumulierter Wert ist und somit zum bereits existierenden Triangle Count addiert wird, wird der Knotengrad des Knotens überschrieben. Der Local Clustering Coefficient errechnet sich aus Gleichung (5.4).

$$LC(v) = \begin{cases} \frac{numTriangles}{deg(v)(deg(v)-1)} & \text{, für } deg(v) \geq 2 \\ 0 & \text{, sonst} \end{cases} \tag{5.4}$$

   Zu bedenken ist, dass ein Dreieck erst ab einem Knotengrad von 2 möglich ist.

4. Im letzten Schritt findet die Berechnung des Average Clustering Coefficient basierend auf Schritt 3 statt. Dieser ergibt sich aus dem Quotienten der Summe aller Local Clustering Coefficients und der Knotenanzahl (s. Gleichung (5.5)).

$$AC = \frac{1}{|V|} \sum_{v \in V} LC(v) \tag{5.5}$$

   Um innerhalb eines Datenstroms nicht für jeden eintreffenden Datensatz über die gesamte Knotenliste iterieren zu müssen, ist eine Optimierung möglich, indem lediglich mit dem Wert der Summe aller Local Clustering Coefficients gearbeitet wird. Die Berechnung des Average Clustering Coefficient lässt sich dadurch folgendermaßen formulieren:

$$\sum LC_{new} = \begin{cases} \sum LC + \Delta LC(v) & \text{, wenn } LC(v) \text{ nicht neu} \\ \sum LC + LC(v) & \text{, sonst} \end{cases} \tag{5.6}$$

   wobei $\Delta LC(v)$ die Differenz der Local Clustering Coefficients vor und nach der Verarbeitung in Schritt 3 darstellt.

5.3.6. Datenstromtransformation

Die Implementation der Metrikberechnungen in Flink bzw. Gelly erfolgte durch das Transformieren von Datenströmen. Listing 5.4 beschreibt die Grobstruktur der Datenstromtransformationen und bildet das Grundgerüst aller implementierten Algorithmen.

```
1 DataStream<String> kafkaStream = getKafkaStream();
2 DataStream<Edge<Long, Long>> edgeStream = kafkaStream.map(new
     TemporalEdgeMap()).assignTimestampsAndWatermarks(new
     TimeStampExtractor());
3 DataStream<TValue, Long> metricStream = edgeStream.map(new
     MetricMapper());
4 DataStream<TValue, Long> windowedStream =
     metricStream.timeWindowAll(windowSize).max("timestamp");
5 windowedStream.addSink(new ElasticSearchSink());
```

Listing 5.4: Transformationskette des temporalen Graphdatenstroms

Im ersten Schritt erfolgt die Erzeugung eines DataStreams aus einem Kafka-Topic. Dies geschieht unter Zuhilfenahme des Kafka-Konnektors, der bereits in Flink enthalten ist. Intern handelt es sich dabei um einen Kafka-Consumer. Da eine Kante in Kafka als Zeichenkette codiert ist, muss jeder eintreffende Datensatz im nächsten Schritt durch eine Map-Funktion geparst und in ein Edge-Objekt umgewandelt werden. Die Klasse `Edge` stammt aus der Gelly-Bibliothek und enthält Felder wie die Identifier der beiden inzidenten Knoten (*sourceId* bzw. *targetId*) sowie ein Property-Feld, das den Wert einer Kante enthält. Das Property-Feld kam in dieser Arbeit für die Speicherung des zugehörigen Timestamp zum Einsatz.

Um eine Window-Verarbeitung in Flink nach der Charakteristik *EventTime* möglich zu machen, muss Flink im nächsten Schritt durch *assignTimestampsAndWatermarks* mitgeteilt werden, welches Feld des Datensatzes den Timestamp enthält. Durch *Watermarks* wird angegeben, dass Events bis zu einem bestimmten Zeitpunkt t verarbeitet wurden und folglich keine Datenstätze mit einem Zeitpunkt $t' < t$ eintreffen sollten. Dies ist deshalb notwendig, damit Flink weiß, ab wann ein Time-Window abgeschlossen ist und mit dessen Verarbeitung durch den Algorithmus fortgefahren werden kann. Da die Verarbeitung in Zeitfenstern erfolgt, muss im nächsten Schritt das Ergebnis des Zeitfensters berechnet werden. Im Rahmen dieser Arbeit besteht dies darin, den aktuellsten Datensatz zu ermitteln.

Das berechnete Ergebnis wird anschließend in ElasticSearch geschrieben. Flink enthält bereits einen ElasticSearch-Konnektor, der als Datensenke angebunden wird und die Übertragung von Datensätzen in eine ElasticSearch-Datenbank möglich macht. Die ElasticSearch-API nimmt hierfür JSON-Dokumente über eine REST-API entgegen. Damit nicht für jeden einzelnen Datensatz ein REST-Call erfolgen muss und somit ein Flaschenhals bei der Verarbeitung entsteht, wurde für den Konnektor ein *Bulk-Flush* Intervall von einer Sekunde konfiguriert.

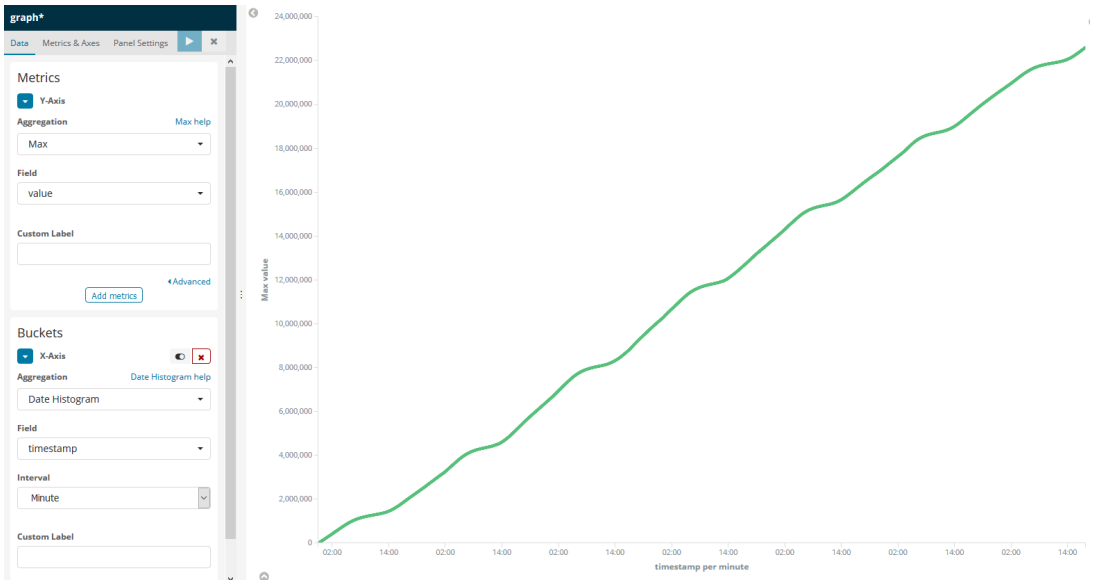


Abbildung 5.1.: Beispiel eines Edge Count Diagramms in Kibana

5.4. Serving

Um in Kibana die Evolution der Metriken verschiedener Graphen zu visualisieren, besteht ein an ElasticSearch übertragener Datensatz neben Metadaten aus den vier Feldern *graphId*, *metricName*, *value* und *timestamp*. Die *graphId* bildet ab, welchem Graph die anderen drei Werte zugehörig sind und ermöglicht letztendlich eine Filterung der Datensätze zur Darstellung eines einzelnen Graphen. Der Metrikname ermöglicht die Zuordnung des Wert-Zeitstempel Paars zur eigentlichen Metrik. Der Value bildet den Wert einer Metrik ab, der Timestamp bildet den Value wiederum auf einen Zeitpunkt ab. Ordnet man diese vier Felder den Komponenten in Kibana zu, so repräsentiert eine Graph-ID jeweils ein Dashboard, eine Metrik wird jeweils durch ein Diagramm abgebildet und das Wert-Zeitstempel Paar entspricht dem Wert eines Datenpunktes auf der Y- respektive X-Achse.

Die Visualisierung der einzelnen Metriken wurde in Kibana mithilfe von Date-Histogrammen realisiert (vgl. Abb. 5.1). Für die X-Achse wird hierbei das Feld eingestellt, welches den Timestamp enthält sowie ein Intervall, welches letztendlich die Schrittweite der X-Achse festlegt. Für die Y-Achse wird angegeben, welches Feld die Daten enthält und welche Art von Aggregation auf den Zeitraum angewandt wird, der zwischen zwei Punkten des Intervalls liegt, welches in der X-Achse eingestellt wurde.

Ein Dashboard dient in Kibana zur Zusammenfassung verschiedener Visualisierungen bzw. Diagramme und lässt sich mithilfe einer grafischen Oberfläche im Dashboard-Menü zusam-

menklicken. Im Dashboard lassen sich außerdem Filter auf die in den Visualisierungen referenzierten Daten anwenden. So lassen sich Metriken letztendlich nach ihrer Graph-ID filtern. Da analysierte Graphen außerdem verschiedene Zeiträume aufweisen können und mehr Datenpunkte die Performance der Visualisierung der Diagramme beeinflussen, erfolgte eine Kategorisierung der Dashboards in verschiedene Anzeigeintervalle. Je nach Zeitraum des analysierten Graphen lässt sich somit das passende Dashboard auswählen. Die Schrittweite der Metriken eines Dashboard reicht somit von Sekunden, Minuten und Stunden bis hin zu Tagen und Wochen.

6. Evaluation

Dieses Kapitel thematisiert die Evaluation des in den vorherigen Kapiteln spezifizierten und realisierten Systems. Ziel des Kapitels ist die Feststellung der Eignung des Systems zur Analyse dynamischer Graphen. Hierfür werden zunächst Testdatensätze dynamischer Graphen aus Leskovec und Krevl (2014) ausgewertet und interpretiert, um im nächsten Schritt den Einsatz der Plattform zur Echtzeitanalyse eines Twitter-Datenstroms zu evaluieren.

6.1. Analyse von Testdatensätzen

Das SNAP-Projekt Leskovec und Krevl (2014) stellt neben Datensätzen für Webgraphen, Review-Netzwerken und Netzwerken autonomer Systeme auch Datensätze dynamischer Graphen zur Verfügung. Konkret handelt es sich um Datensätze der Stack Exchange Foren Stackoverflow, Mathoverflow, Superuser und Askubuntu. Ebenfalls sind E-Mail Kommunikationsnetzwerke einer großen europäischen Forschungseinrichtung und Web of Trust Netzwerke zweier Bitcoin-Marktplätze vertreten. Im Folgenden werden die Ergebnisse der Analyse durch das System aufgeführt und interpretiert.

6.1.1. Bitcoin

Bei Bitcoin handelt es sich um eine sogenannte Kryptowährung, die seit 2009 als digitale Währung auf diversen Online-Marktplätzen gehandelt wird. Da Benutzer von Bitcoin auf diesen Marktplätzen anonym sind, entstand der Bedarf nach einem Netzwerk, das die einzelnen Benutzer mit einer Reputation und somit auch einem Vertrauensstatus versieht. Dadruch sollen riskante Transaktionen mit riskanten bzw. betrügerischen Benutzern verhindert werden. Die beiden analysierten Datensätze *bitcoin-otc* und *bitcoin-alpha* (Kumar u. a. (2016)) bilden jeweils ein Bewertungsnetzwerk des jeweiligen Marktplatzes ab. Die Bewertungen unterliegen hierbei einer zeitlichen Einordnung von 2011 bis 2015.

Vergleicht man den Verlauf der Metriken beider Netzwerke, so ähneln sich diese. Der Verlauf des Edge- und Vertex-Growth im Jahr 2011 kann Abb. 6.1 und Abb. 6.2 entnommen werden. Hier ist im Zeitraum Mitte 2011 sowohl bei der Knoten- als auch bei der Kantenanzahl ein

sprunghafter Anstieg zu beobachten. Legt man die Historie von Bitcoin zugrunde, so kann dieser Peak auf einen ersten starken Kursanstieg im Juni sowie einem anschließenden Fall zurückgeführt werden (vgl. Abb. 6.3).

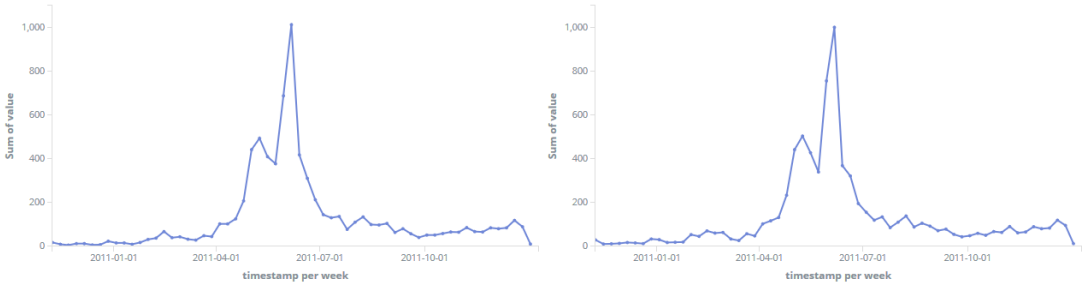


Abbildung 6.1.: Edge Growth im Jahr 2011: Bitcoin Alpha (links) und Bitcoin OTC (rechts)

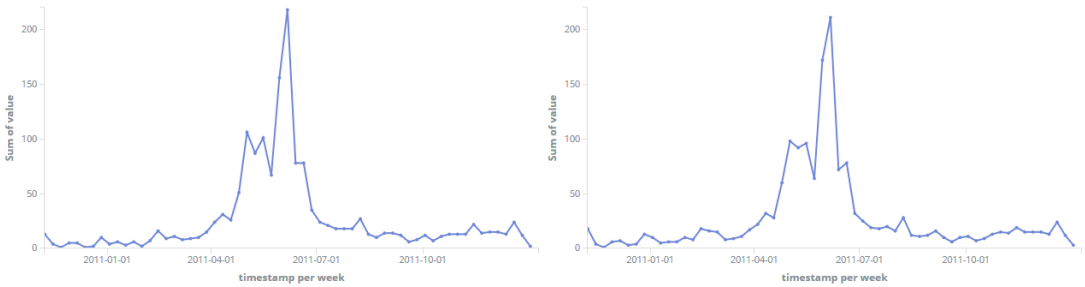


Abbildung 6.2.: Vertex Growth im Jahr 2011: Bitcoin Alpha (links) und Bitcoin OTC (rechts)

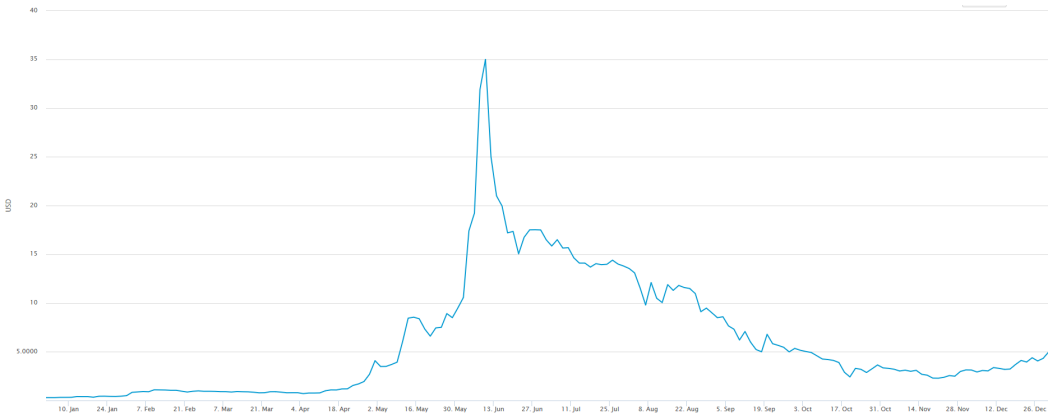


Abbildung 6.3.: Bitcoin-Kurs im Jahr 2011 in USD (Quelle: Luxembourg (2018))

In Abb. 6.4 und Abb. 6.5 kann vor allem im *bitcoin-otc* Datensatz im Jahr 2013 ein weiterer Wachstumsanstieg des Netzwerks beobachtet werden. Legt man den Kurs in diesem Jahr zugrunde, so ist auch hier Ende 2013 ein rasanter Kursanstieg (Abb. 6.6) abzulesen. Dies lässt die Aussage zu, dass eine Korrelation zwischen dem Kursverlauf des Bitcoin und der Aktivität der Bewertungsnetzwerke besteht.

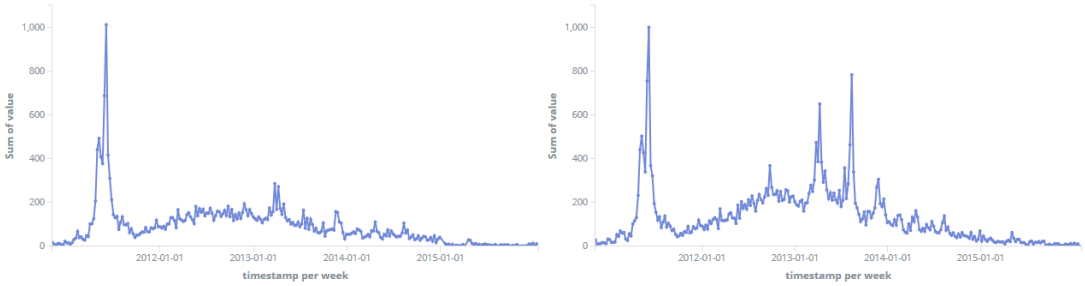


Abbildung 6.4.: Edge Growth 2011 bis 2015: Bitcoin Alpha (links) und Bitcoin OTC (rechts)

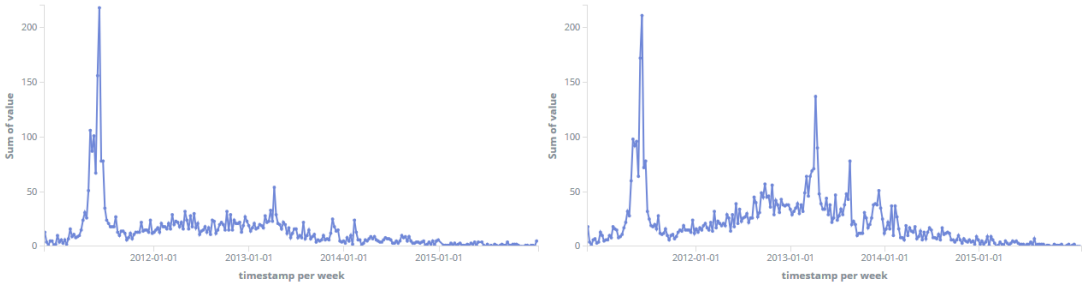


Abbildung 6.5.: Vertex Growth 2011 bis 2015: Bitcoin Alpha (links) und Bitcoin OTC (rechts)

Abbildung 6.6.: Bitcoin-Kurs 2011 bis 2015 in USD (Quelle: Luxembourg (2018))

Bezüglich der Dichte beider Netzwerke (Abb. 6.7) ist ein kontinuierlicher Abfall der Kurvenverläufe abzulesen. Da in einem Netzwerk dieser Größenordnung die Wahrscheinlichkeit sehr gering ist, dass alle Nutzer Beziehungen zu allen anderen Nutzern aufweisen, ist folglich die Wahrscheinlichkeit einer geringen Dichte hoch. In diesem Fall konnten geringe Dichten in beiden Netzwerken gemessen werden.

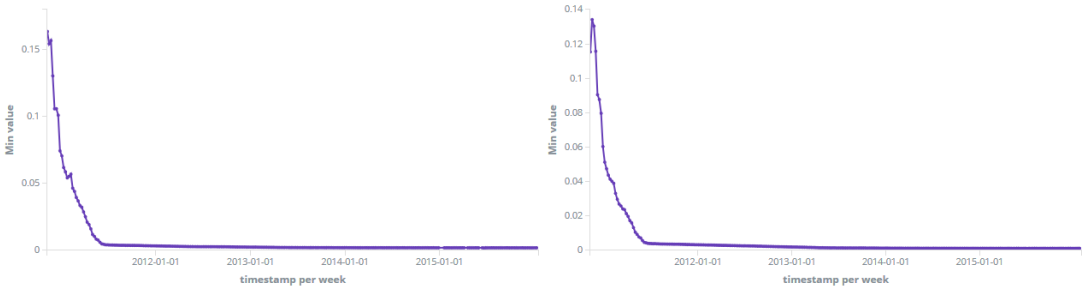


Abbildung 6.7.: Density: Bitcoin Alpha (links) und Bitcoin OTC (rechts)

Der Average Clustering Coefficient ist in beiden Netzwerken zunächst hohen Schwankungen unterworfen (Abb. 6.8), was sich darin äußert, dass der Wert wechselnd steigt und sinkt. Legt man das Wachstum des Netzwerks zugrunde, so kann mit steigendem Wachstum ein Anstieg des Clustering Coefficient abgelesen werden, der wiederum mit sinkendem Wachstum ebenfalls abnimmt. Nach dem Peak im Jahr 2011 nehmen die Schwankungen im Kurvenverlauf ab und führen in beiden Netzwerken zu einem kontinuierlichen Anstieg des Clustering Coefficient.

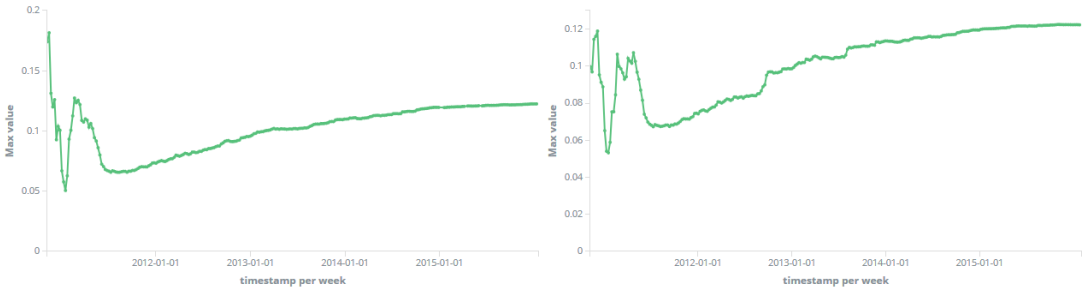


Abbildung 6.8.: Average Clustering Coefficient: Bitcoin Alpha (links) und Bitcoin OTC (rechts)

Die Modularität beider Netzwerke unterliegt ähnlich wie beim Clustering Coefficient ebenfalls Schwankungen im Wachstums-Peak 2011. Nach diesem Peak erfolgt innerhalb beider Netzwerke eine moderate Abnahme der Modularität. Gemeinsam betrachtet mit dem Average Clustering Coefficient lässt sich hier die Vermutung aufstellen, dass es sich um ein Netz-

werk mit Untergemeinschaften geringer, jedoch wachsender Dichte handelt, die berechneten Untergemeinschaften jedoch eine niedrige Modularität aufweisen.

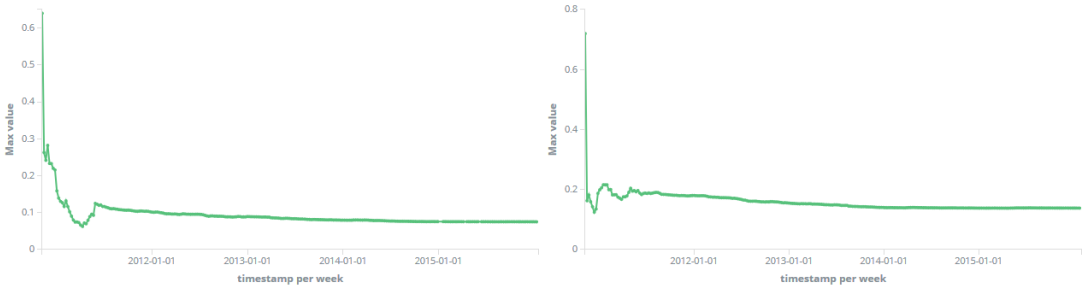


Abbildung 6.9.: Modularity: Bitcoin Alpha (links) und Bitcoin OTC (rechts)

6.1.2. EU-Core

In diesem Abschnitt werden Datensätze aus E-Mail Kommunikationsnetzwerken einer großen, europäischen Forschungseinrichtung betrachtet. Der Name dieser Einrichtung geht aus der Quelle (Paranjape u. a. (2016)) jedoch nicht hervor. Die Datensätze unterteilen sich in ein Netzwerk für die gesamte Institution sowie vier Netzwerke für jeweils ein Department dieser Einrichtung. Jedes Netzwerk bildet die E-Mail Korrespondenz zwischen Mitgliedern der Institution bzw. der einzelnen Departments ab. Jeder Kante in diesen Netzwerken ist ein Zeitstempel zugeordnet, der den Zeitpunkt der Kommunikation abbildet. Aus den Zeitstempeln gehen jedoch keine Annahmen auf die realen Zeitpunkte hervor, an denen die Kommunikation tatsächlich stattgefunden hat. Die Auswertung konnte aus diesem Grund lediglich über die Zeitabstände stattfinden. Ebenfalls sind die IDs der jeweiligen Netzwerke unabhängig voneinander und lassen sich nicht auf die anderen Netzwerke übertragen. Somit musste die Verarbeitung der Datensätze unabhängig voneinander erfolgen.

Bei der Auswertung der Diagramme stellte sich ein lineares Wachstum bezüglich der Kantenanzahl und ein beschränktes Wachstum bezüglich der Knotenanzahl heraus. Während Ersteres eine annähernd gleichbleibende Kommunikationsfrequenz in den Netzwerken aufzeigt, geht Letzteres vermutlich darauf zurück, dass es sich um eine begrenzte Menge an Personen innerhalb der Institution bzw. Departments handelt, die eher langsam wächst. Als interessant gestaltet sich die Evolution der Dichte innerhalb aller Netzwerke. In allen fünf Netzwerken konnte nämlich ein Anstieg der Dichte gemessen werden. Dies lässt sich vermutlich mit dem linearen Kantenwachstum in Kombination mit dem beschränkten Knotenwachstum erklären. Übertragen auf das Netzwerk spräche dies dafür, dass sich über die Zeit eine rege Kommunikation innerhalb des gesamten Netzwerks entwickelt hat. Der Average Clustering Coefficient weist

in allen fünf Netzwerken einen vergleichsweise hohen Wert auf, von 0,36 in *eu-core* bis hin zu 0,65 in *eu-core Dept. 2*. Dass der Koeffizient in den Netzwerken der Departments höher ist als im institutionsweiten Netzwerk, lässt sich damit erklären, dass innerhalb der Departments von einer stärkeren Kommunikation und somit zu einer stärkeren Clusterbildung ausgegangen werden kann. Die Entwicklung der Modularität kann dagegen in zwei Gruppen unterteilt werden. Während die Modularität in *eu-core* und *eu-core Dept. 3* zunächst stark abnimmt und danach auf einem vergleichsweise niedrigem Niveau bleibt, stagniert sie in den restlichen Netzwerken nach einer ersten Abnahme auf einem vergleichsweise hohem Wert. Eine Korrelation mit dem Average Clustering Coefficient konnte in diesem Fall nicht beobachtet werden.

6.1.3. Stack Exchange

In diesem Abschnitt soll die Auswertung zweier Netzwerke aus dem *Stack Exchange* Bereich erfolgen, zum einen der Datensatz *sx-stackoverflow* und zum anderen der Datensatz *sx-askubuntu*, da hier bezüglich Wachstum die größten Auffälligkeiten im Vergleich zu den beiden anderen Datensätzen *sx-mathoverflow* und *sx-superuser* beobachtet werden konnten. Da es sich um Netzwerke mit realen Zeitstempeln handelt, wurde ähnlich wie in *bitcoin-otc* und *bitcoin-alpha* auch hier versucht, Kurvenverläufe auf reale Ereignisse zurückzuführen.

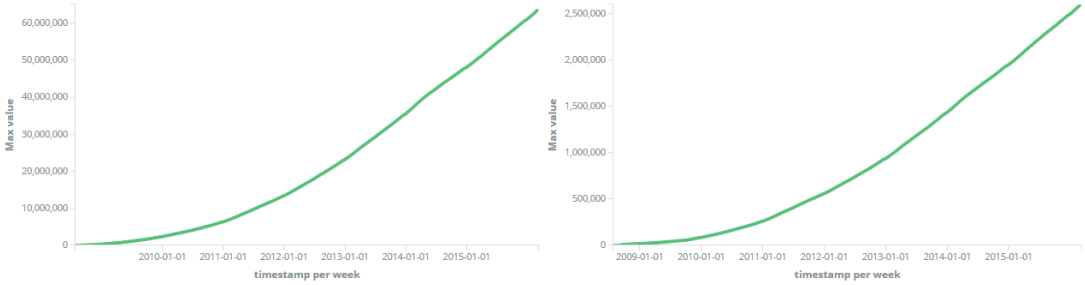


Abbildung 6.10.: sx-stackoverflow: Edge Count (links) und Vertex Count (rechts)

Sowohl das Stackoverflow- als auch Askubuntu-Netzwerk unterlagen innerhalb des Analysezeitraums einem nichtlinearen Wachstum, was sowohl aus der Anzahl der Knoten und Kanten (6.10, 6.11) als auch an deren Wachstum abgelesen werden kann (6.12, 6.13). Besonders bei der Betrachtung des Wachstumsverlaufs konnten diverse Auffälligkeiten beobachtet werden, die einer nähreren Untersuchung bedurften. Betrachtet man im *sx-stackoverflow* Datensatz den ersten Peak (6.12), so kann dieser der Woche um den 15.09.2008 zugeordnet werden.

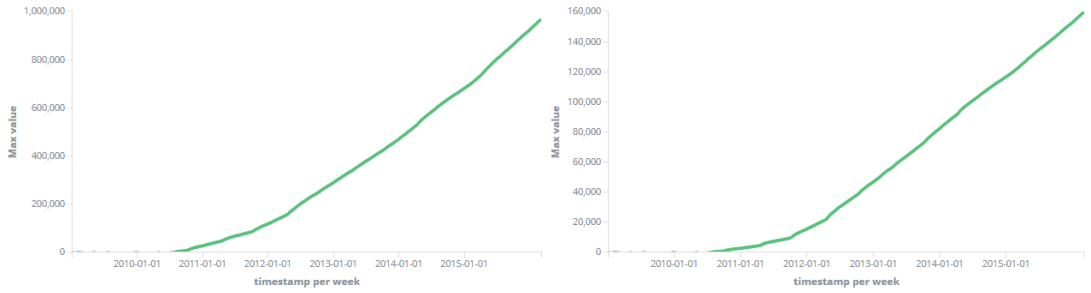


Abbildung 6.11.: sx-askubuntu: Edge Count (links) und Vertex Count (rechts)

Nach Spolsky (2008) erfolgte an diesem Datum der Launch der Plattform Stackoverflow, was nach Analyse des Datensatzes in einem sprunghaften Anstieg sowohl der Nutzerzahlen als auch der Nutzung der Plattform resultierte. Als ebenfalls auffällig erwiesen sich die regelmäßigen Peaks in den darauffolgenden Jahren. Betrachtet man die Zeitpunkte dieser Peaks, so können die Einbrüche des Wachstums auf die Weihnachtszeit der jeweiligen Jahre zurückgeführt werden. Begründen ließen sich die Einbrüche damit, dass die Aktivität von Softwareentwicklern in dieser Zeit durch die Feiertage abnimmt. Da sich dieses Ereignis jährlich wiederholt, ist es sehr wahrscheinlich, dass es sich auf die Zukunft übertragen lässt.

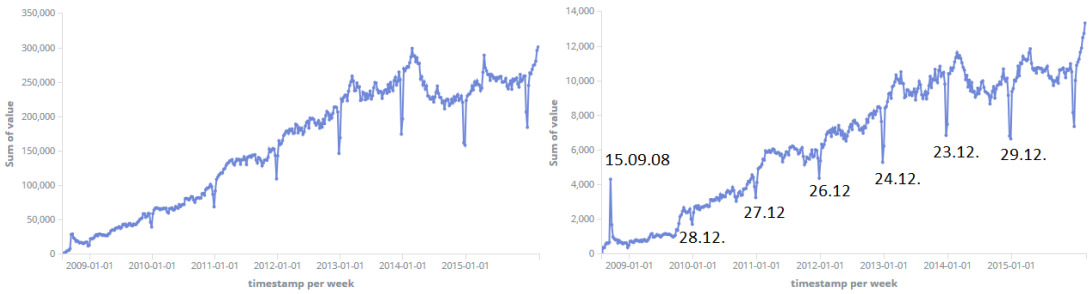


Abbildung 6.12.: sx-stackoverflow: Edge Growth (links) und Vertex Growth (rechts)

Die Peaks beim Wachstum des Netzwerks *sx-askubuntu* ließen sich analog zum *sx-stackoverflow* Datensatz auswerten. Wie aus dem Namen hervorgeht, bietet Askubuntu eine Plattform für Fragen über das linuxbasierte Betriebssystem Ubuntu. Legt man die Versionshistorie von Ubuntu zugrunde, so können die Peaks größtenteils auf Major-Releases von Ubuntu[1] zurückgeführt werden. So erschien etwa Version 10.10 am 10.10.2010, Version 11.04 am 28.04.2011, Version 11.10 am 13.10.2011, Version 12.04 LTS am 26.04.2012 und Version 14.04 LTS am 17.04.2014.

[1] https://en.wikipedia.org/wiki/Ubuntu_version_history#Table_of_versions (Datum: 13.11.2018)

Diese Releases scheinen eine erhöhte Aktivität vor allem vieler neuer Nutzer im Forum hervorzurufen, was damit begründet werden könnte, dass ein Major-Release viele Neuerungen enthält, die wiederum viele Fragen bei den Nutzern aufwerfen.

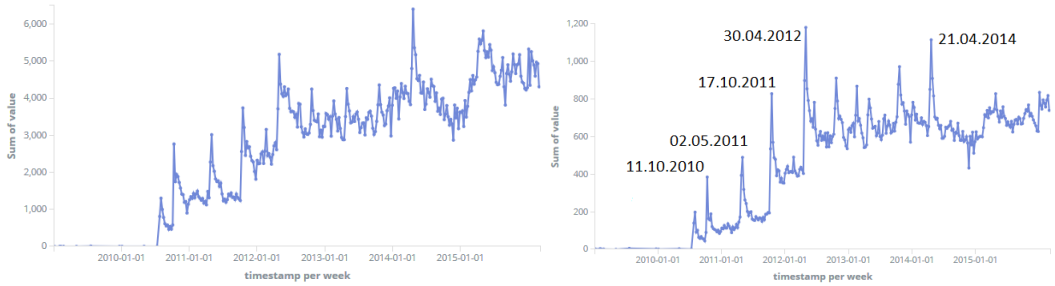


Abbildung 6.13.: sx-askubuntu: Edge Growth (links) und Vertex Growth (rechts)

6.2. Anwendung: Echtzeitanalyse eines Twitter-Datenstroms

In diesem Abschnitt wird die Auswertung eines Twitter-Datenstrom mithilfe der Plattform beleuchtet. Ziel dieser Anwendung ist die Verarbeitung eines realen Datenstroms, da für die Analyse vorheriger Testdatensätze lediglich simulierte Datenströme zugeführt wurden. Im Folgenden findet zunächst eine Kurzeinführung in Twitter statt, um daraufhin die Implementation und Durchführung des Experiments zu beleuchten. Anschließend findet eine Auswertung der Ergebnisse statt.

6.2.1. Beschreibung

Twitter[2] existiert seit 2006 und verzeichnet nach Stand vom Oktober 2018 etwa 326 Millionen monatlich aktive Nutzer (Twitter (2018)). Bei Twitter handelt es sich um eine Mikroblogging-Plattform, die es Nutzern ermöglicht, Kurznachrichten zu verfassen, sogenannte *Tweets*. Diese Tweets sind in der Regel öffentlich einsehbar und können von anderen Nutzern konsumiert werden. Bei Nutzern handelt es sich mehrheitlich um gewöhnliche Personen, es sind jedoch auch Berühmtheiten wie Schauspieler, Politiker und Sänger, aber auch abstraktere Entitäten wie Marken oder Firmen mit ihren Accounts auf dieser Plattform vertreten. Letztere haben durch Twitter die Möglichkeit, möglichst viele Nutzer über einen zusätzlichen Kommunikationskanal anzusprechen und auf diese Art zusätzliche Reichweite zu generieren. Möchte ein Nutzer Inhalte anderer Nutzer, Marken, Berühmtheiten usw. konsumieren, so kann er ihren Seiten folgen und

[2] https://twitter.com/

wird dadurch zum *Follower*. Dies führt dazu, dass auch zukünftige Beiträge abonnierter Seiten in der eigenen Timeline angezeigt und dort konsumiert werden können.

Kommunikation findet in Twitter über die Möglichkeiten statt, andere Tweets zu liken, andere Nutzer über ihren Benutzernamen in eigenen Tweets zu erwähnen (*mention*), Tweets zu kommentieren oder diese zu retweeten. Während das Kommentieren eine Reaktion auf einen Tweet darstellt, wird beim Retweeten der Inhalt des ursprünglichen Tweets eins zu eins geteilt, um dort von eigenen Followern konsumiert zu werden. Bildet man diese Kommunikationsformen ab, so kann eine Modellierung als sozialer Graph erfolgen, in welchem Nutzer durch Knoten und Interaktionen zwischen Nutzern durch Kanten dargestellt werden.

6.2.2. Implementation des Twitter-Crawlers

Twitter bietet eine Streaming-API[3] an, die es ermöglicht, Tweets in Echtzeit zu empfangen und auszuwerten. Hierfür stellt Twitter eine REST-Schnittstelle zur Verfügung, mit ein Datenstrom aus Tweets über ein HTTP-Request angefragt werden kann. Der Response auf diesen Request ist ein endlos langer Stream von JSON-Objekten, die sowohl den Tweet als auch diverse Metadaten in einem strukturierten Format enthalten. Im Rahmen der Anwendung soll die Streaming-API von Twitter dazu verwendet werden, einen Datenstrom zunächst mithilfe eines Clients anzufragen, um dann im nächsten Schritt durch eine Bereinigung der Daten eine Umwandlung in einen Kantenstrom durchzuführen. Listing 6.1 zeigt ein vereinfachtes Beispiel eines Retweets im JSON-Format, wie er in der Streaming-API codiert ist.

```
1  {
2   "created_at":"Wed Oct 17 10:43:36 +0000 2018",
3   "id": 9999999999,
4   "id_str": "9999999999",
5   "text": "RT @RetweetedUser: original tweet", ...
6   "user": {
7     "id": 123456,
8     "id_str": "123456", ...
9   }, ...
10  "retweeted_status":{
11    "created_at": "Wed Oct 17 06:12:45 +0000 2018",
12    "id" :7777777777,
13    "id_str": "7777777777",
```

[3] https://developer.twitter.com/en/docs/tweets/filter-realtime/overview (Zugriff: 14.11.2018)

```
    "text": "original tweet", ...
    "user": {
      "id": 456789,
      "id_str": "456789", ...
    }, ...
  }, ...
  "entities":{ ...
    "user_mentions": [
      { ...
        "id": 456789,
        "id_str": "456789", ...
      }
    ] ...
  }, ...
  "lang": "en",
  "timestamp_ms": "1539773016245"
}
```

Listing 6.1: Vereinfachtes Beispiel eines Retweets

Die für die Analyse relevanten Felder sind hierbei *user.id_str*, der die ID des Nutzers angibt, der einen Tweet verfasst bzw. retweetet hat, *retweeted_status.user.id_str* als ID des ursprünglichen Verfassers, *entities.user_mentions* für alle im Tweet erwähnten Nutzer und *timestamp_ms* für den Zeitpunkt, an dem der Tweet verfasst wurde. Aus diesen Feldern lässt sich eine Liste von Kanten generieren, welche die Interaktion zwischen verschiedenen Nutzern repräsentieren. Aus dem oben genannten Beispiel würde also eine Kante erzeugt werden, wie sie in Listing 6.2 zu sehen ist. Kommentare auf Tweets werden entsprechend über ein *quoted_status* anstelle eines *retweeted_status* abgebildet. Beide Objekte unterscheiden sich letztendlich in ihrem Inhalt und eventuell in der *user_mentions* Menge, da im Gegensatz zu einem Retweet der Nutzer einen eigenen Text verfassen und somit auch andere Nutzer erwähnen kann.

```
123456 456789 1539773016245
```

Listing 6.2: Generierte Kante aus Listing 6.1

Die Implementation des Twitter-Crawlers erfolgte in der Programmiersprache Java unter Zuhilfenahme des Hosebird Client[4], der als Twitter Client fungiert und technische Details der

[4] https://github.com/twitter/hbc (Datum: 13.11.2018)

Streaming-API kapselt. Als Eingabe erfordert der Client Parameter wie Authentifizierungs-Token und einen Filter, nach welchem der Datenstrom bereits serverseitig vorgefiltert wird. Daraufhin stellt der Client eine endlos lange Queue von Tweet-Objekten zur Verfügung, die im nächsten Schritt wie oben dargestellt bereinigt und über den in Kapitel 5 realisierten Konnektor in das Kafka-Cluster des Systems gesendet wurden. Da der Crawler als Container-Anwendung mit Kubernetes als Laufzeitumgebung umgesetzt wurde, erfolgte die Konfiguration der oben genannten Parameter über Umgebungsvariablen. Dies erlaubte es, einen Pod mit verschiedenen Parametern zu starten, ohne das Image anpassen zu müssen.

6.2.3. Auswertung

Die Durchführung des Experiments erfolgte im Zeitraum 14.10.2018 bis 01.11.2018. Als Filter wurde der aktuelle US-Präsident Donald J. Trump gewählt, wodurch nur Tweets berücksichtigt wurden, in denen der Begriff "Trump" vorkommt. Das Beispiel erschien insofern interessant, da der US-Präsident unter anderem auf Twitter als Kommunikationsmedium zurückgreift und eine vergleichsweise hohe Aufmerksamkeit erregt. Dies bestätigte sich auch bei der Auswertung des Datenstroms. Im Rahmen des Auswertungszeitraums wurden durch die Plattform über 63 Mio. Kanten und über 4 Mio. Knoten gezählt. Geht man von einem gleichbleibenden Wachstum aus, so entspräche dieser einem Wert von etwa 40 Kanten und 2,5 Knoten pro Sekunde. An diesem Punkt sollte jedoch erwähnt werden, dass die Streaming-API nicht die reale Datenmenge abbildet, die durch die Nutzerschaft von Twitter generiert wird. Nach Piper (2015) biete die API zu jedem Zeitpunkt Zugriff auf lediglich etwa 1 Prozent des anfallenden Tweet-Volumens. Dass im Rahmen des gewählten Themas nicht von einem gleichbleibenden Wachstum ausgegangen werden kann, zeigt die folgende Betrachtung der Diagramme.

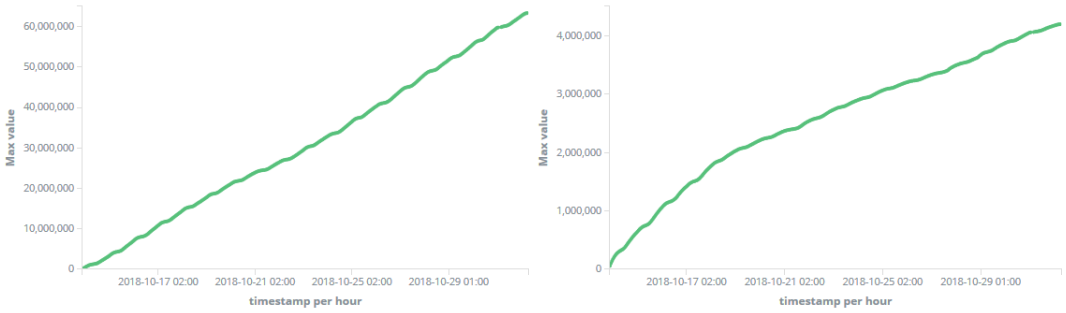


Abbildung 6.14.: Twitter Edge Count (links) und Vertex Count (rechts)

Aus Abb. 6.14 geht ein wellenförmiges Wachstum sowohl der Knoten als auch der Kanten hervor. Während die Steigung der Kantenanzahl grob betrachtet gleich blieb, ist bei der Knoten-

anzahl ein leichter Knick zu erkennen, der darauf hindeutet, dass im Datenstrom zu Anfang des Auswertungszeitraums viele neue Nutzer enthalten waren, die Anzahl neuer Nutzer mit der Zeit abgenommen hat und somit auch das dessen Wachstum. Dieses wellenförmige Wachstum wird vor allem in Abb. 6.15 und Abb. 6.16 deutlich. Hier ist zu erkennen, dass das wellenförmige Wachstum durch die Uhrzeiten zustande kommt, zu der es aufgezeichnet wurde. Die aufgeführten Uhrzeiten entsprechen hierbei der mitteleuropäischen Zeit (MEZ). Das Minimum wird beim Kantenwachstum regelmäßig um 10:00 Uhr erreicht, beim Knotenwachstum liegt diese Zeit zwischen 10:00 und 11:00 Uhr. Ausgehend von der Annahme, dass zum Thema Trump vor allem in den US-Staaten getweetet wird, kann eine Zeitdifferenz von mindestens sechs Stunden angenommen werden. Zu dieser Zeit wäre es in den USA also spätestens 4:00 Uhr nachts, in der die meisten Nutzer schlafen dürften.

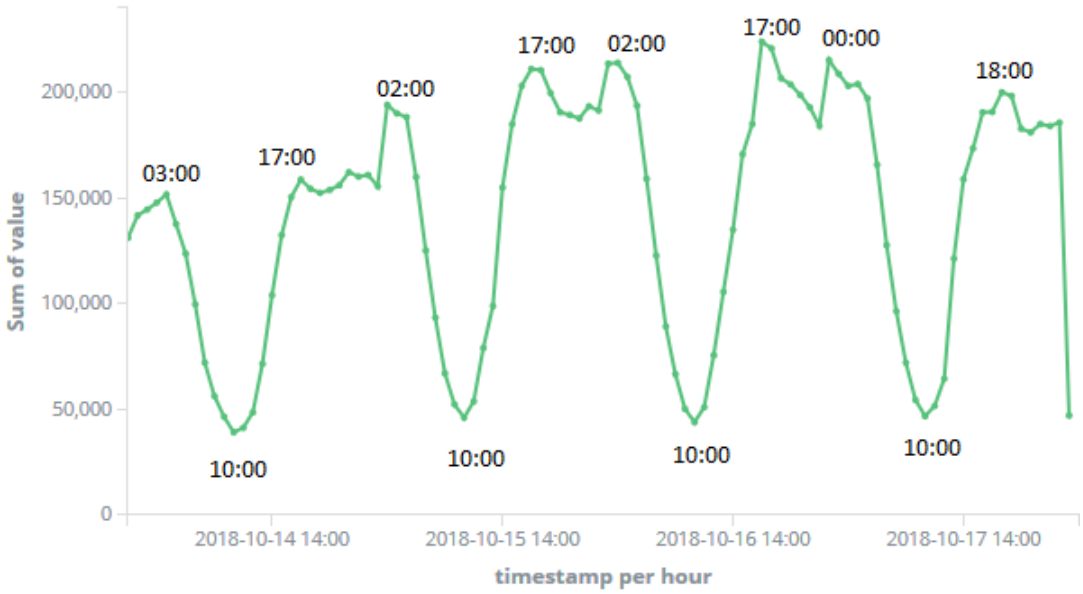


Abbildung 6.15.: Twitter Edge Growth über den Zeitraum 14.10. bis 18.10.2018

Der Hochpunkt wird dagegen beim Kantenwachstum zwischen 17:00 und 18:00 Uhr (11:00 bis 12:00 Uhr US-Zeit[5]) sowie zwischen 0:00 und 2:00 Uhr (18:00 bis 20:00 US-Zeit) erreicht. Beim Knotenwachstum liegt der Hochpunkt je nach Tag zwischen 16:00 und 19:00 Uhr (10:00 bis 13:00 US-Zeit). Während sich der Anstieg je nach Zeitzone am Vormittag bis hin zum Morgen mit der Frühstückszeit bzw. einer Mittagspause erklären ließe, dürfte bei den meisten Nutzern der Arbeitstag am Abend enden, was zu einer dortigen vermehrten Aktivität führt.

[5] Alle folgenden angegebenen US-Zeiten gehen von einer Zeitverschiebung von sechs Stunden aus.

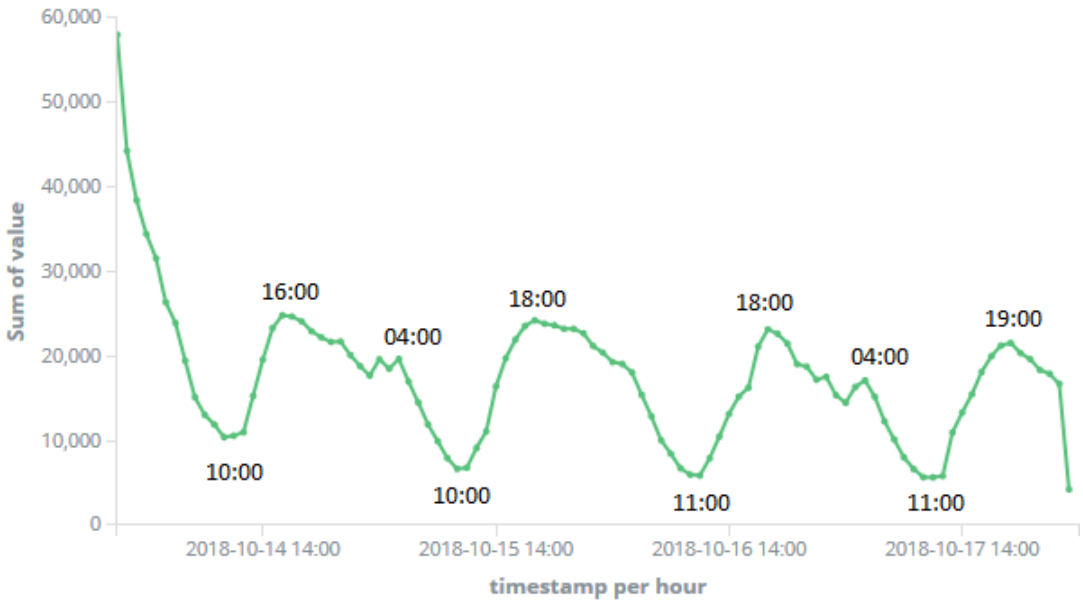


Abbildung 6.16.: Twitter Vertex Growth über den Zeitraum 14.10. bis 18.10.2018

Die Dichte des Twitter-Graphen (Abb. 6.17) wurde mit der Zeit so klein, dass sie sich in Kibana nicht mehr als Dezimalzahl darstellen ließ, sondern mit einer 0 abgekürzt wurde. Geht man jedoch von einer Kantenanzahl von 63 Mio. bei einer Knotenanzahl von 4 Mio. aus, so entspräche die daraus resultierende Dichte einem Wert von $0,000007875$ bzw. $7,875 \cdot 10^{-6}$, was in einem Netzwerk dieser Größe realistisch erscheint.

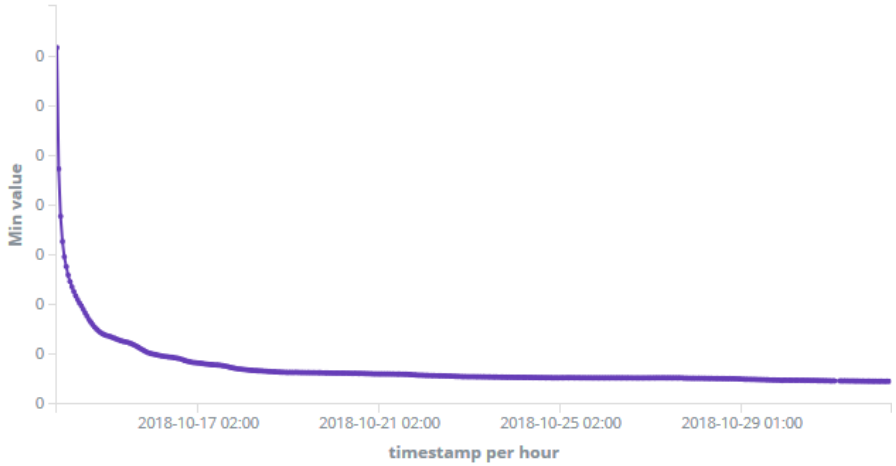


Abbildung 6.17.: Twitter Density

Der Average Clustering Coefficient (Abb. 6.18) unterliegt einem beschränkten Wachstum. Zunächst stark ansteigend erreicht der Clustering Coefficient nach fünf Tagen ein Maximum von 0,174, nimmt jedoch im restlichen Zeitraum wieder ab. Dieses Maximum deckt sich zeitlich mit dem Knick beim Vertex Count, was darauf hindeutet, dass ab diesem Zeitpunkt neu hinzugekommene Nutzer den Clustering Coefficient nach unten ziehen, also weniger dichte Verbindungen in ihrer Nachbarschaft aufweisen. Betrachtet man außerdem den Anstieg des Wertes bis zum Maximum, so kann am Kurvenverlauf ebenfalls eine Wellenform abgelesen werden, was zumindest in diesem Zeitraum einen Zusammenhang zwischen Wachstum des Graphen und Clustering Coefficient aufzeigt.

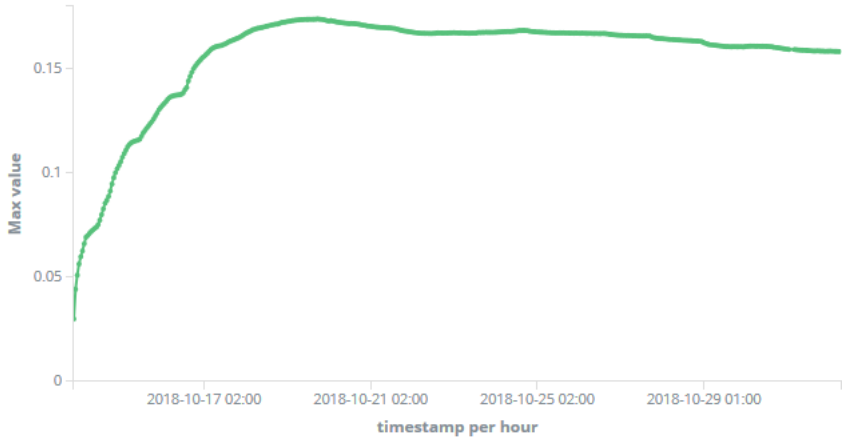


Abbildung 6.18.: Twitter Average Clustering Coefficient

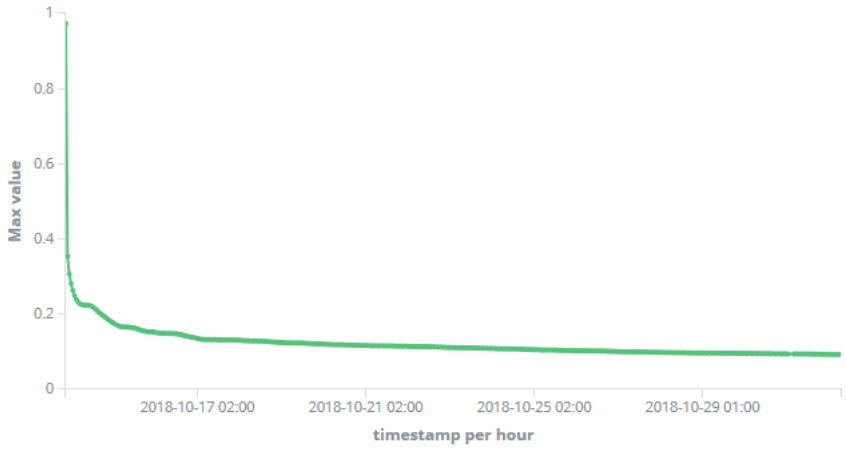


Abbildung 6.19.: Twitter Modularity

Die Modularität des Netzwerks (Abb. 6.19) beginnt mit einem relativ hohen Wert von 0,973, nimmt jedoch mit einer sehr hohen Geschwindigkeit ab und resultiert letztendlich in einer Abnahme von 0,16 bis 0,093, was eine relativ kleine Modularität darstellt. Im Rahmen dieser Arbeit gestaltete sich der Sonderfall, dass zu Anfang keine Communities bekannt waren und deshalb von null aufgebaut werden mussten. Bereits in Shang u. a. (2014) wurde die Beobachtung aufgestellt, dass in diesem Fall verschiedene Reihenfolgen eintreffender Kanten in verschiedenen Communitystrukturen resultieren und somit auch die Modularität des Netzwerks beeinflussen. Eine Lösung war in diesem Fall eine Berechnung einer initialen Communitystruktur im Vorfeld. Dieser initiale Schritt wurde in dieser Arbeit jedoch nicht umgesetzt. Die Aussagekraft der in dieser Arbeit implementierten Berechnung der Modularität sollte aus diesem Grund also mit Vorsicht genossen werden.

6.3. Vergleich mit verwandten Arbeiten

In diesem Abschnitt soll ein Vergleich der eigenen Arbeit mit verwandten Arbeiten stattfinden, die zum Teil in Abschnitt 2.3 behandelt wurden. Die eigene Arbeit basiert zu einem großen Teil auf Santoro u. a. (2011), in welcher aus dynamischen Graphen Metriken berechnet und anschließend interpretiert wurden. Innerhalb der eigenen Arbeit fand eine Auswahl und Implementation von Graphmetriken aus Santoro u. a. (2011) statt, wobei zusätzlich die Metriken Knoten- und Kantenanzahl sowie dessen Wachstum ergänzt wurden. Die Schnittmenge dieser Arbeit und Santoro u. a. (2011) liegt somit in den drei Metriken Dichte, Average Clustering Coefficient und Modularität. Während in der zugrundeliegenden Arbeit die Berechnungen auf temporalen Graphen erfolgten, berechnet und visualisiert das in der eigenen Arbeit realisierte System die Metriken zusätzlich in Echtzeit. Einen ähnlichen Ansatz verfolgten auch Kumar u. a. (2006). In dieser Arbeit wurde die Evolution der sozialen Netzwerke *Flickr* und *Yahoo! 360* analysiert. Ähnlich wie in Santoro u. a. (2011) fand die Analyse ebenfalls auf temporalen Graphen statt und erlaubte eine Darstellung der Evolution, jedoch ohne die Eigenschaft, Echtzeitdatenströme zu verarbeiten. In Henderson u. a. (2010) wurde ein System zur Berechnung diverser Metriken aus Graphdatenströmen entwickelt, der Fokus lag in dieser Arbeit jedoch auf der Analyse von IP-Netzwerken. Dementsprechend wurden Metriken wie Dichte, Clustering Coefficient und Modularität nicht berücksichtigt.

In Taxidou und Fischer (2013) fand die Auswertung eines Twitter-Datenstroms mit dem Ziel statt, die Informationsverteilung im sozialen Netzwerk Twitter nachzuvollziehen. Hierfür wurden Retweet-Kaskaden identifiziert und basierend darauf Ansätze zur Rekonstruktion solcher Kaskaden sowie zugehörige Metriken diskutiert. Die Verarbeitung des Datenstroms

fand in Echtzeit mithilfe von Apache Storm statt. Das Ergebnis war zum einen ein Graph, der die Anzahl der Tweets bzw. Retweets über die Zeit aufzeigt. Am ehesten kann dieser Graph mit den Wachstumsmetriken in dieser Arbeit verglichen werden, da beide einen wellenförmigen Verlauf aufweisen. Der Unterschied liegt lediglich darin, dass verschiedene Metriken betrachtet werden, nämlich zum einen das Knoten und Kantenwachstum in der eigenen Arbeit und das Wachstum von Tweets in Taxidou und Fischer (2013). Dies liegt daran, dass Knoten und Kanten aus dem Twitter-Datenstrom abgeleitet wurden. Zum anderen wurde eine Liste der 50 größten Retweet-Kaskaden mit dem Thema Olympische Spiele 2012 aufgezeigt. Während das Ziel von Taxidou und Fischer (2013) in der Rekonstruktion von Informationskaskaden lag, war das Ziel dieser Arbeit die Berechnung von Graphmetriken in Echtzeit sowie die Darstellung der Evolution über die Zeit.

In Wickramaarachchi u. a. (2015) wurde ein Ansatz zur Echtzeitverarbeitung von Graphen vorgestellt. Hierbei wurde eine Eigenentwicklung einer Stream Processing Engine (Floe) um eine Graphschnittstelle erweitert und anschließend Algorithmen implementiert, mit denen sich Key-Nodes innerhalb eines Graphen ermitteln lassen. Konkret wurden hierbei High-Degree Vertices, Knoten mit einem hohen Clustering Coefficient sowie Knoten berechnet, die als Startknoten einer Informationskaskade infrage kommen. Die Berechnung fand in Echtzeit statt, wobei der Unterschied zur eigenen Arbeit in den Ergebnissen liegt. Die Ergebnisse in Wickramaarachchi u. a. (2015) dienten zur Ermittlung von Key-Nodes, während die Ergebnisse der eigenen Arbeit die Evolution von sozialen Graphen beschreiben. In Marciani u. a. (2016) wurde das Big Data Framework Apache Flink benutzt, um aus einem Datenstrom in Echtzeit die k wichtigsten Posts bzw. die wichtigsten Communities bezogen auf ein Thema zu berechnen. Die Ergebnisse dieser Arbeit bilden somit ausschließlich den Ist-Zustand innerhalb eines Zeitfensters ab, während die Metriken in der eigenen Arbeit bis zu dem Zeitpunkt einsehbar sind, an dem der erste Datensatz des Datenstroms verarbeitet wurde und somit die Evolution des Graphen beschreiben.

7. Fazit

An diesem Punkt ist die Konzeption, Entwicklung und Evaluation der Plattform abgeschlossen. Im Folgenden wird zunächst die Arbeit zusammengefasst und daraufhin ein Ausblick auf potenzielle weiterführende Forschungsarbeiten gegeben.

7.1. Zusammenfassung

Ziel dieser Arbeit war die Konzeption und Entwicklung einer Plattform zur Echtzeitverarbeitung temporaler Graphen. Der Schwerpunkt lag dabei auf der Echtzeitanalyse sozialer Graphen und fokussierte sich auf die Berechnung diverser Graphmetriken sowie ihrer Evolution über die Zeit. In Kapitel 3 erfolgte zunächst eine Problembeschreibung sowie eine Spezifikation der Anforderungen, die das System erfüllen muss. Ebenfalls erfolgte eine Analyse existierender Algorithmen, Technologien und Werkzeuge, die eine Unterstützung bei der Lösung des Problems versprachen. In Kapitel 4 erfolgte daraufhin ein Entwurf des Systems unter Berücksichtigung der in Kapitel 3 aufgestellten Anforderungen. Hierbei kamen verschiedene Sichten zum Einsatz, um möglichst viele Aspekte des Entwurfs zu beleuchten. Ebenfalls erfolgte eine Festlegung auf die Kappa-Architektur, die speziell für die Verarbeitung von Datenströmen entworfen wurde. Innerhalb dieser Architektur kam bei der Realisierung in Kapitel 5 Apache Kafka als Master-Dataset und Ingestion-Layer zum Einsatz. Kafka ermöglichte die Verwaltung verschiedener Graphdatenströme und eine persistente Speicherung dieser Ströme, um eine wiederholbare Verarbeitung durch Flink zu ermöglichen.

Die nachfolgende Berechnung der Metriken erfolgte durch Apache Flink. Hier stellte sich heraus, dass innerhalb dieser Arbeit die Gelly-Streaming-API bis auf die Implementation des Algorithmus Triangle-Count keinen Mehrwert bot und die Implementation der Metriken stattdessen direkt auf DataStreams erfolgte. Durch eine Folge von Datenstrom-Transformationen wurde die Logik der einzelnen Algorithmen implementiert. Deren Gemeinsamkeit lag in der Datensenke ElasticSearch, in der die verschiedenen Metriken als Zeitreihe gespeichert wurden. Diese kontinuierlichen Zeitreihen wurden anschließend von Kibana ausgelesen und grafisch als Date Histogramme visualisiert. In Kapitel 6 erfolgte eine Auswertung dieser Diagramme unter Interpretation der Kurvenverläufe. Dabei stellte sich heraus, dass das Wachstum der Netzwerke

realen Ereignissen zugeordnet werden konnte und diese die Evolution der Netzwerke über die Zeit beeinflussen. Die abschließende Analyse eines Twitter-Datenstroms zeigte, dass sich auch kontinuierliche Echtzeitdatenströme durch die Plattform verarbeiten ließen und somit eine Echtzeitanalyse sozialer Graphen ermöglichten.

Im Rahmen dieser Arbeit erlaubten die Metriken Edge Growth und Vertex Growth die meisten Aussagen über die Entwicklung sozialer Graphen. Bezüglich der Modularität gestaltete sich der Fall, dass Communities von null auf berechnet werden mussten und nicht wie in Shang u. a. (2014) im Vorfeld eine Partitionierung existierte. Da die Reihenfolge der Kanten in diesem Fall die Communitystruktur beeinflusst, konnte die Aussagekraft die in dieser Arbeit implementierten Modularität nicht hinreichend untersucht werden und sollte aus diesem Grund mit Vorsicht genossen werden.

7.2. Ausblick

In diesem Abschnitt wir ein Ausblick auf weiterführende Forschungsarbeiten gegeben, die auf Basis dieser Arbeit denkbar wären:

- Innerhalb dieser Arbeit wurde lediglich ein Ausschnitt verschiedener Graphmetriken implementiert. Es existieren weitaus mehr Metriken als die in dieser Arbeit Behandelten. So kann beispielsweise geprüft werden, ob und wie sich weitere Metriken implementieren lassen und welche Aussagen diese Metriken über die Evolution sozialer Graphen über die Zeit erlauben.

- Die Erfüllung der Anforderung horizontale Skalierbarkeit **N1** erfolgte in dieser Arbeit lediglich durch den Entwurf sowie den gewählten Technologien. So erlauben sowohl Kafka, Flink als auch ElasticSearch durch Broker, Taskmanager bzw. Nodes eine Verteilung auf mehrere Maschinen. Eine Überprüfung der horizontalen Skalierbarkeit durch Skalierungstests fand jedoch innerhalb dieser Arbeit nicht statt und könnte in weiterführenden Arbeiten erfolgen.

- Als ebenfalls interessant gestaltet sich die Analyse sogenannter Multi-Layer Graphen. Diese zeichnen sich dadurch aus, dass ihre Kanten verschiedene Beziehungstypen abbilden. Im Twitter-Beispiel könnte so unterschieden werden, ob Nutzer erwähnt, retweetet oder zitiert werden. Die Verarbeitung von Graphen unter Berücksichtigung ihrer Beziehungstypen könnte eventuell andere Erkenntnisse über deren Evolution liefern.

- Die Analyse des Twitter-Datenstroms zeigte, dass das Interesse am Thema ”Trump” über den Zeitraum annähernd gleich blieb und aus Sicht der Kanten keine Abnahme des

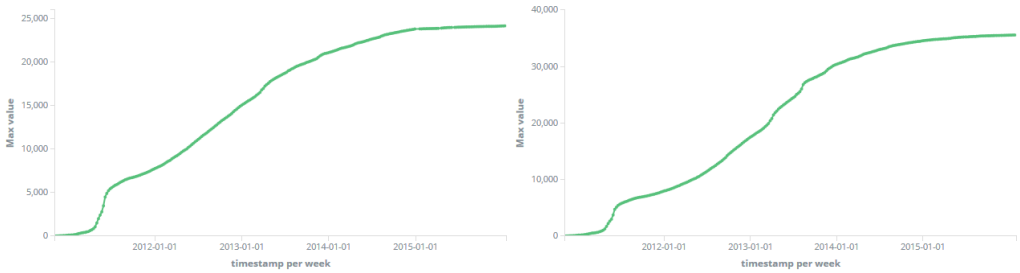
Wachstums erfolgte. Ebenfalls legte die Aufschlüsselung des Wachstums nach Uhrzeit die Vermutung nahe, dass zum Thema Trump am häufigsten in den USA getweetet wird, berücksichtigt man die Zeitverschiebung. Eine weitere Arbeit könnte durch Analysen anderer Themen das Wachstumsverhalten bei zeitlich gebundenen Ereignissen auswerten, etwa Wahlen oder sportliche Ereignisse.

- Ausgehend von der Entwicklung der Dichte und des Clustering Coefficient der EU-Core Datensätze könnte eine weitere Analyse zudem untersuchen, inwiefern sich die Wahl eines Themas auf Twitter auf die Clusterbildung auswirkt. So kann beispielsweise angenommen werden, dass der Average Clustering Coefficient höher ist, je spezieller das Thema ist.

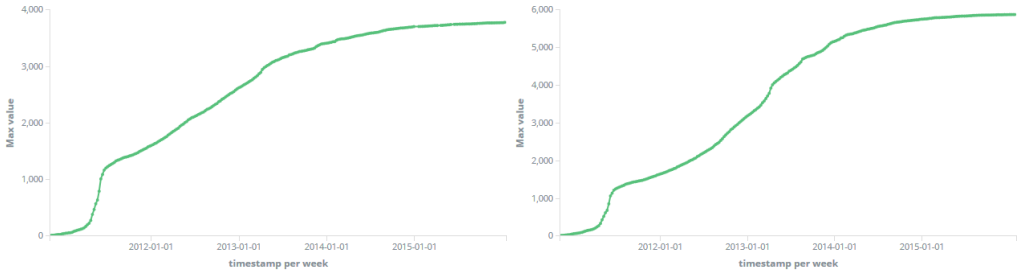
A. Diagramme

In diesem Kapitel sind alle Diagramme aufgeführt, die aus Übersichtlichkeitsgründen nicht in Kapitel 6 enthalten sind.

Bitcoin

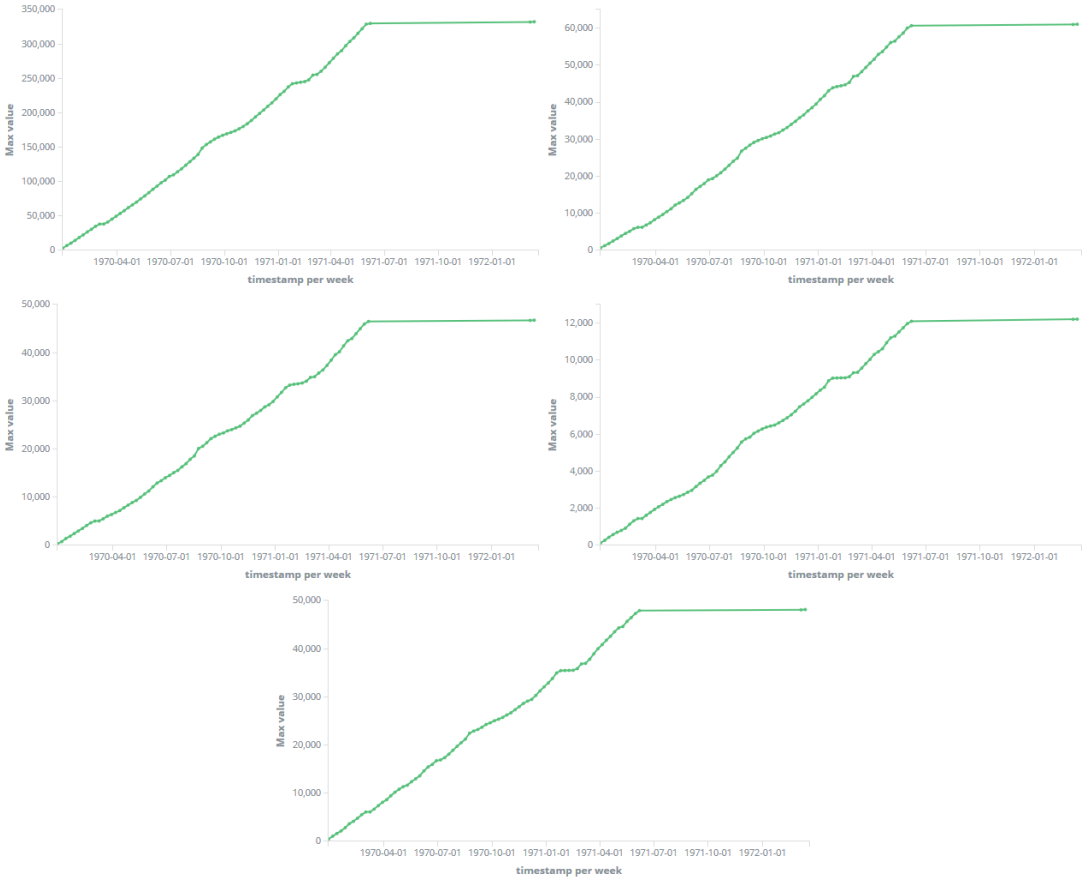


Edge Count: Bitcoin Alpha (links) und OTC (rechts)

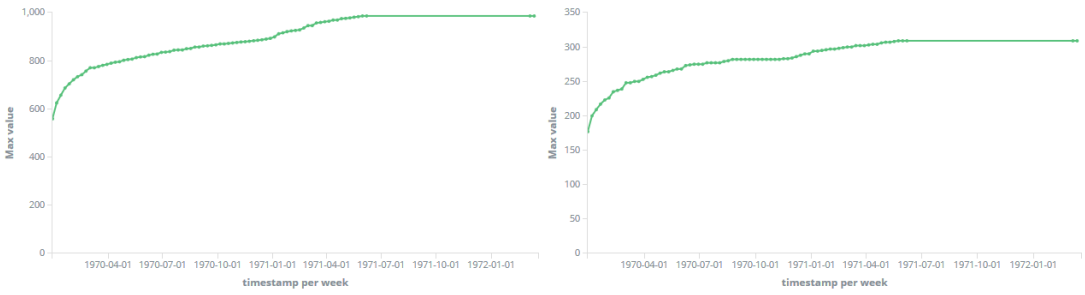


Vertex Count: Bitcoin Alpha (links) und OTC (rechts)

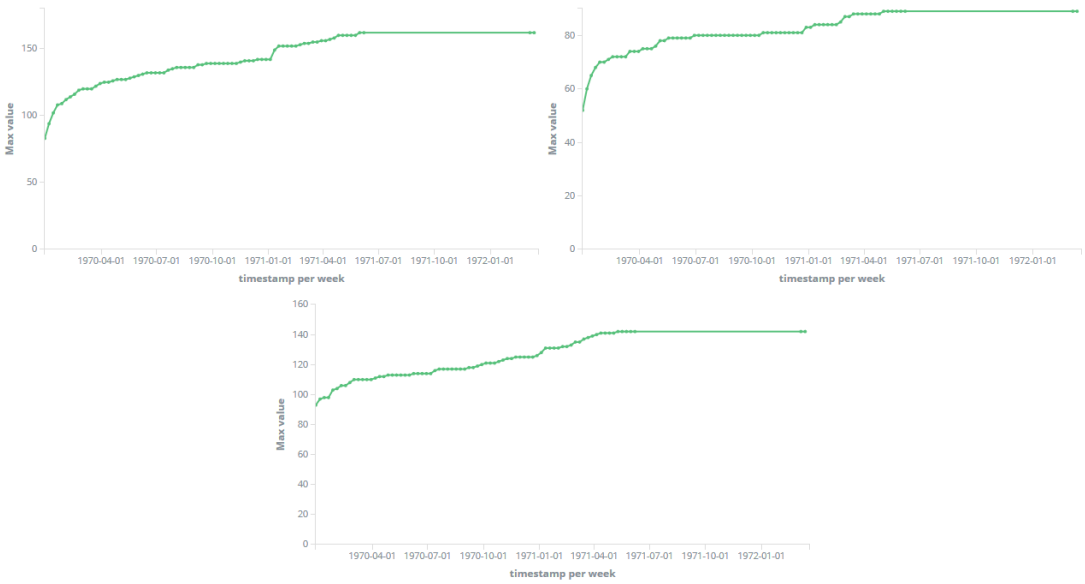
EU Core



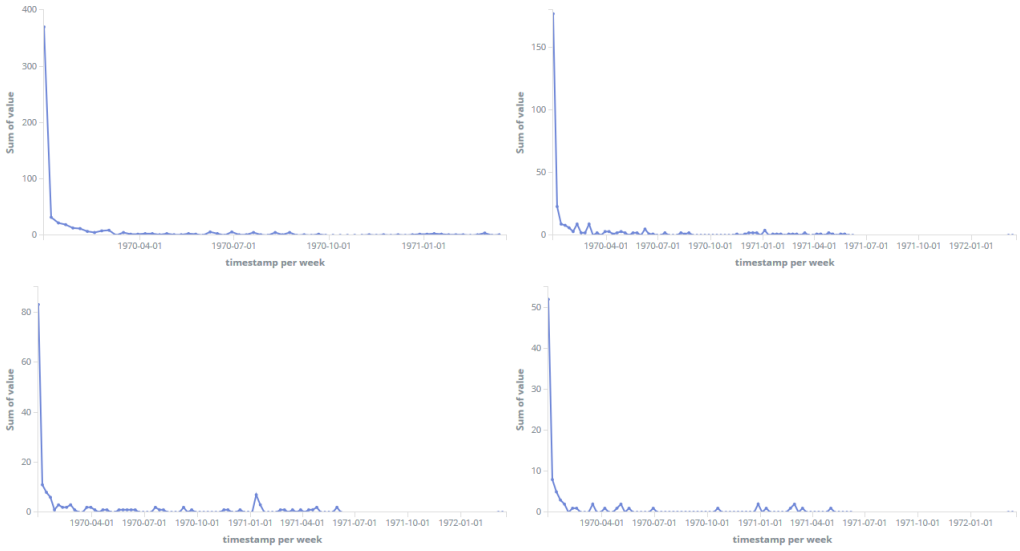
Edge Count: (1) *eu-core*, (2) *eu-core Dept. 1*, (3) *eu-core Dept. 2*, (4) *eu-core Dept. 3*, (5) *eu-core Dept. 4*



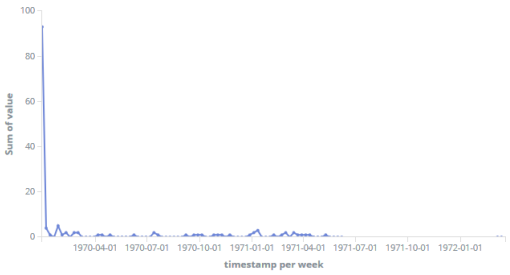
Vertex Count: (1) *eu-core*, (2) *eu-core Dept. 1*



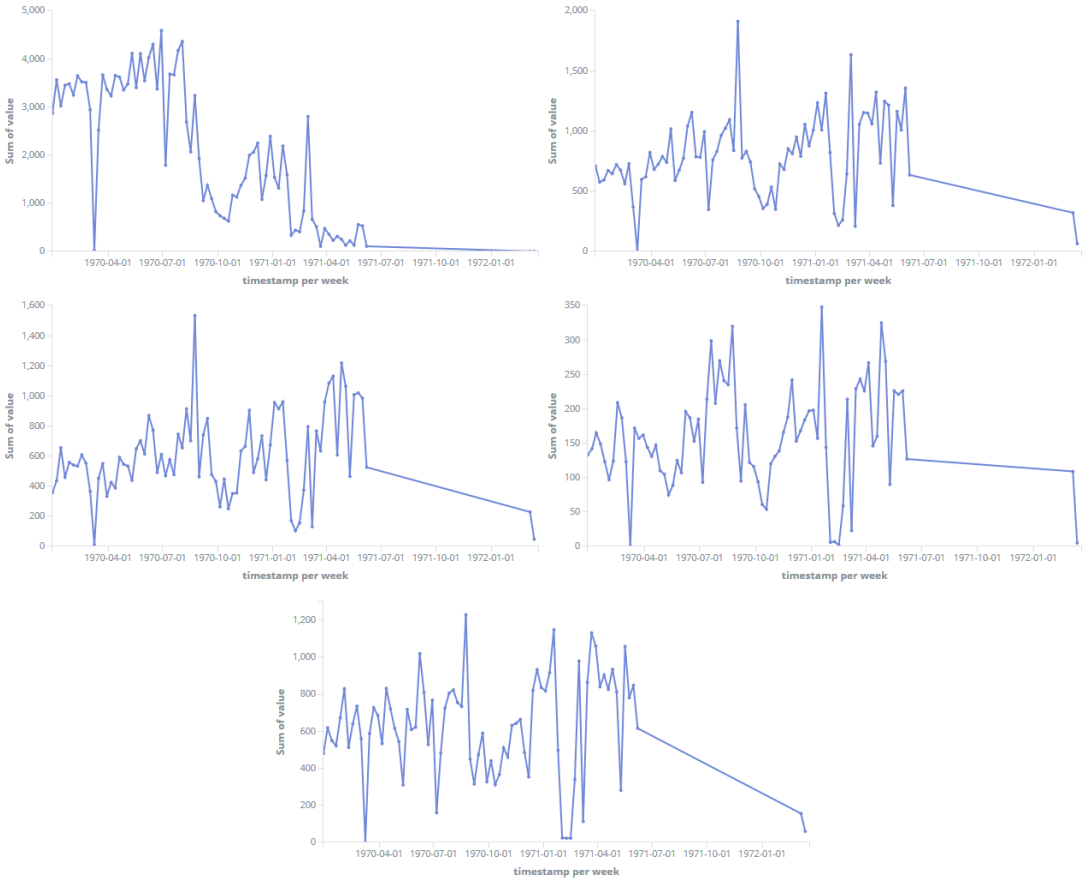
Vertex Count: (3) *eu-core Dept. 2*, (4) *eu-core Dept. 3*, (5) *eu-core Dept. 4*



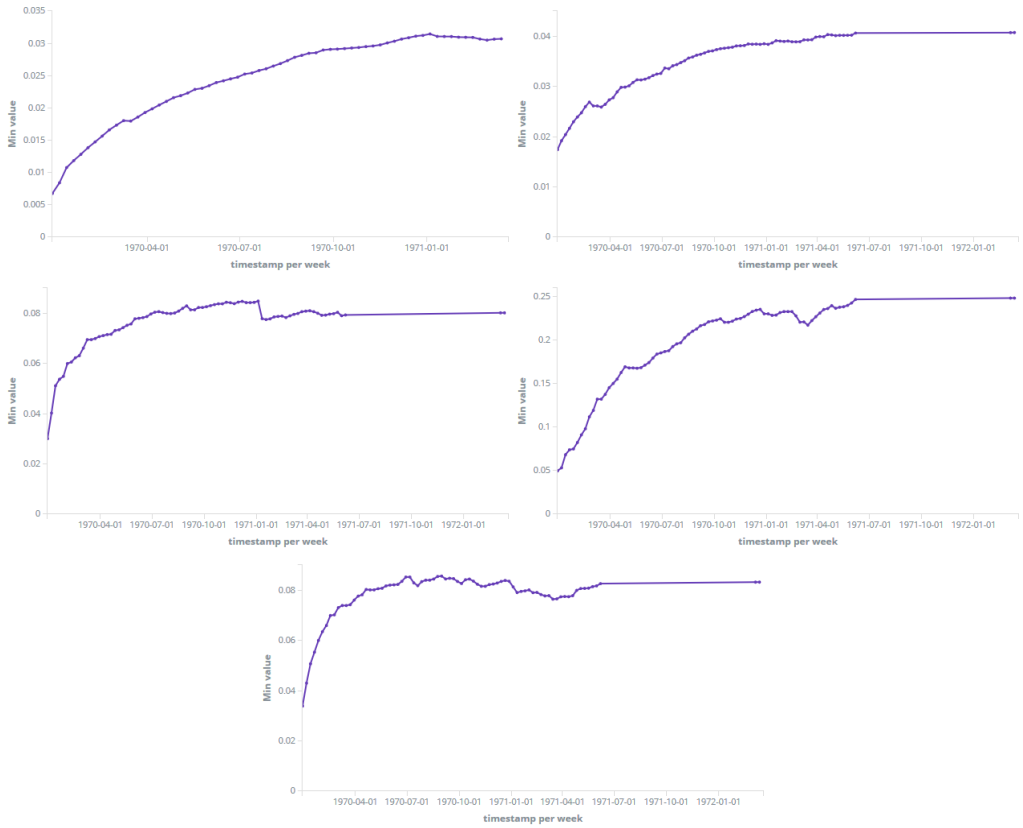
Vertex Growth: (1) *eu-core*, (2) *eu-core Dept. 1*, (3) *eu-core Dept. 2*, (4) *eu-core Dept. 3*



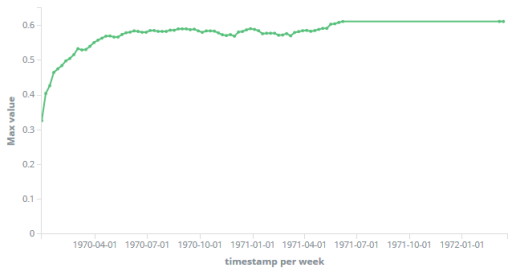
Vertex Growth: (5) *eu-core Dept. 4*



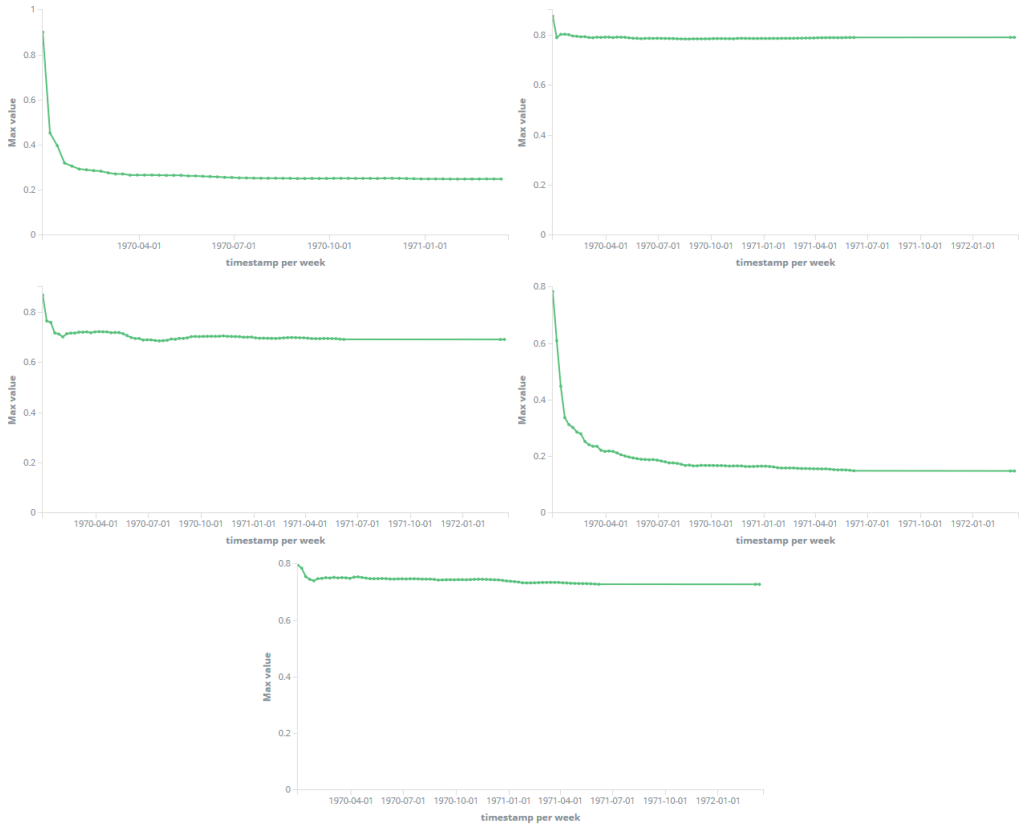
Edge Growth: (1) *eu-core*, (2) *eu-core Dept. 1*, (3) *eu-core Dept. 2*, (4) *eu-core Dept. 3*, (5) *eu-core Dept. 4*



Density: (1) *eu-core*, (2) *eu-core Dept. 1*, (3) *eu-core Dept. 2*, (4) *eu-core Dept. 3*, (5) *eu-core Dept. 4*

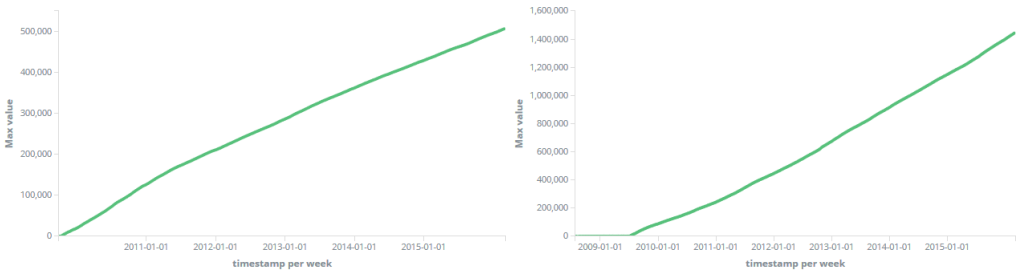


Average Clustering Coefficient: (1) *eu-core*, (2) *eu-core Dept. 1*, (3) *eu-core Dept. 2*, (4) *eu-core Dept. 3*, (5) *eu-core Dept. 4*

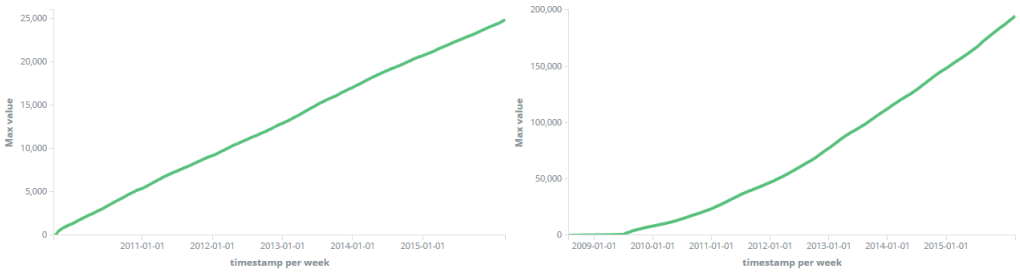


Modularity: (1) *eu-core*, (2) *eu-core Dept. 1*, (3) *eu-core Dept. 2*, (4) *eu-core Dept. 3*, (5) *eu-core Dept. 4*

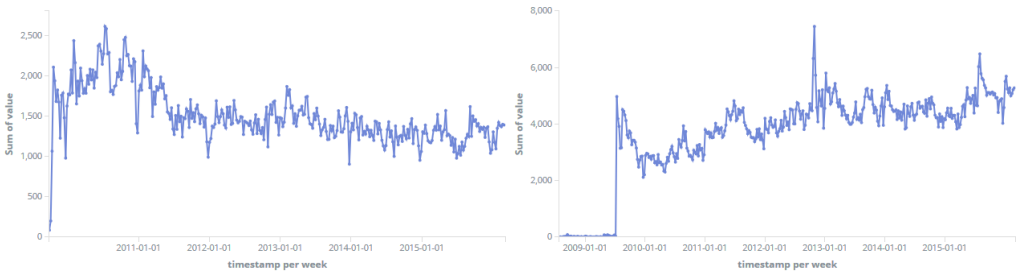
Stack Exchange



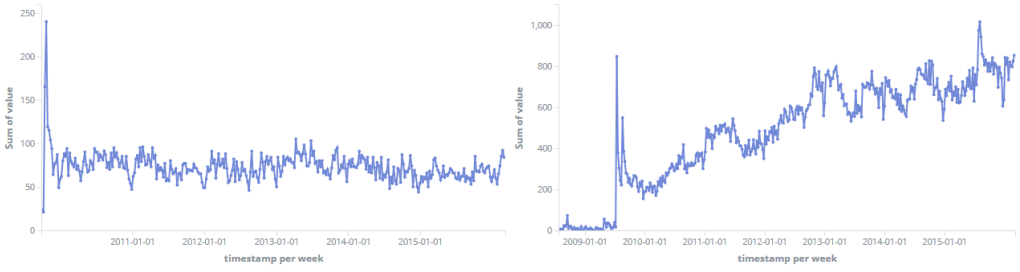
Edge Count: sx-mathoverflow (links) und sx-superuser (rechts)



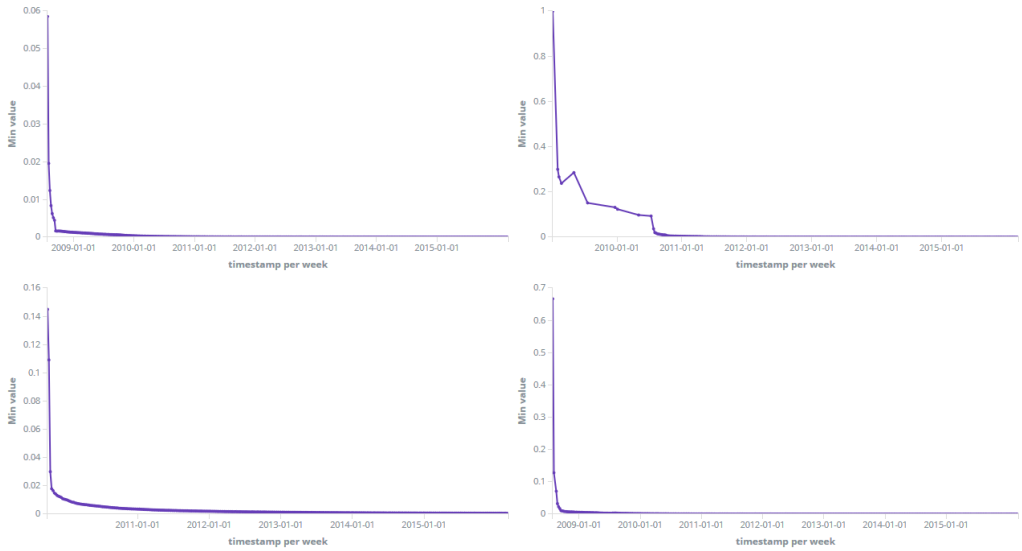
Vertex Count: sx-mathoverflow (links) und sx-superuser (rechts)



Edge Growth: sx-mathoverflow (links) und sx-superuser (rechts)

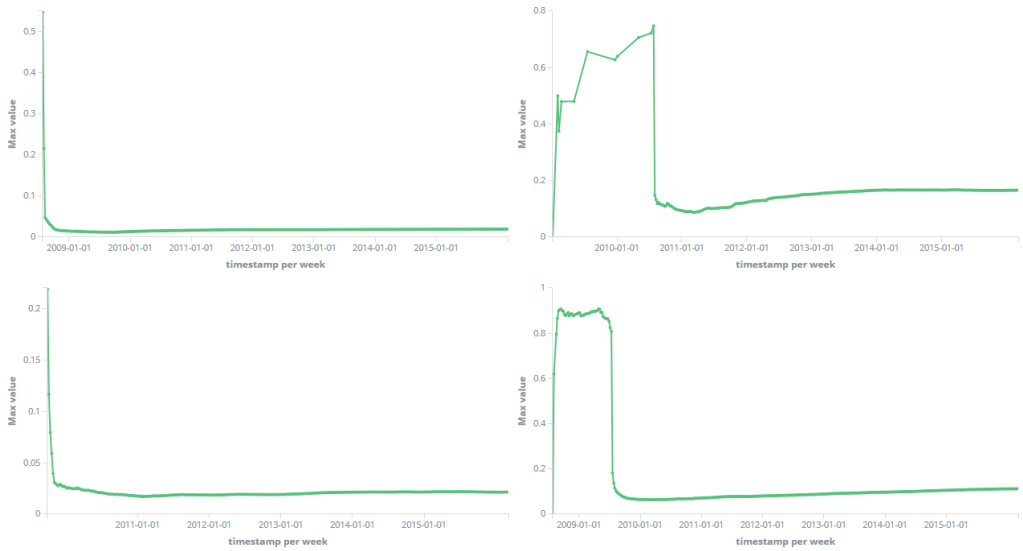


Vertex Growth: sx-mathoverflow (links) und sx-superuser (rechts)



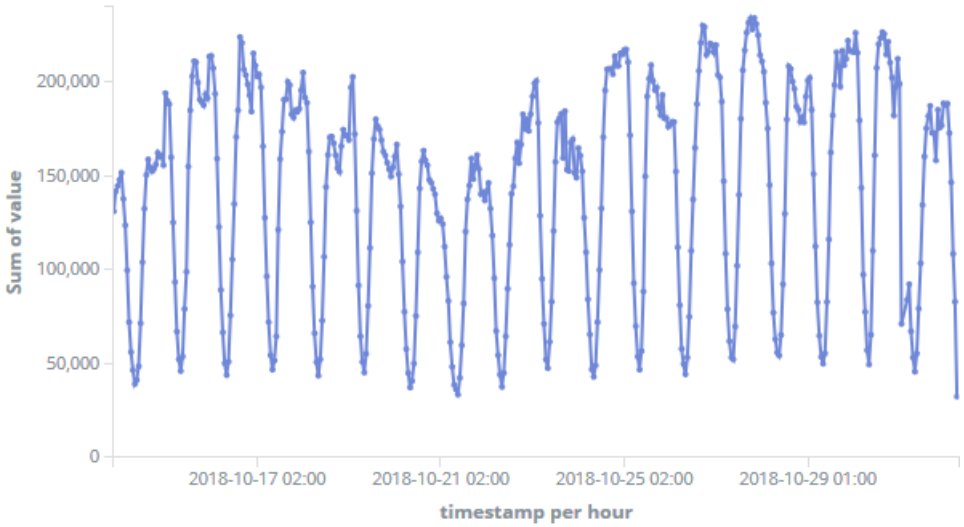
Density: (1) sx-stackoverflow, (2) sx-askubuntu, (3) sx-mathoverflow, (4) sx-superuser

Average Clustering Coefficient: (1) sx-stackoverflow, (2) sx-askubuntu, (3) sx-mathoverflow, (4) sx-superuser

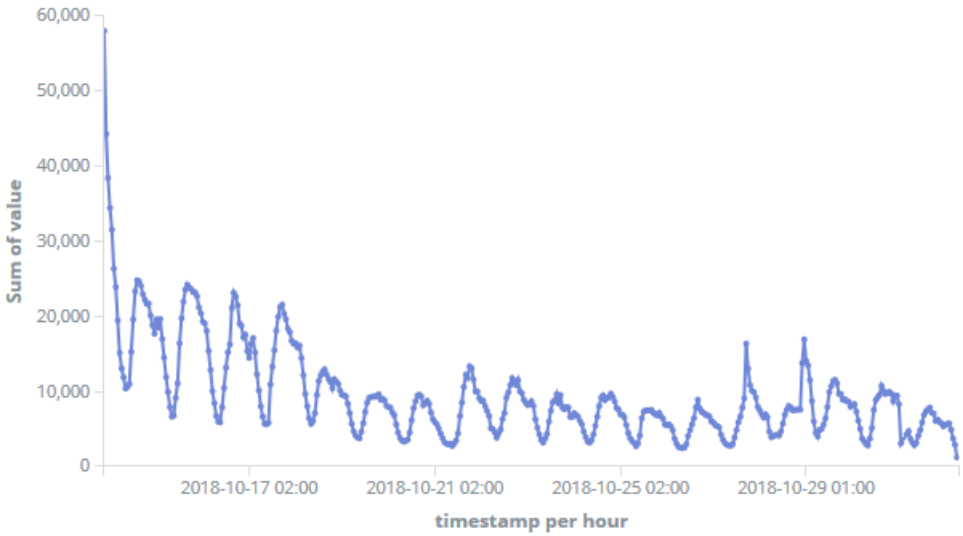


Modularity: (1) sx-stackoverflow, (2) sx-askubuntu, (3) sx-mathoverflow, (4) sx-superuser

Twitter



Twitter Edge Growth (vollständiger Zeitraum)



Twitter Vertex Growth (vollständiger Zeitraum)

Literaturverzeichnis

[Aggarwal 2006] Aggarwal, Charu C.: *Data Streams: Models and Algorithms (Advances in Database Systems)*. Berlin, Heidelberg : Springer-Verlag, 2006. – ISBN 0387287590

[Bali 2015] Bali, János D.: Streaming Graph Analytics Framework Design. URL http://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A830662&dswid=-8072. – Zugriffsdatum: 03.12.2018, 2015. – Forschungsbericht

[Berle 2017] Berle, Lukas: *Streamingarchitekturen in der Praxis: Lambda vs. Kappa*. 2017. – URL https://jaxenter.de/streaming-lambda-kappa-64573#. – Zugriffsdatum: 01.08.2018

[Brandes u. a. 2006] Brandes, U. ; Delling, D. ; Gaertler, M. ; Goerke, R. ; Hoefer, M. ; Nikoloski, Z. ; Wagner, D.: *Maximizing Modularity is hard*. 2006. – URL http://arxiv.org/abs/physics/0608255

[Bry u. a. 2004] Bry, François ; Furche, Tim ; Olteanu, Dan: Datenströme. In: *Informatik-Spektrum* 27 (2004), Apr, Nr. 2, S. 168–171. – URL https://doi.org/10.1007/s00287-004-0376-y. – ISSN 1432-122X

[Casteigts u. a. 2010] Casteigts, Arnaud ; Flocchini, Paola ; Quattrociocchi, Walter ; Santoro, Nicola: Time-Varying Graphs and Dynamic Networks. In: *CoRR* abs/1012.0009 (2010). – URL http://arxiv.org/abs/1012.0009

[Curtiss u. a. 2013] Curtiss, Michael ; Becker, Iain ; Bosman, Tudor ; Doroshenko, Sergey ; Grijincu, Lucian ; Jackson, Tom ; Kunnatur, Sandhya ; Lassen, Soren ; Pronin, Philip ; Sankar, Sriram ; Shen, Guanghao ; Woss, Gintaras ; Yang, Chao ; Zhang, Ning: Unicorn: A System for Searching the Social Graph. In: *Proc. VLDB Endow.* 6 (2013), August, Nr. 11, S. 1150–1161. – URL http://dx.doi.org/10.14778/2536222.2536239. – ISSN 2150-8097

[Dean und Ghemawat 2008] Dean, Jeffrey ; Ghemawat, Sanjay: MapReduce: Simplified Data Processing on Large Clusters. In: *Commun. ACM* 51 (2008), Januar, Nr. 1, S. 107–113. – URL http://doi.acm.org/10.1145/1327452.1327492. – ISSN 0001-0782

[Edlich u. a. 2011] Edlich, Stefan ; Friedland, Achim ; Hampe, Jens ; Brauer, Benjamin ; Brückner, Markus: *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken.* 2. München : Hanser, 2011. – ISBN 978-3-446-42753-2

[Estrada und Ruiz 2016] Estrada, Raul ; Ruiz, Isaac: *Big data SMACK – A Guide to Apache Spark, Mesos, Akka, Cassandra, and Kafka.* Berkeley, CA,; New York : Apress, 2016. – ISBN 9781484221754 1484221753

[Facebook 2016] Facebook: *Thank You! Messenger | Facebook Newsroom.* 2016. – URL https://newsroom.fb.com/news/2016/07/thank-you-messenger/. – Zugriffsdatum: 06.05.2018

[Fasel und Meier 2016] Fasel, Daniel (Hrsg.) ; Meier, Andreas (Hrsg.): *Big Data: Grundlagen, Systeme und Nutzungspotenziale.* Wiesbaden : Springer Vieweg, 2016 (Edition HMD). – ISBN 978-3-658-11588-3

[Feigenbaum u. a. 2005] Feigenbaum, Joan ; Kannan, Sampath ; McGregor, Andrew ; Suri, Siddharth ; Zhang, Jian: On Graph Problems in a Semi-streaming Model. In: *Theor. Comput. Sci.* 348 (2005), Dezember, Nr. 2, S. 207–216. – URL http://dx.doi.org/10.1016/j.tcs.2005.09.013. – ISSN 0304-3975

[Flink 2018a] Flink, Apache: *Apache Flink 1.3 Documentation: Iterations.* 2018. – URL https://ci.apache.org/projects/flink/flink-docs-stable/dev/batch/iterations.html. – Zugriffsdatum: 23.07.2018

[Flink 2018b] Flink, Apache: *Apache Flink 1.5 Documentation: Apache Flink Documentation.* 2018. – URL https://ci.apache.org/projects/flink/flink-docs-stable/. – Zugriffsdatum: 24.07.2018

[Flink 2018c] Flink, Apache: *Apache Flink 1.5 Documentation: Iterative Graph Processing.* 2018. – URL https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/gelly/iterative_graph_processing.html. – Zugriffsdatum: 23.07.2018

[Flink 2018d] Flink, Apache: *Apache Flink 1.5 Documentation: Library Methods.* 2018. – URL https://ci.apache.org/projects/flink/flink-

docs-stable/dev/libs/gelly/library_methods.html. – Zugriffsdatum: 23.07.2018

[Flink 2018e] Flink, Apache: *Apache Flink 1.6 Documentation: Checkpointing*. 2018. – URL https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/state/checkpointing.html#checkpointing. – Zugriffsdatum: 22.08.2018

[Flink 2018f] Flink, Apache: *Apache Flink 1.6 Documentation: Streaming Connectors*. 2018. – URL https://ci.apache.org/projects/flink/flink-docs-stable/dev/connectors/. – Zugriffsdatum: 22.08.2018

[Fortunato 2010] Fortunato, Santo: Community detection in graphs. In: *Physics Reports* 486 (2010), Nr. 3, S. 75 – 174. – URL http://www.sciencedirect.com/science/article/pii/S0370157309002841. – ISSN 0370-1573

[Gonzalez u. a. 2012] Gonzalez, Joseph E. ; Low, Yucheng ; Gu, Haijie ; Bickson, Danny ; Guestrin, Carlos: PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA : USENIX Association, 2012 (OSDI'12), S. 17–30. – URL http://dl.acm.org/citation.cfm?id=2387880.2387883. – ISBN 978-1-931971-96-6

[Gonzalez u. a. 2014] Gonzalez, Joseph E. ; Xin, Reynold S. ; Dave, Ankur ; Crankshaw, Daniel ; Franklin, Michael J. ; Stoica, Ion: GraphX: Graph Processing in a Distributed Dataflow Framework. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA : USENIX Association, 2014 (OSDI'14), S. 599–613. – URL http://dl.acm.org/citation.cfm?id=2685048.2685096. – ISBN 978-1-931971-16-4

[Henderson u. a. 2010] Henderson, Keith ; Eliassi-Rad, Tina ; Faloutsos, Christos ; Akoglu, Leman ; Li, Lei ; Maruhashi, Koji ; Prakash, B. A. ; Tong, Hanghang: Metric Forensics: A Multi-level Approach for Mining Volatile Graphs. In: *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA : ACM, 2010 (KDD '10), S. 163–172. – URL http://doi.acm.org/10.1145/1835804.1835828. – ISBN 978-1-4503-0055-1

[Kalavri u. a. 2016] Kalavri, Vasia ; Bali, Dániel ; Carbone, Paris ; Abbas, Zainab: *gelly-streaming/ExactTriangleCount.java*. 2016. – URL https://github.com/vasia/

gelly-streaming/blob/master/src/main/java/org/apache/flink/graph/streaming/example/ExactTriangleCount.java. – Zugriffsdatum: 03.11.2018

[Kalavri u. a. 2017] Kalavri, Vasia ; Bali, Dániel ; Carbone, Paris ; Abbas, Zainab: *vasia/gelly-streaming: An experimental Graph Streaming API for Apache Flink*. 2017. – URL https://github.com/vasia/gelly-streaming. – Zugriffsdatum: 22.04.2018

[Kalavri und Carbone 2016] Kalavri, Vasia ; Carbone, Paris: *Gelly-Stream: Single-Pass Graph Streaming Analytics with Apache Flink*. 2016. – URL https://de.slideshare.net/vkalavri/gellystream-singlepass-graph-streaming-analytics-with-apache-flink. – Zugriffsdatum: 22.04.2018

[Krzyzanowski 2012] Krzyzanowski, Paul: *Exam 3 study guide, Bulk Synchronous Parallel & Pregel*. 2012. – URL https://www.cs.rutgers.edu/~pxk/417/exam/study-guide-3.html. – Zugriffsdatum: 27.06.2018

[Kubernetes 2018] Kubernetes: *Production-Grade Container Orchestration - Kubernetes*. 2018. – URL https://kubernetes.io/docs/concepts/architecture/cloud-controller/. – Zugriffsdatum: 02.07.2018

[Kumar u. a. 2006] Kumar, Ravi ; Novak, Jasmine ; Tomkins, Andrew: Structure and Evolution of Online Social Networks. In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA : ACM, 2006 (KDD '06), S. 611–617. – URL http://doi.acm.org/10.1145/1150402.1150476. – ISBN 1-59593-339-5

[Kumar u. a. 2016] Kumar, S. ; Spezzano, F. ; Subrahmanian, V. S. ; Faloutsos, C.: Edge Weight Prediction in Weighted Signed Networks. In: *2016 IEEE 16th International Conference on Data Mining (ICDM)*, URL https://ieeexplore.ieee.org/document/7837846, 12 2016, S. 221–230. – ISSN 2374-8486

[Leskovec und Krevl 2014] Leskovec, Jure ; Krevl, Andrej: *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. Juni 2014

[Luxembourg 2018] Luxembourg, Blockchain: *BTC auf USD: Bitcoin auf US-Dollar Marktpreis - Blockchain*. 2018. – URL https://www.blockchain.com/de/charts/market-price?timespan=all. – Zugriffsdatum: 06.11.2018

[Malewicz u. a. 2010] Malewicz, Grzegorz ; Austern, Matthew H. ; Bik, Aart J. ; Dehnert, James C. ; Horn, Ilan ; Leiser, Naty ; Czajkowski, Grzegorz: Pregel: A System for Large-scale Graph Processing. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2010 (SIGMOD '10), S. 135–146. – URL http://doi.acm.org/10.1145/1807167.1807184. – ISBN 978-1-4503-0032-2

[Marciani u. a. 2016] Marciani, Giacomo ; Piu, Marco ; Porretta, Michele ; Nardelli, Matteo ; Cardellini, Valeria: Real-time Analysis of Social Networks Leveraging the Flink Framework. In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. New York, NY, USA : ACM, 2016 (DEBS '16), S. 386–389. – URL http://doi.acm.org/10.1145/2933267.2933517. – ISBN 978-1-4503-4021-2

[Marović und Marić 2016] Marović, Mladen ; Marić, Mario: *Javantura v3 - Husky ? (y)our tool for tracking value in data*. 2016. – URL https://www.slideshare.net/hujak/javantura-v3-husky-your-tool-for-tracking-value-in-data-mladen-marovi-mario-mari. – Zugriffsdatum: 18.07.2018

[Marz 2011] Marz, Nathan: *How to beat the CAP theorem - thoughts from the red planet - thoughts from the red planet*. 2011. – URL http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html. – Zugriffsdatum: 03.11.2018

[McGregor 2014] McGregor, Andrew: Graph Stream Algorithms: A Survey. In: *SIGMOD Rec.* 43 (2014), Mai, Nr. 1, S. 9–20. – URL http://doi.acm.org/10.1145/2627692.2627694. – ISSN 0163-5808

[Page u. a. 1999] Page, Lawrence ; Brin, Sergey ; Motwani, Rajeev ; Winograd, Terry: The PageRank Citation Ranking: Bringing Order to the Web. / Stanford InfoLab. Stanford InfoLab, November 1999 (1999-66). – Technical Report. – URL http://ilpubs.stanford.edu:8090/422/. Previous number = SIDL-WP-1999-0120

[Paranjape u. a. 2016] Paranjape, Ashwin ; Benson, Austin R. ; Leskovec, Jure: Motifs in Temporal Networks. In: *CoRR* abs/1612.09259 (2016). – URL http://arxiv.org/abs/1612.09259

[Piper 2015] Piper, Andy: *Potential adjustments to Streaming API sample volumes - Announcements - Twitter Developers*. 2015. – URL https://twittercommunity.com/t/potential-adjustments-to-streaming-api-sample-volumes/31628. – Zugriffsdatum: 17.11.2018

[Rancher 2018] Rancher: *Playing Catch-up with Docker and Containers | Rancher Labs.* 2018. – URL https://rancher.com/playing-catch-docker-containers/. – Zugriffsdatum: 11.07.2018

[Santoro u. a. 2011] Santoro, Nicola ; Quattrociocchi, Walter ; Flocchini, Paola ; Casteigts, Arnaud ; Amblard, Frédéric: Time-Varying Graphs and Social Network Analysis: Temporal Indicators and Metrics. In: *CoRR* abs/1102.0629 (2011). – URL http://arxiv.org/abs/1102.0629

[Shang u. a. 2014] Shang, Jiaxing ; Liu, Lianchen ; Xie, Feng ; Chen, Zhen ; Miao, Jiajia ; Fang, Xuelin ; Wu, Cheng: A Real-Time Detecting Algorithm for Tracking Community Structure of Dynamic Networks. In: *CoRR* abs/1407.2683 (2014). – URL http://arxiv.org/abs/1407.2683

[SNAP 2018] SNAP: *SNAP: Network datasets: Higgs Twitter Dataset.* 2018. – URL https://snap.stanford.edu/data/higgs-twitter.html. – Zugriffsdatum: 06.06.2018

[Spark 2018a] Spark, Apache: *RDD Programming Guide - Spark 2.3.1 Documentation.* 2018. – URL http://spark.apache.org/docs/2.3.1/rdd-programming-guide.html#transformations. – Zugriffsdatum: 18.07.2018

[Spark 2018b] Spark, Apache: *Spark Streaming - Spark 2.3.1 Documentation.* 2018. – URL https://spark.apache.org/docs/latest/streaming-programming-guide.html. – Zugriffsdatum: 11.07.2018

[Spolsky 2008] Spolsky, Joel: *Stack Overflow Launches ? Joel on Software.* 2008. – URL https://www.joelonsoftware.com/2008/09/15/stack-overflow-launches/. – Zugriffsdatum: 13.11.2018

[Starke 2015] Starke, Gernot: *Effektive Software-Architekturen: Ein praktischer Leitfaden.* 7. München : Hanser, 2015. – ISBN 978-3-446-44361-7

[Storm 2018] Storm, Apache: *Apache Storm.* 2018. – URL http://storm.apache.org/. – Zugriffsdatum: 27.06.2018

[Taxidou und Fischer 2013] Taxidou, Io ; Fischer, Peter: Realtime Analysis of Information Diffusion in Social Media. In: *Proc. VLDB Endow.* 6 (2013), August, Nr. 12, S. 1416–1421. – URL http://dx.doi.org/10.14778/2536274.2536328. – ISSN 2150-8097

[Twitter 2018] Twitter: *Q3 2018 Letter to Shareholders*. 2018. – URL https://investor.twitterinc.com/static-files/a7a842bb-082a-41da-8884-55c6d43ea58a. – Zugriffsdatum: 13.11.2018

[Valiant 1990] Valiant, Leslie G.: A Bridging Model for Parallel Computation. In: *Commun. ACM* 33 (1990), August, Nr. 8, S. 103–111. – URL http://doi.acm.org/10.1145/79173.79181. – ISSN 0001-0782

[Wickramaarachchi u. a. 2015] Wickramaarachchi, C. ; Kumbhare, A. ; Frincu, M. ; Chelmis, C. ; Prasanna, V. K.: Real-Time Analytics for Fast Evolving Social Graphs. In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, URL https://ieeexplore.ieee.org/document/7152566, Mai 2015, S. 829–834

[Wikipedia 2018] Wikipedia: *Datenstrom - Wikipedia*. 2018. – URL https://de.wikipedia.org/wiki/Datenstrom. – Zugriffsdatum: 25.06.2018

[Zaharia u. a. 2012] Zaharia, Matei ; Chowdhury, Mosharaf ; Das, Tathagata ; Dave, Ankur ; Ma, Justin ; McCauley, Murphy ; Franklin, Michael J. ; Shenker, Scott ; Stoica, Ion: Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. Berkeley, CA, USA : USENIX Association, 2012 (NSDI'12), S. 2–2. – URL http://dl.acm.org/citation.cfm?id=2228298.2228301

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

Hamburg, 07.12.2018 Heinrich Latreider