FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Effiziente Evaluierung von Well-Designed Pattern Trees mit Hilfe von In-Memory Datenbanken

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Michael Heinzl
Matrikelnummer 1325545

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.-Prof. Mag. Dr. Reinhard Pichler
Mitwirkung: DI Wolfgang Fischl, BSc

Wien, 19. Juli 2016

Michael Heinzl	Reinhard Pichler

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Efficient Evaluation of Well-Designed Pattern Trees with the help of In-Memory Databases

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Business Informatics

by

Michael Heinzl
Registration Number 1325545

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.-Prof. Mag. Dr. Reinhard Pichler
Assistance: DI Wolfgang Fischl, BSc

Vienna, 19th July, 2016

Michael Heinzl | Reinhard Pichler

Erklärung zur Verfassung der Arbeit

Michael Heinzl
Xaveriweg 4, 7000 Eisenstadt

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19. Juli 2016

Michael Heinzl

Inhaltsverzeichnis

KAPITEL 1

Einleitung

Motivation

Heutzutage steht uns eine große Menge an Daten zur Verfügung. Diese Daten werden meist in relationalen Datenbankmanagementsystemen (DBMS) verwaltet. In den 1970er wurden die ersten relationalen DBMS entwickelt und auch die Abfragesprache SQL. Zu dem Zeitpunkt der Entwicklung wurde die Annahme getroffen, dass die Daten in relationalen DBMS vollständig sind. Dies ist aber häufig nicht der Fall. In der folgenden Abbildung 1.1 ist eine Datenbank schematisch dargestellt, welche in den nächsten Beispielen verwendet wird.

Gebirge	
A	**B**
Mount Everest	Himalaya
K2	Karakorum

Höhe	
A	**C**
Mount Everest	8848
K2	8611

ErstbesteigungWinter	
A	**D**
Mount Everest	17.02.1980

Kontinent	
B	**E**
Himalaya	Asien
Karakorum	Asien

Abbildung 1.1: Datenbank Beispiel

Die SQL Abfrage (Listing 1.1) liefert in diesem Fall keinen Treffer, da der Gipfel **K2** mit der Höhe von **8611** noch nie im Winter bestiegen worden ist. Diese Ergebnis kommt

zustande, da die Tabelle *ErstbesteigungWinter* unvollständig ist. Wenn man auf die zusätzlichen Information der Tabelle *ErstbesteigungWinter* verzichtet, liefert diese Abfrage ein Ergebnis.

Listing 1.1: SELECT DB Beispiel

```
SELECT Gebirge.A, Gebirge.B, ErstbesteigungWinter.D, Kontinent.E
        FROM Gebirge
                JOIN Höhe
                        ON Gebirge.A = Höhe.A
                JOIN ErstbesteigungWinter
                        ON Gebirge.A = ErstbesteigungWinter.D
                JOIN Kontinent
                        ON Gebirge.A = Kontinent.E
        WHERE Höhe.C = "8611"
```

Conjunctive queries (Select-From-Where Abfragen) sind die Grundlage der heutigen DBMS. Diese Art von Abfragen haben ein Problem mit unvollständigen bzw. semistrukturierten Daten. Wenn eine Abfrage nicht exakt mit den Daten übereinstimmt, kann keine Antwort ermittelt werden. Dieses Problem wird in der Abfragesprache SPARQL thematisiert. [BPS15]

Im Semantic Web wird die Abfragesprache SPARQL verwendet. SPARQL steht für *SPARQL Protocol And RDF Query Language* und beruht auf dem Datenmodel RDF (Resource Description Framework). RDF ist das Standard Datenmodell des Sematic Web und ist eine Empfehlung des W3C (World Wide Web Consortium).

Eine besondere Bedeutung in SPARQL hat der OPTIONAL Operator. Diese Operation ermöglicht, Abfragen zu formulieren, welche das Ergebnis um gewisse Teile erweitert falls diese verfügbar sind. Dabei werden die Teilergebnisse bei denen es nicht zutrifft, nicht verworfen, sondern bleiben Teil der Lösung. In [LPPS13] wird hervorgehoben, dass mehr als 45% der Abfragen auf dem SPARQL endpoint DBPedia den OPTIONAL Operator verwenden. Zusammen mit dem AND Operator kann das {AND, OPT}-Fragment als Baum dargestellt werden. In [LPPS13] wurde eine Baumdarstellung von SPARQL Abfragen eingeführt, die sogennaten SPARQL *pattern trees* und im speziellen die *well-designed pattern trees* (WDPTs). Die Baumdarstellung der Abfrage spielt eine zentrale Rolle für Optimierungen.

In Listing 1.2 wird gezeigt wie man durch den Einsatz des OPTIONAL Operators auch Teilergebnisse einer Abfrage erhält.

Das Ergebnis der SPARQL Abfrage (Listing 1.2) enthält das Ergebnis der Wurzel, sowie die Teilergebnisse der OPTIONAL Knoten. (Abbildung 1.3)

In dieser Arbeit gehen wir davon aus, dass die einzelnen Knoten des WDPT ausgewertet sind. Dies ermöglicht uns, dass unser Algorithmus unabhängig von der in den Knoten verwendeten Abfragesprache ist. So ist es auch möglich nicht nur den SPARQL

Listing 1.2: SPARQL DB Beispiel

```
SELECT ?A, ?B, ?D, ?E
        WHERE { ?A Gebirge ?B . ?A Höhe "8611" .
        OPTIONAL { ?A ErstbesteigungWinter ?D } .
        OPTIONAL { ?B Kontinent ?E }}
```

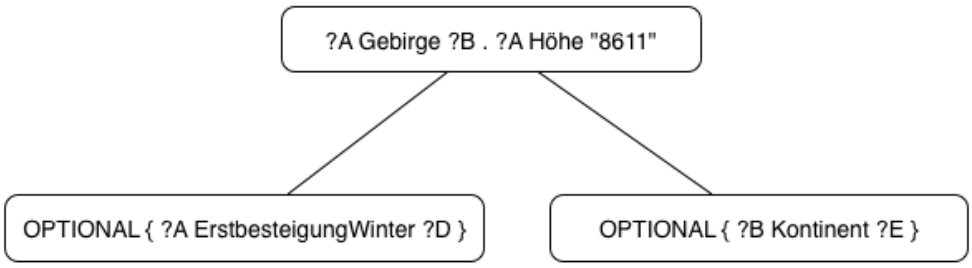

Abbildung 1.2: SPARQL Tree

A	B	D	E
K2	Karakorum		Asien

Abbildung 1.3: SPARQL Ergebnis

Musterabgleich in den Knoten zu verwenden, sondern es können auch allgemeine Conjunctive Queries oder relationale Daten formuliert werden. Dies wurde erst kürzlich vorgeschlagen [BPS15].

Im folgendem Beispiel 1.3 ist ein RDF Graph *G* dargestellt, welcher das Datenbank Beispiel 1.1 repräsentiert. In Listing 1.4 sind die ausgewerteten Knoten des WDPT über den RDF Graph *G* zu finden. Der erste Knoten repräsentiert den Hauptteil der SPARQL Abfrage *?A Gebirge ?B . ?A Hohe "8611"* und ist somit der Wurzelknoten. Der Wurzelknoten beinhaltet die Belegungen der Variablen, welche der Hauptbedingung der SPARQL Abfrage entsprechen und ein Kindknoten. Die Belegung enthält für *?A "K2"* und für *?B "Karakorum"*. Die Kindknoten des Wurzelknotens repräsentieren die OPTIONAL Operatoren der SPARQL Abfrage. Für den ersten OPTIONAL Teil der Abfrage *OPTIONAL{ ?A ErstbesteigungWinter ?D }* wurden keine Belegungen gefunden, somit ist dieser Knoten nicht vorhanden. Der Kindknoten weist für den Teil *OPTIONAL { ?B Kontinent ?E }* eine passende Belegung der Variablen auf, für die Variable *?B "Karakorum"* und für die Variable *?E Äsien"*.

Listing 1.3: Beispiel RDF Graph

```
G = {
        ("Mount Everest",Gebirge,"Himalaya"),
        ("Mount Everest",Höhe,"8848"),
        ("Mount Everest",ErstbesteigungWinter,"17.02.1980"),
        ("K2",Gebirge,"Karakorum"),
        ("K2",Höhe,"8611"),
        ("Himalaya",Kontinent,"Asien"),
        ("Karakorum",Kontinent,"Asien"),
}
```

Listing 1.4: Beispiel WDPT

```
<?xml version="1.0" encoding="UTF-8"?>
<ptresult>
  <node> <!-- ?A Gebirge ?B . ?A Höhe "8611" -->
    <mapping>
      <var name="?A">K2</var>
      <var name="?B">Karakorum</var>
    </mapping>

    <node> <!-- OPTIONAL { ?B Kontinent ?E } -->
      <mapping>
        <var name="?B">Karakorum</var>
        <var name="?E">Asien</var>
      </mapping>
    </node>

  </node>
</ptresult>
```

In WDPT werden die ausgewerteten Knoten mit Hilfe des OPTIONAL Operators zusammengeführt. Derzeit wird dafür der Top-Down-Algorithmus [LPPS13] in einer iterativen Variante implementiert [AFK+16].

Diese Arbeit beschäftigt sich mit der effizienten Auswertung der WDPTs. Diese Auswertung wird mit Hilfe eines DBMS und des LEFT OUTER JOIN realisiert. Um eine möglichst effiziente Auswertung zu ermöglichen wird eine In-Memory Datenbank verwendet. In-Memory Datenbanken haben den Vorteil, dass die Daten nur im Arbeitsspeicher gehalten werden und somit schneller Ergebnisse liefern können.

Ziel und Hauptresultat

Die derzeitige Implementierung [AFK+16] des Algorithmus *Top-Down Evaluation* [LPPS13] basiert auf einem einfachen iterativen Vorgehen. Das Ziel ist die Implementierung eines Algorithmus, welcher unter Zuhilfenahme einer In-Memory Datenbank die Leistung verbessert und somit die Durchlaufzeit verringert. Bei diesen Tests kommen verschiedene In-Memory Datenbanken zum Einsatz, um für diesen speziellen Anwendungsfall eine geeignete Implementierung auszuwählen. Nach Auswertung der Algorithmen konnten die Ergebnisse verglichen werden. Der neue Algorithmus ist durch die Verwendung einer In-Memory Datenbank um bis zu 160-mal schneller als der iterative Algorithmus.

Aufbau der Arbeit

Die Arbeit ist folgendermaßen aufgebaut. Zu Beginn in Kapitel 2 werden Basisdefinitionen erläutert. In Kapitel 3 wird die derzeitige Implementierung beschrieben und analysiert. Im speziellen wird die Leistung des derzeitigen Algorithmus als Orientierungswert für den späteren Vergleich herangezogen. Des Weiteren wird die Datenstruktur der Eingabedaten erläutert, diese Daten dienen als Eingabe für die derzeitige und zukünftige Implementierung.

Das Kapitel 4 beschäftigt sich mit Konzepten und Anforderungen im Bereich In-Memory Datenbanken.

Danach wird im Kapitel 5 die neue Implementierung unter der Verwendung von einer In-Memory Datenbank erläutert. Abschließend wird der neue Algorithmus mit dem bisherigen Algorithmus verglichen.

KAPITEL 2

Basisdefinitionen

RDF Resource Description Framework (RDF) ist ein Datenmodell für den Datenaustausch im Web. Die Grundstruktur begünstigt das Zusammenführen von Daten, auch wenn sich die zugrundeliegenden Schemata unterscheiden. Ein weiterer Vorteil liegt darin, dass einfach Änderungen am Schema vorgenommen werden können ohne, dass sich der Benutzer anpassen muss. RDF baut auf der Struktur von URIs auf. URIs werden verwendet, um ein Verknüpfung und deren Enden zu identifizieren. Diese Verknüpfung ergibt ein Tripel.

I, B und L (IRIs, leere Knoten und Literale) sind paarweise disjunkt. Ein Tripel $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ ist ein RDF Tripel. Dabei ist s das Subjekt, p das Prädikat und o das Objekt. Ein RDF Graph ist jene Menge aus RDF Tripeln. Leere Knoten sind nicht zulässig.
[Gro14, PAG09]

SPARQL SPARQL Protocol And RDF Query Language (SPARQL) ist eine graphenbasierte Abfragesprache für RDF. Grundsätzlich besteht SPARQL aus diesen drei Teilen: Musterabgleich, Lösungsmodifikation und Ausgabe.

Der Musterabgleich beschäftigt sich unter anderem mit dem Abgleich von Mustern in Graphen, wie z.B. optionale Muster, Vereinigung von Mustern oder Filterung von Mustern. Die Lösungsmodifikation macht es uns möglich, nach der Auswertung die Daten mit klassischen Operationen wie z.B. *Projektion*, *distinct*, *limit* oder *offset* zu bearbeiten. Die Ausgabe kann verschiedene Typen annehmen: Ja/Nein Abfragen, Selektion von Werten von Variablen, welche mit einem Muster übereinstimmen, Konstruktion neuer Tripel und Beschreibung von Ressourcen.

Um SPARQL genauer zu erläutern, muss eine gewisse Terminologie eingeführt werden. Sei V eine unendliche Menge an Variablen und U ist eine abzählbar unendliche Menge aus URIs. $U \cap V = \emptyset$. Variablen werden mit einem führenden Fragezeichen markiert z.B.

?x. Ein SPARQL Tripel t ist ein Tupel in $(U \cup V) \times (U \cup V) \times (U \cup V)$. Die Vereinigung $I \cup B \cup L$ ist T. Ein *Mapping* μ von V nach T ist eine partielle Funktion $\mu : V \to T$. $\mu(t)$ symbolisiert die Ersetzung der Variablen im Tripel t entsprechend zu μ. Die Domain von μ, dom(μ), ist das Subset von V in dem μ definiert ist. Zwei Mappings μ_1 und μ_2 sind *kompatibel*, wenn für alle $x \in dom(\mu_1) \cap dom(\mu_2)$, ist der Fall, dass $\mu(x) = \mu(x)$, das heißt wenn $\mu_1 \cup \mu_2$ auch ein Mapping ist. Das leere Mapping ist mit jeden anderen Mapping kompatibel. Ebenfalls sind zwei Mappings mit disjunkten Domains kompatibel. Ω_1 und Ω_2 sind Mengen aus Mappings. JOIN, UNION und DIFFERENCE dieser Mengen sind wie folgt definiert:

$\Omega_1 \bowtie \Omega_2 = \{ \mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ sind kompatibel Mappings} \}$,

$\Omega_1 \cup \Omega_2 = \{ \mu \mid \mu \in \Omega_1 \textit{ or } \mu \in \Omega_2 \}$,

$\Omega_1 \setminus \Omega_2 = \{ \mu \in \Omega_1 \mid \text{für alle } \mu' \in \Omega_2, \mu \text{ und } \mu' \text{ sind nicht kompatibel} \}$.

Beruhend auf diesen Operationen ist der LEFT OUTER JOIN wie folgt definiert:

$\Omega_1 \text{⟕} \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$

Nun können wir die Semantik der *graph pattern* Ausdrücke als Funktion $[\![\cdot]\!]_D$ darstellen. Diese Funktion nimmt als Parameter ein Pattern und gibt eine Menge von Mappings zurück. D bezeichnet die zugrunde liegende RDF Datenmenge und t ein *triple pattern*. Die Auswertung des *graph pattern* über D wird folgendermaßen definiert:

$[\![t]\!]_D = \{ \mu \mid dom(\mu) = var(t) \textit{ und } \mu(t) \in D \}$, wobei $var(t)$ die Menge von Variablen in t ist.

Nun können wir die SPARQL Operatoren *AND*, *OPT* und *UNION* wie folgt definieren, wobei P_1 und P_2 *graph pattern* sind:

$[\![(P_1 \textit{ AND } P_2)]\!]_D = [\![P_1]\!]_D \bowtie [\![P_2]\!]_D$

$[\![(P_1 \textit{ OPT } P_2)]\!]_D = [\![P_1]\!]_D \text{⟕} [\![P_2]\!]_D$

$[\![(P_1 \textit{ UNION } P_2)]\!]_D = [\![P_1]\!]_D \cup [\![P_2]\!]_D$

Listing 2.1: SPARQL Beispiel

```
SELECT ?A, ?B, ?D
        WHERE { ?A Gebirge ?B . ?A Höhe "8611" .
        OPTIONAL { ?A ErstbesteigungWinter ?D }}
```

Das Listing 2.1 zeigt ein Beispiel einer SPARQL Abfrage. Der SELECT Teil wählt die Variablen für das Ergebnis aus. Im WHERE Teil wird das Anfragemuster dargestellt, dieses verwendet die Tripel Syntax. Der Punkt repräsentiert eine AND Verknüpfung. Der OPTIONAL Operator fügt die Teilergebnisse nur hinzu, wenn diese vorhanden sind.

[PS08, PAG09]

WDPT Ein *pattern tree* (PT) $\mathcal{T}$ ist ein Paar aus $(T, \mathcal{P})$, wobei $T = (V, E, r)$ ein ungeordneter Baum ist. V ist die Menge der Knoten des Baums, E die Menge der ungerichteten Knoten des Baums und $r \in V$ ist der Wurzelknoten. $\mathcal{P} = \{P_n \mid n \in V\}$ ist die Beschriftung der Knoten in V, wobei P_n eine Menge von *triple pattern* ist. Ein Teilbaum von $\mathcal{T}$ mit der Wurzel n wird auch als $\mathcal{T}_n$ bezeichnet. Mit $vars(P_n)$ wird die Menge von Variablen bezeichnet, welche in P_n vorkommen und mit $vars(\mathcal{T})$ wird die Menge $\bigcup_{n \in V} vars(P_n)$ bezeichnet. Die geschachtelten OPTIONAL Operatoren des *well-designed SPARQL pattern* eignen sich, um diese als Baumstruktur darzustellen. Hierbei sind die *triple pattern* in den Knoten des Baumes enthalten.

Im Folgenden wird beschrieben, wie *pattern trees* in *SPARQL graph pattern* transformiert werden. Gegeben ein *pattern tree* $\mathcal{T}$ und eine Menge Σ von Funktionen $\{ \sigma_n \mid n \in V \}$ für jeden Knoten $n \in V$. Die Funktion σ_n definiert eine Reihenfolge für die Kindknoten von n. Weiters ist die Menge $P = \{t_1, ..., t_l\}$ aus *triple patterns* und $and(P)$ die *graph pattern* $(t_1\ AND\ t_2\ AND\ ...\ AND\ t_l)$. Die Transformation für $\mathcal{T}$ und $n \in V$, wobei n k Kindknoten hat, ist wie folgt definiert: $TR(\mathcal{T}, n, \Sigma)$ entspricht diesem *graph pattern*

$$(...((and(P_n)\ OPT\ TR(\mathcal{T}, \sigma_n(1), \Sigma))\quad OPT\ TR(\mathcal{T}, \sigma(2), \Sigma))\quad ...\ OPT\ TR(\mathcal{T}, \sigma(k), \Sigma)).$$

Wenn n keine Kindknoten aufweist ist $TR(\mathcal{T}, n, \Sigma) = and(P_n)$.

Hiermit ist zwischen *pattern trees* und *SPARQL graph patterns* eine syntaktische Beziehung hergestellt, im weiteren wird eine semantische Beziehung hergestellt. Beginnend mit der Definition der WDPT (*well-designed pattern tree*): Ein *pattern tree* $\mathcal{T} = ((V, E, r), \mathcal{P})$ ist *well designed*, wenn für jede Variable $?X$ in $\mathcal{T}$, die Knoten $\{ n \in V \mid ?X \in vars(P_n) \}$ einen verbunden Subgraphen beinhalten.

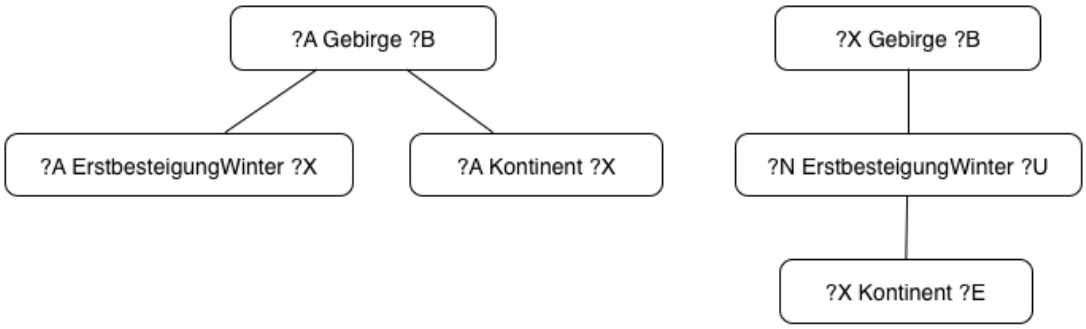

Abbildung 2.1: keine WDPT

In beiden Bäumen in Abbildung 2.1 wird durch die Variable $?X$ die *well designed* Bedingung nicht erfüllt.

Weiters wird eine abgeschwächte Form des WDPT eigeführt, der *quasi well-designed pattern tree*. Die Klasse der *quasi well-designed pattern trees* QWDPTs wird verwendet um Redundanzen in *pattern trees* zu entfernen. Ein *pattern tree* $\mathcal{T} = ((V, E, r), (P_n)_{n \in V})$ ist ein *quasi well-designed pattern tree*, wenn für jedes Paar aus Knoten $u, v \in V$ und Variable

$?X \in vars(P_u) \cap vars(P_v)$ ein Knoten n existiert, sodass u und v einen gemeinsamen Vorfahren haben und $?X \in vars(P_n)$ gilt. Der linke Baum in Abbildung 2.1 ist kein QWDPT, der rechte Baum in Abbildung 2.1 jedoch schon.

Um im folgenden zu zeigen, dass jeder QWDPT in einen WDPT umgewandelt werden kann, wird das Konzept *Duplizierung von Tripel in Kindknoten* beschrieben. Ein *pattern tree* $\mathcal{T}' = ((V', E', r'), (P'_n)_{n \in V'})$ wurde von einem *pattern tree* $\mathcal{T} = ((V, E, r), (P_n)_{n \in V})$ abgeleitet und ein Tripel dupliziert, dies wird mit $\mathcal{T} \hookrightarrow \mathcal{T}'$ bezeichnet, wenn $(V', E', r') = (V, E, r)$, ein Knoten $u \in V$ existiert, ein Tripel $t \in P_u$ und ein Kindknoten v von u, sodass $P'_v = P_v \cup \{t\}$ und $P_n = P'_n$ für alle $n \neq v$ gilt. Weiters wird mit $\hookrightarrow^*$ die reflexive und transitive Hülle von $\hookrightarrow$ bezeichnet, wenn $\mathcal{T} = \mathcal{T}'$ oder es existiert eine Folge $\mathcal{T}_1 \hookrightarrow \mathcal{T}_2 \hookrightarrow .. \hookrightarrow \mathcal{T}_m$ mit $\mathcal{T}_1 = \mathcal{T}$ und $\mathcal{T}_m = \mathcal{T}'$. Durch dieses Konzept kann jeder QWDPT in einen WDPT umgewandelt werden, in den man Tripel entlang des Baumes dupliziert.

Die Reihenfolge der Kindknoten in einem WDPT beeinflusst nicht die Semantik und für zwei beliebige Reihenfolgen sind die Ergebnisse der Transformationen der SPARQL *graph patterns* gleich. $\mathcal{T}$ ist ein WDPT. Σ_1, Σ_2 sind beliebige Reihenfolgen von $\mathcal{T}$ und sei $P_1 = TR(\mathcal{T}, \Sigma_1)$ und $P_2 = TR(\mathcal{T}, \Sigma_2)$ die *graph pattern* der Transformation, dann gilt $P_1 \equiv P_2$. Die Semantik ist unabhängig von der konkreten Folge $\mathcal{T} \hookrightarrow^* \mathcal{T}'$ und alle WDPTs die von $\mathcal{T}$ durch Duplizierung von Tripel abgeleitet werden können sind äquivalent. Sei $\mathcal{T}$ ein QWDPT, Σ eine Reihenfolge für $\mathcal{T}$ und sei $\mathcal{T}_1$ und $\mathcal{T}_2$ WDPTs, sodass $\mathcal{T} \hookrightarrow^* \mathcal{T}_1$ und $\mathcal{T} \hookrightarrow^* \mathcal{T}_2$, wenn $P_1 = TR(\mathcal{T}_1, \Sigma)$ und $P_2 = TR(\mathcal{T}_2, \Sigma)$, dann gilt $P_1 \equiv P_2$.

Eine Menge von SPARQL *graph patterns* von einem QWDPT $\mathcal{T}$ ist wie folgt definiert: $SEM(\mathcal{T}) = \{ TR(\mathcal{T}', \Sigma) \mid \Sigma \text{ ist eine Sortierung für } \mathcal{T}', \mathcal{T} \hookrightarrow^* \mathcal{T}' \text{ und } \mathcal{T}' \text{ ist } \textit{well designed} \}$. Sei $\mathcal{T}$ ein QWDPT. Alle *graph patterns* in $SEM(\mathcal{T})$ sind äquivalent. Für zwei beliebige *graph patterns* $P_1, P_2 \in SEM(\mathcal{T})$ gilt $P_1 \equiv P_2$.

Für jeden *well-designed graph pattern* P existiert ein QWDPT $\mathcal{T}$, sodass $P \equiv \mathcal{T}$ und für einen *well-designed graph pattern* kann eine äquivalenter QWDPT in polynomieller Zeit konstruiert werden.

Jeder *well-designed graph pattern* ist äquivalent zu einem *pattern* in *OPT-normal form*. Ein *pattern* der Gestallt $(t_1\ AND\ t_2\ AND\ ...\ AND\ t_k)$ mit t_i *triple patterns* ist in *OPT-normal form*. Wenn P_1 und P_2 in *OPT-normal form* sind, dann ist auch $(P_1\ OPT\ P_2)$ in *OPT-normal form*. Ein *pattern* P in *OPT-normal form* kann wie folgt in einen WDPT umgewandelt werden. Wenn $P = (t_1\ AND\ t_2\ AND\ ...\ AND\ t_k)$, dann konstruieren wir einen Baum mit einen Knoten und der Beschriftung $\{t_1, ..., t_k\}$. Wenn $P = (P_1\ OPT\ P_2)$, dann konstruieren wir einen *pattern tree* $\mathcal{T}_1$ von P_1, einen *pattern tree* $\mathcal{T}_2$ von P_2 und danach konstruieren wir einen *pattern tree* $\mathcal{T}$ aus $\mathcal{T}_1$ und $\mathcal{T}_2$, dabei wird der Wurzelknoten von $\mathcal{T}_2$ als Kindknoten in $\mathcal{T}_1$ eingefügt und der Wurzelknoten von $\mathcal{T}_1$ wird als Wurzelknoten von $\mathcal{T}$ gesetzt. Der konstruierte *pattern tree* $\mathcal{T}$ ist *well designed* und es existiert eine Reihenfolge Σ für $\mathcal{T}$, sodass $TR(\mathcal{T}, \Sigma) = P$ und $\mathcal{T} \equiv P$.

[LPPS13, PAG09]

KAPITEL 3

Ist-Zustand

3.1 Datenstruktur

Der Ausgangspunkt des Algorithmus ist eine XML-Datei, diese repräsentiert die Auswertung mehrerer Knoten in einem Well-Designed Pattern Tree (WDPT). Das Schema der XML-Datei ist unter Listing 3.1 definiert und wird im weiteren genauer erläutert.

Die XML-Datei enthält ein Hauptelement *ptresult*. Dieses wiederum enthält eine Sequenz von Elementen *ovar*, welche die Variablen der Projektion des WDPT abbilden [AFK+16]. Des Weiteren enthält *ptresult* den Wurzelknoten des Well-Designed Pattern Tree. Dieser wird durch ein *node* Element dargestellt.

Der Wurzelknoten des Baums ist die Basis des WDPT. Alle weiteren Kindknoten stellen die Auswertung einer OPTIONAL-Klausel da. Die OPTIONAL-Klauseln können auch geschachtelt verwendet werden, infolgedessen ergibt sich eine Baumstruktur. Diese Knoten sind ebenfalls durch die Elemente *node* abgebildet. Ein *node* Element besteht aus genau einem *variables* Element, beliebig vielen *mapping* Elementen und danach beliebig vielen *node* Elementen. Diese *node* Elemente stellen die Kindknoten dar.

Das *variables* Element enthält mehrere *nodeVar* Elemente. Diese repräsentieren die Variablen, welche innerhalb der zugehörigen OPTIONAL-Klausel des Knotens verwendet werden.

Nun zum eigentlichen Kernstück der XML-Datei. Das *mapping* Element ist eine konkrete Belegung von Variablen der SPARQL-Abfrage. Diese enthalten die eigentlichen Daten. Somit enthält das *var* Element die konkreten Ausprägungen der SPARQL-Variablen, der Name dieser Variable ist dazu in dem Attribut *name* des *var* Elements hinterlegt.

Listing 3.1: XML Schema der Eingabedatei

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:element name="ptresult">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="ovar" type="xs:string" minOccurs="1"
                    maxOccurs="unbounded"/>
                <xs:element name="node" type="nodeType" minOccurs="1"
                    maxOccurs="1"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:complexType name="nodeType">
        <xs:sequence>
            <xs:element name="variables" type="variablesType"
                minOccurs="1" maxOccurs="1"/>
            <xs:element name="mapping" type="mappingType" minOccurs="0"
                maxOccurs="unbounded"/>
            <xs:element name="node" type="nodeType" minOccurs="0"
                maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="variablesType">
        <xs:sequence>
            <xs:element name="nodeVar" type="xs:string" minOccurs="1"
                maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="mappingType">
        <xs:sequence>
            <xs:element name="var" type="varType"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="varType">
        <xs:simpleContent>
            <xs:extension base="xs:string">
                <xs:attribute name="name" type="xs:string"/>
```

```
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
</xs:schema>
```

Weiters ist unter Listing 3.2 eine Beispiel XML-Datei angegeben, welche später als Referenz dient.

Listing 3.2: Beispiel XML Eingabedatei

```
<?xml version="1.0" encoding="UTF-8"?>
<ptresult>
    <ovar>?X1</ovar>
    <ovar>?Y1</ovar>
    <ovar>?Z1</ovar>
    <node> <!-- Tabelle T0 -->
        <variables>
            <nodeVar>?X1</nodeVar>
            <nodeVar>?X2</nodeVar>
        </variables>
        <mapping>
            <var name="?X1">a</var>
            <var name="?X2">b</var>
        </mapping>
        <mapping>
            <var name="?X1">a</var>
            <var name="?X2">c</var>
        </mapping>
        <node> <!-- Tabelle T1 -->
            <variables>
                <nodeVar>?X1</nodeVar>
                <nodeVar>?Y1</nodeVar>
                <nodeVar>?Z1</nodeVar>
                <nodeVar>?Z2</nodeVar>
            </variables>
            <mapping>
                <var name="?X1">a</var>
                <var name="?Y1">c</var>
                <var name="?Z1">d</var>
                <var name="?Z2">d</var>
            </mapping>
            <node> <!-- Tabelle T2 -->
                <variables>
                    <nodeVar>?Z1</nodeVar>
                    <nodeVar>?Z2</nodeVar>
```

```
                <nodeVar>?A1</nodeVar>
            </variables>
            <mapping>
                <var name="?Z1">d</var>
                <var name="?Z2">d</var>
                <var name="?A1">d</var>
            </mapping>
        </node>
    </node>
    <node> <!-- Tabelle T3 -->
        <variables>
            <nodeVar>?X2</nodeVar>
            <nodeVar>?A1</nodeVar>
        </variables>
        <mapping>
            <var name="?X2">asdf</var>
            <var name="?A1">c</var>
        </mapping>
    </node>
</node>
</ptresult>
```

Der Well-Designed Pattern Tree (WDPT) der XML-Datei (Listing 3.2) ist in Abbildung 3.1 dargestellt.

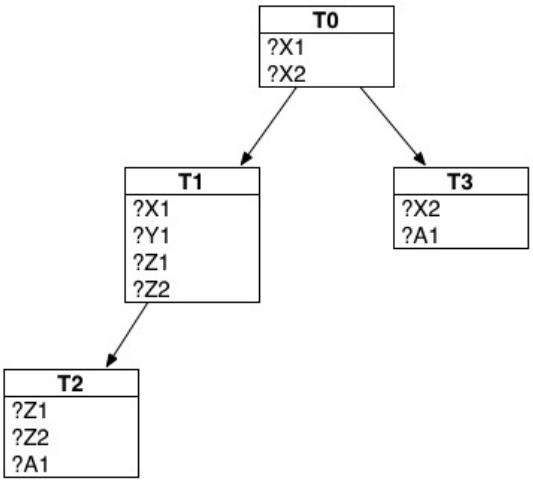

Abbildung 3.1: WDPT der beschriebene XML-Datei (Listing 3.2)

3.2 Implementierung

Die aktuelle Implementierung ohne In-Memory Datenbank besteht aus mehreren Phasen:

Einlesen Die erste Phase besteht darin, eine oben beschriebene XML-Datei einzulesen und die XML-Strukturen in folgende Java Objekte zu konvertieren.

Die Klasse *EvalPT* spiegelt das Hauptelement *ptresult* der XML-Datei wieder. Die Klasse enthält den Wurzelknoten, sowie die Variablen der *ovar* Elemente. Die *node* Elemente werden in in der Klasse *EvalTreeNode* abgebildet. Die Einträge aus dem *variables* Element werden als *Set* in dieser Klasse gespeichert. Zusätzlich enthält die Klasse auch eine Liste von Kindknoten des Typs *EvalTreeNode*. Der Großteil der zu verarbeitenden Daten besteht aus den *mapping* Elementen, dies werden in dem entsprechenden *EvalTreeNode* Objekt in einem *Set* gespeichert. Die *mapping* Elemente werden in Java als *Map* abgebildet. Die Variablennamen bilden die Schlüssel und die konkreten Ausprägungen der SPARQL-Variablen bilden die Werte der *Map*. Diese *Map* Objekte werden im Folgenden als Mapping bezeichnet. In Abbildung 3.2 sind die Klassen als Klassendiagramm dargestellt.

Vereinigung Die zweite Phase des Algorithmus beschäftigt sich mit der eigentlichen Vereinigung des Wurzelknotens und aller Kindknoten. Der Vorgang beginnt bei dem Wurzelknoten, es wird hierbei ein neues *Set* aus Mapping Objekten angelegt, dieses wird im Folgenden als *results* bezeichnet. Als Initialwerte wird *results* mit den Mappings des Wurzelknoten befüllt. Nun werden diese Mappings um passende Mappings aller Kindknoten erweitert. Für jeden Kindknoten werden folgende Schritte durchgeführt:

Kompatibilität Für jedes Mapping in *results* wird für jedes Mapping im Kindknoten überprüft, ob dieses Mappings mit einander kompatibel sind. Zwei Mappings sind mit einander kompatibel, wenn bei dem paarweisen Vergleich alle übereinstimmenden Variablen auch den selben Wert aufweisen. Der Pseudo-Code für die Kompatibilität zweier Mappings ist in Algorithmus 3.1 zu finden.
In Listing 3.4 sind zwei Beispiele zu finden. *m1* und *m2* sind kompatible Mappings, da die gemeinsamen Variablen (*?X*, *?Y*) auch den gleichen Wert aufweisen. Hingegen sind *m3* und *m4* nicht kompatibel, da die Variable *?Z* unterschiedliche Werte aufweist.

Wenn das Mapping aus *results* mit dem Mapping aus dem Kindknoten kompatibel ist, wir das Mapping aus *results* um die neuen Variablen und deren Werten erweitert (siehe Algorithmus 3.2). Nach dem jedes Mapping aus *results* mit allen Mappings des Kindknoten abgeglichen wurde, ist dieser Kindknoten fertig abgearbeitet. Danach durchlaufen die Kindknoten des jetzigen Knoten die selbe Prozedur. Bei jedem Kindknoten wird *results* um einige Werte erweitert. Diese Phase ist beendet, wenn alle Knoten im Baum abgearbeitet sind. Nun enthält *results* alle erweiterten Mappings.

Algorithm 3.1: Kompatibilität von zwei Mappings

input : Mapping $m1$ und Mapping $m2$
output: true wenn $m1$ mit $m2$ kompatibel sind, sonst false

```
1  keys1 ← Menge der Schlüssel von m1;
2  keys2 ← Menge der Schlüssel von m2;
3  interKeys ← m1 ∩ m2;
4  foreach Schlüssel k aus interKeys do
5  |  v1 ← m1[k];
6  |  v2 ← m2[k];
7  |  if v1 ≠ v2 then
8  |  |  return false;
9  |  end
10 end
11 return true;
```

Listing 3.3: Beispiele für Kompatibilität von zwei Mappings

```
// Kompatibel
m1 = {
        (?X, "x"), (?Y, "y"), (?Z, "z")
}

m2 = {
        (?X, "x"), (?Y, "y"), (?A, "a")
}

// Nicht Kompatibel
m3 = {
        (?X, "x"), (?Y, "y"), (?Z, "z")
}

m4 = {
        (?X, "x"), (?Y, "y"), (?Z, "a")
}
```

Algorithm 3.2: Erweiterung der MappingSets

input: Set *results* aus Mappings, Set *M* aus Mappings

```
1  MappingSet toAdd ← leeres Set;
2  foreach Mapping m1 aus results do
3      added ← false;
4      foreach Mapping m2 aus M do
5          if m1 kompatibel mit m2 then
6              Mapping extendedMapping ← Erweitere m1 mit m2;
7              Füge extendedMapping zu toAdd hinzu;
8              added ← true;
9          end
10     end
11     if added then
12         Entferne m1 aus results da m1 mindestens einmal erweitert wurde und am
           Ende hinzugefügt wird;
13     end
14 end
15 Füge alle Mappings aus toAdd zu results hinzu;
```

Listing 3.4: Beispiele für Erweiterung der MappingSets

```
MappingSet results = {
        { (?X, "x1"), (?Y, "y1") },
        { (?X, "x2"), (?Y, "y2") }
}

MappingSet M = {
        { (?X, "x1"), (?A, "b1") },
        { (?X, "x2"), (?B, "b2") },
        { (?X, "x2"), (?C, "c1") }
}

//Erweiterung von results mit M
results = {
        { (?X, "x1"), (?Y, "y1"), (?A, "b1") },
        { (?X, "x2"), (?Y, "y2"), (?B, "b2") },
        { (?X, "x2"), (?Y, "y2"), (?C, "c1") }
}
```

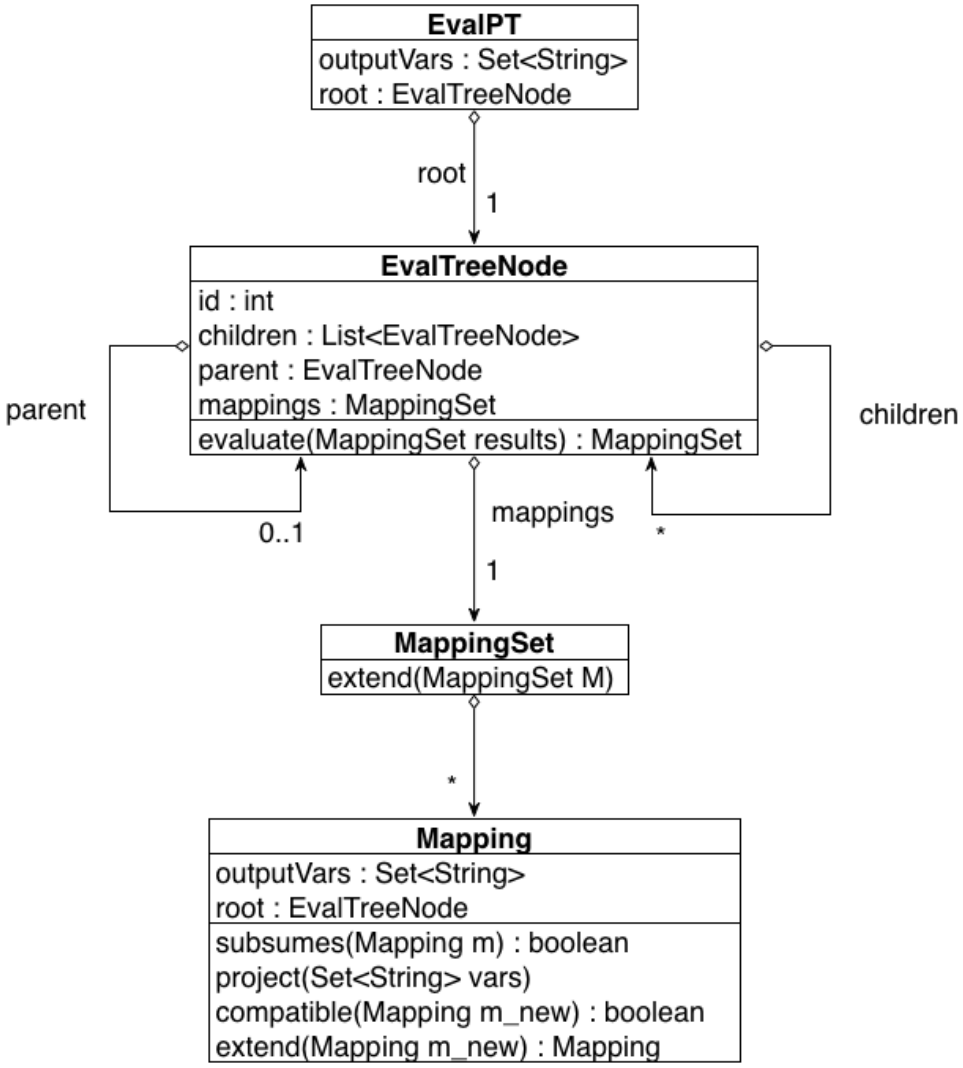

Abbildung 3.2: Klassendiagramm

In der dritten Phase findet die Projektion der Daten statt. Hierbei wird über alle Mappings iteriert und es werden alle Variablen entfernt, welche nicht in der Projektion (*ovar* Elemente) enthalten sind.

Maximierung Die vierte und letzte Phase beschäftigt sich mit der Maximierung der Mappings ([AFP+15] unter CERTAIN ANSWER SEMANTICS) aus *results*. Alle Elemente aus *results* werden in eine neue Menge *maxResults* überführt. Zu Beginn ist die Menge *maxResults* leer. Die Elemente aus *results* werden einzeln in *maxResults* eingefügt. Um ein Element zur Menge hinzuzufügen wird Algorithmus 3.3 angewandt.

Listing 3.5: Beispiel für Subsumierung

```
Mapping m1 = { ("?X","x"), ("?Y","y"), ("?Z","z"), ("?A","a") }

Mapping m2 = { ("?X","x"), ("?Y","y"), ("?B","b") }
//m1 subsumiert NICHT m2

Mapping m3 = { ("?X","x"), ("?Y","y"), ("?Z","b") }
//m1 subsumiert NICHT m3

Mapping m4 = { ("?X","x"), ("?Y","y"), ("?Z","z") }
//m1 subsumiert m4
```

Der Algorithmus verwendet die Funktion *subsums*. Diese erhält zwei Eingabeparameter Mapping *m1* und Mapping *m2*. Sie überprüft wie folgt, ob *m1 m2* subsumiert. Ist die Anzahl der Elemente von *m1* kleiner als die von *m2*, subsumiert *m1* nicht *m2*. Ist das nicht der Fall, wird die Schlüssel der Mappings verglichen. Wenn *keys1* die Schlüssel von *m1* sind und *keys2* die Schlüssel von *m2* sind, dann subsumiert *m1 m2* nicht, wenn *keys1* nicht alle Schlüssel aus *keys2* beinhaltet. Ist das nicht der Fall wird über alle Schlüssel aus *keys2* iteriert. Wenn *k* ein Schlüssel aus *keys2* ist, müssen die Werte der Mappings für *k* für alle Elemente aus *keys2* gleich sein. Trifft das zu, dann subsumiert *m1 m2*. Unter Listing 3.5 ist ein Beispiel zu finden.

Zu Beginn des eigentlichen Algorithmus wird überprüft ob das einzufügende Element *newM* schon in dem Set *maxResults* vorhanden ist, wenn das der Fall ist, wird das Element nicht erneut eingefügt und der Algorithmus ist beendet. Danach wird über das Set *maxResults* iteriert. Wenn *oldM* ein Mapping aus dem Set *maxResults* ist, wird überprüft ob *oldM newM* subsumiert. Ist das der Fall, wird *newM* nicht in das Set eingefügt und der Algorithmus ist beendet. Wenn *newM oldM* subsumiert wird *oldM* aus *maxResults* entfernt. Subsumiert keines der Mappings aus *maxResults newM*, wird *newM* zu *maxResults* hinzugefügt.

Dieser prozedurale Algorithmus ist in [AFK+16] unter Top-Down Evaluation zu finden.

Algorithm 3.3: Hinzufügen zu MaxSet

input: Set *maxResults*, Mapping *newM*

```
1  isSubsumed ← false;
2  if maxResults enthält newM then
3  |  return;
4  end
5  foreach Mapping oldM aus maxResults do
6  |  if subsums(oldM, newM) then
7  |  |  isSubsumed ← true;
8  |  |  break;
9  |  end
10 |  if subsums(newM, oldM) then
11 |  |  Entferne oldM aus maxResults;
12 |  end
13 end
14 if !isSubsumed then
15 |  Füge newM in maxResults ein;
16 end
17 function Boolean subsums(Mapping m1, Mapping m2)
18 |  if Anzahl der Elemente von m1 ≥ Anzahl der Elemente von m2 then
19 |  |  keys1 ← Schlüssel von m1;
20 |  |  keys2 ← Schlüssel von m2;
21 |  |  if keys1 beinhaltet alle Elemente aus keys2 then
22 |  |  |  foreach Schlüssel k aus keys2 do
23 |  |  |  |  v1 ← m1[k];
24 |  |  |  |  v2 ← m2[k];
25 |  |  |  |  if v1 ≠ v2 AND v2 ≠ null then
26 |  |  |  |  |  return false;
27 |  |  |  |  end
28 |  |  |  end
29 |  |  else
30 |  |  |  return false;
31 |  |  end
32 |  else
33 |  |  return false;
34 |  end
35 |  return true;
36 end
```

3.3 Benchmarking

Die Zeitmessung des aktuellen Systems beruht auf mehreren Etappen, in welchen über die Java Funktion *System.nanoTime()* die vergangene Zeit seit der letzen Etappe gemessen wird. Folgende Etappen wurden gemessen:

1. Read - Die Zeit für das Einlesen der Datei
2. Evaluation - Die Auswertung der Daten ohne Maximierung des Ergebnisses
3. MaxSet - Die Maximierung des Ergebnisses

Es wurden jeweils drei Durchläufe erfasst. Am Ende wurden die Ergebnisse der Etappen aufsummiert. Weiters wurden die durchschnittlichen Zeiten der Durchläufe pro Etappe errechnet. Die Rohdaten sind im Kapitel 6 zu finden.

Listing 3.6: Benchmark ITERATIVE, Datei: lubm-ex-20-15.sparql.xml

Einlesen	Evaluierung	Maximierung	Summe
0.185869	7.136303	7.037891	14.360063

Listing 3.7: Benchmark ITERATIVE, Datei: lubm-ex-20-17.sparql.xml

Einlesen	Evaluierung	Maximierung	Summe
1.022490	484.785088	0.190920	485.998499

Diese Dateien wurden zum Test herangezogen:

- lubm-ex-20-15.sparql.xml
  - Dateigröße: 2,44 MB
  - Zeilenanzahl: 40.827
  - Auswertungsergebnis:
    - Zusammenfassung in Listing 3.6
    - Rohdaten in Listing 6.1
- lubm-ex-20-17.sparql.xml
  - Dateigröße: 24,95 MB
  - Zeilenanzahl: 378.450
  - Auswertungsergebnis:
    - Zusammenfassung in Listing 3.7

* Rohdaten in Listing 6.2

Es fällt auf, dass die Phase Evaluierung den größten Optimierungsbedarf aufweist. Bei der kleineren Datei lubm-ex-20-15.sparql.xml benötigt die Evaluierung nur knapp länger als die Maximierung. Bei der größeren Datei lubm-ex-20-17.sparql.xml ist die Maximierung zu vernachlässigen, da diese Phase nur ca. 0,04% der Gesamtdauer in Anspruch nimmt. Weiters fällt auf, dass bei steigender Dateigröße die Dauer der Phase Einlesen nicht so stark wächst wie die Phase Evaluierung.

Aus *LUBM: A benchmark for OWL knowledge base systems* [GPH05] und *Towards Reconciling SPARQL and Certain Answers — Extended Version.* [AFP+16] wurden die oben angeführten Dateien generiert. Alle Zeitangaben sind in Sekunden angegeben.v

KAPITEL 4

In-Memory Datenbanken

Die folgenden Abschnitte befassen sich mit neuen Anforderungen und Konzepten im Bereich In-Memory Datenbanken. Verwendete Literatur: [Pla13]

4.1 Neue Anforderungen an Datenbanksysteme

Wenn man heutzutage ein von Grund auf neues Datenbanksystem entwickeln würde, hätte dieses andere Anforderungen als konventionelle Datenbanksysteme. Moderne Unternehmen arbeiten täglich mit einer Fülle von Daten, was vor ein paar Jahren noch undenkbar war. Fertigungsunternehmen sind ein Beispiel, für solche datenlastige Betriebe. Während des Fertigungsprozesses werden enorme Mengen an Daten produziert. Diese müssen oft in Echtzeit verarbeitet werden und liefern Grundlagen für weitere Entscheidungsprozesse. Dies ist nur ein Beispiel der steigenden Bedürfnisse moderner Datenbanksysteme.

Es gibt zwei wesentliche Anforderungen für moderne Datenbanksysteme. Daten aus unterschiedlichen Eingabequellen müssen in einem Datenbanksystem aufgenommen werden und diese Daten müssen in Echtzeit verarbeitet werden, um abgestimmte Entscheidungen treffen zu können. Die weiteren Abschnitte beschreiben verschiedene Einsatzgebiete, in denen diese Anforderungen zu tragen kommen.

Ereignisdaten

Ereignisdaten sind der Kern zukünftiger Produktionsanlagen, diese sind wie folgt charakterisiert. Die Datenmenge eines einzelnen Elements ist sehr klein, nur ein paar Bytes oder Kilobytes groß. Im Gegenzug ist die Anzahl der Ereignisse für eine gewisse Einheit im Verhältnis enorm hoch.

In der Produktion von alltäglichen Produkten kommt eine Vielzahl von Sensoren zum Einsatz. Besonders in den Bereichen heikler Produkte müssen alle Schritte der Produk-

tion und Auslieferung dokumentiert werden. Um Produkte während dieser Prozesse genau beobachten zu können, werden RFID (Radio Frequency Identification) Tags oder zweidimensionale Strichcodes eingesetzt. Bei jedem Schritt der Produktion bzw. der Auslieferung kommen Sensoren zum Einsatz, um den Weg des Produktes zu identifizieren.

BigPoint ist ein Spieleentwickler mit dem Sitz in Deutschland. In der Spieleindustrie sind Ereignisdaten von zentraler Bedeutung. BigPoint verwendet diese Daten, um in Echtzeit den Spielern in schwierigen Situationen zahlungspflichtige Hilfen anzubieten. Diese Spiele produzieren 10.000 Ereignisse pro Sekunde. Herkömmliche Datenbanken sind für diesen Anwendungszweck nur begrenzt einsetzbar. Flexible und auf den Anwendungsfall zugeschnittene Abfragen von Entwicklern können nicht interaktiv beantwortet werden. In-Memory Datenbanken werden eingesetzt, um in kürzester Zeit Entscheidungen über den Einsatz von zahlungspflichtigen Hilfen zu treffen. Es werden verschiedene Gruppen von Spielern analysiert und es wird entschieden ob alle Spieler diese Angebote bekommen.

Strukturierte und unstrukturierte Daten

Man muss zwischen strukturierten und unstrukturierten Daten unterscheiden. Strukturierte Daten weisen ein Schema auf, auf Basis dessen man die Daten analysieren kann. Daten welche in relationalen Datenbanken gespeichert werden, sind zum Beispiel strukturierte Daten. Im Gegensatz dazu weisen unstrukturierte Daten kein Schema auf, somit kann nicht so einfach eine Analyse der Daten vorgenommen werden, z.B.: Bilder, Videos, freie Texte, etc. Im Laufe der Zeit sind in vielen Unternehmen eine große Menge von unstrukturierten Daten wie Berichte, Tabellen oder Textdokumente angefallen. In diesen unstrukturierten Daten verbirgt sich eine Menge an Informationen. Es besteht eine enorme Nachfrage, in diesen Dokumenten schnell und flexibel nach Informationen zu suchen.

In Spitälern sammeln sich große Mengen von strukturierten und unstrukturierten Patientendaten an. In-Memory Datenbanken ermöglichen die Kombination von strukturierten und unstrukturierten Daten, sowie die Einbindung von externen Daten z.B. klinischen Studien oder Nebenwirkungen. Somit können Ärzte in Echtzeit, die benötigten Daten interaktiv kombinieren und schneller Diagnosen stellen. Dieser Prozess verringert die manuelle und zeitaufwendige Arbeit der Ärzte enorm.

Auch bei Wartungsarbeiten von Flugzeugen werden In-Memory Datenbanken eingesetzt. Bei der Bearbeitung von Wartungsaufträgen fallen strukturierte wie auch unstrukturierte Daten an. Mit Hilfe der In-Memory Technologie können Korrelationen zwischen Schwachstellen erkannt werden und dies wiederum kann das Risiko eines Fehlers verringern.

In Spitälern wie auch in der Flugzeug Industrie spielt die Zeit und somit die Dauer einer Datenbankabfrage, vor allem in Notfällen, eine wichtige Rolle. Durch In-Memory Daten-

banken können auch zeitkritische Abfragen oder Operationen auf mobilen Endgeräten durchgeführt werden.

4.2 Konzepte

Die folgenden Abschnitte beschäftigen sich mit den verwendeten Konzepten in einer In-Memory Datenbank, am Beispiel der SanssouciDB. Die SanssouciDB ist ein modellhaftes Datenbanksystem, welches auf Prototypen des Hasso-Plattner-Institut und auf eine existierende SAP Datenbank aufbaut. SanssouciDB ist eine SQL Datenbank mit ähnlichen Komponenten wie eine gewöhnliche Datenbank.

4.2.1 Verwendung des Hauptspeichers

Der große Unterschied zwischen In-Memory Datenbanken und gewöhnlichen Datenbanken ist das verwendete Speichermedium. In-Memory Datenbanken halten die Daten permanent im Hauptspeicher des Systems. Jedoch nicht alle Funktionen können über den Hauptspeicher abgewickelt werden. Logging und Recovery müssen dennoch auf einen nicht flüchtigen Speicher zurückgreifen. Doch alle gängigen Operatoren wie *join*, *find* oder *aggregation* können mit den Daten im Hauptspeicher arbeiten. Bei der Implementierung dieser Operatoren muss keine Rücksicht auf die langen Zugriffszeiten einer Festplatte genommen werden. Die Verwendung des Hauptspeichers führt auch zu einer angepassten Organisation der Daten sowie zu einem optimierten Umgang mit den Daten im Speicher.
Ein zusätzlicher Vorteil des Hauptspeichers sind die konstanten Zugriffszeiten. Die Laufzeit einer Datenverarbeitung im Hauptspeicher kann berechnet werden. Diese Eigenschaft begünstigt die Implementierung. Bei Festplatten kann die Zugriffszeit kaum oder gar nicht berechnet werden, sie hängt von den mechanischen Bauteilen ab.

4.2.2 Spaltenorientierung

In herkömmlichen Datenbanken werden die Daten zeilenweise abgespeichert, in Gegensatz zu SanssouciDB. Hier werden die Daten spaltenweise abgelegt. Bei Spaltenorientierung werden alle Werte einer Spalte in benachbarten Blöcken gespeichert. Wenn die Daten zeilenorientiert abgespeichert werden, werden ganze Zeilen in benachbarten Blöcken abgelegt. Spaltenorientierung eignet sich also zum Lesen von einzelnen Spalten. Vorteilhaft sind hierbei Spaltenaggregation oder Spalten-Scans.

Viele Datenbanksysteme verwenden spaltenorientierte Speicherung, um diese Vorteile zu nutzen. Bei einem Großteil der SQL Abfragen, werden alle Spalten abgefragt, aber es wird nur ein kleiner Teil der Spalten verwendet. Um den Vorteil der spaltenorientierten Speicherung optimal zu nutzen, sollten *SELECT* * Abfragen vermieden werden. Eine Analyse von Unternehmensapplikationen hat gezeigt, das fast nie alle Spalten einer Tabelle im Endeffekt verwendet werden. Ein Beispiel dieser Analyse zeigte, dass bei

einer konkreten Tabelle mit 300 Spalten, nur 17 Spalten benötigt wurden. Hier kann bei der Verwendung von Spaltenorientierung ein bedeutender Vorteil erreicht werden.

Ein weiterer Vorteil entsteht bei der Verwendung von Indizes. Da die Daten spaltenweise angeordnet sind, können die einzelnen Spalten als Indizes verwendet werden. Nachdem alle Daten im Hauptspeicher liegen und die Daten einer Spalten nacheinander angeordnet sind, ist die Leistung eines Scan-Vorgangs meist ausreichend. Es können dennoch bestimmte Indizes angelegt werden.

Jedoch gibt es bei spaltenorientierter Speicherung auch Nachteile. Das Einfügen von großen Datenmengen ist komplizierter, deswegen werden Differentialspeicher verwendet. Wenn neue Daten eingefügt werden, werden diese zuerst in den Differentialspeicher eingefügt. Nach einem gewissen Schwellenwert, z.B. Anzahl an Neudaten, werden diese Daten in die eigentliche Datenbank überführt.

4.2.3 Aktive und passive Daten

In SanssouciDB wird zwischen aktiven und passiven Daten unterschieden. Aktive Daten sind Datenbestände, welche noch im Arbeitsspeicher verarbeitet werden. Passive Daten werden nicht mehr verarbeitet, diese Datenbestände werden auf langsamere Speichermedien ausgelagert. Diese Vorgehensweise entlastet den Hauptspeicher, da ein Teil der Daten ausgelagert wird. Wenn neue Daten eingefügt oder bestehende Daten geändert werden müssen Log-Dateien erstellt werden. Diese können aber nicht im Hauptspeicher abgelegt werden, sondern müssen auf ein nicht-flüchtiges Speichermedium (Festplatte) gespeichert werden.

4.3 Konkrete In-Memory Datenbanken

Zur Umsetzung wurden drei In-Memory Datenbanken ausgewählt, diese werden im Folgenden näher beschrieben.

4.3.1 H2 Database Engine

Die H2 Database Engine (H2) ist ein relationales Datenbankmanagementsystem, welches in Java geschrieben wurde. Der Programmcode ist Open Source und somit öffentlich zugänglich. H2 unterstützt Standard SQL und als Schnittstelle die JDBC API. Weiters kann der PostgreSQL ODBC Treiber verwendet werden.

ACID

Die ACID-Eigenschaften finden in H2 wie folgt Anwendung:

- Atomicity
  - Transaktionen in der H2 Datenbank sind immer atomar.

- Consistency
  - Die Datenbank ist standardmäßig in einem konsistenten Zustand. Auch referentielle Integrität ist gewährleistet, außer sie wird explizit ausgeschalten.
- Isolation
  - Standardmäßig ist in H2 das Isolation Level auf *read commited* gesetzt. Hierbei sind die Transaktionen nicht total isoliert. Diese Einstellung führt aber im Normalfall zur besserer Leistung.
- Durability
  - Die Eigenschaft der Dauerhaftigkeit ist im Bereich der In-Memory Datenbanken ein grundsätzliches Problem, da der verwendetet Speicher, der Hauptspeicher, ohne Stromversorgung die Daten nicht halten kann. H2 garantiert also nicht, dass alle Transaktionen einen Stromausfall überstehen. Wenn die Dauerhaftigkeit der Daten hohe Priorität hat, kann der von H2 zur Verfügung gestellte *clustering mode* verwendet werden.

Problem bei der Dauerhaftigkeit

Dauerhaftigkeit in einer In-Memory Datenbank wie H2 zu erreichen, gestaltet sich schwierig. Das Limit wird hier durch die Schreibgeschwindigkeit der Festplatte festgelegt. Die Log-Dateien werden auf die Festplatte geschrieben, um im Fehlerfall einen konsistenten Zustand wieder herstellen zu können. Um das Durchschreiben auf die Festplatte bei jedem Eintrag zu erreichen, verwendet H2 *synchronous write*. Hierbei benutzt H2 in Java *RandomAccessFile*, diese Klasse unterstützt zwei Modelle *rwd* und *rws*. Bei *rwd* wird jede Aktualisierung der Datei synchron auf die Festplatte geschrieben. Zusätzlich wird bei *rws* bei jeder Änderung der Metadaten synchron auf die Festplatte geschrieben.

H2 hat einen Test *org.h2.test.poweroff.TestWrite* mit diesem Model implementiert. Dabei werden 50.000 Schreiboperationen erreicht. Auch wenn die Buffer des Betriebssystems deaktiviert werden, können nicht alle Daten synchron auf die Festplatte geschrieben werden. Eine herkömmliche Festplatte läuft mit 7200 RPM, die Umdrehungszahl reicht dafür nicht aus. Der Buffer kann in Java manuell über *FileDescriptor.sync()* und *FileChannel.force()* geleert werden. Doch ein Aufruf dieser Funktionen stellt nicht sicher, dass die Daten auch wirklich auf der Festplatte sind.

Abgesehen davon, dass man nicht garantieren kann, dass bei jedem Schreibvorgang die Daten auf der Festplatte persistiert werden, sollte man den Buffer nicht manuell leeren. Das manuelle Leeren des Buffer ist also schwer und verschlechtert die Leistung dramatisch. Um die Verzögerung des Schreibvorgangs auszugleichen, kann in H2 die Funktion *SET WRITE DELAY* verwendet werden.

Verbindungsmöglichkeiten

H2 unterstützt drei verschiedene Verbindungsmöglichkeiten.

Im eingebetteten Modus (embedded mode) wird die Datenbank in der selben JVM wie die Applikation geöffnet. Dies hat den Vorteil, einer hohen Zugriffsgeschwindigkeit und ist auch die einfachste Variante. Es gibt keine Einschränkung in der Anzahl der offenen Datenbanken bzw. offenen Verbindungen.

Weiters stellt H2 einen Server Modus zur Verfügung. Dabei wird ein Server gestartet auf welchen intern eine Datenbank im eingebetteten Modus geöffnet wird. Um auf die Datenbank zu zugreifen, muss man sich mit dem Server verbinden. Die Client Applikation kann über JDBC oder ODBC API mit der Datenbank kommunizieren. Anders wie beim eingebetteten Modus können beim Server Modus mehrere Client Applikationen gleichzeitig Daten lesen und manipulieren. Der Nachteil dabei ist, dass alle Daten über TCP/IP transportiert werden müssen. Dieser zusätzliche Schritt hemmt die Geschwindigkeit des Systems. Wie auch bei dem eingebetteten Modus gibt es keine Einschränkungen in der Anzahl der offenen Datenbanken bzw. offenen Verbindungen.

Der dritte Modus stellt eine Mischung aus dem eingebetteten Modus und dem Server Modus dar. Zuerst wird eine Applikation gestartet, welche sich zur Datenbank im eingebetteten Modus verbindet. Dieselbe Applikation startet einen Server, damit sich auch andere Applikationen zur Datenbank verbinden können. Die Latenz der lokalen Applikation ist sehr niedrig, da diese den eingebetteten Modus verwendet. Applikationen die den Server der Datenbank verwenden, haben eine etwas langsamere Verbindung. Über die Server API kann die lokale Applikation den Server starten bzw. stoppen.

[Eng16]

4.3.2 HyperSQL DataBase

HSQLDB (HyperSQL DataBase) ist ein modernes relationales Datenbankmanagementsystem. Es verhält sich beinahe wie der SQL:2011 und der JDBC 4 Standard. HSQLDB unterstützt die Kernfunktionalität des SQL:2008 Standards und viele optionale Funktionen. Seit 2001 gibt es Version 1, erst 2010 wurde Version 2 veröffentlicht. In dieser Version wurde der Kern der Datenbankfunktionalität neu geschrieben und überarbeitet.

Die Mechanismen zur Persistierung der Daten werden seit der Version 1.8 verwendet. Persistenz in einer In-Memory Datenbank wie HSQLDB beruht auf mehreren Faktoren, der Hardware, dem Betriebssystem und der JVM. In jeden Bereich kann es zu Fehlern oder Ausfällen kommen, daher empfiehlt HSQLDB regelmäßig Sicherheitskopien zu erstellen. Hierfür stellt HSQLDB viele eingebaute Funktionen zur Verfügung.

Eine HSQL Datenbank wird Katalog genannt, dabei wird zwischen drei Typen von Katalogen unterschieden:

- mem - Hierbei werden die Daten im RAM gehalten.

- file - Hierbei werden die Daten im Dateisystem gespeichert.
- res - Hierbei werden die Daten in einer Java Resource (z.B. Jar) abgelegt. Dabei kann nur lesend zugegriffen werden.

ACID

Die ACID-Eigenschaften finden in HSQLDB wie folgt Anwendung:

- Atomicity
  - Operationen sind in HSQLDB atomar. Dies wird auch bei einem Systemabsturz sicher gestellt.
- Consistency
  - HSQLDB setzt zu jeder Zeit implizite als auch explizite Beschränkungen durch.
- Isolation
  - Die Regeln des Datenbank Isolationsmodell werden von HSQLDB umgesetzt.
- Durability
  - Dauerhaftigkeit wird über *WRITE DELAY MILLIS* in HSQLDB umgesetzt. Diese Verzögerung gibt an, mit welcher Verzögerung auf das Speichermedium geschrieben wird. Tritt ein Systemabsturz genau während dieser Verzögerung auf, kann es zu einem inkonsistenten Zustand der Daten kommen.

Verbindungsmöglichkeiten

Es gibt zwei grundsätzliche Verbindungsarten. Die *in-process* Verbindung erlaubt der Applikation direkt auf die Datenbank zuzugreifen. Bei dieser Variante müssen die Daten keinen Umweg über das Netzwerk machen, anders wie bei den Server Modi. Bei diesen Verbindungsmöglichkeiten findet die Kommunikation nicht Prozessintern statt, sondern wird über eine Client-Server Architektur geregelt. Der größte Nachteil dieser Methode ist der zusätzliche Zeitaufwand, welcher bei der Kommunikation über das Netzwerk entsteht.

Wenn eine Datenbank im Server Modus gestartet wird, wird intern ein *in-process* Katalog gestartet. Der Server wartet während des Betriebes auf eingehende Verbindungen. Da die Verbindung über das Netzwerk erfolgt, kann der Zugriff auch von einem anderen Computer aus erfolgen, nicht so bei der reinen *in-process* Verbindung. Es wird von HSQLDB empfohlen einen Server Modus während der Entwicklungsphase zu verwenden, um die Daten von einer separaten Datenbank abzurufen.

HyperSQL HSQL Server ist der bevorzugte Server Modus, da ein proprietäres Protokoll für die Kommunikation verwendet wird.

HyperSQL HTTP Server ist die zweite Möglichkeit die Datenbank im Server Modus zu starten. Dieser Modus ist für Situationen geeignet, in denen man nur über HTTP kommunizieren kann. Dieser spezielle Web Server erlaubt den JDBC Clients über HTTP zu kommunizieren.

[Gro15]

4.3.3 Apache Derby

Apache Derby ist eine relationale Datenbank und ein Unterprojekt der Apache DB. Die gesamte Datenbank ist in Java realisiert und als Open Source Projekt verfügbar. Apache Derby bietet wie auch andere Anbieter einen eingebetteten Modus und einen Server Modus an, um sich zur Datenbank zu verbinden. In der Standardeinstellung kann kein separater Datenbank Server installiert oder gewartet werden. Das Datenformat, welches von Apache Derby verwendet wird, ist plattformunabhängig. So können diverse Datenbestände in Apache Derby ohne Rücksichtnahme auf das Zielgerät verschoben werden. Apache Derby ist ACID kompatibel und unterstützt referentielle Integrität.

Verbindungsmöglichkeiten

Wie auch andere In-Memory Datenbanken bietet Apache Derby verschiedene Verbindungsmöglichkeiten an. Unter Apache Derby werden diese Verbindungsmöglichkeiten Entwicklungsoptionen genannt. Von diesen Entwicklungsoptionen stellt Apache Derby zwei zur Verfügung.

In der eingebetteten Entwicklungsoption wird die Datenbank in der selben JVM wie das eigentliche Java Programm gestartet. Diese Option ist für den Endbenutzer fast nicht zu erkennen, da sich die Datenbank mit dem Java Programm automatisch startet und auch wieder beendet. Hierbei hat nur ein Benutzer Zugriff auf die Datenbank.

Die Client/Server Entwicklungsoption baut auf einen Datenbankserver auf, welcher über das Netzwerk erreichbar ist. Bei dieser Variante können sich mehrere Benutzer mit der Datenbank verbinden. Auf dem Server läuft Apache Derby in einer eigenen JVM und Programme, welche sich zu dem Datenbankserver verbinden, können mit verschiednen JVMs ausgeführt werden.

[Apa15]

KAPITEL 5

Umsetzung

5.1 Konzept

Der oben beschriebene iterative Algorithmus (siehe Kapitel 3.2) soll durch eine performantere Vorgehensweise abgelöst werden. Kern des neuen Systems ist eine relationale Datenbank, im speziellen eine In-Memory Datenbank. Diese wird verwendet, um die Zuordnung der Mappings aus den OPTIONAL-Knoten zu realisieren. Diese Zuordnung der Mappings entspricht einem LEFT OUTER JOIN und kann somit in einer relationalen Datenbank umgesetzt werden.

Um im Folgenden diesen LEFT OUTER JOIN durchzuführen, muss davor eine geeignete Datenbank angelegt werden. Nachdem die Daten eingelesen wurden, muss aus der Struktur der Daten ein Datenbankschema abgeleitet werden. Für jeden Durchlauf bzw. für jede Datei muss deshalb individuell ein Datenbankschema erstellt werden. Das Datenbankschema umfasst hierbei das Erstellen geeigneter Tabellen und das Anlegen von Indizes. Pro Knoten der Eingabe muss eine Tabelle angelegt werden. Dabei werden keine Primärschlüssel oder Fremdschlüssel erstellt. Die Variablen aus den *nodeVar* Elementen bilden die Spalten der Tabelle eines Knoten. Um den späteren LEFT OUTER JOIN effizienter zu gestallten, werden für die Spalten der JOIN Bedingung Indizes erstellt.

Während der Erstellung des Datenbankschemas wird parallel dazu der LEFT OUTER JOIN vorbereitet. Bei der Generierung der Indizes müssen bereits die Spalten der JOIN Bedingung bekannt sein. Deswegen kann an dieser Stelle gleichzeitig der LEFT OUTER JOIN vorbereitet werden.

Nachdem die Daten eingelesen wurden und das passenden Datenbankschema bestimmt und angelegt wurde, können die eingelesenen Daten in die zuvor erstellten Tabellen eingefügt werden.

Anknüpfend an das erfolgreiche Einfügen der Daten wird der zuvor generierte LEFT OUTER JOIN ausgeführt und die Auswertung zurück in die oben beschriebene Datenstruktur überführt.

Das System beinhaltet drei verschiedene In-Memory Datenbanken, beim Start des Programms kann ausgewählt werden, welche dieser drei In-Memory Datenbanken verwendet werden soll.

5.2 Implementierung

Die Umsetzung besteht aus mehreren Komponenten, welche in diesem Abschnitt genauer beschrieben und erläutert werden. Das neue System baut auf der zuvor beschriebenen Datenstruktur auf (siehe Kapitel 3.1).

Der Einstiegspunkt ist die Klasse *PTEvaluator*. Das Programm bietet mehrere Startoptionen, in der Klasse *PTEvaluator* werden die Startoptionen aufgesetzt und nach dem Start wird entschieden welcher Programmzweig ausgeführt wird. Hierfür wurde *Apache Commons CLI* (`https://commons.apache.org/proper/commons-cli/`) verwendet. Diese Startoptionen stehen zur Verfügung:

- -db / --database <*arg*>
  - Mit dieser Option wir nicht das bisherige System gestartet, sondern es kommt eine In-Memory Datenbank zum Einsatz. Es kann eine dieser drei Datenbanken als <*arg*> angegeben werden: H2, HSQLDB, DERBY
- -i / --input <*arg*>
  - Nach dieser Option kann als <*arg*> ein Dateipfad zu einer geeigneten (siehe Kapitel 3.1) XML Datei angegeben werden. Diese Datei wird als Eingabedatei verwendet. Wenn diese Option nicht angegeben wird, wird standardmäßig 'resources/test.xml' verwendet.
- -o / --output <*arg*>
  - Nach dieser Option kann als <*arg*> ein Dateipfad als Ausgabedatei angegeben werden. Standardmäßig wird bei dem iterativen Algorithmus 'output/-test.txt' verwendet und bei dem Datenbank Algorithmen 'output/test-db.txt'.
- -r / --runs <*arg*>
  - Bei jeder Variante kann mit dieser Option als <*arg*> die Anzahl der Durchläufe angegeben werden. Bei der Ausgabe wird der Durchschnitt der ermittelten Zeiten berechnet.
- -ni / --noIndices

  - Wenn die Option *--database* oder *--benchmark* verwendet wird, kann mit dieser Option die Verwendung von Datenbank Indizes deaktiviert werden. Standardmäßig ist die Verwendung von Indizes aktiviert.

- -b / --benchmark
  - Wenn diese Option angegeben wird werden alle Datenbanken nacheinander verwendet, um den direkten Vergleich zu sehen. Bei dieser Option wird keine Ausgabedatei generiert.

- -h / --help
  - Gibt die Hilfe aus.

Nach dem das Programm mit den jeweiligen Optionen gestartet wurde, wird die entsprechende Eingabedatei eingelesen. Dieser Ablauf hat sich zum ursprünglichen Ablauf (siehe Kapitel 3.2) nur wenig verändert. Für jedes Objekt der Klasse *EvalTreeNode* wird eine fortlaufende Nummer vergeben, diese wird später als Teil des Tabellennamens verwendet. Weiters wird für jedes Objekt der Klasse *EvalTreeNode* der Elternknoten gespeichert (außer für den Wurzelknoten). Der Elternknoten wird verwendet, um die Spalten der *JOIN* Bedingung zu identifizieren.

Bei jeder Auswertung eines WDPT werden folgende vier Schritte durchgeführt:

1. Identifizieren der Spalten der *JOIN* Bedingung
2. Anlegen der Tabellen
3. Einfügen der Daten
4. Generieren und Ausführen der *SELECT* Abfrage
5. Zurücksetzen der Datenbank

Bevor diese Schritte ausgeführt werden, muss eine konkrete Datenbankverbindung initialisiert werden. Dies geschieht durch die Klasse *DBConnectionFactory*. Die beim Start ausgewählte Option wird dieser Klasse weitergereicht.
Wird die Option *--benchmark* verwendet, werden die oben angeführten Schritte für jede Datenbank durchgeführt.

Identifizieren der Spalten der *JOIN* Bedingung Nachdem alle Vorbereitungen abgeschlossen sind, beginnt die Evaluation mit der Identifizierung der Spalten für die *JOIN* Bedingung. Beim Einlesen der Daten wurde für jeden Knoten, außer für den Wurzelkonten, der Elternknoten abgespeichert. Wie in Well-Designed Pattern Trees [AFK+16] beschrieben, handelt es sich um zusammenhängende Teilbäume. Aus diesen Grund

können die Spalten für die *JOIN* Bedingung im Elternknoten des jeweiligen Knoten gefunden werden.

Die Suche wird folgendermaßen durchgeführt: Rekursiv werden alle Spalten der zukünftigen Tabellen, welche für den *JOIN* verwendet werden, durchsucht. Für einen Knoten sind jene Spalten relevant, welche auch im Elternknoten verwendet werden.

Für die Knoten aus dem Beispiel 3.2 sind folgende Variablen/Spalten relevant:

- T1: T0.X1
- T2: T1.Z1, T1.Z2
- T3: T0.X2

Bemerkung: Die '?' am Beginn der Variablen wurden entfernt, da der Spaltenname nicht mit diesem Zeichen beginnen darf. Diese Werte werden bei der Erstellung der Indizes und bei der Generierung der *SELECT* Abfrage verwendet.

Anlegen der Tabellen Die Erstellung der Tabellen ist der nächste Schritt. Für jeden Knoten wir eine Tabelle angelegt, dabei wird bei dem Wurzelknoten begonnen. Für unser Beispiel 3.2 würden die *INSERT* Befehle folgendermaßen aussehen:

Listing 5.1: CREATE Befehle

```
CREATE TABLE T0(X1 VARCHAR(200),
                X2 VARCHAR(200));

CREATE TABLE T1(Z1 VARCHAR(200),
                Y1 VARCHAR(200),
                Z2 VARCHAR(200),
                X1 VARCHAR(200));

CREATE TABLE T2(Z1 VARCHAR(200),
                Z2 VARCHAR(200),
                A1 VARCHAR(200));

CREATE TABLE T3(X2 VARCHAR(200),
                A1 VARCHAR(200));
```

Nachdem eine Tabelle erstellt wurde, werden die Indizes für diese Tabelle angelegt. Es wird für jede Spalte aus Kapitel 5.2 der betreffenden Tabelle ein Index angelegt und auch für die Spalten des Elternknotens.

Einfügen der Daten Nachdem Anlegen einer Tabelle und den zugehörigen Indizes wird diese mit den Daten des entsprechenden Knotens befüllt. Um diesen Schritt auch mit einer Vielzahl an Daten effizient zu bewältigen, werden die Daten mittels eines *PreparedStatement* und der Funktion *addBatch()* eingefügt [Ora16]. Nach einer Gruppe von 5.000 Datensätzen wird der Befehl ausgeführt.

Generieren und ausführen der *SELECT* Abfrage Während die Tabellen erstellt werden, wird für jede Tabelle die *SELECT* Abfrage erweitert. Die Projektion besteht aus den Variablen der Elemente *ovar* der XML-Datei. Hierbei muss die zugehörige Tabelle ermittelt werden, da es bei den Spaltennamen zu Überschneidungen kommen kann. Im *FROM* Teil der Abfrage wird der Tabellenname des Wurzelknotens angeführt.
Nachdem eine Tabelle angelegt wurde wird die Abfrage mit einem *LEFT OUTER JOIN* <*Tabellenname*> und einer *ON* Bedingung erweitert. Die Bedingung enthält die Spalten aus Kapitel 5.2. Die gesamte *SELECT* Abfrage ist unter Kapitel 5.2 zu finden.

Listing 5.2: SELECT Abfrage

```
SELECT T1.Z1, T1.Y1, T0.X1
  FROM T0
  LEFT OUTER JOIN T1
       ON T1.X1=T0.X1
  LEFT OUTER JOIN T2
       ON T2.Z1=T1.Z1 AND T2.Z2=T1.Z2
  LEFT OUTER JOIN T3
       ON T3.X2=T0.X2;
```

Nachdem alle Tabellen angelegt und befüllt sind, ist auch die *SELECT* Abfrage fertig generiert und kann ausgeführt werden. Jetzt müssen die Daten noch maximiert werden. Dieser Algorithmus wurde schon bei der iterativen Vorgehensweise verwendet (siehe Kapitel 15 Maximierung). Nach diesem Schritt ist das Ergebnis identisch mit dem aus der iterativen Vorgehensweise (siehe Kapitel 3.2).

5.3 Benchmarking

Die Zeiterfassung dieses Systems beruht auf den selben Eingabedateien, welche in Kapitel 3.3 verwendet wurden. Für jede In-Memory Datenbank wurden drei Durchläufe ausgeführt.
Am Ende wurden die Ergebnisse der Etappen aufsummiert. Weiters wurden die durchschnittlichen Zeiten der Durchläufe pro Etappe errechnet. Die Durchläufe wurden ohne Indizes und mit Indizes durchgeführt. Hierbei sind große Differenzen erkennbar.

Die Tabelle 5.1 enthält die durchschnittlichen Ergebnisse der Zeitmessungen aus Kapitel 3.3 und den Listings 6.3, 6.4, 6.5 und 6.6. Diese Zeitmessungen sind in der Spalte *Laufzeiten* zu finden. In der Spalte *Differenz* wurden der Unterschied (in Sekunden) zu dem

Tabelle 5.1: Zeitersparnis

Datei	Algorithmus	Typ	Laufzeit	Differenz	Zeitersparnis
lubm-ex-20-15	Ohne IMDB		14.360s		
	IMDB ohne Indizes	H2	12.266s	2.094s	14.58%
		HSQLDB	12.237s	2.123s	14.78%
		DERBY	8.986s	5.374s	37.43%
	IMDB mit Indizes	H2	8.782s	5.578s	38.84%
		HSQLDB	8.408s	5.952s	41.45%
		DERBY	8.812s	5.548s	38.63%
lubm-ex-20-17	Ohne IMDB		485.998s		
	IMDB ohne Indizes	H2	154.292s	331.706s	68.25%
		HSQLDB	190.675s	295.323s	60.77%
		DERBY	356.831s	129.168s	26.58%
	IMDB mit Indizes	H2	2.929s	483.070s	99.40%
		HSQLDB	1.617s	484.381s	99.67%
		DERBY	4.474s	481.524s	99.08%

Algorithmus ohne In-Memory Datenbank der entsprechenden Datei berechnet. Weiters wurde in der Spalte *Zeitersparnis* die prozentuelle Zeitersparnis zu dem Algorithmus ohne In-Memory Datenbank der entsprechenden Datei berechnet.
Hier ist auch gut der Unterschied zwischen *IMDB ohne Indizes* und *IMDB mit Indizes* zu erkennen. Vor allem bei der Datei *lubm-ex-20-17* ist ein enormer Unterschied zu erkennen.

Die beiden Zeiten aus den Zeilen *Ohne IMDB* wurden aus dem Kapitel 3.3 zum Vergleich herangezogen. Darunter befinden sich jeweils die Laufzeiten der Implementierung mit IMDB. Bei der kleineren Datei lubm-ex-20-15 hat die Verwendung einer IMDB nur geringe Auswirkungen, vor allem ohne die Zuhilfename von Indizes. Der Einsatz von Indizes ergibt bei jeder verwendeten IMDB eine Laufzeitersparnis.

Bei der größeren Datei lubm-ex-20-17 ist die Laufzeitersparnis auch ohne Indizes beachtlich. Doch das beste Ergebnis konnte mit der IMDB HSQLDB mit Indizes erreicht werden. Die Laufzeit beträgt nur noch 1.617 Sekunden. Ohne IMDB dauerte der Durchlauf im Durchschnitt über 8 Minuten. Das ergibt eine relative Zeitersparnis von über 99%.

KAPITEL 6

Schlussfolgerung

In dieser Arbeit wurde die Evaluierung von WDPTs untersucht. Ausgangspunkt der Arbeit war ein bereits vorhandener Algorithmus, welcher auf einer iterativen Vorgehensweise basiert. Die Eingabe ist eine XML-Datei, welche die Auswertung mehrerer Knoten in einem WDPT darstellt. Für dieses Format wurde ein XML Schema erstellt.

Der iterative Algorithmus wurde beschrieben und analysiert. Im speziellen wurden folgende Teilbereiche betrachtet: Kompatibilität, Maximierung und Subsumierung.

Um einen Ausgangswert zu bestimmen, wurde eine Zeitmessung mit unterschiedlichen Dateien durchgeführt. Danach wurden die Grundkonzepte und Funktionsweisen einer In-Memory Datenbanken beschrieben.

Im Weiteren wurden drei konkrete In-Memory Datenbanken ausgewählt, welche in dem neuen Algorithmus Anwendung finden: H2 Database Engine, HyperSQL DataBase und Apache Derby. Für den neuen Algorithmus wurde ein Konzept erstellt, welches auf relationalen DBMS sowie dem LEFT OUTER JOIN basiert. Dementsprechend beruht der neue Algorithmus auf diesen Kernpunkten: Identifizieren der Spalten der *JOIN* Bedingung, Anlegen der Tabellen und Indizes, Einfügen der Daten, Generieren und Ausführen der *SELECT* Abfrage.

Nach Evaluierung des neuen Systems konnten die Ergebnisse mit den Ausgangswerten verglichen werden. Der im Rahmen dieser Bachelorarbeit implementierte Algorithmus ist durch die Verwendung einer In-Memory Datenbank um bis zu 160-mal schneller als der iterative Algorithmus.

Ausblick Der Teilbereich der Maximierung wurde aus dem iterativen Algorithmus übernommen und nicht verändert. An dieser Stelle besteht noch Optimierungspotential. Bei der kleineren XML-Datei lubm-ex-20-15 macht diese Phase ca. 50% der Gesamtdauer aus. Eine Möglichkeit wäre die Optimierung ebenfalls mit Hilfe der In-Memory Datenbank durchzuführen. Dazu müsste eine bzw. mehrere geeignete SQL Abfragen erstellt

werden, welche die Maximierung übernehmen. Somit könnte der komplette Ablauf effizient über die In-Memory Datenbank abgewickelt werden.

Beim Anlegen der Tabellen in der In-Memory Datenbank werden Indizes angelegt, um die spätere SQL Abfrage zu optimieren. Abhängig von der verwendeten In-Memory Datenbank könnte man hier die Wahl der indizierten Spalten auf die Eingabedaten abstimmen. Ebenfalls könnte bei der Wahl des Indextyps noch Optimierungspotential bestehen. Besonders bei großen Datenmengen könnten diese Optimierungen eine Rolle spielen.

Appendix

Listing 6.1: Benchmark ITERATIVE, Datei: lubm-ex-20-15.sparql.xml

Mode	Run	Einlesen	Evaluierung	Maximierung	Summe
ITERATIVE	1	0.385529	8.455290	7.540692	16.381510
	2	0.085544	6.745229	6.971593	13.802366
	3	0.086534	6.208391	6.601388	12.896313
Avg		0.185869	7.136303	7.037891	14.360063

Listing 6.2: Benchmark ITERATIVE, Datei: lubm-ex-20-17.sparql.xml

Mode	Run	Einlesen	Evaluierung	Maximierung	Summe
ITERATIVE	1	1.522765	447.589002	0.222999	449.334766
	2	0.787926	502.231722	0.176240	503.195888
	3	0.756780	504.534540	0.173522	505.464842
Avg		1.022490	484.785088	0.190920	485.998499

Listing 6.3: Benchmark DB **ohne** Indizes, Datei: lubm-ex-20-15.sparql.xml

Mode	Run	Einlesen	Evaluierung	Maximierung	Summe
DB – H2	1	0.393023	3.373871	9.225094	12.991987
	2	0.088311	3.191264	9.883913	13.163488
	3	0.086260	2.101740	8.453585	10.641584
Avg		0.189198	2.888958	9.187530	12.265686
DB – HSQLDB	1	0.199173	4.125125	9.223750	13.548048
	2	0.088926	3.181243	8.422373	11.692542
	3	0.081991	3.420152	7.968684	11.470826
Avg		0.123363	3.575506	8.538269	12.237139
DB – DERBY	1	0.105095	1.495696	8.609222	10.210012
	2	0.080760	0.489058	7.914831	8.484650
	3	0.103856	0.454262	7.704118	8.262236
Avg		0.096570	0.813005	8.076057	8.985633

Listing 6.4: Benchmark DB **mit** Indizes, Datei: lubm-ex-20-15.sparql.xml

Mode	Run	Einlesen	Evaluierung	Maximierung	Summe
DB – H2	1	0.352825	0.996445	7.788576	9.137846
	2	0.119607	0.430720	7.882023	8.432350
	3	0.135565	0.243901	8.396036	8.775502
Avg		0.202666	0.557022	8.022212	8.781899
DB – HSQLDB	1	0.175181	0.449496	7.665398	8.290074
	2	0.087545	0.110780	8.651116	8.849442
	3	0.122253	0.094972	7.868634	8.085859
Avg		0.128326	0.218416	8.061716	8.408458
DB – DERBY	1	0.100363	1.752615	7.845295	9.698273
	2	0.091927	0.646743	7.753848	8.492519
	3	0.090985	0.430713	7.723789	8.245487
Avg		0.094425	0.943357	7.774310	8.812093

Listing 6.5: Benchmark DB **ohne** Indizes, Datei: lubm-ex-20-17.sparql.xml

Mode	Run	Einlesen	Evaluierung	Maximierung	Summe
DB - H2	1	1.628729	151.798836	0.027196	153.454761
	2	0.932875	168.368246	0.009638	169.310759
	3	0.727528	139.374418	0.009642	140.111588
Avg		1.096377	153.180500	0.015492	154.292370
DB - HSQLDB	1	1.094776	190.624221	0.011772	191.730769
	2	0.792045	188.351868	0.012156	189.156068
	3	0.707224	190.420893	0.010178	191.138296
Avg		0.864682	189.798994	0.011369	190.675044
DB - DERBY	1	0.867574	360.264563	0.009865	361.142002
	2	0.829071	352.535138	0.009780	353.373989
	3	0.853393	355.113818	0.009660	355.976871
Avg		0.850013	355.971173	0.009768	356.830954

Listing 6.6: Benchmark DB **mit** Indizes, Datei: lubm-ex-20-17.sparql.xml

Mode	Run	Einlesen	Evaluierung	Maximierung	Summe
DB - H2	1	1.628217	2.913512	0.034014	4.575743
	2	1.008941	1.200323	0.010076	2.219340
	3	0.848121	1.131859	0.010851	1.990831
Avg		1.161759	1.748565	0.018314	2.928638
DB - HSQLDB	1	1.080915	1.066928	0.011829	2.159672
	2	0.800061	0.510261	0.010493	1.320815
	3	0.835029	0.526642	0.009950	1.371621
Avg		0.905335	0.701277	0.010757	1.617369
DB - DERBY	1	0.879342	4.859523	0.010681	5.749546
	2	0.784419	3.099707	0.010011	3.894137
	3	0.787638	2.981813	0.010333	3.779783
Avg		0.817133	3.647014	0.010342	4.474489

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[AFK+16] Shqiponja Ahmetaj, Wolfgang Fischl, Markus Kröll, Reinhard Pichler, Mantas Simkus, and Sebastian Skritek. The challenge of optional matching in SPARQL. In *Foundations of Information and Knowledge Systems - 9th International Symposium, FoIKS 2016, Linz, Austria, March 7-11, 2016. Proceedings*, pages 169–190, 2016.

[AFP+15] Shqiponja Ahmetaj, Wolfgang Fischl, Reinhard Pichler, Mantas Šimkus, and Sebastian Skritek. Towards reconciling SPARQL and certain answers. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 23–33, New York, NY, USA, 2015. ACM.

[AFP+16] Shqiponja Ahmetaj, Wolfgang Fischl, Reinhard Pichler, Mantas Simkus, and Sebastian Skritek. Towards reconciling SPARQL and certain answers — extended version. unpublished extension of [AFP+15], 2016.

[Apa15] Apache. Apache Derby: Documentation. `https://db.apache.org/derby/manuals/index.html`, 2015. [Online; accessed 01.04.2016].

[BPS15] Pablo Barcelo, Reinhard Pichler, and Sebastian Skritek. Efficient evaluation and approximation of well-designed pattern trees. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '15, pages 131–144, New York, NY, USA, 2015. ACM.

[Eng16] H2 Database Engine. H2 Advanced. `http://www.h2database.com/html/advanced.html`, 2016. [Online; accessed 31.03.2016].

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for {OWL} knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(23):158 – 182, 2005. Selected Papers from the International Semantic Web Conference, 2004 ISWC, 2004 3rd. International Semantic Web Conference, 2004.

[Gro14] RDF Working Group. Resource Description Framework (RDF). `https://www.w3.org/RDF/`, 2014. [Online; accessed 31.08.2016].

[Gro15] The HSQL Development Group. HyperSQL User Guide. http://hsqldb.org/doc/2.0/guide/index.html, 2015. [Online; accessed 31.03.2016].

[LPPS13] Andrés Letelier, Jorge Pérez, Reinhard Pichler, and Sebastian Skritek. Static analysis and optimization of semantic web queries. *ACM Trans. Database Syst.*, 38(4):25:1–25:45, December 2013.

[Ora16] Oracle. PreparedStatement. https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html#addBatch--, 2016. [Online; accessed 06.04.2016].

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009.

[Pla13] Hasso Plattner. *A Course in In-Memory Data Management: The Inner Mechanics of In-Memory Databases*. Springer, 2013.

[PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation, W3C, 2008. http://www.w3.org/TR/rdf-sparql-query/.