TECHNISCHE UNIVERSITÄT
ILMENAU

Institut für Praktische Informatik und Medieninformatik
Fakultät für Informatik und Automatisierung
Fachgebiet Datenbanken und Informationssysteme

MASTERARBEIT

Effiziente In-Memory Verarbeitung von SPARQL-Anfragen auf großen Datenmengen

---

vorgelegt von: Steve Göring

Matrikelnummer: 43952

Betreuer: Prof. Dr.-Ing. Kai-Uwe Sattler
M. Sc. Heiko Betz

Ilmenau, den 2. Dezember 2013

Danksagung

In erster Linie möchte ich mich für die sehr gute Betreuung durch Professor Sattler und Heiko Betz bedanken, da sie mir in allen Phasen dieser Arbeit mit ihrem Rat zur Seite standen. Im Rahmen des Oberseminars konnten viele Probleme und Fragen direkt geklärt werden. Auch war die konstruktive Kritik durch alle Mitarbeiter des Datenbankfachgebiets immer sehr hilfreich.

Gerade für die Korrekturleser stellte meine Arbeit eine große Herausforderung dar. Besonders bedanken möchte ich mich an dieser Stelle für die vielen Hinweise und sprachlichen Verbesserungen von Martin, Philipp, Susi, Rebecca, Rainer, Tim und Rene.

Die letzten sechs Monate waren ein ständiger Wechsel. Viele Freunde standen mir zur Seite und haben mir an einigen Tagen geholfen abzuschalten um somit einen klaren Gedanken zu finden. Ich bin dankbar für die vielen Zusprüche und die Unterstützung in dieser Zeit.

Am Ende möchte ich mich auch bei meiner Mutti bedanken, da sie mich in allen Punkten meines Studiums unterstützt hat. Gerade der Erstellungsprozess war nicht immer einfach, dennoch zeigte sie immer großes Verständnis auch für meinen seltsamen Schlafrhythmus.

„I'm trying to free your mind, Neo.
But I can only show you the door.
You're the one that has to walk through it."

The Matrix

Zusammenfassung

Die Verarbeitung von Linked-Open-Data (LOD) stellt neue Anforderungen an Datenbanksysteme, dabei sind die Datenmengen sehr groß und weisen besondere Eigenschaften auf. Aus diesem Grund haben sich neue Ansätze speziell für LOD etabliert. Moderne Rechnerarchitekturen bieten für Datenbankanwendungen interessante Features (z.B. Parallelisierung, Vektorisierung, NUMA, Caching, ...), welche von wenigen Datenbanksystemen benutzt werden. Durch die Entwicklung von Hauptspeichertechnologien sind In-Memory Datenbanken auch in der Lage große Datenmengen zu verarbeiten. Es wurden spezielle In-Memory LOD-Stores (RDFHDT, Hexastore, BitMat und RDF-3X) betrachtet. Dabei konnte festgestellt werden, dass sie moderner Hardware kaum nutzen bzw. auslasten. Allen gemeinsam ist der Fakt, dass sie Kompression und Indizierung benutzen um die Datenmenge zu reduzieren bzw. effizient zu verarbeiten. Das CameLOD-Projekt ist ein weiterer moderner Ansatz für die LOD-Verarbeitung. Es ist ein komprimierter In-Memory Store und bietet bessere Unterstützung für moderne Hardware. In dieser Arbeit ist CameLOD der Hauptuntersuchungsgegenstand. Zunächst wurde das System mittels VTune analysiert und darauf aufbauend Performanceschwachstellen (Smart-Pointer, Bit-Sets, Hash-Kollisionen) geschlussfolgert. Eine Untersuchung der Kompression fand ebenso statt und es wurde festgestellt, dass durch verschiedene Techniken (FOR, RLE, Wörterbücher) eine Verbesserung erzielt werden kann. In mehreren Mini-Benchmarks mit eigenen Implementierungen wurden die Ansätze zur Effizienzsteigerung bezogen auf Geschwindigkeit und Speicher untersucht. Speziell für die Kompressionstechniken wurden effiziente Implementierungen beschrieben um den Dekompressionsoverhead zu minimieren. Es wurde festgestellt, dass einige Stellen des Systems verbessert werden können. Z.B. verursacht der Einsatz von Smart-Pointer viel Overhead. Abschließend wurde in einer Auswertung ausgewählte Erweiterungen in CameLOD integriert. Ein Benchmark mit 13 Anfragen wurde durchgeführt und als Ergebnis konnte die Anfrageverarbeitung bezogen auf die originale Version verbessert werden. Weiterhin profitieren auch andere Schritte (Aufbau der Datenbasis, Laden der Daten) von CameLOD von den Erweiterungen. Es konnte außerdem gezeigt werden, dass durch den Einsatz der Kompressionstechniken mindestens 40% an Speicher eingespart werden kann.

Abstract

The query processing for Linked-Open-Data (LOD) provides new requirements for database systems. The datasets are quite large and have special properties. Especially for LOD, new approaches were established. Modern computer architectures provide interesting features (e.g. parallelization, vectorization, numa, caching, ...) for database tasks, which are used only by a few systems. Because of new main memory technologies, in-memory database systems are able to handle large datasets. Specific in-memory LOD stores (RDFHDT, Hexastore, BitMat and RDF-3X) were reviewed. It was found out that they barely support modern hardware features. Common to all is the usage of compression and indexing techniques to reduce the datasets respectively process them efficiently. Another modern approach for LOD processing is the CameLOD project. It is a compressed in-memory store with more support of modern hardware. In this thesis, CameLOD is the main study object. First, the system was analysed by using VTune and based on the results performance bottlenecks (smart pointer, bit-sets, hash-collisions) were identified. Also, the compression was analysed and it was identified that with different techniques (FOR, RLE, dictionary) improvements could be realized. With several small benchmarks based on own implementations, the new approaches for better efficiency in speed or memory were evaluated. Efficient implementations for compression techniques were described to minimize the decompression overhead. In conclusion, different parts of the system could be improved. For example the usage of smart pointer produce a lot of overhead. Concluding an evaluation of chosen extensions that were integrated in CameLOD was conducted. A benchmark with 13 queries was executing resulting in the improvement of the query processing time compared to the original version. Furthermore, other steps (building up database, load data) of CameLOD can benefit from these extensions. Additionally, it was shown that the examined compression techniques can save about 40% of memory.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das Teilen von Informationen ist ein Grundgedanke des World Wide Webs (WWWs). Keine andere Technologie hat in den letzten Jahren die Art und Weise wie Menschen mit Informationen umgehen so stark verändert. Im WWW verbinden Hyperlinks verschiedene Dokumente, wodurch sie zueinander in Beziehung gebracht werden. Als nächsten Schritt können nun die Daten miteinander verknüpft werden. Im klassischen WWW werden Daten einfach als Tabellen, Grafiken oder Downloads angeboten und weisen keinen semantischen Bezug zueinander auf. Folglich ist ein aktuelles Konzept zur Weiterentwicklung des WWWs der Aufbau eines semantischen Webs [SHB06]. Informationen und Daten können im WWW nur von Menschen wahrgenommen und in Zusammenhang gebracht werden. Beispielsweise erkennt ein Mensch beim Lesen des Wikipedia Artikels über Ilmenau (vergleiche [Wik]), dass Ilmenau in Thüringen liegt und eine Universitätsstadt ist. Das semantische Web verfolgt den Ansatz solche Zusammenhänge in einer Form darzustellen, die auch von Maschinen verarbeitet werden können.

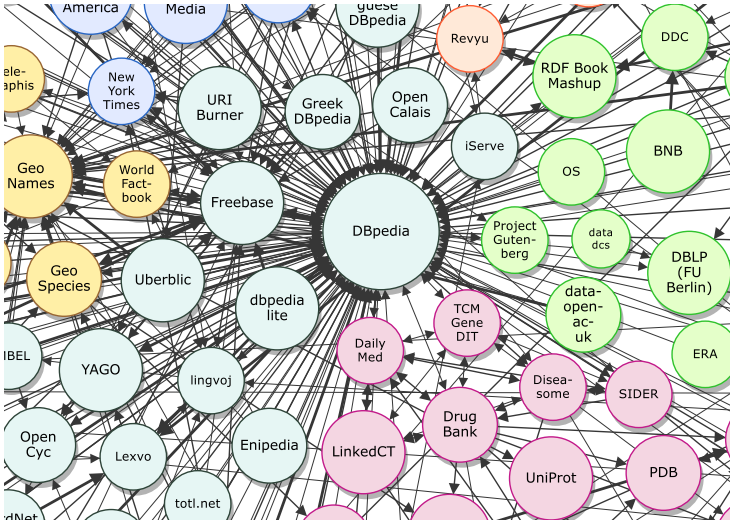

Abbildung 1.1: Ausschnitt aller Linked-Open-Data Quellen [CJ]

Einer der Mitbegründer des WWWs – Tim Berners-Lee – legte in [BHB09] Regeln des semantischen Webs als 'Linked Data principles' fest. Demzufolge sind Linked-Data dadurch gekennzeichnet, dass die Daten durch Universal Resource Identifiers (URIs) identifizierbar sind. Außerdem müssen sie mit dem HTTP-Protokoll erreichbar sein. Weiterhin werden Anfragen mittels SPARQL oder RDF realisiert. Die Anfrageergebnisse enthalten wiederum Links zu weiteren nützlichen Informationen.

Wenn diese Linked-Data frei verfügbar sind, so spricht man von Linked-Open-Data (LOD). Es gibt viele Projekte, welche LOD zur Verfügung stellen. Die Abbildung 1.1 zeigt einen Ausschnitt bekannter LOD-Projekte. DBpedia ist zum Beispiel eines der größten Projekte, welches die semantischen Informationen des Wikipedia-Projekts als Linked-Open-Data zur Verfügung stellt [DBP; Leh+13]. Weiterhin existieren auch noch andere Projekte (beispielsweise Friend-of-a-Friend [FOA]), die LOD zur Verfügung stellen. Insgesamt gibt es ungefähr 62 Milliarden öffentliche RDF-Tripel[1], wobei die englische Version von DBpedia circa 470 Millionen Tripel[2] umfasst.

1.1 Verarbeitung von Linked-Open-Data

Die Verarbeitung von Linked-Open-Data stellt, allein durch die enorme Datenmenge, neue Herausforderungen an Datenbanksysteme. Große Datenmengen werden bei typischen SPARQL-Anfragen ausgewertet. Dabei sind sie meist als RDF-Tripel, bestehend aus Subjekt-Prädikat-Objekt, gespeichert.

Verschiedene Ansätze und Konzepte haben sich für die LOD-Verarbeitung bisher etabliert. Beispielsweise gibt es die Möglichkeit die als RDF-Tripel gespeicherten Datensätze in einer relationalen Datenbank zu speichern und anschließend per Transformation der SPARQL-Anfragen zu relationalen SQL-Anfragen die erforderlichen Ergebnisse zu ermitteln (vergleiche [Har05]). Der Einsatz einer relationalen Datenbank unterliegt dabei aber den speziell auf RDF-Tripeln optimierten Datenbanksystemen, wie zum Beispiel AllegroGraph (vergleiche [Roh+07; w3o]).

Konventionelle relationale Datenbanken sind nur bedingt für RDF-Tripel geeignet, da sie weitere, auch für relationale Datenbanksysteme, wichtige Konzepte (wie zum Beispiel Transaktionsmanagement, Logging, ...) verwenden, welche aber bei der RDF-Verarbeitung keine Rolle spielen. Dadurch wird die Verarbeitungsgeschwindigkeit drastisch reduziert. Hauptsächlich werden bei der LOD-Verarbeitung Leseanfragen an das Datenbanksystem gestellt. Ein weiterer Punkt bei herkömmlichen Datenbanksystemen ist die Notwendigkeit, Daten auf Festspeicher abzulegen und teilweise dort zu verarbeiten. Dagegen kann heutzutage die In-Memory Verarbeitung einen erheblichen Geschwindigkeitsvorteil bringen.

Aktuelle Server-Systeme können bereits sehr große Datenmengen In-Memory verarbeiten, zum Beispiel sind in naher Zukunft Hauptspeicher-Größen von 12 TB und mehr möglich [heib]. Daher ist es unter Umständen gar nicht notwendig, die Daten auf einer Festplatte zu speichern beziehungsweise die Verarbeitung in Teilmengen der Daten stattfinden zu lassen. Ein reines In-Memory Datenbanksystem benötigt auch Möglichkeiten Daten dauerhaft auf Primärspeicher abzulegen (Recovery/Backup-Strategien vergleiche [GS92]).

Im Gegensatz zur herkömmlichen Verarbeitung der Daten, stellt die In-Memory Verarbeitung neue Anforderungen an das Datenbanksystem. Zum Beispiel müssen Datenstrukturen benutzt werden, die sparsam mit dem Hauptspeicher umgehen und nach Möglichkeit Kompressionsverfahren anwenden. Weiterhin soll der Geschwindigkeitsvorteil der In-Memory Verarbeitung nicht verloren gehen. Das bedeutet also ein ständiger Kompromiss aus Geschwindigkeit und Hauptspeicherbedarf.

[1] http://stats.lod2.eu, Stand 11.08.2013, Datenbasis: 870 verschiedene LOD-Projekte
[2] http://wiki.dbpedia.org/Datasets, Stand 11.08.2013, dort beschrieben als Fakten

1.2 Problemstellung und Motivation

Im Rahmen dieser Arbeit sollen für einen bestehenden modernen C++-Prototypen eines RDF-Stores (CameLOD) Performancesteigerungen durch Ausnutzung moderner Hardware-Features realisiert werden. Das CameLOD-Projekt stellt Möglichkeiten zur komprimierten In-Memory Verarbeitung von SPARQL-Leseanfragen bereit.

Zunächst soll eine genaue Analyse des Projektes durchgeführt werden, um Performance- und Speicherengstellen zu finden. Es wurden bereits Vorkehrungen für die parallelisierte Verarbeitung der Daten getroffen, welche genauer untersucht werden sollen. Besonders die verwendeten Operatoren für Scan-, Join- und Filter-Aufgaben sowie benutzten Datenstrukturen sollen im Bezug auf die Möglichkeiten moderner Rechnerarchitekturen (Parallelisierung, Memory-Layout (NUMA), Vektorisierung (SSE), Cache-Awareness, vergleiche [Intd]) analysiert werden.

Aufbauend auf der Analyse sollen kleine Testszenarien betrachtet werden, um das Potential verschiedener Optimierungsansätze abschätzen zu können. Außerdem soll auch ein Vergleich mit dem technisch realisierbaren optimalen Fall durchgeführt werden.

Nach den experimentellen Tests sollen im CameLOD-Projekt an den entsprechenden Stellen Erweiterungen vorgenommen werden, so dass beispielsweise ein Geschwindigkeitszuwachs beim Verarbeiten von Datenmengen des DBpedia Projektes erzielt werden kann. Die Erweiterungen zielen aber nicht nur auf zeiteffizientere Verarbeitung, sondern ebenso auf Speichereffizienz ab. Daher sollen auch Möglichkeiten zur Verbesserung der Kompression und Auslagerung (mittels Memory-Mapped-Files) der Daten betrachtet werden. Besonders wichtig ist weiterhin, dass der Geschwindigkeitszuwachs möglichst speicherplatzschonend ist. Eine Optimierung, die sehr viel Speicher kostet, hätte nur wenig Vorteile für die In-Memory Verarbeitung großer Datenmengen. Grundlegend muss also bei den Optimierungen ein ständiger Kompromiss aus Speichereffizienz und Performance gefunden werden.

Aufbauend auf den implementierten Optimierungen im Projekt soll abschließend eine Evaluierung, welche die Daten des DBpedia Projektes als Grundlage benutzt, durchgeführt werden. Mittels dieser RDF-Daten sollen Mikrobenchmarks auf einem Serversystem ausgeführt und mit der nicht optimierten Variante verglichen werden. Insgesamt soll das CameLOD-Projekt als Ergebnis dieser Arbeit effizienter im Sinne des Speicherplatzbedarfs und der Verarbeitungsgeschwindigkeit werden.

1.3 Aufbau

In Kapitel 2 sollen zunächst einige grundlegende aktuelle Techniken für die In-Memory Verarbeitung von LOD, moderne CPU-Architekturen, Kompressionstechniken und einige Begriffe beschrieben werden. Anschließend wird in Kapitel 3 das CameLOD-Projekt, das Analysewerkzeug und das Referenzsystem, welches für alle Messungen benutzt wird, genau dargestellt. Im Abschnitt 4 wird das CameLOD-System analysiert und Verbesserungsideen werden darauf aufbauend geschlussfolgert. Die Ideen sollen in Kapitel 5 mittels Referenzimplementierung verbesserter Verfahren und Mini-Benchmarks evaluiert und anschließend in das CameLOD-Projekt integriert werden. Am Ende der Arbeit in Kapitel 6 werden mittels Benchmarks einige integrierten Optimierungen mit der ursprünglichen Version des CameLOD-Projekts verglichen. Abschließend folgt in Abschnitt 7 ein Fazit.

Kapitel 2

Stand der Technik

In diesem Kapitel werden kurz die aktuellen Entwicklungen in den Themenbereichen SPARQL und RDF-Verarbeitung, In-Memory Datenbanksysteme, moderne Rechnerarchitekturen und Kompressionsverfahren für Datenbanksysteme beschrieben. Insbesondere sollen bei der Beschreibung auch moderne Ansätze für die jeweiligen Probleme kurz dargestellt werden.

2.1 SPARQL und RDF-Daten

SPARQL und RDF sind zentrale Begriffe des semantischen Webs. SPARQL ist die vom W3C festgelegte Anfragesprache zur Verarbeitung von RDF-Daten[1].

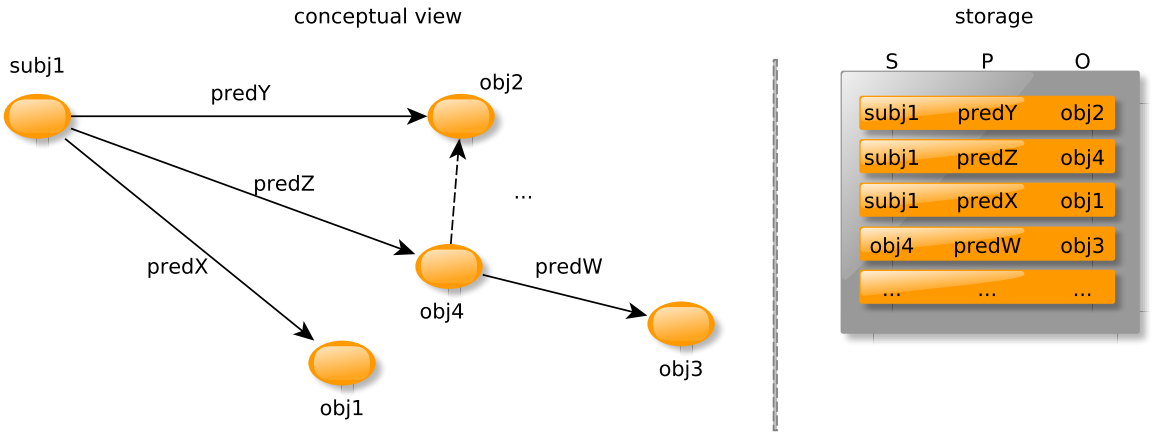

Abbildung 2.1: Strukturieller Aufbau eines RDF-Graphs, mit Bezug zur Speicherung als SPO-Tripel

RDF-Daten können als Graph dargestellt werden und stellen Aussagen der Form Subjekt-Prädikat-Objekt dar. Strukturell ergeben sich dabei ähnliche Graphen wie in Abbildung 2.1 dargestellt. Dabei sind Kantenbezeichnungen immer Prädikate, das bedeutet Beschreibungen, die ein spezielles Subjekt mit einem anderen Objekt in Beziehung setzen.

Ein Ansatz zur Verarbeitung von RDF-Daten ist, sie als globale Tabelle bestehend aus Tripeln zu speichern (vergleiche [Cyg05]). Dabei ist die Tripel-Repräsentation nur eine von vielen Möglichkeiten. Es kann zum Beispiel auch eine Darstellung mittels XML erfolgen [w3c].

[1] http://www.w3.org/standards/semanticweb/query, Stand 03.10.2013

Die Tripel-Darstellung kann direkt aus dem Graphen abgeleitet werden. Jedes Tripel repräsentiert dabei einen Fakt bestehend aus Subjekt, Prädikat und Objekt.

Beispielsweise ist in DBpedia das Tripel

S `<http://dbpedia.org/resource/Star_Trek:_Enterprise>`

P `<http://dbpedia.org/ontology/abstract>`

O "Star Trek: Enterprise (originally titled simply Enterprise...."

zu finden. Als Subjekt wurde ein konkreter Film festgelegt. Das Prädikat beschreibt hingegen, dass im Objektteil eine Kurzfassung des Films dargestellt wird.

Bei der Verarbeitung von RDF-Daten mittels SPARQL Anfragen müssen die Besonderheiten der Daten berücksichtigt werden. Dazu zählen die Tripel-Darstellung, die enormen Datenmengen (vergleiche das DBpedia-Projekt) sowie die globale Sicht auf die Daten.

In [Cyg05] werden neben einer relationalen Algebradefinition für SPARQL auch Transformationen zu SQL beschrieben. Dadurch ist erkenntlich, dass SPARQL-Anfragen auf RDF-Daten mittels relationaler Datenbanken realisiert werden können. Problematisch an diesem Ansatz ist jedoch, dass allgemeine relationale Datenbanksysteme die Kerneigenschaften der RDF-Daten nicht ausnutzen. Daher sollen kurz die besonderen Eigenschaften der RDF-Daten zusammengefasst werden.

2.1.1 Eigenschaften und Anfragen von RDF-Daten

Ein RDF-Graph kann als eine globale Tabelle mit drei Spalten (Subjekt, Prädikat und Objekt) betrachtet werden. Insgesamt gibt es weniger Prädikatwerte als Objektwerte oder Subjektwerte, da sie Eigenschaften darstellen und mehrfache Anwendung finden. Die Prädikatwerte und Subjektwerte weisen, durch ihre Darstellung als URIs, häufig eine Präfixübereinstimmung auf.

Anfragen werden durch Selektion der globalen Tabelle oder per Joins realisiert, für einige Anfragetypen müssen sogenannte Star-Joins verwendet werden. Sie dienen dem Auffinden spezieller Basic Graph Pattern (BGP)[2]. BGPs sind dabei mehrere Tripel-Muster, die im Graphen gefunden werden sollen.

Die globale Tabelle umfasst sehr viele Zeilen, da der Graph in einer Tripeldarstellung gespeichert wird. Es gibt aber auch Varianten, bei denen andere Darstellungen verwendet werden. Grundlegend werden beim Ansatz des semantischen Webs viele Leseanfragen durchgeführt. Daher sind Einfüge- und Update-Anfragen von geringerer Bedeutung.

2.1.2 RDF-Stores

Es gibt bereits eine Vielzahl von RDF-Stores, welche verschiedene technische Ansätze verwenden. In [Ste+09] wurden verschiedene RDF-Datenbankansätze verglichen. Darunter finden sich keine reinen In-Memory Lösungen, aber RDF-Lösungen, die Möglichkeiten bieten, Daten

[2] vergleiche http://www.w3.org/TR/sparql11-query/#GraphPattern, Stand 28.10.2013

In-Memory zu speichern, zum Beispiel Jena[3] und Sesame[4]. Grundlegend können aber auch diese Lösungen insgesamt nicht überzeugen, da sie nur Wege bieten die Daten unkomprimiert In-Memory zu halten, weil sie beispielsweise Schnittstellen zur Verfügung stellen müssen mittels Disk-Speicherung zu arbeiten (über relationale Datenbanksysteme oder eigene Triple-Storages). Daher können die beiden Stores keine Vorteile der reinen In-Memory Verarbeitung ausnutzen, wie etwa eine Kompression der Daten. Demzufolge erfüllen die dort dargestellten RDF-Stores nicht die Anforderungen.

Es gibt noch weitere Projekte welche von Interesse sind, zum Beispiel RDFHDT, Hexastore, BitMat und RDF-3X. Sie sollen im folgenden Abschnitt kurz betrachtet werden.

RDFHDT

Das RDFHDT-Projekt[5] stellt Möglichkeiten der komprimierten Speicherung und Verarbeitung von großen RDF-Datensätzen bereit. Die RDF-Daten werden bei RDFHDT in der Form HDT gespeichert (vergleiche [Fer+13]). HDT steht dabei für die Abkürzung Header, Dictionary sowie Triples und beschreibt die Hauptkomponenten der Speicherung. Der Header beinhaltet dabei Metainformationen für den RDF-Graphen, das Dictionary übernimmt die Aufgabe eines Katalogs für alle Bezeichnungen des Graphs und die Tripel-Komponente stellt eine kompakte Darstellung der gespeicherten Tripel dar. Der Aufbau der komprimierten Datenstruktur erfolgt

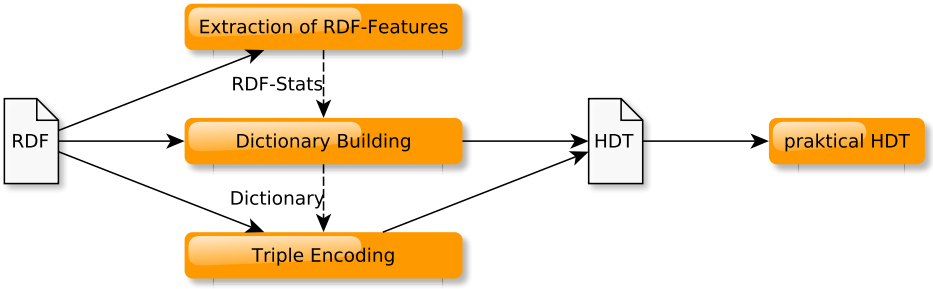

Abbildung 2.2: RDFHDT Aufbau, [Fer+13]

dabei in mehreren Schritten, dargestellt in Grafik 2.2. Zunächst werden RDF-Statistiken erzeugt und anschließend das Wörterbuch aufgebaut. Im letzten Schritt werden die Tripel mit Hilfe des Wörterbuchs dargestellt. Die HDT-Repräsentation der RDF-Daten stellt eine Wörterbuchkompression dar und erzielt somit gute Kompression. Die Kompressionsraten sind nach [Fer+13] sogar besser als herkömmliche Verfahren wie gzip, bzip2 oder ppmdi.

Die Anfrageverarbeitung des HDT-FoQ-Stores (beschrieben in [MAF12]) bei RDFHDT erfolgt In-Memory. Dabei werden beim Laden der Daten verschiedene Indizes aufgebaut, dazu zählen ein OP-S-Index, ein PS-O-Index und ein SP-O-Index. Dadurch können die Anfragen direkt auf den sortierten Indizes ausgeführt werden. Die Operatoren sind nicht parallelisiert und die Anfragen erfolgen auf dem jeweiligen kompletten Index.

[3] zu finden unter http://jena.apache.org, Stand 07.10.2013
[4] zu finden unter http://www.openrdf.org/, Stand 07.10.2013
[5] zu finden unter http://www.rdfhdt.org/, Stand 11.10.2013

In [MAF12] werden Benchmarks mit anderen RDF-Stores dargestellt. Insgesamt zeigt RDFHDF gute Eigenschaften beim Laden und Austausch von RDF-Daten und bei Anfragen kann die dargestellte In-Memory Lösung (HDT-FoQ) Verbesserungen bei großen Datenmengen erzielen.

Hexastore

Der in [WKB08] beschriebene Hexastore ist ein RDF-Store, welcher für jede Sortierreihenfolge (SPO, SOP, PSO, POS, OPS, OSP) einen Index zur Verfügung stellt. Zur Speicherung der Indizes wird ebenfalls ein Wörterbuchansatz verwendet, so dass die Indizes nur noch aus Schlüsseln bestehen. Es werden im Index Listen aufgebaut, welche untereinander geteilt werden. Somit kann gesichert werden, dass insgesamt nicht der 6-fache Speicher benötigt wird.

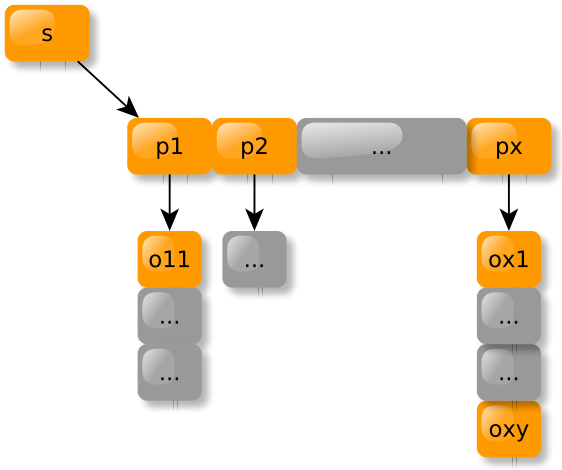

Abbildung 2.3: Hexastore Aufbau des SPO-Index, dargestellt ist ein spezieller Subjekt-Wert (s), [WKB08]

In Abbildung 2.3 ist allgemein die Speicherung des SPO-Index dargestellt. Ein Subjekt-Wert (s) steht mit einer sortierten Prädikatliste ($p_1, ..., p_x$) in Verbindung. Jedes Prädikat ist mit einer Liste von Objekten ($o_{11}, ...$ beziehungsweise $o_{x1}, ..., o_{xy}$) verknüpft. Die Objektlisten werden mit dem PSO-Index geteilt. Analog sind auch die anderen Indizes aufgebaut.

Insgesamt kann durch das Teilen von gleichen Informationen eine gute Kompression erzielt werden, auch wenn 6 Indizes angelegt werden. Im Vergleich zur Verwendung eines Indizes (PSO) benötigt die Hexastore-Variante ungefähr 4 mal so viel Hauptspeicher, kann aber durch die Indizierung einen erheblichen Geschwindigkeitsvorteil erzielen.

In [WKB08] werden keine Aussagen zur Ausnutzung von mehreren CPUs des Testsystems dargestellt (das angegebene Testsystem benutzt nur einem Dual-Core-CPU). Der verwendete Prototyp in der Evaluierung wurde in Python implementiert. Zur genauen Implementierung und den verwendeten Python-Datenstrukturen werden keine Informationen angegeben,. Zusätzlich ist die Implementierung nicht öffentlich zugänglich. Daher kann schlecht abgeschätzt werden, ob im System Parallelisierung oder moderne CPU-Features benutzt werden.

BitMat

Als ein weiterer Ansatz zur Speicherung und Verarbeitung von RDF-Daten soll das BitMat-Projekt betrachtet werden. In [ASH08; Atr+09] wird der Aufbau beschrieben. BitMat ist ein In-Memory Store für RDF-Tripels. Die RDF-Tripel werden als 3-dimensionale Matrix betrachtet (Bit-Cube). Für jedes Prädikat wird eine 2-dimensionale Matrix von Subjekt- und Objektwerten gespeichert.

Ein Bit in der Matrix stellt dabei die Anwesenheit beziehungsweise Abwesenheit des Tripels dar. Durch die Bit-Matrix-Datenstruktur sind Join Operationen mit einfachen Bitoperationen (AND und OR) durchführbar. Zur Kompression der Bit-Vektoren der Matrix benutzt das BitMat-Projekt D-gap [Bit]. D-gap ist eine Form der Run-Length-Encodierung für dünn besetzte Bit-Vektoren. Es können für die Bit-Vektoren gute Kompressionsergebnisse erzielt werden. Beispielsweise wird in [ASH08] ein Datensatz UniProt (mit 200k Tripel) in einer BitMatrix komprimiert und somit ein Speicherbedarf von 1.4 MB erzielt.

RDF-3X

RDF-3x[6] ist der letzte RDF-Ansatz der betrachtet werden soll. In [NW10] wird der Aufbau des Projekts beschrieben. Hauptsächlich wird ein Wörterbuch benutzt und 6 Indizes für die Tripel in komprimierten B^+-Bäumen gespeichert. Nach einem eigenen byte-orientierten Kompressionsschema werden die Tripel im jeweiligen Index gespeichert.

Durch den Index-All Ansatz kann für jedes Muster von Anfrage-Tripeln ein passender Index gefunden werden und die Verarbeitung anschließend auf den sortierten Indizes erfolgen. Operatoren werden hauptsächlich auf den durch das Wörterbuch bereitgestellten Integer-Keys durchgeführt. Spezielle parallelisierte Operatoren sind nicht vorhanden.

Schlussfolgerung

Die vier vorgestellten In-Memory Ansätze zur Speicherung und Verarbeitung von RDF-Daten benutzen alle Indizierung und Kompression. Bei den betrachteten Ansätzen wird kein Fokus auf die Ausnutzung moderner Hardware-Features gelegt.

RDFHDF zeigt zwar gute Kompressionseigenschaften jedoch kann die Anfrageverarbeitung nicht überzeugen. Es ist somit eher als Austauschformat für RDF-Daten geeignet, was auch eines der Kernziele des Projekts ist.

Hexastore ist ein Index-All System und erzeugt einen viel zu großen Overhead bei der Speicherung. Auch stehen keine tiefgründigen Informationen über die Implementierung zur Verfügung. Es stellt insgesamt nur ein experimentelles System dar. Parallelisierung und moderne CPU-Architekturen wurden nicht speziell betrachtet.

BitMat benutzt eine Run-Length-Kodierung um den benötigten Bit-Cube für die Speicherung aller RDF-Tripel zu komprimieren. Die Anfrageverarbeitung kann mittels logischer Bit-Operatoren durchgeführt werden. Es ist nicht erkenntlich, in welcher Form BitMat speziell auf Multi-Core Architekturen optimiert wurde.

[6] zu finden unter http://www.mpi-inf.mpg.de/~neumann/rdf3x/, Stand 13.10.2013

RDF-3X ist auch ein Index-All System. Alle Benchmark-Ergebnisse wurden auf einem Dual-Core System durchgeführt. Aus diesem Grund ist erkenntlich, dass keine Optimierung bezüglich hochgradiger Parallelisierung oder moderner Hardware angedacht ist.

Zusammenfassend stellen die betrachteten Ansätze gute und neue Ideen für die RDF-Verarbeitung bereit, benutzen aber wenige beziehungsweise keine expliziten moderne CPU-Features, wie zum Beispiel Parallelisierung oder Vektorisierung. Die dargestellten RDF-Stores konnten nicht überzeugen, da sie die für diese Arbeit wichtigen Kriterien Kompression, In-Memory Verarbeitung und Verwendung moderne CPU-Features nicht vollständig erfüllen. Die Verarbeitung von RDF-Daten kann aber durch den Einsatz der genannten Kriterien effizienter erfolgen.

2.2 Moderne Rechnerarchitekturen

Die Verarbeitung von großen Datenmengen kann durch moderne Rechnerarchitekturen profitieren. Folglich soll im nächsten Abschnitt kurz auf die Entwicklungen bezüglich der CPU-Architekturen und des Hauptspeichers eingegangen werden.

2.2.1 CPUs

In erster Linie sollen im Folgenden moderne Intel-CPUs und CPU-Konzepte betrachtet werden. Allgemein bieten modernen CPU-Architekturen einen 64-Bit Befehlssatz [Intd; Inte]. Sie sind somit für 64-Bit Operationen optimiert und bieten zusätzlich verschiedene Erweiterungen.

Parallelisierung

Die aktuellen CPU-Technologien sind Multi-Core-Architekturen. Allgemein gibt es dabei verschiedene Ansätze. Zum einen kann ein Multi-Core-System aus mehreren physikalischen CPUs bestehen oder aber es befinden sich mehrere CPU-Kerne auf einer physikalischen CPU. Außerdem sind auch asymmetrische Systeme möglich, bei denen spezielle CPUs für Spezial-Aufgaben bereitgestellt werden. Hauptsächlich soll in dieser Arbeit der Fokus auf symmetrische Multi-Core Architekturen gelegt werden. Der Vorteil dieser Variante ist zum Beispiel die Möglichkeit, dass CPU-Kerne, welche physikalisch in einem Gehäuse sind, sich einen gemeinsamen Cache teilen können (vergleiche Intels i7 [Inte, Abschnitt 2.2.9]).

Im Gegensatz zu mehreren physikalischen CPUs, bietet Intel darüber hinaus auch eine Technologie die Hyper-Threading genannt wird. Sie wurde ursprünglich bei 32 Bit-CPUs eingeführt um die Performance von Multi-Threaded Anwendungen zu verbessern (vergleiche [Inte, Abschnitt 2.2.8]). Durch Hyper-Threading kann ein physikalischer Kern zwei logische Kerne besitzen. Diese teilen sich etwa den System-Bus und die arithmetisch-logische Einheit (ALU).

In [Jar+12] werden mittels Parallelisierungsframeworks Intel-TBB und OpenMP Benchmarks durchgeführt und dabei kann durch einfache Änderungen am Code ein erheblicher Geschwindigkeitszuwachs erzielt werden. Bei OpenMP müssen beispielsweise Compileranweisungen (Pragma Direktiven) im Code eingefügt werden. Dagegen muss bei Intel-TBB ein Ersatz der zu parallelisierenden For-Schleife stattfinden. Bei parallelen Anwendungen können verschiedene

Formen unterschieden werden. Beispielsweise gibt es Datenparallelität und Funktionsparallelität (oder auch Task-Parallelität) (vergleiche [Rei10, Seite 9ff]). Datenparallelität steht für das Konzept, die Daten in Teilmengen zu organisieren und anschließend parallel zu verarbeiten. Funktionsparallelität dagegen benutzt den Ansatz verschiedene unabhängige Tasks parallel auszuführen. Weiterhin gibt es auch noch ein Misch-Konzept (Pipelining), welches Task- und Daten-Parallelität umsetzt.

Allerdings lässt sich nicht jedes Problem mit mehreren parallelen CPUs schneller lösen. Hauptsächlich besteht jedes Problem aus zwei Komponenten, einem parallelisierbaren Anteil und einem streng sequentiellen Anteil (Amdahl's-Law). In [HM08] wird die Gültigkeit von Amdahl's-Law auf moderne CPU-Architekturen überprüft. Insgesamt wird das Softwaremodell von Amdahl durch verschiedene Hardwaremodelle erweitert und es wird gezeigt, dass Amdahl's Law immer noch Gültigkeit besitzt.

Vektorisierung und SIMD

Multimedia-Anwendungen stellen hohe Ansprüche an CPUs. Deswegen gibt es verschiedene Befehlssatzerweiterungen für die Verarbeitung von Multimedia-Daten, welche Vektorisierung benutzen. Bei Intel sind es zum Beispiel die SSE und AVX Erweiterung (vergleiche [Inth]). In zukünftigen Intel-CPUs wird die AVX-512 Erweiterung vorhanden sein (vergleiche [Intg; heia]). AVX-512 bietet zusätzlich zu speziellen Registern noch weitere interessante Erweiterungen, wie etwa Befehle für kryptographische Hash-Funktionen (SHA) oder eine Hardware-Implementierung von Verfahren für Prüfsummen (Intel-CRC32).

Insgesamt bieten die aufgeführten Intel-Erweiterungen in Bezug auf Vektorisierung breitere (zum Beispiel 128 Bit bei SSE, 256 Bit bei AVX oder 512 Bit bei AVX-512) Register und Befehle zur vektoriellen Verarbeitung von weniger breiten Werten, welche in den Registern verpackt sind.

Abbildung 2.4: Single Instruction Multiple Data (SIMD) Beispiel

Daher ist es möglich zum Beispiel im Falle eines 512 Bit Registers, in einem Schritt 8 Long-Werte (jeweils 64 Bit) zu verarbeiten. Das vektorielle Verarbeiten verschiedener Daten wird Single Instruction Multiple Data (SIMD) genannt.

Der SIMD-Ansatz ist schematisch in Abbildung 2.4 dargestellt. Dabei werden 4 Long-Werte $l_1, ...l_4$, welche in einem SSE/AVX Register geladen wurden, mittels des SSE-Befehls $f(x)$ verarbeitet. Exemplarisch kann $f(x)$ dabei eine Addition eines festen Wertes sein. Als Ergebnis wird die Funktion $f(x)$ auf alle Long-Werte angewandt und im SSE-Register gespeichert. Eine Durchführung dieser Operation bedarf dabei nur einer CPU-Instruktion, im Gegensatz zur nicht vektorisierten Ausführung mit 4 Einzel-Operationen.

Die Compiler GCC und ICC bieten gute Unterstützung zur Auto-Vektorisierung (vergleiche [Inta]). Es muss aber beim Programmieren darauf geachtet werden. Zur Überprüfung kann

mittels Log ermittelt werden, an welchen Stellen der Code vektorisiert wurde oder erkannt werden, an welchen Stellen der Code dahingehend verändert werden muss.

In [Jar+12] wurden neben Parallelisierungsansätzen auch Auto-Vektorisierung untersucht und festgestellt, dass durch Vektorisierung eine Geschwindigkeitsverdoppelung im Vergleich zur nicht-vektorisierten Anwendung erzielt werden kann. Vektorisierung kann somit viele Anwendungen beschleunigen. Allerdings kann in zeiteffizienten Systemen eine manuelle Vektorisierung unter Umständen eine noch höhere Geschwindigkeit erreichen, jedoch ist der entstehende Code sehr stark auf die jeweilige Architektur optimiert. Beispielsweise müsste der Code komplett neu angepasst werden, wenn die AVX-512 Erweiterung in Intel-CPUs zur Verfügung steht. Daher stellt die Auto-Vektorisierung einen sehr interessanten Aspekt dar, da der Compiler automatisch für die jeweilige verwendete CPU-Architektur die Vektorisierung durchführt. Für die manuelle Vektorisierung von C/C++-Code, stehen verschiedene Möglichkeiten zur Verfügung. Sie kann etwa über Intrinsic Anweisungen (Befehlsreferenz siehe [Intg]) erfolgen oder über spezielle Vector-Datentypen beziehungsweise per inline Assember Code.

Superskalarität und Pipelining

Heutige CPUs sind superskalar, das bedeutet, dass sie die Möglichkeit bieten mehrere unabhängige Anweisungen gleichzeitig auszuführen (vergleiche [Jar+12]). Superskalarität und Pipelining kann nicht direkt durch den Programmierer beeinflusst werden.

Cache

Um den Zugriff auf Daten und Anweisungen zu beschleunigen bieten moderne CPUs verschiedene Caches (vergleiche [Inte, Abschnitt 2.2.9] und [Dre07, Seite 13ff]). Dabei gibt es verschiedene Cache-Level, typisch sind 3-Level: L1, L2 und L3, welche eine Hierarchie bilden. In Abbildung 2.5 ist eine mögliche L3-Cache-Hierarchie dargestellt. L1i ist dabei ein Cache

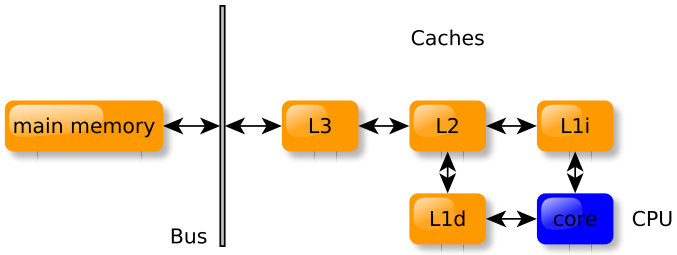

Abbildung 2.5: einfache Level-3-Cache-Hierarchie (nach [Dre07, Seite 15]), L1 ist geteilt in Daten-Cache (L1d) und Instruktions-Cache (L1i)

für CPU-Instruktionen und L1d für Daten. Typischerweise ist der L1-Cache schneller und kleiner als der L2- beziehungsweise L3-Cache. In einigen Multi-Core Architekturen (beispielsweise Intels i7 vergleiche [Inte, Abschnitt 2.2.9]) teilen sich sogar mehrere logische CPUs einen L1 oder L2 Cache. Daher können, ohne explizites Laden aus dem Hauptspeicher, Daten zwischen mehreren Kernen parallel verarbeitet werden.

So besitzt Intels Core i7 - 2600K (vergleiche [Inte, Abschnitt 2.3]) einen 64 KB großen L1 Cache, 256 KB großen L2 Cache und einen für alle Kerne geteilten L3 Cache mit 8 MB.

Virtualisierung und weitere Erweiterungen

Virtualisierung ist für die Anwendung im Sinne eines effizienten Datenbanksystems unwichtig. Aber weitere Erweiterungen wie AES-NI [Intf] oder Intel-CRC32 können durchaus in Datenbanksystemen Anwendung finden. Die AES-NI-Erweiterung kann benutzt werden um Hash-Funktionen mit geringen Kollisionen zu konstruieren.

Fazit

Insgesamt können Datenbanksysteme von modernen CPU-Architekturen profitieren. Anfrageverarbeitung kann durch Multi-Core CPUs parallel stattfinden. Mittels Vektorisierung können Prädikatauswertungen beschleunigt werden. Auch Caches und Cache-Hierarchien können die Verarbeitung positiv beeinflussen. Wie in [BMK99] beschrieben stellen Hauptspeicher Zugriffe für Datenbanksysteme einen Flaschenhals dar und mittels CPU-Cache können sie reduziert werden. Auch weniger direkt einsetzbare Befehlserweiterungen können im Datenbankfokus Verbesserungen erbringen, beispielsweise Vektorisierung.

2.2.2 Hauptspeicher

Nicht nur die CPU-Architekturen wurden in den letzten Jahren weiterentwickelt, sondern auch der Hauptspeicher durchläuft zunehmend eine Verbesserung. Insgesamt kann festgestellt werden, dass in aktuellen Systemen sehr viel Hauptspeicher vorhanden ist, beispielsweise bis zu 12 TB (vergleiche [heib]). Im Bezug auf die Datenverarbeitung bieten sich somit neue Möglichkeiten an. Die Datenmengen die typischerweise verarbeitet werden können in modernen Systemen meist In-Memory gehalten werden.

Die In-Memory Verarbeitung stellt somit eine neue Möglichkeit dar. Auch wenn Hauptspeicher in der Regel sehr schnell im Vergleich zu disk-Speicher ist, wurde er in [BMK99] sogar als Flaschenhals bezeichnet. Gegen diesen Flaschenhals gibt es auch einige Konzepte, zum Beispiel der bereits erwähnte CPU-Cache.

NUMA und UMA

In klassischen Systemen wird ein gleicher (uniformer) Zugriff auf den gesamten Hauptspeicher von jeder CPU aus realisiert (UMA). Dagegen gibt es aber in aktuellen hoch-parallelen Systemen mit einer Vielzahl von CPU-Kernen auch die Möglichkeit mittels NUMA auf den Hauptspeicher zuzugreifen (vergleiche [Dre07, Seite 43ff]). Grundlegend wird bei NUMA der Speicher in zwei Typen eingeteilt (vergleiche Abbildung 2.6). Aus der Sicht einer CPU gibt es lokalen Speicher und entfernten Speicher (remote memory). Der Zugriff auf den lokalen Speicher kann sehr effizient erfolgen und wird daher zur lokalen Verarbeitung der aktuellen Daten des CPUs genutzt. Dagegen ist der remote memory Hauptspeicher eines anderen CPU-Kerns.

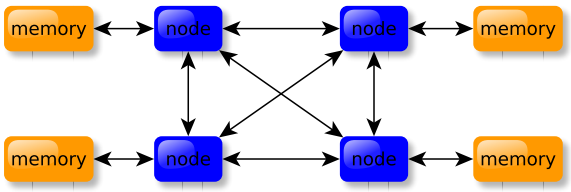

Abbildung 2.6: NUMA-System, 4 vollvermaschte Numa Nodes

Es müssen folglich beim Zugriff die Daten vorher übertragen werden. Mehrere CPU-Kerne werden zu einem NUMA-Node zusammengefasst und teilen sich einen lokalen Speicherbereich. Zwischen den verschiedenen NUMA-nodes gibt es Kommunikationspfade, wobei nicht zwingend eine Vollvermaschung nötig ist. In Abbildung 2.6 ist ein mögliches NUMA-System dargestellt. Es besteht aus 4 NUMA-Nodes, welche vollvermascht sind. Jeder NUMA-Node besitzt seinen eigenen lokalen Speicher und kann mehrere CPU-Kerne enthalten. Die verschiedenen Zugriffszeiten und Topologien sind Faktoren, welche bei der Entwicklung von hochparallelen Anwendungen beachtet werden.

Für die Realisierung von NUMA in Anwendungen gibt es verschiedene Ansätze. Zum einen kann es manuell über libnuma (vergleiche [Dre07, Seite 72ff]) oder über verschiedene Frameworks benutzt werden. In [Ros+13] wird ein NUMA-aware-Task-Scheduler beschrieben (Nas). Es wurde experimentell festgestellt, dass beispielsweise ein Programm zum Lösen von Gleichungen durch Nas beschleunigt werden kann.

In [KSL13] wird dargestellt, dass auch Datenbanksysteme von NUMA profitieren können. Dabei wurden zunächst synthetische Tests bezüglich der Zugriffe auf den Hauptspeicher in einem NUMA-System durchgeführt. Da Datenbanksysteme generell eine Teilung der Daten vornehmen, können sie durch den Einsatz von NUMA mehr Performance erreichen. Es wurde experimentell MySQL mit In-Memory Storage angepasst und festgestellt, dass durch den Einsatz von NUMA ein Geschwindigkeitszuwachs von 10%-15% erzielt werden kann.

Persistenter Speicher

Gerade im Bezug zu In-Memory Systemen stellt persistenter Speicher eine interessante Entwicklung dar. Persistenter Speicher bietet die Möglichkeit Datenstrukturen im Hauptspeicher anzulegen und dabei ohne Serialisierung für Festspeicher dauerhaft zu speichern.

In [VTS11] wird Mnemosyne beschrieben. Es handelt sich dabei um ein einfaches Interface zur Programmierung mit persistenten Speicher. Dabei werden zwei Hauptprobleme untersucht. Zum einen wie solcher Speicher erstellt und verwaltet wird und zum anderen wie Konsistenz im Fehlerfall gesichert wird, da sonst bei Systemfehlern ungültige Daten im System vorhanden sind. Es können globale persistente Daten mit dem Schlüsselwort 'pstatic' oder dynamisch angelegt werden. Außerdem werden Methoden zum direkten Verändern der persistenten Variablen und Datenstrukturen mittels eines leichtgewichtigen Transaktions-Mechanismus zur Verfügung gestellt. In Benchmarks wurde gezeigt, dass Mnemosyne wesentlich schneller als beispielsweise Boost-Serialisierung ist. Mnemosyne stellt insgesamt nur eine Schnittstelle dar um persistenten Speicher (zum Beispiel NVRAM oder Phase-Change Memory (PCM)) effizient einzusetzen.

Dagegen wird in [Gue+12] eine Softwareimplementierung von persistentem Speicher (SoftPM) dargestellt. Das bedeutet, es wird herkömmlicher Speicher um Konzepte zur Persistenz erweitert. SoftPM besteht im Kern aus zwei Komponenten (vergleiche Abbildung 2.7). Die erste

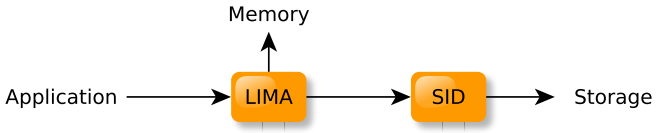

Abbildung 2.7: SoftPM Architektur, nach [Gue+12]

Komponente ist 'Location Independent Memory Allocator' (LIMA) und die zweite 'Storage-optimized I/O Driver' (SID). LIMA ist für die Verwaltung von Containern persistenter Daten verantwortlich, die aus mehreren Memory Pages bestehen, verantwortlich. LIMA besteht aus den Modulen: Container-Manager, Write-Handler, Discovery & Allocator sowie Flusher. Der Flusher ist für das Senden der veränderten Chunks zur SID Komponente verantwortlich. SID ist für das automatische Speichern der Daten auf persistenten Speicher zuständig. Es arbeitet dabei transaktionsartig und bietet verschiedene Module für eine Reihe von Speichertypen (zum Beispiel SSD oder HDD).

Um SoftPM zu nutzen müssen in einer Anwendung nur die nötigen Speicherallokationen, auf die von SoftPM zur Verfügung gestellten, geändert werden. Insgesamt wird gezeigt, dass 'memcachedb' und 'sqlite3' (mit In-Memory Speicherung), deutlich im Vergleich zur Serialisierung der Datenstrukturen, beschleunigt werden können (bei memcachedb bis zu 10 mal schneller).

2.3 In-Memory Datenbanksysteme

Durch die ständige Entwicklung von neuen Hauptspeichertechnologien sind In-Memory Datenbanken ein aktueller Forschungsgegenstand (siehe [heic]). Hauptsächlich unterscheiden sie sich nur in der Speicherung der Daten (vergleiche [GS92]). Bei einem In-Memory Datenbanksystem (IMDB) werden dauerhaft alle Daten im Hauptspeicher gehalten und eine permanente Kopie (Dump) auf Festspeicher gesichert. Dagegen werden bei herkömmlichen disk-basierten Datenbanksystemen die Daten nur im Hauptspeicher verarbeitet. Demzufolge zuerst von Festplatte geladen. Der Hauptspeicher dient somit nur als Cache zur schnelleren Verarbeitung. Klassisch besitzt daher eine disk-basierte Datenbank einen Buffer-Manager, welcher zur Verwaltung der bereits geladenen Seiten beziehungsweise Objekte einer Datenbank dient. Folglich bedeutet für IMDBs das dauerhafte Halten aller Daten im Speicher, dass die Rechnerarchitektur genügend Hauptspeicher zur Verfügung stellen muss.

Es gibt verschiedene Methoden Persistenz in IMDBs zu realisieren (vergleiche [GS92]). Ähnlich wie in herkömmlichen Datenbanksystemen können Checkpoints oder explizite Backups/Logs genutzt werden. Speziell bei IMDBs werden diese Techniken aber nur zur Erzeugung einer dauerhaften Kopie verwendet. Transaktionen benötigen daher keinen einzigen Festplattenzugriff. Auch zur Wiederherstellung der In-Memory Daten wird ein Backup benötigt, da der Hauptspeicher in der Regel flüchtig ist (abgesehen von persistenten Hauptspeichertechniken).

Neben den bereits genannten In-Memory RDF-Stores sollen an dieser Stelle die In-Memory Datenbanken SAP HANA und MonetDB kurz betrachtet werden.

2.3.1 SAP HANA

SAP HANA ist eine In-Memory Datenbank für herkömmliche transaktionale SQL-Anwendungen und für spezielle SAP-Anwendungen (vergleiche [Fär+12]). Ein Schwerpunkt des Systems liegt auf Parallelisierung bezüglich mehrerer Threads, CPU-Kerne oder bis zu komplett verteilten Systemen. Im Kern besteht SAP HANA aus mehreren In-Memory Verarbeitungs-Engines. Es bietet verschiedene Engines für relationale, Graph-, und Text-Daten. Dabei werden alle Daten so lange komplett im Hauptspeicher gehalten, wie Speicher zur Verfügung steht. Alle Datenstrukturen sind für cache-effiziente Verarbeitung optimiert und werden mittels verschiedener Techniken komprimiert. Falls nicht genügend Hauptspeicher vorhanden ist, werden Teile ausgelagert, beispielsweise einzelne Tabellen oder Teiltabellen. Sie werden automatisch in den Hauptspeicher geladen, wenn sie von der Anwendung benötigt werden. SAP HANA bietet verschiedene Schnittstellen für den Anwender, beispielsweise SQL und MDX. Zur persistenten Speicherung sind Backup und Recovery-Strategien basierend auf Logging und Sicherungspunkten vorhanden.

2.3.2 MonetDB

MonetDB stellt eine weitere bekannte relationale In-Memory Datenbank dar. Primär wurde MonetDB für Data-Warehouse Anwendungen entwickelt, somit für die Verarbeitung großer Datenmengen. In [Idr+12] wird die Architektur beschrieben. MonetDB ist ein column-Store, das bedeutet die Daten werden spaltenweise gespeichert und verarbeitet. Sie werden dabei in BAT-Strukturen (Binary Association Table) abgelegt. Mittels Memory-Mapping wird die Speicherung der BATs auf Festplatte realisiert, daher werden die Daten im Hauptspeicher und auf Festplatte in der gleichen Art und Weise organisiert. Anfragen werden cache-effizient (mittels Verfahren, wie zum Beispiel partitionierte Hash-Joins) und vektorisiert durchgeführt (wenn die X100 Erweiterung verwendet wird). Für die Anfrageoptimierung werden im Kostenmodell auch Speicherzugriffe einberechnet. In allen Schritten der Anfrageverarbeitung werden die Daten spaltenweise verarbeitet. Erst bei einer Ausgabe beziehungsweise dem Senden der Ergebnisse zum Clienten werden die Ergebnistupel konstruiert.

Da MonetDB modular aufgebaut ist gibt es einige Erweiterungen, wie beispielsweise einen Anfragekern (X100, vergleiche [Zuk+05]), welcher vektorisiert und komprimiert arbeitet. Es können auch Graphdaten verarbeitet werden, weiterhin gibt es auch Erweiterungen bezüglich adaptiver Indexstrukturen, Selbstorganisation/-optimierung, Wiederverwendung von Ergebnissen und andere (vergleiche [Idr+12]).

2.4 Kompressionstechniken

Kompressionsverfahren finden in verschiedenen Bereichen Anwendung. In Bezug auf Datenbanksysteme stehen dabei meist Indexkompressionstechniken im Vordergrund. Bei Betrachtung reiner In-Memory Datenbanken kann Kompression auch an anderen Stellen verwendet werden, da sie in diesem Falle benutzt wird, um den wichtigen und begrenzten Hauptspeicher zu sparen. In [Kru+12] wird beispielsweise untersucht wie durch den Einsatz einfacher Kompressionstechniken in In-Memory Datenbanken die Performance verbessert werden kann.

Insgesamt wird festgestellt, dass durch Kompression cache-effizientere und somit schnellere Zugriffe erzielt werden können. Auch in [Lem+10] wurden Kompressionstechniken für den Einsatz in Datenbanksystemen betrachtet. Besonders wichtig ist dabei, dass die Techniken einen möglichst geringen Dekompressionsaufwand verursachen. Außerdem wurden Optimierungen für Datenbankoperatoren beschrieben, welche direkt auf den komprimierten Daten arbeiten. Datenbanksysteme können somit von Kompression profitieren, aber die Kompressionsart muss sorgfältig ausgesucht werden.

Es gibt viele verschiedene Kompressionstechniken, daher soll im folgenden Abschnitt nur eine Auswahl von allgemeinen Kompressionsschemata betrachtet werden. Dazu zählen Wörterbuchkompression, Entropiekodierung und mehrere Integer-Sequenz-Verfahren. Die Techniken sollen besonders einen geringen Dekompressions-Overhead besitzen.

2.4.1 Wörterbücher

Wörterbücher können zur Kompression von Zeichenketten benutzt werden (vergleiche [Sal07, Kapitel 3, Seite 173]). Zum Beispiel eignen sie sich um natürlichsprachliche Texte zu komprimieren, da in solchen Texten häufig Wörter mehrfach auftreten. Auch im Fokus auf RDF-Daten können Wörterbücher zur Kompression eingesetzt werden, da sie eine Tripel-Darstellung mittels Integer-Keys ermöglichen.

Ein Wörterbuch D sei dabei eine Datenstruktur $D : s \mapsto x$, mit $s =$ Menge der Zeichenketten sowie $x =$ Menge der Schlüsselwerte. Das Wörterbuch kann beispielsweise jedes Wort (oder jeweils n Buchstaben) eines Textes aufnehmen und anschließend wird der Text durch die generierten Schlüsselwerte repräsentiert. Zusätzlich ist auch das Speichern der einzelnen Zeichenketten mit der Abbildung des Wörterbuchs nötig.

Wörterbücher können beispielsweise mit Hash-Tabellen implementiert werden. Nachteilig an der Hash-Tabellen-Implementierung ist, dass etwa keine Präfixsuchen durchgeführt werden können. Außerdem verursachen Hash-Kollisionen unnötige Zeichenkettenvergleiche.

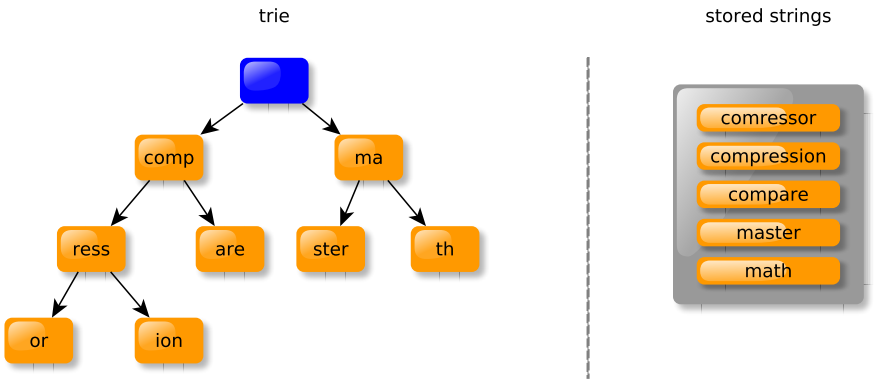

Abbildung 2.8: Trie-Beispiel, dargestellt sind die Wörter: compressor, compression,compare, master,math

Eine weitere Möglichkeit Wörterbücher zu realisieren sind Tries (oder auch Prefixbäume beziehungsweise Radix-Tree genannt). In Abbildung 2.8 ist beispielsweise ein kleiner Trie dargestellt. Es wurden die Wörter 'compressor', 'compression','compare', 'master' und 'math' im Trie ab-

gelegt. Ein Teil der Wörter weist dabei ein gemeinsames Präfix auf, so ergeben sie die Präfixe 'comp' und 'ma'.

Bei einer einfachen Speicherung der Wörter in einer Liste würden alle Präfixe der Wörter mehrfach abgespeichert werden. Im Trie dagegen ist erkennbar, dass die Präfixe nur einmal gespeichert werden. Dagegen müssen aber für die Baumstruktur verschiedene Pointer verwaltet werden. Eine Pfadkompression ist daher zwingend erforderlich, weil sonst alle Buchstaben einzeln in jeweils einem Knoten gespeichert werden. Mittels Trie können die herkömmlichen Wörterbuchoperatoren Lookup, Insert, Modify und Delete realisiert werden. Darüber hinaus ist aber auch eine Präfixabfrage möglich, das heißt es können Anfragen zur Suche von Zeichenketten mit einem bestimmten Präfix durchgeführt werden. In [Bri+11] werden verschiedene Verfahren zur Kompression von String-Wörterbücher verglichen. Bei schnellen Lookup-Operationen kann insgesamt bis zu 20% Speicher eingespart werden. Hauptsächlich werden dort keine Trie Implementierungen betrachtet, sondern verschiedene Techniken (beispielsweise Re-Pair: das Ersetzen von häufigen Buchstaben-Paaren, oder Huffman/Hu-Tucker-Kodierung).

In [LKN13] wird eine adaptive Radix-Tree (Trie) Implementierung für den Einsatz in In-Memory Datenbanken dargestellt. Im Kern ist die Implementierung erweitert worden um die automatische Reduktion der Baumtiefe (mittels Pfadkompression) und dem Löschen von unnötig langen Pfaden zu einzelnen Blättern. Außerdem werden effiziente Einfüge- und Löschoperationen für diesen Trie beschrieben. Mittels dynamischer Wahl von kompakten Datenstrukturen kann die Speichereffizienz der Trie Implementierung im Vergleich zu herkömmlichen Tries verbessert werden. Dazu werden verschiedene Knotentypen zum Speichern von 4, 16, 48 und 256 Kindern bereitgestellt. Durch Benchmarks wird gezeigt, dass die Trie-Datenstruktur vergleichbar mit einer Hash-Tabellen-Implementierung bezüglich der Performance ist. Außerdem wurde die Datenstruktur in das In-Memory Datenbanksystem HyPer (vergleiche [KN11]) integriert und gezeigt, dass sie durchaus herkömmliche Indexstrukturen ersetzen kann.

2.4.2 Entropiekodierung

Entropiekodierung ist ein bekanntes Verfahren zur Kompression. Ein algorithmischer Vertreter ist zum Beispiel das Huffman-Verfahren (genau beschrieben in [Sal07, Kapitel 2, Seite 74ff]).

Beim allgemeinen Huffman-Verfahren werden die Häufigkeiten der verwendeten Buchstaben des Textes ermittelt und anhand dessen werden die Code-Wörter berechnet. Dabei wird dem häufigsten Buchstaben das kürzeste Code-Wort zugeordnet. Die Code-Wörter bilden einen Präfixbaum.

Allgemein besteht das Huffman-Verfahren aus zwei Schritten. Im ersten Schritt wird der Text analysiert (absteigend sortierte Häufigkeiten der Buchstaben) und der Präfixbaum aufgebaut. Im zweiten Schritt erfolgt die Kodierung des gespeicherten Textes mit Hilfe des Präfixbaums. Zur Dekompression ist neben dem kodierten Text auch der Präfixbaum nötig, an dieser Stelle kann beispielsweise auch ein einheitlicher statischer Präfixbaum benutzt werden.

Das reine Huffman-Verfahren ist statisch, das bedeutet, dass der Text vorher komplett vorhanden sein muss. Es gibt aber auch adaptive Huffman-Varianten, bei denen ohne vollständiges Wissen über den Text ein Präfixcode erzeugt wird. Es sind auch Varianten, bei denen keine vollständige Sortierung der Häufigkeiten nötig ist, möglich (vergleiche [KS11]).

Das Huffman-Verfahren beziehungsweise Entropiekodierungen können zwar enorm gute Kompressionsraten erzielen, jedoch bieten sie auf die komprimierten Werte keinen wahlfreien Zugriff oder nur mit Einschränkungen, folglich ist die Dekompression aufwändiger.

2.4.3 Integer-Sequenzen

Indexstrukturen in Datenbanksystemen können als Integer-Sequenzen aufgefasst werden. Für solche Sequenzen gibt es verschiedene Ansätze zur Kompression. Eine einfache Änderung der Kodierung kann bereits gute Raten erzielen. In [WZ99] sind einige Kodierungsverfahren (Golomb, Elias Gamma, Elias Delta, Variable-Byte) für den Einsatz zur Integer-Kompression dargestellt. Integer-Kompression kann daher bei disk-basierten Datenbanksystemen nützlich sein um Festplattenzugriffe zu reduzieren. Es gibt aber noch weitere Verfahren die besonders im Bezug auf Dekompression und wahlfreien Zugriff überzeugen können. Aus diesem Grund sollen nun zwei Varianten betrachtet werden. Zunächst soll das Frame of Reference-Verfahren und anschließend die Run-Length-Kodierung betrachtet werden.

Frame of Reference-Varianten

Das Frame of Reference (FOR)-Verfahren stellt ein einfaches Kompressionsschema dar. Daher ist es auch Grundlage verschiedener erweiterter Verfahren (vergleiche [Zuk+06; YDS09]). Häufig wird es mit einer Delta-Kompression verknüpft (zum Beispiel PFOR-Delta in [Zuk+06]). Weiterhin wird in [LB13] ein FOR-Verfahren, welches mittels Vektorisierung teilweise doppelt so schnell wie andere FOR-Varianten ist, beschrieben.

Die Grundidee des FOR-Verfahren lässt sich in zwei Schritten darstellen. Zunächst sei eine Integer-Sequenz $S = [....]$ gegeben, welche komprimiert werden soll. Bei einer unkomprimierten Speicherung würde für die Sequenz $|S| \cdot container$ (mit $container = sizeof(Long) = 64\ Bit$) Speicher benötigt. Beim FOR wird die Eingabesequenz in Teile (Frames) fester beziehungsweise bekannter Größe eingeteilt. Für jeden Sequenzblock S_i wird anschließend das Minimum min (oder je nach Verfahren ein anderer Referenzwert) bestimmt, welcher explizit gespeichert werden muss. Anschließend wird ein neuer Block C_i mit

$$C_i = [s - min \mid s \in S_i]$$

gespeichert. Dabei besteht C_i nicht aus $Long$-Werten, sondern aus dem kleinsten möglichen Speicherbedarf, der zur Speicherung der ermittelten Werte benötigt wird. Anhand

$$\lceil \log_2(max\{c \mid c \in C_i\} + 1) \rceil$$

kann der Speicherbedarf eines Elements von C_i berechnet werden.

Besonders interessant ist nicht nur die Kompression, sondern für den späteren Einsatz auch die Dekompression. Beim reinen FOR-Verfahren erfolgt die Dekompression in wenigen Schritten. Zunächst muss der jeweilige Frame bestimmt werden, was direkt durch die feste Größe realisiert werden kann. Anschließend erfolgt ein Speicherzugriff und der benötigte Speicherbereich für ein Element wird gelesen. Für die Rekonstruktion zum Originalwert ist nur noch eine Addition des Referenzwertes (min) erforderlich.

Um einen Einblick in die Variationen der FOR-Methode zu geben, sollen die Besonderheiten des in [Zuk+06] dargestellten PFOR-Verfahren kurz betrachtet werden.

PFOR. PFOR steht für Patched Frame-of-Reference. Bei PFOR werden zwei Klassen von Werten unterschieden zum einen Ausnahmen und zum anderen kodierte Werte. Die kodierten Werte sind als Integer-Zahlen mit einer Bit-Breite von b, welche sich im Bereich von $1 \leq b \leq 24$ befindet, dargestellt. Innerhalb eines Blocks ist die Bit-Breite konstant. Die Ausnahmen werden unkomprimiert gespeichert, da sie seltene Ausreißer darstellen. Der Referenzwert bei PFOR ist daher nicht zwingend das Minimum, da es auch Ausnahmen unterhalb des Referenzwertes geben kann. In [Zuk+06] wird auch eine effiziente Implementierung für die Kompression und Dekompression dargestellt, welche loop-unrolling benutzt und sehr cache-effizient ist.

Run-Length-Encoding

Die Run-Length-Encoding (RLE) stellt ein weiteres Verfahren zur Kompression von Integer-Sequenzen dar. Die Hauptidee ist (vergleiche [Sal07, Kapitel 1, Seite 22ff]), dass in einer Integer-Sequenz $S = [...]$ beispielsweise

$S = [bbbbbaaacccddddfffff]$, mit a, b, c, d, f beliebige Integer Zahlen

häufig Blöcke von gleichen Zahlen auftreten. Die Blöcke können effizienter gespeichert werden, beispielsweise in dem die Anzahl an Wiederholungen eines Elementes gespeichert wird. Für das Beispiel ergibt sich:

$C = [5b3a3c4d4f]$

Grundlegend bedeutet das für eine Integer-Sequenz mit Blöcken der Länge 1, dass sie schlecht komprimiert werden kann. Auch ein wahlfreier Zugriff auf die unkomprimierten Elemente ist nicht direkt möglich.

Ein Beispiel für ein RLE-Verfahren für Bit-Vektoren stellt das bereits erwähnte D-gap dar (vergleiche [Bit]). Run-Length-Encoding allein oder auch in Kombination mit anderen Verfahren (zum Beispiel mit Erkennung von Wortblöcken) finden in vielen Kompressionsalgorithmen Anwendung.

2.4.4 Fazit

Die dargestellten Kompressionstechniken können teilweise für die In-Memory Verarbeitung eines Datenbanksystems benutzt werden. Besonders interessant sind dabei die Run-Length-Kodierung und das Frame-Of-Reference Verfahren, da die Dekompression mit wenig Overhead implementiert werden kann. Außerdem kann durch den Einsatz von effizient implementierten Wörterbüchern Speicher gespart werden.

2.5 Zusammenfassung

Es wurden zunächst die Besonderheiten der RDF-Verarbeitung herauskristallisiert und anschließend folgte die Betrachtung verschiedener moderner Ansätze. Die vier vorgestellten In-Memory RDF-Stores (RDFHDT, Hexastore, BitMat und RDF-3X) bieten zwar interessante Ideen, können aber insgesamt im Bezug auf moderne Rechnerarchitekturen nicht überzeugen. Denn heutige Rechnerarchitekturen bieten einige nützliche Eigenschaften für Datenbanksysteme. Beispielsweise Befehlssatzerweiterungen (Vektorisierung), Caches, Parallelisierung oder neue Befehle. Durch die neuen Entwicklungen im Bereich der Hauptspeichertechnologie sind In-Memory Systeme zur Verarbeitung von großen Datenmengen keine Illusion mehr. Effizienter Hauptspeicherzugriff kann mittels NUMA erfolgen. Auch persistenter Memory kann für In-Memory Datenbanksysteme interessant sein. Der Aufbau von IMDBs wurde dargestellt. Wichtig für IMDBs ist weiterhin ein effizienter Zugriff auf den Speicher, denn Hauptspeicher ist im Vergleich zu Cache sehr langsam. Daher sind auch Kompressionsschemata interessant. Hauptsächlich müssen die Kompressionsverfahren effiziente Zugriffe auf die komprimierten Daten liefern, das bedeutet beispielsweise, dass eine explizite Dekompression aller Daten nicht in Betracht kommt. Es wurden unter diesen Kriterien verschiedene Kompressionsschemata, Wörterbücher, Entropiekodierung und Kompression von Integer-Sequenzen, betrachtet. Einfache Kompressionsalgorithmen können durchaus in Datenbanksystemen Anwendung finden und den Zugriff auf Hauptspeicher effizienter gestalten.

Kapitel 3

Grundlagen

Zunächst müssen einige Grundlagen erklärt werden. Dazu zählt die Funktionsweise des CameLOD-Projekts. Weiterhin werden das für die Analyse verwendete Profiling-Tool und das Referenzsystem für die Messungen beschrieben.

3.1 CameLOD

In [HS13] ist der Aufbau von CameLOD ausführlich dargestellt. Konzeptionell sollen kurz die wichtigsten Merkmale des Systems betrachtet werden. CameLOD ist eine komprimierte In-Memory Datenbank zur Verarbeitung von RDF-Daten und ist der Hauptuntersuchungsgegenstand in dieser Arbeit.

3.1.1 Ablauf

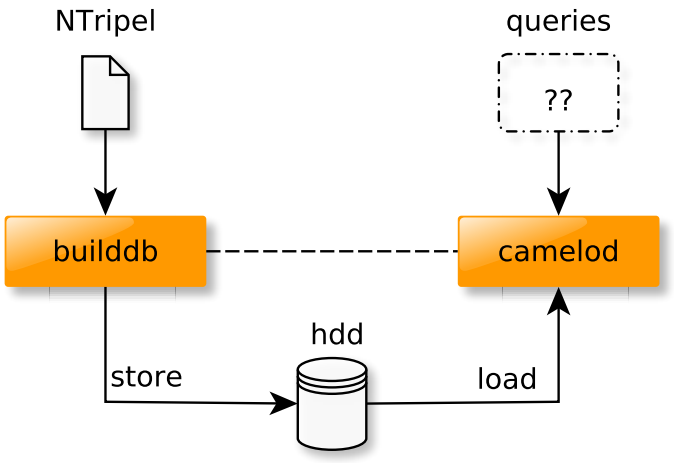

Abbildung 3.1: Systematischer Ablauf beim Arbeiten mit dem CameLOD Projekt

Für das Arbeiten mit CameLOD ist ein zweigeteilter Ablauf (dargestellt in Abbildung 3.1) nötig. Zunächst bedarf es einer Datenbasis in Form einer NTripel-Datei, welche im Builddb-Schritt geladen wird. In diesem Schritt werden die Eingabedaten in das für CameLOD benötigte Format

konvertiert. Der Vorgang stellt die eigentliche Kompression dar und ist zwingend erforderlich. Je nach Datenbasis benötigt dieser Schritt viel Zeit.

Nachdem die Daten in das nötige Format überführt wurden, werden die im Hauptspeicher angelegten Datenstrukturen für die Weiterverarbeitung mittels Serialisierung auf der Festplatte gespeichert. Der Kompressionsvorgang ist somit nur bei veränderter Datenbasis nötig. Für den laufenden Betrieb wird keine neue Kompression durchgeführt. Deutlich erkennbar ist, dass die Daten nur lesend verarbeitet werden können. Es sind bisher auch keine Operatoren zum dynamischen Einfügen neuer Datensätze vorgesehen, folglich ist CameLOD ein reiner Lese-RDF-Store.

Für die Verarbeitung von LOD sind Anfragen der wichtigste Bestandteil, daher werden im CameLOD-Schritt die komprimierten Daten beim Systemstart geladen und im Hauptspeicher abgelegt und stehen solange die Anwendung läuft zur Verfügung. Anschließend können die Anfragen über verschiedene Schnittstellen (Python, Konsole, Anfragedateien, REST) gestellt werden.

3.1.2 Aufbau und Funktionsweise

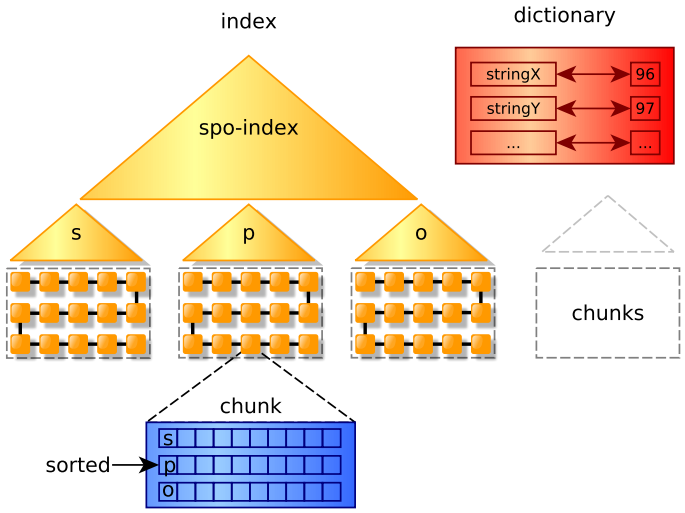

Abbildung 3.2: Strukturieller Aufbau des CameLOD-Projekts, dargestellt sind die Indizes, das Dictionary und die Chunks

In Abbildung 3.2 sind die grundlegenden Bestandteile der Speicherung von CameLOD dargestellt. Die NTripel der Eingabe-Datei werden in Chunks gespeichert. Ein Chunk besteht aus drei Vektoren (für Subjekt-, Prädikat- und Objektinformationen). Die Informationen sind in den Chunks als Integer-Werte gespeichert, welche durch das Dictionary zur Verfügung gestellt werden. Beim Aufbau der Datenbasis findet eine tripelweise Verarbeitung der NTripel-Datei statt. Die Zeichenketten für Subjekt, Prädikat und Objekt werden gespeichert und in einer Hash-Map abgelegt. Nach anschließender Sortierung erfolgt für die Zeichenketten eine Zuordnung von Integer-Werten (long). Die Long-Werte ergeben sich durch die Sortierreihenfolge. Das Dictionary stellt eine Übersetzung von Zeichenketten zu Long-Werten und umgekehrt zur Verfügung. Im nächsten Schritt wird die NTripel-Datei ein weiteres Mal verarbeitet, wodurch

die Chunks für die jeweiligen Indizes aufgebaut werden. Beim Subjektindex ist der Vektor für die Subjektwerte sortiert. Analog sind die Prädikat- und Objektindizes nach ihrer Spalte sortiert. Die Tripel werden der Reihe nach in die Index-Chunk-Strukturen passend eingefügt. Falls ein Chunk mehr als die festgelegten Elemente enthält, erfolgt ein Split des Chunks. Die Chunks eines Indizes sind untereinander verkettet.

Nach dem erfolgreichen Aufbau der Datenbasis erfolgt eine Serialisierung aller Datenstrukturen auf Festplatte. Im CameLOD-Schritt werden bei Anfragen die passenden Indexstrukturen auf Integer-Ebene verarbeitet und nur bei einer Ausgabe sind Übersetzungen zu Zeichenketten durchzuführen. Eine Anfrage, welche sortierte Ergebnisse erfordert, verwendet die passende sortierte Indexstruktur.

CameLOD ist ein Index-Every-System, das heißt, es wird eine materialisierte sortierte Sicht für Subjekt, Prädikat und Objekt in Form des Index zur Verfügung gestellt. Demzufolge werden alle Tripel mehrfach (dreimal) im Hauptspeicher gehalten. Weiterhin gibt es nicht nur Indexstrukturen für die SPO-Tripel, sondern auch einen Geo-Index für räumliche Anfragen, der im Rahmen dieser Arbeit aber nicht von großer Bedeutung ist.

3.1.3 Operatoren und Anfragealgebra

Im CameLOD-Projekt müssen zwei Grundtypen von Operatoren unterschieden werden. Zum einen gibt es Scan- und Filter-Operatoren und zum anderen Join-Operatoren, welche neue Ergebnis-Chunks anlegen. Des Weiteren gibt es auch Operatoren, die die Verarbeitung der Chunks parallelisiert stattfinden lassen.

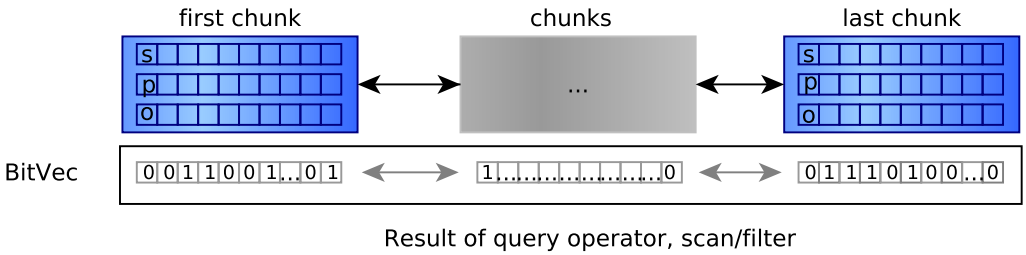

Abbildung 3.3: Anfrageverarbeitung im CameLOD-Projekt

Scan/Filter. In Abbildung 3.3 ist beispielsweise ein Filter- oder Scan-Aufruf dargestellt. Bei einer Filter- und Scan-Anfrage wird im einfachsten Fall der jeweilige Index ausgewählt und die dort gespeicherten Chunks sequentiell verarbeitet. Für jedes gespeicherte Tripel wird das Anfrageprädikat ausgewertet und das Ergebnis der Auswertung wird in einem Bit-Vektor gespeichert. Die Bit-Vektor-Ergebnisse werden nun im Operatorbaum weitergereicht, so dass der nächste Operator auf die Ergebnisse des vorhergehenden zugreifen kann. Auffallend ist, dass hauptsächlich die Verarbeitung mittels der gespeicherten Integer-Werte im Index stattfindet (daher werden die Integer-IDs für die Werte auch im Dictionary sortiert vergeben). Ein 'printer' Operator stellt am Ende mittels der Einträge im Dictionary die Tripel als Zeichenketten in der Ausgabe dar.

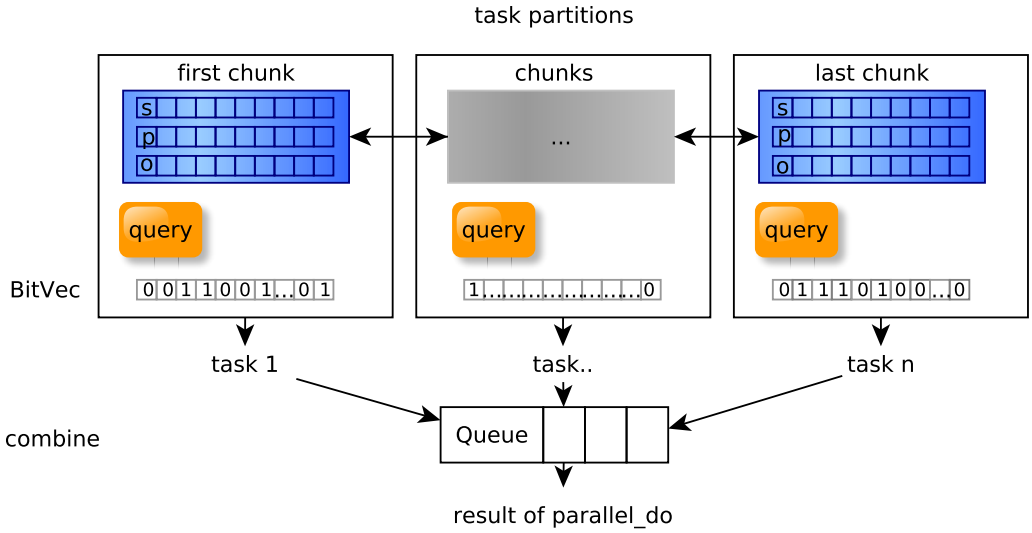

Abbildung 3.4: parallele Anfrageverarbeitung im CameLOD-Projekt

parallel_do. Zur intra-query Parallelisierung besitzt das CameLOD-Projekt den speziellen Operator 'parallel_do(query,n)'. Dabei wird die Anfrage auf n verschiedenen Tasks/Threads parallel ausgeführt. Hauptsächlich erfolgt die Verarbeitung in drei Schritten, vergleiche Abbildung 3.4. Im ersten Schritt wird der Anfrage-Trie $n-1$ mal repliziert, so dass jeder Thread seinen eigenen Anfragebaum hat. Anschließend wird eine Partitionierung durchgeführt, somit erhält jeder Task/Thread einen eigenen Bereich an Chunks, welchen er bearbeiten soll, zugewiesen. Am Ende werden alle Threads parallel ausgeführt (mittels des Intel TBB-Frameworks). Weiterhin übernimmt der 'parallel_do' Operator auch das Zusammenfügen und Sammeln der Ergebnisse der jeweiligen Tasks in einer Queue.

CameLOD stellt auch neben den Index-Scans eine Reihe von Filter-, Projektions-, Distinct-, Sortier- sowie verschiedene Join-Operatoren (index-join, merge-join, spatial-join, star-join, ...) zur Verfügung. Für die RDF-Verarbeitung sind dabei die Join-Operatoren von Interesse.

merge-join/star-join. Durch die vorhandene Sortierung der Daten kann bei den Join-Operatoren der Aufbau einer Hash-Tabelle oder eine explizite Sortierung entfallen.

Bei einem merge-join werden die Informationen eines Chunks über den min- und max-Wert der sortierten Spalte benutzt, um zu überprüfen, ob zwei Chunks in der für den join verantwortlichen Spalte übereinstimmen. Je nachdem kann anschließend ein Join tupelweise erfolgen oder der Chunk wird ignoriert.

Eine besondere Join-Implementierung stellt der star-join dar. Er wird benutzt um spezielle Graph Muster (Basic Graph Pattern) zu finden (beispeilsweise $?spred_1 ?obj_1, ..., ?spred_n ?o_n$). Der star-join benutzt zur Realisierung eine Variante des index-joins. Es wird für jeden Subjektwert überprüft, ob in dem dazugehörigen Chunk Prädikatwerte vorhanden sind, die mit der Liste von Prädikaten übereinstimmen. Das Verfahren wird solange wiederholt bis die BGP-Kette vollständig abgearbeitet ist und alle Chunks verarbeitet wurden.

Algebra. In CameLOD wird eine eigene Algebra benutzt. Nach [HS13] gibt es einen Parser, der SPARQL-Anfragen auf die CameLOD-Algebra umsetzt. Für die Anfragen in dieser Arbeit wird die CameLOD-Algebra verwendet. Zum besseren Verständnis soll nun an einem Beispiel kurz erklärt werden wie sie aufgebaut ist.

```
$1 := scan(s-index, "<http://example.org/foaf/alice>", =[_1, ↘ 1
   →"<http://xmlns.com/foaf/0.1/name>"]);
printer($1) 2
```

Die dargestellte Anfrage stellt einen einfachen Scan über den Subjektindex (s-index) dar. Dabei werden alle Tripel ermittelt, die Alice <http://example.org/foaf/alice> als Subjekt haben und als Prädikat, gefiltert durch =[_1,".../name"], einen Namen aufweisen. Insgesamt sollen also alle Tripel mit Alice als Namen ermittelt werden. Zur Übersicht wurde der Scan in der Variable '$1' abgelegt. Für die Aggregation oder Filterung werden Zugriffe auf Spalten der jeweiligen Tupel benötigt. In der Algebra kann dies durch '_I' (dabei ist I die zu benutzende Spalte des Tupels) erfolgen. Der Aufruf von 'printer($1)' realisiert am Ende der Anfrageverarbeitung eine Textausgabe der Tripel mittels des Wörterbuchs. Dabei werden die Integer-Keys zu den jeweiligen Zeichenketten übersetzt.

3.2 DBpedia

Für alle Messungen und Anfragen mit CameLOD werden Daten aus dem DBpedia-Projekt verwendet. DBpedia ist dabei ein Projekt, welches LOD auf Basis des Wikipedias zur Verfügung stellt. In [Leh+13] wird das genaue Vorgehen beim Aufbauen der LOD beschrieben. Insgesamt können 4 Schritte unterschieden werden. Im ersten Schritt wird die jeweilige Wikipedia-Seite von einer Quelle gelesen (entweder aus einem Wikipedia Dump oder einer (lokalen) Wikipedia Installation). Anschließend wird die Wikipedia Seite geparst, wobei ein Abstract Syntax Tree (AST) aufgebaut wird. Der AST wird dann zu verschiedenen Extraktoren weitergereicht. Zum Beispiel können Labels, Zusammenfassungen, geographische Koordinaten, usw. extrahiert werden. Ein Extraktor erhält einen Syntax Tree und liefert die resultierenden RDF-Statements. Am Ende werden die gesammelten RDF-Statements in verschiedenen Ausgabeformaten gespeichert.

DBpedia bietet diverse Datensätze an. Als Basis dienen in dieser Arbeit die englischen RDF-Daten[1], wobei auch verschiedene Ausgabeformate gewählt werden können. Insgesamt werden 42 einzelne Datensätze angeboten. Für CameLOD ist dabei nur das NTripel-Format wichtig. Aus den von DBpedia bereitgestellten Datensätzen wurde ein kombinierter Datensatz für CameLOD erstellt (DBpedia-Half). Er umfasst ungefähr $30\ GB$ im NTripel-Format und beinhaltet $215\ Mio$ Tripel. Auch ein kompletter Datensatz (DBpedia-Full) mit $110\ GB$ und $740\ Mio$ Tripel wurde erstellt.

[1] zu finden unter http://downloads.dbpedia.org/3.8/en/, Stand 06.10.2013

3.3 Intel VTune - Amplifier 2013

Um eine genaue Analyse des bestehenden CameLOD-Projekts durchzuführen, wird das Profiling-Werkzeug Intel VTune Amplifier 2013 Update 9 benutzt. Es bietet eine große Vielzahl von Analysemöglichkeiten, welche im Folgenden näher beschrieben werden sollen [Intj].

3.3.1 Profiling

Profiling stellt eine Methode dar, um ein bestehendes System auf verschiedene Schwachstellen zu untersuchen. Diese können verschiedener Art sein. Zum Beispiel ist es möglich sogenannte Hotspots zu bestimmen. Hotspots sind Code-Abschnitte/Funktionsaufrufe, welche viel Zeit benötigen. Im Zuge der hochgradigen Parallelisierung von Anwendungssoftware stellen sich häufig Fragen der Form: Wie gut nutzt eine Anwendung einen Multi-Core-CPU aus? Wird durch schlechtes Design eines parallelen Algorithmus viel Zeit benötigt? Oder verursacht eine Anwendung viele Cache-Miss-Zugriffe?

Mittels Profiling können solche Fragen geklärt werden, da es die Anwendungssoftware dahingehend untersucht. Dabei wird das Programm während der Laufzeit analysiert und alle Ergebnisse gespeichert, unter anderem beinhalten diese Ergebnisse Laufzeiten von Funktionsaufrufen.

Intel VTune ist nur ein mögliches Profiling Tool. Beispielsweise gibt es auch noch valgrind[2] (zur Analyse des Speicherverbrauchs und eventueller Memory-Leaks) sowie perf[3] (zur Hardware-Event Analyse) und weitere. VTune dagegen bietet verschiedene Analyseprofile und Methoden, wobei die Ergebnisse auch gleichzeitig gut dargestellt und gefiltert werden.

3.3.2 Analyse-Profile

Intel VTune bietet zwei grundlegende Varianten des Profilings an. Es bietet zum einen 'User-Mode Sampling and Tracing Collection'-Profile und zum anderen die Möglichkeit über 'Hardware Event-based Sampling' ein Programm zu profilen. Aufbauend auf den zwei Grundmöglichkeiten bietet VTune eine Reihe von Analyseprofilen an, dargestellt in Tabelle 3.1. Für eine

Tabelle 3.1: VTune-Analyse-Profile

User-Mode Sampling and Tracing		
concurrency	hotspots	locksandwaits
Hardware Event-based Sampling		
snb-general-exploration	snb-access-contention	snb-branch-analysis
snb-memory-access	snb-cycles-uops	...

Performance-Analyse sind zunächst die User-Mode-Profile von höherem Interesse, da mit ihnen schnell unter anderem Hotspots oder Codeabschnitte, welche viele Locks verursachen, gefunden werden können. Soll eine genauere Analyse (welche zum Beispiel Cache-Hits oder Anzahl

[2] http://valgrind.org/, Stand 24.09.2013
[3] https://perf.wiki.kernel.org/, Stand 24.09.2013

an MicroOPs beinhalten soll) durchgeführt werden, so können die Hardware-Event-Profile benutzt werden.

3.4 Referenzsystem

Alle im weiteren Verlauf dieser Arbeit entstehenden Messungen und Profiling-Ergebnisse wurden auf einem Referenzsystem ausgeführt. Das Serversystem hat dabei folgende Eigenschaften, welche aus dem System ausgelesen wurden:

- Betriebsystem: Ubuntu 12.04.2 LTS
- Kernel-Version: 3.2.0-48-generic
- CPU: zwei 6 Kern CPUs vom Typ: Intel(R) Xeon(R) CPU E5-2630 @ 2.30GHz
- RAM: 128GB DDR3 RAM
- HDD: 2TB S-ATA Festplatten im RAID-Verbund
- GCC: gcc-Version 4.6.3, alle Compiliervorgänge wurden mit -O3 Optimierung durchgeführt

Besonders die verwendete CPU im System ist von Interesse. Es handelt sich um einen Server-CPU von Intel (Xeon CPU E5-2630). Das System verwendet zwei solcher CPUs und hat somit physisch 12 CPU-Kerne, mittels Hyperthreading ergeben sich somit 24 logische Kerne. Es sollen kurz die besonderen Eigenschaften des CPUs dargestellt werden (vergleiche [Intc])

3.4.1 Befehlssatz

Der E5-2630 bietet den herkömmlichen Intel-64 Befehlssatz, besitzt also 64 Bit breite Register. Weiterhin wurde der Befehlssatz um die Advanced-Vektor-Extension-Version 1 (AVX1) erweitert. AVX1 bietet hauptsächlich alle SSE Befehle und Erweiterungen bezüglich der Verarbeitung mittels 256 Bit Registern (vergleiche [Intg]). Es werden auch die Befehlserweiterungen Intel-CRC32 und AES-NI unterstützt.

3.4.2 Cache und Cache-Struktur

Nach [Inte, Abschnitt 18.9.8] basiert der Intel-E5 auf Intels Microarchitektur Sandy Bridge. In

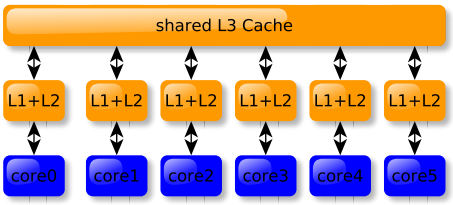

Abbildung 3.5: Intel Xeon CPU E5-2630 Cache Struktur

[Intb, Abschnitt 1.1.1] ist die Cache-Struktur beschrieben (vergleiche Abbildung 3.5). So besitzt

jeder Kern eines Intel Xeon CPU E5-2630 jeweils einen 32 KB L1 Cache für Instruktionen und Daten, sowie jeweils einen L2 Cache mit 256 KB Größe. Der L3 Cache ist 15 MB groß und wird von allen Kernen benutzt.

3.4.3 NUMA

Das Referenz-Server-System besteht aus zwei Intel Xeon CPUs und stellt somit zwei NUMA-Nodes, mit jeweils 6 pyhsischen beziehungsweise 12 logischen Kernen, bereit. Die genauen

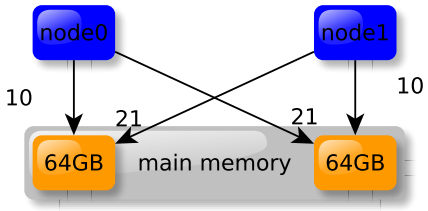

Abbildung 3.6: NUMA Struktur des Serversystems

Informationen wurden mit numactl[4] ermittelt und sind in Abbildung 3.6 dargestellt. Beide NUMA-Nodes sind mittels Links verbunden. Ein Knoten erhält jeweils eine circa 64 GB große Speicherpartition als lokalen Speicher. Dargestellt sind auch die relativen Distanzen zwischen den beiden Speicherbereichen. Insgesamt kann durch die Distanzen festgestellt werden, dass ein Zugriff auf den remote-Speicher ungefähr doppelt so lange dauert wie auf den lokalen. Durch zwei NUMA-Nodes kann nicht viel Gewinn erhofft werden, es muss aber eine genauere Auswertung stattfinden.

3.5 Effizienz

In dieser Arbeit sollen nur zwei Effizienzbegriffe von Bedeutung sein. Zum einen wird der Fokus auf Laufzeiteffizienz und zum anderen auf Speichereffizienz gesetzt. Unter Laufzeiteffizienz soll im weiteren Verlauf der Arbeit die Verbesserung der CPU-Laufzeit eines Prozesses verstanden werden. Für In-Memory Datenbanksysteme ist Hauptspeicher eine beschränkte Ressource. Daher sollen Speichersparmaßnamen betrachtet werden, der Begriff Speichereffizienz stellt dabei das Verringern des benutzten Hauptspeichers dar.

3.6 Zusammenfassung

Im letzten Abschnitt wurde zunächst der Hauptuntersuchungsgegenstand dieser Arbeit, das CameLOD-Projekt, beschrieben. Zunächst erfolgte die Beschreibung des zweistufigen Ablaufs beim Arbeiten mit CameLOD. Es wurde auch der grundlegende Aufbau der Speicherung und des Zugriffs dargestellt. Außerdem wurde beschrieben, wie die Umsetzung der Anfragen mittels

[4] numactl ist Teil der libnuma, vergleiche http://linux.die.net/man/8/numactl

der Indexstrukturen erfolgt. Dabei fand eine kurze Erläuterung der wichtigsten Datenbankoperatoren und ihrer Algebra statt. Für die Messungen sollen realistische Daten benutzt werden. Daher wurde auch die Datengrundlage, welche durch DBpedia zur Verfügung gestellt wird, dargestellt. Es soll eine Analyse des CameLOD-Projekts mittels VTune stattfinden, wofür eine Beschreibung der verschiedenen Analyseprofile des VTune-Tools nötig war. Das Referenzsystem wurde abgesteckt, da alle Messungen dieser Arbeit auf diesem System stattfinden sollen. Besonders wichtig ist dabei die verwendete CPU, ihr Befehlssatz, Cache-Größen und Hierarchien sowie die NUMA-Topologie. Auch die Effizienzbegriffe, welche für die Arbeit wichtig sind, wurden festgelegt.

Kapitel 4

Analyse und Profiling

In dem folgenden Abschnitt soll das bestehende CameLOD-Projekt auf Performance- beziehungsweise Memory-Schwachstellen untersucht werden. Dabei wurde das Profiling-Tool "Intel VTune - Amplifier 2013" verwendet. Es ist nötig die beiden Prozesse Builddb und CameLOD zu untersuchen. Als Grundlage dient die *Revision 7346* vom 16.05.2013 des CameLOD-V2-Branches. Dieser Analyseschritt soll in erster Linie dazu dienen Schwachstellen aufzufinden, um anschließend Ansätze zur Verbesserung zu entwickeln.

4.1 Profiling

Zunächst soll der Builddb-Schritt näher betrachtet werden und anschließend verschiedene Anfragen im CameLOD-Schritt. In allen Analysen wurde mit einer Anzahl von 24 Threads gearbeitet.

4.1.1 builddb

Zum Profilen des Builddb-Schritts wurde der DBpedia-Half-Datensatz geladen und mittels VTune analysiert. Dabei wurden verschiedene Analyse-Profile betrachtet, zunächst fand eine Untersuchung der Hotspots, das bedeutet zeitintensive Teilschritte, und anschließend der Parallelität statt.

Hotspots und Concurency

In Diagramm 4.1 ist die Auslastung aller CPU Kerne beim Einfügen des DBpedia-Half Datensatzes dargestellt. Deutlich erkennbar ist, dass im Builddb-Schritt hauptsächlich nur einer der CPU-Kerne ausgelastet wird, verursacht durch das sequentielle Einlesen der NTripel-Datei. Insgesamt wurde für das Einfügen und Vorbereiten der Daten etwa 2 h Zeit benötigt.

In Tabelle 4.1 sind die Laufzeiten der einzelnen Funktionsaufrufe beim Einfügen aufgeschlüsselt. Es wurden nur Aufrufe mit einer CPU-Zeit größer als ungefähr 17 Sekunden dargestellt. Auffallend ist die hohe Laufzeit, verursacht durch den Boost-Shared-Pointer (P). Beim 'boost::shared_ptr<>' handelt es sich um einen Smart Pointer, bei dem der Speicher automatisch freigegeben wird. Er arbeitet mittels Referenzzählung, wobei viel Laufzeit durch 'shared_count' verursacht wird. Bei der Referenzzählung muss eine Synchronisation stattfin-

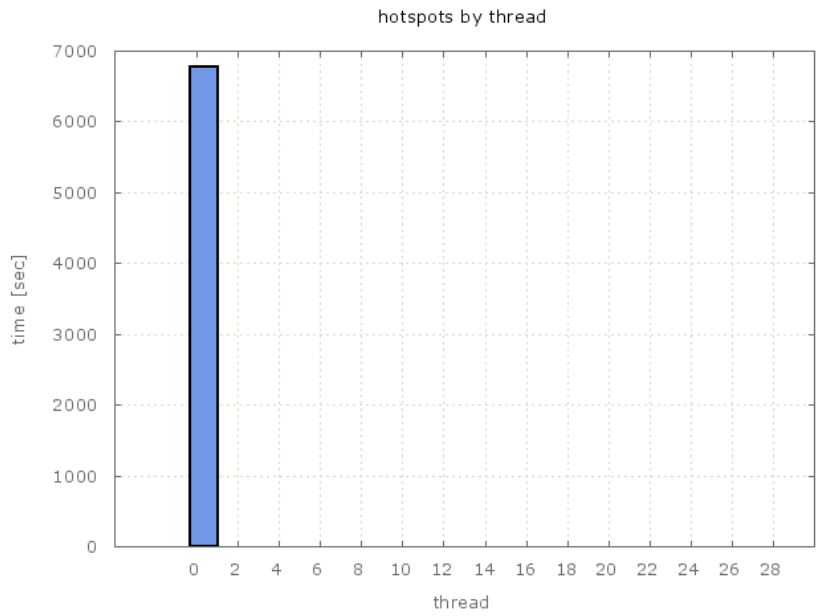

Abbildung 4.1: Hotspots pro CPU-Kern – Übersicht

den, weil mehrere Threads parallel auf den Referenzzähler zugreifen können. Weiterhin wird viel CPU-Zeit durch 'find_insert_position'-Aufrufe (C) verbraucht, weil dort eine lineare Suche verwendet wird, obwohl eine binäre Suche geeigneter wäre. Der Operator zum Vergleichen von Zeichenketten 'CharArrayHashEqual::operator()' wird ebenso häufig aufgerufen, da er zur Behandlung von Kollisionen in der Hash-Table benötigt wird. Daher ist der Einsatz einer Hash-Funktion mit weniger Kollisionen vorteilhaft.

Das 'getline<>' (F) ist für das zeilenweise Einlesen der NTripel-Datei nötig und könnte durch Memory-Mapped-Files beschleunigt werden. Auch für das Parsen der NTripel-Datei wird viel Zeit benötigt, deutlich an dem Aufruf des 'n3_sep::operator()' (T) erkennbar.

Zur besseren Übersicht wurde eine Gruppierung der Funktionsaufrufe vorgenommen, dargestellt in Abbildung 4.2. Es ist schlüssig, dass ein großer Teil der Zeit für Speicherallokation (A) verbraucht wird, da viele Zeichenketten und Objekte angelegt werden müssen. Deshalb ist es auch nicht möglich, sofern nicht unnötig Zeichenketten kopiert beziehungsweise erzeugt werden, an dieser Stelle Verbesserungen erwarten zu können, weil für die spätere Weiterverarbeitung aller Zeichenketten der NTripel-Datei Speicher angelegt werden muss.

Auffallend ist der bereits angesprochene hohe Zeitaufwand durch den Einsatz von Smart-Pointern (P). Aber auch die Chunk-Operationen (C), das Einfügen und Finden der Einfügeposition, benötigen viel Zeit. Außerdem ist erkennbar, dass auch die Dictionary- und Hash-Tabellen-Aufrufe (D) viel Zeit benötigen.

Locks and Wait

Außer den Hotspots war auch die Wartezeit und das Sperren der CPUs ein Untersuchungsgegenstand. Da der Builddb-Schritt aktuell keine Parallelisierung verwendet, ergaben sich keine wichtigen Erkenntnisse beim Profiling. Es muss genauer untersucht werden, ob ein paralleles

Tabelle 4.1: Builddb: Hotspots - Funktionen (Botton-Up) – CPU-Zeit, Wartezeit und Spinzeit [sec], unkategorisierte Werte wurden entfernt

()	Source Function / Function / Call Stack	CPU Time	Spin Time
P	boost::shared_ptr<...>	1393.5	0
A	operator< <char,..., std::allocator<char> >	722.811	0
C	CameLOD::storage::chunk::find_insert_position	720.435	0
P	boost::detail::sp_counted_base::release	570.523	0
P	shared_count	503.27	0
T	n3_sep::operator()	316.256	0
P	boost::detail::shared_count::swap	294.083	0
A	__copy_move_b<long unsigned int>	259.321	0
D	CameLOD::dict::dictionary::CharArrayHashEqual::operator()	183.861	0
F	getline<char, ..., ...>	156.712	0.0480252
G	CameLOD::storage::geo_index::insert	146.545	0.205977
A	std::string::_S_construct<...>	110.359	0
T	CameLOD::db::ntriple_parser::next	71.7346	0
D	hash_combine<char>	66.8627	0
I	CameLOD::storage::spo_index::find_nearest_chunk	63.1748	0
D	CameLOD::dict::dictionary::insert_string	47.5431	0
C	CameLOD::storage::chunk::insert_triple	46.9789	0
A	__gnu_cxx::new_allocator<unsigned long>::construct	35.684	0
D	CameLOD::dict::dictionary::str_to_id	32.2543	0
A	google::libc_allocator_with_realloc<...>::reallocate	25.3692	0
T	CameLOD::db::loddb::import_ntriples	20.5813	0
D	CameLOD::dict::dictionary::CharArrayHashEqual::operator()	18.9782	0
A	__gnu_cxx::new_allocator<...>::deallocate	17.3	0
F	CameLOD::storage::SerializableStorageManager::loadByteArray	17.2038	0
A	__gnu_cxx::new_allocator<...>::allocate	16.9494	0
	...	<17	...
	...	$\sum \approx 6557$	$\sum \approx 0.654$

Einlesen der NTripel-Datei oder das parallele Speichern der Indexstrukturen einen Geschwindigkeitsgewinn hervorbringen kann.

General Exploration

VTune bietet im Analyse-Profile 'General Exploration' eine architekturbezogene Hardware-Event-basierte Analyse. Dabei können Sprungvorhersagen, CPI-Raten (CPU-Clocks per Instruction) und weitere Werte untersucht werden. In Tabelle 4.2 sind die ersten 8 Werte bei der Analyse dargestellt. Auffallend sind die rot markierten Werte, da sie von VTune als kritisch eingestuft werden. Hauptsächlich werden diese kritischen Werte durch Fremdbibliotheken verursacht, daher kann hier nur minimale Verbesserung erwartet werden. Die verwendeten Shared-Pointer von Boost scheiden wiederum schlecht ab.

Auch die anderen von VTune bereitgestellten Analyse-Profile wurden untersucht, ergaben aber keine weiteren wichtigen Erkenntnisse.

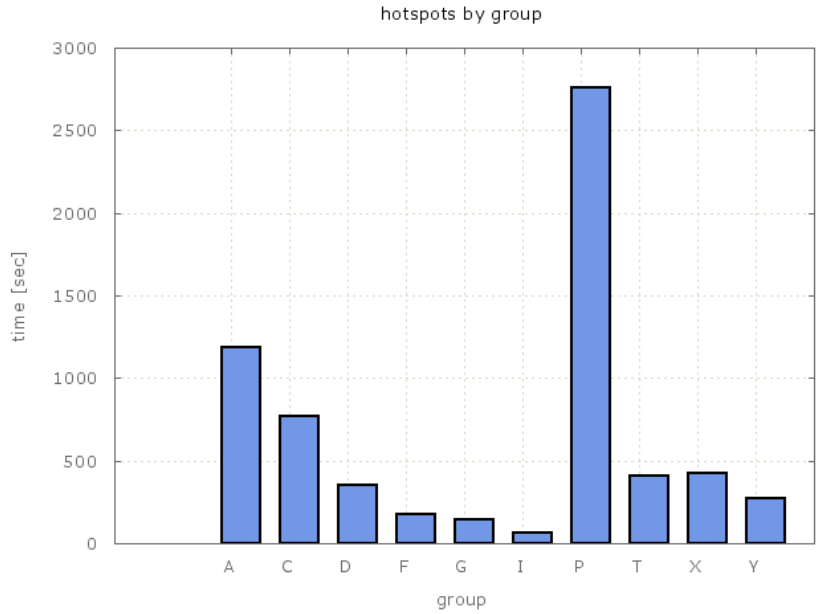

Abbildung 4.2: Gruppierte Übersicht der CPU-Zeit verschiedener Teilprozesse, A: Speicherallokation, C: Chunks, D: Dictionary, F: Datei, G: geo-Index, I: SPO-Index, P: (Smart-)Pointer, T: Triple-Parser, X: ohne Gruppe, Y: Rest (<17 sec)

Tabelle 4.2: Builddb: General Exploration: Bottom-Up, dargestellt sind die ersten 8 Funktionsaufrufe, rot markiert sind die von VTune als kritisch eingestuften Werte

Function / Call Stack	CPI Rate	Bad Speculation	Back-end Bound	Front-end Bound
libc-2.15.so	1.3385	0.115162	0.563134	0.159582
boost::shared_ptr<...>	7.63218	0	0.927226	0.0189811
CameLOD::storage::chunk::find_insert_position	0.502107	0.000211568	0.595467	0.0058461
std::_Rb_tree<...>::_M_insert_unique	2.76018	0.200198	0.603977	0.182035
shared_count	10.9879	0	0.968665	0.0067217
boost::detail::sp_counted_base::release	4.58885	0.0255799	0.841527	0.0178735
boost::detail::shared_count::swap	9.55194	0	0.943673	0.0200293
boost::detail::sp_counted_base::release	3.49447	0.0769893	0.685455	0.0615651

Speicher

Aktuell kann nur der DBpedia-Half-Datensatz mit ≈ 30 GB geladen werden. VTune bietet keine Möglichkeit den Speicherbedarf bestimmter Datenstrukturen zu untersuchen, daher kann an dieser Stelle nur eine Vermutung geäußert werden. Ursachen für das nicht erfolgreiche Laden des kompletten DBpedia-Datensatzes liegen vermutlich in der Verwendung einer 'std::set<string>' Datenstruktur des Dictionarys im Builddb-Schritt.

Ein einfacher Ersatz der 'std::set' Datenstruktur durch einen Präfixbaum könnte Speicher einsparen. Anschließend wäre die Verarbeitung des DBpedia-Full-Datensatzes mit ≈ 110 GB möglich.

4.1.2 CameLOD

Auch der CameLOD-Schritt wurde näher untersucht, dabei wurden verschiedene Anfragen ausgeführt und mittels VTune die Profiling-Ergebnisse analysiert.

Anfragen

CameLOD stellt verschiedene Operatoren zum Aufbau komplexer Anfragen bereit. Als wichtigster Operator ist hier der Scan-Operator anzusehen, weil die meisten Anfragen Scans benutzen. Es wurden 13 Anfragen untersucht (teilweise mit und ohne Parallelisierung, siehe Anhang A), dabei wurden 4 repräsentative Anfragen ausgewählt, welche im folgenden Abschnitt genauer mit VTune analysiert werden sollen. Die Analyse beschränkt sich nur auf das Profil 'Hotspots und Concurency', da in den anderen Profilen nur wenige neue Informationen zu erwarten sind. Zunächst sollen die Anfragen kurz beschrieben werden.

Listing 1: Q1: profiling-2.q2

```
$1 := scan(s-index);
$2 := sorted_aggregation($1, [count(_1)], [_0]);
$3 := filter($2, >[_1, 100]);
null_op($3, "SI")
------- sparql:
select count ?s
where {
    ?s ?p ?o . filter( count ?p > 100)
}
```

Listing 2: Q2: profiling-2-par.q2

```
$1 := co_scan(s-index);
$2 := sorted_aggregation($1, [count(_1)], [_0]);
$3 := filter($2, >[_1, 100]);
$4 := parallel_do($3, 24);
null_op($4, "SI")
```

Die Anfragen Q1 und Q2 ermitteln identische Anfrageergebnisse. Das Besondere an Anfrage Q2 ist, dass Operatoren zur parallelen Verarbeitung verwendet werden. Beide Anfragen stellen einen Scan mit Aggregation und Filterung dar. Dabei wird der Subjektindex gescannt und eine sortierte Aggregation nach der Subjektspalte durchgeführt. Die Aggregation beinhaltet das Zählen der Prädikatspalte. Anschließend wird eine Filterung durchgeführt, so dass nur Tripel betrachtet werden, welche mindestens 100 Prädikatwerte aufweisen.

Listing 3: Q3: profiling-4b-par.q2

```
$1 := co_scan(p-index, ↘
   →"<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>");
$2 := index_join($1, s-index, _0, =[_1, ↘
   →"<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>"]);
$3 := filter($2, <>[_2, _5]);
$4 := parallel_do($3, 24);
null_op($4)
------- sparql:
prefix T <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
select ?s
```

```
where {                                                                    9
    ?s T ?o . ?s T ?o2 . filter(?o != ?o2)                                 10
}                                                                          11
```

In Anfrage Q3 wird dagegen ein index-join mit Filterung und parallelisiertem Scan ausgeführt. Abgesehen von den parallelen Operatoren wird dabei zunächst ein Scan durchgeführt, welcher nur Prädikate eines Typs 'T="<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>"' ermittelt. Anschließend wird ein index-join dieser gefilterten Tripel mit dem Subjektindex durchgeführt. Dabei wird der Join nur ausgeführt, wenn das Subjekt mit T übereinstimmt. Weiterhin erfolgt auch noch eine Filterung, so dass die Spalten 2 und 5 nicht gleich sein dürfen.

Listing 4: Q4: profiling-6.q2

```
$1 := scan(p-index);                                                       1
$2 := sorted_aggregation($1, [count(_2)], [_1]);                           2
$3 := sort($2, [_1], [_0, _1], limit 50);                                  3
printer($3)                                                                4
------- sparql:                                                            5
select count ?p                                                            6
order by desc(count) limit 50                                              7
```

Die Anfrage Q4 besteht aus einem Scan, einer Aggregation und der Sortierung der Ergebnisse. In erster Linie wird ein Scan des Prädikatindex vorgenommen, anschließend wird, gruppiert nach der Prädikatspalte, die Anzahl der verschiedenen Objektwerte ermittelt. Abschließend erfolgt eine Sortierung nach der Prädikatspalte und es werden die ersten 50 Subjekt- und Prädikatwerte ausgegeben.

Listing 5: Q5: People born in Berlin

```
$1 := scan(s-index, ↘                                                      1
   →=[_1,"<http://dbpedia.org/ontology/birthPlace>"]);
$2 := filter($1, =[_2, "<http://dbpedia.org/resource/Berlin>"]);           2
$3 := star_join($2, s-index, _0, _1, [                                     3
    "<http://dbpedia.org/ontology/birthDate>",                             4
    "<http://xmlns.com/foaf/0.1/name>" ,                                   5
    "<http://dbpedia.org/ontology/deathDate>"]);                           6
printer($3)                                                                7
------- sparql:                                                            8
prefix foaf: <http://xmlns.com/foaf/0.1/>                                  9
prefix dbo: <http://dbpedia.org/ontology/>                                 10
prefix : <http://dbpedia.org/resource/>                                    11
select ?name ?birth ?death ?person                                         12
where {                                                                    13
     ?person dbo:birthPlace :Berlin .                                      14
     ?person dbo:birthDate ?birth .                                        15
     ?person foaf:name ?name .                                             16
     ?person dbo:deathDate ?death .                                        17
}                                                                          18
```

In Q5 werden alle Namen, Geburts- und Todesdaten von gespeicherten Personen, welche in Berlin geboren sind, ermittelt. Dabei besteht die Anfrage aus einem Scan und Filterung aller Tripel mit Geburtsort Berlin. Anschließend wird ein star-join, welcher die Tripel ermittelt, die das Geburts- und Todesdatum und den Namen der Person enthalten, durchgeführt.

Q1

Ein Ausschnitt der Top-Down-Ansicht ist in Tabelle 4.3 dargestellt. Insgesamt dauerte die Anfrage $202.46\ s$, wobei nur ein CPU-Kern ausgenutzt wurde, da keine parallelisierten Operatoren im Einsatz kamen. Es fallen dabei $10.3\ s$ auf die Bearbeitung der Anfrage. Die restliche Zeit wird zum Laden der Daten von Festplatte benötigt. Wenig Zeit wird durch (B), die Bitset-Operationen, verursacht. Das ist soweit verständlich, da in der Anfrage nur wenige Ergebnisse durch die Filterung betrachtet werden. Die meiste Zeit wird im 'sorted_aggregation::next'-Schritt (A) benötigt, weshalb nun eine genauere Betrachtung dessen, dargestellt in Tabelle 4.4, erfolgen soll.

Insgesamt fallen $6.01\ s$ auf die inneren Funktionsaufrufe des Schritts. Es ist deutlich erkennbar, dass die Laufzeit wesentlich durch Smart-Pointer ($\approx 3.8\ s$), sowie auch durch die BitSets (C) verursacht wird.

Q2

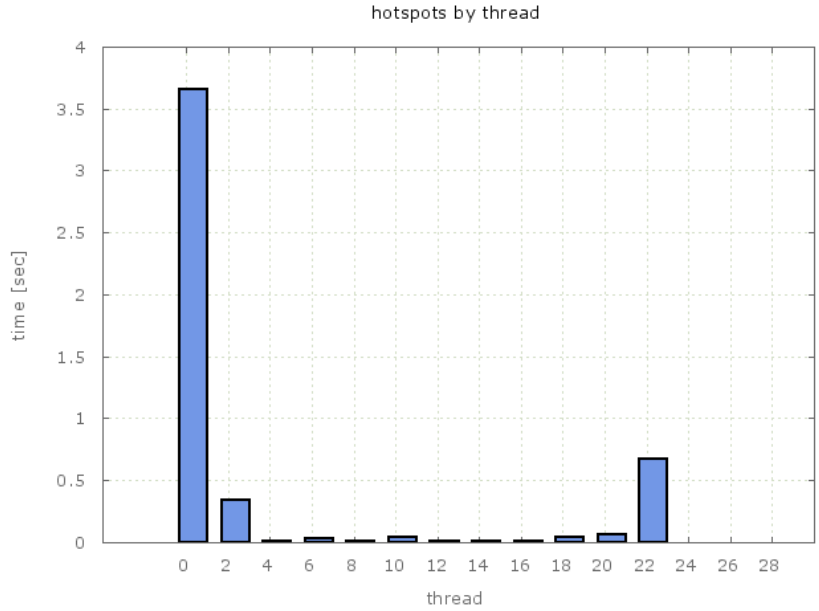

Abbildung 4.3: Hotspots pro CPU-Kern für Anfrage Q2, ohne das sequentielle Einlesen – Übersicht

In Abbildung 4.3 ist sind die Hotspots pro CPU-Thread für die reine Verarbeitung der Anfrage Q2 dargestellt. Es ist deutlich erkennbar, dass alle CPU-Threads genutzt werden. Verständlich ist auch, dass Thread 0 deutlich mehr als alle anderen Threads ausgelastet wird, da am Ende der parallelisierten Anfrage ein Zusammenführen der Teilergebnisse stattfinden muss. Dieser sequentielle Anteil ist nicht vermeidbar.

Insgesamt dauert die Verarbeitung der Anfrage Q2 $183.69\ s$. In der dargestellten Tabelle 4.5 ist die reine (kumulierte) CPU-Zeit von $199.25\ s$ (also ohne Parallelisierung) dargestellt. Dabei wird (A) parallelisiert ausgeführt. Erkennbar ist auch, dass bei Q2 sehr viel Zeit für das Laden ($\approx 179\ s$) benötigt wird. Maßgebend für die Ausführung ist der Aufruf 'CameLOD::query::parallel_do::next' verantwortlich.

Tabelle 4.3: CameLOD: Hotspots Q1 - Top-Down, '+' kennzeichnen Level – CPU-Zeit [sec]

	Call Stack	CPU Time: Total	CPU Time: Self
	CameLOD::db::loddb::restore_from_file	192.160s	0.008s
	CameLOD::query::null_op::open	10.300s	0s
	+ CameLOD::query::filter::next	10.184s	0s
(A)	++ CameLOD::query::sorted_aggregation::next	10.044s	4.032s
	++ CameLOD::query::aggregation::aggr_state::iterate	0.124s	0.124s
(B)	++ boost::function3<void, boost::dynamic_bitset<...>...>	0.016s	0s
	+ CameLOD::query::qresult_t::col_str_value	0.116s	0s
	++ CameLOD::dict::dictionary::id_to_str	0.116s	0.116s

Tabelle 4.4: CameLOD: Hotspots Q1 - Top-Down, genauere Darstellung von 'sorted_aggregation::next' – CPU-Zeit [sec], gekürzt

	Call Stack	CPU Time: Total	CPU Time: Self
	~shared_ptr	1.962s	0s
	~shared_ptr	1.536s	0s
	CameLOD::query::scan::next	0.796s	0.452s
	CameLOD::query::sorted_aggregation::belongs_to_current_group	0.608s	0.608s
	CameLOD::query::sorted_aggregation::build_aggregate_tuple	0.284s	0.012s
	CameLOD::query::aggregation::aggr_state::iterate	0.244s	0.244s
(C)	CameLOD::query::qresult_t::is_set	0.174s	0s
	shared_ptr	0.124s	0s

Tabelle 4.5: CameLOD: Hotspots Q2 - Top-Down, '+' kennzeichnen Level – CPU-Zeit [sec]

	Call Stack	CPU Time: Total	CPU Time: Self
	CameLOD::db::loddb::restore_from_file	178.900s	0.012s
	CameLOD::query::null_op::open	19.560s	0.040s
(A)	+ CameLOD::query::parallel_do::next	19.320s	0s
	+ CameLOD::query::qresult_t::col_str_value	0.158s	0s
	+ ~shared_ptr	0.020s	0s
(B)	+ CameLOD::query::qresult_t::is_set	0.012s	0s
	+ boost::shared_ptr<CameLOD::query::qresult_t>::operator->	0.010s	0.010s
	CameLOD::query::parser::parse_query	0.430s	0s
	~shared_ptr	0.350s	0s

Aus diesem Grund ist dieser in Tabelle 4.6 näher dargestellt. Der parallel_do-Aufruf besteht aus mehreren TBB Aufrufen. In der Tabelle sind nur die relevanten Stellen dargestellt. Auffällig ist wiederum die hohe CPU-Zeit, die durch die Verwendung von Smart-Pointern verursacht wird. Die CPU-Zeiten von (B) und (C) werden durch BitSet-Operationen bestimmt und sind in der parallelisierten Variante minimal höher als in der sequentiellen.

Q3

Anfrage Q3 ist die erste Testanfrage die Join-Operationen beinhaltet. Insgesamt wird für die Anfrage $171.250\ s$ Zeit benötigt. Wobei $233.880\ s$ reine (kumulierte) CPU-Zeit anfällt.

In Tabelle 4.7 ist ein Überblick über die Aufteilung der CPU-Zeit dargestellt, insgesamt werden ungefähr $70\ s$ für die Ausführung der Anfrage genutzt. Die meiste Bearbeitungszeit entsteht durch den 'CameLOD::query::parallel_do::next'-Aufruf (A).

Der Aufruf (A) soll näher betrachtet werden und wird in Tabelle 4.8 dargestellt. Ein geringer Anteil der CPU-Zeit wird durch Smart-Pointer und BitSet-Operationen verursacht, jedoch ist hauptsächlich der Aufruf von 'join_tuples' zeitintensiv.

In Tabelle 4.9 wird daher der 'join_tuples' Schritt näher betrachtet. Es ist erkenntlich, dass maßgebend BitSet-Operationen, Smart-Pointer, die 'insert-tuple' und 'find-tuple'-Methoden der Chunks die CPU-Zeit des 'join_tuple'-Aufrufs verursachen.

Q4

In Anfrage Q4 werden Aggregations-, Scan- und Sortier-Operationen ausgeführt. In Tabelle 4.10 sind die Funktionsaufrufe dargestellt. Insgesamt dauert die Verarbeitung ungefähr $169\ s$.

Betrachtet man den Ladevorgang nicht, so ergibt sich für die Anfrage eine Dauer von circa $4\ s$. Besonders der 'sorted_aggregation::next'-Aufruf ist dafür verantwortlich, welcher in der Tabelle 4.11 genauer aufgeschlüsselt ist. In diesem Ausschnitt bestimmen Smart-Pointer den Großteil der CPU-Zeit.

Q5

Insgesamt wird eine Zeit von ungefähr $178\ s$ für die Anfrage Q5 benötigt. Dabei fallen ungefähr $175\ s$ auf das Laden der Daten an, womit sich circa $3\ s$ Zeit für die Anfrageverarbeitung ergeben. In Tabelle 4.12 sind die Aufrufe dargestellt. Es ist erkennbar, dass für den Smart-Pointer Zeit benötigt wird und für das Ermitteln des nächsten Tripels, welches mit dem Pattern übereinstimmt. Daher ist in Tabelle 4.13 der 'next'-Aufruf genauer dargestellt. In diesem Ausschnitt ist erkennbar, dass der Hauptanteil der CPU-Zeit wiederum durch BitSet-Operationen und Smart-Pointer verursacht wird.

Tabelle 4.6: CameLOD: Hotspots Q2 - Top-Down – parallel_do , '+' kennzeichnen Level – CPU-Zeit [sec], Tabelle wurde gekürzt

	Call Stack	CPU Time: Total	CPU Time: Self
	CameLOD::query::parallel_do::parallel_work::operator()	18.646s	0s
	+ CameLOD::query::filter::next	18.646s	0s
	+++ CameLOD::query::sorted_aggregation::next	18.538s	6.922s
	++++ ~shared_ptr	3.405s	0s
	++++ ~shared_ptr	3.136s	0s
	++++ CameLOD::query::sorted_aggregation::belongs_to..	1.214s	1.190s
	++++ CameLOD::query::sorted_aggregation::build_aggregate..	1.118s	0.244s
	++++ CameLOD::query::co_scan::next	1.045s	0.497s
	++++ CameLOD::query::aggregation::aggr_state::iterate	0.426s	0.426s
	++++ shared_ptr	0.206s	0s
	++++ shared_ptr	0.206s	0s
	++++ ~shared_ptr	0.200s	0s
(C)	++++ CameLOD::query::qresult_t::is_set	0.160s	0s
	++++ ~shared_ptr	0.140s	0s
	++++ ~shared_ptr	0.102s	0s
(C)	+++ boost::function3<void, boost::dynamic_bitset<..>	0.020s	0s

Tabelle 4.7: CameLOD: Hotspots Q3 - Top-Down, '+' kennzeichnen Level – CPU-Zeit [sec], gekürzt

	Call Stack	CPU Time: Total	CPU Time: Self	Overhead
	CameLOD::db::loddb::restore_from_file	163.850s	0.008s	71.40%
	CameLOD::query::null_op::open	70.030s	0.532s	28.60%
(A)	+ CameLOD::query::parallel_do::next	69.130s	0.060s	28.60%
	+ CameLOD::query::parallel_do::num_columns	0.186s	0.186s	0.00%
	+ ~shared_ptr	0.090s	0s	0.00%

Tabelle 4.8: CameLOD: Hotspots Q3 - Top-Down – parallel_do , '+' kennzeichnen Level – CPU-Zeit [sec], gekürzt

	Call Stack	CPU Time: Total	CPU Time: Self
	lodcache::query::parallel_do::parallel_work::operator()	67.670s	0s
	+ lodcache::query::filter::next	67.477s	0s
	++ lodcache::query::index_join::next	67.315s	0.008s
(B)	+++ lodcache::query::index_join::join_tuples	67.159s	21.512s
	+++ lodcache::query::co_scan::next	0.068s	0.044s
(C)	++ boost::function3<void, boost::dynamic_bitset<..>	0.154s	0s
	++ boost::shared_ptr<lodcache::query::qresult_t>::operator->	0.008s	0.008s

Tabelle 4.9: CameLOD: Hotspots Q3 - Top-Down – join_tuples – CPU-Zeit [sec], gekürzt

Call Stack	CPU Time: Total	CPU Time: Self
boost::function3<void, boost::dynamic_bitset<...>::operator()	31.903s	0.054s
boost::dynamic_bitset<...>::operator[]	7.665s	0s
CameLOD::storage::chunk::insert_tuple	2.205s	1.056s
CameLOD::storage::spo_index::find_first_chunk	1.342s	0.040s
dynamic_bitset	0.734s	0s
~dynamic_bitset	0.498s	0s
boost::dynamic_bitset<...>::reference::operator bool	0.390s	0.390s
boost::shared_ptr<...>::operator=	0.262s	0s
boost::dynamic_bitset<...>::set	0.122s	0s

Tabelle 4.10: CameLOD: Hotspots Q4 - Top-Down, '+' kennzeichnen Level – CPU-Zeit [sec]

Call Stack	CPU Time: Total	CPU Time: Self
CameLOD::db::loddb::restore_from_file	162.040s	0s
CameLOD::query::printer::open	4.040s	0s
+ CameLOD::query::sort_op::next	4.040s	0s
++ CameLOD::query::sorted_aggregation::next	3.976s	1.708s
++ CameLOD::query::aggregation::aggr_state::iterate	0.056s	0.056s
++ CameLOD::storage::chunk::sort	0.008s	0s

Tabelle 4.11: CameLOD: Hotspots Q4 -Aggregation, gekürzt, '+' kennzeichnen Level – CPU-Zeit [sec]

Call Stack	CPU Time: Total	CPU Time: Self
~shared_ptr	0.864s	0s
~shared_ptr	0.620s	0s
CameLOD::query::scan::next	0.276s	0.136s
CameLOD::query::sorted_aggregation::belongs_to_current...	0.264s	0.264s
CameLOD::query::aggregation::aggr_state::iterate	0.068s	0.068s
CameLOD::query::qresult_t::is_set	0.044s	0s

Tabelle 4.12: CameLOD: Hotspots Q5 - Top-Down, '+' kennzeichnen Level – CPU-Zeit [sec]

Call Stack	CPU Time: Total	CPU Time: Self
CameLOD::db::loddb::restore_from_file	175.240s	0.012s
CameLOD::query::printer::open	2.490s	0s
+ CameLOD::query::star_join::next	2.490s	0s
+++ CameLOD::query::filter::next	1.814s	0.008s
+++ CameLOD::query::star_join::join_tuples	0.612s	0.172s
+++ ~shared_ptr	0.056s	0s

4.2 Kompression

CameLOD stellt im Grundprinzip eine Wörterbuchkomprimierung dar, da die verwendeten Zeichenketten in einem Wörterbuch abgelegt werden und die Repräsentation der Tripel anschließend durch die kodierten Zeichenketten stattfindet.

Der DBpedia-Half Datensatz mit ungefähr $30\ GB$ an RDF-Tripeln kann durch CameLOD auf ungefähr $18\ GB$ komprimiert werden. Ein großer Teil verbraucht dabei das Wörterbuch und die Mehrfachspeicherung der Indizes.

Der DBpedia-Half Datensatz umfasst 214999298 S-P-O-Tripel. Für Subjekt, Prädikat und Objekt werden jeweils 64 Bit Long-Werte gespeichert. Somit ergibt sich für einen Index (beispielsweise der Subjektindex):

$$3 \cdot 64\ Bit \cdot 214999298 = 41279865216\ Bit \approx 4.8\ GB$$

Somit folgt insgesamt für die Subjekt-, Prädikat- und Objekt-Indizes ein Speicherverbrauch von:

$$3 \cdot 4.8\ GB = 14.4\ GB$$

Die verbleibende Differenz zu den benutzten $18\ GB$ wird für den Geo-Index und das Wörterbuch verwendet ($\approx 3.6\ GB$).

Demzufolge sind für die Speicherung hauptsächlich die Indizes für die SPO-Tripel verantwortlich. Aktuell wird für ein Tripel $3 \cdot 64\ Bit = 192\ Bit$ zur Speicherung benötigt. Angenommen man könnte diese Speicherung auf $128\ Bit$ reduzieren, so ergäbe sich ein Speicherverbrauch von:

$$128\ Bit \cdot 214999298 = 27519910144\ Bit \approx 3.2\ GB$$

für einen Index. Also insgesamt für die SPO-Indizes:

$$3 \cdot 3.2\ GB = 9.6\ GB$$

Um den geforderten Speicherverbrauch eines Tripels auf $128\ Bit$ zu reduzieren, wäre eine homogene Aufteilung beispielsweise in eine Speicherung von drei $\frac{128}{3} Bit \approx 42 Bit$ Teilen möglich. Diese Aufteilung bietet aber wenig Vorteile, da allgemein nur effiziente Speicherzugriffe auf Standard-Datentypen (ubyte $8\ Bit$, uword $16\ Bit$, uint $32\ Bit$ und ulong $64\ Bit$) möglich[1] sind. Daher ist eine Byte-gerechte Aufteilung der jeweiligen Werte für Subjekt-, Prädikat- und Objektwerte sinnvoller. Durch den Aufbau der SPO-Tripel bestehen einige der Spalten (zum Beispiel die Subjekt-Spalte eines Chunks) aus Blöcken gleicher Integer-Zahlen, auch hier kann eine Kompression Speicher einsparen.

Eine Kompression (mittels Huffman oder ähnlichen Verfahren) des Wörterbuches kann den Speicherbedarf reduzieren. Momentan werden die Zeichenketten durch ein globales Dictionary gespeichert. Der Subjekt- und Prädikat-Teil von RDF-Daten besteht aus URIs, welche häufig eine Präfixübereinstimmung aufweisen. Aktuell werden alle Zeichenketten gespeichert, auch wenn sie gemeinsame Präfixe aufweisen. Durch das Einsparen der mehrfachen Speicherung

[1] beispielsweise über einen Cast auf den jeweiligen Datentyp

der Präfixe ist weniger Speicher nötig. Ein Präfixbaum bietet sich hierfür an. Dennoch muss der Einsatz eines solchen Baums näher untersucht werden, da unter Umständen mehr Zeit für die Zugriffe auf die gespeicherten Zeichenketten erforderlich ist. Hauptsächlich findet die Verarbeitung der Daten in CameLOD nicht auf Basis der Zeichenketten, sondern in den Indexstrukturen, statt. Sie bestehen dabei aus Integer-Werten, weshalb für die Zeichenketten eine langsamere Datenstruktur benutzt werden kann.

4.3 Zusammenfassung der Analyseergebnisse

Alle im letzten Abschnitt durchgeführten Betrachtungen und Ergebnisse sollen im Folgenden kurz zusammengefasst werden.

4.3.1 Profiling

Grundlegend ist für einen großen Teil der Laufzeit der Einsatz von Smart-Pointern und BitSet-Datenstrukturen verantwortlich. Auch konnten einige Implementierungsverbesserungen bezüglich des Einfügens und Suchens eines Tripels innerhalb eines Chunks ermittelt werden. An diesen Stellen ist eine binäre Suche aufgrund der Sortierung zwingend erforderlich. Der Builddb-Schritt ist in der aktuellen Version in keiner Weise parallelisiert, demnach könnte ein paralleles Einlesen und Speichern der Indexstrukturen Verbesserungen erzielen. Da die verwendete Hash-Funktion von Boost viele Kollisionen aufweist, entstehen beim Aufbau viele Zeichenkettenvergleiche. Eine andere Hash-Funktion könnte an dieser Stelle weniger Vergleiche erzielen. Durch das Profiling konnte festgestellt werden, dass die parallelisierten Operatoren gut umgesetzt sind. Ein gewisser sequentieller Anteil lässt sich jedoch nicht vermeiden, da die parallel erzeugten Ergebnisse am Ende zusammengefasst werden müssen.

In den Tabellen 4.14 und 4.15 sind die prozentualen Anteile an der Gesamtzeit der geschlussfolgerten Problemstellen (BitSet-Operationen, Smart-Pointer, Hash-Kollisionen, Einfügeoperationen in den Chunks) der beiden Schritte Builddb und CameLOD dargestellt.

4.3.2 Kompression

Auch die verwendete Kompression durch das CameLOD-Projekt wurde genauer betrachtet und festgestellt, dass weitere Untersuchungen bezüglich der Indexstrukturen und deren inneren Speicherung der Tripel mittels Integer-Sequenzen durchgeführt werden müssen. Außerdem kann auch das Dictionary zum Speichern der Zeichenketten durch den Einsatz einer anderen Datenstruktur (beispielsweise Trie) speichereffizienter gestalltet werden. Dabei muss aber untersucht werden, wie sich die Zugriffszeiten auf die Zeichenketten verändern.

Tabelle 4.13: CameLOD: Hotspots Q5 - Top-Down des filter next Aufrufs, gekürzt, '+' kennzeichnen Level – CPU-Zeit [sec]

Call Stack	CPU Time: Total	CPU Time: Self
CameLOD::query::scan::next	1.250s	0.326s
+ boost::function3<void, boost::dynamic_bitset<...>::operator()	0.492s	0s
+ CameLOD::query::qresult_t::set_bit	0.356s	0s
+ shared_ptr<CameLOD::query::qresult_t>	0.032s	0s
boost::function3<void, boost::dynamic_bitset<...>::operator()	0.556s	0s

Tabelle 4.14: Prozentualer Zeitbedarf im Builddb-Schritt

	Hash-Funktion	Kollisionen	Chunks Find & insert	Smart-Pointer
Builddb	$\approx 1\%$	$\approx 4\%$	$\approx 12\%$	$\approx 42\%$

Tabelle 4.15: Prozentualer Zeitbedarf im CameLOD-Schritt, reine Anfrageverarbeitung ohne das Laden der Daten

	BitVektoren	Smart-Pointer
Q1	$\approx 1.8\%$	$\approx 35\%$
Q2	$\approx 1\%$	$\approx 40\%$
Q3	$\approx 69\%$	$\approx 0.5\%$
Q4	$\approx 1\%$	$\approx 36\%$
Q5	$\approx 56\%$	$\approx 3.5\%$

Kapitel 5

Ansätze zur Effizienzsteigerung

Im folgenden Abschnitt sollen für die im vorhergehenden Kapitel ermittelten Performance- beziehungsweise Speicherschwachstellen Verbesserungen erläutert werden. Zunächst werden die jeweiligen Schwachstellen kurz dargestellt und verschiedene Lösungsansätze, welche moderne CPU-Features verwenden, beschrieben. Weiterhin folgt für jedes geschlussfolgerte Problem ein Mini-Benchmark, um abzuschätzen ob an dieser Stelle Verbesserungen zu erwarten sind. Dabei werden auch Vergleiche mit einer maximal möglichen technischen Referenzimplementierung betrachtet. Auch der eigene Code wurde mittels Profiling untersucht.

5.1 Ansätze

In der Analyse des CameLOD-Projekts konnten einige problematische Stellen gefunden werden. Generell ergab sich dabei eine hohe Laufzeit, verursacht durch den Einsatz von Smart-Pointern. Bei der Anfrageverarbeitung benötigt weiterhin der Einsatz von BitSets viel CPU-Zeit. Außerdem konnte festgestellt werden, dass beim Aufbau der Datenbank viel Zeit durch Hash-Kollisionen und die damit verbundene Kollisionsbehandlung benötigt wird. Auch der Speicherverbrauch wurde betrachtet und es kann durch den Einsatz von Kompressionsschemata für Integer-Sequenzen beziehungsweise Dictionary-Strukturen Speicher und eventuell auch Verarbeitungszeit gespart werden. Die beschriebenen Grundprobleme sollen im folgenden Abschnitt genauer betrachtet werden, dabei wird grundlegend immer zunächst das Problem im Bezug auf das CameLOD-Projekt beschrieben. Anschließend folgen verschiedene Ansätze, um das Problem effizienter zu lösen. Am Ende werden mittels Mini-Benchmarks verschiedene Implementierungen zur Lösung verglichen und in der Auswertung werden Schlussfolgerungen für das CameLOD-Projekt gezogen.

Als Kernprobleme ergaben sich in der Analyse folgende Bereiche:

- Smart-Pointer (an allen Stellen im System)
- Hash-Funktionen (beim Aufbau der Datenbasis)
- Bit-Vektoren (zur Speicherung der Anfrageergebnisse)
- Kompression (Speicher kann eingespart werden)
- Speicher (NUMA, Memory Mapped Files)
- Chunk-Größe

5.2 Smart-Pointer

Für einen großen Teil der Laufzeit des CameLOD-Projektes ist die Verwendung von Smart-Pointern (genauer boost::shared_ptr) verantwortlich. Der Einsatz von Smart-Pointern bietet viele Vorteile, da beispielsweise eine manuelle Speicherverwaltung entfällt (vergleiche [Ale01, Kapitel 7, S.157ff]). Es ist daher nicht ohne Weiteres möglich, die Smart-Pointer durch Raw-Pointer zu ersetzen und auch nicht gewollt. Aber an einigen Stellen im Projekt könnten durchaus Raw-Pointer eingesetzt werden, beispielsweise bei der Verkettung der Chunks zu einer Liste.

5.2.1 Problem

Es gibt verschiedene Ansätze Smart-Pointer (genauer shared-Pointer, im Weiteren soll Smart-Pointer synonym für einen shared-Smart-Pointer benutzt werden) zu implementieren. Klassisch sind zwei Verfahren zur Implementierung solcher Smart-Pointer üblich, zum einen über Referenzzählung und zum anderen mittels Referenzverlinkung (vergleiche [Ale01, S.165f]).

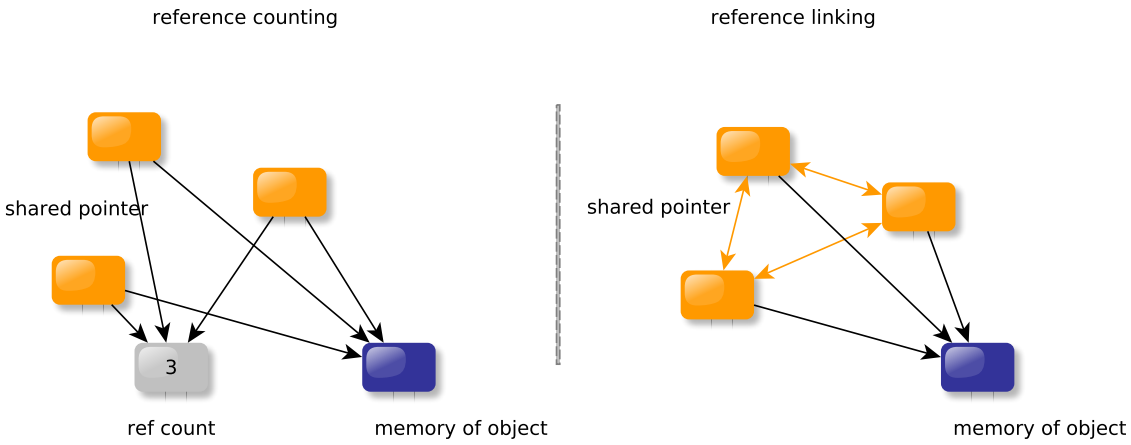

Abbildung 5.1: Smart-Pointer-Konzepte

In Abbildung 5.1 sind die zwei allgemeinen Konzepte zur Realisierung von Smart-Pointern dargestellt. Beim Ansatz der Referenzzählung wird global gezählt, wie häufig ein Shared-Pointer auf das Objekt zeigt. Ist die Anzahl an Referenzen auf Null gesunken, so kann das Objekt freigegeben werden. Wichtig für dieses Verfahren beim Einsatz in multi-threaded Anwendungen ist eine Synchronisation des Referenzzählers. Im Gegensatz dazu benutzt das Referenzverlinkungsverfahren keinen Zähler, sondern alle Smart-Pointer die auf ein Objekt zeigen bilden eine verkettete Liste. Ist in dieser Liste nur noch ein Element, so kann nach der Freigabe dessen auch das referenzierte Objekt freigegeben werden. Nachteilig an der Referenzverlinkung ist, dass alle Listenoperationen synchronisiert werden müssen. Dadurch entsteht ein erheblicher Overhead, weshalb diese Variante im weiteren Verlauf nicht betrachtet wird.

Neben den vorgestellten allgemeinen Varianten gibt es weitere ähnliche Verfahren, zum Beispiel kann das referenzierte Objekt die Referenzzählung selbstständig verwalten. In dem Fall spricht man von intrusive-shared-Pointer.

5.2.2 Benchmark

In einem Benchmark wurden verschiedene Smart-Pointer Implementierungen miteinander verglichen. Dabei wurden in einem Array eine variable Anzahl an Smart-Pointer angelegt und zwei mal zugegriffen. Es soll somit die Verwendung in einem einfachen System simuliert werden. Natürlich müsste das Testszenario komplexer gestaltet werden, aber es reicht für einen allgemeinen Eindruck aus. Insgesamt wurden 5 verschiedene (Smart-)Pointer Implementierungen betrachtet. Es kamen keine Smart-Pointer basierend auf dem Referenzverlinkungsprinzip zum Einsatz, da sie nur schlecht für Multi-Thread-Anwendungen geeignet sind.

boost, std. Die von Boost und dem neuen C++11 Standard zur Verfügung gestellten Shared-Pointer Implementierungen wurden als boost und std untersucht. Grundlegend sind beide Implementierungen ähnlich, da sie mittels Referenzzählung arbeiten und im neuen C++ Standard teilweise Boost-Code übernommen wurde.

counted. Die counted-Variante stellt eine eigene minimale Implementierung eines thread-safe Smart-Pointer, basierend auf Referenzzählung, dar.

intrusive. Die Boost-Bibliothek bietet neben den eigenen Smart-Pointern auch die Möglichkeit spezialisierte Smart-Pointer zu erstellen. Mit der intrusive-Variante kann eine Klasse die Referenzzählung selbst übernehmen. Im Benchmark wurde eine einfache Klasse mit der intrusive-Variante benutzt.

raw. Als allgemeiner Referenzwert soll der Einsatz von normalen Pointern dienen. Es ist eine manuelle Speicherdeallokation nötig, welche in den Messungen nicht mit einfließt. Da bei den Smart-Pointer Varianten das Aufräumen des Speichers erst am Ende des Funktionsaufrufs stattfindet und somit nicht in der eigentlichen Messung erfasst wird.

5.2.3 Auswertung

In Diagramm 5.2 sind die ermittelten Ergebnisse des Benchmarks dargestellt. Es wurden für jeden Datenpunkt 32 Messungen durchgeführt außerdem sin im Diagramm Intervalle mit 99% Konfidenz dargestellt. Zunächst ist deutlich erkennbar, dass die Varianten boost und std sehr dicht beieinander liegen, was durch den Fakt, dass sie grundsätzlich ähnliche Implementierungen darstellen, erklärbar ist. Besser als diese beiden Varianten sind nur die counted und intrusive Varianten. Bei der Implementierung konnte ein erheblicher zeitlicher Overhead, welcher durch die Verwendung von Synchronisationsmechanismen für die Referenzzählung verursacht wird, beobachtet werden. Sie sind aber zwingend erforderlich, da das CameLOD-Projekt multi-threaded ist. Intrusive-Pointer sind ungefähr 20% schneller als die boost-Variante. Sehr deutlich ist der Fakt, dass Smart-Pointer erheblich viel Zeit benötigen, da die dargestellte Raw-Pointer ungefähr 10 mal schneller sind als alle Smart-Pointer Varianten.

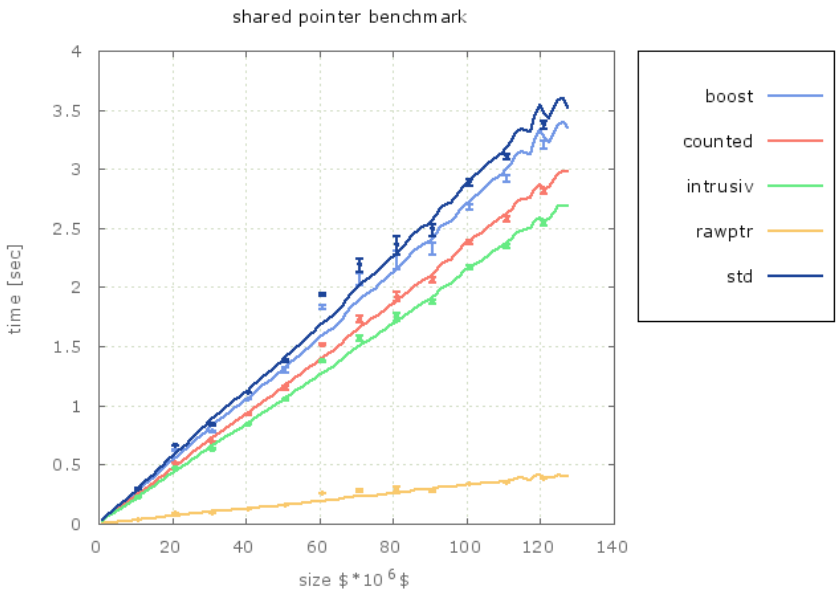

Abbildung 5.2: Smart-Pointer-Benchmark, Zeit beim mehrfachen Zugriff auf Zufallszahlen mit verschiedenen Ansätzen, dargestellt sind Mittelwerte über 32 Messungen mit 99%-Konfidenzintervall

5.2.4 Fazit

Im Benchmark konnten verschiedene Smart-Pointer-Implementierungen begutachtet werden. Grundlegend ergibt sich, dass Smart-Pointer in hocheffizienten Teilen des Systems gemieden werden sollten. Unter Umständen wäre eine Verwendung von intrusive-Pointern besser als die allgemeinen boost-Pointer, aber auch nicht an allen Stellen des Systems. Konkret kann für das CameLOD-Projekt der Einsatz eines Memory-Contexts für die Anfrageverarbeitung geschlussfolgert werden. Dadurch erhält jede Anfrage ihren eigenen Speicherbereich und nachdem die Anfrage verarbeitet wurde, kann dieser aufgeräumt werden oder sogar durch das Speichern und Wiederverwenden von Ergebnis-Chunks ein Caching-Effekt entstehen. Eine manuelle Speicherverwaltung ist daher erforderlich.

Auch können als Kompromiss die Index-Chunks komplett mit raw-Pointern implementiert werden, dagegen aber die Chunks, welche bei Join-Operationen zur Speicherung der Ergebnisse verwendet werden, mit intrusive-Pointern realisiert werden. Somit folgt eine konzeptionelle Trennung zwischen read-only-Chunks und result-Chunks, da für die Index-Chunks das Freigeben des Speichers nur beim Beenden von CameLOD erforderlich ist.

In der Analyse konnte festgestellt werden, dass für die boost Smart-Pointer-Variante der in Tabelle 5.1 dargestellte prozentuale Anteil an der Gesamtzeit für die beiden Vorgänge Builddb und CameLOD benötigt wird. In der Spalte 'raw' sind die prozentualen Anteile, welche entstehen, wenn die Smart-Pointer theoretisch durch Raw-Pointer ersetzt werden, dargestellt. Als Grundlage dienen dabei die Ergebnisse des Benchmarks. Das bedeutet es wurde angenommen, dass Raw-Pointer etwa 10 mal schneller sind als Smart-Pointer. Es ergibt sich zum Beispiel im Builddb-Schritt eine Ersparnis von ungefähr 37% an der Gesamtzeit. Auch die Anfragen können durch den Einsatz von Raw-Pointern profitieren, wobei beispielsweise bei Q1, Q2 und Q4 mindestens 30% eingespart werden können. Die Anfragen Q3 und Q5 können nur wenig Ver-

Tabelle 5.1: % Zeitbedarf für Smart-Pointer/Raw-Pointer im CameLOD-Projekt

	Smart-Pointer (boost)	raw
Builddb	$\approx 42\%$	$\approx 4.2\%$
Q1	$\approx 35\%$	$\approx 3.5\%$
Q2	$\approx 40\%$	$\approx 4\%$
Q3	$\approx 0.5\%$	$\approx 0.05\%$
Q4	$\approx 36\%$	$\approx 3.6\%$
Q5	$\approx 3.5\%$	$\approx 0.35\%$

besserung durch Raw-Pointer erzielen, weil die Gesamtzeit bei diesen Anfragen hauptsächlich durch Bit-Set-Operationen bestimmt wird.

5.3 Hash-Funktion

In dem folgenden Abschnitt soll der Einsatz einer auf AES-NI basierenden Hash-Funktion als Ersatz für die aktuell benutzte Boost-Hash-Funktion untersucht werden.

Für das Hashen der verschiedenen Zeichenketten wird im CameLOD-Projekt viel Zeit benötigt, weil zunächst ein Dictionary aller Zeichenketten aufgebaut werden muss, um für die jeweiligen Chunks nur Long-Werte zur Verarbeitung nutzen zu können. Dieses Konzept stellt eine Wörterbuchkomprimierung dar, weshalb eine schnelle und gute Hash-Funktion nötigt ist.

5.3.1 Problem

Listing 6: Boost hash range

```
template <class It>
inline std::size_t hash_range(It first, It last) {
    std::size_t seed = 0;
    for(; first != last; ++first) {
        boost::hash<T> hasher; //< generate hash for single char
        seed ^= hasher(v) + 0x9e3779b9 + (seed<<6) + (seed>>2);
    }
    return seed;
}
```

Aktuell wird die 'boost::hash_range' Funktion verwendet. Ihre Grundfunktionalität ist in Code-Auszug 6 dargestellt (Quelle: [Booa]). Grundlegend wird für jedes Zeichen i der Zeichenkette s, welche durch die Anfangs- und End-Zeiger $first$ und $last$ festgelegt ist, eine Kompressionsfunktion angewandt und mit dem vorhergehenden Hash-Wert verknüpft.

Auffallend ist die sequentielle Verarbeitung der Zeichenkette. Es wird jedes Zeichen einzeln betrachtet und verarbeitet. Moderne CPUs besitzen jedoch 64 Bit Register, daher können durch die byteweise Berechnung solche Register nicht vollständig ausgenutzt werden. Besser wäre eine Verarbeitung von Long- (64 Bit) oder sogar SSE- Blöcken (128 Bit) der Zeichenkette.

In [BÖS11] werden kryptographische Hash-Funktionen betrachtet, welche Hardware AES (AES-NI) zur blockweisen Verarbeitung verwenden. Leider sind die dort aufgeführten Hash-Funktionen kryptographische Hash-Funktionen, was bedeutet, sie müssen zusätzliche Eigenschaften erfüllen, welche für die Anwendung in einer Hash-Tabelle nicht notwendig sind. Zum Beispiel müssen sie kollisionsarm und schlecht rekonstruierbar sein. Die Kollisionsarmut ist auch für eine Hash-Tabelle eine gute Eigenschaft, da weniger Kollisionen behandelt werden müssen, hingegen wird der Sicherheits-Aspekt jedoch nicht benötigt.

Auch viele der SHA3-Kandidaten benutzen AES-NI zur Konstruktion einer kryptographischen Hash-Funktion (beispielsweise Grøstl, vergleiche [Gau+08]). Die SHA3-Kandidaten setzen auf Sicherheit und sind daher um einiges langsamer als die Boost-Hash-Funktion. Inspiriert durch die SHA3 Kandidaten wurde eine für Hash-Tabellen einsetzbare Hash-Funktion entwickelt. Sie verwendet dabei die Damgård-Merkle-Konstruktion (DM-Konstruktion, vergleiche [Cor+05]) einer Hash-Funktion basierend auf der Blockchiffre AES mit so wenig AES-Runden wie möglich.

AES-NI Hash Funktion

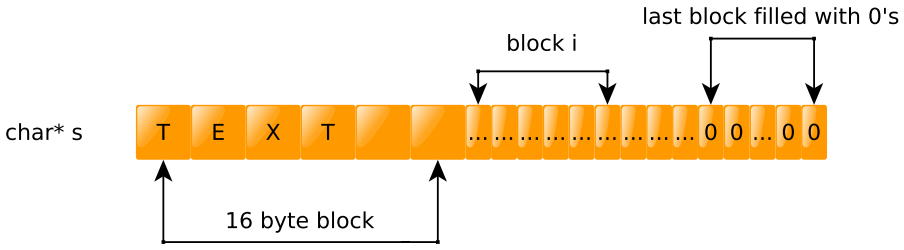

Abbildung 5.3: AES-NI-Hash Grundfunktionalität

Die Grundidee der AES-NI-basierten Hash-Funktion ist in Abbildung 5.3 dargestellt und lässt sich durch wenige Kernideen beschreiben. Eine Zeichenkette wird dabei blockweise verarbeitet. Die Blöcke sind 16 Byte groß und können demnach in einem SSE-Register verarbeitet werden. Für jeden Block wird die DM-Konstruktion angewendet. Dabei wird ein neuer Hash-Wert H_i basierend auf dem alten Hash-Wert H_{i-1} mittels einer Kompressionsfunktion $AES(msg, key)$ folgendermaßen berechnet

$$H_0 = 0$$

$$H_i = AES(Block_i, H_{i-1}), \forall i > 0.$$

Damit die Zeichenkette blockweise verarbeitet werden kann, werden am Ende Nullen angefügt, um eine durch die Blockgröße teilbare Zeichenkettenlänge zu erhalten. Der AES Verschlüsselungsschritt besteht aus nur zwei relevanten AES-Runden (die Start- und End-Runde), wobei am Anfang die Nachricht mit dem Schlüssel XOR verknüpft wird. Die AES-Runde wird durch die Hardwarefunktionalität von AES-NI realisiert (vergleiche [Intf]). In der genauen Beschreibung der DM-Konstruktion werden nicht nur Nullen an die Zeichenkette angefügt, sondern eine Kodierung der Zeichenkettenlänge um kryptographische Sicherheit zu gewährleisten. In der hier beschriebenen Implementierung wurde aus Performance-Gründen darauf verzichtet. Genauso werden in der Implementierung zunächst jeweils zwei 16 Byte Blöcke in einer Iteration berechnet, anschließend die restlichen vollen 16 Byte Blöcke und am Ende werden durch geschicktes Laden der Zeichenkette in eines der SSE-Registern die angehängten Nullen erzeugt.

Der so erzeugte 128 Bit Hash-Wert wurde mittels einer Finalisierungsfunktion $final()$ in einen 64 Bit Hash-Wert transformiert und zurückgegeben.

$$final(H) = low(H) \oplus high(H)$$

Zunächst wurde mittels CryptoPP-Bibliothek testweise auf dieser Grundidee eine Hash-Funktion erzeugt, um festzustellen, ob sich eine effiziente Implementierung lohnt. Die so erzeugte Funktion zeigte erstaunlich gute Kollisionsresistenz, deshalb wurde eine hardwarenahe Implementierung mit C Intrinsics vorgenommen (vergleiche [Intf]). Neben der AES-NI basierten Hash-Funktion wurde auch eine auf der Intel-CRC32 Befehlssatzerweiterung basierende Hash-Funktion erzeugt. Sie zeigte keine guten Eigenschaften bezüglich Laufzeit und Kollisionen im Vergleich zur Boost-Hash-Funktion und wurde daher im Benchmark nicht betrachtet.

5.3.2 Benchmark

Um die Implementierung der Hash-Funktion mit der Boost-Hash Funktion zu vergleichen, wurde ein Benchmark entwickelt, dabei wird eine beliebige NTripel-Datei geladen und in ihre Bestandteile Subjekt, Prädikat und Objekt zerlegt. Anschließend wurden alle Zeichenketten in einer std::map abgelegt. Diese Map wurde durchlaufen und mit den beiden Funktionen die jeweiligen Zeichenketten gehasht. Insgesamt wurde die Zeit zum Hashen aller Zeichenketten ermittelt und in einem weiteren Schritt die summierten Kollisionen berechnet.

Als Datengrundlage dienten die englischen NTripel-Daten des DBpedia-Projektes und ein kombinierter Datensatz (DBpedia-Half $\approx 30\ GB$).

5.3.3 Auswertung

In Grafik 5.4 wird die Zeit für die Berechnung der Hash-Werte für beide Funktionen dargestellt. Es wurden Mittelwerte basierend auf 32 Messungen berechnet und Intervalle mit 99% Konfidenz. Auffallend ist, dass beide Hash-Funktionen in etwa gleich schnell sind, da sie in der Darstellung fast deckungsgleich verlaufen. Vermutlich ist die AES Variante nicht wesentlich schneller als die Boost-Implementierung, da für die SSE-Register Kopiervorgänge benötigt werden.

Dagegen zeigt Abbildung 5.5 die summierten Kollisionen der untersuchten Hash-Funktionen. Deutlich erkennbar ist, dass die AES-Funktion bei allen untersuchten Datensätzen keine einzige Kollision aufwies, dagegen gab es bei der Boost-Hash-Funktion immer Kollisionen. Kollisionen in Hash-Tabellen sind zeitaufwändig, da beispielsweise für die Behandlung Zeichenkettenvergleiche stattfinden müssen.

5.3.4 Fazit

Die Kollisionsarmut der AES-NI-basierten Hash-Funktion bringt einen wesentlichen Performance-Vorteil beim Einsatz in einer Hash-Tabelle. Hash-Kollisionen treten sehr häufig beim Aufbau der Datenbank auf. Der Builddb-Schritt kann somit beschleunigt werden.

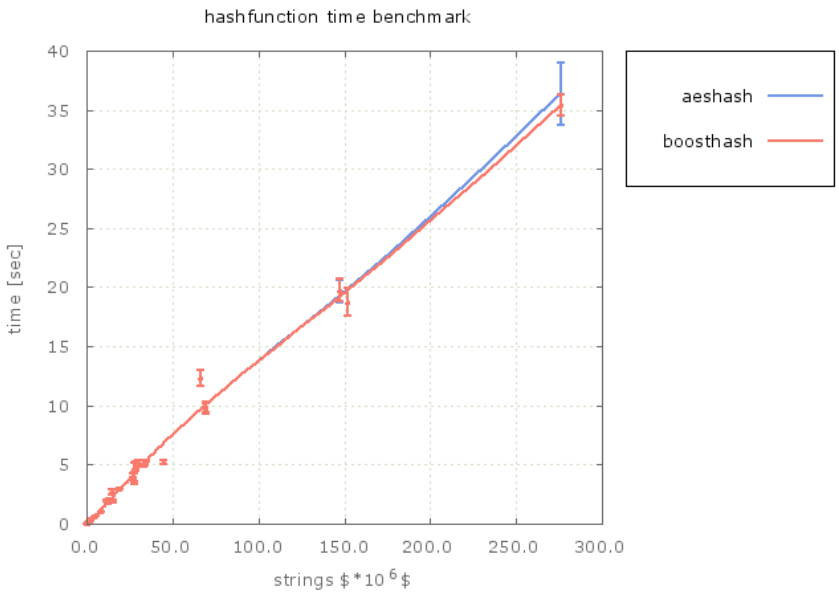

Abbildung 5.4: AES-NI-Hash vs Boost-Hash Zeit für DBpedia Datensätze, dargestellt sind Mittelwerte über 32 Messung mit 99%-tigem Konfidenzintervall

Tabelle 5.2: Prozentualer Zeitbedarf für Hash-Kollisionen im Builddb-Schritt

	boosthash	aeshash
Builddb	3%	0%

In Tabelle 5.2 ist der prozentuale Zeitbedarf der Hash-Kollision im Builddb-Schritt dargestellt. Durch den Einsatz der AES-Hash-Funktion kann erreicht werden, dass nicht mehr Zeit für das Hashen der Zeichenketten benötigt wird, aber weniger Kollisionsfälle behandelt werden. Daher kann ungefähr 3% an der Gesamtzeit eingespart werden (genauer wird hier sicherlich noch mehr Ersparnis stattfinden, da auch weniger Zeichenketten neu gehasht werden müssen).

5.4 Bit-Vektoren

Bei der Analyse des bestehenden Systems konnte ermittelt werden, dass durch die Verwendung der boost-Bitvektoren (als Bitmaps) sehr viel Zeit benötigt wird. Daher sollen im folgenden Abschnitt Verbesserungen bezüglich effizienterer Bitmap-Datenstrukturen betrachtet werden.

In der aktuellen Implementierung des CameLOD-Projekts werden häufig Bitvektoren eingesetzt. Zum Beispiel benutzt ein einfacher Scan für jeden Chunk einen Bitvektor um die Elemente zu markieren, welche das Scan-Prädikat erfüllen. Typisch sind daher Schleifenstrukturen mit folgendem Aufbau:

```
for (size_t i = 0; i < num; i++) {
 res->set_bit(i, XYZBOOL);
}
```

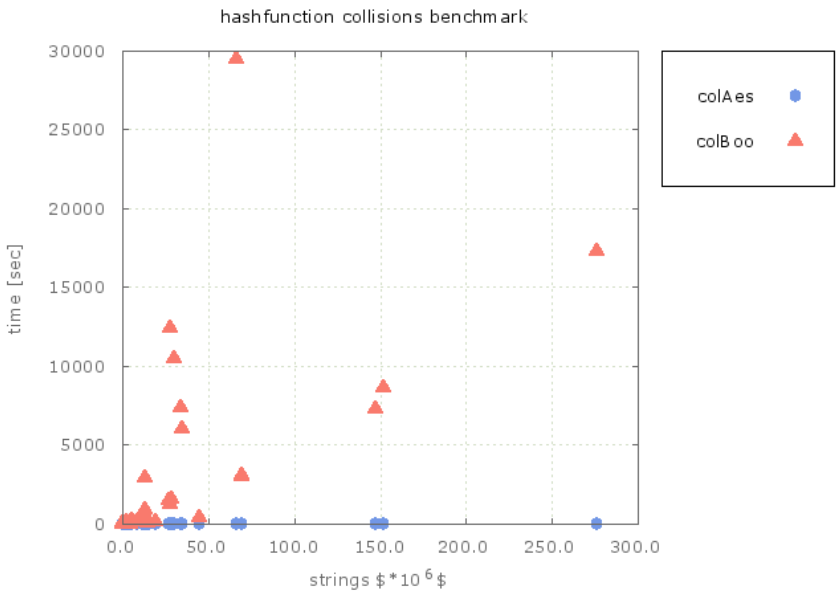

Abbildung 5.5: Summierte Kollisionen beider Hashfunktionen für DBpedia Datensätze

Dabei wird eine Iteration über alle Elemente des Chunks durchgeführt und das Ergebnis einer Prädikatauswertung im Bitvektor gespeichert.

Zur Speicherung der Ergebnisse in allen Operatoren werden BitSets/BitMaps/Bit-Vektoren in dieser Art und Weise eingesetzt. Aktuell wird die Boost-BitSet-Implementierung benutzt, da sie weniger Speicher als eine naive Implementierung benötigt. Eine naive Implementierung würde ein Array bestehend aus Bool Werten nutzen, dabei werden intern für die Darstellung der Bool-Werte Bytes verwendet. Dieser Ansatz benötigt 8 mal mehr Speicher als eine Implementierung auf Bit-Ebene, dennoch soll er als Referenz betrachtet werden, weil er besonders laufzeiteffizient ist.

5.4.1 Benchmark

Durch einen experimentellen Benchmark soll das Optimierungspotential nur durch Austausch beziehungsweise Modifikation der Bitvektoren untersucht werden. Dazu wurden die Laufzeiten beim Festlegen aller Elemente eines Bitvektors durch vorher ermittelte Zufallsbits mit verschiedenen Methoden gemessen.

Varianten der Bit-Vektoren

Insgesamt wurden 6 Bit-Vektor-Implementierungen betrachtet. Die Funktionen der verschiedenen Methoden sollen kurz beschrieben werden.

Std-Bit-Vektoren. Die Standard Bibliothek von C++ bietet eine effiziente Datenstruktur für Bit-Vektoren. Wenn ein std::vector<bool> angelegt wird, so greift eine Optimierung, welche intern Bytes zur Speicherung benutzt und dort jeweils auf Bit-Ebene Operationen durch-

führt (kann durch die Beobachtung des Speicherverbrauchs geschlussfolgert werden, vergleiche [sgi]).

Boost-Bit-Vektoren. Die Bit-Vektor-Implementierung von Boost (genauer wird sie dort als boost::dynamic_bitset benannt) ist flexibler als die Std-Bitvektoren, da sie die Möglichkeit bietet verschiedene Grundelemente als Blöcke zu benutzen.

Zugriffe auf die gespeicherten Bits (speziell beim Setzen eines neuen Bit-Wertes) erfolgen in der Boost-Implementierung anhand der Auswertung des zu setzenden Bit-Wertes (siehe Quelltext-Auszug der Implementierung des Setzens eines Bits innerhalb eines Blocks 7, Quelle [Boob])

Listing 7: Boost bit-set Schreib-Zugriffe

```
...
void do_set() { m_block |= m_mask; }
void do_reset() { m_block &= ~m_mask; }
void do_assign(bool x) { x? do_set() : do_reset(); }
...
```

long. Zum Vergleich wurde eine BitVektor-Implementierung erstellt, welche zur internen Speicherung einen Vektor mit uLong-Werten benutzt. Auf ein dynamisches Verhalten wurde verzichtet, da es für die Verarbeitung in CameLOD nicht benötigt wird (auch dort werden die Boost-Bit-Vektoren am Anfang mit einer festen Anzahl von Elementen initialisiert). Es wurden verschiedene Zugriffsvarianten getestet und festgestellt, dass sich die folgende Zugriffsvariante beim sequentuellen Setzen effizienter als die Boost-Variante erwies. Ein Zugriff zum Setzen eines Bits erfolgt schematisch anhand des Quelltextauszugs 8.

Listing 8: Skizze der Schreib-Zugriffe der eigenen Bit-Set Implementierung

```
const unsigned long _masks[64] = ... // mask for each bit access
inline void set(const long& idx, const bool& value) {
    // calc outer index
    long l = idx / (sizeOfByte * sizeof(long));
    // calc inner index
    long i = idx % (sizeOfByte * sizeof(long));
    if(!(_store[l] & ( _masks[i] )) ) { // current stored bit = 0
        _store[l] |=  _masks[i] * value; // 0 OR 1*X = X
    } else { // current stored bit = 1
        _store[l] &=  ~(_masks[i] * !value); // 1 AND ↘
           →NOT(1*NOT(X)) = X
    }
}
```

Ignoriert man die Berechnung der Indizes für den Zugriff auf den Block und der jeweiligen Maske, so ist deutlich der Unterschied zur Boost-Implementierung erkennbar, denn in der eigenen Variante wird zunächst der Speicherinhalt des angesprochenen Bits überprüft und darauf aufbauend eine Bit-Operation zur Veränderung ausgeführt. Die angegebenen Operationen sind äquivalent mittels Intrinsics als SSE/AVX-Befehle umsetzbar.

sse. Analog zur long-Variante wurde eine Variante mit einem 128 Bit-Integer Typ realisiert. Die Zugriffe erfolgen in ähnlicher Weise, nur dass SSE-Operatoren zum Einsatz kommen.

byte. Die byte-Variante stellt nur eine Referenz dar und ist analog zur long-Variante implementiert worden. Anstelle der uLong-Werte im Vektor wurden uByte-Werte benutzt.

naiv. Eine weitere mögliche Referenz soll die naive Variante darstellen, hierbei wird für ein Bit ein komplettes uByte an Speicher verwendet.

5.4.2 Auswertung

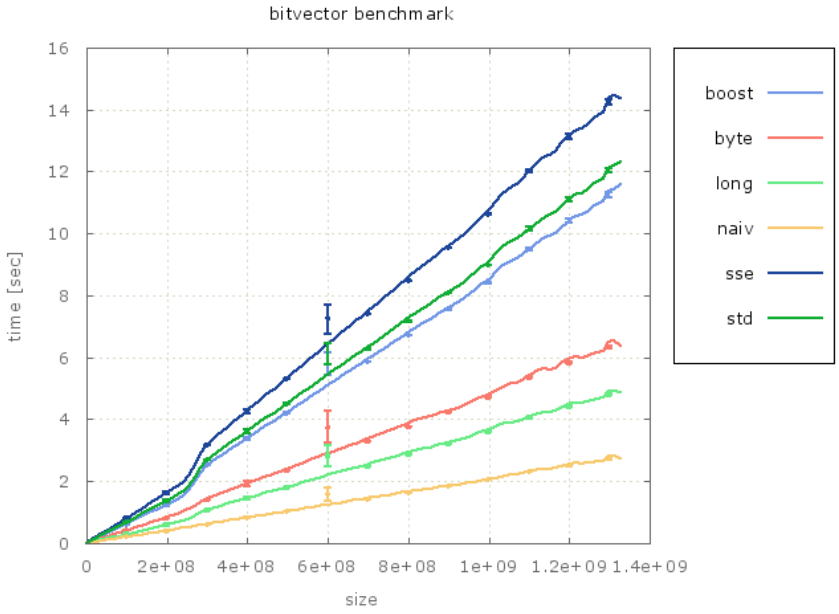

Abbildung 5.6: Bitvektoren-Benchmark, Zeit beim Kopieren von Zufallsbits verschiedener Ansätze, dargestellt sind Mittelwerte über 32 Messungen mit 99%-Konfidenzintervall

Die Abbildung 5.6 stellt die ermittelten Messungen dar. Es ist die Entwicklung der Laufzeiten der verschiedenen Methoden bei Variation der Anzahl der Bitvektorelemente dargestellt. Die dargestellten Laufzeiten sind dabei Mittelwerte über 32 Messungen. Zusätzlich sind Intervalle der jeweiligen Mittelwerte mit 99% Konfidenz visualisiert.

Weiterhin ist in beiden Abbildungen erkennbar, dass die Boost-Datenstruktur in etwa der Std-Variante entspricht. Die schlechteste Variante ist die SSE-Version, auch wenn sie nur geringfügig ($\approx 10\%$) langsamer als die Boost-Variante ist. Grundlegend ist der Ansatz, SSE-Register für einzelne Bit-Operationen zu benutzen nicht effizient, da die SSE-Register für mehrfache Operationen optimiert sind.

Durch die veränderte Zugriffsvariante konnte erreicht werden, dass die byte- und long-Varianten deutlich schneller sind als die Boost-Variante. Die byte-Variante ist cira 40% schneller als die Boost-Variante. Betrachtet man dagegen die long-Variante, so ist sie in etwa 60% schneller als die Boost-Version. Die byte- und long-Varianten unterscheiden sich, weil es auf einer 64 Bit Architektur effizienter ist, Long-Werte zu verarbeiten.

Verständlich ist auch der dargestellte Fakt, dass die naiv-Version allen anderen Versionen in Bezug auf Setzgeschwindigkeit überlegen ist, jedoch aus Speichergründen soll sie nur als Referenz des technisch Möglichen angesehen werden.

Es wurde auch der Speicherverbrauch der verwendeten Datenstrukturen untersucht und festgestellt, dass alle Varianten (außer die naive Variante) etwa gleich viel Speicher benötigen.

5.4.3 Fazit

Durch den Einsatz einer anderen Bit-Vektor-Datenstruktur kann das CameLOD-Projekt profitieren, da die Anfrageergebnisse durch Iteration über die jeweiligen Chunks entstehen und die Ergebnisse von Prädikatauswertungen in Bit-Vektoren gespeichert werden.

Tabelle 5.3: Prozentualer Zeitbedarf für Bit-Set-Operationen im CameLOD-Projekt

	boost	long
Q1	$\approx 1.8\%$	$\approx 1\%$
Q2	$\approx 1\%$	$\approx 0.6\%$
Q3	$\approx 69\%$	$\approx 41\%$
Q4	$\approx 1\%$	$\approx 0.6\%$
Q5	$\approx 56\%$	$\approx 33\%$

In Tabelle 5.3 sind die prozentualen Zeiten für die Bit-Set-Operationen mittels boost für die Anfragen Q1, Q2, Q3, Q4 und Q5 dargestellt. Deutlich erkennbar ist, dass bei vielen Anfragen viel Zeit durch die Bit-Set-Operationen verbraucht wird. Beispielsweise wird bei den Anfragen Q3 und Q5 ungefähr $56 - 69\%$ der Gesamtzeit durch Bit-Set-Operationen verbraucht. Mit der vorgestellten long-Variante kann dieser Anteil auf ungefähr $33 - 41\%$ gesenkt werden. Das bedeutet es kann mindestens 20% Zeit eingespart werden. Bei den Anfragen Q1, Q2 und Q4 haben Bit-Operationen nur einen geringen Anteil an der Verarbeitungszeit, weshalb hier nur wenig Gewinn erzielt werden kann.

5.5 Kompression

CameLOD profitiert in erster Linie aus der verwendeten Wörterbuchkompression zum Speichern der RDF-Daten. Es sollen im nächsten Abschnitt weitere Verbesserungen bezüglich der Kompression der vorhandenen Daten beschrieben werden, welche keinen negativen Einfluss auf die Geschwindigkeit bei der Verarbeitung ausüben. Zunächst wird durch Analyse der gespeicherten Daten theoretisch nachgewiesen, ob bestimmte Kompressionsschemata sinnvoll sind. Weiterhin soll experimentell untersucht werden, ob durch einfache Kompressionsschemata sogar die Verarbeitungsgeschwindigkeit erhöht werden kann, da beispielsweise weniger große Speicherbereiche durchlaufen werden müssen.

5.5.1 Integer-Chunk-Kompression

Die Indizes für Subjekt, Prädikat und Objekt in CameLOD bestehen aus Chunks von Integer-Sequenzen. Jeder Chunk besteht im einfachsten Fall aus 3 Integer-Sequenzen (jeweils für Subjekt, Prädikat und Objekt eine Sequenz). Eine Kompression hinsichtlich einer Sequenz liefert nur geringe Vorteile für die Verarbeitung, daher ist es besser alle zu komprimieren. Besonders wichtig für die Kompression ist weiterhin, dass die Zugriffe auf die unkomprimierten Werte nicht wesentlich langsamer sein dürfen als die Direktzugriffe.

Aus Performance-Gründen entfallen daher schon verschiedene Entropieverfahren (Huffman und ähnliche), weil sie zur Dekompression mehrere elementare Operationen benötigen. Bei Huffman zum Beispiel das Durchlaufen des Huffman-Baums. Außerdem bietet eine Huffman-Kompression keinen wahlfreien Zugriff. Dagegen soll die zusätzliche Kompression möglichst transparent und effizient integrierbar sein. Da die Daten bereits in Teilmengen durch die Chunk-Struktur aufgeteilt sind, bietet sich ein Frame-of-Reference-Verfahren an.

Für die gespeicherten Indizes wurden verschiedene wichtige Werte ermittelt, zunächst:

$max(chunk, column) =$ Maximum einer Spalte eines Chunks

$min(chunk, column) =$ Minimum einer Spalte eines Chunks

Weiterhin wurden die Differenzen des Maximums/Minimums einer Spalte innerhalb eines Chunks betrachtet:

$$diff(chunk, column) = max(chunk, column) - min(chunk, column)$$

Falls eine Integersequenz eine Differenz ihrer Werte von d aufweist, so können die gespeicherten Integer-Zahlen mit $\lceil \log_2(d+1) \rceil$ kodiert werden. Darauf aufbauend wird die Funktion $bits$ definiert:

$$bits(chunk, column) = \lceil \log_2(diff(chunk, column) + 1) \rceil$$

Sie gibt an, wie viele Bits für einen Wert einer Sequenz (Subjekt, Prädikat oder Objekt) innerhalb eines Chunks zur Kodierung benötigt werden. Um möglichst einfach auf die Elemente zuzugreifen, ist es besser nur spezielle Bit-Werte, welche Vielfache von 8 sind (byte-aligned), zu betrachten. Dazu wird die Funktion $ALbits$ festgelegt mit:

$$ALbits(chunk, column) = \lceil bits(chunk, column)/8 \rceil \cdot 8$$

Betrachtet man nun einen speziellen Index (zum Beispiel den Subjektindex), so kann für den Speicherbedarf eines Tripels innerhalb eines Chunks c folgendes festgelegt werden:

$$Tbits(c) = bits(c, subject) + bits(c, predicate) + bits(c, object)$$

Analog zu $ALbits$ wird auch $TALbits$ festgelegt:

$$TALbits(c) = ALbits(c, subject) + ALbits(c, predicate) + ALbits(c, object)$$

Insgesamt ergibt sich für den gesamten Index also ein Speicherbedarf:

$$mem(index) = \sum_{c \in Chunks(index)} c.size() \cdot Tbits(c)$$

und für den aligned-Speicherverbrauch:

$$ALmem(index) = \sum_{c \in Chunks(index)} c.size() \cdot TALbits(c)$$

Um eine Abschätzung über den erwarteten Gewinn dieser Kompressionsmethode zu bestimmen, wurden die beschriebenen Werte für die Indizes Subjekt, Prädikat und Objekt auf Grundlage des DBpedia-Half-Datensatzes (30 GB mit $TupleCount = 214999298$ Tripeln) ermittelt. Die Daten wurden nach dem Laden innerhalb des CameLOD Projekts durch Iteration über alle Chunks aller Indizes extrahiert. Da in dieser Darstellung (vergleiche Tabelle 5.4) nicht alle min/max Werte aller Chunks dargestellt werden können, werden nur die folgenden Werte aufgelistet:

$$maxdiff(index, column) = \max\{diff(c, index, column) | c \in Chunks(index)\}$$

$$maxTbits(index) = \max\{Tbits(c, index) | c \in Chunks(index)\}$$

$$maxTALbits(index) = \max\{TALbits(c, index) | c \in Chunks(index)\}$$

$$maxmem(index) = maxTbits(index) \cdot TupleCount$$

$$compRatio() = compressed/uncompressed$$

Tabelle 5.4: Analyse der Indexspeicherung

$index$	Subjekt	Predikat	Objekt	
$maxdiff(index, subject)$	4526	8318395	8318039	
$maxdiff(index, predicate)$	33356138	10616487	33356138	
$maxdiff(index, object)$	$\approx 1.7 \cdot 10^{19}$	$\approx 1.6 \cdot 10^{19}$	$\approx 1.5 \cdot 10^{19}$	
$maxTbits(index)$	102	111	112	
$maxTALbits(index)$	112	112	120	
$maxmem(index)$	≈ 2.6	≈ 2.8	≈ 2.8	[GB]
$ALmem(index)$	≈ 2.6	≈ 1.1	≈ 2.1	[GB]
$mem(index)$	≈ 2.4	≈ 0.9	≈ 1.8	[GB]
aktueller Speicherbedarf	≈ 4.8	≈ 4.8	≈ 4.8	[GB]
$ALmem\text{-}compRatio$	0.34	0.23	0.44	

Deutlich erkennbar in Tabelle 5.4 ist, dass die maximalen Differenzen für die jeweils sortierte Spalte der Indizes recht gering ausfallen. Grundlegend ist dafür die Sortierung verantwortlich, hier würde sich eine andere Art der Kompression anbieten, beispielsweise eine Delta-Methode. Die Differenzen zwischen zwei aufeinander folgenden Zahlen innerhalb der sortierten Spalte

sollten gering ausfallen. Dennoch ist der Einsatz einer Delta-Methode nicht vorteilhaft, da bei wahlfreiem Zugriff auf ein Element innerhalb des Chunks zunächst eine Rekonstruktion mit allen Vorgängerwerten stattfinden muss. In Tabelle 5.4 sind auch die erzielbaren Kompressionsraten für die aligned-Variante dargestellt. Es ist gut erkennbar, dass eine Kompressionsrate von $0.23 - 0.44$ erzielt werden kann. Dabei bedeutet eine Kompressionsrate von 0.44, dass 56% des Speichers eingespart werden kann. Für einen der Indizes bedeutet das bei einer unkomprimierten Größe von 4.8 GB ein Speichergewinn von 2.6 GB.

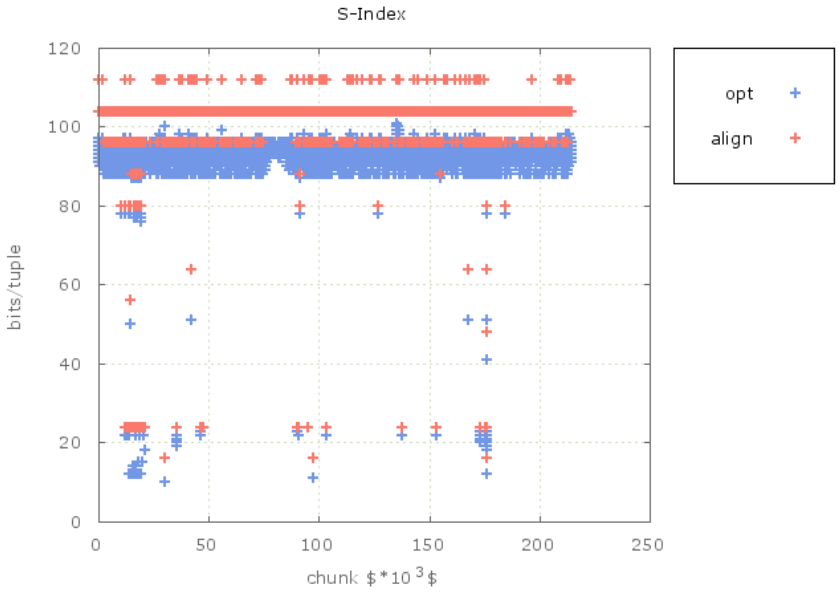

Abbildung 5.7: benötigte Bits für die Tripel Speicherung pro Chunk für den Subjektindex

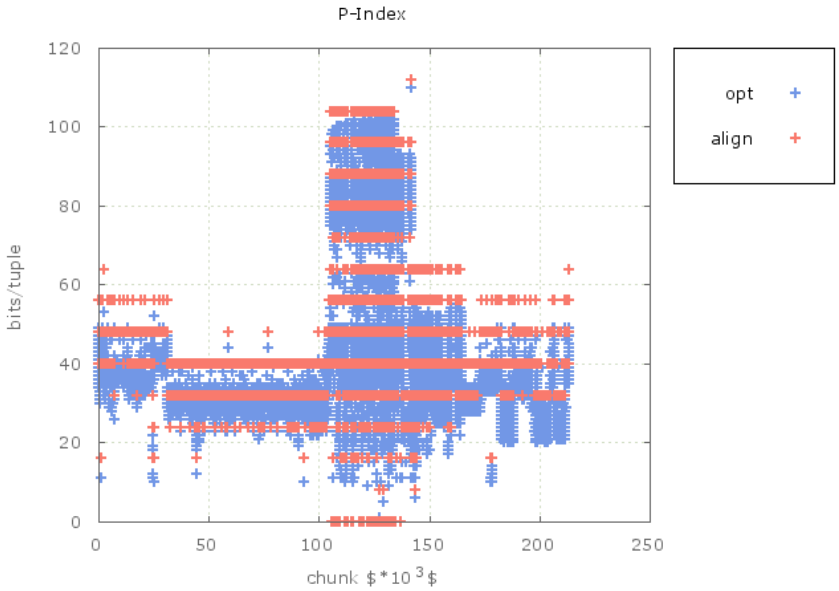

Abbildung 5.8: benötigte Bits für die Tripel Speicherung pro Chunk für den Prädikatindex

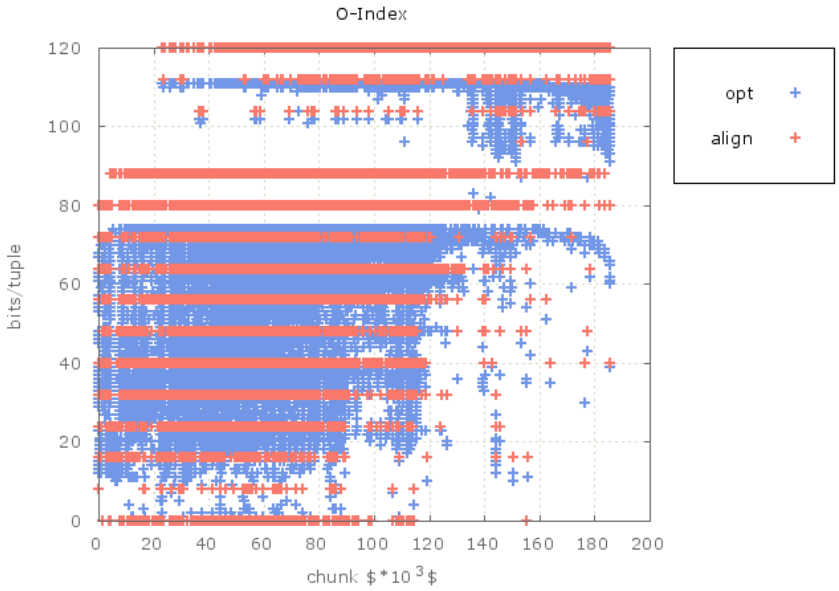

Abbildung 5.9: benötigte Bits für die Tripel Speicherung pro Chunk für den Objektindex

Weiterhin ist in der Tabelle erkennbar, dass selbst die $maxTbits$ Schätzung, welche nur die globalen maximalen Differenzen betrachtet, bereits Werte unterhalb 192 Bits liefert. Alle drei Indizes könnten demzufolge mit 128 Bits kodiert werden. Eine reine 128 Bit Kodierung bietet aber wiederum wenig Vorteile, da ein großer Teil der Chunks (vergleiche Abbildungen 5.7, 5.8, 5.9, dargestellt ist die optimale und aligned-Variante) sogar weniger Bits benötigen. Beispielsweise ist in Abbildung 5.8 deutlich zu erkennen, dass für mehr als 80% der Chunks weniger als 64 Bits in beiden dargestellten Varianten nötig sind.

Bei der Analyse der Daten wurden weitere potentielle Speichergrößen zum Vergleichen ermittelt. Die $mem(index)$ Werte geben dabei den Speicherbedarf bei reiner FOR-Kompression an. Deutlich erkennbar ist, dass beim Prädikatindex sehr viel Speicher ($\approx 80\%$) eingespart werden kann. Problematisch an der reinen FOR-Methode sind die Größen die kein Alignment bieten, so kann es durchaus möglich sein, dass für das Subjekt 2 Bits und für das Prädikat 9 Bits in Betracht gezogen werden. Da bei der Speicherung aber Byte-Grenzen eingehalten werden müssen, sind bei der reinen Methode verschiedene Bitmasken nötig, um die gespeicherten Werte später zu dekomprimieren. Die Dekompression ist daher aufwändiger.

Einen ähnlichen Ansatz verfolgt die als $ALmem(index)$ angegebene Methode. Hierbei werden keine krummen Elementbreiten verwendet, das bedeutet, dass alle berechneten Bits für eine Spalte sind Vielfache von 8. Durch diesen Kompromiss können zwei grundlegende Probleme ausgeschlossen werden. Zunächst entfällt das Dekomprimieren mittels verschiedener Masken (und den dadurch erhöhten Einsatz von Bit-Operationen). Außerdem kann zum Speichern ein Byte-Vektor mit geringem Overhead eingesetzt werden. Zusätzlich zeigen die Abbildungen (5.7,5.8 und 5.9), dass die aligned-Variante nur gering mehr Speicher im Vergleich zum reinen FOR-Verfahren benötigt. Die Kompression und Dekompression ist durch die Vereinfachung auf aligned-Elementbreiten mit wenigen elementaren Operationen möglich.

Kompressions- und Dekompressions-Methode

In diesem Abschnitt soll die Kompression beziehungsweise Dekompression einer Integer-Sequenz dargestellt werden. Der Kompressionsschritt teilt sich dabei in zwei Schritte. Zunächst wird von dem zu komprimierenden Vektor (*INValues*) das Minimum min und Maximum ermittelt, anschließend werden die benötigten Bits mittels $ALbits()$ berechnet. Darauf aufbauend kann die Anzahl benötigter Bytes berechnet werden: $n = ALbits()/8$. Im zweiten Schritt werden alle Elemente e der *INValues*-Sequenz durchlaufen und intern wird der Wert $e - min$ in einem passenden n-Byte-Array[1] abgelegt. Das n-Byte-Array ist in der C++-Implementierung ein reines Byte-Array, bei dem Zugriffe über Indexberechnungen erfolgen. Genauer betrachtet, werden einfach die nötigen Bytes der Originalsequenz in das n-Byte-Array kopiert.

Das Dekomprimieren besteht nur aus dem Ermitteln des Indizes innerhalb des n-Byte-Arrays, dem Extrahieren/Laden der dort gespeicherten n-Byte-Zahl und dem anschließenden Addieren des min-Wertes. In der C++-Implementierung wird die Dekompression mittels Cast auf ein Long-Array und einer vorberechneten Maske ermittelt.

Benchmark

Bisher wurde ohne Nachweis behauptet, dass die dargestellte Kompressionsmethode effiziente Zugriffe auf die Chunks erlaubt und eine gute Kompressionsrate aufweist. In einem experimentellen Benchmark wurde ein Vergleich zwischen den Zugriffszeiten einer unkomprimierten (un) sowie komprimierten Integer-Sequenz (comp) durchgeführt. In dem Benchmark wurden zuerst zufällige Long-Werte (mit festgelegter Bit-Anzahl) in einem Vektor gespeichert, anschließend wurde für beide Varianten (un, comp) die benötigte Zeit zur Summation aller Elemente ermittelt. Außerdem wurde eine modifizierte Summation (compM) bei der komprimierten Variante betrachtet, dabei wurden nur die Rohwerte des komprimierten Vektors zusammen mit $\min \cdot elements$ addiert. Prinzipiell kann dieses Vorgehen auch bei komplexeren Operationen (wie zum Beispiel bei den CameLOD-Operatoren) verwendet werden.

Es wurde bei den Messungen die Bit-Anzahl im Bereich [0,64] variiert. In den Abbildungen 5.10 (7 Bits), 5.11 (24 Bits) und 5.12 (64 Bits) ist eine repräsentative Auswahl der Ergebnisse dargestellt. Allgemein wurde bei einer festen Bit-Anzahl die Menge an Integer Elementen verändert (von 0 bis ≈ 50 Millionen Elementen) und die benötigte Zeit für alle drei Varianten ermittelt. Es sind Mittelwerte über 32 Messungen und Intervalle mit 99% Konfidenz dargestellt.

Auffallend ist, dass alle Varianten sehr dicht beieinander sind. Dennoch ist in 5.10 und 5.12 die unkomprimierte Variante am schnellsten. Für Abbildung 5.12 ist dies einfach zu begründen, weil bei der Dekompression mehr als ein reiner Speicherzugriff stattfindet. In Grafik 5.11 ist aber auch ein Fall erkennbar, in dem die modifizierte Variante (compM) schneller als die anderen beiden Varianten ist.

In Abbildung 5.13 ist die summierte Laufzeit über alle Messfolgen, gruppiert nach der verwendeten Bit-Anzahl, dargestellt. Es zeigt sich, dass die compM-Variante im Bereich [0,32] die schnellste ist (ungefähr 15% schneller als die unkomprimierte). Dagegen sind bei 64 Bit die komprimierten Varianten circa 20% langsamer. Gut erkennbar ist weiterhin, dass die unkomprimierte Variante im kompletten Bit-Bereich nahezu konstant ist. Als Ursache dafür, das

[1] ein n-Byte-Array soll ein Array bestehend aus n-Byte großen Integer Werten sein

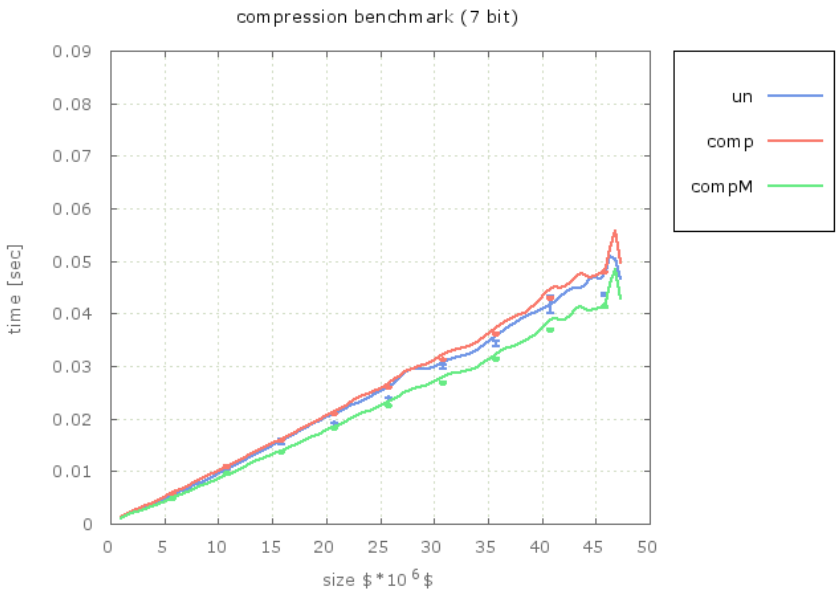

Abbildung 5.10: Kompressions-Benchmark mit 7 Bits, dargestellt ist der Mittelwert über 32 Messungen mit 99%-Konfidenzintervall

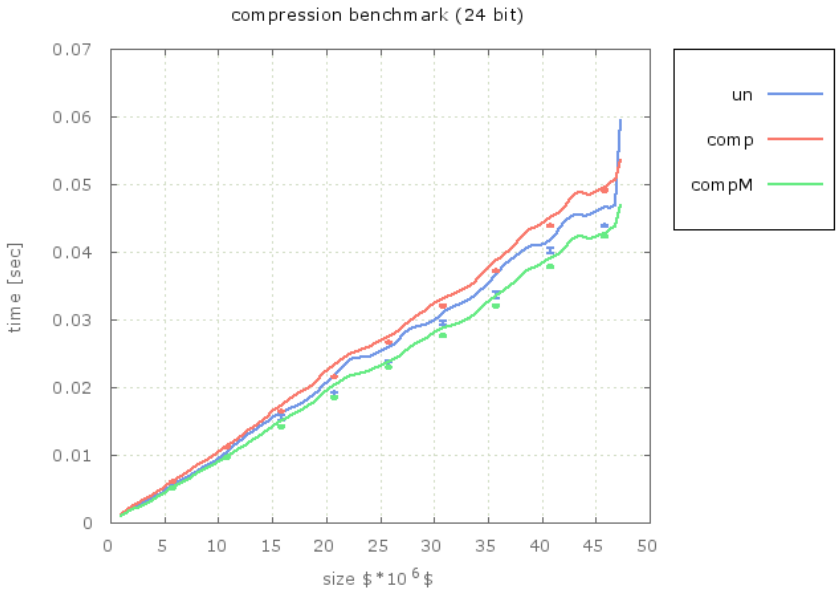

Abbildung 5.11: Kompressions-Benchmark mit 24 Bits, dargestellt ist der Mittelwert über 32 Messungen mit 99%-Konfidenzintervall

die komprimierten Varianten bei höheren Bits (ab 32) langsamer sind, ist ,dass Long (64 Bit) Zugriffe sehr effizient auf 64-Bit Architekturen sind. Dagegen sind Zugriffe auf einzelne Bytes weniger effizient. Dennoch kann im Bereich [0,32] ein erheblicher Gewinn mit der compM-Variante erzielt werden. Maßgeblich verantwortlich dafür ist der veränderte Zugriff und der Fakt, dass insgesamt weniger Speicherzellen geladen werden müssen.

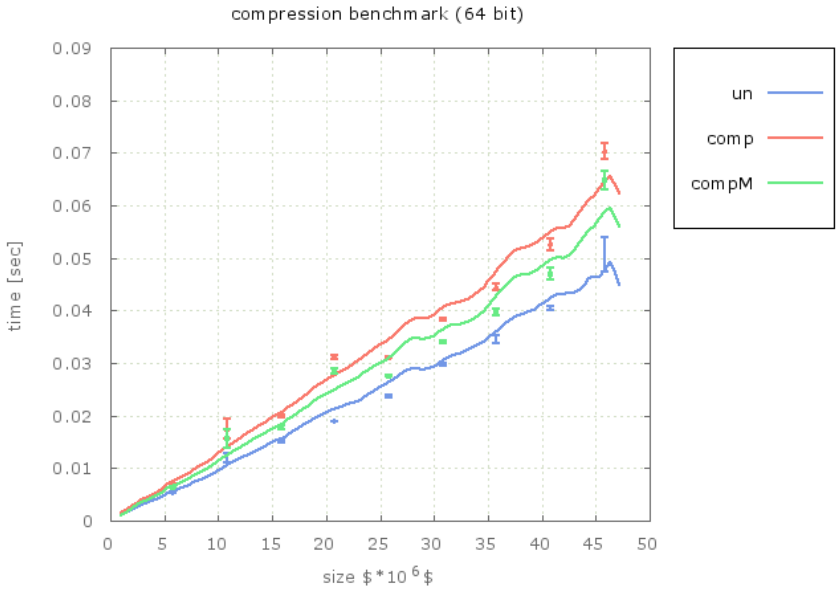

Abbildung 5.12: Kompressions-Benchmark mit 64 Bits, dargestellt ist der Mittelwert über 32 Messungen mit 99%-Konfidenzintervall

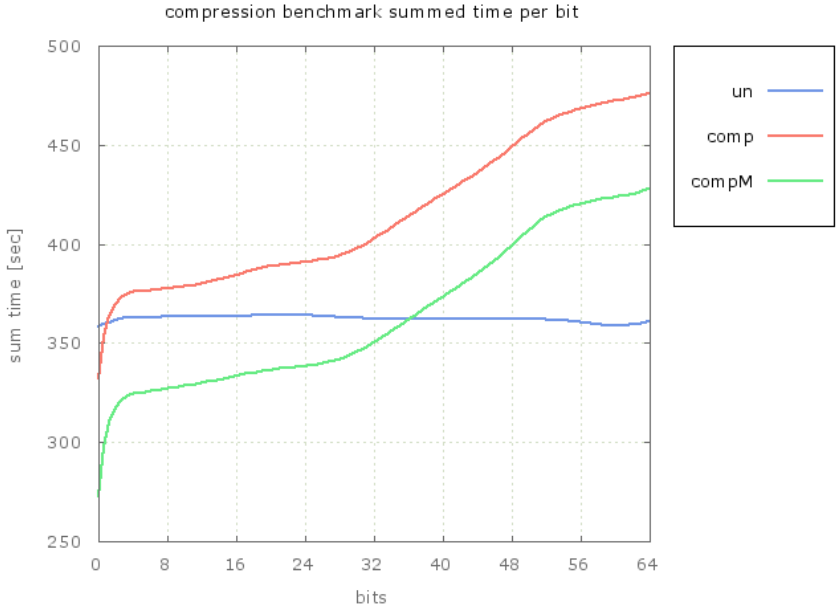

Abbildung 5.13: Summierte Zeit aller durchgeführten Messungen gruppiert nach Bit-Anzahl

Betrachtet man dagegen den Speicherverbrauch, so ist er bei der unkomprimierten Variante unabhängig von der Bit-Anzahl, demzufolge 64 Bit pro Element. Dagegen folgt für die komprimierten Varianten $bits$ pro Element. Daher lässt sich eine Kompressionsrate von $bits/64$ erzielen.

Fazit

Es konnte gezeigt werden, dass FOR in einer modifizierten Variante sehr gute Kompressionsraten im CameLOD-Projekt erzielen kann. Dabei können mindestens 50% des Speichers der Indexstrukturen eingespart werden. Gerade für die sortierte Spalte des jeweiligen Index kann mittels FOR sehr viel Speicher eingespart werden. Für das CameLOD-Projekt können mittels veränderter Zugriffsmethoden auf die komprimierten Sequenzen annähernd die gleichen oder sogar besseren Zugriffszeiten realisiert werden. Im CameLOD-Projekt müssen verschiedene Spaltentypen eingeführt werden, da die komprimierten Sequenzen nur Leseoperationen zur Verfügung stellen. Dagegen sind bei Join-Operationen Ergebnis-Chunks nötig, das heißt in den Sequenzen müssen Werte dynamisch eingefügt werden.

5.5.2 Integer-Sequenzen Run-Length-Encoding

Im CameLOD-Projekt besteht ein Chunk jeweils aus einer sortierten Integer-Sequenz, da beim Aufbau zunächst die Zeichenketten sortiert und anschließend fortlaufend nummeriert werden. Daher ergibt sich beispielsweise für die sortierte Spalte eines Chunks folgender schematischer Aufbau:

$$S = [aaaaabbbbbbbccccccc], \text{ mit } a \neq b \neq c$$

Sequenzen dieser speziellen Struktur können effizienter als mit den bisherigen Verfahren gespeichert werden. Eine Sortierung ist nicht zwingend erforderlich, sondern nur das Auftreten von Zahlenblöcken.

Verfahren

Es soll eine Array-ähnliche Struktur realisiert werden, die transparenten und effizienten Zugriff auf die sortierte Zahlensequenz bietet sowie das Run-Length-Encoding (RLE)-Verfahren benutzt. Zunächst werden in der normalen Variante die Zahlen mehrfach gespeichert. Effizienter ist dagegen die einfache Speicherung der Zahlen in folgender Form (beispielsweise bezogen auf S):

$$M = [abc]$$

Betrachtet man nun ein einen Zugriff auf $S[i]$ muss das passende Element mittels M rekonstruiert werden. Hierfür wird eine weitere Sequenz P benötigt. P speichert die ersten auftretenden Positionen der jeweiligen Elemente aus M in S und für das Beispiel ergibt sich

$$P = [0, 5, 13].$$

Soll nun ein Zugriff auf beispielsweise $i = 7$ erfolgen, so wird mittels binärer Suche (denn P ist sortiert) der Index $j = \min\{k \mid P[k] \geqq i\} = 1$ ermittelt und anschließend wird $M[j] = M[1] = b$ zurückgegeben.

Wahlfreier Zugriff ist in wenigen Fällen in CameLOD erforderlich, meist wird über eine komplette Spalte iteriert. Da es wenig effizient ist, jeweils eine binäre Suche bei der Interation

auszuführen, soll nun auch noch eine Möglichkeit beschrieben werden, wie eine effiziente Iteration über die dargestellte Speicherung erfolgen kann (es wird weiterhin auch die Anzahl an Elementen, die in der ursprünglichen Sequenz abgelegt sind gespeichert und mit $size(M)$ angesprochen). Die Funktion $csize(M)$ gibt die Anzahl von Elementen in der komprimierten Sequenz an.

```
// orginal:
for(size_t i = 0; i < size(S); i++ ) {
    doSomethingWith(S[i]);
} // size(S) memory access
// compressed variant:
j = 0;
c = M[j]
nextP = P[j+1]
for(size_t i = 0; i < size(S); i++ ) {
    if(i>= nextP) {
        j++;
        c= M[j]
        nextP = P[j+1]
    }
    doSomethingWith(c);
} // |M| + |P| times memory access
```

In der optimierten Variante wird direkt auf die Sequenzen M und P zugegriffen und falls $nextP$ überschritten wird, das aktuelle Element c aktualisiert. Daher ist nicht in jedem Schritt eine binäre Suche notwendig.

Analyse

Das Verfahren kann nicht direkt in einem Benchmark evaluiert werden, da die Kompression datenabhängig ist. Um dennoch eine Auswertung zu ermöglichen, erfolgten Messungen im CameLOD-Projekt mit Hilfe der gespeicherten Daten. Es wurde über alle Chunks aller Indizes iteriert und die gespeicherten Sequenzen für Subjekt, Prädikat und Objekt mit dem Run-Length-Verfahren komprimiert. Die verwendete Array-Klasse bietet nur reine Kompressionsfähigkeiten. Für die Sequenz der Positionen folgte als Grunddatentyp uShort (16 Bit), da die Chunks nicht größer als 2000 Elemente sind ($\log_2(2000) \approx 11 \approx 2\ Bytes$). Auf eine weitere Kompression der Sequenzen mit dem FOR Verfahren wurde verzichtet.

Zur Auswertung wurden folgende Werte ermittelt:

$$compressedSize(chunk) = (8+2) \cdot \sum_{col \in \{sub,pred,obj\}} csize(chunk.col)$$

$$size(chunk) = size(chunk) \cdot 24$$

Beide Werte sind in Bytes und für den gesamten Index folgt:

$$compressedSize_{glob}(index) = \sum_{c \in index} compressedSize(c)$$

$$size_{glob}(index) = \sum_{c \in index} size(c).$$

In die Berechnung der komprimierten Größe $compressedSize$ fließt auch der Overhead durch das Speichern der Positionen ein (2 Bytes für jeweils einen uShort Wert).

Weiterhin wurde auch die Blockgröße untersucht und für einen Chunk $chunk$ und einer Spalte col (beispielsweise Subjekt, Prädikat oder Objekt) die Werte

$$maxBlockSize(chunk, col) = \text{ größte Blockgröße innerhalb eines Chunkes einer Spalte}$$

$$minBlockSize(chunk, col) = \text{ kleinste Blockgröße innerhalb eines Chunkes einer Spalte}$$

ermittelt. Für den Gesamten Index wurden anschließend die Werte

$$maxBlockSize_{glob}(index, col) = \max\{maxBlockSize(c, col) \mid c \in Chunks(index)\}$$

$$minBlockSize_{glob}(index, col) = \min\{minBlockSize(c, col) \mid c \in Chunks(index)\}$$

berechnet. Für eine komprimierte Sequenz v sei die Auslastung $\alpha(v)$ als

$$\alpha(v) = \frac{csize(v)}{size(v)}$$

festgelegt. Darauf aufbauend wurde für einen Index die durchschnittliche Auslastung α_{avg} mit

$$\alpha_{avg}(index) = \frac{\sum_{c \in Chunks(index)} (\alpha(c.sub) + \alpha(c.pred) + \alpha(c.obj))}{3 \cdot |Chunks(index)|}$$

ermittelt.

Tabelle 5.5: Analyse der Indexspeicherung

$index$	Subjekt	Predikat	Objekt	
$maxBlockSize_{glob}(index, subject)$	1998	1001	1027	
$minBlockSize_{glob}(index, subject)$	1	1	1	
$maxBlockSize_{glob}(index, predicate)$	1961	1999	1999	
$minBlockSize_{glob}(index, predicate)$	1	1	1	
$maxBlockSize_{glob}(index, object)$	79	1459	2000	
$minBlockSize_{glob}(index, object)$	1	1	1	
$compressedSize_{glob}(index)$	≈ 2.8	≈ 2.7	≈ 3.1	[GB]
$size_{glob}(index)$	≈ 4.8	≈ 4.8	≈ 4.8	[GB]
$\alpha_{avg}(index)$	≈ 0.47	≈ 0.45	≈ 0.52	
$compRatio$	≈ 0.58	≈ 0.56	≈ 0.64	

In Tabelle 5.5 sind die ermittelten Werte aufgelistet. Allgemein ergeben sich sehr hohe maximale Blockgrößen, das bedeutet, es gibt Chunks, in denen in einer Spalte nur ein oder wenige Werte stehen. Betrachtet man die Blockgröße von 1999 so können dort nur maximal 2 Werte gespeichert sein, da die maximale Chunk-Größe 2000 beträgt. In diesem Fall ergibt sich eine sehr hohe Kompression, da nur 2 uLong Werte und 2 uShort Werte in der komprimierten Sequenz gespeichert werden müssen. Im Vergleich zur unkomprimierten Version mit

$2000 \cdot 24\ Byte \approx 46\ KB$ Speicherbedarf ergibt sich $(8+2) \cdot 2\ Byte = 20\ Byte$, und somit eine Kompressionsrate von 0.04. Es kann daher für einige Spalten über 96% an Speicher eingespart werden. Bei der Analyse wurde auch der gesamte Speicherbedarf berechnet. Für alle drei Indizes kann festgestellt werden, dass sehr viel Speicher bei der RLE-Kompression eingespart wird. Die Kompressionsraten der Indizes liegen alle bei ungefähr 60%. Demnach kann circa 40% an Speicher durch das RLE-Verfahren eingespart werden. Der durchschnittliche Auslastungsfaktor α_{avg} für die Indizes ist bei ungefähr 0.5. Das bedeutet, dass die komprimierte Sequenz ungefähr halb so viele Elemente wie die unkomprimierte besitzt.

Abbildung 5.14: benötigte Bytes für die Tripel Speicherung mittles RLE pro Chunk für den Subjektindex

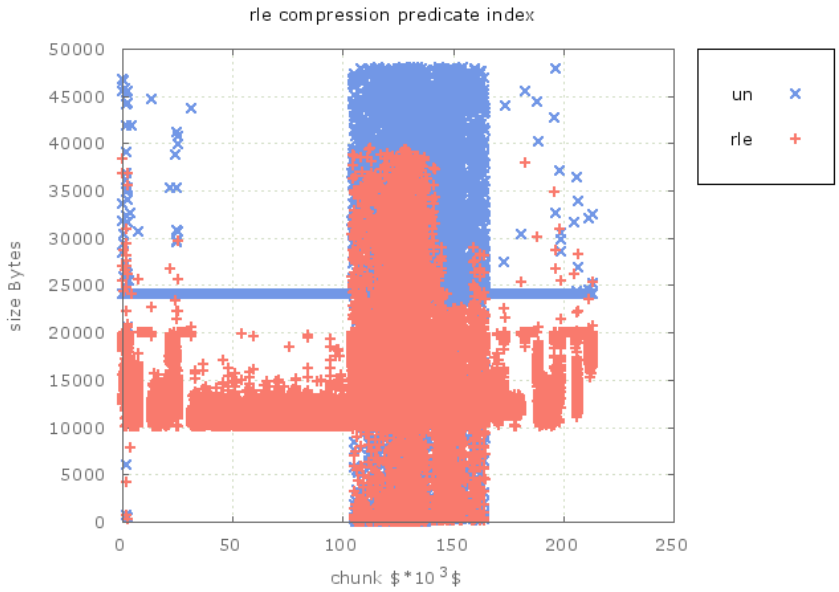

Abbildung 5.15: benötigte Bytes für die Tripel Speicherung mittles RLE pro Chunk für den Prädikatindex

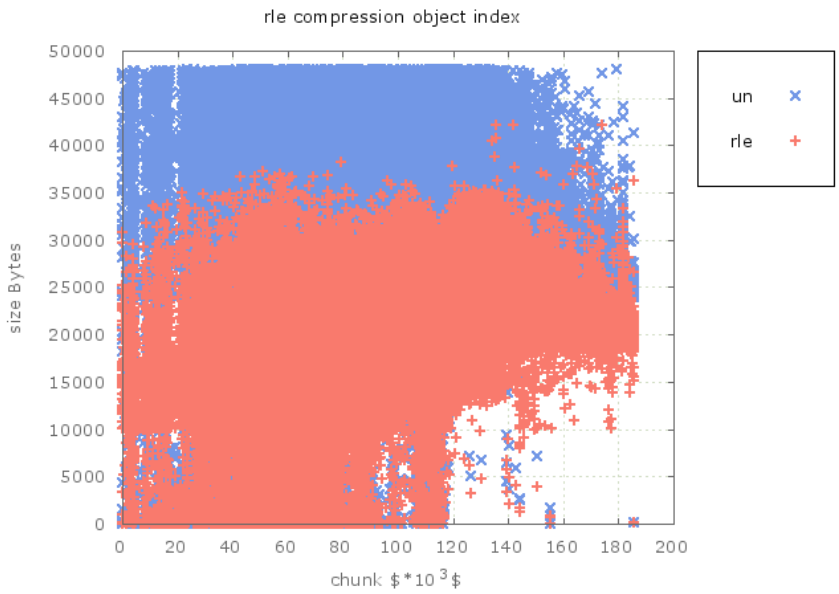

Abbildung 5.16: benötigte Bytes für die Tripel Speicherung mittles RLE pro Chunk für den Objektindex

In den Abbildungen 5.14, 5.15 und 5.16 sind die benötigten Bytes für die Tripel-Speicherung der drei Indizes dargestellt. Dabei bedeutet 'rle', dass die beschriebene RLE-Kompression verwendet wurde und 'un' steht für die unkomprimierte Variante. In allen drei Diagrammen ist deutlich sichtbar, dass viel Speicher eingespart werden kann. Obwohl für das RLE-Verfahren zusätzlicher Speicher für die Positionen benötigt wird. Durch die Speicherung der Positionen in einem uShort-Vektor wurde der Overhead verringert. Daher benötigt die 'rle'-Methode in allen drei Indizes weniger Speicher als aktuell verwendet wird. Für den Subjekindex können beispielsweise ungefähr 40% an Speicher eingespart werden. Auch für die anderen Indexstrukturen ergeben sich ähnlich hohe Kompressionsraten. Hauptsächlich kann durch das RLE-Verfahren sehr gut die sortierte Spalte komprimiert werden, da sie viele Blöcke aufweist. Aber auch die anderen Spalten weisen häufig Blockstrukturen auf.

Benchmark

Das dargestellte Kompressionsverfahren muss auch effiziente Zugriffe auf die Elemente bieten. Ein Benchmark soll die unkomprimierte Version mit der komprimierten vergleichen. Dabei wurden zufällige Werte (uLong) ermittelt und anschließend eine Summation über alle Werte durchgeführt. Die zufälligen Werte werden dabei sortiert und mittels des Auslastungsparameters α wird die Anzahl an möglichen Zufallszahlen variiert um die Kompressionsrate zu steuern. Durch eine Sortierung der Zufallssequenz kann erreicht werden, so dass Blöcke entstehen, welche dem Auslastungsfaktor entsprechen.

In der Analyse wurden Auslastungsfaktoren von $\alpha \approx 0.5$ ermittelt. Daher sollen im Benchmark nur insgesamt zwei α-Werte (0.4 und 0.5) untersucht werden. Die Größe der Sequenzen wurde von 10^6 bis $45 \cdot 10^6$ in 10^5 Schritten angepasst. Im Gegensatz zur beschriebenen Implementierung benutzt die Version für den Benchmark uLong-Werte für die Positionen, da sonst nur 2^{16} Elemente möglich wären.

Insgesamt wurden vier Varianten betrachtet. Die un-Variante ist die Summation der unkomprimierten Sequenz als Vergleichswert. Die Variante (bi) benutzt binäre Suche bei jedem Elementzugriff der komprimierten Sequenz. Weiterhin ist (mod) der modifizierte sequentielle Zugriff auf die komprimierten Werte. Und (calc) ermittelt die Summation basierend auf einer Multiplikation. Da über die Positionsangaben die Blockgröße eines aktuellen Elementes ermittelt werden kann, ist es möglich die Summe mittels Multiplikation zu berechnen. Ähnliche Modifikationen sind auch bei der Anfrageverarbeitung möglich. In Abbildung 5.17 sind die Messwerte für $\alpha = 0.4$ und in Diagramm 5.18 für $\alpha = 0.5$ dargestellt. Es wurden Mittelwerte von 32 Messungen und Intervalle mit 99% Konfidenz berechnet.

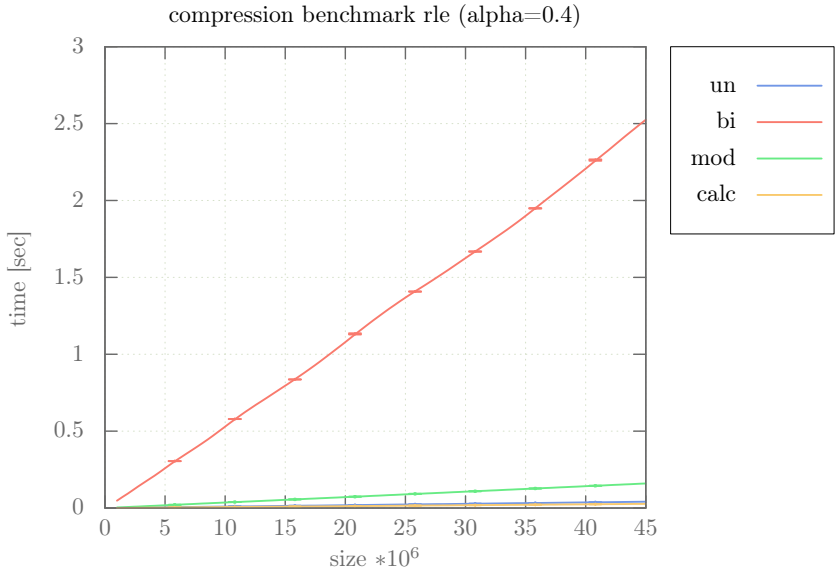

Abbildung 5.17: RLE Kompressions-Benchmark mit $\alpha = 0.4$, dargestellt sind Mittelwerte über 32 Messungen mit 99% Konfidenzintervall

In Grafik 5.17 und 5.18 ist erkennbar, dass die bi-Variante nicht sehr effizient ist. Hingegen sind die beiden anderen Varianten (mod und calc) annähernd so gut wie die unkomprimierte Version (un). Die Werte für un und calc sind nahezu identisch. Bei den Raten 0.5 und 0.4 können etwa 40% Speicher eingespart werden, da der Overhead für die Speicherung der Positionen mit einbezogen werden muss.

Fazit

Die Idee der Run-Lenght-Kodierung zeigt nur bei Sequenzen mit Wiederholungen gute Ergebnisse. Die im CameLOD gespeicherten Daten weisen eine sehr hohe Wiederholungsrate auf, da beispielsweise Spalten sortiert sind. Daher kann bei der ermittelten Auslastung von $\alpha = 0.5$ effizient auf die komprimierten Werte zugegriffen werden. Je geringer die Auslastung, desto effizienter sind die Zugriffe, wenn sie nicht wahlfrei erfolgen. Auch der Overhead durch das Speichern der Positionen kann durch geeignete Wahl des Datentyps ausgeglichen werden. Es ist auch denkbar, die im RLE-Verfahren benutzten Sequenzen M und P mittels des vorher beschriebenen FOR-Verfahren stärker zu komprimieren.

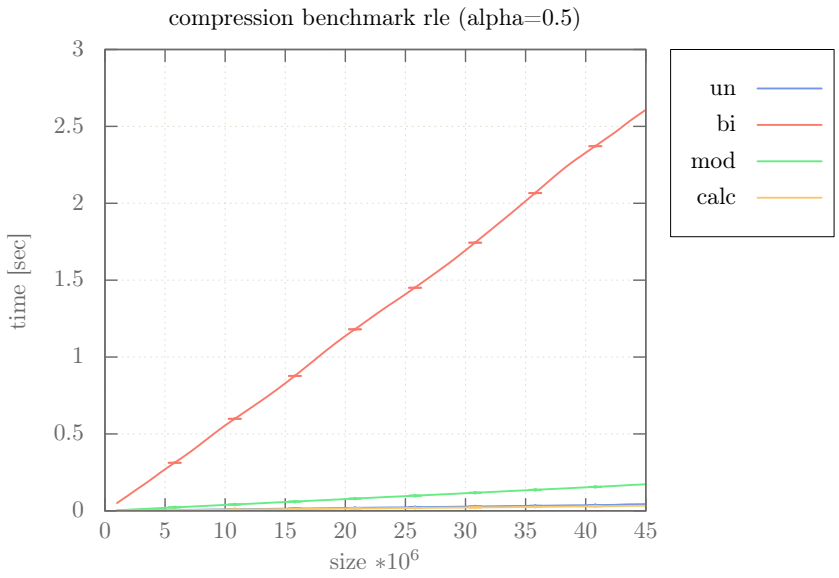

Abbildung 5.18: RLE Kompressions-Benchmark mit $\alpha = 0.5$, dargestellt sind Mittelwerte über 32 Messungen mit 99% Konfidenzintervall

5.5.3 Dictionary

Alle Zeichenketten werden in der CameLOD-Implementierung sortiert in einer Datenstruktur gespeichert. Typische Zeichenketten von RDF-Tripel weisen aber hohe Ähnlichkeiten in ihrem Aufbau auf, da die Subjekt- und Prädikatangaben meist URIs sind. Daher besitzen sie zum großen Teil die gleichen Präfixe, welche aktuell mehrfach gespeichert werden. Es ist also möglich durch den Einsatz einer anderen Datenstruktur Speicher zu sparen. Eine mögliche Datenstruktur für solche Zeichenketten ist ein Präfix-Tree / Patricia Trie.

Im folgenden Abschnitt soll untersucht werden, ob sich der Einsatz eines solchen Trie's lohnt. Denn auch wenn Speicher eingespart wird, entsteht eine möglicherweise erhöhte Zugriffszeit auf die Zeichenketten. Ursache dafür könnte die Baumstruktur und die damit verbundenen Zeigeroperationen sein. Auch kann durch die Verwaltung des Baums ein Speicher-Overhead entstehen, da für die Verkettung 64 Bit-Pointer notwendig sind.

Trie

Zur experimentellen Untersuchung wurde ein einfacher Präfix-Tree implementiert. Zunächst wurden alle Zeichenketten in einen Trie (trie) eingefügt. Der Trie besitzt keine Pfadkompression, weshalb jedes nicht zu einem Präfix gehörende Zeichen in einem neuen Knoten gespeichert wird. Beim Einfügen findet auch die Sortierung statt. Anschließend dient der Trie als Grundlage für den Aufbau eines Patricia-Tries (compTrie). Der compTrie komprimiert die Pfade des Tries, daher werden weniger Knoten benötigt. Mittels der in CameLOD ermittelten Zeichenketten des DBpedia-Half-Datensatzes wurde ein reduzierter Datensatz (mit maximaler Zeichenkettenlänge von 80 Zeichen) erzeugt, da viele Zeichenketten der RDF-Tripel des DBpedia-Half Datensatzes sehr lang sind. Diese Daten wurden nun zunächst in einen Trie eingefügt und anschließend mittels compTrie komprimiert.

verschiedene Zeichenketten	41119285	
Speicherung in Array	≈ 1.71	[GB]
Speicherung in trie	≈ 11	[GB]
Speicherung in compTrie	≈ 2.2	[GB]

Tabelle 5.6: Kompressionsraten des Wörterbuchs mittels Tries

In Tabelle 5.6 sind die ermittelten Werte aufgelistet. Gut erkennbar ist, dass die trie Variante keine gute Kompression aufweist, da für den Baum sehr viele Pointer nötigt sind. Außerdem zeigt die compTrie Version keine guten Ergebnisse, hauptsächlich begründet durch einen hohen Anteil von Zeichenketten die keine gemeinsamen Präfixe besitzen. Auch wenn der Trie eine interessante Datenstruktur darstellt, eignet er sich nicht gut für die Kompression aller RDF-Zeichenketten.

Präfixe

Aus den Beobachtungen des DBpedia-Datensatzes kann geschlussfolgert werden, dass ungefähr 70% aller Zeichenketten URIs darstellen. URIs von Linked-Data besitzen sehr häufig gemeinsame Präfixe. Beispielsweise

$$< http://dbpedia.org/resource/Star\_Trek:\_Enterprise >,$$

$$< http://dbpedia.org/resource/Category:Star\_Trek\_music >$$

und

$$< http://dbpedia.org/ontology/abstract >$$

besitzen das gemeinsame Präfix $< http://dbpedia.org/$. Eine Idee ist es, häufig auftretende Präfixe verkürzt zu speichern.

Für die URIs wurden verschiedene Präfixmuster betrachtet. Zunächst wurden in Variante (A) die kürzesten Präfixe, welche mit `rA ="(<http://.*\\..*?/).*"` übereinstimmen, extrahiert, wie beispielsweise

$'<http://dbpedia.org/ressource/A/B..'$ match $'<http://dbpedia.org/'$.

Der reguläre Ausdruck `rA` beschreibt dabei den Teil der URI bis zum ersten Auftreten des '/'-Zeichens nach dem Protokoll-Teil.

In Variante (B) wurden dagegen die längsten Präfixe, welche mit `rB ="(<http://.*\\..*/).*"` übereinstimmen, ermittelt, beispielsweise

$'<http://dbpedia.org/ressource/A/B..'$ match $'<http://dbpedia.org/resource/A/'$.

Das Muster `rB` ermittelt das Präfix der URI bis zum letzten vorkommenden '/'-Zeichen. Anschließend wurden die Häufigkeiten der beiden Varianten ermittelt.

URIs	(A)	% Anteil
<http://dbpedia.org/	11501749	28
<http://en.wikipedia.org/	10992548	27
<http://upload.wikimedia.org/	681870	2
<http://fr.dbpedia.org/	522075	1
<http://de.dbpedia.org/	448874	1
<http://es.dbpedia.org/	400468	1
<http://it.dbpedia.org/	399061	1
<http://pl.dbpedia.org/	356740	1
<http://pt.dbpedia.org/	337704	1
<http://ru.dbpedia.org/	336413	1

Tabelle 5.7: 10 häufigsten URI Präfixe – Variante (A)

URIs	(B)	% Anteil
<http://dbpedia.org/resource/	11429222	28
<http://en.wikipedia.org/wiki/	10954262	27
<http://fr.dbpedia.org/resource/	521551	1
<http://de.dbpedia.org/resource/	441340	1
<http://es.dbpedia.org/resource/	399635	1
<http://it.dbpedia.org/resource/	398360	1
<http://pl.dbpedia.org/resource/	353769	1
<http://pt.dbpedia.org/resource/	337217	1
<http://ru.dbpedia.org/resource/	334571	1
<http://nl.dbpedia.org/resource/	309779	1

Tabelle 5.8: 10 häufigsten URI Präfixe – Variante (B)

Es ergeben sich für Variante (A) die in Tabelle 5.7 dargestellten Werte. Variante (B) ist in Tabelle 5.8 aufgelistet. Für Beide wurde der prozentuale Anteil bezogen auf das komplette Wörterbuch ermittelt.

An den Tabellen ist erkennbar, dass '<http://dbpedia.org/...' in beiden Fällen das häufigste Präfix von URIs ist. Direkt danach folgen URIs mit '<http://en.wikipedia.org/...'. Sogar die Anteile der zwei häufigsten Präfixe in beiden Varianten ist annähernd gleich. Variante (A) ist universeller, da hier die Präfixe kürzer sind und mehr Freiheit bei der Wahl des verbleibenden Restes der URI besteht. Dagegen zeigt (B) für den DBpedia-Half Datensatz sehr gute Ergebnisse. Variante (B) ist für eine Kompression in Form von Ersetzungen besser geeignet, da die gefundenen Präfixe dort länger sind und somit mehr Speicher eingespart werden kann.

Basierend auf diesen Beobachtungen und Messungen kann geschlussfolgert werden, dass eine Kompression der Zeichenketten mittels Ersetzungsregeln erfolgversprechend ist. Grundlegend können offensichtlich nicht alle Präfixe ersetzt werden, da es im schlechtesten Fall sehr viele Präfixe gibt. Weiterhin ist es auch nicht sinnvoll, kurze Präfixe mit längeren Ersatzzeichenketten zu ersetzen. Eine Betrachtung der 100 häufigsten Präfixe ist ausreichend. Es ist auch denkbar, die Präfixe nach dem möglichen Kompressionsgrad und der Häufigkeit auszuwählen. In den Tabellen 5.7 und 5.8 ist bereits gut erkennbar, dass nach den ersten 10 Werten nur

wenige Präfixe sehr häufig auftreten. Es ergibt sich ein prozentualer Anteil von weniger als 1% für alle weiteren Präfixe.

Ersetzungsregeln. Sei p_i ein häufiges Präfix an der i-ten Stelle, so wird eine Ersetzung mit

$$p_i \rightarrow '<' + code(i) + '>'$$

durchgeführt. Da insgesamt nur 100 häufige Präfixe betrachtet werden, kann die Ersetzung eines Präfixes mittels 3 Zeichen erfolgen. Dazu wird als Funktion $code(i)$ folgende Abbildung benutzt:

$$code(i) = chr(ord(' ') + i)$$

Die Funktion $ord(a)$ gibt den ASCII-Code wieder und $chr(x)$ wandelt einen ASCII-Code in ein Zeichen um.

Rückersetzung. Die Rückersetzung kann einfach durchgeführt werden. Sei s eine Zeichenkette, welche eventuell mit den Ersetzungsregeln verkürzt wurde. Dazu muss zunächst überprüft werden, ob das erste Zeichen $s[0]$ ein '$<$' ist.

Bei RDF-Daten, welche keine URIs sind, darf am Anfang kein '$<$'-Zeichen stehen, daher kann das erste Zeichen von s als Indikator benutzt werden. Weiterhin muss das zweite Zeichen ein Buchstabe $a \in \{ord(' '), ..., chr(ord(' ') + 100)\}$ sein. Das dritte Zeichen $s[2]$ muss ein '$>$'-Zeichen sein, für herkömmliche unverkürzte URIs gilt dies nicht. Basierend auf dem zweiten Zeichen kann in einem Array das jeweilige Präfix i ermittelt und somit die Rekonstruktion erfolgreich durchgeführt werden.

Benchmark

Aufbauend auf den Ersetzung- und Rückersetzungsregeln wurden Datenstrukturen (urianalyser und uricompressor) implementiert. Der urianalyser ist für die komplette Präfixanalyse verantwortlich. Im ersten Schritt werden daher alle Zeichenketten gelesen und an den Analyser weitergegeben. Anschließend wird der uricompressor erzeugt. Er erhält die Präfixanalyse, ermittelt die Top-100 Präfixe und speichert sie effizient ab. Für die Speicherung werden zwei Arrays benötigt. Das erste Array ist für die Präfixe und das zweite für die Längen der Präfixe. Der uricompressor bietet weiterhin Methoden zur Kompression und Dekompression der Zeichenketten. Im zweiten Durchlauf wurden die Zeichenketten komprimiert.

In Tabelle 5.9 sind für verschiedene Datensätze die ermittelten Ergebnisse dargestellt. Für den Datensatz, welcher nur aus URIs besteht, kann eine Kompressionsrate von ungefähr 62% erzielt werden. Das bedeutet, es können knapp 38% Speicher eingespart werden. Im Gegensatz dazu wurde auch das vollständige Wörterbuch untersucht. Eine ähnlich hohe Kompressionsrate von ungefähr 72% kann auch hier erzielt werden, da das Wörterbuch überwiegend aus URIs besteht. Für das gesamte Wörterbuch kann also mittels des Präfixersetzungsverfahrens ungefähr 28% Speicher eingespart werden. Da RDF-Daten hauptsächlich aus URIs bestehen, kann für größere Datensätze als den DBpedia-Half-Datensatz eine wesentlich höhere Kompressionsrate erzielt werden.

Datensatz	nur URIs	komplettes Wörterbuch	
verschiedene Zeichenketten	34662679	47637110	
Speicherung unkomprimiert	≈ 1.99	≈ 2.77	[GB]
Speicherung uricompressor	≈ 1.22	≈ 2.0	[GB]
Kompressionsrate	≈ 0.62	≈ 0.72	

Tabelle 5.9: Kompressionsraten des Wörterbuchs mittels Präfixersetzung (uricompressor), dabei wurde in der uricompressor Variante auch der Overhead durch Speicherung der Präfixtabelle einbezogen

Fazit

Auch wenn die Datenstruktur Trie nicht überzeugen konnte, kann dennoch mittels Präfixverkürzung ein erheblicher Gewinn an Speicher erzielt werden. Die implementierten Datenstrukturen (urianalyser und uricompressor) können transparent im CameLOD-Projekt integriert werden. Dazu ist es nötig im Builddb-Schritt vor dem Speichern der Zeichenketten auf Festplatte die Prefixanalyse mittels des urianalysers durchzuführen. Anschließend muss der uricompressor-Vorgang ausgeführt werden. Das Wörterbuch muss um die ermittelte Ersetzungstabelle ergänzt werden. Anschließend werden in den Zeichenketten die Ersetzungen mittels des uricompressors durchgeführt und die verkürzte Zeichenkette (sofern es sich um eine URI handelt) gespeichert. Alle nicht URI-Zeichenketten werden ungekürzt gespeichert.

Im CameLOD-Schritt werden Zeichenketten nur bei Ausgaben oder Anfragen direkt benutzt. Die Ausgaben können durch ein vorhergehendes Dekomprimieren mittels des uricompressors erfolgen. Werden in Anfragen URIs benutzt, so müssen sie für ein Suchen im Wörterbuch vorher durch den uricompressor komprimiert werden. Weitere Veränderungen sind nicht nötig.

Insgesamt kann das Verkürzen der Präfixe gute Kompressionsraten erzielen, auch wenn nur 100 Präfixe betrachtet werden. Durch mehr Präfixverkürzungen kann zwar unter Umständen eine höhere Kompressiosrate folgen, dafür ist jedoch die Rekonstruktion nicht so effizient umsetzbar.

5.6 Speicher

Auch die Art der Zugriffe auf den Hauptspeicher kann verbessert werden. Daher sollen im folgenden Abschnitt zwei Möglichkeiten dargestellt werden. Zunächst kann mittels Memory-Mapped-Files eine Auslagerung der Daten auf disk-Speicher erfolgen und durch das seitenweise Laden effiziente Zugriffe ermöglicht werden. Weiterhin ist mittels NUMA eine Einteilung des Hauptspeichers in lokalen und entfernten Speicher möglich, womit parallelisierte Verarbeitungen beschleunigt werden können.

5.6.1 Memory-Mapped-Files

Beim Aufbau der Datenbasis erfolgen viele Lese-Zugriffe auf Daten. Außerdem ist ein hoher Zeitbedarf beim Laden der serialisierten komprimierten Datenstrukturen vor der eigentlichen

Anfrageverarbeitung nötig. Beide Fälle sind nicht maßgebend für die Verarbeitungszeit verantwortlich, aber dennoch für das gesamte System wichtig.

Weiterhin kann durch Memory-Mapped-Files (MMF) das Serialisieren mittels Boost-Methoden umgangen werden und im Bezug auf eine spätere Erweiterung der In-Memory Verarbeitung zu einer disk-basierten Lösung ein einfacher Buffer-Manager realisiert werden.

Benchmark

Um einen Vergleich durchzuführen, wurden zwei Varianten betrachtet. In der Variante (std) wurde mittels herkömmlicher C++-Datei-Operationen der Zugriff auf eine Datei durchgeführt. Dagegen wurden bei (mmf) mittels MMF die Zugriffe durchgeführt. Zur Auswertung wurde die Summe aller Zeichen verschiedener Dateien des DBpedia-Projektes gebildet und die benötigte Zeit ermittelt. Dabei wurden 32 Messungen durchgeführt und die Mittelwerte und Intervalle mit 99% Konfidenz berechnet.

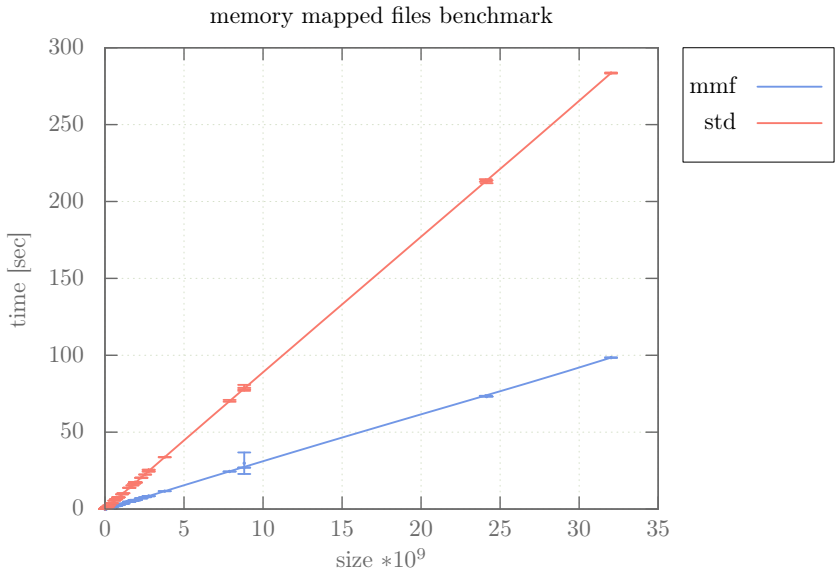

Abbildung 5.19: MMFs Benchmark, dargestellt sind Mittelwerte über 32 Messungen mit 99% Konfidenzintervall

In Abbildung 5.19 sind die ermittelten Zeiten für beide Varianten dargestellt. Deutlich erkennbar ist, dass die MMFs-Variante ungefähr $\frac{1}{3}$ so viel Zeit wie die (std)-Variante benötigt.

Fazit

Durch MMFs kann das Lesen von Daten für das CameLOD-Projekt im Builddb-Schritt beschleunigt werden. Der den Einsatz von MMFs kann zusätzlich ein Geschwindigkeitsgewinn beim Laden vor der Anfrageverarbeitung erzielt werden, da die Serialisierung von Boost ersetzt werden kann. Die benutzte Implementierung im Bechmark bietet eine Datenstruktur, welche äußerlich wie ein Vektor wirkt, jedoch im Hintergrund MMFs benutzt. Durch Erweiterungen kann ein transparenter Buffer-Manager für die Spaltenvektoren realisiert werden. Ähnlich wurde bei bei MonetDB vorgegangen.

5.6.2 NUMA

Das CameLOD-Projekt soll auch in Bezug auf NUMA untersucht werden. Da ein aussagekräftiger Benchmark große Teile des CameLOD-Projekts simulieren müsste, wurde an dieser Stelle die Vorgehensweise verändert. Das CameLOD-Projekt benutzt Intel-TBB zur Parallelisierung. Nach [Inti] kann der von TBB zur Verfügung gestellte Partitionierungs-Scheduler 'affinity_partitioner' verwendet werden um cache-coherent-NUMA besser auszunutzen. Zur Untersuchung wurde der Scheduler in der aktuellen Implementierung des CameLOD Projekts ersetzt. Wichtig für den 'affinity_partitioner' ist, dass vor der zu betrachtenden Ausführung bereits die Daten lokal durch eine parallelisierte Schleife vorliegen. Der 'affinity_partitioner' wird daher mehrfach genutzt und somit die Lokalität gewährt. Zur experimentellen Untersuchung auf den NUMA-Effekt im CameLOD-Projekt wurde der Partitioner im parallel_do Operator global angelegt, dadurch wird er von jeder parallelisierten Anfrage benutzt. Zum Testen muss demnach zunächst eine Anfrage zur Herstellung der Datenlokalität und anschließend die eigentliche Anfrage, welche zur Zeitmessung benutzt wird, durchgeführt werden.

Benchmark

Es wurden die Anfragen, welche den Parallelisierungsoperator ('parallel_do') beinhalten, als Grundlage benutzt. Demzufolge die beiden in Kapitel 4.1.2 eingeführten Anfragen Q2, Q3 und weiterhin die Anfragen Q9, Q10, Q11, Q12 und Q13 (dabei handelt es sich um einfache parallelisierte Scan-, Filter-, Join- und Sortier-Anfragen, siehe Anhang A).

Für die Messungen wurde zunächst die unveränderte Version (org) des CameLOD-Projekts benutzt und anschließend die mit dem veränderten Partitions-Scheduler (mod). Insgesamt wurden 32 Messungen durchgeführt. Für die Mittelwerte wurde die Anfrage zur Herstellung der Datenlokalität nicht betrachtet. Es wurden auch Intervalle mit 99% Konfidenz berechnet.

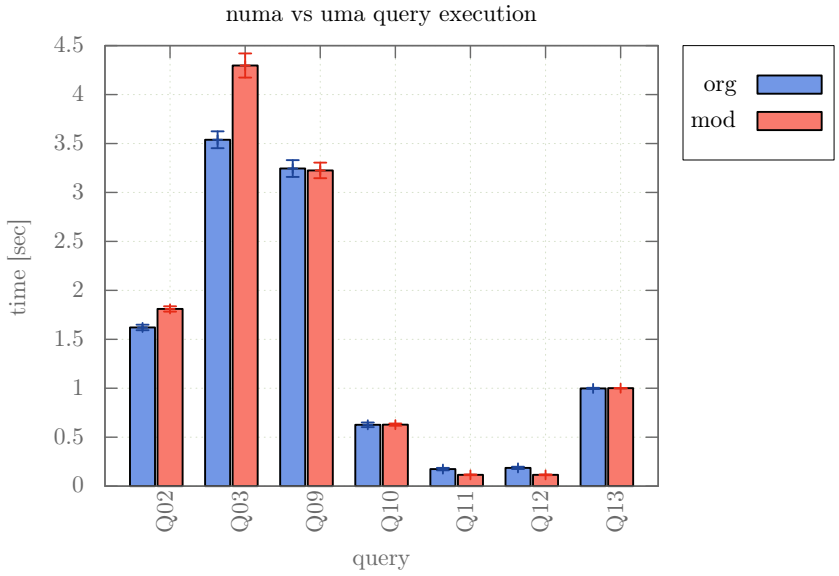

Abbildung 5.20: Zeit für parallelisierte Anfragen mit 'affinity_partitioner' (mod) und ohne (org), dargestellt sind Mittelwerte über 32 Messungen mit 99% Konfidenzintervall

In Grafik 5.20 sind die Ergebnisse für die 6 Anfragen dargestellt. Es ist kaum ein Unterschied zwischen den beiden Varianten zu erkennen. Teilweise sind die Anfragen mit der Veränderung schneller oder langsamer. Für Anfrage Q3 ist etwa eine Verschlechterung um circa $1\ s$ erkennbar. Auch in [Inti] wurden Messungen durchgeführt. Dabei konnte ein Geschwindigkeitszuwachs bei sehr hoher Thread-Anzahl erzielt werden.

Fazit

Durch den Ersatz des Schedulers, welcher nach [Inti] cache-coherentes-NUMA bieten soll, konnte kein Gewinn erzielt werden. Das Referenzsystem besitzt aber nur zwei NUMA-Knoten. Unter Umständen kann bei einer höheren Knotenanzahl eine signifikantere Verbesserung erzielt werden.

5.7 Chunk-Größe

Hauptsächlich hängt die Verarbeitung von der grundlegenden Struktur des CameLOD-Projektes ab. Dabei werden die Daten in Chunks eingeteilt. Die maximale Chunk Größe ist von entscheidender Rolle für die Verarbeitung, weil beim Überschreiten dieser Größe eine Teilung des Chunks stattfindet. Ist die reale Chunk-Größe zu klein, so wird der CPU-Cache (L1 und L2) nicht vollständig genutzt. Hingegen kann eine zu hohe reale Chunk-Größe bewirken, dass für die Verarbeitung eines Chunks mehrmals Daten in den Cache geladen werden müssen. Es liegt also nahe an dieser Stelle zu versuchen die Chunks bezogen auf die Cache Größe anzugleichen.

Theoretisch bietet der verwendete Intel Xeon CPU E5-2630 für jeden physischen Kern einen L1 Cache der Größe 32KB für Daten und einen 256KB größen L2 Cache. Ein Tripel eines Chunks benötigt in der unveränderten Variante ohne Kompression 3 uLong Werte, welche jeweils 8 Byte Speicher benötigen. Hierbei wird der geringe Overhead durch Speichern von Pointern und anderen Werten für die Organisation der Chunks untereinander vernachlässigt. Somit ergibt sich ein Chunk-Größen Bereich von:

$$\left[\frac{L1}{3 \cdot 8}, \frac{L2}{3 \cdot 8}\right] \approx [1365, 10922]$$

Eine optimale Größe der Chunks muss also in diesem Intervall liegen. Der Bereich kann natürlich nur als Näherung angesehen werden, da in der Anfrageverarbeitung auch weitere Variablen im Cache gehalten werden müssen. Aus diesem Grund soll in einem experimentellen Benchmark der Einfluss der Chunk-Größe auf die Verarbeitungszeit der Anfragen betrachtet werden.

5.7.1 Benchmark

Insgesamt wurden 14 verschiedene Anfragen untersucht und Messungen mit verschiedenen Chunk-Größen durchgeführt. Die maximale Chunk-Größe variierten dabei im Bereich von 1000 bis 57000 mit einer Schrittweite von 1000 Elementen. Das Intervall wurde gewählt, um auch das Verlassen des theoretisch optimalen Bereichs darstellen zu können. Die reale Chunk-Größe wurde auch ermittelt und lag im Intervall von 524 bis 29984. Der angegebene Bereich stellt

Mittelwerte über alle Indizes dar. Dafür wurde im System bei fester maximaler Chunk-Größe ermittelt, wie viele Chunks jeder Index besitzt und anschließend mit Hilfe der Tripelanzahl ($TupleCount$) der Mittelwert gebildet:

$$avgChunkSize = \frac{3 \cdot TupleCount}{\sum_{index} |Chunks(index)|}$$

Da die Größe im Quelltext als Konstante festgelegt wurde und maßgebend beim Aufbau der Datenbank erforderlich ist, weil die Chunks serialisiert auf Festplatte gespeichert werden, bestand die Messung aus 3 Schritten. Zunächst wurde die zu untersuchende Chunk-Größe als Konstante festgelegt und anschließend neu kompiliert. Weiterhin wurden die Daten geladen und auf Festplatte gespeichert. Im letzten Schritt wurden alle Anfragen durchgeführt und die Ausführungszeiten exportiert. Um Messschwankungen auszugleichen fanden 32 Wiederholungen einer Anfrage statt und dabei wurden Mittelwerte und Intervalle mit 99% Konfidenz gebildet.

5.7.2 Auswertung

Grundlegend wurden alle 14 Anfragen untersucht, aber es sollen im folgenden Abschnitt nur die ersten 4 Anfragen, welche in Abschnitt 4.1.2 eingeführt wurden, betrachtet werden. Alle Anfragen weisen ähnliche Ergebnisse auf. Der zweite Datenpunkt der Diagramme ist dabei immer die aktuell verwendete maximale Chunk-Größe von 2000 Elementen. In den Diagrammen wird immer $avgChunkSize$ dargestellt. Die gewählte maximale Chunk-Größe ist durch das Splitten der Chunks in der Implementierung immer ungefähr doppelt so groß.

Q1

In Grafik 5.21 ist dargestellt wie sich die Ausführungszeit der Anfrage Q1 in Abhängigkeit der Chunkgröße verändert. Es ist erkennbar, dass zwischen [5000,15000] die Verarbeitung am schnellsten erfolgt. Bei höherer Chunk-Größe schwanken die Werte sehr stark, aber die Verarbeitungszeit fällt im Mittel höher aus. Im Vergleich zur aktuellen Chunkgröße ergibt sich bei $avgChunkSize = 7500$ ein Geschwindigkeitszuwachs von ungefähr $0.3\ s$, was ungefähr 3% entspricht.

Q2

In Diagram 5.22 dagegen wird die parallelisierte Variante der Anfrage Q1 dargestellt. Auch hier stellt sich ein ähnliches Minimum im Bereich von [5000,30000] ein. Im Gegensatz zu Q1 ergibt die veränderte Chunkgröße einen größeren Geschwindigkeitszuwachs. Während bei der aktuellen Chunkgröße $\approx 2.3\ s$ zur Verarbeitung benötigt werden, liegt die Verarbeitungszeit im genannten Bereich bei $\approx 1.2\ s$. Das entspricht einem Zuwachs um etwa 50%. Der erzielte Gewinn kann dadurch erklärt werden, dass die verschiedenen CPU-Kerne eigenen Cache (L1 und L2) zur Verfügung haben und somit die Chunks sehr cache-effizient parallel verarbeitet werden können.

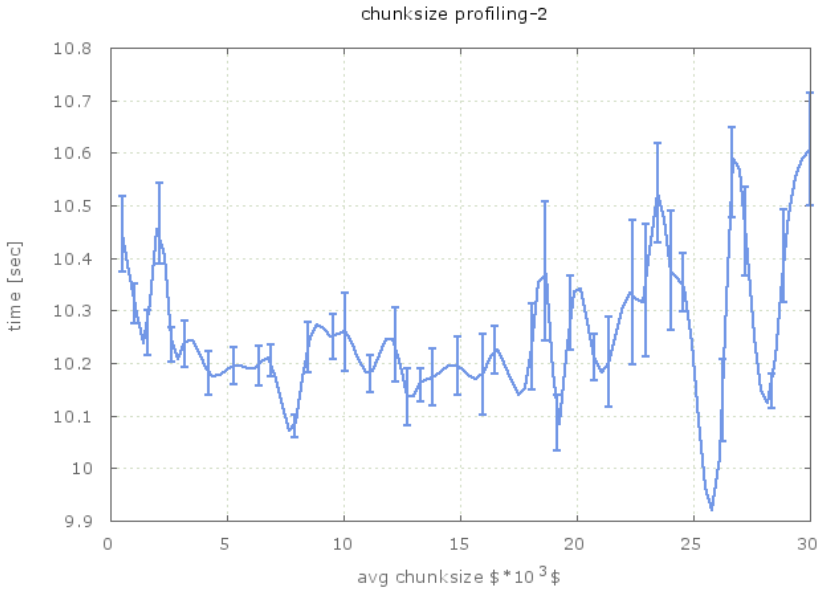

Abbildung 5.21: Einfluss der realen Chunkgröße für Anfrage Q1, dargestellt sind Mittelwerte der Zeit über 32 Messungen mit 99%-Konfidenzintervall

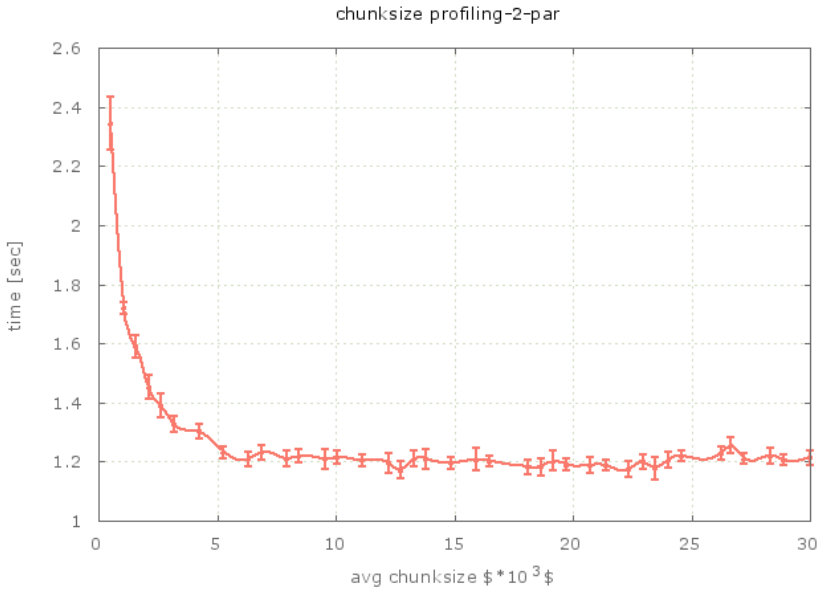

Abbildung 5.22: Einfluss der realen Chunkgröße für Anfrage Q2, dargestellt sind Mittelwerte der Zeit über 32 Messungen mit 99%-Konfidenzintervall

Q3

Die Ergebnisse für Anfrage Q3 sind in Abbildung 5.23 dargestellt. Im Vergleich zu Q1 und Q2 kann hier ein negativer Einfluss auf die Verarbeitungszeit bei vergrößerter Chunk-Size geschlussfolgert werden. Grundlegend liegt dies an der Implementierung des Index-Join-Operators, da ein Join bezogen auf ein Chunk tupelweise mit allen anderen Chunks durchgeführt wird. Sind

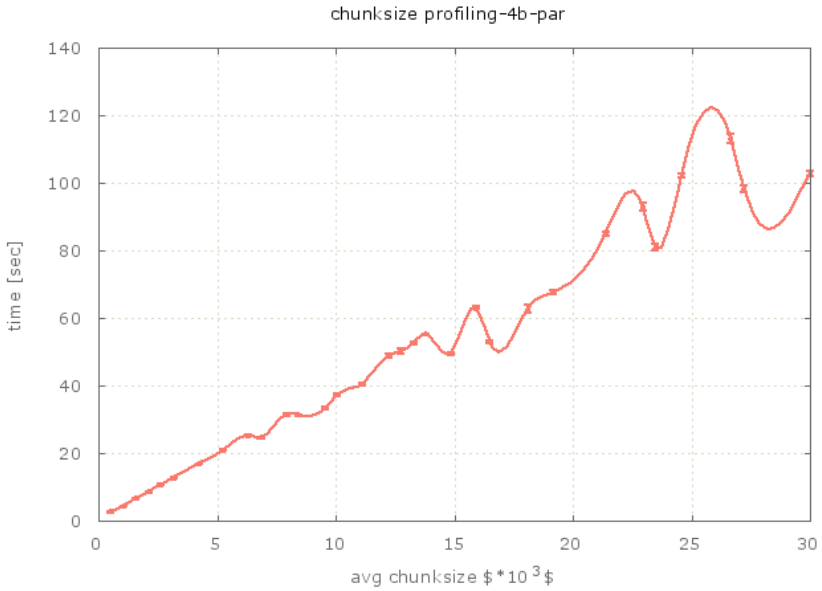

Abbildung 5.23: Einfluss der realen Chunkgröße für Anfrage Q3, dargestellt sind Mittelwerte der Zeit über 32 Messungen mit 99%-Konfidenzintervall

die Chunks sehr groß ergeben sich daher mehr Tupel-Join Versuche. Für kleine Chunk-Größen im Bereich [0,2000] ist die Anfrageverarbeitung schnell.

Q4

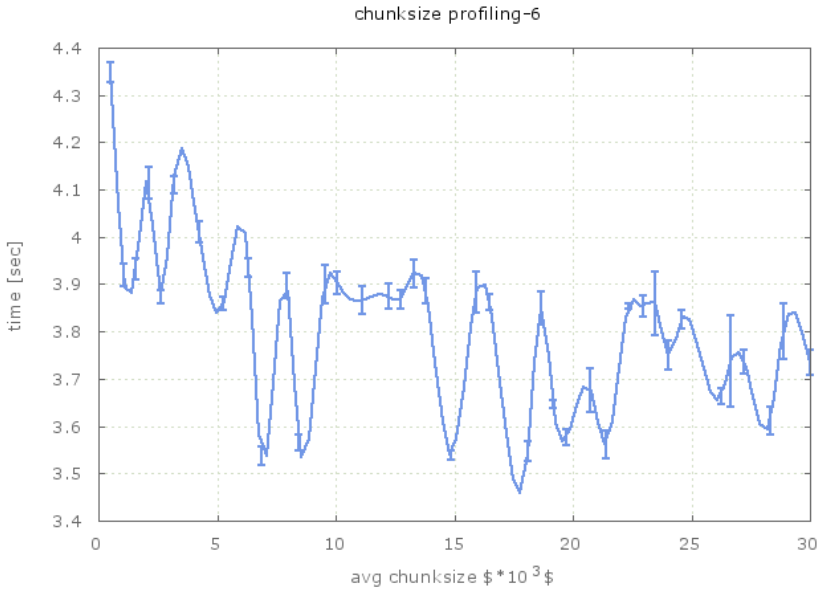

Abbildung 5.24: Einfluss der realen Chunkgröße für Anfrage Q4, dargestellt sind Mittelwerte der Zeit über 32 Messungen mit 99%-Konfidenzintervall

In Abbildung 5.24 ist die letzte Testanfrage Q4 dargestellt. Ähnlich wie bei den Anfragen Q1 und Q2 liegt hier das Minimum der Verarbeitungszeit im Bereich [5000,20000]. Beispielsweise ergibt sich bei ungefähr $avgChunkSize = 7000$ eine Verarbeitungsgeschwindigkeit von circa $3.5\ s$ im Mittel, dagegen wird bei der originalen Chunk-Größe in etwa $3.9\ s$ benötigt. Es ergibt sich also ein Gewinn von $0.4\ s$, was annähernd 10% entspricht.

5.7.3 Fazit

In einem experimentellen Benchmark wurde die Korrelation zwischen Chunk-Größen und Verarbeitungszeit näher betrachtet. Insgesamt konnte festgestellt werden, dass alle Anfragen, welche keine Join-Operationen benutzten, von realen Chunk-Größen im Bereich [7000,15000] profitieren. Der ermittelte Bereich entspricht in etwa des theoretisch ermittelten L1 und L2 Cache-Bereichs für die Tripel. Für die maximale Chunk-Größe ergibt sich somit eine Wahl zwischen ungefähr [14000,30000]. Besonders parallelisierte Anfragen zeigen einen erheblichen Geschwindigkeitszuwachs bei höherer Chunk-Größe. Zum Beispiel ergibt sich ein Zuwachs von ungefähr 50% bei Q2. Nur Join-Operationen profitieren nicht von höheren Chunk-Größen. Eine kleine Chunk-Größe ist hier erforderlich.

Die offenbar architekturabhängige Chunk-Größe erfordert ein automatisiertes Verfahren zur Approximation eines Optimums. Ein solcher Chunk-Größen Optimierer kann an verschiedenen Stellen im System integriert werden. Ein Ansatz wäre die Chunk-Größe beim Aufbau der Datenbasis experimentell zu ermitteln. Dazu kann ein Beispiel-Chunk durchlaufen werden und die CPU-Zeit ermittelt werden. Das Vorgehen könnte ähnlich einem Gradientenabstiegsverfahren durchgeführt werden, wodurch schnell eine approximierte Lösung gefunden werden kann. Auch eine sehr grobe Abschätzung der Chunk-Größe kann mit der L1 und L2 Cache-Größe stattfinden. Die aktuell gewählte Chunk-Größe von 2000 stellt einen guten Kompromiss für Scan und Join Anfragen dar.

5.8 Zusammenfassung und Konsequenzen für CameLOD

Anhand der im vorhergehenden Kapitel identifizierten Performance-Schwachstellen wurden verschiedene eigene Implementierungen untersucht. Zunächst wurden Smart-Pointer betrachtet. Insgesamt kann nur durch den Einsatz von Raw-Pointern oder intrusive-Pointern ein Gewinn erzielt werden. Da nicht alle Stellen des CameLOD-Projekts zeitkritisch sind, ist es unter Umständen nicht nötig alle vorkommenden Smart-Pointer zu ersetzen. Weiterhin gibt es auch Stellen, welche aktuell Smart-Pointer benutzen, obwohl die referenzierten Objekte bereits von anderen Smart-Pointern verwaltet werden. Beispielsweise kann die Verkettung der Chunks untereinander ohne Smart-Pointer realisiert werden, da die Chunks in einem Index organisiert werden und während der gesamten Anwendungszeit zur Verfügung stehen müssen. Dagegen gibt es aber spezielle Chunks für Join-Ergebnisse, welche dynamisch erzeugt und gelöscht werden müssen. Eine manuelle Speicherverwaltung und somit der Ersatz von Smart-Pointer durch Raw-Pointer kann ungefähr 30% mehr Geschwindigkeit erreichen.

Weiterhin wurde im Builddb-Schritt festgestellt, dass viele Hash-Kollisionen auftreten und dardurch Zeichenketten-Vergleiche durchgeführt werden müssen. Durch den Einsatz einer auf AES-NI basierten Hash-Funktion können die Kollisionen stark verringert werden.

Es wurde in der Analyse auch festgestellt, dass BitSet-Operationen bei einigen Anfragetypen sehr stark in die Laufzeit einfließen. Verschiedene eigene Implementierungen und andere Ansätze wurden in einem Benchmark gegeneinander verglichen und es kann im Einsatz von CameLOD ungefähr 20% an Zeit eingespart werden.

Besonders wichtig für In-Memory Systeme ist das Sparen von Hauptspeicher. Aus diesem Grund wurden verschiedene Kompressionstechniken, welche einfache Dekompressionsmethoden bereitstellen, untersucht. Es konnte festgestellt werden, dass die Sequenzen für die Tripel-Darstellung in CameLOD durch die dargestellten Verfahren FOR und RLE profitieren können. Mittels Benchmarks konnte gezeigt werden, dass die Methoden auch effizient umsetzbar sind. Für CameLOD kann also geschlussfolgert werden, dass einige Sequenzen sehr gute Kompressionseigenschaften aufweisen. Zum Beispiel können die sortierten Spalten bis zu 50% komprimiert werden. Durch Einsatz von verschiedenen Typen für die Sequenzen kann erreicht werden, dass das jeweils optimale Verfahren ausgewählt wird. Eine Kombination von beiden Verfahren ist auch denkbar, wurde aber nicht näher betrachtet. Weiterhin kann auch Speicher im Wörterbuch von CameLOD gespart werden. Hierzu kann beispielsweise eine Ersetzung von Präfixen stattfinden und knapp 38% Speicher gespart werden.

Hauptspeicherzugriffe können, wenn sie nicht cache-effizient erfolgen, einen Flaschenhals darstellen. Daher fand eine Betrachtung der Zugriffe mittels NUMA-Optimierung oder Memory-Mapped-Files statt. Mit den durchgeführten Optimierungen bezüglich NUMA konnte fast kein Gewinn erzielt werden. Dagegen stellen MMF ein Ansatz dar, die Persistenz ohne Boost-Serialisierung zu ermöglichen. Schließlich wurde die festgelegte Chunk-Größe des CameLOD-Projektes experimentell untersucht. Die aktuell gewählte Größe stellt einen guten Kompromiss zwischen Join und Scan Anfragen dar und entspricht in etwa der theoretisch optimalen L1-Cache-Größe.

Kapitel 6

Auswertung

In diesem Abschnitt soll eine proof-of-concept Auswertung von ausgewählten Verbesserungsideen des letzten Kapitels im CameLOD-Projekt durchgeführt werden. Dabei wird die unveränderte Version und die Variante mit den Verbesserungen in verschiedenen Szenarien verglichen. Es wurden allgemeine Verbesserungen (binäre Suche und parallelsiertes Laden/Speichern der Indizes), welche im Analysekapitel zusätzlich geschlussfolgert wurden und an einigen Stellen Raw-Pointer integriert. Zunächst soll der Builddb-Schritt ausgewertet werden. Anschließend folgen verschiedene Anfragen im CameLOD-Teil. Als Datengrundlage dient der DBpedia-Half Datensatz.

6.1 Benchmark

Zur Auswertung der genannten allgemeinen Verbesserungen eignet sich ein getrennter Mini-Benchmark nicht. Aus diesem Grund wurden die Änderungen direkt in einem eigenen Entwicklungszweig umgesetzt. Der Quellcode des CameLOD-Projekts wurde auf den neuen C++11 Standard umgestellt. Dieser bietet zum Beispiel die Möglichkeit lambda-Funktionen zu benutzen, welche bei TBB zur Parallelisierung genutzt werden können.

org. Die Version (org) stellt die im Analysekapitel 4 betrachtete Version mit wenigen Erweiterungen bezüglich des C++11 Standards als Referenz dar. Darunter zählen beispielsweise das Entfernen von Warnungen und die Auflösung von Namenskonflikte zwischen Boost und Std bezüglich Shared-Pointern. Zur spätereren aussagekräftigen Simulation und Evaluierung von Raw-Pointern ist in der org Version bei der Verkettung der Chunks, dem SPO-Index und den Anfrageoperatoren eine Ersetzung der Smart-Pointer durchgeführt wurden. Sie wurden durch referenzzählende intrusive-Pointer ersetzt und bieten somit ähnliche Eigenschaften wie die Shared-Pointer.

mod. Es wurde eine Version (mod) mit den Veränderungen bezüglich binärer Suche, bei den Chunk-Datenstrukturen zum Finden der Einfügepositionen, betrachtet. Außerdem wurde das Laden beziehungsweise Speichern der Indexstrukturen parallelisiert. Jedoch ist dieser Ansatz von der Festplattengeschwindigkeit abhängig und wird auch nicht alle CPU-Kerne gleichmäßig auslasten. Dennoch kann das CameLOD-Projekt davon profitieren. Die in der org Version eingeführten intrusive-Pointer für die Chunks sind dahingehend verändert worden, dass sie keine

Referenzzählung und Speicherdeallokation vornehmen. Dieser Ansatz dient zur Simulation von Raw-Pointern in relevanten Teilen der Anfrageverarbeitung.

6.2 Aufbau und Laden

Zunächst soll der Builddb-Schritt und das Laden der Daten bei der Anfrageverarbeitung betrachtet werden. Da das Aufbauen der Datenbasis circa 1.5 h benötigt, wurde an dieser Stelle auf Mittelwerte verzichtet, auch weil dieser Schritt als nicht zeitkritisch eingestuft werden kann.

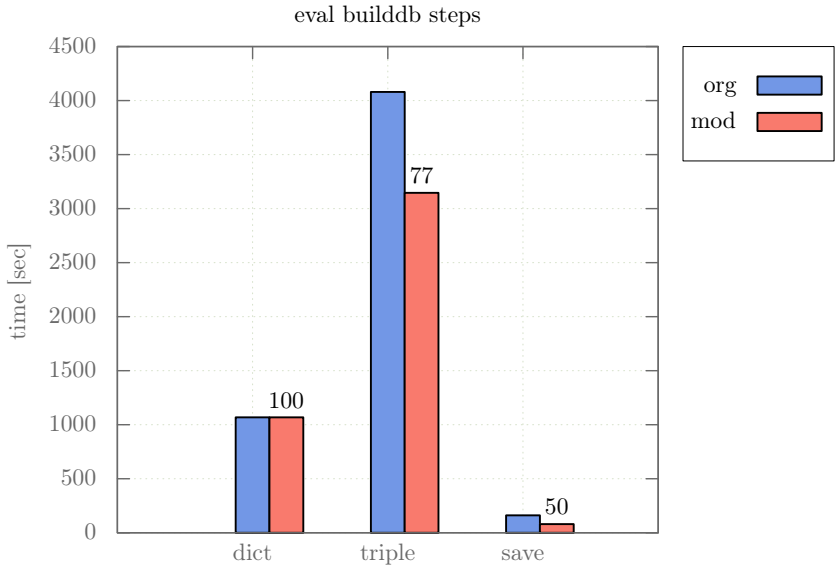

Abbildung 6.1: Evaluierung von Builddb, dargestellt sind Zeiten für die einzelnen Schritte des Prozesses und der prozentuale Anteil der (mod)-Variante bezüglich der (org)-Version

Aufbau. In Diagramm 6.1 sind die Zeiten für die Schritte des Builddb-Prozesses dargestellt. Er teilt sich in den Aufbau des Wörterbuchs (dict), das Erzeugen der Tripelindizes (triple) und dem abschließenden Speichern der Daten (save) auf Festplatte.

Es ist deutlich erkennbar, dass die Änderungen in der (mod)-Version in zwei der drei Teilschritten einen Geschwindigkeitsgewinn verursachen, besonders beim Erzeugen der Tripel kann viel Zeit eingespart werden. Ungefähr 27% an Zeit können hier durch den Einsatz der binären Suche gespart werden. Beim Aufbau des Wörterbuchs (dict) kann keine Zeit eingespart werden, besonders weil in diesem Teilprozess keine Ersetzungen stattgefunden haben. Wogegen die Speicherung (save), durch die Parallelisierung ungefähr 50% weniger Zeit benötigt. Insgesamt benötigt die (mod) Variante nur noch ungefähr 80% der CPU-Zeit im Vergleich zur unmodifizierten. Somit benötigt der Aufbau der Datenbasis in der (mod)-Version nur noch ungefähr 1 h Zeit.

Laden. Es wurde auch das Laden der Daten im CameLOD-Schritt, demzufolge vor der Anfrageverarbeitung, analysiert. In Abbildung 6.3 sind die Zeiten beider Varianten dargestellt.

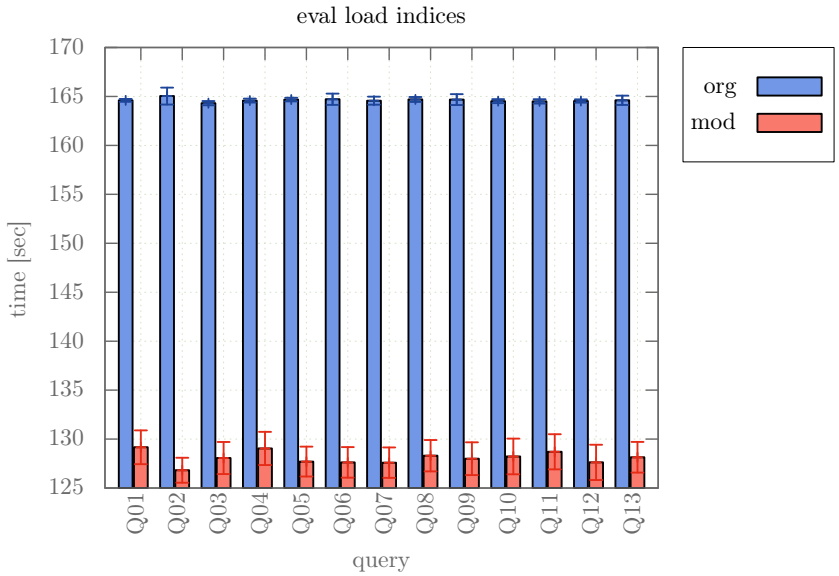

Abbildung 6.2: Zeit für das Laden der Indizes im CameLOD-Schritt, dargestellt sind Mittelwerte über 32 Messungen mit 99%-Konfidenz-Intervallen für 13 Anfragen

Dabei wurden für 13 verschiedene Anfragen jeweils über 32 Messungen die Mittelwerte und Intervalle mit 99% Konfidenz berechnet.

Für die org Variante ergibt sich bei allen Anfragen ungefähr eine Ladezeit von $165s$. Dagegen benötigt die mod Version nur ungefähr $130s$. Insgesamt kann durch das parallele Laden und die Simulation von Raw-Pointern eine Verbesserung um etwa 22% erzielt werden.

6.3 Anfrageverarbeitung

Im letzten Schritt der Auswertung wurde die Anfrageverarbeitung betrachtet. Obwohl nur an wenigen Stellen des Systems Veränderungen bezüglich Smart-Pointer vorgenommen wurden, ist in den Diagrammen 6.3 und 6.4 erkennbar, dass die Verarbeitung in fast allen Anfragen beschleunigt werden konnte. Beispielsweise kann die Zeit für die Anfrage Q6 mit der (mod) Variante auf ungefähr 85% reduziert werden. Das bedeutet eine Verbesserung um etwa 15%. Q6 benutzt häufig die Chunk-Verkettung und kann durch die Veränderungen profitieren. Anfrage Q13, welche parallelisiert ausgeführt wird, benötigt in der (org) Version ungefähr $1s$. Durch die Modifikationen konnte die Zeit auf etwa $0.9s$ reduziert werden, somit ist die Anfrage um circa 10% schneller. Nur die Anfragen Q3, Q4 und Q5 sind sichtbar nicht schneller geworden, jedoch ist auch keine Verschlechterung erkennbar. Bei einigen Anfragen fällt der Gewinn geringer aus, weil nur wenige Shared-Pointer in der (mod) Variante ersetzt wurden. Ein höherer Gewinn kann durch weiteres Ersetzen erzielt werden.

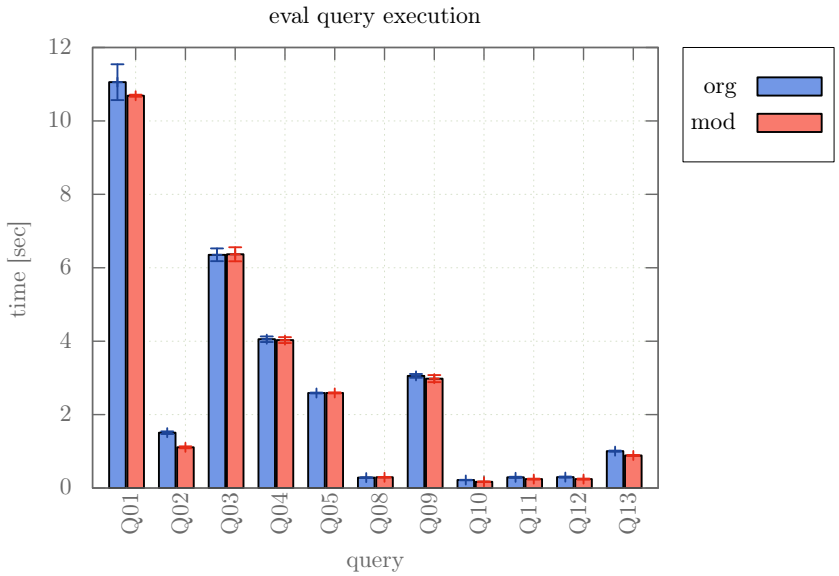

Abbildung 6.3: Zeit für die Anfrageverarbeitung im CameLOD-Schritt, dargestellt sind Mittelwerte über 32 Messungen mit 99%-Konfidenz-Intervallen für 11 Anfragen

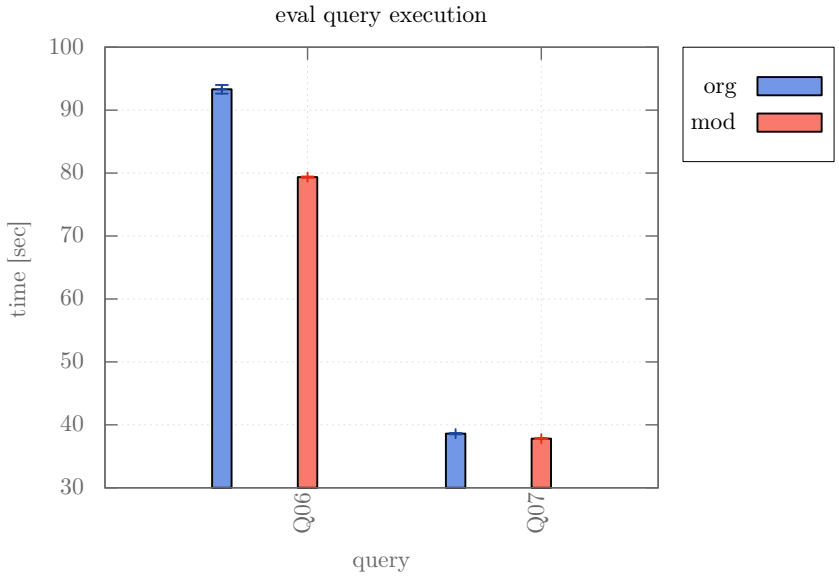

Abbildung 6.4: Zeit für die Anfrageverarbeitung im CameLOD-Schritt, dargestellt sind Mittelwerte über 32 Messungen mit 99%-Konfidenz-Intervallen für Anfragen Q6 und Q7

6.4 Zusammenfassung

Es wurde in einem Benchmark gezeigt, dass durch den Einsatz einiger der ermittelten Verbesserungen in fast allen Teilbereichen des CameLOD-Projekts Verarbeitungszeit gespart werden kann. Besonders das Aufbauen der Datenbasis konnte um etwa 20% beschleunigt werden. Auch das Laden der Daten vor der Anfrageverarbeitung konnte um circa 22% verbessert werden. Einige der Anfragen konnten wenig durch die Veränderungen beschleunigt werden, andere hingegen um ungefähr 10%. Folglich müssen an mehr Stellen im System Ersetzungen der Pointer stattfinden, um weitere Verbesserungen erzielen zu können.

Kapitel 7

Fazit

Im letzten Kapitel dieser Arbeit sollen die dargestellten Ideen und Verbesserungen zusammengefasst werden. Außerdem soll auch ein Ausblick bezüglich der effizienten Verarbeitung von Linked-Open-Data gegeben werden.

7.1 Zusammenfassung

Im Zuge des semantischen Webs werden immer mehr Linked-Open-Data öffentlich zur Verfügung gestellt. Dabei wird der Ansatz verfolgt, semantische Informationen in einem Format zur Verfügung zu stellen, so dass sie beispielsweise von Computern verarbeitet werden können. Die Datenmengen, die bei der LOD Verarbeitung anfallen, sind sehr groß und stellen somit neue Anforderungen an Datenbanksysteme. Reine relationale Datenbanken können zwar auch verwendet werden um LOD zu verarbeiten, sie nutzen aber ihre speziellen Eigenschaften nicht aus. Darüber hinaus verwenden viele relationale Datenbanksysteme nur wenige Funktionen (Parallelisierung, Vektorisierung, ...) moderner CPU-Architekturen. Auch sind in modernen Rechner-Architekturen große Mengen an Hauptspeicher möglich. Spezielle In-Memory Datenbanksysteme für LOD, wie zum Beispiel CameLOD, können von solchen Features profitieren. Ziel dieser Arbeit ist es Erweiterungen des CameLOD-Projekts zu entwickeln um eine Erhöhung der Effizienz in Bezug auf Speicher und Verarbeitungszeit zu erzielen.

Im semantischen Web werden LOD durch RDF-Graphen dargestellt. Die Anfrageverarbeitung erfolgt dabei durch den festgelegten Standard mittels SPARQL. Ein RDF-Graph besteht aus Subjekt-, Prädikat- und Objektwerten, wobei ein Subjekt-Prädikat-Objekt Tripel genau eine Aussage darstellt. LOD können als globale Tabelle bestehend aus den Tripel aufgefasst werden, es gibt aber auch andere Speichermöglichkeiten. Für die Verarbeitung haben sich bereits einige Ansätze etabliert. Beispielsweise wurden die In-Memory Stores (RDFHDT, Hexastore, BitMat und RDF-3X) betrachtet. Wobei festgestellt werden konnte, dass sie zwar interessante Ansätze bieten, aber moderne Rechnerarchitekturen nicht vollständig beziehungsweise nur wenig ausnutzen. Grundlegend verwenden alle betrachteten Ansätze Kompressionsverfahren und Indizierung um die Datenmengen effizienter verarbeiten zu können. Moderne CPU-Architekturen bieten eine Reihe von Erweiterungen. Bezogen auf Intel-CPUs wurden verschiedene Erweiterungen näher betrachtet, beispielsweise Parallelisierung, Vektorisierung (SIMD), Superskalarität und Pipelining sowie Caches und Cache-Hierarchien. Es wurden ebenso Erneuerungen bezüglich Hauptspeicher, dazu zählen Non-Uniform Memory Access (NUMA) und persistenter Speicher, betrachtet. In-Memory Datenbanksysteme unterscheiden sich nur in der Speicherung beziehungsweise Verarbeitung der Daten, denn im Gegensatz zu herkömmlichen disk-basierten

Datenbanksystemen werden alle Daten im Hauptspeicher verarbeitet und gehalten. Dennoch müssen In-Memory Systeme auch Möglichkeiten bieten Daten persistent zu speichern. Neben der In-Memory LOD Verarbeitung gibt es bereits andere relationale In-Memory Datenbanksysteme, wie etwa SAP HANA und MonetDB.

Gerade bei der In-Memory Verarbeitung ist Hauptspeicher eine begrenzte Ressource. Kompressionstechniken können genutzt werden um Speicher zu sparen. Dabei müssen die Techniken aber auch effiziente Zugriffe auf die komprimierten Daten liefern. Es wurden verschiedene Grundverfahren betrachtet, beispielsweise Wörterbuchkompression, Entropiekodierung und Verfahren für die Kompression von Integer-Sequenzen (Frame of Reference und Run-Length Encoding). Gerade die Verfahren für Integer-Sequenzen bieten gute Möglichkeiten zur Kompression von Indexstrukturen in Datenbanksystemen.

Hauptuntersuchungsgegenstand dieser Arbeit war das CameLOD-Projekt. Zuerst wurde der grundlegende Ablauf beim CameLOD-Projekt beschrieben. Dabei werden im ersten Schritt die Daten komprimiert gespeichert und im zweiten Schritt werden die Anfragen an das System gestellt. Es wurde auch die Grundstruktur des Systems beschrieben. Es findet eine Indizierung aller RDF-Tripel in den drei möglichen Sortierreihenfolgen statt. Die Anfrageverarbeitung benutzt zur effizienten Ermittlung der Ergebnisse die Indexstrukturen. Die verschiedenen zur Verfügung gestellten Operatoren wurden kurz beschrieben. Als Datengrundlage für alle Messungen dient das DBpedia-Projekt. Es bietet verschiedene RDF-Daten zum Export an. Darauf aufbauend wurde ein kombinierter Datensatz mit ungefähr 30 GB erstellt. Das CameLOD-Projekt wurde mittels VTune analysiert und es wurden kurz die verschiedenen Analyseprofile des VTune-Tools beschrieben. Alle Messungen erfolgten dabei auf einem Referenzsystem, dessen Besonderheiten (CPU, der verwendete Befehlssatz, Cache-Größen und die NUMA-Topologie) beschrieben wurden. Auch die Effizienzbegriffe, welche für die Arbeit wichtig sind, wurden festgelegt.

Für die zwei Teilprozesse, das Aufbauen und spätere Anfragen, fand mittels VTune eine Analyse statt. Verschiedene Analyseprofile kamen dabei zum Einsatz. Grundlegend ist für einen großen Teil der Laufzeit der Einsatz von Smart-Pointern und BitSet Datenstrukturen verantwortlich. Auch einige Implementierungsverbesserungen konnten identifiziert werden. Durch das Profiling konnte festgestellt werden, dass die parallelisierten Operatoren gut umgesetzt sind. Neben der Analyse wurde auch die Kompression betrachtet und bezüglich der Indexstrukturen und des Wörterbuchs kann Speicher eingespart werden.

Aufbauend auf den Analyseergebnissen konnten verschiedene Ideen und Ansätze zur Verbesserung der Effizienz des Systems geschlussfolgert werden. Es wurden dabei Ansätze für Smart-Pointer, Hash-Funktionen, Bit-Vektoren, Kompression, Speicherzugriffe und Chunk-Größe betrachtet. In mehreren Mini-Benchmarks wurde überprüft, ob eine eigene Implementierung oder bestimmte Kompressionsschemata vielversprechend sind. An einigen Stellen kann ein hoher Geschwindigkeits- beziehungsweise Speichergewinn erzielt werden. Beispielsweise kann durch den vollständigen Ersatz von Smart-Pointern durch Raw-Pointer ungefähr 30% mehr Geschwindigkeit erreicht werden. Außerdem kann durch Kompressionsschemata für Integer-Sequenzen (veränderte FOR/RLE-Kompression) ungefähr 50% Speicher eingespart werden. Auch die Speicherung des Wörterbuchs für die Zeichenketten der RDF-Tripel kann mittels Präfixersetzung effizienter realisiert werden. Eine etwa 38% große Speicherersparnis kann erzielt werden. Hauptspeicherzugriffe können, wenn sie nicht cache-effizient erfolgen, einen Performance-Flaschenhals darstellen. Daher wurden auch mittels NUMA-Optimierung oder Memory-Mapped-Files die Zugriffe auf den Speicher betrachtet. Durch die NUMA-Optimierung

konnte fast kein Gewinn erzielt werden. Dagegen stellen Memory-Mapped-Files einen Ansatz dar, die Persistenz ohne Serialisierung zu realisieren. Am Ende wurde die festgelegte Chunk-Größe des CameLOD-Projektes experimentell untersucht. Die aktuell gewählte Größe stellt einen guten Kompromiss zwischen Join- und Scan-Anfragen dar und entspricht in etwa der theoretisch optimalen L1-Cache-Größe.

Es konnten nicht alle Verbesserungsansätze direkt in das CameLOD-Projekt umgesetzt werden. Demzufolge wurde eine Evaluierung ausgewählter Ansätze vorgenommen. Dazu zählt der Ersatz von Smart-Pointern durch Raw-Pointern an den für die Anfrageverarbeitung kritischen Stellen. Weiterhin wurden einigen allgemeine Verbesserungen umgesetzt. In fast allen Teilprozessen des CameLOD-Systems konnten Verbesserungen erzielt werden. Dabei kann festgestellt werden, dass das Aufbauen der Datenbank mit den Verbesserungen nur noch 80% der Zeit benötigt. Auch das Laden der Daten vor jeder Anfrageverarbeitung konnte um etwa 22% beschleunigt werden. Die Anfrageverarbeitung von 13 Testanfragen konnte effizienter gestalltet werden. Auch wenn nicht alle Erweiterungen in das CameLOD-Projekt integriert werden konnten, wurde gezeigt, dass eine Verbesserung prinzipiell möglich ist und sollte Gegenstand weitere Untersuchungen sein.

7.2 Ausblick

Das CameLOD-Datenbanksystem stellt einen neuen modernen Ansatz zur Verarbeitung von LOD dar. Nicht alle Erweiterungen bezüglich Smart-Pointer, BitSet-Strukturen oder Hash-Funktionen konnten in der Evaluierung in das System integriert werden und muss daher nächer untersucht werden. Besonders im Bezug auf Smart-Pointer ist der Einsatz eines Memory-Kontextes zur Verwaltung allokierter Speicherbereiche nötig.

Kompression ist ein wichtiger Bestandteil von In-Memory-Systemen. Das CameLOD-Projekt kann von Kompressionsschemata für Integer-Sequenzen oder Wörterbüchern profitieren. Aber zur Integration der Kompressionstechniken müssen Erweiterungen bezüglich verschiedener Chunk- oder Spalten-Typen im System vorgenommen werden. Auch eine Kombination verschiedener Techniken wäre denkbar und muss näher betrachtet werden.

Memory-Mapped-Files können dem CameLOD-Projekt eine transparente Schnittstelle zur Speicherung der Indexstrukturen auf Festplatte ermöglichen. Es wurden die Zugriffszeiten einfacher Memory-Mapped-Files untersucht. Zur Integration hingegen müssen Konzepte zur Speicherung in einer oder mehreren Dateien entwickelt werden. Da das Referenzsystem nur 2 NUMA-Knoten zur Verfügung hatte, konnte fast kein Unterschied zwischen den TBB-Schedulern erzielt werden. Außerdem bietet TBB eine geringe Unterstützung bezüglich NUMA-Systemen. Eine genauere Betrachtung ist an dieser Stelle nötig, zum einen auf einem System mit einer höheren Anzahl von NUMA-Knoten und zum anderen durch einen besseren Scheduler. Denkbar ist auch der Ersatz der TBB-Funktionalität durch ein anderes Framework mit expliziter Unterstützung von NUMA.

„Everything that has a beginning has an end, Neo.“

Matrix Revolutions

Literatur

[Ale01] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. C++ in-depth series. Addison-Wesley, 2001. ISBN: 9780201704310.

[ASH08] Medha Atre, Jagannathan Srinivasan und James A Hendler. „BitMat: A Main-memory Bit Matrix of RDF Triples for Conjunctive Triple Pattern Queries." In: *International Semantic Web Conference (Posters & Demos)*. 2008.

[Atr+09] Medha Atre u. a. „Bitmat: An in-core rdf graph store for join query processing". In: *Rensselaer Polytechnic Institute Technical Report* (2009).

[BHB09] Christian Bizer, Tom Heath und Tim Berners-Lee. „Linked data-the story so far". In: *International Journal on Semantic Web and Information Systems (IJSWIS)* 5.3 (2009), S. 1–22.

[Bit] BitMagic. *D-gap Compression Scheme*. http://bmagic.sourceforge.net/dGap.html. 20.11.2013.

[BMK99] Peter A Boncz, Stefan Manegold und Martin L Kersten. „Database architecture optimized for the new bottleneck: Memory access". In: *VLDB*. Bd. 99. 1999, S. 54–65.

[Booa] Boost. *Boost Documentation for hash implementation*. http://www.boost.org/doc/libs/1_54_0/doc/html/hash/reference.html. 22.09.2013.

[Boob] Boost. *Boost Source for bitset implementation*. http://www.boost.org/doc/libs/1_54_0/boost/dynamic_bitset/dynamic_bitset.hpp. 22.09.2013.

[BÖS11] Joppe W Bos, Onur Özen und Martijn Stam. „Efficient hashing using the AES instruction set". In: *Cryptographic Hardware and Embedded Systems–CHES 2011*. Springer, 2011, S. 507–522.

[Bri+11] Nieves R Brisaboa u. a. „Compressed string dictionaries". In: *Experimental Algorithms*. Springer, 2011, S. 136–147.

[CJ] Richard Cyganiak und Anja Jentzsch. *Linking Open Data cloud diagram*. http://lod-cloud.net/. 20.07.2013.

[Cor+05] Jean-Sébastien Coron u. a. „Merkle-Damgård revisited: How to construct a hash function". In: *Advances in Cryptology–CRYPTO 2005*. Springer. 2005, S. 430–448.

[Cyg05] Richard Cyganiak. „A relational algebra for SPARQL". In: *Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170* (2005), S. 35.

[DBP] DBPedia. *DBPedia-Project*. http://dbpedia.org/About. 27.04.2013.

[Dre07] Ulrich Drepper. „What every programmer should know about memory". In: *Red Hat, Inc* (2007).

[Fär+12] Franz Färber u. a. „The SAP HANA Database–An Architecture Overview.“ In: *IEEE Data Eng. Bull.* 35.1 (2012), S. 28–33.

[Fer+13] Javier D. Fernández u. a. „Binary RDF Representation for Publication and Exchange (HDT)“. In: *Journal of Web Semantics* (2013). ISSN: 1570-8268.

[FOA] FOAF. *Friend-of-a-Friend-Project*. http://www.foaf-project.org/. 27.04.2013.

[Gau+08] Praveen Gauravaram u. a. „Grøstl–a SHA-3 candidate“. In: *Submission to NIST* (2008).

[GS92] Hector Garcia-Molina und Kenneth Salem. „Main memory database systems: An overview“. In: *Knowledge and Data Engineering, IEEE Transactions on* 4.6 (1992), S. 509–516.

[Gue+12] Jorge Guerra u. a. „Software persistent memory“. In: *Proc. of the USENIX Annual Technical Conf., Boston, MA*. 2012.

[Har05] Steve Harris. „SPARQL query processing with conventional relational database systems“. In: *International Workshop on Scalable Semantic Web Knowledge Base Systems*. 2005, S. 235–244.

[heia] heise. *Details zu Intels neuen Instruktionssätzen AVX512, MPX, SMAP*. http://heise.de/-1930039. 05.08.2013.

[heib] heise. *Intel Server mit 12-TByte-RAM*. http://heise.de/-1839164. 04.05.2013.

[heic] heise. *Oracle kündigt erneut In-Memory-Technik an*. http://heise.de/-1964197. 03.10.2013.

[HM08] Mark D Hill und Michael R Marty. „Amdahl's law in the multicore era“. In: *Computer* 41.7 (2008), S. 33–38.

[HS13] Stefan Hagedorn und Kai-Uwe Sattler. „Efficient Parallel Processing of Analytical Queries on Linked Data“. Graz, Sep. 2013.

[Idr+12] Stratos Idreos u. a. „MonetDB: Two decades of research in column-oriented database architectures“. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 35 (2012), S. 40–45.

[Inta] Intel. *A Guide to Auto-vectorization with Intel C++ Compilers*. http://download-software.intel.com/sites/default/files/8c/a9/CompilerAutovectorizationGuide.pdf. 20.11.2013.

[Intb] Intel. *E5-2630 Datasheet*. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-1600-2600-vol-1-datasheet.pdf. 12.11.2013.

[Intc] Intel. *E5-2630 Webpage*. http://ark.intel.com/de/products/64593. 20.11.2013.

[Intd] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html. 01.03.2013.

[Inte] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf. 23.10.2013.

[Intf] Intel. *Intel AES-NI*. http://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf. 20.11.2013.

[Intg] Intel. *Intel Architecture Instruction Set Extensions Programming Reference*. http://download-software.intel.com/sites/default/files/319433-016.pdf. 23.11.2013.

[Inth] Intel. *Intel Instruction Extensions*. http://software.intel.com/en-us/intel-isa-extensions. 17.11.2013.

[Inti] Intel. *Intel Threading Building Blocks: Ready for Non-Uniform Memory Access Platforms*. http://software.intel.com/sites/billboard/article/non-uniform-memory-access-gains-and-challenges. 08.11.2013.

[Intj] Intel. *VTune Amplifier XE 2013*. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe-documentation. 15.11.2013.

[Jar+12] Sverre Jarp u. a. „Comparison of Software Technologies for Vectorization and Parallelization". In: *CERN openlab* (2012).

[KN11] A. Kemper und T. Neumann. „HyPer: A hybrid OLTP amp;OLAP main memory database system based on virtual memory snapshots". In: *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. 2011, S. 195–206.

[Kru+12] Jens Krueger u. a. „Leveraging Compression in In-Memory Databases". In: *DBKDA 2012, The Fourth International Conference on Advances in Databases, Knowledge, and Data Applications*. 2012, S. 147–153.

[KS11] Shmuel T Klein und Dana Shapira. „Huffman coding with non-sorted frequencies". In: *Mathematics in Computer Science* 5.2 (2011), S. 171–178.

[KSL13] Tim Kiefer, Benjamin Schlegel und Wolfgang Lehner. „Experimental Evaluation of NUMA Effects on Database Management Systems." In: *BTW*. 2013, S. 185–204.

[LB13] Daniel Lemire und Leonid Boytsov. „Decoding billions of integers per second through vectorization". In: *Software: Practice and Experience* (2013).

[Leh+13] Jens Lehmann u. a. „DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia". In: *Semantic Web Journal* (2013). Under review.

[Lem+10] Ch. Lemke u. a. „Speeding Up Queries in Column Stores - A Case for Compression". In: *DaWak*. 2010, S. 117–129.

[LKN13] Viktor Leis, Alfons Kemper und Thomas Neumann. „The Adaptive Radix Tree: ARTful indexing for main-memory databases". In: ICDE. 2013.

[MAF12] Miguel A. Martínez-Prieto, Mario Arias und Javier D. Fernández. „Exchange and Consumption of Huge RDF Data". In: *9th Extended Semantic Web Conference (ESWC)*. 2012, S. 437–452.

[NW10] Thomas Neumann und Gerhard Weikum. „The RDF-3X engine for scalable management of RDF data". In: *The VLDB Journal* 19.1 (2010), S. 91–113.

[Rei10] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2010. ISBN: 9781449390860.

[Roh+07] Kurt Rohloff u. a. „An evaluation of triple-store technologies for large data stores". In: *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*. Springer. 2007, S. 1105–1114.

[Ros+13] Corentin Rossignon u. a. „A NUMA-aware fine grain parallelization framework for multi-core architecture". In: *PDSEC-14th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing-2013*. 2013.

[Sal07] D. Salomon. *Data Compression: The Complete Reference*. Molecular biology intelligence unit. Springer, 2007. ISBN: 9781846286032.

[sgi] sgi. *C++ STL Documentation*. http://www.sgi.com/tech/stl/bit_vector.html. 22.09.2013.

[SHB06] Nigel Shadbolt, Wendy Hall und Tim Berners-Lee. „The semantic web revisited". In: *Intelligent Systems, IEEE* 21.3 (2006), S. 96–101.

[Ste+09] Florian Stegmaier u. a. „Evaluation of current RDF database solutions". In: *Proceedings of the 10th International Workshop on Semantic Multimedia Database Technologies (SeMuDaTe), 4th International Conference on Semantics And Digital Media Technologies (SAMT)*. 2009, S. 39–55.

[VTS11] Haris Volos, Andres Jaan Tack und Michael M Swift. „Mnemosyne: Lightweight persistent memory". In: *ACM SIGARCH Computer Architecture News*. Bd. 39. 1. ACM. 2011, S. 91–104.

[w3c] w3c. *RDF/XML Syntax Specification*. http://www.w3.org/TR/rdf-syntax-grammar/. 07.10.2013.

[w3o] w3.org. *LargeTripleStores*. http://www.w3.org/wiki/LargeTripleStores. 27.04.2013.

[Wik] Wikipedia. *Ilmenau*. http://de.wikipedia.org/wiki/Ilmenau. 27.04.2013.

[WKB08] Cathrin Weiss, Panagiotis Karras und Abraham Bernstein. „Hexastore: sextuple indexing for semantic web data management". In: *Proceedings of the VLDB Endowment* 1.1 (2008), S. 1008–1019.

[WZ99] Hugh E Williams und Justin Zobel. „Compressing integers for fast file access". In: *The Computer Journal* 42.3 (1999), S. 193–201.

[YDS09] Hao Yan, Shuai Ding und Torsten Suel. „Inverted index compression and query processing with optimized document ordering". In: *Proceedings of the 18th international conference on World wide web*. ACM. 2009, S. 401–410.

[Zuk+05] Marcin Zukowski u. a. „MonetDB/X100-A DBMS In The CPU Cache." In: *IEEE Data Eng. Bull.* 28.2 (2005), S. 17–22.

[Zuk+06] Marcin Zukowski u. a. „Super-scalar RAM-CPU cache compression". In: *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*. IEEE. 2006, S. 59–59.

Anhang A

Anfragen

Für die durchgeführen Messungen wurden verschiedene Anfragen benutzt. Da nicht alle Anfragen in der Arbeit dargestellt wurden, sollen hier die CameLOD-Algebra der fehlenden Anfragen aufgelistet werden.

Q6

```
$1 := scan(s-index, =[_1, ↘
   →"<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>"]);
$2 := scan(s-index, =[_1, ↘
   →"<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>"]);
$3 := merge_join($1, $2, _0, _0);
$4 := filter($3, <>[_2, _5]);
null_op($4)
```

Q7

```
$1 := scan(p-index, ↘
   →"<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>");
$2 := index_join($1, s-index, _0, =[_1, ↘
   →"<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>"]);
$3 := filter($2, <>[_2, _5]);
null_op($3)
```

Q8

```
$1 := spatial_scan(intersects, region 29 -127 49 -68);
null_op($1)
```

Q9

```
$1 := co_scan(s-index);
$2 := sorted_aggregation($1, [count(_1)], [_0]);
$3 := parallel_do($2,12);
$4 := sort($3, [_1], [_0, _1], limit 50, desc);
printer($4)
```

Q10

```
$1 := co_scan(p-index, >=[_0, "<http://dbpedia.org/resource/F>"]);
$2 := filter($1, <[_0, "<http://dbpedia.org/resource/H>"]);
```

```
$3 := sorted_aggregation($2, [count(_0)], [_1]);
$4 := parallel_do($3, 12);
$5 := sort($4, [_1], [_0, _1], limit 50, desc);
null_op($5)
```

Q11

```
$1 := co_scan(p-index, ↘
   →"<http://dbpedia.org/ontology/wikiPageWikiLink>");
$2 := filter($1, =[_0,
"<http://dbpedia.org/resource/Alphabetical_list_of_comuni_of_Italy>"]);
$3 := parallel_do($2,24);
null_op($3)
```

Q12

```
$1 := co_scan(p-index, ↘
   →"<http://dbpedia.org/ontology/wikiPageWikiLink>",=[_2, ↘
   →"<NULL>"]);
$2 := parallel_do($1,24);
null_op($2)
```

Q13

```
$1 := co_scan(p-index, ↘
   →"<http://dbpedia.org/ontology/wikiPageWikiLink>");
$2 := parallel_do($1,12);
null_op($2)
```

Eidesstattliche Erklärung

Ich, Steve Göring, Matrikel-Nr 43952, versichere hiermit, dass ich meine Masterarbeit mit dem Thema:

"*Effiziente In-Memory Verarbeitung von SPARQL-Anfragen auf großen Datenmengen* "

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ilmenau, den

Steve Göring