Hochperformante Analysen in Graph-Datenbanken

Moritz Kaufmann, Tobias Mühlbauer, Manuel Then,
Andrey Gubichev, Alfons Kemper, Thomas Neumann
Technische Universität München
{kaufmanm,muehlbau,then,gubichev,kemper,neumann}@in.tum.de

Abstract: Ziel des ACM SIGMOD Programming Contest 2014 war es ein hochperformantes System für die Analyse von großen Graph-Daten zu entwickeln. Insbesondere die unregelmäßigen Speicherzugriffsmuster und Kontrollflussverzweigungen von Graphalgorithmen stellen dabei eine große Herausforderung dar, da diese bisher nicht effizient auf modernden superskalaren Mehrkern-Prozessoren ausgeführt werden können. Um diese Prozessoren optimal auszulasten bedarf es zudem der Nutzung aller parallelen Ausführungseinheiten.

In der vorliegenden Arbeit präsentieren wir das Gewinnersystem des Wettbewerbs. Der Erfolg unseres Systems beruht, neben gutem Engineering, auf den folgenden Entwicklungen: (i) Daten-parallelisierte Graph-Breitensuche, welche Cache-Misses effizient amortisiert, (ii) Heuristiken zur Reduzierung des Suchraums bei Top-k-Anfragen, (iii) schnelles parallelisiertes Laden von textuellen Rohdaten, und (iv) feingranulares Task-Scheduling um Mehrkern-Prozessoren optimal auszulasten. Die in dieser Arbeit beschriebenen Neuentwicklungen werden derzeit in unser Hauptspeicher-Datenbanksystem HyPer integriert und lassen sich unserer Einschätzung nach auch in bestehende Graph-Datenbanksysteme integrieren.

1 Einführung

Mit Facebook und Google+ existieren heute soziale Netzwerke mit mehreren Hundert Millionen aktiven Benutzern. Die Beziehungen der Menschen, ihre Interessen und Beiträge ergeben dabei einen Graphen mit Millionen von Knoten, Kanten und Eigenschaften. Neben sozialen Graphen existieren Graph-Daten unter anderem auch für Straßen- und Schienennetze und die Verlinkung von Webseiten. Durch die Analyse dieser Graph-Daten können nicht nur werberelevante Zielgruppen bestimmt werden, sondern auch wichtige Erkenntnisse für die Sozialwissenschaften, Medizin und Astronomie gewonnen werden.

Graphen lassen sich in traditionellen relationalen Datenbanken abbilden, indem Knoten und Kanten in Relationen gespeichert werden. Auch können Graphalgorithmen in SQL-Anfragen übersetzt werden. Zur Traversierung der Graph-Daten werden dazu häufig rekursive SQL-Anfragen (mit sog. *Recursive Common Table Expressions*) benutzt. Es ist jedoch leicht ersichtlich, dass die rekursive Berechnung von Joins sehr schnell zu einer hohen Komplexität bei der Berechnung der Anfrageergebnisse führt. Um diesem Engpass bei der Analyse großer Graph-Daten zu begegnen, werden seit einigen Jahren spezialisierte Graph-Datenbanksysteme in Industrie und Forschung entwickelt. Diese zeichnen sich durch eingebaute Graphalgorithmik und eine für diese Algorithmik angepass-

te Speicherung der Daten aus. Die mitgelieferten Algorithmen erlauben unter anderem die Traversierung von Graphen durch Tiefen- und Breitensuche und das Auffinden von kürzesten Pfaden und Cliquen. Zu den bekanntesten Graph-Datenbanksystemen gehören Neo4j [Neo14], HyperGraphDB [Ior10], und Sparksee (früher DEX) [MBMMGV+07]. Mit Pregel [MAB+10], FlockDB [Flo14] und Trinity [SWL13] haben Google, Twitter und Microsoft zudem jeweils ein Framework zur Analyse von großen Graph-Datenbanken in verteilten Systemen entwickelt. Neueste Forschungsergebnisse zeigen zudem, dass auch spaltenorientierte relationale Datenbanksysteme [JRW+14] zu den zuvor genannten Systemen konkurrenzfähig sein können.

Der ACM SIGMOD Programming Contest setzt sich zum Ziel jedes Jahr die Aufmerksamkeit auf eine forschungsrelevante Aufgabenstellung zu lenken und Studenten und Doktoranden zu animieren in einem Zeitraum von drei Monaten nach möglichst effizienten Lösungen zu suchen. Ziel des diesjährigen, mit einem Preisgeld von 5.000 US-$ dotierten, Programmierwettbewerbs war es die technischen Möglichkeiten eines Systems zur Analyse großer sozialer Graphen auf einem modernen superskalaren Mehrkern-Prozessor zu ermitteln. Die konkrete Aufgabe bestand darin vier verschiedene, parametrisierte Anfragetypen auf einem synthetisch generierten sozialen Graphen möglichst schnell zu beantworten [Pcs14]. Der Datengenerator wurde dabei dem *The Social Network Benchmark* des Linked Data Benchmark Council (LDBC) [BFG+13] entnommen[1]. Neben der Korrektheit war auch die Skalierbarkeit der Lösung bis hin zu sehr großen Graphen mit mehreren Millionen Knoten gefordert. Bezüglich der Ausführung waren die Programme des Wettbewerbs auf eine nicht verteile Umgebung beschränkt. Mit heutiger Rechenleistung und Arbeitsspeichermengen jenseits eines Terabytes in einem Server ist es aber möglich selbst Graphen mit mehreren hundert Millionen Knoten und Kanten im Hauptspeicher auf einem einzigen Server zu analysieren [APPB10]. Die effiziente Ausführung von Graphalgorithmen auf solchen Servern stellt allerdings eine große Herausforderung dar [LGHB07]. Die meisten Algorithmen zur Graphanalyse besitzen unregelmäßige Speicherzugriffsmuster und eine Vielzahl an Kontrollflussverzweigungen. Dies wird besonders bei Graph-Daten, die größer als die Caches auf modernen Prozessoren sind zum Problem, da die Ausführungszeit dann durch die Speicherzugriffslatenz bestimmt wird. Es ist daher unerlässlich neue Daten-parallele Algorithmen zu entwickeln, welche die Speicherzugriffslatenzen, die durch Cache-Misses entstehen, effizient amortisieren. Zudem ist es auch hilfreich, die Anzahl der schwer vorhersagbaren Kontrollflussverzweigungen und die damit verbundene Anzahl der *Branch-Misses* zu minimieren. Um moderne superskalare Mehrkern-Prozessoren optimal auszulasten, ist neben der Nutzung von Daten-Parallelität auch die parallele Ausführung auf allen Prozessorkernen notwendig. Dies erschwert die Implementierung von Graphalgorithmen zunehmend, da die Ausführungsgeschwindigkeit in vielen Dimensionen beeinflusst wird.

Durch die Implementierung von effizienten Graphalgorithmen, welche für Server mit modernen Mehrkern-Prozessoren und einem großen Hauptspeicher optimiert sind, war es mehreren Teams des SIGMOD Programming Contest möglich die Anfragezeiten der im Wettbewerb gestellten Aufgabentypen um mehr als eine Größenordnung gegenüber bisher

[1] Das LDBC [BFG+13] ist ein EU-Projekt, welches sich die Entwicklung von Benchmarks für die Analyse großer Graph- und RDF-Datenbanken zum Ziel gesetzt hat.

Rank	Team	Small (215 MB) [s]	Medium (7 GB) [s]	Large (23 GB) [s]
1	**AWFY (TUM)**	0.126	0.237	2.849
2	~~unknown~~	0.124	0.331	3.674
3	VIDA (NYU)	0.207	0.469	4.571
4	H_minor_free (UToyko)	0.127	0.366	6.046
5	~~VSB_TUO~~	0.185	0.699	6.824
6	~~Bolzano_Nguyen~~	0.127	0.497	9.833
7	blxlrsmb (Tsinghua Univ.)	0.126	0.555	9.852
8	Cracker (Peking Univ.)	0.193	0.765	12.276
9	UCY_YouSeeWhy	0.228	1.193	14.255
10	GGWP	0.471	2.372	28.689
11	Cloud11	1.358	2.651	29.411
12	zero	0.564	3.097	35.459
13	VedgeX	2.775	3.629	36.537
14	LogisticAggression	0.310	2.510	54.592
15	QQn00bs	0.174	2.734	61.193
16	car3x	0.385	3.885	186.663
17	GenericPeople	0.409	4.732	199.794
18	tow	0.740	9.843	411.609
19	Palios	1.239	9.282	N/A
20	parallel_while	0.773	14.018	N/A

Tabelle 1: Die Platzierungen der Top 20 des ACM SIGMOD 2014 Programming Contest vom 15. April 2014 [Pcs14]. Durchgestrichene Teams wurden nachträglich wegen falscher Ergebnisse disqualifiziert.

verfügbaren Graph-Datenbanksystemen zu verringern. Insgesamt nahmen 33 internationale Teams am Wettbewerb teil, aus welchen fünf Finalisten und ein Gewinner ermittelt wurden. Keiner der Finalisten baute dabei auf einem bestehenden Graph-Datenbanksystem auf; die Systeme wurden von Grund auf neu implementiert.

In der vorliegenden Arbeit präsentieren wir unser Gewinnersystem des Wettbewerbs (Team AWFY). Tabelle 1 zeigt die Platzierungen der Top 20 vom 15. April 2014. Der Erfolg unseres Systems beruht, neben gutem Engineering, auf den folgenden Entwicklungen: (i) Daten-parallelisierte Graph-Breitensuche, welche Cache-Misses effizient amortisiert, (ii) Heuristiken zur Reduzierung des Suchraums bei Top-k-Anfragen, (iii) schnelles parallelisiertes Laden von textuellen Rohdaten, und (iv) feingranulares Task-Scheduling um Mehrkern-Prozessoren optimal auszulasten. Die in dieser Arbeit beschriebenen Neuentwicklungen lassen sich nach unserer Einschätzung in die meisten bestehenden Graph-Datenbanksysteme integrieren.

In den ersten beiden Teilen dieser Arbeit stellen wir zunächst eine Übersicht unseres Systems für den Wettbewerb und die vier parametrisierten Graph-Anfragetypen vor. Nachfolgend gehen wir auf bekannte Ansätze zur Beantwortung dieser Anfragen ein. Aufbauend auf diesen Ansätzen stellen wir dann unsere Neuentwicklungen vor. Teil 5 beschreibt unsere Daten-parallelisierte Graph-Breitensuche. Teil 6 präsentiert unsere Heuristiken für

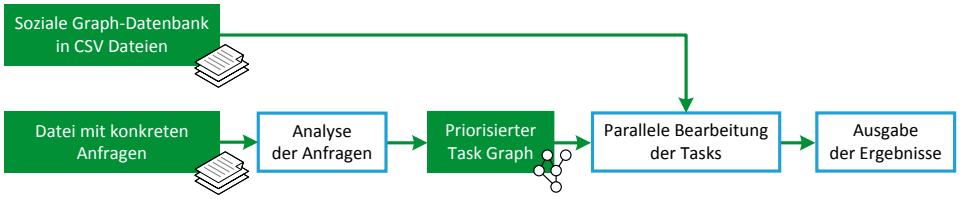

Abbildung 1: Bearbeitung der Wettbewerbsanfragen in unserem System.

Top-k-Anfragen. Im Anschluss zeigen wir, wie in textuellen Rohdaten gespeicherte Graphen schnell geladen werden können und beschreiben den Task-Scheduling-Ansatz unseres Systems. Abschließend fassen wir unsere Erkenntnisse zusammen und geben einen Überblick über zukünftige Aufgaben.

2 Übersicht unseres Systems und des Wettbewerbs

Unser System arbeitet, grob gesehen, in drei Phasen. Am Anfang müssen die konkreten Anfragen sowie der soziale Graph aus den textbasierten Dateien in unser System geladen werden. Die Anfragen analysieren wir und führen sie möglichst schnell aus. Die Verbindung zwischen dem Laden der notwendigen Datenbasis und der Ausführung der Anfragen gelingt uns mit einem priorisierten Task-Graph, der die verschiedenen Schritte parallel ausführt. In einem letzten Schritt werden dann die Ergebnisse in der erwarteten Reihenfolge ausgegeben. Dieser Ablauf ist in Abbildung 1 dargestellt.

Ausschlaggebend für die Bewertung im Wettbewerb war die Gesamtlaufzeit des Systems. Eingesandte Systeme wurden auf einer kleinen Graph-Datenbank mit 1.000 Personen (215 MB), einer mittleren Graph-Datenbank mit 10.000 Personen (7 GB) und einer großen Graph-Datenbank mit 100.000 Personen (23 GB) getestet. Nach Einsendeschluss wurden dann alle Systeme auch auf einer sozialen Graph-Datenbank mit einer Million Personen gebenchmarkt. Die Graph-Datenbanken lagen dabei jeweils in 33 mit dem LDBC *Social Network Data Generator* generierten CSV-Dateien vor [LDB14]. Die zentralen Dateien enthielten personenbezogene Informationen, Interessen der gespeicherten Personen, Freundschaftsbeziehungen und ausgetauschte Nachrichten. Die Freundschaftsbeziehung und der dadurch definierte Graph ist dabei ungerichtet, d.h., wenn Person a mit Person b befreundet ist, so ist auch b mit a befreundet. Dieser Graph dient als Grundlage für alle Anfragetypen. Alle in dieser Arbeit besprochenen Algorithmen operieren daher auf ungerichteten Graphen. Da nicht alle Informationen aus dem LDBC Datensatz zur Beantwortung der Anfragetypen des Wettbewerbs benötigt wurden, konnten nicht benötigte Daten bereits im Ladevorgang verworfen werden.

Um eine Vorberechnung der Ergebnisse zu verhindern, wurden die Anfragen parametrisiert und die Anfrageparameter auf dem Auswertungsserver verdeckt gewählt. Da die Aufgabentypen in ihrer Berechnungskomplexität stark schwanken, wurde die Anzahl der konkreten Anfragen eines Typs mit abnehmender Komplexität hochskaliert. Weil zudem

alle Anfragen von Anfang an zur Verfügung standen, konnte die Ausführung der Anfragen intern umsortiert werden. Bei der Ausgabe der Ergebnisse hingegen musste die ursprüngliche Reihenfolge wieder berücksichtigt werden. Wie in Teil 8 beschrieben, war es uns so möglich eine effiziente Ausführungsreihenfolge der Tasks zu wählen.

Die offizielle Auswertung der Systeme erfolgte auf einem Server des Organisators des Wettbewerbs. Dieser war fest spezifiziert und stellte die Referenzplattform dar. Der Server hatte zwei Intel Xeon E5430 CPUs mit je 4 Kernen ohne Hyper-Threading und 15 GB Hauptspeicher. Die Taktfrequenz der Kerne betrug 2.66 GHz, die letzte Cache-Ebene hatte eine Größe von 12 MB pro CPU. Als Betriebssystem wurde Linux eingesetzt. Die Menge des Hauptspeichers war für die getesteten Graphen mit bis zu einer Million Personen ausreichend, um alle Daten dort zu halten. Keines der Finalsysteme musste während der Ausführung Daten auf die Festplatte auslagern. Die Experimente in diesem Artikel wurden auf eigener Hardware ausgeführt. Unser System hat zwei Intel Xeon X5460 CPUs mit je 4 Kernen (8 Hyper-Threads) und 48 GB Hauptspeicher. Die Taktfrequenz beträgt 3.16 GHz und die letzte Cache-Ebene hat eine Größe von 12 MB pro CPU. Da nur Anfragen für die sehr kleinen 1.000- und 10.000-Personen-Netzwerke veröffentlicht wurden, haben wir darauf basierend entsprechende Anfragen für die 100.000 und 1 Million Netzwerke generiert. Falls nicht anders gekennzeichnet, wurden alle Experimente auf einer Graph-Datenbank mit 100.000 Personen ausgeführt.

3 Anfragetypen

Anfragetyp 1: Kürzeste Pfade entlang von Kommunikationswegen

Im ersten Anfragetyp wird der kürzeste Pfad zwischen zwei Personen a und b in einem ungerichteten Kommunikationsgraphen gesucht. In diesem Graphen sind Kanten zwischen befreundeten Personen enthalten, die miteinander kommuniziert haben. Durch einen zusätzlichen Anfrageparameter x ist definiert, wie oft zwei Personen mindestens miteinander Nachrichten ausgetauscht haben müssen, damit ihre Verbindung in der Pfadsuche berücksichtigt werden soll. Effektiv führt dies dazu, dass jede konkrete Anfrage des ersten Typ auf einem eigenen Subgraphen ausgeführt wird.

Eingabe: query1(a, b, x)

Anfragetyp 2: Suche der Interessengebiete mit den meisten Fans

Im zweiten Anfragetyp werden die k Interessen mit den größten zusammenhängenden Gruppen von Fans gesucht; wir nennen diese Gruppen im Folgenden Interessengruppen. Im sozialen Graph kann jede Person mehrere Interessen haben. Wenn sich mehrere befreundete Personen ein Interessengebiet teilen, bilden sie eine Interessengruppe. Für die Anfrage ist für jedes Interessengebiet nur die größte Gruppe von Personen relevant, in der alle Personen das Interessengebiet teilen und transitiv mit allen anderen Personen in dieser Gruppe befreundet sind. Als weitere Einschränkung müssen die Personen in dieser Gruppe weiterhin nach einem bestimmten Tag b geboren sein. Durch den Parameter b wird somit auch jede konkrete Anfrage des zweiten Typs auf einem eigenen Subgraphen ausgeführt.

Eingabe: query2(k, b)

Anfragetyp 3: Kontaktvorschläge

Das Ziel der Anfragen des dritten Typs ist es die k ähnlichsten Personenpaare in einem Netzwerk zu finden. Als Kriterium für die Ähnlichkeit zwischen zwei Personen dient die Anzahl der geteilten Interessengebiete. Als weitere Einschränkung müssen beide Personen am gleichen Ort p ansässig sein. Im Falle einer Einschränkung auf $p =$ Deutschland sind somit nur Personen relevant, welche innerhalb Deutschlands leben, arbeiten oder studieren. Die geographischen Orte sind dabei als Hierarchien definiert. Neben der örtlichen Einschränkung müssen beide Personen zusätzlich über eine begrenzte Anzahl von Freunden h transitiv miteinander verbunden sein. Diese Distanz wird über den Freundschaftsgraph bestimmt. Die Freunde, über die sie verbunden sind, müssen dabei die Ortseinschränkung nicht zwangsweise erfüllen.

Eingabe: query3(k, h, p)

Anfragetyp 4: Zentrale Personen

Anfragen des vierten Typs suchen die k einflussreichsten Personen in Foren zu einem bestimmten Interessengebiet i. Der Einfluss einer Person bestimmt sich dadurch, wie gut sie mit allen anderen Personen die in diesen Foren teilnehmen befreundet ist. Als Maß wird die *Closeness Centrality* [PKST11] genutzt:

$$\frac{(r(p)-1)*(r(p)-1)}{(n-1)*s(p)}$$

Der Wert von $r(p)$ gibt dabei für eine Person p an, wie viele Personen von ihr aus im Freundschaftsgraph erreicht werden können. $s(p)$ gibt für eine Person die Summe über die Distanzen zu allen anderen erreichbaren Personen an. n ist die Anzahl der Personen im Graph. Falls der Zähler oder Nenner 0 sind, ist der Wert der *Closeness Centrality* auch 0. Es wird wiederum nur ein Subgraph des Freundschaftsgraph betrachtet, welcher definiert ist durch Freunde, die auch Mitglieder in einem der Foren für ein Interessengebiet i sind. Eine statische Vorberechnung der Distanzen ist daher nicht möglich.

Eingabe: query4(k, i)

4 Bekannte Ansätze

Die vier verschiedenen Anfragetypen lassen sich großteils auf wenige generische Graphprobleme abbilden: Anfragetyp 1 ermittelt kürzeste Pfade, Anfragetypen 2 berechnet Zusammenhangskomponenten, Anfragetypen 3 untersucht Personenpaare in *k-hop*-Nachbarschaften und berechnet die Schnittmenge von deren Interessen und Anfragetypen 4 entspricht der Berechnung des kürzesten Pfades zwischen allen Paaren von Personen (*All Pairs Shortest Path*).

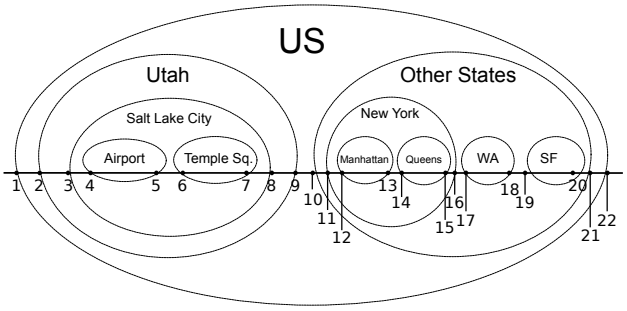

Abbildung 2: Beispiel eines *Nested Interval Encoding* für Orte

Eine sehr effizienter Algorithmus um kürzeste Pfade zwischen zwei Knoten in einem ungerichteten Graphen zu berechnen ist bidirektionale Breitensuche [ES11]. Hierbei wird jeweils am Start- und Ziel-Knoten eine Breitensuche gestartet. Die beiden Breitensuchen werden abwechselnd so lange ausgeführt, bis sie sich in mindestens einem Knoten treffen. Die Summe der Tiefen beider Breitensuchen ergibt dann die Länge des kürzesten Pfades zwischen den beiden Knoten. Breitensuchen sind weiterhin die effizienteste Lösung um Zusammenhangskomponenten in einem ungerichteten Graphen zu finden. *k-hop*-Nachbarschaften, also alle Knoten mit Distanz $\leq k$ vom Startknoten, entsprechen genauso einer beschränkten Breitensuche, die nach spätestens k Schritten terminiert. All Pairs Shortest Path entspricht einer Breitensuche von jedem Knoten aus. Zusammengefasst lassen sich also alle Anfragetypen mit einer Variante der Breitensuche lösen.

Als potenziell teure Probleme verbleiben noch das Ermitteln der Elemente im Schnitt von zwei Interessen-Mengen und die Überprüfung sowie das Feststellen der Ortszugehörigkeit von Personen im dritten Aufgabentyp. Wir verwenden dafür bekannte Ansätzen aus der Literatur. Zur Bildung von Schnittmengen nutzten wir einen neuen Daten-parallelen Ansatz von Lemire et al. [LBK14], welcher SIMD-Instruktionen auf Intel Prozessoren nutzt, um die Schnittmenge zwischen zwei sortierten Listen zu bilden. Das Sortieren der Interessenslisten pro Person konnte günstig während des Ladens der Daten durchgeführt werden. Um die Ortszugehörigkeit möglichst effizient zu testen, nutzten wir das *Nested Interval Encoding* [Tro05] um die Hierarchie der Orte zu speichern. Diese Technik weist jedem Ort einen Bereich auf einem Zahlenstrahl so zu, dass alle untergeordneten Orte innerhalb dieses Bereichs liegen. Jeder Ort ist somit durch einem Anfangs- und End-Zahlenwert beschrieben. Die Überprüfung, ob ein Ort Teil eines anderen Orts ist beschränkt sich somit auf zwei Vergleiche von Intervallgrenzen. Ein Beispiel hierfür haben wir in Abbildung 2 dargestellt. Falls beispielsweise in einer Anfrage die Zugehörigkeit des Orts *New York* zu *Utah* überprüft wird, so genügt es die Intervallgrenzen von New York (hier durch das Intervall $[11, 16]$ codiert) mit denen von Utah (mit dem Intervall $[2, 9]$) zu vergleichen. Da bereits $11 > 9$ ist, liegt New York nicht in Utah.

Im Folgenden gehen wir genauer auf die bekannten Ansätze zur Durchführung einer Breitensuche ein.

Den klassischen Algorithmus für eine Breitensuche[CSRL01] zeigen wir in Abbildung 3.

```
1  Input: G, s
2  seen ← {s}
3  visit ← {s}
4  visitNext ← ∅
5
6  while visit ≠ ∅
7      for each v ∈ visit do
8        for each n ∈ neighbors_v do
9            if n ∉ seen then
10             seen ← seen ∪ {n}
11             visitNext ← visitNext ∪ {n}
12             Mögl. Knotenspez. Berechn.
13     visit ← visitNext
14     visitNext ← ∅
```

Abbildung 3: Klassischer BFS Algorithmus

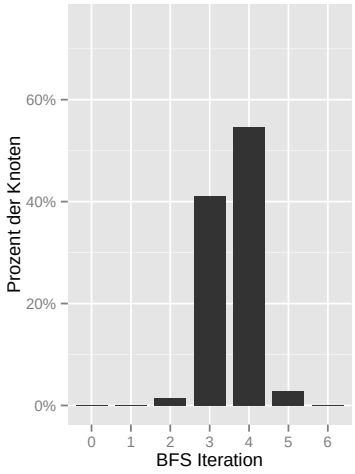

Abbildung 4: Anzahl der entdeckten Knoten pro Iteration

Als Eingabe erwartet die Breitensuche eine Graph-Struktur $G = (V, E)$ bestehend aus einer Knotenmenge V und einer Kantenmenge E und den Startknoten der Suche s. Während der Ausführung wird ausgehend von bekannten Knoten versucht neue bisher nicht besuchte Knoten zu erreichen. Um zu markieren, dass ein Knoten bereits bekannt ist, wird er bei erstmaligem Entdecken in der $seen$-Liste gespeichert. Initial enthält diese nur den Startknoten. Außerdem wird jeder Knoten in eine FIFO-Warteschlange eingereiht. Die Ausführung ist in Runden aufgeteilt. In jeder Runde wird die Liste der aktuell zu besuchenden Knoten $visit$ abgearbeitet und es wird versucht über die ausgehenden Kanten dieser Knoten neue Knoten zu entdecken. Alle neu entdeckten Knoten werden zur $visitNext$-Liste hinzugefügt. Diese Liste wird dann in der nächsten Iteration abgearbeitet. Die Distanz zum Startknoten ergibt sich aus der Anzahl der Iterationen, die benötigt werden, um einen Knoten zu erreichen.

Basierend auf diesem Algorithmus wurden verschiedene Ansätze entwickelt, um die Breitensuche zu beschleunigen. Diese können zur Vereinfachung in zwei Gruppen aufgeteilt werden: (i) Ansätze, die sich auf maximale Performanz auf einem Prozessorkern fokussieren und (ii) Ansätze die eine einzelne Breitensuche mit Hilfe mehrerer Prozessorkerne beschleunigen. Da die Parallelisierung einer einzelnen Breitensuchen immer zusätzliche Kosten für Synchronisation und Kommunikation bedeutet [BM11], hilft dies nur falls die Kerne sonst nicht ausgelastet werden können. Da wir in unserem Anwendungsfall um mehrere Größenordnungen mehr Breitensuchen als Prozessorkerne durchführen müssen, ist es sinnvoller mehrere sequenzielle Breitensuchen auf die Kerne zu verteilen. Allein Anfragetyp 4 muss potenziell von jedem Knoten aus eine Breitensuche starten.

Der schnellste sequenzielle Ansatz eine einzelne Breitensuche zu berechnen wurde von Beamer et al. [BAP12] vorgestellt. Die Verbesserung beruht auf der Beobachtung, dass in den letzten Iterationen einer BFS die Warteschlange $visit$ größer ist als die Menge

der noch nicht besuchten Knoten. Nur sehr wenige Nachbarn der Knoten in der Warteschlange sind noch nicht besucht, aber trotzdem müssen alle Nachbarn überprüft werden. Dies macht das Finden der letzten Knoten sehr teuer. Diese Beobachtung trifft auch auf den sozialen Graphen im Wettbewerb zu. In Abbildung 4 ist dargestellt, in welchen Iterationen jeweils wie viel Prozent der Knoten des 100.000-Personen-Netzwerks gefunden wurden. Nach der vierten Iteration sind nur noch 10% der Knoten nicht gefunden, aber die $visit$-Liste für die fünfte Iteration enthält 55% der Knoten im Graph. Um die Suchkosten für die letzten Knoten zu senken, schlagen die Autoren eine *bottom-up*-Variante vor, die das Vorgehen der Suche umkehrt. Anstatt ausgehend von den Knoten in der $visit$-Liste nach neuen Knoten zu suchen, überprüfen sie ob Knoten, die noch nicht gefunden wurden, einen Nachbarn haben, der in der $visit$-Liste enthalten ist. Dadurch wird in den letzten Iterationen die Performanz von der Anzahl der fehlenden Knoten bestimmt. Da dies nur Sinn macht, wenn die Anzahl der gefundenen Knoten abnimmt, wird zusätzlich eine Heuristik eingeführt um den Zeitpunkt des Wechsels auf die *bottom-up*-Variante zu entscheiden. Insgesamt schaffen die Autoren mit ihrem Ansatz auf sozialen Netzwerken eine Laufzeitverbesserung von ungefähr Faktor 4 [BAP12].

Ein großes verbleibendes Problem sind aber immer noch die vielen unregelmäßigen Speicherzugriffe, die ein effizientes *Prefetching* der Daten durch den Prozessor unterbinden. Da die Warteschlange nicht nach Knotenreihenfolge, sondern in Reihenfolge des Einfügens sortiert ist, entstehen beim Zugriff auf die Nachbarliste Sprünge im Speicher. Auch bei der Kontrolle jedes Nachbarn müssen die entsprechenden $seen$-Einträge aus nicht vorhersehbaren Speicherbereichen geladen werden. Diese Zugriffe schränken die Performanz signifikant ein, was besonders bei großen Graphen die nicht mehr in den Cache passen zu Geschwindigkeitseinbrüchen führt, da die Ausführungszeit in diesem Fall durch die Speicherzugriffslatenz bestimmt wird.

5 Daten-Parallelisierte Graph-Breitensuche

Um den Wettbewerb zu gewinnen, haben wir nach neuen Ansätzen zur Beschleunigung der Breitensuche gesucht. Eine zentrale Erkenntnis ist, dass Knoten in einem sozialen Graphen sehr dicht miteinander verknüpft sind. Die Breitensuche in Abbildung 4 beispielsweise terminiert nach nur sechs Iterationen. Eine weitere Beobachtung ist, dass die meisten Knoten in nur zwei Iterationen entdeckt werden. Daraus lässt sich ableiten, dass sich die Warteschlangen mehrerer Breitensuchen jeweils stark überlappen, da die meisten Breitensuchen über 80% der Knoten in diesen Iterationen entdecken. Sobald verschiedene Breitensuchen den gleichen Knoten in der gleichen Iteration entdecken, ist die Wahrscheinlichkeit, dass sie auch weitere Knoten zusammen entdecken, sehr groß. Es ergibt sich somit ein Potential diese Breitensuchen zu bündeln.

Basierend auf dieser Erkenntnis haben wir einen Ansatz entwickelt der erkennt, wann Breitensuchen an Knoten überlappen, diese dann zusammen ausführt und somit einen signifikanten Geschwindigkeitsgewinn erzielt. Die Hauptprinzipien beim Entwurf des neuen Ansatzes waren: (i) die hohen Kosten für die Zugriffe auf die Datenstrukturen zu reduzieren, indem man die gebündelten Suchen zusammen abarbeitet und somit die Speicher-

```
1  Input: G, 𝔹, S
2  seen[s_{b_i}] ← {b_i} for all b_i ∈ 𝔹
3  visit[s_{b_i}] ← {b_i} for all b_i ∈ 𝔹
4  visitNext[v] ← ∅ for all v ∈ G
5  queue_empty ← false
6
7  while queue_empty ≠ false
8      queue_empty ← true
9      for all v ∈ G
10       if visit[v] ≠ ∅
11           for each n ∈ neighbors_v
12             𝔻 ← visit[v] \ seen[n]
13             if 𝔻 ≠ ∅
14                 queue_empty ← false
15                 visitNext[n] ← visitNext[n] ∪ 𝔻
16                 seen[n] ← seen[n] ∪ 𝔻
17                 Mögl. Knotenspez. Berechn.
18     visit ← visitNext
19     visitNext[v] ← ∅ for all v ∈ G
```

Abbildung 5: Bitparallele Breitensuche

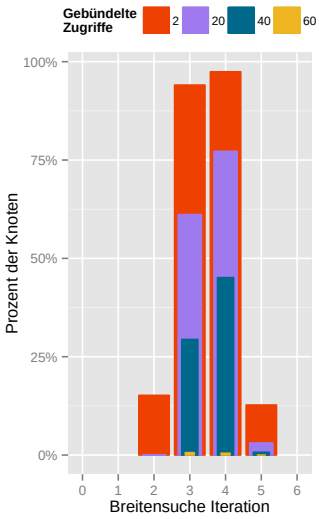

Abbildung 6: Anteil gebündelter Knotenzugriffe pro Iteration. Gruppiert nach Anzahl der gebündelten Breitensuchen.

zugriffslatenz der Cache-Misses amortisiert und (ii) die Bündelung auszunutzen, um die Ausführung durch den Einsatz von Bit-Operationen weiter zu beschleunigen. Der von uns entworfene Algorithmus ist in Abbildung 5 gezeigt.

Die Grundstruktur des Algorithmus im Vergleich zu einer klassischen Breitensuche bleibt erhalten. In jeder Iteration wird weiterhin versucht ausgehend von den entdeckten Knoten neue Knoten zu finden. Die Datenstrukturen und Operationen darauf müssen jedoch angepasst werden, um die Bündelung der Breitensuchen effizient zu ermöglichen. Statt eines einzelnen Startknotens als Eingabeparameter gibt es nun eine Menge von Breitensuchen $\mathbb{B}$ und entsprechend eine Menge von Startknoten S. Jede Breitensuche ist dabei durch einen Index b_i codiert, über den auch auf den entsprechenden Startknoten s_{b_i} zugegriffen werden kann. Für die Breitensuchen müssen dabei jeweils die Informationen, welche Knoten bereits gesehen wurden und welche Knoten in der Iteration bearbeitet werden, gespeichert werden. Die bisherigen binären $seen$- und $visit$-Listen reichen dafür nicht aus.

Um diese Informationen für mehrere Breitensuchen zu speichern, gibt es intuitiv zwei Möglichkeiten. Es kann zum Einen für jede Breitensuche separat jeweils eine binäre $seen$- und $visit$-Liste genutzt werden. Zum Anderen können weiterhin zwei $seen$- und $visit$-Listen genutzt werden, die zu jedem Eintrag die Menge der betreffenden Breitensuchen speichern; falls Knoten a z.B. schon von den BFS b_1, b_2 und b_3 gesehen wurde, dann sähe der entsprechende $seen[a]$ Eintrag so aus: $\{b_1, b_2, b_3\}$. Während die Implementierung der ersten Variante sehr leicht ist, hilft sie nicht das Problem der vielen Speicherzugriffe durch Bündelung signifikant zu reduzieren. Die Zugriffe auf die Nachbarschaftsinformationen

für gemeinsame Breitensuchen können zwar gebündelt werden, aber beim Zugriff auf die $seen$- und $visit$-Informationen der Nachbarn wären weiterhin für jede Breitensuche separate Speicherzugriffe notwendig. Die zweite Variante bietet unter diesem Aspekt mehr Potenzial um Kosten zu sparen.

Ein speichereffizienter und günstiger Weg die Mengen in den jeweiligen Einträgen zu speichern sind *Bitsets*. Für jede Breitensuche wird dabei die Information, ob sie den jeweiligen Knoten gesehen hat, oder besuchen wird mit einem Bit gespeichert. Jede Breitensuche besitzt eine einzigartige feste Bitposition. Um beispielsweise zu markieren, dass BFS b_3 den Knoten a besucht hat, wird im Bitset des Knoten das dritte Bit auf 1 gesetzt. Dies passiert z.B. bei der Initialisierung des Algorithmus in Zeile 2 des Algorithmus. Die Vereinigungsoperation $\cup$ kann dabei durch den binären oder-Operator | mit einem entsprechenden Bitvektor z.B. 00001000 für b_3 in einem Bitset mit acht Einträgen implementiert werden.

Bei der Umsetzung der Warteschlangen muss berücksichtigt werden, dass die Breitensuchen gebündelt werden sobald sie die gleichen Knoten in der gleichen Iteration erreichen. Falls beispielsweise BFS b_1 den Knoten x von Knoten n aus erreicht und BFS b_2 den Knoten x von m aus, würden ohne Anpassungen zwei Einträge in die Warteschlange eingefügt werden. Um die BFS zu bündeln ist es aber erforderlich einen einzigen Warteschlangeneintrag $\{b_1, b_2\}$ für den Knoten x in der Warteschlange zu haben. Die effizienteste Art dies zu implementieren ist es statt einer klassischen Warteschlange, in die Einträge eingefügt, werden, eine Liste zu nutzen, die für jeden Knoten einen festen Eintrag hat. Um einen Knoten für eine BFS nun für die nächste Runde einzureihen, wird beim entsprechenden Eintrag des Knoten in der Liste das Bit für diese Breitensuche gesetzt. Somit existiert nach einer Iteration immer genau ein Eintrag für jeden Knoten, der angibt welche Breitensuchen diesen Knoten in der aktuellen Iteration entdeckt haben und somit für welche in der nächsten Iteration von diesem Knoten aus weiter gesucht werden muss. Im Unterschied zu der ursprünglichen Warteschlange ist kein FIFO-Verhalten mehr garantiert, dies wird aber für die Korrektheit des Algorithmus nicht benötigt. Es muss nur gesichert sein, dass die Warteschlangen zwischen Iterationen getrennt sind. Der algorithmische Nachteil dieser Variante ist, dass in einer Iteration immer alle Einträge der Knoten des Graphen in der Liste überprüft werden müssen, um die zu finden, die bearbeitet werden müssen. In der Praxis ist diese sequenzielle Suche aber sehr schnell; sie ist neben den Kosten der anderen Operation des Algorithmus vernachlässigbar.

Die angepassten Datenstrukturen erfordern auch, dass die Operationen auf ihnen angepasst werden. Die Ausführung des Algorithmus geschieht nicht mehr als individuelle Binäroperationen, sondern diese werden in die entsprechenden Mengenoperationen umgesetzt. Die Überprüfung zum Beispiel, ob ein Knoten bereits gesehen wurde, war bisher einfach eine Überprüfung ob der entsprechende Eintrag in der $seen$ Liste den Wert $true$ hat. Mit unserem neuen Ansatz muss dies nun äquivalent für eine Menge an Breitensuchen gemacht werden. Wichtig dabei ist nur die Breitensuchen zu berücksichtigen, die an dem jeweiligen Knoten auch aktiv sind. Um nun herauszufinden, welche Breitensuchen den Knoten neu entdecken, muss die Differenz zwischen der Menge der aktiven Knoten und der Menge des $seen$ Eintrags des Knoten bestimmt werden. Die verbleibende Menge an Breitensuchen hat dann diesen Knoten zum ersten Mal gesehen und es müssen entsprechend die $seen$ Einträge aktualisiert (Zeile 16) und die $visitNext$ Liste für die nächste

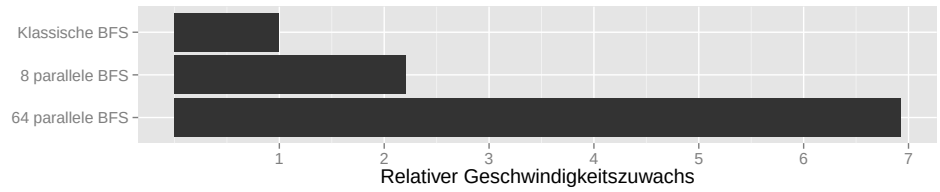

Abbildung 7: Laufzeitverbesserung mit 8 beziehungsweise 64 gebündelten Breitensuchen

Iteration im diese BFS erweitert werden (Zeile 15).

Diese Mengenoperationen wie Differenz, Vereinigung, etc. können, da die Mengen durch Bitsets repräsentiert werden, effizient mit or-, and- und $andnot$-Instruktionen umgesetzt werden. Um die Instruktionen nutzen zu können, dürfen die Bitsets aber nicht größer als die Register der CPU sein. Das Wettbewerbssystem unterstützte zwar 128 bit breite SSE Register, aber unterstützt auf diesen nur einen zu sehr eingeschränkten Befehlssatz. Aus diesem Grund ist die Bitsetgröße auf 64 bit und somit 64 parallele Breitensuchen beschränkt. Durch die Bündelung von maximal vielen Breitensuchen entstehen zwei Vorteile: (i) die Anzahl der Instruktionen für die bitparallele BFS ist unabhängig von der Registergröße und somit werden die Kosten bei breiteren Registern besser verteilt und (ii) mit der Anzahl der Breitensuchen steigt auch die Wahrscheinlichkeit, dass mehrere an einem Knoten überlappen. Gerade in den Iterationen in denen viele Knoten entdeckt werden, können so noch mehr Breitensuchen gebündelt werden.

Um die Effektivität der Bündelung zu prüfen, haben wir bei der 64-fach Daten-Parallelen Breitensuchen gemessen auf wie viele Knoten pro Iteration gebündelt zugegriffen wird. Die Messungen sind dabei unterteilt nach Knoten, die von mindestens 2, 20, 40 oder 60 gebündelten Breitensuchen besucht wurden. Wie in Abbildung 6 zu sehen ist, können gerade in den Iterationen, in denen auf viele Knoten zugegriffen wird, die Breitensuchen oft stark gebündelt werden. Abbildung 7 zeigt dementsprechend auch, dass wir mit der Bündelung gegenüber der klassischen Variante die Laufzeit um über Faktor 6 reduzieren konnten. Als weiteren Vergleich haben wir die Laufzeit zu Berechnung der Distanzsumme für jeden Knoten mit dieser Breitensuchen mit der Laufzeit des in Neo4j integrierten Algorithmus verglichen. Unsere Laufzeit war um über Faktor 100 schneller. Nach dem Wettbewerb haben wir diesen Ansatz stark weiter verfeinert und auch auf breiteren Registern getestet. Diese Ergebnisse wurden in einem separaten Artikel veröffentlicht [TKC+14].

6 Heuristiken für Top-k-Anfragen

Neben der Beschleunigung der Breitensuche gibt es für die *Top-k*-Anfragen noch die Möglichkeit den Suchraum mit Hilfe von oberen und unteren Schranken, die mit dem bisherigen besten Ergebnis verglichen werden können, zu verkleinern. Dies hilft Breitensuchen zu sparen oder früher abzubrechen.

Der Benchmark enthält mit Anfragetypen 2–4 einige solcher Anfragen. Für diese haben

Interessen	Metadaten		Ergebnis
	jüng. Geb.	#Pers.	
Vorl. Top-1			
Fußball	02.06.1998	2.600	2.460
Kandidaten			
Bratwurst	06.12.1999	2.500	~~2.430~~
Urlaub	08.07.1996	~~1.400~~	—
Auto	~~13.11.1992~~	~~1.350~~	—

Tabelle 2: Beispiel der Heuristik für Anfragetyp 2 mit den Parametern $k = 1$ und $b = 01.01.1993$. Interessen mit durchgestrichenen Werten werden eliminiert.

wir verschiedene Heuristiken entwickelt, die günstig obere und untere Schranken liefern. Im Folgenden stellen wir die Heuristiken für jeden Anfragetyp vor.

In Anfragen vom Typ 2 werden möglichst große Interessengruppen mit Personen unterhalb einer Altersgrenze b gesucht. Für jedes Interesse bildet die Gesamtzahl der Personen, die sich dafür begeistern, eine obere Schranke für die Größe von dessen maximaler Interessengruppe. Gleichzeitig sind die Interessen unterschiedlich über die verschiedenen Altersgruppen verteilt. Falls das Alter der jüngsten Person, die ein Interesse hat, höher ist als von einer Anfrage gefordert, kann für dieses sofort die Maximalgröße 0 bestimmt werden. Beide Metriken, die Anzahl der Personen und die jüngste Person pro Interesse, lassen sich günstig mit einem Scan aller Personen ermitteln. Tabelle 2 zeigt anhand eines Beispiels das Potenzial dieses Verfahrens. Es zeigt die aktuellen Zwischenausführung während der Ausführung einer *Top-1* Anfrage die sich auf Personen die nach dem $01.01.1993$ geboren wurden beschränkt. Beim Interesse *Bratwurst* helfen die Schranken noch nicht die exakte Berechnung zu vermeiden. Die 2.500 Personen, die dieses Interesse teilen reichen theoretisch aus um die bisher größte Gruppe von 2.460 zu übertreffen. Auch mit Hilfe der Geburtstagsinformation kann keine ausreichende Schranke ermittelt werden. Erst nach Berechnung des exakten Wertes von 2.430 wird ersichtlich, dass es dieses Interesse nicht Teil der Top-1 ist. Für die anderen Interessen hingegen genügt jeweils eine Überprüfung der vorberechneten Metadaten, um eine exakte Berechnung zu vermeiden. Somit kann jeweils eine Breitensuche eingespart werden. Um möglichst früh eine starke Schranke zu erhalten, werten wir die Interessen in der Reihenfolge gemäß der Anzahl der Personen aus Abbildung 8 zeigt die Wirksamkeit dieser Heuristiken anhand von Messungen auf dem 100.000-Personen-Netzwerk. Gemessen wurde jeweils die Gesamtlaufzeit inklusive dem Laden der benötigten Dateien. Die Ausführung erfolgte nur auf einem CPU-Kern. Allein mit der Heuristik basierend auf der Anzahl der Personen kann die Laufzeit auf unter fünf Sekunden reduziert werden. Kombiniert mit Schranken basierend auf der jüngsten Person für jedes Interesse sinkt die Laufzeit auf $2.059\,\text{ms}$. Davon werden $9\,\text{ms}$ zur Generierung der Metadaten für die Heuristiken benötigt. Die Ermittlung der exakten Werte für die Interessen, die nicht übersprungen werden konnten, dauerte $1.778\,\text{ms}$. Der Rest der Laufzeit wurde zum Laden der Eingabedaten benötigt.

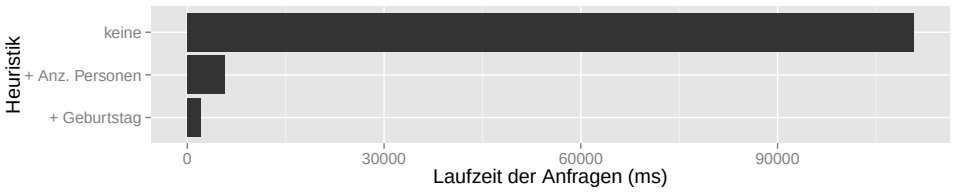

Abbildung 8: Gesamtlaufzeit der Auswertung von zehn Anfragen vom zweiten Typ bei gestaffelter Aktivierung der Heuristiken. Jede Variante berechnet jeweils nur die minimal benötigten Metadaten.

Anfrage 3 beschäftigt sich mit Personenpaaren, die sich an einem vorgegebenen Ort aufhalten, sich über eine begrenzte Anzahl von Freunden kennen und die meisten Interessen miteinander teilen. Bei dieser Anfrage gibt es zwei teure Schritte. Zum einen müssen alle Freundschaftspaare bestimmt werden, die diese Kriterien erfüllen und zum Anderen muss für jedes Personenpaar überprüft werden, wie viele gemeinsame Interessen sie haben. Für diesen zweiten Schritt kann eine simple, aber effektive Heuristik genutzt werden: Für jedes Personenpaar bildet die Anzahl der Interessen einer der Personen eine obere Schranke für die Anzahl der gemeinsamen Interessen. Diese Schranke kann während der Auswertung mit den bisherigen besten Ergebnissen zur Anzahl der gemeinsamen Interessen verglichen werden. Ist in diesem Vergleich die Schranke kleiner, ist es nicht nötig die genaue Größe der Schnittmenge zu bestimmen, da dieses Paar nicht Teil der Top-k-Ergebnisse sein kann. Um früh ein möglichst guten Top-k-Grenzwert zu erhalten und so möglichst viele andere Paare überspringen zu können werten wir die Personenpaare absteigend geordnet nach der Anzahl ihrer Interessen aus.

Das meiste Potenzial zur Reduzierung der Gesamtlaufzeit steckt jedoch im teuren Anfragetyp 4. Zur Berechnung der Zentralität muss von jeder Person die Distanz zu allen anderen erreichbaren Personen berechnet werden. Mit einer Heuristik, die eine verlässliche günstige untere Schranke auf die Summe der Distanzen zu allen anderen Personen liefert, ließen sich potenziell durch den Vergleich mit den aktuellen Top-k-Werten viele Distanzberechnungen vermeiden. In der Literatur existieren probabilistische Ansätze die Summe günstig mittels Hash-Sketches abzuschätzen [PKST11]. Diese Verfahren können aber keine Worst-Case-Garantien für die Fehler der Ergebnisse liefern. Somit können sie auch nicht zur Berechnung von Schranken genutzt werden, da sonst eventuell Personen fälschlicherweise übersprungen werden und das Ergebnis verfälscht wird. Wir adaptieren den Ansatz und entwickeln eine neue Abschätzung, die im Schnitt zwar schlechter ist, dafür aber verlässlichere untere Schranken für die Distanzsumme liefert, welche zur Berechnung der Closeness Centrality benötigt wird.

Als Basis unseres Verfahrens analysieren wir zunächst die Struktur des probabilistischen Distanzschätzungsalgorithmus [PKST11]. Die Grundintuition ist, dass in einem ungerichteten Graph die Mengen der Knoten, die von einem Knoten aus mit x Schritten erreicht werden können, der Menge der Knoten entspricht, die alle seine Nachbarn mit $x-1$ Schritten erreichen. Vom Nachbar zum Knoten ist es schließlich genau ein Schritt weiter. Dies nutzt der Algorithmus aus, indem pro Knoten jeweils die Vereinigung der von allen Nach-

barn in der letzten Runde erreichten Knoten gebildet wird. Der Unterschied in der Größe zwischen den Mengen eines Knotens in der Iteration $x-1$ von der in Iteration x gibt die Anzahl der Personen an, die mit genau x Schritten erreicht werden können. Um die Distanzsumme pro Knoten zu ermitteln, wird die Menge der erreichbaren Knoten vor der ersten Iteration jeweils nur mit sich selbst gefüllt. In Iteration 1 wird dann für jeden Knoten die Vereinigung der initialen Mengen aller seiner Nachbarn gebildet und die Größe dieser Menge dann mit der der intitalen Menge verglichen. Dies ergibt die Knoten, die genau einen Schritt entfernt sind. In Iteration 2 dann für jeden Knoten die Vereinigung aller von den Nachbarn in Iteration 1 erreichten Knoten, etc. Indem man diese Distanzen aufsummiert bis in einer Iteration keiner der Knoten mehr eine neue Person erreicht hat, kann jeweils die Distanzsumme für jeden Knoten ermittelt werden. Im Kontrast zu einer einfachen Breitensuche wäre es aber prohibitiv teuer die Vereinigungen der Nachbarmengen exakt zu berechnen, da in den letzten Iterationen für jeden Knoten die Zahl der Elemente seiner Nachbarknotenmenge in der Größenordnung der Zahl von Knoten im Graphen ist. Um die damit verbundenen hohen Speicher- und Rechenkosten zu vermeiden, werden approximative Datenstrukturen, die den Speicherbedarf reduzieren genutzt. Die Anforderungen dabei sind, dass sie die Vereinigungs-Operation günstig unterstützen müssen und die Größe der Menge abgeschätzt werden kann. Dieses zweite Kriterium wird z.B. von Bloom-Filtern [Blo70] nicht unterstützt. Stattdessen können Strukturen wie *FM-Sketches* [FNM85] oder *HyperLogLog* [HNH13] genutzt werden. Da diese aber nur probabilistische Werte für die Größe der Mengen bestimmen können, sind sie nicht zur Berechnung exakter Ergebnisse geeignet.

Die Basis unseres Ansatzes ist es die Schätzung der Menge der über die Nachbarn erreichten Knoten noch weiter zu vereinfachen, indem wir o.B.d.A. annehmen, dass diese Mengen unabhängig sind. Wenn dieses unabhängig sind, können einfach ihre einzelnen Größen zusammenaddiert werden, um die Größe der Vereinigung der Mengen zu berechnen. Somit werden als einzige Information immer nur die Größe der Menge benötigt. Auf das Verfahren angewandt bedeutet dies, dass statt der probabilistischen Mengen dann nur auch jeweils die Größe gespeichert wird. Die Distanzschätzung, die mit dieser Anpassung berechnet wird, ist eine strikte Obergrenze, auch wenn die Unabhängigkeitsannahme verletzt ist. Wenn sich die Mengen der Nachbarn überlappen würden, wäre die Anzahl der erreichten Knoten pro Iteration geringer, damit wäre die Anzahl der Knoten, die mit geringer Distanz erreicht werden auch kleiner und somit die Distanzsumme größer als unsere Schätzung. In einem Iterationsschritt werden die Summe der Schätzungen der Nachbarn mit der Schätzung für den eigenen Knoten der letzten Iteration verglichen und die Differenz ist die Anzahl der neu erreichten Knoten. Die Abschätzung der Distanzsumme kann auch zur Wahl der Auswertungsreihenfolge der Knoten genutzt werden. Die Hoffnung dabei ist, dass so durch gute vorläufige Top-k Werte möglichst früh viele Breitensuchen mit Hilfe der Schätzung übersprungen werden können.

Als letzte Optimierung um diese Heuristik zu verbessern, haben wir die Information der Breitensuchen der Knoten, die nicht übersprungen werden können, genutzt, um die Schätzungen weiter zu verbessern. Dazu speichern wir bei der Berechnung der Distanzschätzung für jeden Knoten, wie viele Knoten jeweils mit x Iteration geschätzt erreicht werden können. Falls für einen Knoten nun eine Breitensuche komplett ausgeführt wird, können

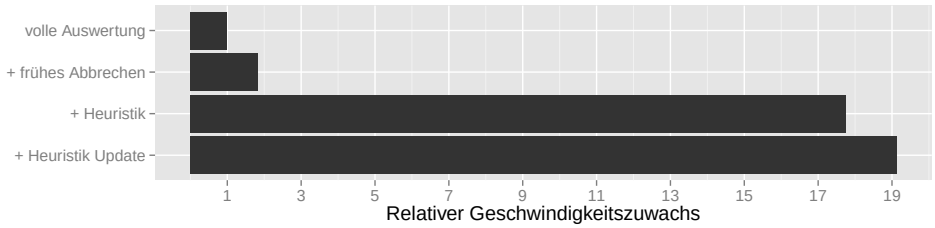

Abbildung 9: Relative Geschwindigkeitszuwächse durch Heuristiken bei Anfragetyp 4

diese Schätzungen dann durch die echten Werte ersetzt werden. Bei der Neuberechnung der Schätzung für einen seiner Nachbarn können nun diese exakten Werte genutzt werden, um die Schätzung zu verbessern.

Zusammengefasst berechnen wir für jeden Knoten zuerst eine Schätzung, wie viele Knoten jeweils mit x Schritten erreicht werden können, speichern diese und berechnen eine erste Schätzung für die Distanzsumme. Danach beginnt die Auswertung der Knoten. Bevor ein Knoten ausgewertet wird, wird überprüft ob er mit Hilfe der ursprünglichen Schätzung übersprungen werden kann. Ist dies nicht der Fall wird die Schätzung neu berechnet mit der Hoffnung, dass für einige seiner Nachbarn die genauen Werte der mit x Schritten erreichbaren Knoten statt der Schätzungen zur Verfügung stehen und somit eine bessere Schätzung erreicht werden kann. Falls mit dieser neuen Schätzung der Knoten auch nicht übersprungen werden kann, muss die Breitensuche gestartet werden. Die Laufzeitverbesserungen dieser aufeinander aufbauenden Verbesserungen haben wir gemessen und in Abbildung 9 dargestellt. Für die Wettbewerbsanfragen vom Typ 4 konnten wir auf dem 100.000-Personen-Datensatz somit eine Verbesserung der Laufzeit von Faktor 19 erreichen.

7 Beschleunigtes Laden von Rohdaten

Auf Grundlage der in [MRS+13] beschriebenen Techniken um das Laden von textbasierten Rohdaten in relationalen Hauptspeicher-Datenbanksystemen zu beschleunigen haben wir neue Methoden entwickelt um Graphdaten effizient in unser Analysesystem zu laden. Die LDBC-Daten des sozialen Graphen lagen beim Wettbewerb als CSV Dateien vor. Dies ist auch in realen Anwendungsszenarien häufig der Fall, denn CSV ist weiterhin ein beliebtes Speicherformat. Die Vorteile von CSV Dateien liegen in ihrer Lesbarkeit, da alle Daten textuell gespeichert sind und daher in einem Texteditor gelesen und verändert werden können. Des Weiteren sind CSV Dateien, anders als proprietäre Binärformate, sehr portabel. Allerdings können textuell gespeicherte Daten nicht effizient analysiert werden. Die Zeilen einer CSV Datei müssen zunächst entlang des Feldtrennzeichens, häufig ",", aufgespalten werden (*Parsing*). Des Weiteren müssen für manche Datentypen die textuellen Daten noch in ein maschinenverständliches Format, z.B. Integer, umgewandelt werden (*Deserialization*). Beides, Parsing und Deserialization sind auf modernen Prozessoren

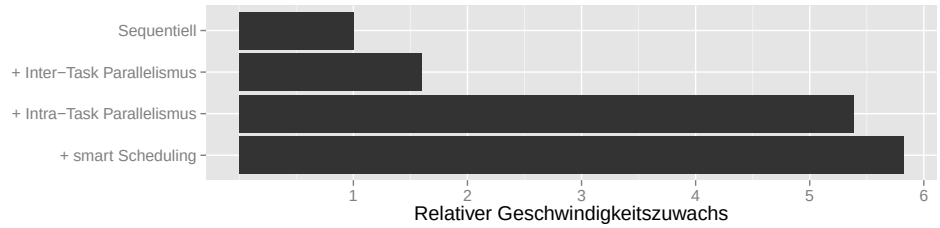

Abbildung 10: Geschwindigkeitszuwachs bei der Auswertung der Benchmarkanfragen beim Hinzufügen der verschiedenen Techniken

sehr teuer. Naive Implementierungen vergleichen jedes Byte mit dem Feld- und Zeilen-Trennzeichen, was zu einem unregelmäßigen Kontrollfluss und daher zu vielen *Branch-Misses*[2] führt. Daten-paralleles Parsing wie in [MRS+13] beschrieben verarbeitet mehrere Vergleiche in einer Instruktion. Mit der Intel SSE 2 Instruktion _mm_cmpeq_epi8 können etwa 16 Byte mit einem Vergleichs-Byte in einer Instruktion verglichen werden. Die resultierende Bitmaske liefert dann die Stellen zurück, an denen das Vergleichs-Byte gefunden wurde. Dies erzeugt deutlich weniger Branch-Misses und führt daher zu einer höheren Performanz. Für das Wettbewerbssystem konnten wir so die Ladegeschwindigkeit um über 60% gegenüber naivem Parsing und Deserialization mit Methoden aus der Standardbibliothek (z.B. atol) steigern.

Ebenfalls, wie in [MRS+13] beschrieben, wird das Laden einer einzelnen Datei in mehrere Tasks aufgespalten, welche dann von mehreren Prozessorkernen parallel bearbeitet werden. Je nach Geschwindigkeit der Datenquelle kann so die Geschwindigkeit des Ladens mit der Anzahl der verwendeten Prozessorkerne skaliert werden. Jeder Task bearbeitet dabei den ihm zugewiesenen Bereich der Datei (Parsing und Deserialization) und speichert die von den Anfragen benötigten Daten in geeigneten Datenstrukturen. Nicht benötigte Daten werden frühestmöglich verworfen.

8 Task-Scheduling

Die bisherigen Kapitel waren jeweils auf einzelne Aspekte und Aufgaben fokussiert, in diesem liegt der Fokus auf die Integration aller Teilaspekte in ein System. Alle Bausteine sollen möglichst effizient verknüpft werden um die Kerne des Systems möglichst gut auszulasten und so eine möglichst kurze Gesamtlaufzeit des Programms zu erreichen.

Bei der Parallelisierung des Systems sind verschiedene Varianten möglich. Ein klassischer Ansatz ist die *Inter-Query Parallelisierung*. Hier werden die einzelnen Aufgaben parallel zueinander ausgeführt. Bei Abhängigkeiten wäre eine einfache Lösung, die Ausführung

[2]Moderne Prozessoren müssen eine Vielzahl an zukünftig ausgeführten Instruktionen kennen um optimal ausgelastet zu sein (*pipelining*). Dazu wird der Pfad, den der Kontrollfluss bei einer Verzweigung nimmt, vom Prozessor erraten. Dies basiert auf einer Heuristik und ist daher nicht immer richtig, was dann zu einer teuren Invalidierung der CPU-Pipeline führt (*Branch-Miss*).

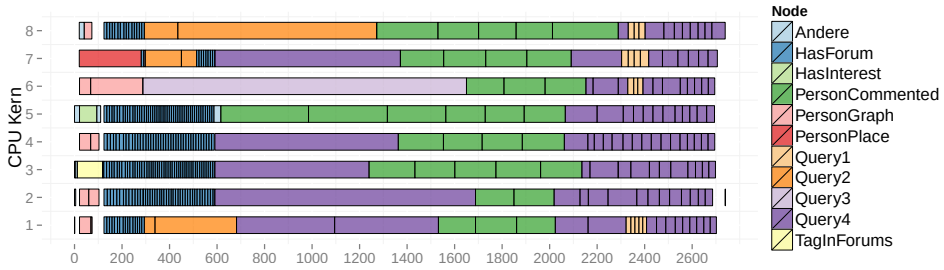

Abbildung 11: Ausführung der verschieden Arbeitspakete auf den CPU Kernen während der Programmlaufzeit.

der abhängigen Aufgaben durch eine globale Barriere von den vorhergehenden Anfragen zu trennen. Zur Verteilung der Aufgaben auf die Kerne können diese in eine zentrale synchronisierte Warteschlange eingereiht werden, Arbeiter die auf jedem der Kerne laufen überwachen diese Warteschlange und versuchen aus dieser immer Arbeit zu entnehmen, diese zu bearbeiten und dann das nächste Arbeitspaket zu holen. Durch die Synchronisierung der Warteschlange wird sichergestellt, dass eine Aufgabe jeweils immer nur einem Arbeiter zugewiesen wird. Durch diese dynamische Verteilung der Aufgaben ist sichergestellt, dass bis zur Verteilung der letzten Aufgabe aus der Warteschlange alle Kerne voll ausgelastet sind.

Zusätzlich zur parallelen Ausführung der verschiedenen Komponenten unseres Systems, haben wir die teueren Komponenten, wie zum Beispiel das Lesen der Dateien und die Ausführung von Anfragen vom Typ 4, zudem noch in sich parallelisiert. Diese *Intra-Query Parallelisierung* sorgt dafür, dass zum einen mehr Arbeitspakete zur Verfügung stehen und somit mehr Kerne genutzt werden können und aber auch dafür, dass es kein einzelnes großes Aufgabenpaket gibt, welches am Ende dann nur auf einem Kern ausgeführt wird, während die anderen Kerne warten müssen.

Mit diesen beiden Parallelisierungsvarianten kann bereits eine gute Auslastung der CPU-Kerne erreicht werden. An den globalen Barrieren gibt es jedoch einen Punkt, an dem die meisten Kerne warten müssen, bis auch der letzte Kern mit seiner Arbeit fertig ist und die Aufgaben hinter der Barriere begonnen werden können. Dies führt zu einem Verlust potenzieller Rechenleistung. Um dies zu vermeiden, haben wir die globalen Barriere entfernt und stattdessen die genauen Abhängigkeiten zwischen den verschiedenen Komponenten in einem Abhängigkeitsgraph modelliert. Für jede Komponente werden die Arbeitspakete erst an den Scheduler übergeben, sobald alle Aufgaben von vorhergehenden Komponenten abgearbeitet sind. Durch dieses Prinzip sind die Arbeiter bei der Bearbeitung der Arbeitspakete nie durch das Warten auf andere Arbeitspakete blockiert. Als weitere Optimierung unterstützt unser Scheduler die Priorisierung von Arbeitspaketen. Mit dieser Priorisierung ordnen wir die Ausführungsreihenfolge der zur Verfügung stehen Arbeitspakete so, dass Abhängigkeiten von Komponenten die viele Arbeitspakete generieren zuerst ausgeführt werden. Durch diese Priorisierung erreichen wir, dass schon früh möglichst viele Arbeitspakete zur Verfügung stehen, so dass kein Arbeiter dadurch blockiert ist, dass er keine Arbeit findet. Die Prioritäten wurden von uns anhand von Experimenten festgelegt.

Die Effektivität unserer Parallelisierung haben wir getestet, indem wir die Laufzeit unter Nutzung der verschiedenen Parallelisierungsmodelle für die Ausführung aller Wettbewerbsanfragen auf dem 100.000-Personen-Netzwerk gemessen haben. In Abbildung 10 ist sichtbar, wie die Gesamtlaufzeit durch zunehmend feinere Parallelisierung reduziert werden konnte. Abbildung 11 zeigt die zeitliche Ausführung der Komponenten auf den Kernen (markiert durch Farben). Dank der feingranularen Ausführung gibt es kaum Phasen in denen ein Kern nicht ausgelastet ist.

9 Zusammenfassung und Ausblick

In diesem Artikel wurde das Gewinnersystem des ACM SIGMOD Programming Contest 2014 beschrieben. Die Aufgabe des Programmierwettbewerbs bestand darin, ein hochperformantes Analysesystem für soziale Graph-Daten zu entwickeln. Nur durch die Kombination von algorithmischen Verbesserungen wie unserer Daten-parallelisierten Graph-Breitensuche, eigenen Heuristiken zur starken Einschränkung des Suchraums bei Top-k-Anfragen, feingranularem Task-Scheduling, und effizientem parallelisierten Laden von textuellen Rohdaten konnten wir das Mehrkern-System optimal ausnutzen und den Wettbewerb mit über 30% schnellerer Laufzeit als der Zweitplatzierte gewinnen.

Die präsentierten Techniken wurden bisher in einem eigenständigen System implementiert, können aber in bestehende Graph-Datenbanksysteme integriert werden. Wir arbeiten des Weiteren daran die Techniken in unser Hauptspeicher-Datenbanksystem HyPer [KN11] zu integrieren um Deep Analytics auf großen Graph-Daten effizient zu unterstützen.

Literatur

[APPB10] Virat Agarwal, Fabrizio Petrini, Davide Pasetto und David A. Bader. Scalable Graph Exploration on Multicore Processors. In *SC*, Seiten 1–11, 2010.

[BAP12] Scott Beamer, Krste Asanović und David Patterson. Direction-Optimizing Breadth-First Search. In *SC*, Seiten 12:1–12:10, 2012.

[BFG+13] Peter A. Boncz, Irini Fundulaki, Andrey Gubichev, Josep-Lluis Larriba-Pey und Thomas Neumann. The Linked Data Benchmark Council Project. *Datenbank-Spektrum*, 13(2):121–129, 2013.

[Blo70] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *CACM*, 13(7):422–426, Juli 1970.

[BM11] Aydin Buluç und Kamesh Madduri. Parallel Breadth-First Search on Distributed Memory Systems. In *SC*, Seiten 65:1–65:12, 2011.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest und Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd. Auflage, 2001.

[ES11] Stefan Edelkamp und Stefan Schroedl. *Heuristic search: theory and applications*. Elsevier, 2011.

[Flo14] FlockDB, 2014. https://github.com/twitter/flockdb.

[FNM85] Philippe Flajolet und G. Nigel Martin. Probabilistic Counting Algorithms for Data Base Applications. *JCSS*, 31(2):182–209, 1985.

[HNH13] Stefan Heule, Marc Nunkesser und Alex Hall. HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm. In *EDBT*, 2013.

[Ior10] Borislav Iordanov. HyperGraphDB: A Generalized Graph Database. In *WAIM*, Seiten 25–36, 2010.

[JRW+14] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande und Mike Stonebraker. VERTEXICA: Your Relational Friend for Graph Analytics! *PVLDB*, 7(13):1669–1672, 2014.

[KN11] A. Kemper und T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, Seiten 195–206, 2011.

[LBK14] Daniel Lemire, Leonid Boytsov und Nathan Kurz. SIMD Compression and the Intersection of Sorted Integers. *CoRR*, abs/1401.6399, 2014.

[LDB14] LDBC SNB: Generated CSV Files, 2014. https://github.com/ldbc/ldbc_snb_datagen/wiki/Generated-CSV-Files.

[LGHB07] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson und Jonathan Berry. Challenges in Parallel Graph Processing. *PPL*, 17(1):5–20, 2007.

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser und Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *SIGMOD '10*, Seiten 135–146, 2010.

[MBMMGV+07] Norbert Martínez-Bazan, Victor Muntés-Mulero, Sergio Gómez-Villamor, Jordi Nin, Mario-A. Sánchez-Martínez und Josep-L. Larriba-Pey. Dex: High-performance Exploration on Large Graphs for Information Retrieval. In *CIKM*, Seiten 573–582, 2007.

[MRS+13] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper und Thomas Neumann. Instant Loading for Main Memory Databases. *PVLDB*, 6(14):1702–1713, 2013.

[Neo14] Neo4j, 2014. http://neo4j.com/.

[Pcs14] SIGMOD 2014 Programming Contest Leaderboard, September 2014. http://www.cs.albany.edu/~sigmod14contest/.

[PKST11] Spiros Papadimitriou, U Kang, Jimeng Sun und Hanghang Tong. Centralities in Large Networks: Algorithms and Observations. In *SDM '11*, Seiten 119–130, 2011.

[SWL13] Bin Shao, Haixun Wang und Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *SIGMOD*, Seiten 505–516, 2013.

[TKC+14] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann und Huy T. Vo. The More the Merrier: Efficient Multi-Source Graph Traversal. *PVLDB*, 8(4):449–460, 2014.

[Tro05] Vadim Tropashko. Nested intervals tree encoding in SQL. *ACM SIGMOD Record*, 34(2):47–52, 2005.