Optimierung einer Graphdatenbank für Big Data

Lars Bengel
Universität Koblenz-Landau
larsbengel@uni-koblenz.de

28. November 2017

1 Motivation

In den letzten Jahren werden Informationen immer mehr in Graphen gespeichert, wie zum Beispiel der Google Knowledge Graph[1], der mehrere Milliarden Fakten enthält und damit die Suchergebnisse verbessert und durch Zusatzinformationen erweitert. Ein Graph, wie er beispielsweise in Abbildung 1 zu sehen ist, besteht aus mehreren Knoten, die mit Kanten untereinander verbunden sind. Die Bezeichner der Knoten und Kanten können dabei Informationen, wie Namen oder Relationen, enthalten.

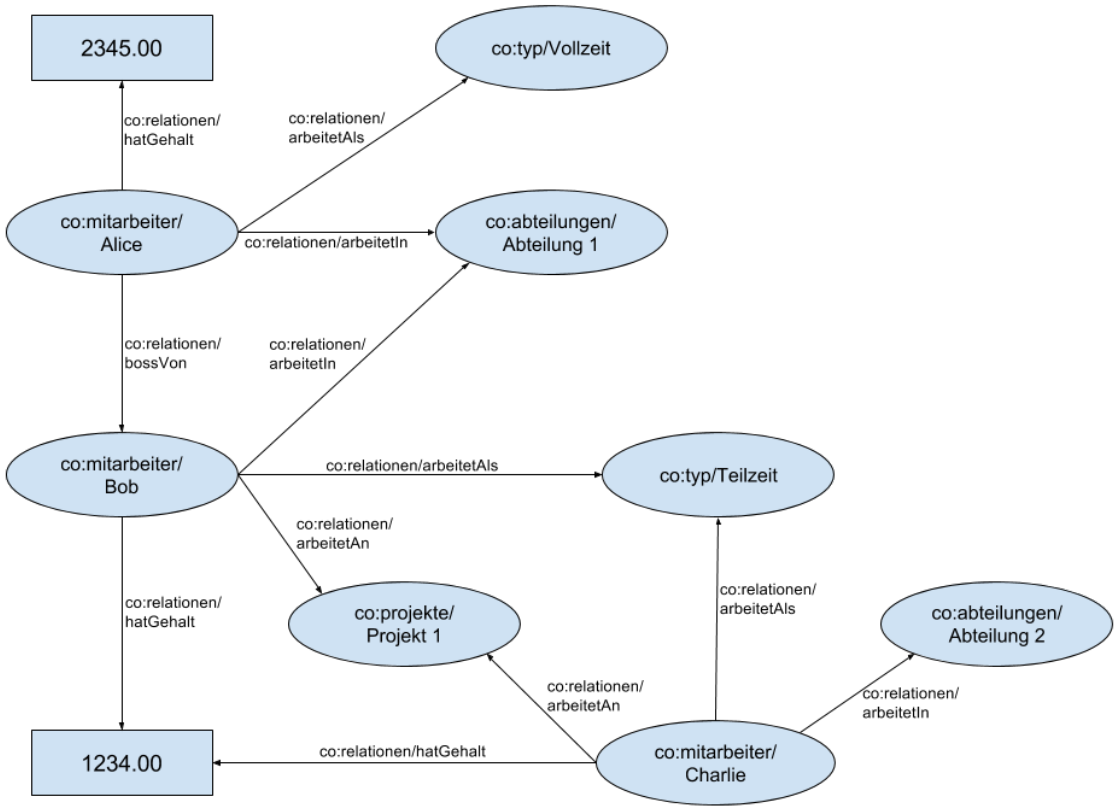

Abbildung 1: Beispiel für einen Graphen.

Beispiel 1. *Der abgebildete Graph stellt eine fiktionale Firma* `company` *und deren Angestellte und Abteilungen dar. Der Graph besteht aus 10 Knoten und 12 Kanten. Das Präfix* `http://www.company.com/` *für jeden Bezeichner wurde aus Übersichtsgründen in der Abbildung 1 mit* `co` *abgekürzt.*

[1] https://www.google.com/intl/bn/insidesearch/features/search/knowledge.html zugegriffen am 28. November 2017

Im Bereich des Semantic Web hat sich das RDF-Format (*Resource Description Framework*) [2] zur Repräsentation von Graphen durchgesetzt. Diese Graphen bestehen nur aus sogenannten RDF-Tripeln. Eine Ressource kann dabei ein beliebiges Objekt sein, das über eine IRI (*Internationalized Resource Identifier*) identifiziert wird. Ein RDF-Tripel besteht dabei aus 3 Teilen: Subjekt, Prädikat und Objekt. Subjekt und Objekt stellen die Bezeichner von Knoten dar, während das Prädikat den Bezeichner einer Kante repräsentiert. Das Objekt kann außerdem statt einer IRI auch ein Literal enthalten. Außerdem gibt es noch sogenannte Blank-Nodes, die eine anonyme Ressource ohne IRI darstellen und nur an Subjekt- oder Objektposition stehen können.

```
1 <http://www.company.com/mitarbeiter/Alice>
2 <http://www.company.com/relationen/bossVon>
3 <http://www.company.com/mitarbeiter/Bob>
4
5 <http://www.company.com/mitarbeiter/Alice>
6 <http://www.company.com/relationen/hatGehalt>
7 "1234.00"^^<http://www.w3.org/2001/XMLSchema#float>
```

Listing 1: Beispiel für RDF-Tripel.

Beispiel 2. *Dieses Beispiel zeigt in den Zeilen 1-3 die Beziehung* `Alice ist die Chefin von Bob` *modelliert als RDF-Tripel. Die IRI* `<http://www.company.com/mitarbeiter/Alice>` *in Zeile 1 steht dabei für die Mitarbeiterin Alice und ist das Subjekt. Die IRI* `<http://www.company.com/relationen/bossVon>` *in Zeile 2 steht für das Prädikat und die IRI* `<http://www.company.com/mitarbeiter/Bob>` *repräsentiert den Mitarbeiter Bob und ist das Objekt des dargestellten RDF-Tripels.*

Das zweite Tripel stellt die Beziehung `Alice erhält ein Gehalt von 1234,00$` *dar. Alice ist hier ebenfalls das Subjekt des Tripels. In diesem Beispiel ist das Prädikat die IRI* `<http://www.company.com/relationen/hatGehalt>` *in Zeile 6. Das Objekt dieses RDF-Tripels ist das Literal* `"1234.00"^^<http://www.w3.org/2001/XMLSchema#float>` *in Zeile 7. Dabei handelt es sich um ein Literal vom Typ Float, das dementsprechend als 1234,00 zu interpretieren ist.*

Eine Datenbank, die RDF-Graphen speichert, wird auch RDF-Store genannt. Die Graphen können sehr groß werden und aus Milliarden von Kanten bestehen [1]. Eine Möglichkeit um so große Graphen zu verarbeiten sind verteilte Datenbanksysteme, bei denen der Graph auf mehreren Computern verteilt wird. Dafür muss der Graph durch einen Partitionierungsalgorithmus anhand bestimmter Kriterien in kleinere Graphdatenblöcke unterteilt werden. Ein solcher Algorithmus ist das Minimaler-Kantenschnitt-Verfahren, welches die Anzahl der Kanten zwischen zwei Partitionen minimiert und gleichzeitig ähnlich große Partitionen erzeugt.

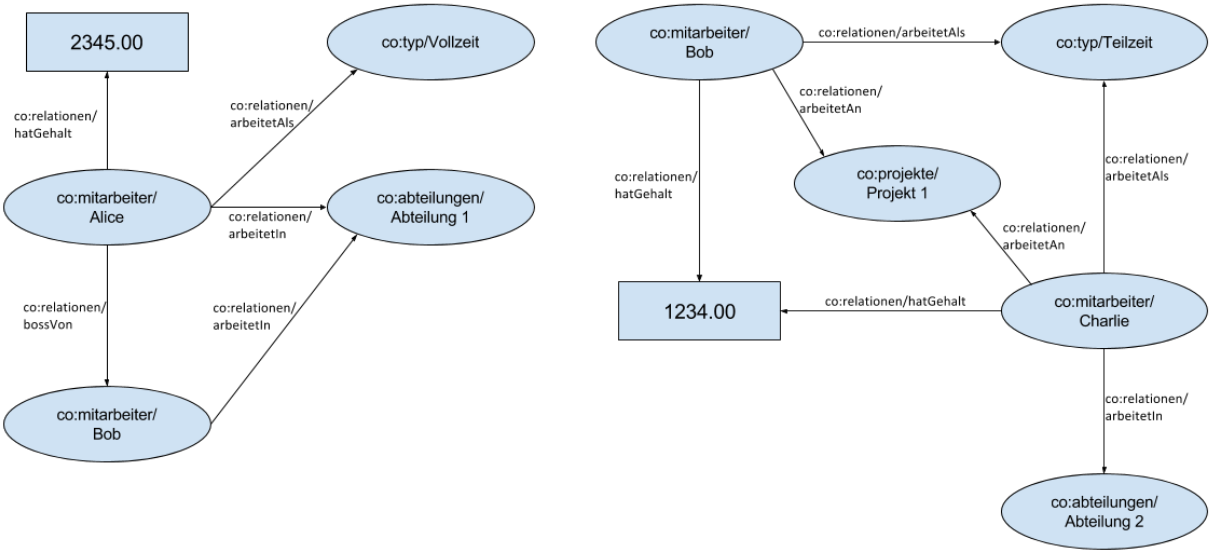

Abbildung 2: Mögliche Partitionierung des Beispielgraphen.

Beispiel 3. *Dargestellt ist eine mögliche Partitionierung des Beispielgraphen in 2 Graphpartitionen, die auf den verschiedenen Slaves gespeichert werden können. Der Knoten mit dem Bezeichner Bob kommt dabei in beiden Partitionen vor.*

Es gibt verschiedene Ansätze wie ein solches Datenbanksystem umgesetzt werden kann. Eine verbreitete Architektur ist das *Master-Slave-Sytem*. Diese Architektur beinhaltet einen Hauptrechner, den *Master*, sowie eine Menge anderer Rechner, die *Slaves*. Der *Master* verwaltet dabei Anfragen auf den Graphen und die *Slaves* speichern den ihnen zugeteilten Graphpartition und führen darauf Anfragen aus.

Koral[2] ist ein solcher verteilter RDF-Store, das gemäß der Master-Slave Architektur aufgebaut ist. In Abbildung 3 ist die Architektur von Koral dargestellt. Es besteht demnach aus einem Hauptrechner, dem *Master*, und n weiteren Rechnern, den *Slaves*. Für die Kommunikation untereinander besitzen alle Rechner einen Netzwerkmanager. Beim Laden des Graphen wird zuerst das Dictionary erzeugt, welches den textuellen Ressourcen numerische IDs zuweist. Im Graphen werden die Ressourcen durch diese IDs ersetzt, um den Speicherbedarf zu reduzieren. Anschließend wird eine Graphpartitionierung erzeugt. Das bedeutet, dass der Graph in eine Menge kleinerer Graphpartitionen zerlegt wird, ohne dass dabei Informationen verloren gehen. Danach wird die Statistikdatenbank angelegt und die Graphdatenpartitionen an die einzelnen *Slaves* gesendet, diese erstellen daraus ihre lokalen Tripelindizes.

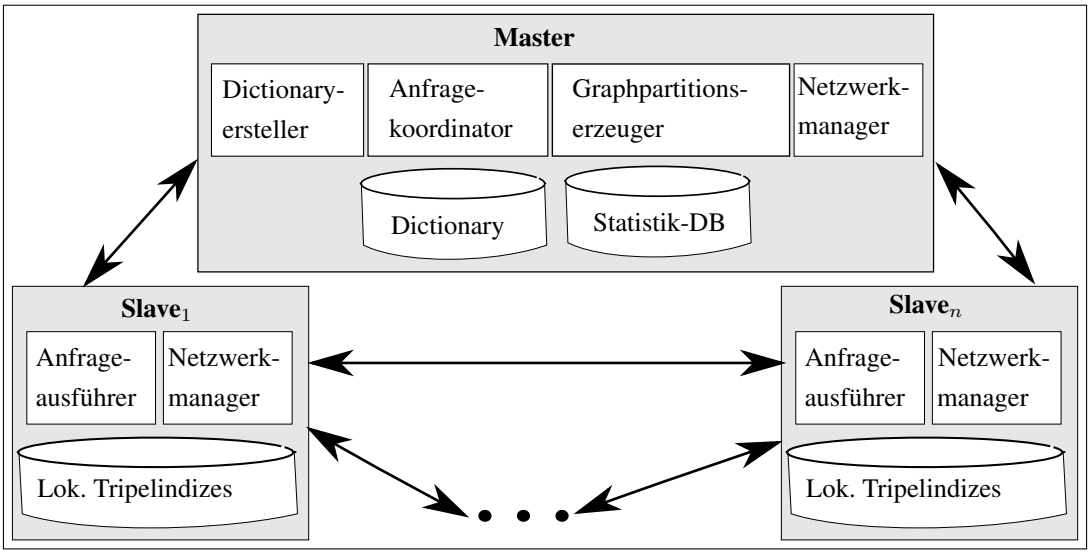

Abbildung 3: Architektur von Koral [3].

Eine Anfrage an den Graphen wird vom Anfragekoordinator verarbeitet und anschließend an die verschiedenen *Slaves* verteilt. Jeder *Slave* führt die Anfrage auf den lokalen Tripelindizes aus. Die Ergebnisse werden zurück an den Koordinator gesendet, der die IDs anhand des Dictionarys decodiert und das Resultat zurückgibt.

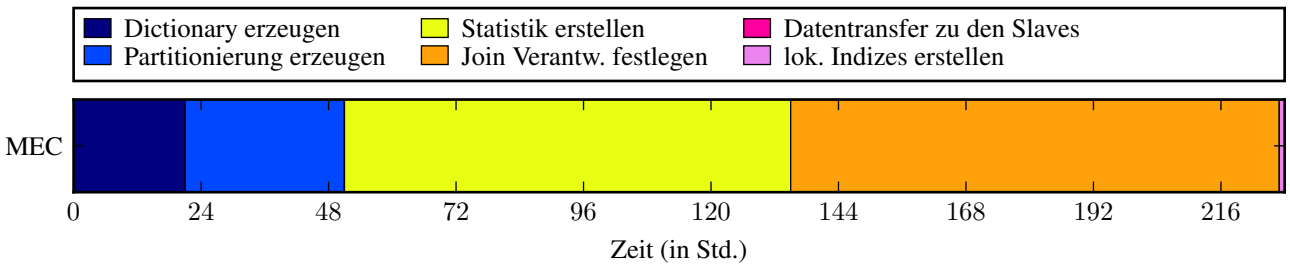

Abbildung 4: Dauer der einzelnen Ladeschritte.

In der aktuellen Implementierung wird die komplette Statistikdatenbank in einer einzelnen, sehr großen Datei auf dem Master gespeichert. Auf ihre Sektoren wird beim Lesen und Schreiben zufällig zugegriffen. Diese Implementierung ist äußerst ineffizient und führt dazu, dass die Schritte, bei denen mit der Statistikdatenbank gearbeitet wird, die längste Zeit in Anspruch nehmen. Abbildung 4 zeigt die Dauer der einzelnen Ladeschritte für einen RDF-Store mit 1 Milliarde Tripeln. Der Graph wurde mit dem Minimaler-Kantenschnitt Verfahren in 20 Graphpartitionen unterteilt. Wie man erkennen kann benötigt das Sammeln der statistischen Daten 84 Stunden. Beim Festlegen der Join Verantwortung, was weitere 93 Stunden benötigt, erfolgt ebenfalls lesender Zugriff auf die Statistikdatenbank. Dabei bekommt jede Ressource einen Slave zugeteilt, abhängig von der Häufigkeit des Auftretens als Subjekt. Dieser ist beim Kombinieren der Daten mehrerer

[2]https://github.com/Institute-Web-Science-and-Technologies/koral zugegriffen am 28. November 2017

Slaves bei der Anfrageausführung für die Ressource verantwortlich. Zusammen machen diese beiden Schritte ungefähr 77% des gesamten Ladevorgangs aus. Dieser Teil kann durch eine Optimierung der Statistikdatenbank beschleunigt werden.

Im Rahmen dieser Bachelorarbeit soll diese Implementation verbessert werden. Zum Beispiel durch die Verwendung von verschiedenen Kompressions- oder Cachingstrategien oder einem optimierten Dateilayout. Die gefundenen Optimierungsstrategien sollen anschließend gegen die aktuelle Implementation sowie eine relationale Datenbank wie SQLite[3] evaluiert werden.

2 Verwandte Arbeiten

In diesem Abschnitt werden einige Arbeiten vorgestellt, die sich mit verschiedenen Optimierungsstrategien auseinandersetzen, die für diese Bachelorarbeit interessant sein könnten. Dazu gehören Techniken zur Kompression oder Cachingstrategien.

Wang et. al. führen in [5] einen Vergleich zwischen verschiedenen Arten von Kompressionsverfahren durch. Es werden insgesamt 9 Bitmap Kompressionsstrategien und 12 invertierte Listen Techniken getestet und deren Performanz miteinander verglichen. Bei der Bitmap-Kompression werden aufeinanderfolgende Bits mit dem selben Wert durch Bitwert und Anzahl ersetzt. Bei der invertierte-Listen-Kompresssion werden die Abstände zwischen den Werten einer Liste bestimmt und nur diese gespeichert. Die Qualität der 21 Strategien wird anhand von verschiedenen realistischen wie auch synthetischen Datenmengen gemessen. Die Performanz der Algorithmen wird in den vier folgenden Kategorien bewertet:

- Speicheraufwand
- Dekompressionszeit
- Zeit zur Berechnung der Schnittmenge
- Zeit zur Berechnung der Vereinigung

Abschließend gibt der Artikel einen Überblick welche der untersuchten Kompressionsalgorithmen für die vier Kategorien am besten geeignet sind. So haben die invertierte-Listen-Kompressionsverfahren in den Bereichen Dekompressionszeit und Zeit zur Berechnung der Vereinigung eine bessere Performanz, während die Bitmap-Kompression besser in der Berechnung der Schnittmenge ist. In der Kategorie Speicheraufwand ist die invertierte-Listen-Kompression ebenfalls besser, außer bei großen Listen.

In [4] untersuchen Lin et. al. ob es sinnvoll ist Kompressionsstrategien auf Datenbanken anzuwenden und welche Strategien dafür geeignet sind. Es werden 4 Kompressionsverfahren betrachtet:

- **Dictionary Encoding (DICT)** Mapping von Strings zu 32-bit Integern in einem globalen Dictionary.
- **Run-Length Encoding (RLE)** Aufeinanderfolgende gleiche Werte werden durch das Paar (Anzahl, Wert) ersetzt.
- **Bitmap Encoding** Jeder mögliche Wert wird durch einen Bit-Vektor ersetzt, der angibt an welchen Stellen der Datenbank der Wert auftritt. Zusätzlich wird jeder Vektor mit RLE erneut komprimiert.
- **Huffman Encoding** Basierend auf der Häufigkeitsverteilung der Daten werden weniger Bits benutzt um Daten zu kodieren, die häufig vorkommen.

Abschließend stellen die Autoren fest, dass die Kompression von Datenbanken mit DICT, RLE oder Bitmap Encoding sinnvoll ist, da sie neben dem Reduzieren des Speicheraufwands auch das Bearbeiten von Anfragen beschleunigen. Dies kommt daher, dass Anfragen an Daten, die mit den 3 Verfahren komprimiert wurden, ohne Dekompression beantwortet werden können. Da dies beim Huffman Encoding nicht möglich ist, ist dieses Verfahren nicht für die Kompression von Datenbanken geeignet.

3 Optimierungsstrategien für die Statistikdatenbank

Die Statistikdatenbank speichert die Häufigkeit der Vorkommen aller Ressourcen im Graphen als Subjekt, Prädikat oder Objekt. Sie besteht aus Datenblöcken, die immer die selbe Länge haben und in 3 Abschnitte unterteilt sind. Die Abschnitte stehen für die 3 möglichen Vorkommen als Subjekt, Prädikat oder Objekt. Jeder Abschnitt besteht aus n Feldern, die jeweils die Anzahl der Vorkommen der Ressource in dem Teil des Graphen, der im Slave gespeichert ist, beinhalten. Ein Feld hat eine Größe von 8 Byte.

ID		Subjekt		Prädikat		Objekt	
0	mitarbeiter:Alice	4	0	0	0	0	0
1	mitarbeiter:Bob	1	3	0	0	1	0
2	mitarbeiter:Charlie	0	4	0	0	0	0
3	abteilungen:Abteilung 1	0	0	0	0	2	0
4	abteilungen:Abteilung 2	0	0	0	0	0	1
5	projekte:Projekt 1	0	0	0	0	0	2
6	typ:Teilzeit	0	0	0	0	0	2
7	typ:Vollzeit	0	0	0	0	1	0
8	1234.00	0	0	0	0	0	2
9	2345.00	0	0	0	0	1	0
10	relationen:arbeitetAls	0	0	1	2	0	0
11	relationen:arbeitetIn	0	0	2	1	0	0
12	relationen:arbeitetAn	0	0	0	2	0	0
13	relationen:hatGehalt	0	0	1	2	0	0
14	relationen:bossVon	0	0	1	0	0	0

Abbildung 5: Aufbau der Statistikdatenbank für die Partitionierung aus Abb. 2.

Beispiel 4. *Dieses Beispiel zeigt den Aufbau der Statistikdatenbank für die Partitionierung des Beispielgraphen aus Beispiel 3. So tritt die Ressource Bob mit der ID 1 beispielsweise 3-mal in Slave 2 als Subjekt auf. Wie in Abbildung 5 gut zu erkennen ist, sind die meisten Felder in der Datenbank gleich 0. Dies gibt die Möglichkeit zur Optimierung dieser Statistikdatenbank mithilfe von Kompressionsverfahren.*

Jeder Datenblock hat eine feste Größe und es werden alle Blöcke aneinandergehängt in einer einzigen Datei gespeichert. Die Position eines Datenblocks in dieser Datei ist durch die ID der inkrementell durchnummerierten Ressourcen gegeben. Diese Implementierung ist sehr ineffektiv, wie anhand von Abbildung 4 bereits erklärt wurde. Es gibt mehrere Möglichkeiten die aktuelle Implementierung zu verbessern, die in den folgenden Abschnitten erläutert werden.

3.1 Caching

Durch die Größe der Statistikdatenbank ist es nicht möglich sie vollständig in den Hauptspeicher zu laden. Das heißt beim Zugriff muss direkt auf der Festplatte gearbeitet werden. Um die langsamen Festplattenzugriffe zu vermeiden, werden beim Caching Teile der Datenbank anhand festgelegter Kriterien in den Hauptspeicher vorgeladen. Ein anderer Ansatz ist das Halten von bereits gelesenen Daten im Hauptspeicher. Dafür ist es relevant die Zugriffsarten auf die Statistikdatenbank zu kennen, denn anhand derer kann festgelegt werden, welche Datenblöcke in den Hauptspeicher vorgeladen werden sollen.

Eine Möglichkeit um das Lesen zu beschleunigen ist das LRU-Caching(*Least Recent Used*). Dabei wird, wenn der angefragte Block nicht im Cache vorliegt, ein sogenannter Miss, der jeweils am längsten nicht verwendete Datenblock aus dem Cache verdrängt und durch einen neuen Block ersetzt.

Beispiel 5. *Abbildung 6 zeigt die Anwendung des LRU-Verfahrens auf einen Cache, der 4 Datenblöcke speichern kann. So ist beispielsweise der Zugriff auf den Datenblock 2 ein Hit und der Datenblock wird an die oberste Stelle im Cache geschrieben. Der Zugriff auf Datenblock 6 ist jedoch ein Miss, deshalb muss dieser Datenblock erst in den Cache geladen werden und verdrängt damit den am längsten nicht verwendeten Datenblock 3.*

[3]https://www.sqlite.org/ zugegriffen am 28. November 2017

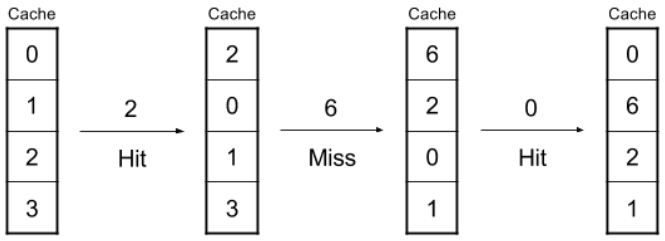

Abbildung 6: Anwendung des LRU-Verfahrens mit einem Speicher der Größe 4.

Eine mögliches Verfahren, bei dem Lesen und Schreiben beschleunigt werden, ist das Verwenden einer Memory-mapped File, bei der die Datenbank in mehrere Segmente unterteilt wird, von denen jeweils einer in den Hauptspeicher geladen wird.

Beispiel 6. *Für dieses Verfahren könnte die Datenbank aus Beispiel 4 in 2 Segmente zerteilt werden. Segment 1 umfasst die Datenblöcke 0 bis 7 und Segment 2 die Blöcke 8 bis 14. Solange die Zugriffe auf die Datenbank im selben Segment stattfinden, können die Anfragen schnell beantwortet werden. Wenn jedoch ein Datenblock aus einem anderen Segment angefragt wird, muss nun das vorherige Segment auf die Festplatte zurückgeschrieben und das gesamte andere Segment unter hohem Zeitaufwand in den Hauptspeicher geladen werden.*

3.2 Kompression

Die aktuelle Implementierung der Statistikdatenbank benötigt sehr viel Speicherplatz. Um dies zu reduzieren können die einzelnen Datenblöcke komprimiert werden. Dies ist allerdings nicht ohne Weiteres möglich, da die Größe der Datenblöcke in der aktuellen Implementierung fest gewählt ist und dadurch die Position des Datenblocks in der Datei anhand der Ressourcen-ID berechnet werden können. Um nun das Anwenden von Kompressionsverfahren zu ermöglichen, muss eine zusätzliche Indexdatei angelegt werden, die den IDs der Ressourcen ihre genaue Position in der Statistikdatenbank zuordnet. Mithilfe dieser Datei können die Datenblöcke verschiedene Größen haben.

Indexdatei

ID	Offset	ID	Offset
0	0	8	44
1	4	9	48
2	14	10	54
3	20	11	62
4	26	12	70
5	30	13	76
6	34	14	84
7	38		

Datenbankdatei

1	4	5	0	1	1	1	3	2	0
1	1	1	0	1	0	1	4	4	0
4	0	1	2	1	0	5	0	1	1
5	0	1	2	5	0	1	2	4	0
1	1	1	0	5	0	1	2	4	0
1	1	1	0	2	0	1	1	1	2
2	0	2	0	1	2	1	1	2	0
3	0	1	2	2	0	2	0	1	1
1	2	2	0	2	0	1	1	3	0

Abbildung 7: Aufbau der Statistikdatenbank nach der Anwendung von Run Length Encoding.

Beispiel 7. *Abbildung 7 zeigt den Aufbau der Statistikdatenbank aus Beispiel 4 nach dem Anwenden des Run Length Encoding, bei dem aufeinanderfolgende gleiche Werte durch das Paar (Anzahl, Wert) ersetzt werden. Der Datenblock* $[4, 0, 0, 0, 0, 0]$ *für die Ressource Alice mit der ID 0 wird dementsprechend dargestellt als* $[1, 4, 5, 0]$*. Die Statistikdatenbank besteht jetzt aus zwei Dateien, einer Indexdatei und der eigentlichen Datenbank. In der Indexdatei ist die Zuordnung zwischen Ressourcen-ID und der Position des Blocks in der Datenbankdatei gespeichert. Die Datenbankdatei enthält die eigentlichen Daten in komprimierter Form.*

4 Ziele

Die Ziele dieser Bachelorarbeit sind wie folgt:

1. Welche Charakteristiken haben die gespeicherten Daten in der Statistikdatenbank?

   Bestimmung der Häufigkeitsverteilung von Ressourcen. Wie oft kommen Ressourcen als Subjekt, Prädikat oder Objekt vor? Treten Ressourcen häufig in Kombinationen dieser Positionen auf? Kommen die selben Werte oft mehrmals hintereinander vor? Diese Informationen können genutzt werden um Kompressionsstrategien anzuwenden.

2. Welche Zugriffsmuster auf die Statistikdatenbank existieren?

   Bestimmung der Anzahl und Art der verwendendeten Operationen. Welche Zugriffsarten werden häufig genutzt? Zugriff auf benachbarte Ressourcen oder Springen zwischen Sektoren. Diese Inforamtionen ermöglichen das Anwenden einer geeigneten Cachingstrategie beim Datenbankzugriff.

3. Wie kann man die Statistikdatenbank gemäß der zuvor festgestellten Anforderungen optimieren?

   - Entwurf verschiedener Optimierungsstrategien gemäß den Anforderungen
   - Implementierung der Strategien
   - Evaluation gegen aktuelle Implementierung und relationale Datenbank

5 Vorgehen

Zu Beginn der Arbeit wird eine Erhebung der Anforderungen an die Statistikdatenbank stattfinden. Dies beinhaltet eine statistische Erhebung über die Verteilung der Häufigkeiten der einzelnen Ressourcen. Dabei soll festgestellt werden, wie oft jede Ressource als Subjekt, Prädikat oder Objekt vorkommt und außerdem wie oft Ressourcen an mehreren dieser Positionen auftreten. Des Weiteren werden die Zugriffe auf die Statistikdatenbank untersucht. Hier soll festgestellt werden, welche Art von Zugriffen auftreten und wie häufig diese stattfinden. Die erfassten Anforderungen werden im Folgenden zur Entwicklung geeigneter Optimierungsstrategien verwendet.

Anschließend wird in der Literatur nach Optimierungsstrategien gesucht, die in den zuvor festgestellten Bereichen Verbesserungen bringen könnten. Die so gefundenen Strategien werden einer Vorabevaluation unterzogen. Das heißt es werden die Vor- und Nachteile der einzelnen Techniken betrachtet sowie ihr Abschneiden im Vergleich zu anderen möglichen Strategien. Dazu wird Literatur wie [5] herangezogen. Im Zuge dieser Evaluation werden einige gut geeignete Strategien ausgewählt, die im Folgenden weiter untersucht werden.

Es werden verschiedene Optimierungsstrategien gemäß den Anforderungen entworfen und implementiert. Anschließend werden diese Strategien mit einer festgelegten Partitionierungstechnik getestet. Dabei können verschiedene Parameter, wie beispielsweise die Größe des Cache, noch variiert werden. Jede Strategie wird gegen die aktuelle Implementation sowie eine relationale Datenbank evaluiert.

6 Zeitliche Einteilung

Die Bachelorarbeit ist aufgeteilt in 5 Abschnitte. Die zeitliche Einteilung ist in Tabelle 1 dargestellt. Zu Beginn wird eine Analyse der Zugriffsmuster auf die Statistikdatenbank sowie eine Literaturrecherche von je 2 Wochen stattfinden. Im Anschluss werden anhand der festgestellten Anforderungen verschiedene Optimierungsstrategien entworfen. Anschließend werden die entworfenen Strategien implementiert. Für diese zwei Schritte sind jeweils 4 Wochen vorgesehen. Da die Durchführung und Auswertung der Evaluation aller implementierten Optimierungsstrategien mehr Zeit benötigt, sind dafür 8 Wochen geplant. Abschließend bleiben noch 4 Wochen, die als Puffer und für die Finalisierung der Arbeit gedacht sind. Das Schreiben der Bachelorarbeit wird parallel während des gesamten Zeitraums stattfinden.

Literatur

[1] Big data graph. http://km.aifb.kit.edu/projects/btc-2014/. zugegriffen am: 07.11.2017.

[2] Rdf. https://www.w3.org/2001/sw/wiki/RDF. zugegriffen am: 07.11.2017.

Tabelle 1: Zeitliche Einteilung der Arbeit

Woche	**Vorgehen**
Woche 1-2	Analyse der Zugriffsmuster
Woche 3-4	Literaturrecherche
Woche 5-8	Entwurf von Optimierungsstrategien
Woche 9-12	Implementation der Optimierungsstrategien
Woche 13-20	Evaluation aller Strategien
Woche 21-24	Überarbeitung, Finalisierungsphase

[3] D. Janke, S. Staab, and M. Thimm. Koral: A Glass Box Profiling System for Individual Components of Distributed RDF Stores. In R. Usbeck, A. Ngonga, J.-D. Kim, K.-S. Choi, P. Cimiano, I. Fundulaki, and A. Krithara, editors, *Joint Proceedings of BLINK2017: Benchmarking Linked Data and NLIWoD3: Natural Language Interfaces for the Web of Data (BLINK2017-NLIWoD3)*, number 1932 in CEUR Workshop Proceedings, Aachen, 2017.

[4] C. Lin, J. Wang, and Y. Papakonstantinou. Data compression for analytics over large-scale in-memory column databases (summary paper). *CoRR*, abs/1606.09315, 2016.

[5] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson. An experimental study of bitmap compression vs. inverted list compression. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 993–1008, New York, NY, USA, 2017. ACM.