Cloud-Services und effiziente Anfrageverarbeitung für Linked Open Data

[Work in Progress]

Heiko Betz, Kai-Uwe Sattler
Department of Computer Science and Automation
TU Ilmenau, Germany
{first.last}@tu-ilmenau.de

ABSTRACT

Der Verbreitungsgrad von Linked Open Data hat in den letzten Jahren massiv zugenommen. Stetig erscheinen neue Quellen, die RDF-Daten frei zur Verfügung stellen. Aktuell diskutiert die Bundesregierung über ein neues Gesetz, welches zur Offenlegung von Daten der öffentlichen Hand verpflichtet. Durch diese Maßnahme, steigt u. a. die Menge an Linked Open Data sehr schnell an. Es werden neue Verfahren benötigt, die mit sehr vielen RDF-Daten gleichzeitig eine effiziente, skalierbare und mit Garantien versehene Abfrage von Daten mittels SPARQL gewährleistet. In dieser Arbeit wird ein reines In-Memory Shared-Nothing-System vorgestellt, das die genannten Anforderungen in effizienter Weise erfüllt. Hierfür werden verschiedenste Optimierungsmaßnahmen ergriffen, die das Potenzial moderner Hardware umfassend ausnutzen.

1. EINFÜHRUNG

Die Anzahl an verfügbaren Quellen für Linked Open Data (LOD) wächst stetig an[1]. Unter anderem wurde vor kurzem von der Bundesregierung die Open Data-Initiative gestartet. Es sollen Daten, die durch Steuergelder finanziert wurden, der Öffentlichkeit frei zur Verfügung stehen[2]. Insgesamt geht die Anzahl an verfügbaren LOD-Quellen in diesem hochverteilten System in die Hunderttausende bzw. Millionen über. Es werden disjunkte bzw. teils überlappende Daten bereitgestellt, die häufig untereinander verlinkt sind, jedoch nicht immer einem gemeinsamen Qualitätsstandard entsprechen.

Die innere Struktur von LOD entspricht dem Ressource Description Framework [1] (RDF) Aufbau. In diesem werden Daten als Tripel repräsentiert (Subjekt, Prädikat und Objekt). Hierauf baut wiederum die Abfragesprache SPARQL [2] auf. Dies ist eine graph-basierte Anfragesprache für RDF-Daten. Sie definiert neben der Möglichkeit Daten abzufragen, ähnlich dem SQL aus der Datenbankwelt, ein Transportprotokoll zwischen verschiedenen SPARQL-Endpoints. In SPARQL selbst werden Basic Graph Patterns (BGP) für die Abfrage von Daten verwendet, die aus einem oder mehreren Tripeln bestehen. Hierdurch ergeben sich joinintensive Abfragen, die eine weitere detaillierte Betrachtung benötigen.

[1]Die LOD-Cloud, welche nur einen minimalen Ausschnitt darstellt und unter http://www4.wiwiss.fu-berlin.de/lodcloud/state/ zu finden ist, umfasst mittlerweile ca. 31 Milliarden Einträge.

[2]http://heise.de/-1429179

24th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 29.05.2012 - 01.06.2012, Lübbenau, Germany.
Copyright is held by the author/owner(s).

Fallunterscheidung.

Für die Beantwortung von (komplexen) quellenübergreifenden Anfragen müssen nun zwei Fälle unterschieden werden [7]:

1. Verteilter Ansatz: Daten bleiben in den Quellen. Anfragen werden von einem Koordinator entgegengenommen, in Subanfragen aufgesplittet, den entsprechenden Quellen zugesandt, von diesen bearbeitet und das Resultat vom Koordinator entgegengenommen.

2. Data Warehouse Ansatz: Daten werden von verschiedenen Quellen geladen, vorverarbeitet und lokal abgelegt (materialisiert). Anfragen können direkt durchgeführt werden. Die Quellen bleiben gleichzeitig erhalten.

Punkt 1 besitzt mehrere potenzielle Vorteile. Durch die Verteilung wird ein Single Point of Failure (SPOF) vermieden. Der Koordinator muss keine Daten speichern, da sie alle in den Quellen vorliegen. Er muss nur einen geringen Speicherplatz und nur eine geringe Rechenkapazität zur Verfügung stellen. Ein wichtiger Punkt, der für eine Verteilung der Quellen spricht, ist, dass die Daten nicht an andere Unternehmen herausgegeben werden müssen. Es können 'Firmengeheimnisse' bewahrt werden. Als Nachteile ergeben sich jedoch einige Punkte. Die Zerlegung von Anfragen in Subqueries und alle daraus folgenden Schritte sind ein hoch komplexes Problem (Parsing, Normalisierung, eingebettete Anfragen 'herausdrücken', Vereinfachung, Datenlokalisation, Optimierung). Zwei weiter nicht zu unterschätzende Nachteile sind die fehlenden Kontroll- und Überwachungsmechanismen der Quellen. Die Antwortzeit in einem solchen Szenario ist nach oben unbeschränkt. Es ergibt sich, dass keinerlei Garantien abgegeben werden können. Weiterhin ist durch die Verteilung das verwendete Schema dem Koordinator gänzlich unbekannt. Viele Quellen sind zudem nicht in der Lage, SPARQL-Anfragen zu verarbeiten. Sie liefern nur die Quellen über einen Webserver aus. Wären SPARQL-Anfragen

möglich, wären Analyseanfragen à la Data Warehouse-Anfragen auch nur bedingt möglich.

Die zentralen Nachteile des Punktes 2 sind die (i.) benötigte Zeit zum Einsammeln aller Daten, die (ii.) Durchführung einer Vorberechnung und das Problem, dass niemals gewährleistet werden kann, dass (iii.) alle Daten aktuell sind. Zudem wird (iv.) ein SPOF, sowie ein (v.) extrem hoher Speicherplatz an einer einzigen zentralen Instanz in Kauf genommen. Ein weiterer Nachteil beider Fälle ist das (vi.) unbekannte Schema und die (vii.) geringe Datenqualität. Im zentralisierten Fall kann jedoch auf die letzten beiden Probleme effizient reagiert werden, indem entsprechende Indizes angelegt und/oder verschiedenste Optimierungen durchgeführt werden. Eine entsprechende Datenvorverarbeitung kann zudem davor gestartet werden, um die Datenqualität zu verbessern. Beides wird durch die einheitliche Form der Daten als reine Tripel unterstützt. Der große Vorteil, der sich aus dem zentralisierten Ansatz ergibt, ist die Möglichkeit, Anfragezeiten zu einem bestimmten Prozentsatz zu gewährleisten, da sich das gesamte System unter der Kontrolle einer Instanz befindet. Weiterhin kann ein Scale-Out (Erhöhung der Speicherkapazität (v.), Verbesserung der Antwortzeiten) und eine Replikation (Beseitigung SPOF (iv.), Verbesserung der Antwortzeiten) angestrebt werden. Zur Verringerung der Speicherkapazität (v.) können effiziente Kompressionstechniken verwendet werden, die zugleich lese- und schreiboptimiert sind, was zu einer weiteren Verringerung des benötigten Speicherplatzes führt. Durch die Verwendung von push und Bulk Load-Techniken zur Datenintegration anstelle von reinen pull-Techniken kann garantiert werden, dass die abgefragten Daten aktuell (iii.) sind. Das Einsammeln von Daten ist nur zu Beginn einmal nötig, wenn eine neue Datenquelle erschlossen wird (i. und ii.).

Der Mechanismus des Scale-Outs und der Replikation bedingen jedoch, dass eine Zerlegung der Anfrage durchgeführt werden muss. Wobei dieser eine Nachteil durch die oben genannten Vorteile aufgewogen wird. Insgesamt sprechen viele Faktoren für die Verwendung eines zentralisierten Ansatzes.

Garantien bezüglich den soeben erwähnten Antwortzeiten sind äußerst komplex in ihrer Umsetzung und benötigen im extremen Fall viele vorallokierte Kapazitäten, die während normaler Nutzung brachliegen und hohe Kosten verursachen. Stattdessen werden sie nur während Lastspitzen benötigt. Folglich werden aus diesem Grund häufig prozentuale Werte angegeben. Es sollen beispielsweiße 99,99 % aller Anfragen in der maximal erlaubten Zeitspanne beantwortet werden. Erst ein solches Vorgehen erlaubt es, dass während einer Lastspitze neue Kapazitäten hinzugenommen werden können (automatisierter Scale-Out). Das Ziel hierbei ist, zukünftige Anfragen wieder innerhalb der gesetzten Antwortzeit beantworten zu können. Am Ende der Lastspitze werden die zusätzlich allokierten Ressourcen wieder freigegeben. Letztendlich werden hierdurch Kosten für den Betreiber des Services eingespart. Wichtig zu erwähnen ist, dass die Garantie bezüglich der Antwortzeit nicht für jede beliebige SPARQL-Anfrage garantiert werden kann. Stattdessen soll dies nur für „gewöhnliche“ Anfragen gelten.

Projektziel.

Das Ziel dieses Projektes ist es, ein hochverfügbares, -skalierbares und -effizientes System aufzubauen, welches mehrere hundert Milliarden bzw. mehrere Billionen Tripel von RDF-Daten in einer materialisierten Form vorhält. Es soll hierbei als Framework und nicht als reine Datenablage angesehen werden. Neben SPARQL-Anfragen soll es den Benutzer im gesamten Workflow unterstützen. Unter Workflow werden die Folgenden Operationen verstanden: Einfügen/Löschen/Ändern von Tripeln sowie die Abfrage/Analyse von Daten. Unter dem Stichpunkt Analyse fallen sämtliche Data-Mining-Aufgaben, die in einem Data-Warehouse-Szenario möglich sind. Ein weiteres Ziel ist die Unterstützung von SLAs. Neben dem Garantieren von maximalen Antwortzeiten sollen auch Garantien bezüglich der Verfügbarkeit und der Aktualität von Daten, sowie einige andere mehr beachtete werden. Letztendlich soll das System als ein Cloud-Service für LOD-Daten propagiert werden. Das Ziel hierbei ist, Kosten für den Endbenutzer einzusparen. Er möchte nur für den von ihm selbst verursachten Aufwand bezahlen (Service on Demand). Im alternativen Fall müsste eine eigene Infrastruktur aufgebaut werden, was häufig zu viel höheren Kosten führt.

Diese Arbeit beschäftigt sich mit einem Teil aus dem soeben beschriebenen Zieles. Es soll ein erster Ansatz für einen Datenspeicher für Linked Open Data aufgezeigt werden, der mit sehr großen Datenmengen eine effiziente Anfrageverarbeitung ermöglicht. Hierfür ist ein zuverlässiges Scale-Out-Verfahren notwendig, welches vorgestellt wird. Des Weiteren soll das gewählte Verfahren auf eine wechselnde Anfragelast reagieren können.

2. STATE-OF-THE-ART

Für die Verarbeitung von LOD existieren bereits viele verschiedene Systeme, die RDF-Daten entgegennehmen und SPARQL-Queries ausführen. Einige bekanntere System sind Virtuoso [5], MonetDB [3], Hexastore [14], Jena [15] und RDF-3X [11]. Virtuoso ist ein weitverbreitetes relationales Datenbanksystem, das aus allen Kombinationen des Subjektes, Prädikates und Objektes Indizes erzeugt. Somit kann sehr schnell auf Daten zugegriffen werden. MonetDB ist ein OpenSource Column-Store, der eine Menge von Zwischenergebnissen im Speicher hält, um neue und ähnliche Anfragen schneller bearbeiten zu können. Hexastore ist eine In-Memory-Lösung. Es erzeugt aus allen möglichen Kombinationen von Tripeln Indizes, die daraufhin in einer Liste abgelegt werden. Für die Vermeidung von Dopplungen und zur Kompression werden Strings in einem Wörterbuch abgelegt. Jena ein bekanntes OpenSource-Projekt kann verschiedene Datenspeicher verwenden. Einerseits können die Daten in einem eigenen Format abgelegt werden, andererseits in einer relationalen Datenbank. Auch Jena verwendet ein Wörterbuch zur Kompression. RDF-3X verwendet auch mehrere Indizes, um alle möglichen Kombinationen abzuspeichern und legt diese in B+-Bäumen ab. Zudem enthält es eine ausgeklügelte Anfrageoptimierung, die speziell auf die Besonderheiten von RDF abgestimmt ist.

Werden alle vorgestellten Systeme genauer betrachtet, stellt man fest, dass diese einige Nachteile besitzen. Entweder sind diese auf bestimmte Operationen (lesen, ändern) und/oder Domänen getrimmt und/oder sind nicht skalierbar im Sinne eines Scale-Outs und somit ungeeignet für sehr große Datenmengen. Stattdessen setzten sie auf die Leistung einer einzigen Maschine, was zu einem erheblichen Performance-, Speicherplatz- und Verfügbarkeitsproblem werden kann. Weiterhin unterstützt keines der erwähnten Systeme Garantien bezüglich Antwortzeiten. Sie setzen zudem auf einen Permanentspeicher, welcher I/O-Zugriffe erfordert. Dies er-

Figure 1: Aufbau des Systems mit mehreren unabhängigen Knoten, die untereinander verbunden sind. Annahme einer Query und Weitergabe an einem beliebigen Knoten. Einspielen von neuen Datenquellen durch Push und Pull.

höht wiederum sehr schnell die Antwortzeit. Eine Gewährung von Garantien wird erschwert. Einige dieser Ansätze sind zudem bereits mit wenigen Milliarden Tripeln überfordert (Jena). Andere verwenden alte Konzepte für neuere Probleme (Virtuoso), was nicht immer zu einer optimalen Lösung beiträgt.

Bezüglich den Garantien von Antwortzeiten, beschreiben die Autoren in [12] und [9] ein Verfahren zur Beantwortung von Anfragen in einem konstanten Zeitintervall. Dies wird durch verschiedenste Optimierungen erreicht. Unter anderem wird beschrieben, wie ein Row- und Column-Store miteinander verschmolzen wird und trotzdem eine effiziente Anfrageverarbeitung ermöglicht wird.

3. ARCHITEKTUR

Die Architektur des Gesamtsystems ist in Abbildung 1 aufgezeigt. Sie folgt dem typischen Scale-Out-Paradigma mit verschiedenen Shared-Nothing-Systemen (Kreise in der Mitte), die alle gleichberechtigt sind. Anfragen können von beliebigen Knoten entgegengenommen werden. Im nächsten Schritt wird ein Queryrewriting durchgeführt und an die entsprechenden Knoten weitergegeben. Die Empfänger nehmen die umgeschriebenen Anfragen entgegen, bearbeiten diese und senden letztendlich das Ergebnis zurück. Nachdem alle Einzelergebnisse eingetroffen sind werden diese mittels verschiedener Join-Operationen zu einem Ergebnis verknüpft. Dieses wird letztlich an den ursprünglich anfragenden Client als Ergebnis zurück gesandt. Auf der rechten Seite der Grafik ist ersichtlich, dass neue Daten mittels push- oder pull-Techniken eingespielt werden können und dies während der Laufzeit, mit Einhaltung der Antwortzeitgarantien.

3.1 Lokale Verarbeitung

Die lokale Verarbeitungseinheit liegt auf einem Knoten und besteht aus einem Java-Programm, sowie vorwiegend aus einer C++ In-Memory-Datenhaltung und Datenabfragemöglichkeit, die LODCache genannt wird. Dies ist eine hochperformante, reine In-Memory Lösung für LOD, die den L2-, wenn vorhanden L3-Cache und die RAM-Struktur moderner Hardware effizient ausnutzt. Allokierte Datenfelder, sogenannte Chunks, entsprechen genau der Größe des L2-Caches. Dies ermöglicht es sehr effizient auf Daten zuzugreifen, diese zu Bearbeiten und Berechnungen auf diesen durchzuführen. Des Weiteren können Anfragen an den LODCache gestellt, sowie Änderungen und Einfügeoperationen durchgeführt werden. Die Verwendung einer reinen In-Memory Lösung basiert auf der Grundlage, zukünftig Antwortzeiten garantieren zu können.

Linked Open Data bestehen ausschließlich aus Tripeln, die wiederum aus Strings bestehen. Werden die darin abgespeicherten Werte genauer untersucht, stellt man fest, dass dieselben Stringwerte mehrmals auftreten. Hier ist eine Kompression nützlich, um Speicherplatz einzusparen. Dies fällt umso mehr ins Gewicht, da der Data-Warehouse Ansatz angestrebt wird. Die naheliegenste Technik, ist der Einsatz eines *rein* lokalen Wörterbuches. In einem solchen wird jeder Stringwert auf eine eindeutige ID gemappt. Hierfür ist wiederum eine effiziente Anfrageverarbeitung von Nöten, die im folgenden Abschnitt beschrieben wird.

Kombinierte Row- und Column-Store.

Ein Chunk besteht intern aus einer Kombination von Row- und Column-Store [9]. Jede logische Zeile enthält drei Spalten für Subjekt, Prädikat und Objekt, wobei in einem jeden Feld nur ein einzelner Integerwerte abgelegt ist. Werden nun alle einzelnen Integerwerte eines Tripels mittels Shift-Operationen disjunkt zu einem Wert vereint, ergibt sich ein Wort mit 96 Bit Breite, wenn jeweils 32 Bit für Subjekt, Prädikat und Objekt angenommen werden. Somit wird die Datenzeile aus drei logischen Spalten zu einer physischen Spalte vereint, die in einem theoretischen Drittel der Zeit überprüft werden kann.

Moderne CPUs besitzen interne SIMD[3]-Register die eine Breite von 256-Bit besitzen. Diese wurde mit dem neuen AVX[4]-Befehlsatz von Intel eingeführt. In einem solchen Register werden mehrere logisch getrennte Einheiten zu einer physischen Einheit verbunden, die in einer Instruktion gleichzeitig bearbeitet werden können. Hierfür müssen die einzulesenden Daten auf eine volle 2-er Potenz ergänzt und die Wortgrenze bekannt sein.

Werden die Fakten vereint, muss zuerst das 96 Bit Wort auf 128 Bit, ergänzt werden. Mit Hilfe der verfügbaren 256-Bit können nun zwei logische Datenzeilen mit jeweils 128-Bit gleichzeitig überprüft werden. Theoretisch ergibt sich eine bis zu sechsfache Performancesteigerung. Mittels der Vermeidung von Sprung- und sonstigen Befehlen, die Leerlauf produzieren, ist dieses Verfahren sehr Cache freundlich. Einmal verwendete Daten können stur linear abgearbeitet werden. Zugriffe auf dem langsamen RAM werden damit vermieden. Insgesamt wird die Struktur moderner Hardware sehr gut ausgenutzt und eine Performancesteigerung ergibt sich.

Wie oben erwähnt, muss in diesem Fall eine Ergänzung von 96-Bit auf 128-Bit durchgeführt werden. Es verbleiben 32 ungenützte Bit. Diese können für eine weitere Spalte verwendet werden. In dieser kann die Sprache bzw. der Datentyp des (Literal)Objektes vermerkt werden [1]. Sollten entsprechende Filterbedingungen auftreten, können diese direkt mit übernommen werden, was zu einer weiteren Performancesteigerung beiträgt. Andererseits können die 32-Bit pro Spalte zu gering sein und entsprechend erweitert werden. Es ist auch eine Kombination aus beiden möglich (z. B. 40 Bit pro Spalte und 8 Bit für Sprache und Datentyp).

Durch die Verwendung von Integerwerten, die eine feste Bitbreite besitzen, wird auf einen expliziten und unvorher-

[3]Single instruction, multiple data

[4]Advanced Vector Extensions

sehbar langen Stringvergleich verzichtet. Dies erhöht weiter die Performance und verbessert die Antwortzeit.

Wie zu erkennen ist, wird auf eine Erzeugung von zusätzlichen Indixes verzichtet. Dies ergibt sich aus der Speicherrestriktion und dem nötigen Wartungsaufwand, der während jeder Änderung anfällt. Hierdurch können Operationen unvorhersehbar lange blockiert werden. Es sind keine Garantien mehr möglich. Anstelle werden wie in einem gewöhnlichen Column-Store alle Elemente stetig geprüft. Dies ist durch moderne Hardware und der ständig steigenden Parallelisierung problemlos möglich. Sind c Cores gegeben, muss jeder Core nur maximal $\lceil \frac{cC}{c} \rceil$ Elemente überprüfen, wobei cC die Anzahl der Chunks angibt.

Lokale Anfrageverarbeitung.

Der LODCache besitzt keinerlei Logik für die Optimierung von Anfragen. Aus diesem Grund ist eine vorgelagerte Java-Applikation vorhanden. Diese nimmt SPARQL-Anfragen entgegen, zerlegt diese, optimiert sie (lokale Optimierung) und überführt sie in die Sprache des LODCaches. Dieser führt die entsprechende Operation durch und übergibt das Ergebnis dem Java-Programm mittels JNI[5], dass die ermittelten Daten an den Aufrufer sendet.

Wird eine exakt Match Anfrage mit Subjekt_1, Objekt_1 und einem freien Prädikat `?p` an den LODCache gesandt, werden zuerst die Integerwerte der fest definierten Strings gesucht und in ein Wort mittels Shift und logischen OR-Operationen zu einer Maske (`m`) ergänzt. Dies wird mittels vier CPU-Operationen durchgeführt werden (COPY, SHIFT, OR, SHIFT). Die Operationen zum Filtern der Daten aus einem Chunk beschränken sich daraufhin nur noch auf ein logisches AND und einen Vergleich (CMP), je Eintrag und Chunk. Die genaue Berechnung ist im Folgenden nochmals dargestellt. Wobei `e` das zu überprüfende Element ist, `m` die Maske und `r` das Ergebnis als boolescher Wert.

```
r = (e AND m) CMP m
```

Werden Bereichsanfragen betrachtet, muss das Mapping der Strings auf Integerwerte eindeutig und ordnungserhaltend sein, da sonst immer das Wörterbuch zu Rate gezogen werden müsste. Dies bedeutet, jeder Integerwert muss mit der lexikografischen Ordnung seines gegenüberliegenden Strings übereinstimmen. Nur so können die in [9] aufgeführten Operationen angewandt und die Performance von SIMD-Befehlen ausgenutzt werden. Hierfür muss zuerst der minimale und maximale Integerwert der Bereichsanfrage ermittelt werden. Daraufhin werden zwei Masken erstellt und die Daten in den Chunks mit diesen, durch verschiedene logische Operationen, verglichen.

Ordnungserhaltender Baum.

Ein Problem tritt jedoch beim Einfügen von neuen Stringwerten auf. Diese werden meist lexikografisch zwischen zwei bereits bestehenden Elementen eingeordnet. Somit müssten sich in einem naiven Ansatz alle IDs aller nachfolgenden Stringwerte ändern. Das Wörterbuch und alle bestehenden Chunks müssen daraufhin überprüft und ggf. abgeändert werden.

Dies ist ein sehr großes und äußerst schwerwiegendes Problem, welches sich nicht vermeiden lässt. Es kann nur versucht werden, dass die Reorganisation möglichst selten und

[5]Java Native Interface

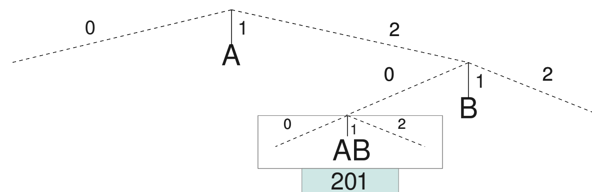

Figure 2: Aufbau des Baumes nach dem Einfügen von 'AB' (Rechteck), welches über den Pfad 201 erreichbar ist.

gleichzeitig effizient durchgeführt werden kann. Dies könnte naiv erreicht werden, indem jeder nachfolgende Mappingwert einige Zahlen zu seinem Vorgänger überspringt. Treten nun neue Einträge auf, können diese solange eingefügt werden, wie freie Plätze existieren. Ein Problem dieses Ansatzes ist jedoch, dass nicht bekannt ist, in welcher Reihenfolge Strings auftreten und es ist gänzlich unbekannt an welcher freien Position ein Element eingefügt werden muss/soll (in der Mitte, an der ersten oder letzten freien Position?). Eine falsche Position bedingt eine schnellere Reorganisation. Aus diesem Grund muss ein besser geeignetes Verfahren verwendet werden.

Ein solches wird für XML-Daten in [8] beschrieben. Die Autoren zeigen ein rekursives Verfahren, um Daten in einem Baum abzulegen und eine eindeutige ID zu generieren. Hierfür wird auf jeder Ebene jede mögliche Verzweigung durchnummeriert[6]. Nun wird zwischen geraden und ungeraden Elementen unterschieden. Die ungeraden Elemente sind vorhanden, um darin Werte sortiert aufzunehmen (durchgezogene Linie). Im Gegensatz zu den Ungeraden, denn diese spannen einen neuen Unterbaum auf (gestrichelte Linie). Soll ein neuer Wert eingefügt werden, wird dieser auf der höchsten Ebene (sortiert) hinzugefügt, die (i.) mindestens einen freien Platz besitzt und (ii.) lexikografisch den String korrekt einordnet. Das neue Element kann nun über die Konkatenation aller traversierten Baumverzweigungen eindeutig identifiziert werden und wird sinnvollerweise in einem String überführt. In Abbildung 2 ist ein Beispiel mit den Elementen 'A', 'B' und dem neuen Element 'AB' gegeben. Durch diesen Ansatz ist gewährleistet, dass sich neue Elemente immer zwischen zwei bereits bestehenden Elementen einsortieren lassen. Problematisch an diesem Ansatz ist jedoch, dass beliebig viele Unterbäume generiert werden können, der Baum zu einer Liste entartet und somit die Pfadlänge zu einem Element sehr lang werden kann. Dies widerspricht jedoch der oben genannten Forderung, nach einer festen Breite für Elemente, sowie der Verwendung eines Integerwertes. Aus diesem Grund muss eine Modifikation dieses Ansatzes verwendet werden.

Anstelle eines Strings, um den Pfad eindeutig zu identifizieren, werden nun beispielsweise 32 Bit Werte verwendet. Im Folgenden werden weiterhin jeweils 2 Bit pro Ebene verwendet. Wobei für jede neue Ebene, die neu hinzukommenden Bits an den bereits bestehenden Bits dahinter gehangen werden (von der größten zur kleinsten Wertigkeit). Es ergeben sich insgesamt maximal 16 Ebenen. Von den möglichen vier ($2^b = 2^2$) Werten pro Ebene können nur drei verwendet werden (`00`, `01`, `10`). Dies ergibt sich daraus, da der Wert `11` ungerade ist. In einem solchen müsste nach der

[6]In den weiteren Ausführungen wird von 0 gestartet.

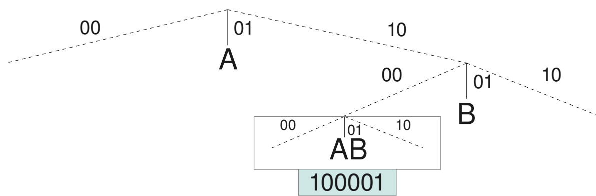

Figure 3: Aufbau des Baumes nach dem Einfügen von 'AB' (Rechteck), welches über den Pfad 100001 erreichbar ist.

oben genannten Definition ein Element ablegt werden. Jedoch existiert nach diesem kein weiteres gerades Element, womit nach einem falsch einsortierten Element kein weiteres lexikografisch dahinter liegendes Element eingefügt werden kann. Dies würde eine frühere Reorganisation erzwingen, was zu vermeiden ist. In Abbildung 3 ist das bereits weiter oben gezeigte Beispiel nochmals auf diesen Sachverhalt abgebildet.

Bewertung.

Ein großes Problem dieses Ansatzes ist jedoch die geringe Auslastung. So werden höchstens 50 % der möglichen Werte verwendet und auch nur genau dann, wenn keine zusätzlichen Ebenen vorhanden sind. Somit treten wiederum sehr schnelle Reorganisationen auf. Auf der anderen Seite kann eine Reorganisation sehr schnell durchgeführt werden. Sie ist auf wenige logische Operationen pro Wörterbuch- und Chunk-Eintrag beschränkt (SHIFT- und OR-Operationen). Gleichzeitig kann durch eine Reorganisation die Ebenenanzahl verringert und somit die Anzahl an Bits pro Ebene vergrößert werden. Die theoretisch mögliche Auslastungsgrenze steigt automatisch an.

3.2 Globale Verarbeitung

Die globale Architektur besteht aus der Zusammenfassung aller lokalen Verarbeitungseinheiten zu einer globalen Einheit. Diese sind mittels der Technik eines Chord-Ringes [13] miteinander verbunden. Er besteht aus n unabhängigen und gleichwertigen Elementen, die in einem geschlossenen Ring angeordnet sind. Wobei jedes Element einen Vorgänger und einen Nachfolger besitzt (siehe Abbildung 1; Doppeltpfeile mit durchgehender Linie). Neben diesen Verbindungen existieren noch sogenannte Finger (siehe Abbildung 1; einfacher Pfeil mit gestrichelter Linie). Dies sind zusätzliche unidirektionale Verbindungen, die mehrere Nachfolger überspringen und somit für eine schnellere Kontaktaufnahme mit einem beliebigen Element vorhanden sind. Wären diese nicht vorhanden, müsste eine Anfrage von einem Element zum anderen solange weitergereicht werden, bis der Empfänger erreicht ist ($O(n)$). Dies wird durch die Finger auf $O(\log_2 n)$ verkürzt.

Fragmentierungsfunktion.

Für die Umsetzung eines Scale-Outs wird eine Fragmentierungs- und Allokationsfunktion benötigt. Erstere definiert wie welche Daten aufgeteilt werden, letztere auf welchen Knoten welches Element abgelegt wird. Beide sollten möglichst einfach zu berechnen sein, da sie für jede Anfrage benötigt werden (Datenlokalisation). Der Chord-Ring definiert hier bereits eine Hashfunktion $h(x)$. Diese erzeugt durch die Eingabe von Daten einen Hashwert, der genau einem Knoten im Chord-Ring zugeordnet ist. Eine einfache Datenlokalisationsfunktion ist hierdurch gegeben. Der Vorteil, der hieraus entsteht ist ein globaler und flüchtiger Index, der keinerlei Wartung benötigt und auf allen n Elementen gleichermaßen zur Verfügung steht.

Für die Indizierung von RDF-Tripeln werden alle einstelligen Kombinationen aus Subjekt, Prädikat und Objekt erzeugt (s→op, p→so, o→sp). Diese werden nun mittels $h(x)$ auf den Ring verteilt, indem das zu indizierende Element in $h(x)$ gegeben wird und der entsprechende Knoten bestimmt wird. Für dasselbe Literal ergibt sich immer derselbe Hash und somit ist immer derselbe Knoten zuständig.

Globale Anfrageverarbeitung.

Im ersten Schritt wird eine beliebige SPARQL-Anfrage einem Client an einem beliebigen Knotenelement gesandt. Der Empfänger wird zum Koordinator dieser Anfrage. Nun zerlegt er die SPARQL-Anfrage und überprüft, ob er alle Daten lokal besitzt. Ist dies der Fall, liest er die Daten aus und führt die gesamte Berechnung durch. Ansonsten leitet er die umgeschriebenen Subanfragen an die zuständigen Knoten weiter. Hierfür wird die Hashfunktion $h(x)$ benötigt, in dem der nicht variable Teil einer jeden WHERE-Bedingung eingetragen wird. Während des Rewriting-Vorganges werden FILTER-Statements beachtet und in die Subqueries einbezogen, falls dies möglich ist (globale Optimierung). Durch diese Technik arbeiten mehrere Knoten parallel an derselben Ausgangsquery. Als Nebeneffekt wird die Datenmenge verkleinert. Nach dem Erhalt der Subqueries arbeiten die Knoten diese ab (lokale Anfrageverarbeitung; siehe Kapitel 3.1) und senden das Ergebnis an den Koordinator zurück. Dieser führt die benötigten Joins und die restlichen FILTER-, GROUP BY-, HAVING-, ORDER BY-, LIMIT-, OFFSET-, Projektions-, usw. Operationen durch. Zum Schluss wird das Ergebnis an den Client gesandt.

Bewertung.

Ein Problem dieses Ansatzes ist die mehrfache redundante Datenhaltung derselben Daten in maximal drei verschiedenen Knoten und die dafür zweimal höhere Speicherkapazität. Dies ist jedoch nur bedingt ein Nachteil. Durch den automatisierten Scale-Out-Mechanismus kann ein lokales Element entlastet werden, indem ein neuer Knoten einen Teilbereich der Hashwerte und die darin abgebildeten Daten für sich beansprucht. Ein weiterer Vorteil ist die Möglichkeit auf sehr große Anfragelasten dynamisch reagieren zu können. Es können automatisch neue Knoten hinzugefügt werden, die einen Teil der Anfragelast übernehmen. Die Aufteilung des Speicherplatzes und Anfragelast, wird jeweils durch die konsistente Hash-Funktion $h(x)$ garantiert [10]. Durch den beschriebenen Scale-Out, wächst die maximale Pfadlänge nur bedingt an, da diese durch $O(\log_2 n)$ definiert ist (was einem Vorteil entspricht). Ein weiterer Vorteil ist, dass zu jeder WHERE-Bedingung maximal ein Knoten involviert ist. Es müssen keine knotenübergreifenden Daten für bspw. Bereichsanfrage gesammelt werden. Dies ist nur zwischen verschiedenen WHERE-Bedingungen nötig, die mittels Join verknüpft werden.

4. ZUSAMMENFASSUNG

In dieser Arbeit wurde ein Ansatz vorgestellt, der mehre-

re Hundert Milliarden bzw. mehrere Billionen RDF-Tripel Linked Open Data effizient verwalten kann. Es wurde eine Unterscheidung in lokaler und globaler Verarbeitung durchgeführt. Die lokale Verarbeitungseinheit besteht aus einem hochoptimierten In-Memory C++-Programm zur Datenhaltung und -abfrage, das die Strukturen moderner Hardware effizient ausnutzt. Im selben Abschnitt wurde eine Möglichkeit aufgezeigt, Bereichsanfragen effizient zu verarbeiten, indem ein ordnungserhaltendes Mapping von Strings auf Integerwerten dargestellt wurde. Dies wird durch einen ordnungserhaltenden und updatefreundlichen Baum garantiert.

Die globale Verarbeitungseinheit besteht aus dem Zusammenschluss mehrerer lokaler Komponenten und verwendet hierzu die Technik des Chord-Ringes. Jedes Element ist gleichberechtigt, es existiert somit kein zentraler Koordinator. Für die Verbindung untereinander existieren Finger, die eine maximale Anzahl an Weiterleitungen garantieren. Zur Verarbeitung von beliebigen SPARQL-Anfragen werden diese von einem Knoten entgegengenommen, optimiert und an den betreffenden Knoten gesandt. Zur Datenlokalisation wird eine einfache Hashfunktion und kein global zu pflegender Index benötig. Des Weiteren ist dieses Verfahren unabhängig gegenüber der Anzahl an Tripeln und dem benötigten Speicherplatz, da ein automatischer Scale-Out-Mechanismus existiert.

5. AUSBLICK

In der weiteren Forschung müssen einige Punkte näher betrachtet werden, die als Motivation zu diesem Ansatz dienen. Hierunter fällt die Einhaltung der garantierten Antwortzeit aller Anfragen bis zu einem vorher definierten Prozentsatz. Zur Unterstützung dieser Forderung ist eine Replikation der Daten denkbar.

Dies führt zum nächsten Problem, der effizienten Replikation. Bis zu diesem Zeitpunkt werden alle Daten nur auf einem Knoten vorgehalten. Stürzt dieser ab, sind all seine Daten verloren und nachfolgende Anfragen können nicht mehr beantwortet werden. Als Ausweg bestünde die Möglichkeit, ein ähnliches Vorgehen umzusetzen, wie es in Amazons Dynamo [4] implementiert ist.

Zur weiteren Performancesteigerung ist es notwendig den LODCache weiter zu optimieren. Es existiert zum Beispiel die Möglichkeit auf einer SandyBridge-CPU eine Schleife direkt im CPU eigenen Loop-Cache abzulegen. Eine Performancesteigerung von über 100 % soll möglich sein. Jedoch ist die Bedingung hierfür, dass alle Instruktionen höchstens 28 μ-Operationen lang sind.

Der oben beschriebene Ansatz zum Mapping von Strings auf ordnungserhaltende Integerwerte muss weiter erforscht werden. Es ist u. a. notwendig, die Operationen zur Reorganisation effizienter anzuordnen. Eine Möglichkeit zur Erhöhung der Auslastung wird außerdem angestrebt. In diesem Zusammenhang soll gleichzeitig ein Verfahren entwickelt werden, um Updates auf bestehende Daten möglichst effizient durchzuführen.

Für die weitere Entwicklung und Evaluierung des Systems wird zukünftig ein Benchmark eingesetzt. Es wurde der Berlin SPARQL Benchmark [6] ausgewählt. Dieser erzeugt akzeptierte Resultate und ist für das Testen von beliebigen SPARQL-Systemen entwickelt worden. Für Skalierungstests kann ein Skalierungsfaktor angepasst werden, der die Anzahl an automatisch erzeugten Tripeln vergrößert.

Die Performanz des Systems wird durch verschiedenartige SPARQL-Anfragen ermittelt, die sich in Äquivalenzklassen einteilen lassen. Den einfachen SPARQL-Anfragen, den Anfragen, die eine Datenmanipulation erfordern sowie den analytischen SPARQL-Anfragen.

6. REFERENCES

[1] Rdf - semantic web standards. *http://www.w3.org/RDF*.

[2] Sparql query language for rdf. *http://www.w3.org/TR/rdf-sparql-query*.

[3] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[5] O. Erling and I. Mikhailov. RDF support in the virtuoso DBMS. In S. Auer, C. Bizer, C. Müller, and A. V. Zhdanova, editors, *CSSW*, volume 113 of *LNI*, pages 59–68. GI, 2007.

[6] FU-Berlin. Berlin sparql benchmark. *http://www4.wiwiss.fu-berlin.de/bizer/berlinsparqlbenchmark*.

[7] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K.-U. Sattler, and J. Umbrich. Data summaries for on-demand queries over linked data. In *WWW*, pages 411–420, 2010.

[8] M. P. Haustein, T. Härder, C. Mathis, and M. W. 0002. Deweyids - the key to fine-grained management of xml documents. *JIDM*, 1(1):147–160, 2010.

[9] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *PVLDB*, 1(1):622–634, 2008.

[10] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In F. T. Leighton and P. W. Shor, editors, *STOC*, pages 654–663. ACM, 1997.

[11] T. Neumann and G. Weikum. RDF-3X: A RISC-Style Engine for RDF. In *VLDB*, Auckland, New Zealand, 2008.

[12] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In G. Alonso, J. A. Blakeley, and A. L. P. Chen, editors, *ICDE*, pages 60–69. IEEE, 2008.

[13] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[14] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. In *VLDB*, Auckland, New Zealand, 2008.

[15] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. In *Proc. First International Workshop on Semantic Web and Databases*, 2003.