PigSPARQL: Übersetzung von SPARQL nach Pig Latin

Alexander Schätzle, Martin Przyjaciel-Zablocki, Thomas Hornung, Georg Lausen

Lehrstuhl für Datenbanken und Informationssysteme
Albert-Ludwigs-Universität Freiburg
Georges-Köhler Allee, Geb. 51
79110 Freiburg im Breisgau
{schaetzl, zablocki, hornungt, lausen} @ informatik.uni-freiburg.de

Abstract: Dieser Beitrag untersucht die effiziente Auswertung von SPARQL-Anfragen auf großen RDF-Datensätzen. Zum Einsatz kommt hierfür das Apache Hadoop Framework, eine bekannte Open-Source Implementierung von Google's MapReduce, das massiv parallelisierte Berechnungen auf einem verteilten System ermöglicht. Zur Auswertung von SPARQL-Anfragen mit Hadoop wird in diesem Beitrag PigSPARQL, eine Übersetzung von SPARQL nach Pig Latin, vorgestellt. Pig Latin ist eine von Yahoo! Research entworfene Sprache zur verteilten Analyse von großen Datensätzen. Pig, die Implementierung von Pig Latin für Hadoop, übersetzt ein Pig Latin-Programm in eine Folge von MapReduce-Jobs, die anschließend auf einem Hadoop-Cluster ausgeführt werden. Die Evaluation von PigSPARQL anhand eines SPARQL spezifischen Benchmarks zeigt, dass der gewählte Ansatz eine effiziente Auswertung von SPARQL-Anfragen mit Hadoop ermöglicht.

1 Einleitung

Die Menge der Daten im Internet und damit auch das potentiell zur Verfügung stehende Wissen nimmt schnell zu. Leider können große Teile dieses Wissens nicht automatisiert erfasst und verarbeitet werden, da sich die Aufbereitung und Darstellung an einem menschlichen Betrachter orientiert. Das Ziel des *Semantic Web* [BHL01] ist die Erschließung und automatisierte Verarbeitung dieses Wissens. Zu diesem Zweck wurde das Resource Description Framework (RDF) [MMM04] entwickelt, ein Standard zur Repräsentation von Daten in einem maschinenlesbaren Format. SPARQL [PS08] ist die vom W3C[1] empfohlene Anfrage-Sprache für RDF.

Die zunehmende Größe von Datensätzen erfordert die Entwicklung neuer Konzepte zur Datenverarbeitung und Datenanalyse. Google entwickelte hierfür 2004 das sogenannte MapReduce-Modell [DG04], das es dem Anwender erlaubt, parallele Berechnungen auf sehr großen Datensätzen verteilt auf einem Computer-Cluster durchzuführen, ohne sich um die Details und die damit verbundenen Probleme eines verteilten Systems Gedanken machen zu müssen. Die bekannteste frei verfügbare Implementierung des MapReduce-

[1] World Wide Web Consortium – siehe [http://www.w3.org/]

Modells ist das Hadoop Framework[2], das maßgeblich von Yahoo! weiterentwickelt wird. Da die Entwicklung auf MapReduce-Ebene trotz aller Vorzüge dennoch recht technisch und anspruchsvoll ist, entwickelten Mitarbeiter von Yahoo! eine Sprache zur Analyse von großen Datensätzen mit Hadoop, Pig Latin [Ol08], die dem Anwender eine einfache Abstraktionsebene zur Verfügung stellen soll. Pig, die Implementierung von Pig Latin für Hadoop, ist mittlerweile ein offizielles Subprojekt von Hadoop.

Da insbesondere auch die Menge der verfügbaren RDF-Datensätze stetig zunimmt[3], müssen auch hier neue Konzepte zur Auswertung solcher Datensätze entwickelt werden. Dabei sind klassische, auf nur einem Computer ausgeführte, Systeme aufgrund der begrenzten Ressourcen zunehmend überfordert. Die grundlegende Idee dieser Arbeit ist es daher, die Mächtigkeit von Hadoop zur Auswertung von SPARQL-Anfragen auf großen RDF-Datensätzen zu nutzen. Hierfür wurde eine Übersetzung von SPARQL nach Pig Latin entwickelt und implementiert, die eine SPARQL-Anfrage in ein äquivalentes Pig Latin-Programm überführt. Dieser Ansatz hat den Vorteil, dass die Übersetzung direkt von bestehenden Optimierungen bzw. zukünftigen Weiterentwicklungen von Pig profitiert. Die wesentlichen Beiträge unserer Arbeit sind wie folgt: Zunächst definieren wir für ein ausdrucksmächtiges Fragment von SPARQL eine Übersetzung in ein äquivalentes Pig Latin-Programm. Das betrachtete Fragment deckt dabei insbesondere einen Großteil der Anfragen ab, die in der offiziellen Dokumentation [PS08] enthalten sind. Nach unserem Kenntnisstand ist diese Arbeit die erste umfassende Darstellung einer Übersetzung von SPARQL nach Pig Latin. Darüber hinaus untersuchen wir Optimierungsstrategien für die Übersetzung und bestätigen deren Wirksamkeit. Abschließend zeigen wir anhand eines SPARQL spezifischen Performance Benchmarks, dass die von uns entwickelte Übersetzung eine effiziente Auswertung von SPARQL-Anfragen auf sehr großen RDF-Datensätzen ermöglicht.

Der weitere Verlauf dieser Arbeit ist wie folgt strukturiert. In Kapitel 2 werden die nötigen Grundlagen von RDF, SPARQL, MapReduce und Pig Latin kurz dargestellt. Kapitel 3 erläutert die von uns entwickelte Übersetzung von SPARQL nach Pig Latin. In Kapitel 4 folgt die Evaluation der Übersetzung mit einem SPARQL spezifischen Performance Benchmark und Kapitel 5 gibt einen Überblick über verwandte Arbeiten. Abschließend werden die Ergebnisse der Arbeit in Kapitel 6 zusammengefasst.

2 Grundlagen

Dieses Kapitel stellt die Grundlagen der von uns verwendeten Technologien kurz dar.

2.1 RDF

Das Resource Description Framework (RDF) ist ein vom World Wide Web Consortium (W3C) entwickelter Standard zur Modellierung von Metainformationen über beliebige Ressourcen (z.B. Personen oder Dokumente). Eine ausführliche Darstellung von RDF

[2] siehe [http://hadoop.apache.org]
[3] siehe [http://esw.w3.org/TaskForces/CommunityProjects/LinkingOpenData/DataSets/Statistics]

findet sich in [MMM04]. Im Folgenden werden nur die grundlegenden Konzepte von RDF kurz vorgestellt.

RDF-Tripel. Grundlage der Wissensrepräsentation in RDF sind Ausdrücke der Form <Subjekt, Prädikat, Objekt>. Ein RDF-Tripel lässt sich folgendermaßen interpretieren:

<Subjekt> hat die Eigenschaft <Prädikat> mit dem Wert <Objekt>.

RDF-Tripel können URIs, Blank Nodes und RDF-Literale enthalten. URIs (Uniform Resource Identifier) sind weltweit eindeutige Bezeichner für Ressourcen (z.B. http://example.org/Peter), Blank Nodes sind lokal eindeutige Bezeichner (z.B. _:address) und RDF-Literale sind atomare Werte (z.B. "27").

RDF-Graph. Ein RDF-Dokument besteht im Wesentlichen aus einer Abfolge von RDF-Tripeln und lässt sich als gerichteter Graph interpretieren. Jedes Tripel entspricht dabei einer beschrifteten Kante (Prädikat) von einem Knoten im Graph (Subjekt) zu einem anderen Knoten (Objekt).

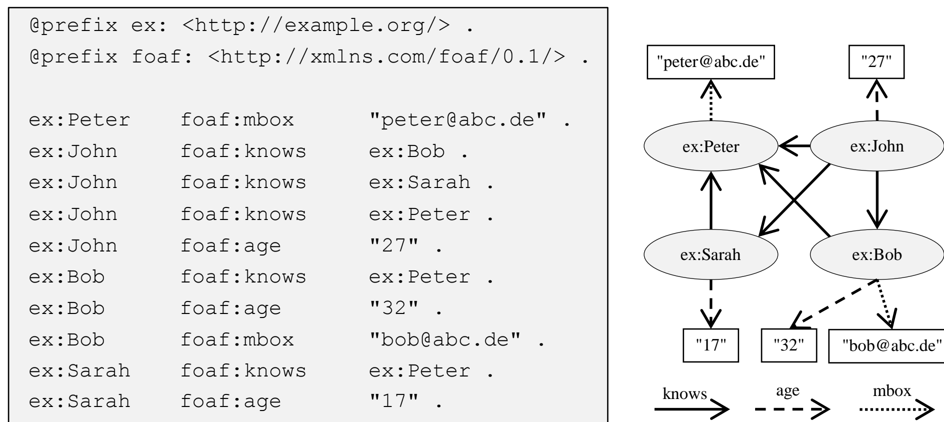

Abbildung 1: RDF-Dokument mit entsprechendem RDF-Graph

2.2 SPARQL

SPARQL[4] ist die vom W3C empfohlene Anfragesprache für RDF. Die folgende kurze Darstellung beruht auf der offiziellen Dokumentation von SPARQL aus [PS08]. Eine formale Definition der Semantik von SPARQL findet sich ebenfalls in der offiziellen Dokumentation oder in [PAG09].

Graph Pattern. Eine SPARQL-Anfrage definiert im Wesentlichen ein Graph Pattern (Muster), das auf dem RDF-Graph *G*, auf dem die Anfrage operiert, ausgewertet wird. Dazu wird überprüft, ob die Variablen im Graph Pattern durch Knoten aus *G* ersetzt werden können, sodass der resultierende Graph in *G* enthalten ist (Pattern Matching). Grundlage jedes Graph Patterns bilden die *Basic Graph Patterns* (BGP). Ein BGP

[4] SPARQL ist ein rekursives Akronym und steht für *SPARQL Protocol and RDF Query Language*.

besteht aus einer endlichen Menge an *Triple-Patterns*, die mittels AND (.) verkettet werden. Ein Triple-Pattern ist ein RDF-Tripel, wobei Subjekt, Prädikat und Objekt mit einer Variablen (?var) belegt sein können (z.B. ?s :p ?o). Ein Graph Pattern lässt sich dann rekursiv definieren:

- Ein *Basic Graph Pattern* ist ein Graph Pattern.
- Sind P und P' Graph Pattern, dann sind auch $\{P\}$. $\{P'\}$, P UNION $\{P'\}$ und P OPTIONAL $\{P'\}$ Graph Pattern.
- Ist P ein Graph Pattern und R eine Filter-Bedingung, dann ist auch P FILTER (R) ein Graph Pattern.

Mit Hilfe des FILTER-Operators lassen sich die Werte von Variablen im Graph Pattern beschränken und der OPTIONAL-Operator erlaubt das optionale Hinzufügen von Informationen zum Ergebnis einer Anfrage. Sollten die gewünschten Informationen nicht vorhanden sein, so bleiben die entsprechenden Variablen im Ergebnis *ungebunden*, d.h. es wird ihnen kein Wert zugeordnet. Mit Hilfe des UNION-Operators lassen sich zwei alternative Graph Patterns in einer Anfrage definieren. Ein Anfrage-Ergebnis muss dann mindestens eines der beiden Patterns erfüllen. Darüber hinaus gibt es in SPARQL auch einen GRAPH-Operator, der die Referenzierung mehrerer RDF-Graphen in einer Anfrage ermöglicht. Im Folgenden beschränken wir uns allerdings auf Anfragen, die sich nur auf einen RDF-Graph beziehen.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT *
WHERE {
  ?person foaf:knows ex:Peter .
  ?person foaf:age ?age
  FILTER (?age >= 18)
  OPTIONAL {
    ?person foaf:mbox ?mb
  }
}
```

Abbildung 2: SPARQL-Anfrage mit FILTER und OPTIONAL

Die Anfrage aus Abbildung 2 ermittelt alle Personen, die "Peter" kennen und mindestens 18 Jahre alt sind. Sollte außerdem eine Mailbox-Adresse bekannt sein, so wird diese zum Ergebnis der Anfrage hinzugefügt. Tabelle 1 zeigt das Ergebnis der Anfrage auf dem RDF-Graph aus Abbildung 1.

?person	?age	?mb
ex:John	27	
ex:Bob	32	bob@abc.de

Tabelle 1: Auswertung der Anfrage

2.3 MapReduce

MapReduce ist ein von Google im Jahr 2004 vorgestelltes Modell für nebenläufige Berechnungen auf sehr großen Datenmengen unter Einsatz eines Computer-Clusters [DG04]. Inspiriert wurde das Konzept durch die in der funktionalen Programmierung häufig verwendeten Funktionen *map* und *reduce*. Ausgangspunkt für die Entwicklung war die Erkenntnis, dass viele Berechnungen bei Google zwar konzeptuell relativ einfach sind, jedoch zumeist auf sehr großen Datensätzen ausgeführt werden müssen. Eine parallelisierte Ausführung ist daher oftmals unerlässlich, weshalb selbst einfache Berechnungen eine komplexe Implementierung erforderten, um mit den Problemen einer parallelisierten Ausführung umgehen zu können. Bei der Entwicklung von MapReduce standen daher insbesondere eine gute Skalierbarkeit und Fehlertoleranz des Systems im Mittelpunkt, da bei großen Computer-Clustern immer mit Ausfällen gerechnet werden muss.

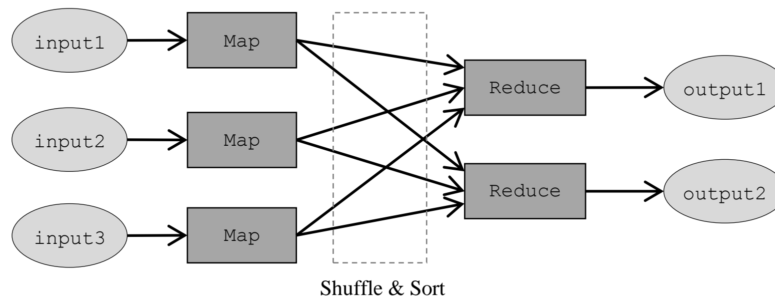

Abbildung 3: MapReduce-Datenfluss

Im Prinzip muss der Entwickler bei der Erstellung eines MapReduce-Jobs lediglich eine *Map*- und eine *Reduce*-Funktion implementieren, die vom Framework parallelisiert auf den Eingabe-Daten ausgeführt werden. Abbildung 3 zeigt den schematischen Ablauf eines MapReduce-Jobs mit drei Mappern und zwei Reducern. Technisch gesehen berechnet ein MapReduce-Job aus einer Liste von Schlüssel-Wert-Paaren (Eingabe) eine neue Liste von Schlüssel-Wert-Paaren (Ausgabe):

$$[(k_1, v_1), \dots, (k_n, v_n)] \mapsto [(k'_1, v'_1), \dots, (k'_m, v'_m)]$$

Beispiel. Angenommen es soll für eine Menge von Dokumenten berechnet werden, welche Wörter mit welcher Häufigkeit darin vorkommen. Eine Vorverarbeitung habe bereits eine Liste an Paaren (*docID*, *word*) ergeben, wobei *docID* eine Referenz auf ein Dokument und *word* ein Wort aus dem entsprechenden Dokument repräsentieren. Diese Vorverarbeitung lässt sich ebenfalls mit Hilfe eines MapReduce-Jobs berechnen. Ein mögliches Ergebnis der Auswertung könnte dann so aussehen:

$$[(doc1, \text{Peter}), (doc1, \text{Bob}), (doc2, \text{Peter}), (doc2, \text{Sarah})] \mapsto [(\text{Peter}, 2), (\text{Bob}, 1), (\text{Sarah}, 1)]$$

Eine wichtige Eigenschaft des MapReduce-Modells ist das sogenannte *Lokalitätsprinzip*. Um das Netzwerk zu entlasten wird dabei ausgenutzt, dass die Daten verteilt auf den Computern im Cluster abgespeichert sind. Das System versucht die Mapper so auf die Rechner zu verteilen, dass möglichst viele Daten lokal gelesen werden können und nicht über das Netzwerk übertragen werden müssen.

2.4 Pig Latin

Pig Latin ist eine von Yahoo! entworfene Sprache zur Analyse von großen Datenmengen [Ol08], die für den Einsatz im Hadoop Framework entwickelt wurde, einer Open-Source Implementierung von Google's MapReduce. Die Implementierung von Pig Latin für Hadoop, *Pig*, übersetzt ein Pig Latin-Programm in eine Folge von MapReduce-Jobs und ist mittlerweile ein offizielles Subprojekt von Hadoop.

Datenmodell. Pig Latin besitzt ein vollständig geschachteltes Datenmodell und erlaubt dem Entwickler damit eine größere Flexibilität als die von der ersten Normalform vorgeschriebenen flachen Tabellen von relationalen Datenbanken. Das Datenmodell von Pig Latin kennt vier verschiedene Typen:

- *Atom*: Ein Atom beinhaltet einen einfachen, atomaren Wert wie eine Zeichenkette oder eine Zahl. Beispiel: `'Sarah'` oder `24`
- *Tupel*: Ein Tupel besteht aus einer Sequenz von *Feldern*, wobei jedes Feld einen beliebigen Datentyp besitzen kann. Jedem Feld in einem Tupel kann zudem ein *Name* (Alias) zugewiesen werden, über den das Feld referenziert werden kann.
  Beispiel: `('John','Doe')` mit Alias-Namen `(Vorname,Nachname)`
- *Bag*: Eine Bag besteht aus einer Kollektion von Tupeln, wobei ein Tupel auch mehrfach vorkommen darf. Darüber hinaus müssen die Schemata der Tupel nicht übereinstimmen, d.h. die Tupel können eine unterschiedliche Anzahl von Feldern mit unterschiedlichen Typen aufweisen.
  Beispiel: $\left\{ \begin{matrix} (\texttt{'Bob'},\texttt{'Sarah'}) \\ (\texttt{'Peter'}, (\texttt{'likes'},\texttt{'football'})) \end{matrix} \right\}$
- *Map*: Eine *Map* beinhaltet eine Kollektion von Datenelementen. Jedes Element kann dabei über einen zugeordneten Schlüssel referenziert werden.
  Beispiel: $\left[\begin{matrix} \texttt{'name'} \rightarrow \texttt{'John'} \\ \texttt{'knows'} \rightarrow \left\{ \begin{matrix} (\texttt{'Sarah'}) \\ (\texttt{'Bob'}) \end{matrix} \right\} \end{matrix} \right]$

Operatoren. Ein Pig Latin-Programm besteht aus einer Sequenz von Schritten, wobei jeder Schritt einer einzelnen Daten-Transformation entspricht. Da Pig Latin für die Bearbeitung von großen Datenmengen mit Hadoop entwickelt wurde, müssen die Operatoren gut *parallelisierbar* sein. Daher wurden konsequenterweise nur solche Operatoren aufgenommen, die sich in eine Folge von MapReduce-Jobs übersetzen und damit parallel ausführen lassen. Im Folgenden werden die für die Übersetzung wichtigsten Operatoren in aller Kürze vorgestellt. Für eine genauere Darstellung sei auf die offizielle Dokumentation von Pig Latin [Ap10] verwiesen.

- LOAD: Für die Bearbeitung mit Pig Latin müssen die Daten deserialisiert und in das Datenmodell von Pig Latin überführt werden. Hierfür kann eine *User Defined Function* (UDF) implementiert werden, die vom LOAD-Operator verwendet werden soll und das tupelweise Laden beliebiger Daten ermöglicht.

  Beispiel: `persons = LOAD 'file' USING myLoad() AS (name,age,city);`

- FOREACH: Mit Hilfe des FOREACH-Operators lässt sich eine Verarbeitung auf jedes Tupel in einer Bag anwenden. Insbesondere lassen sich damit Felder eines Tupels entfernen oder neue Felder hinzufügen.

  Beispiel:
```
result1 = FOREACH persons GENERATE
          name, age>=18? 'adult':'minor' AS class;
```

persons

name	age	city
Sarah	17	Freiburg
Bob	32	Berlin

result1

name	class
Sarah	minor
Bob	adult

- FILTER: Der FILTER-Operator ermöglicht das Entfernen ungewollter Tupel aus einer Bag. Dazu wird die Bedingung auf alle Tupel in der Bag angewendet.

  Beispiel:
```
result2 = FILTER persons BY age>=18;
```

result2

name	age	city
Bob	32	Berlin

- [OUTER] JOIN: Equi-Joins lassen sich in Pig Latin mit Hilfe des JOIN-Operators ausdrücken. Eine Besonderheit von Pig Latin ist darüber hinaus, dass sich ein JOIN auch auf mehr als zwei Relationen beziehen kann (*Multi-Join*). Über die Schlüsselwörter LEFT OUTER bzw. RIGHT OUTER können auch Outer Joins in Pig Latin realisiert werden.

  Beispiel:
```
result3 = JOIN result1 BY name LEFT OUTER,
               result2 BY name;
```

result3

result1:: name	result1:: class	result2:: name	result2:: age	result2:: city
Bob	adult	Bob	32	Berlin
Sarah	minor			

- UNION: Zwei oder mehr Bags können mit Hilfe des UNION-Operators zu einer Bag zusammengeführt werden, wobei Duplikate (mehrfach vorkommende Tupel) erlaubt sind. Im Gegensatz zu relationalen Datenbanken müssen die Tupel dabei nicht das gleiche Schema und insbesondere nicht die gleiche Anzahl an Feldern besitzen. Im Regelfall ist es allerdings nicht besonders empfehlenswert, Bags mit unterschiedlichen Schemata zu vereinigen, da die Schema-Informationen (speziell die Alias-Namen für die Felder der Tupel) dabei verloren gehen.

3 Übersetzung von SPARQL nach Pig Latin

Die direkte Übersetzung einer SPARQL-Anfrage in ein Pig Latin-Programm wäre aufgrund der komplexen Syntax und der datenorientierten Struktur von SPARQL äußerst schwierig. Deshalb wird die Anfrage zunächst nach dem offiziellen Schema des W3C in einen SPARQL Algebra-Baum überführt und anschließend in ein Pig Latin-Programm übersetzt. Abbildung 4 zeigt das grundlegende Konzept der Übersetzung, wie sie in dieser Arbeit vorgestellt wird. Die modulare Vorgehensweise bietet mehrere Vorteile, so können beispielsweise Optimierungen auf der Algebra-Ebene durchgeführt werden, ohne die Übersetzung der Algebra in ein Pig Latin-Programm verändern zu müssen.

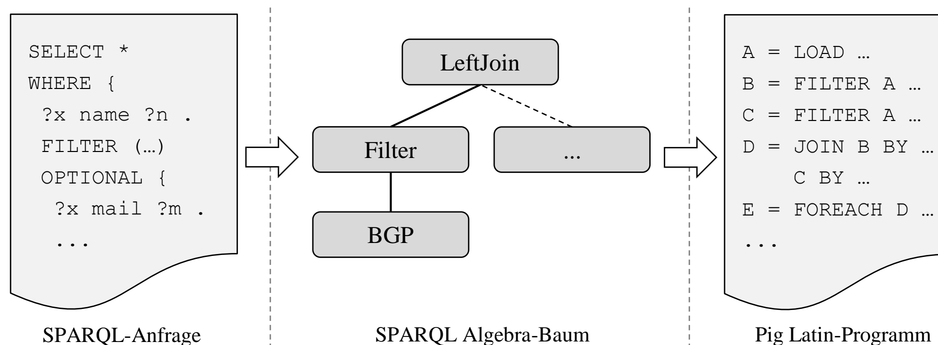

Abbildung 4: Schematischer Ablauf der Übersetzung

3.1 SPARQL Algebra

Zur Auswertung einer SPARQL-Anfrage wird die Anfrage zunächst in einen Ausdruck der SPARQL Algebra überführt, da die Semantik einer Anfrage auf der Ebene der SPARQL Algebra definiert ist. Ein solcher Ausdruck lässt sich in Form eines *Algebra-Baums* repräsentieren. Tabelle 2 stellt die Operatoren der SPARQL Algebra ihren Entsprechungen in der SPARQL Syntax gegenüber.

Algebra	SPARQL Syntax
BGP	Menge von Triple-Patterns (verkettet über Punkt-Symbol)
Join	Verknüpfung zweier Gruppen ({…}.{…})
LeftJoin	OPTIONAL
Filter	FILTER
Union	UNION
Graph	GRAPH

Tabelle 2: Zusammenhang zwischen SPARQL Algebra und Syntax

3.2 Abbildung der RDF-Daten in das Datenmodell von Pig

Ein RDF-Datensatz besteht im Wesentlichen aus einer Menge von RDF-Tripeln und ein RDF-Tripel setzt sich aus URIs, RDF-Literalen und Blank Nodes zusammen. URIs sind nach ihrer Syntax spezielle ASCII-Zeichenfolgen und lassen sich daher im Datenmodell von Pig als Atome repräsentieren (<URI>). RDF unterscheidet des Weiteren zwischen *einfachen* und *getypten* Literalen. Einfache Literale sind Unicode-Zeichenfolgen und können daher ebenfalls als Atome repräsentiert werden. Getypte Literale haben entweder einen zusätzlichen *Language-Tag* oder einen *Datentyp* und lassen sich durch einen zusammengesetzten Wert (`"literal"@lang` bzw. `"literal"^^datatype`) repräsentieren. Bei der Auswertung von arithmetischen Ausdrücken werden die Literale zur Laufzeit in den entsprechenden Typ umgewandelt. Hierfür wurden für die bei RDF gebräuchlichen Datentypen (z.B. `xsd:integer`) spezielle Umwandlungen definiert. Die RDF-Syntax macht keine näheren Angaben zur internen Struktur von Blank Nodes, sie müssen lediglich von URIs und Literalen unterscheidbar sein. Um dies zu gewährleisten, kann eine Darstellung der Form `_:nodeID` verwendet werden. Ein RDF-Tripel lässt sich somit als Tupel aus drei atomaren Feldern mit dem Schema `(s:chararray, p:chararray, o:chararray)` darstellen.

3.3 Übersetzung der SPARQL Algebra

Im Folgenden werden für die Operatoren der SPARQL Algebra Vorschriften zur Übersetzung in eine Folge von Pig Latin-Befehlen angegeben. Hierfür muss zunächst die benötigte Terminologie eingeführt werden, die analog zu [PAG09] definiert wird: Sei *V* die unendliche Menge an Anfrage-Variablen und *T* die Menge der gültigen RDF-Terme (URIs, RDF-Literale, Blank Nodes).

Definition 1. Ein *(Solution) Mapping* μ ist eine partielle Funktion $\mu: V \rightarrow T$. Die Domain von μ, $dom(\mu)$, ist die Teilmenge von *V*, wo μ definiert ist. Im Folgenden wird ein Solution Mapping umgangssprachlich auch als *Ergebnis* bezeichnet.

Definition 2. Zwei Solution Mappings μ_1 und μ_2 sind *kompatibel*, wenn für alle Variablen $?X \in dom(\mu_1) \cap dom(\mu_2)$ gilt, dass $\mu_1(?X) = \mu_2(?X)$. Es folgt, dass $\mu_1 \cup \mu_2$ wieder ein Solution Mapping ergibt und zwei Mappings mit disjunkten Domains immer kompatibel sind.

Probleme bei der Auswertung von SPARQL mit Pig Latin bereiten *ungebundene* Variablen, die durch die Anwendung des OPTIONAL-Operators entstehen können. Im Gegensatz zu einem NULL-Wert in der relationalen Algebra führt eine ungebundene Variable in SPARQL bei der Auswertung eines Joins nicht dazu, dass das entsprechende Tupel verworfen wird [Cy05]. Da sich Joins in Pig Latin an der relationalen Algebra orientieren und ungebundene Variablen als NULL-Werte in Pig Latin dargestellt werden, führt dies zu unterschiedlichen Ergebnissen bei der Auswertung. Aus diesem Grund betrachten wir bei der Übersetzung nach Pig Latin *schwach wohlgeformte* Graph Pattern, die in Anlehnung an *wohlgeformte* Graph Pattern nach Pérez et al. [PAG09] definiert werden aber weniger restriktiv sind. Insbesondere sind die meisten Anfragen aus der offiziellen SPARQL Dokumentation [PS08] schwach wohlgeformt.

Definition 3. Ein Graph Pattern P ist *schwach wohlgeformt*, wenn es kein GRAPH enthält, UNION nicht in einem anderen Operator enthalten ist und für jedes Sub-Pattern $P' = (P_1 \text{ OPTIONAL } P_2)$ von P und jede Variable $?X$ aus P gilt: Falls $?X$ sowohl in P_2 als auch außerhalb von P' vorkommt, dann kommt sie auch in P_1 vor.

Ist das Graph Pattern einer Anfrage schwach wohlgeformt, so treten keine Joins über NULL-Werte auf, da dies nur der Fall ist, falls eine Variable in P_2 und außerhalb von P' vorkommt aber nicht in P_1 oder nach einem UNION noch weitere Operatoren folgen. Anfragen, die nicht schwach wohlgeformt sind und somit bei der Auswertung zu einem NULL-Join führen, werden von unserem Übersetzer erkannt.

Basic Graph Pattern (BGP). BGPs bilden die Grundlage jeder SPARQL-Anfrage und werden direkt auf dem entsprechenden RDF-Graph ausgewertet. Sie liefern als Ergebnis eine Menge von Solution Mappings, die als Eingabe für die weiteren Operatoren dienen. Genauer gesagt handelt es sich dabei um eine *Multi-Menge*, da ein Solution Mapping mehrfach vorkommen kann.

Beispiel: `P1 = BGP(?A knows Bob . ?A age ?B . ?A mbox ?C)`

Das BGP ermittelt alle Personen, die Bob kennen und für die sowohl das Alter als auch eine Mailbox-Adresse bekannt sind. Solution Mappings lassen sich in Pig in Form einer Relation (flache Bag) repräsentieren. Jedes Tupel der Relation entspricht einem Solution Mapping und die Felder des Tupels entsprechen den Werten für die Variablen. Das Schema der Relation bilden die Namen der Variablen ohne führendes Fragezeichen, da diese bei Alias-Namen in Pig nicht erlaubt sind. Abbildung 5 zeigt die Übersetzung von P1 in eine Folge von Pig Latin-Befehlen.

```
graph = LOAD 'pathToFile' USING rdfLoader() AS (s,p,o) ; (1)

t1 = FILTER graph BY p == 'knows' AND o == 'Bob' ;       (2)
t2 = FILTER graph BY p == 'age' ;
t3 = FILTER graph BY p == 'mbox' ;

j1 = JOIN t1 BY s, t2 BY s ;                             (3)
j2 = JOIN j1 BY t1::s, t3 BY s;

P1 = FOREACH j2 GENERATE                                 (4)
     t1::s AS A, t2::o AS B, t3::o AS C ;
```

Abbildung 5: Übersetzung eines BGPs

(1) Das Laden der Daten erfolgt mit Hilfe des LOAD-Operators. Hierfür muss eine spezielle *Loader-UDF* implementiert werden. Anschließend stehen die Daten im Format aus Abschnitt 3.1 zur Verfügung.
(2) Für jedes Triple-Pattern im BGP wird ein FILTER benötigt, der diejenigen RDF-Tripel selektiert, die das Triple-Pattern erfüllen (Pattern Matching).
(3) Die Ergebnisse der FILTER werden dann sukzessive mit Hilfe des JOIN-Operators verknüpft. In jedem Schritt wird dabei ein weiteres Triple-Pattern zur berechneten Lösung hinzugenommen. Besteht das BGP aus n Triple-Patterns, so sind folglich n-1 Joins erforderlich. Das Prädikat des Joins ergibt sich jeweils aus

den gemeinsamen Variablen der beiden Argumente. Der Join verknüpft damit die kompatiblen Solution Mappings der beiden Argumente und erzeugt daraus neue Solution Mappings. Sollte es keine gemeinsamen Variablen geben, so muss das Kreuzprodukt der beiden Argumente berechnet werden.

(4) Durch ein abschließendes FOREACH werden die überflüssigen Spalten der Relation mit den berechneten Solution Mappings entfernt und das Schema der Relation an die Variablennamen angepasst.

Filter. Der Filter-Operator der SPARQL Algebra dient dazu, aus einer Multi-Menge an Solutions Mappings diejenigen Mappings zu entfernen, welche die Filter-Bedingung nicht erfüllen.

Beispiel: `P2 = Filter(?B >= 30 && ?B <= 40, P1)`

Aus den Ergebnissen (Solution Mappings) für das Pattern P1 sollen diejenigen Personen entfernt werden, die jünger als 30 oder älter als 40 Jahre sind. Ein Filter lässt sich in Pig Latin mit Hilfe des FILTER-Befehls ausführen. Nicht alle Filter-Bedingungen in SPARQL lassen sich allerdings direkt in Pig Latin ausdrücken. So ist z.B. die Syntax von regulären Ausdrücken in SPARQL und Pig Latin verschieden[5].

```
P2 = FILTER P1 BY (B >= 30 AND B <= 40) ;
```

Abbildung 6: Übersetzung eines Filters

Join. Der Join-Operator der SPARQL Algebra bekommt als Eingabe zwei Multi-Mengen von Solution Mappings. Er kombiniert die kompatiblen Solution Mappings aus beiden Mengen und erzeugt so eine neue Multi-Menge an Solution Mappings für das zusammengesetzte Pattern.

Beispiel: `P3 = Join(BGP(?A knows ?B), BGP(?A age ?C . ?B age ?C))`

Das linke Pattern (*P*) liefert alle Personen, die eine andere Person kennen und das rechte Pattern (*P'*) liefert alle Personen-Paare, die das gleiche Alter haben. Über den Join-Operator werden die beiden Mengen von Solution Mappings zu einer Menge verknüpft. Das Ergebnis der Anfrage sind somit alle Paare von Personen, die sich kennen und gleich alt sind. Ein Join lässt sich in Pig Latin analog zum BGP mit Hilfe des JOIN-Befehls realisieren. Das Prädikat des Joins ergibt sich auch hier aus den gemeinsamen Variablen der beiden Eingabe-Relationen (Multi-Mengen von Solution Mappings). Sollte es keine gemeinsamen Variablen geben, so muss das Kreuzprodukt der beiden Relationen berechnet werden. Abschließend werden mit FOREACH die überflüssigen Spalten entfernt und das Schema der Ergebnis-Relation angepasst.

```
j1 = JOIN BGP1 BY (A,B), BGP2 BY (A,B) ;
P3 = FOREACH j1 GENERATE
     BGP1::A AS A, BGP1::B AS B, BGP2::C AS C ;
```

Abbildung 7: Übersetzung eines Joins

[5] SPARQL unterstützt reguläre Ausdrücke wie in XPath 2.0 oder XQuery 1.0 während Pig Latin die umfangreicheren regulären Ausdrücke von Java unterstützt (vgl. [Ol08]).

LeftJoin. Mit Hilfe des LeftJoin-Operators können zusätzliche Informationen zum Ergebnis hinzugenommen werden, falls diese vorhanden sind. Konzeptionell entspricht der LeftJoin damit einem klassischen Left-Outer Join. Der LeftJoin kann auch eine Filter-Bedingung beinhalten, die als Bedingung für den Outer Join interpretiert werden kann. Eine Darstellung der Übersetzung eines LeftJoins mit Filter ist im Rahmen dieses Beitrags nicht möglich. Hierfür sei auf die vollständige Ausarbeitung [Sc10] verwiesen.

Beispiel: `P4 = LeftJoin(BGP(?A age ?B), BGP(?A mbox ?C), true)`

Enthält der LeftJoin keinen Filter, so wird dies wie in P4 durch eine Filter-Bedingung ausgedrückt, die immer erfüllt ist (true). Der LeftJoin aus P4 liefert alle Personen, für die das Alter bekannt ist. Sollte außerdem noch eine Mailbox-Adresse bekannt sein, so wird auch diese zum Ergebnis hinzugenommen. Ohne Filter lässt sich der LeftJoin als normaler OUTER JOIN in Pig Latin auf den gemeinsamen Variablen der beiden Eingabe-Relationen realisieren. Gibt es keine gemeinsamen Variablen, so muss auch hier das Kreuzprodukt der beiden Relationen berechnet werden. Ein abschließendes FOREACH entfernt die überflüssigen Spalten und passt das Schema der Ergebnis-Relation an.

```
lj = JOIN BGP1 BY A LEFT OUTER, BGP2 BY A ;
P4 = FOREACH lj GENERATE
     BGP1::A AS A, BGP1::B AS B, BGP2::C AS C ;
```

Abbildung 8: Übersetzung eines LeftJoins ohne Filter

Union. Der Union-Operator der SPARQL Algebra fasst zwei Multi-Mengen von Solution Mappings zu einer Multi-Menge zusammen. Damit lassen sich folglich die Ergebnisse von zwei Graph Patterns vereinigen.

Beispiel: `P5 = Union(BGP(?A knows Bob . ?A mbox ?B), BGP(?A knows John))`

Das linke Pattern (P) liefert alle Personen, die Bob kennen und deren Mailbox-Adresse bekannt ist. Das rechte Pattern (P') hingegen liefert alle Personen, die John kennen, unabhängig von einer Mailbox-Adresse. Obwohl der Union-Operator zunächst relativ unproblematisch wirkt, ist die Übersetzung mit einigen Problemen verbunden. Das liegt daran, dass für zwei Mappings $\mu \in P$ und $\mu' \in P'$ gelten kann, dass $dom(\mu) \neq dom(\mu')$, wie es beispielsweise bei P5 der Fall ist. In diesem Fall müssen zunächst die Schemata der beiden Relationen mit Hilfe von FOREACH aneinander angepasst werden, indem für ungebundene Variablen Null-Werte eingeführt werden. Andernfalls gehen die Schemata der beiden Relationen verloren, da die Ergebnis-Relation weder das Schema der einen noch das Schema der anderen Relation übernehmen kann. Da die Variablen-Namen der Solution Mappings allerdings im Schema der Relation definiert sind, wäre dies äußerst problematisch. Im Beispiel von P5 muss z.B. zunächst das Schema der rechten Relation an das Schema der linken Relation angepasst werden (Abbildung 9).

```
BGP2 = FOREACH BGP2 GENERATE A, null AS B ;
P5   = UNION BGP1, BGP2 ;
```

Abbildung 9: Übersetzung eines Unions

Eine vollständige Darstellung der entwickelten Übersetzung ist im Rahmen dieses Beitrags leider nicht möglich. Der interessierte Leser sei hierfür auf die vollständige Darstellung in [Sc10] verwiesen.

3.4 Optimierungen

Die Optimierung von SPARQL-Anfragen ist Gegenstand aktueller Forschung [HH07, St08, SML10]. Im Folgenden werden einige von uns untersuchte Optimierungsstrategien für die entwickelte Übersetzung kurz dargestellt. Bei den ersten Evaluationen hat sich gezeigt, dass eine Optimierung vor allem die Reduktion des Datenaufkommens (Input/Output, I/O) zum Ziel haben sollte. Das beinhaltet zum einen die Daten, welche innerhalb eines MapReduce-Jobs von den Mappern zu den Reducern übertragen werden und zum anderen die Daten, welche zwischen zwei MapReduce-Jobs in das verteilte Dateisystem von Hadoop, *HDFS*, übernommen werden müssen.

(1) **SPARQL Algebra.** Zur Reduktion der erzeugten Zwischenergebnisse einer Anfrage wurden Optimierungen des *Filter*- und des *BGP*-Operators betrachtet. Ziel dieser Optimierungen ist die möglichst frühzeitige Auswertung von Filtern sowie die Neuanordnung der Triple-Patterns in einem BGP entsprechend ihrer Selektivität [St08]. Dabei werden die Triple-Patterns nach der Anzahl und Position ihrer Variablen geordnet, da ein Triple-Pattern mit zwei Variablen als weniger selektiv angesehen wird wie ein Triple-Pattern mit nur einer Variablen und Subjekte im Allgemeinen selektiver sind als Prädikate.

(2) **Übersetzung der Algebra.** Es hat sich als äußerst wirksam erwiesen, unnötige Daten so früh wie möglich aus einer Relation zu entfernen (*"Project early and often"*). Darüber hinaus spielt die effiziente Auswertung von Joins [NW09] eine entscheidende Rolle. Hier hat sich insbesondere die Verwendung von Multi-Joins in Pig Latin bei bestimmten Anfragen bewährt, da dadurch die Anzahl der benötigten Joins reduziert werden kann. Ein Multi-Join ist dann möglich, wenn sich mehrere aufeinander folgende Joins auf die gleichen Variablen beziehen. Betrachten wir folgendes Beispiel: Angenommen es sollen drei Relationen (A, B, C) über die Variable ?x zusammengeführt werden. Normalerweise sind hierfür zwei Joins und somit zwei MapReduce-Phasen erforderlich (1). In Pig Latin lassen sich die beiden Joins allerdings zu einem Multi-Join und somit auch zu einer MapReduce-Phase zusammenfassen (2).

```
j1 = JOIN A BY x, B BY x ;          (1)
j2 = JOIN j1 BY A::x, C BY x ;
--------------------------------------
j1 = JOIN A BY x, B BY x, C BY x ; (2)
```

Abbildung 10: Multi-Join in Pig Latin

Bei der Übersetzung eines BGPs in eine Folge von Joins in Pig Latin wird die Reihenfolge der Triple-Patterns daher so angepasst, dass möglichst viele Joins zu einem Multi-Join zusammengefasst werden können, auch wenn dadurch das Selektivitäts-Kriterium aus (1) verletzt wird.

(3) **Datenmodell.** Betrachtet man eine typische SPARQL-Anfrage genauer, so sind die Prädikate in den Triple-Patterns in den meisten Fällen gebunden. Eine *vertikale Partitionierung* der RDF-Daten nach Prädikaten [Ab07] reduziert daher oftmals die Menge an RDF-Tripeln, die zur Auswertung einer Anfrage geladen werden müssen. Bei einem ungebundenen Prädikat muss allerdings weiterhin der komplette Datensatz geladen werden.

3.5 Übersetzung einer Beispiel-Anfrage

Im Folgenden wird die Übersetzung einer SPARQL-Anfrage in ein entsprechendes Pig Latin-Programm anhand eines kleinen Beispiels dargestellt. Abbildung 11 zeigt eine SPARQL-Anfrage mit dem entsprechenden Algebra-Baum. Der Baum wird von unten nach oben traversiert und in eine Folge von Pig Latin-Befehlen übersetzt (Abbildung 12), wobei eine vertikale Partitionierung der Daten unterstellt wird. Dabei werden unnötige Daten mit Hilfe von FOREACH so früh wie möglich entfernt, was die Menge an Daten reduziert, die über das Netzwerk übertragen werden müssen.

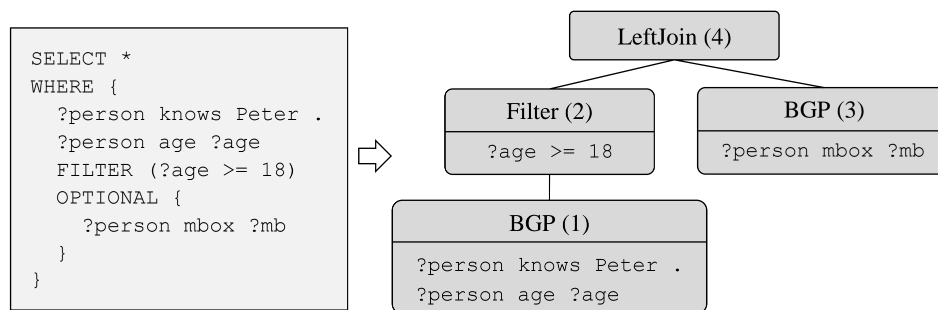

Abbildung 11: SPARQL Algebra-Baum

```
knows = LOAD 'pathToFile/knows' USING rdfLoader() AS (s,o) ;          (1)
age   = LOAD 'pathToFile/age' USING rdfLoader() AS (s,o) ;
f1    = FILTER knows BY o == 'Peter' ;
t1    = FOREACH f1 GENERATE s AS person ;
t2    = FOREACH age GENERATE s AS person, o AS age ;
j1    = JOIN t1 BY person, t2 BY person ;
BGP1 = FOREACH j1 GENERATE t1::person AS person, t2::age AS age ;

F1 = FILTER BGP1 BY age >= 18 ;                                         (2)

mbox = LOAD 'pathToFile/mbox' USING rdfLoader() AS (s,o) ;              (3)
BGP2 = FOREACH mbox GENERATE s AS person, o AS mb ;

lj  = JOIN F1 BY person LEFT OUTER, BGP2 BY person ;                    (4)
LJ1 = FOREACH lj GENERATE
      F1::person AS person, F1::age AS age, BGP2::mb AS mb ;
STORE LJ1 INTO 'pathToOutput' USING resultWriter();
```

Abbildung 12: Übersetzung des Algebra-Baums

4 Evaluation

Für die Evaluation wurden zehn Dell PowerEdge R200 Server mit jeweils einem Dual Core Intel Xeon E3120 3,16 GHz Prozessor, 4 GB DDR2 800 MHz Arbeitsspeicher, 1 TB SATA-Festplatte mit 7200 U/min und einem Dual Port Gigabit Ethernet Adapter verwendet. Die Server wurden über einen 3Com Baseline Switch 2824 zu einem Gigabit-Netzwerk zusammengeschaltet und auf den Servern wurde Ubuntu 9.10 Server (x86_64), Java in der Version 1.6.0_15 und Cloudera's Distribution for Hadoop 3 (CDH3)[6] installiert. Zum Zeitpunkt der Evaluation beinhaltete CDH3 unter anderem Hadoop in der Version 0.20.2 sowie Pig in der Version 0.5.0. Insgesamt standen knapp 8 TB an Festplattenspeicher zur Verfügung, was bei einem Replikationsfaktor von drei des verteilten Dateisystems von Hadoop (HDFS) ungefähr 2,5 TB an Nutzdaten entspricht.

Als Kennzahlen wurden neben der *Ausführungszeit* einer Anfrage auch die Menge an Daten ermittelt, die aus dem HDFS gelesen (*HDFS Bytes Read*), in das HDFS geschrieben (*HDFS Bytes Written*) sowie von den Mappern zu den Reducern übertragen (*Reduce Shuffle Bytes*) wurden. Für die Evaluation wurde der SP²Bench [Sc09] verwendet, ein SPARQL spezifischer Performance Benchmark. Der Datengenerator des SP²Bench erlaubt das Erzeugen beliebig großer RDF-Dateien auf der Grundlage der DBLP-Bibliothek von Michael Ley [Le10]. Er berücksichtigt dabei insbesondere die charakteristischen Eigenschaften und Verteilungen eines DBLP-Datensatzes und liefert somit ein realistisches Datenmodell. Im Folgenden wird die Auswertung von zwei charakteristischen Anfragen des SP²Bench präsentiert. Weitere Evaluationsergebnisse finden sich in der vollständigen Ausarbeitung [Sc10].

```
SELECT ?article
WHERE {
  ?article rdf:type bench:Article . ?article ?property ?value
  FILTER (?property = swrc:pages)
}
```

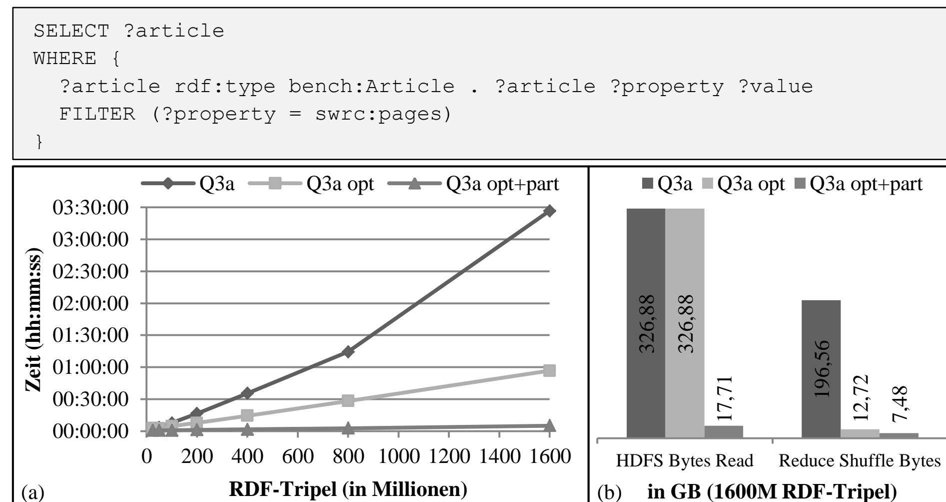

Abbildung 13: Auswertung von Q3a

Abbildung 13 zeigt die Auswertung von Q3a des SP²Bench. Die Anfrage benötigt zur Auswertung zwar nur einen Join, dieser berechnet aber sehr viele Zwischenergebnisse,

[6] siehe [http://www.cloudera.com/hadoop/]

da alle RDF-Tripel in der Eingabe das zweite Triple-Pattern erfüllen. Das spiegelt sich auch in den erzeugten Reduce Shuffle Bytes wieder, die bei der Berechnung des Joins anfallen. Da die Filter-Variable `?property` allerdings nicht in der Ausgabe enthalten sein soll, lässt sich die Anfrage auf Algebra-Ebene durch eine Filter-Substitution optimieren. Dabei wird die Variable durch ihren entsprechenden Filter-Wert ersetzt, wodurch der ursprüngliche Filter überflüssig wird. Durch diese Optimierung lässt sich die Ausführungszeit der Anfrage (a) beim größten Datensatz um über 70% verringern (Q3a opt), was auf eine signifikante Reduktion der Reduce Shuffle Bytes (b) zurückzuführen ist. Ein positiver Nebeneffekt der Optimierung ist die Eliminierung des ungebundenen Prädikats im zweiten Triple-Pattern, wovon insbesondere die Auswertung auf einem vertikal partitionierten Datensatz profitiert (Q3a opt+part). Da dadurch nur noch die beiden Prädikate `rdf:type` und `swrc:pages` betrachtet werden müssen, wird die Menge der Daten, die aus dem HDFS gelesen werden, deutlich reduziert. Durch Anwendung der Filter-Optimierung und der vertikalen Partitionierung lässt sich die Ausführungszeit der Anfrage auf dem größten Datensatz somit insgesamt um über 97% reduzieren.

```
SELECT *
WHERE {
  ?inproc rdf:type bench:Inproceedings . ?inproc dc:creator ?author .
  ?inproc bench:booktitle ?booktitle . ?inproc dc:title ?title .
  ?inproc dcterms:partOf ?proc . ?inproc rdfs:seeAlso ?ee .
  ?inproc swrc:pages ?page . ?inproc foaf:homepage ?url .
  ?inproc dcterms:issued ?yr
  OPTIONAL { ?inproc bench:abstract ?abstract }
} ORDER BY ?yr
```

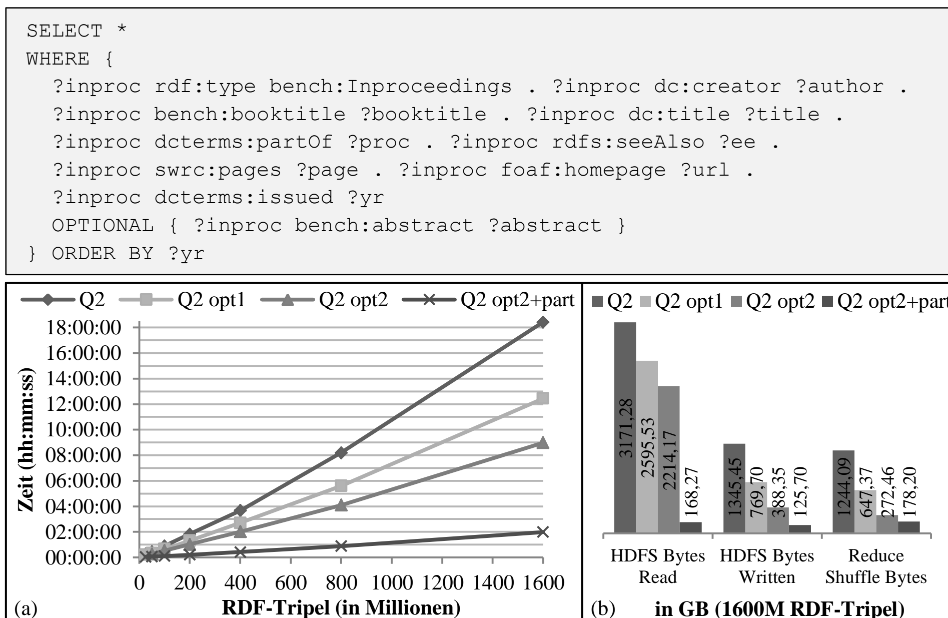

Abbildung 14: Auswertung von Q2

Abbildung 14 zeigt die Auswertung von Q2 des SP^2Bench. Dabei handelt es sich um eine komplexe Anfrage, die viele Joins erfordert und zudem ein OPTIONAL enthält. Die Ergebnisse sollen darüber hinaus in sortierter Reihenfolge ausgegeben werden. Das linke BGP der Anfrage besteht aus neun Triple-Patterns, weshalb zur nativen Auswertung insgesamt acht Joins und ein Outer Join erforderlich sind. Durch das frühzeitige Entfernen unnötiger Spalten (Projektion, Q2 opt1) lässt sich die Ausführungszeit (a) der Anfrage auf dem größten Datensatz bereits um mehr als 30% verkürzen, was sich auch in den ermittelten Daten-Kennzahlen (b) niederschlägt. Darüber hinaus lässt sich bei der Übersetzung von Q2 die Multi-Join-Fähigkeit von Pig Latin ausnutzen. Da sich alle acht

Joins auf die Variable `?inproc` beziehen, können sie in Pig Latin zu einem einzigen Join zusammengefasst werden (Q2 opt2). Dadurch reduziert sich auch die benötigte Anzahl an MapReduce-Jobs zur Auswertung von Q2 von ursprünglich zwölf bei iterierten Joins auf fünf bei Verwendung eines Multi-Joins. Da alle Prädikate in der Anfrage gebunden sind, wirkt sich auch eine vertikale Partitionierung der Daten nach Prädikaten (Q2 opt2+part) vorteilhaft aus, was sich in den Daten-Kennzahlen ganz deutlich zeigt. Durch die Anwendung aller Optimierungen lässt sich die Ausführungszeit der Anfrage auf dem größten Datensatz folglich insgesamt um fast 90% reduzieren.

4.1 Erkenntnisse der Evaluation

Durch die Evaluation konnte die ursprüngliche Vermutung bestätigt werden, dass die Ausführungszeit einer Anfrage stark mit dem erzeugten Datenaufkommen korreliert, was in den betrachteten Kennzahlen deutlich zum Ausdruck kommt. Darüber hinaus konnte auch die Wirksamkeit der untersuchten Optimierungen bestätigt werden, die einen großen Einfluss auf die Ausführungszeiten hatten, was primär auf eine Reduktion des erzeugten Datenaufkommens zurückgeführt werden konnte. Ermutigend für zukünftige Weiterentwicklungen des gewählten Ansatzes zur Auswertung von SPARQL-Anfragen sind auch die beobachteten, linearen Skalierungen der Ausführungszeiten sowie die auf Anhieb erreichten Datensatz-Größen von bis zu 1600 Millionen RDF-Tripeln, selbst ohne vertikale Partitionierung der Daten. Es ist anzunehmen, dass durch die vertikale Partitionierung und eine feinere Optimierung des Hadoop-Clusters dieser Wert noch gesteigert werden kann, von einer Vergrößerung des Clusters ganz abgesehen.

In [Hu10] wird ebenfalls die Auswertung von SPARQL-Anfragen mit Hadoop betrachtet, wobei im Gegensatz zu unserem Ansatz eine Anfrage direkt in eine Folge von MapReduce-Jobs übersetzt wird. Dabei werden auch Evaluations-Ergebnisse für die SP^2Bench-Anfragen Q1, Q2 und Q3a mit unterschiedlichen Datensatz-Größen gezeigt, die auf einem Hadoop-Cluster aus zehn Knoten erzielt wurden, das unserem Cluster sehr ähnlich ist. Ein Vergleich der Ergebnisse zeigt, dass die beiden Ansätze eine ähnliche Performance aufweisen, wobei unser Ansatz bei Q3a um bis zu 40% bessere Werte erzielt, was wahrscheinlich auf die Optimierung des enthaltenen Filters zurückzuführen ist. Das zeigt, dass unser Ansatz einer Übersetzung von SPARQL nach Pig Latin eine effiziente Auswertung ermöglicht, die mit der Performance einer direkten Auswertung in MapReduce mithalten kann und zudem von der schnellen Weiterentwicklung von Pig profitiert [Ga09].

5 Verwandte Arbeiten

Die Übersetzung von Anfrage-Sprachen in andere Sprach-Konstrukte ist eine übliche Vorgehensweise, insbesondere im Bereich der relationalen Algebra [Bo05, Cy05]. Da die Semantik von Pig Latin stark an der relationalen Algebra orientiert ist, treten bei der Übersetzung von SPARQL nach Pig Latin die gleichen Probleme mit NULL-Werten bei Joins auf, wie sie auch bei der Übersetzung von SPARQL in die relationale Algebra zu finden sind [Cy05]. In [MT08] wurde bereits von einer Übersetzung von SPARQL nach

Pig Latin berichtet, zu der allerdings keine genaueren Angaben gemacht wurden. Nach unserem Kenntnisstand ist die in diesem Beitrag beschriebene Übersetzung die erste vollständige und detaillierte Darstellung einer Übersetzung von SPARQL nach Pig Latin, die darüber hinaus auch effiziente Optimierungen betrachtet und mit einem SPARQL spezifischen Benchmark evaluiert wurde.

Die Auswertung von SPARQL-Anfragen spielt im Bereich des semantischen Webs eine wichtige Rolle. Sesame [BKH02], Jena [Mc01], RDF-3X [NW08] und 3store [HG03] sind in diesem Zusammenhang bekannte Beispiele für die Auswertung von SPARQL-Anfragen auf Einzelplatz-Systemen. Mit der Zunahme an verfügbaren semantischen Daten[7] rückt auch die Auswertung von großen RDF-Datensätzen zunehmend in den Blickpunkt wissenschaftlicher Forschung. [HBF09] und [QL08] betrachten in diesem Zusammenhang die Auswertung von SPARQL-Anfragen auf mehreren verteilten RDF-Datensätzen. Die meisten Ansätze zur Verwaltung und Auswertung von sehr großen RDF-Datensätzen mit mehreren Milliarden RDF-Tripeln setzen auf den Einsatz von Computer-Clustern. [Hu10] und [MYL10] befassen sich ebenfalls mit der Auswertung von SPARQL-Anfragen in einem MapReduce-Cluster. Im Gegensatz zu dem von uns vorgestellten Ansatz wird eine SPARQL-Anfrage dabei direkt in eine Folge von MapReduce-Jobs übersetzt, wobei allerdings größtenteils nur Basic Graph Patterns unterstützt werden. Die von uns vorgestellte Übersetzung nach Pig Latin unterstützt hingegen alle Operatoren der SPARQL Algebra (mit Ausnahme des GRAPH-Operators) und profitiert zudem von Optimierungen und Weiterentwicklungen von Pig. SHARD [RS10] ist ein RDF-Triple-Store für Hadoop, der auch SPARQL-Anfragen unterstützt. Experimentelle Ergebnisse liegen allerdings nur für den allgemeinen LUBM-Benchmark [GPH05] vor, der wichtige Eigenschaften von SPARQL-Anfragen nicht berücksichtigt. Die Autoren machen hier leider auch keine genauen Angaben zum unterstützen Sprachumfang. SPIDER [Ch09] verwendet HBase zur Speicherung von RDF-Daten in Hadoop in Form von flachen Tabellen und unterstützt auch grundlegende SPARQL-Anfragen, wobei auch hier keine genaueren Angaben zum unterstützten Sprachumfang gemacht werden. In [RDA10] wird die Verwendung von UDFs zur Reduzierung der I/O-Kosten bei der Auswertung von analytischen Anfragen auf RDF-Graphen mit Pig Latin untersucht. Dabei wurde gezeigt, dass durch die Verwendung spezieller UDFs die I/O-Kosten in einigen Situationen gesenkt werden können, weshalb sich eine Übertragung des Ansatzes auf die hier vorgestellte Übersetzung als vorteilhaft erweisen könnte.

Neben dem Einsatz eines allgemeinen MapReduce-Clusters setzen einige Systeme auch auf spezialisierte Computer-Cluster. Virtuoso Cluster Edition [Er10] und Clustered TDB [Ow09] sind Cluster-Erweiterungen der bekannten Virtuoso und Jena RDF-Stores. 4store [HLS09] ist ein einsatzbereiter RDF-Store, bei dem das Cluster in Storage und Processing Nodes unterteilt wird. YARS2 [Ha07] setzt auf die Verwendung von Indexen zur Anfrage-Auswertung und MARVIN [Or10] verwendet einen Peer-to-Peer-Ansatz zur verteilten Berechnung von RDF Reasoning. Die Verwendung von spezialisierten Clustern hat allerdings den Nachteil, dass hierfür eine eigene Infrastruktur aufgebaut werden muss, wohingegen unser Ansatz auf der Verwendung eines allgemeinen Clusters beruht, das für verschiedene Zwecke verwendet werden kann.

---

[7] siehe [http://esw.w3.org/TaskForces/CommunityProjects/LinkingOpenData/DataSets/Statistics]

6 Zusammenfassung

In diesem Beitrag wird ein neuer Ansatz zur effizienten Auswertung von SPARQL-Anfragen auf großen RDF-Datensätzen unter Verwendung des Hadoop MapReduce-Frameworks vorgestellt. Dazu wurde für schwach wohlgeformte SPARQL-Anfragen eine Übersetzung nach Pig Latin entwickelt und implementiert. Schwach wohlgeformte Anfragen sind ein ausdrucksmächtiges Fragment von SPARQL, die in der Praxis sehr häufig vorkommen. Es wurden für die Operatoren der SPARQL Algebra entsprechende Übersetzungsvorschriften entwickelt und eine Abbildung des RDF-Datenmodells in das Datenmodell von Pig definiert. Das resultierende Pig Latin-Programm wird von Pig, der Implementierung von Pig Latin für Hadoop, in eine Folge von MapReduce-Jobs überführt und verteilt auf einem Hadoop-Cluster ausgeführt. Die Evaluationsergebnisse haben gezeigt, dass die Verwendung von Pig Latin ein geeigneter und effizienter Ansatz zur Auswertung von SPARQL-Anfragen auf großen RDF-Datensätzen mit Hadoop ist. Dabei konnte der vermutete Zusammenhang zwischen der Ausführungszeit einer Anfrage und dem von der Anfrage erzeugten Datenaufkommen bestätigt werden. Besonders deutlich wurde dies durch die untersuchten Optimierungen, mit deren Hilfe die Ausführungszeit einer Anfrage teilweise deutlich reduziert werden konnte. Die verwendeten Datensatz-Größen von bis zu 1600 Millionen RDF-Tripeln übertreffen die Möglichkeiten von Systemen, die nur auf einem Computer ausgeführt werden, bereits um ein Vielfaches, was der Vergleich in [Sc09] belegt. Bedenkt man die Tatsache, dass für die Evaluation nur ein kleiner Hadoop-Cluster zum Einsatz kam (Yahoo! betreibt z.B. einen Hadoop-Cluster mit mehreren tausend Computern) und Pig sich in einer relativ frühen Entwicklungsphase befunden hat (die Evaluation wurde mit Pig 0.5.0 durchgeführt), wird das Potential des Ansatzes deutlich. Die vorgestellte Übersetzung bietet somit eine einfache und zugleich effiziente Möglichkeit, die Leistungsfähigkeit eines Hadoop-Clusters zur verteilten und parallelisierten Auswertung von SPARQL-Anfragen auf großen RDF-Datensätzen zu nutzen.

Literaturverzeichnis

[Ab07] Abadi, D. J. et al.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: Proc. VLDB 2007; S. 411-422.

[Ap10] Apache Hadoop: Pig Latin Reference Manual. 2010. http://hadoop.apache.org/pig/docs/

[BHL01] Berners-Lee, T.; Hendler, J.; Lassila, O.: The Semantic Web. In: Sc. American, 2001.

[BKH02] Broekstra, J.; Kampman, A.; Harmelen, F. van: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: Proc. ISWC 2002; S. 54-68.

[Bo05] Boncz, P. et al.: Pathfinder: XQuery – The Relational Way. In: Proc. VLDB 2005; S. 1322-1325.

[Ch09] Choi, H. et al.: SPIDER: A System for Scalable, Parallel/Distributed Evaluation of large-scale RDF Data. In: Proc. CIKM 2009; S. 2087-2088.

[Cy05] Cyganiak, R.: A relational algebra for SPARQL. TR. HP Laboratories Bristol. 2005.

[DG04] Dean, J.; Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Communications of the ACM, Vol. 51, Nr. 1, 2008; S. 107-113.

[Er10] Erling, O.: Towards Web Scale RDF. In: SSWS, Karlsruhe, Germany, 2008.

[Ga09] Gates, A. F. et al.: Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience. In: VLDB Endow., Vol. 2, Nr.2, 2009; S. 1414-1425.

[GPH05] Gui, Y.; Pan, Z.; Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. In: J. Web Sem., Vol. 3, Nr. 2-3, 2005; S. 158-182.
[Ha07] Harth, A. et al.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In: Proc. ISWC/ASWC 2007, Vol. 4825; S. 211-224.
[HBF09] Hartig, O.; Bizer, C.; Freytag, J.: Executing SPARQL Queries over the Web of Linked Data. In: Proc. ISWC 2009; S. 293-309.
[HG03] Harris, S.; Gibbins, N.: 3store: Efficient Bulk RDF Storage. In: Proc. PSSS 2003; S. 1-20
[HH07] Hartig, O.; Heese, R.: The SPARQL Query Graph Model for Query Optimization. In: ESWC 2007, Vol. 4519; S. 564-578.
[HLS09] Harris, S.; Lamb, N.; Shadbolt, N.: 4store: The Design and Implementation of a Clustered RDF Store. In: Proc. SSWS 2009; S. 94-109.
[Hu10] Husain, M. F. et al.: Data Intensive Query Processing for Large RDF Graphs Using Cloud Computing Tools. In: IEEE 3rd International Conference on Cloud Computing (CLOUD) 2010; S. 1-10.
[Le10] DBLP Bibliography, 2010. http://www.informatik.uni-trier.de/~ley/db/
[Mc01] McBride, B.: Jena: Implementing the RDF Model and Syntax Specification. In: Proc. of the 2nd International Workshop on the Semantic Web, 2001.
[MMM04] Manola, F.; Miller, E.; McBride, B.: RDF Primer. World Wide Web Consortium (W3C), 2004. http://www.w3.org/TR/rdf-primer/
[MT08] Mika, P.; Tummarello, G.: Web Semantics in the Clouds. In: IEEE Intelligent Systems, Vol. 23, Nr. 5, 2008; S. 82-87.
[MYL10] Myung J.; Yeon J.; Lee S.: SPARQL Basic Graph Pattern Processing with Iterative MapReduce. In: Proc. MDAC 2010; S. 1-6.
[NW08] Neumann, T.; Weikum, G.: RDF-3X: a RISC-style Engine for RDF. In: Proc. VLDB Endow., Vol. 1, Nr. 1, 2008; S. 647-659.
[NW09] Neumann, T.; Weikum, G.: Scalable Join Processing on Very Large RDF Graphs. In: Proc. SIGMOD 2009; S. 627-640.
[Ol08] Olston, C. et al.: Pig Latin: A Not-So-Foreign Language for Data Processing. In: Proc. SIGMOD 2008; S. 1099-1110.
[Or10] Oren, E. et al.: MARVIN: A platform for large-scale analysis of Semantic Web data. In: Proc. of the International Web Science Conference 2009.
[Ow09] Owens, A. et al.: Clustered TDB: A Clustered Triple Store for Jena. In: WWW 2009.
[PAG09] Pérez, J.; Arenas, M.; Gutierrez, C.: Semantics and Complexity of SPARQL. In: ACM Trans. Database Syst., Vol. 34, Nr. 3, 2009; S. 1-45.
[PS08] Prud'hommeaux, E.; Seaborne, A.: SPARQL Query Language for RDF. World Wide Web Consortium (W3C), 2008. http://www.w3.org/TR/rdf-sparql-query/
[QL08] Quilitz, B.; Leser, U.: Querying Distributed RDF Data Sources with SPARQL. In: Proc. ESWC 2008. LNCS, Vol. 5021; S. 524-538.
[RDA10] Ravindra, P.; Deshpande, V.; Anyanwu, K.: Towards Scalable RDF Graph Analytics on MapReduce. In: Proc. MDAC 2010; S. 1-6.
[RS10] Rohloff, K.; Schantz, R. E.: High Performance, Massively Scalable Distributed Systems using the MapReduce Software Framework: The SHARD Triple-Store.
[Sc09] Schmidt, M. et al.: SP^{2}Bench: A SPARQL Performance Benchmark. In: Proc. ICDE 2009; S. 222-233.
[Sc10] Schätzle, A.: PigSPARQL: Eine Übersetzung von SPARQL nach Pig Latin, Masterarbeit. Lehrstuhl für Datenbanken und Informationssysteme, Institut für Informatik, Albert-Ludwigs-Universität Freiburg. 2010.
[SML10] Schmidt, M.; Meier, M.; Lausen, G.: Foundations of SPARQL query optimization. In: Proc. ICDT 2010; S. 4-33.
[St08] Stocker, M. et al.: SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In: Proc. WWW 2008; S. 595-604.