B. Mitschang et al. (Hrsg.): BTW 2017 – Workshopband,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 385

Vergleich und Evaluation von RDF-on-Hadoop-Lösungen

Wolfgang Amann[1]

Abstract: Mit der steigenden Anzahl von Daten, welche in Form des Resource Description Framework (RDF) veröffentlicht werden entsteht eine Menge von Daten, bei der Datenoperationen nicht mehr von einem einzelnen Rechner zu bewältigen sind. In dieser Arbeit werden Systeme vorgestellt, welche zur Lösung dieses Problems das Hadoop-Framework ausschließlich bzw. in Kombination mit anderen Big-Data-Frameworks nutzen. Danach werden mit PigSPARQL und Rya zwei dieser Ansätze, welche exemplarisch für die neuere Entwicklung dieser RDF-on-Hadoop-Systeme stehen, anhand der Benchmark-Queries der Waterloo SPARQL Diversity Test Suite auf spezifische Stärken und Schwächen analysiert.

Keywords: Semantic Web, Resource Description Framework, SPARQL, Big Data, Benchmarking.

1 Einleitung

Seit den 1990/2000er Jahren schreitet die Digitalisierung der Gesellschaft mit großen Schritten voran – durch die stetige Digitalisierung von Informations- und Kommunikationsprozessen wurde die Wissens- und Informationsproduktion immer weiter beschleunigt, was einen starken Anstieg der Daten, die elektronisch gespeichert werden, zur Folge hat. Viele dieser Informationen werden ohne formale Struktur im Internet angeboten, sodass sie von Maschinen nur schwer zu interpretieren sind. 2001 stellte Tim Berners-Lee als Lösung seine Idee des *Semantic Web* vor, wodurch Computern mithilfe des einheitlichen Datenmodells *Resource Description Framework (RDF)* das einfache automatische Ableiten von Wissen aus dem Internet ermöglicht werden soll. Die Idee des Semantic Web wird zunehmend durch ein breiteres Publikum angenommen, wodurch eine Menge an Daten entsteht, bei der Datenoperationen nicht mehr von einem einzelnen Rechner zu bewältigen sind. Eine Lösung dieses Problems bieten die in den letzten Jahren vermehrt entstanden Ansätze, welche mithilfe des Big-Data-Frameworks Hadoop die Arbeit auf skalierbare Rechnernetze auslagern. Ziel dieser Arbeit ist es, einen Überblick über diese bestehenden Ansätze hinsichtlich ihrer Eigenschaften zu schaffen und zwei Systeme zu evaluieren, welche die neueren Entwicklungen der RDF-on-Hadoop-Systeme widerspiegeln.

2 Funktionaler Vergleich von RDF-on-Hadoop-Lösungen

Da die verteilte Speicherung und Verarbeitung von RDF-Daten ein noch relativ junges Anliegen ist, gibt es anders als bei zentralisierten RDF-Stores keine etablierten Systeme

[1] Universität Leipzig, Big Data Kompetenzzentrum, Ritterstraße 9-13, 04109 Leipzig, amann@studserv.uni-leipzig.de

wie Apache Jena, Sesame oder Virtuoso. Vielmehr gibt es eine Vielzahl von forschungsorientierten Machbarkeitsstudien, die entweder neue Lösungen vorstellen oder versuchen, durch Kombination verschiedener bestehenden Ansätzen Laufzeiten zu optimieren. Analog zu [GFM14] wird eine Klassifizierung in *native* und *hybride* Systeme vorgeschlagen:

Native Lösungen verwenden ausschließlich MapReduce gemeinsam mit HDFS bzw. HBase zur Bearbeitung von SPARQL-Anfragen. Dazu wird eine Serie von MapReduce-Jobs aufgerufen – deswegen bemühen sich Ansätze dieser Kategorie vor allem darum, physische Optimierungen vorzunehmen und Strategien zu entwickeln, um die Anzahl der benötigten MapReduce-Jobs zu verringern. Die Speicherung geschieht dabei exklusiv im HDFS, teilweise auch mithilfe von NoSQL-Datenbanken, die auf dem Hadoop-Dateisystem aufbauen, um Indexierungsstrategien auszunutzen.

Hybride Lösungen benutzen im Gegensatz dazu eine Kombination aus Hadoop mit anderen Systemen. Die hybriden Lösungen wurden in drei Unterkategorien unterteilt: (1) Ansätze, die auf den Cluster-Knoten zentralisierte RDF-Stores installieren. Dadurch können Teile von SPARQL-Queries direkt auf den jeweiligen Knoten berechnet werden, auf die der RDF-Graph aufgeteilt wird. Dies macht MapReduce-Jobs nur noch notwendig, um die Ergebnisse der einzelnen Knoten miteinander zu vereinigen. (2) Systeme, welche SPARQL-Anfragen zur Bearbeitung in andere Anfragesprachen wie Pig, Impala oder HiveQL übersetzen. Die zugehörigen Publikationen behandeln vor allem die korrekte und effiziente algebraische Übersetzung von SPARQL in andere Query-Sprachen. (3) Ansätze, die zentralisierte Query-Engines basierend auf der verteilten Speicherung im HDFS nutzen. (4) Ein Ansatz, der im Gegensatz zu allen anderen Ansätzen graph-parallele Berechnungen implementiert, wodurch mehr Rücksicht auf die Struktur von RDF-Daten genommen wird.

2.1 Native Ansätze

2.1.1 HDFS-basierte Systeme

SHARD [RS10] repräsentiert die einfachste Herangehensweise an die Verarbeitung von RDF-Daten mithilfe von Hadoop: RDF-Tripel werden zeilenorientiert direkt im HDFS gespeichert. Für die Anfrageverarbeitung wird MapReduce und der „Klausel-Iterations"-Ansatz genutzt, d. h. für jedes Tripelmuster (Klausel) einer Anfrage wird ein eigener MapReduce-Job initiiert, was zu einer hohen Anzahl sequenziell auszuführender Berechnungen führt, v. a. durch die fehlende Optimierung der Anfragen.

HadoopRDF [Hu09] will diese vielen MapReduce-Jobs vermeiden, indem der optimale physische Auswertungsplan für eine SPARQL-Anfrage durch einen Greedy-Algorithmus ausgewählt wird, welcher aufgrund der Gesamtanzahl der Variablen einer Query deren Join-Selektivitäten abschätzt, um die Anzahl der nötigen MapReduce-Jobs zu verringern. Letztendlich werden Joins in der reduce-Phase bearbeitet, was einen hohen Datenverkehr in der shuffle-Phase bedeutet.

Diesen hohen Kosten für Joins widmen sich die Systeme **[MYL10]**, **MapMerge** [Pr13] und **[LYG13]**. Bei [MYL10] geschieht dies über die Implementierung eines Mehrwege-

Joins, über den mehrere Joins zusammengelegt werden und so die MapReduce-Jobs pro Iteration verringert werden. MapMerge ist eine Adaption von Sort-Merge-Joins für die Verarbeitung von RDF im HDFS. Die Berechnung der Joins wird komplett in der map-Phase vollzogen während die reduce-Phase lediglich für die Nachbearbeitung der Join-Ergebnisse für folgende Iterationen verwendet wird. Dieses Vorgehen hat eine Reduzierung des Datenaufkommens in der shuffle-Phase auf Kosten einer verlängerten Partitionierungs- und Indexierungsphase beim Einlesen der Datensätze zur Folge. [LYG13] schlägt einerseits den „Multiple-Join-With-Filter“ vor, der den SQL-Join ersetzten soll, andererseits versucht die „Select-Join“-Primitive, Projektionen nicht als eigenständigen MapReduce-Job auszuführen, sondern in die Ausführung eines Joins zu integrieren.

[NLH08] und **CliqueSquare** [Go13] wollen das Vorgehen von SHARD über verbesserte Datenspeicherung lösen. [NLH08] erweitert dazu eine RDF-Speicherungstechnik namens „RDF molecules“, welche im Umfang zwischen Tripel und Graph einen Graphen in „Moleküle“ aufteilt, von denen jedes jeweils ein verbundener Subgraph des Originals ist. CliqueSquare nutzt die dreifache Replikation innerhalb des HDFS aus, um jedes RDF-Tripel in drei verschiedenen Formen partitioniert und gruppiert nach Subjekt, Objekt und Prädikat zu speichern. Diese Partitionen sind so gewählt, dass zur Bearbeitung der Queries durch den eingeführten clique-basierten Algorithmus ein möglichst kleiner Datenaustausch in den shuffle-Phasen der MapReduce-Jobs enstehen soll und somit „partitionierte“ Joins ausgeführt werden können, d. h. Joins, die in der map-Phase evaluiert werden.

2.1.2 NoSQL-basierte Systeme

Bei der Nutzung von NoSQL-Datenbanken zur Speicherung von RDF-Daten nutzen die jeweiligen Systeme spezifische Eigenschaften der Stores für eine beschleunigte Anfragebearbeitung aus. Die frühen Ansätze **[SJ10]** und **[Fr11]** stützen sich hierzu auf die Idee, mehrere Indextabellen mit möglichen Tripelpermutationen zu nutzen. Die Tripel selbst werden direkt im Zeilenschlüssel von HBase gespeichert. Da die Anfragebearbeitung der Systeme einfach gehalten ist, resultieren aus SPARQL-Queries zahlreiche sequentielle MapReduce-Jobs.

MAPSIN *(map-side index nested loop-join)* [Sc12] will dies durch die Join-Verarbeitung komplett in der map-Phase (ähnlich wie bei MapMerge) umgehen. Der Vorteil der Nutzung von HBase ist, dass mehrere MapReduce-Jobs aneinandergereiht werden können, ohne dass die Ausgabe für den jeweils nächsten Job sortiert werden muss. Hierdurch und durch das Zusammenlegen von Joins verschiedener Tripelmuster (v. a. bei sternförmigen Anfragen) wird überflüssiger Datenfluss über das Netzwerk vermieden. **RDFChain** [CJL13] baut direkt auf MAPSIN auf und will durch eine Kombination von drei Tabellen die namensgebenden linearen Basic Graph Patterns schnell bearbeiten. Tripelterme, die nicht nur als Subjekt, sondern auch als Objekt erscheinen, werden in einer gesonderten Tabelle abgelegt, alle anderen Tripel, die dort nicht vorkommen in den von anderen Ansätzen bekannten Subjekt-Prädikat-Objekt- und Objekt-Prädikat-Subjekt-Tabellen.

H_2RDF+ [Pa13] will die Anfragegeschwindigkeit über Heuristiken beschleunigen: Zusätzlich zu den Indextabellen legen sie aggregierte Indexstatistiken in HBase an, mithilfe deren Joinkosten und Selektivität von Tripelmustern abgeschätzt werden sollen. Auf Grundlage dieser Schätzungen wird entschieden, ob Anfragen zentralisiert auf nur einem Clusterknoten berechnet oder über MapReduce an alle Knoten verteilt werden.

Einen anderen Weg schlägt **SPIDER** *(Scalable, Parallel/Distributed Evaluation of large-scale RDF data)* [Ch09] ein. Es sortiert RDF-Tripel nach ihrer URI und teilt sie in Subgraphen auf die verschiedenen Cluster-Server auf. Queries werden nach den so generierten URI-Schlüsseln in Subqueries aufgeteilt und an die Server gesendet, welche die entsprechenden Subgraphen speichern. Da jeder Knoten nur einen Teil der RDF-Daten speichert, muss SPIDER zur Ergebnisbildung die Subquery-Ergebnisse jedes Knotens per MapReduce vereinen, was viel Rechenzeit und Netzwerkkommunikation benötigt.

2.2 Hybride Ansätze

2.2.1 Ansätze mit zentralisierten Datenbanken

[Du12] greift die Idee von SPIDER auf und will die Vorteile „traditioneller" Triple Stores mit MapReduce verbinden. Die Datensätze werden hierzu so partitioniert, dass Tripel mit dem selben Prädikat auf dem selben Knoten, auf denen jeweils ein OpenRDF-Sesame-Triplestore läuft, verteilt werden können. Zur Auffindung der Tripel wird dazu eine Hash-Map angelegt. Gestellte SPARQL-Anfragen werden ebenfalls nach dieser Hash-Map partitioniert und an die verschiedenen Knoten gesendet. Die so entstehenden Einzelergebnisse werden anschließend über MapReduce-Jobs vereinigt. Auch **GraphPartition** [HAR11] und **[LL13]** versuchen, die Netzwerkkommunikation zwischen den Knoten eines Clusters zu minimieren, indem RDF-Graphen in Subgraphen unterteilt werden, von denen jeder auf einem eigenen Knoten gespeichert wird, auf dem jeweils RDF-3X läuft. Je nach System werden die Daten unterschiedlich partitioniert – einerseits basierend auf der Entfernung im Originalgraphen, andererseits über Subjekt-Objekt-Kombinationen. Bei der Queryausführung wird dann zunächst entschieden, ob eine Anfrage innerhalb einer Partition und damit ohne MapReduce ausgeführt werden kann, oder ob mehrere Partitionen und MapReduce genutzt werden müssen.

2.2.2 Ansätze basierend auf Query-Übersetzungen

RAPID+ [RKA11] fokussiert sich wie viele native Ansätze durch die Interpretation sternförmiger Joins als Tripelgruppen auf die Reduzierung von MapReduce-Zyklen bei der Join-Verarbeitung. Dazu baut das System die Sprache von Apache Pig um eine zwischenstufliche Algebra *(Nested TripleGroup Algebra)* aus, um die Tripel effizienter bearbeiten zu können. Die Daten werden direkt aus dem HDFS gelesen.

[Ko12] und **PigSPARQL** [Sc13] übersetzen SPARQL-Anfragen nach Pig Latin, der Sprache von Apache Pig. Pig übersetzt diese Anfragen wiederum in MapReduce Jobs, welche

die Daten, die direkt im HDFS gespeichert werden oder bei PigSPARQL optional vertikal partitioniert werden, anfragen. Die Systeme können hierdurch stark von der Weiterentwicklung von Pig profitieren, ohne Optimierungen am eigenen System vornehmen zu müssen. **Hive-HBase-RDF** [Ha13], **Sempala** [Sc14] und **S2RDF** *(SPARQL on Spark for RDF)* [Sc15b] gehen einen ähnlichen Weg. Die Systeme basieren auf der Übersetzung von SPARQL nach SQL (bzw. HiveQL) und dessen anschließender Bearbeitung mit Hive, Impala oder der SQL-API von Spark. Während das erste System Tripel in HBase speichert, nutzen die beiden letzteren Systeme das spaltenorientierte HDFS-Format Parquet. Hive-HBase-RDF und Sempala können durch die Nutzung von Property Tables vor allem sternförmige SPARQL-Anfragen ohne Joins bearbeiten, die Besonderheit an S2RDF ist die neu vorgeschlagene Indexierungsstrategie namens *Extended Vertical Partitioning* (ExtVP).

2.2.3 Ansätze mit zentralisierter Query-Bearbeitung

Rya *(RDF y Accumulo)* [PCR12] und **Jena-HBase** [Kh12] speichern RDF-Tripel mithilfe von Accumulo bzw. HBase und kombinieren dies mit den zentralisierten Query-Engines von OpenRDF Sesame/Apache Jena. Im Gegensatz zu Jena-HBase nutzt Rya die lexikographische Sortierung der Zeilenschlüssel in Accumulo, wodurch einzelne Tripelmuster durch einen Scan von nur einer der drei Indextabellen beantwortet werden können. Anfragen werden in Rya durch einen geschachtelten Loop-Join zentralisiert auf einem Server bearbeitet, was besonders bei nicht-selektiven Queries einen potentiell limitierenden Faktor dieses Ansatzes hinsichtlich der Skalierung von Datensätzen darstellt.

2.2.4 Ansätze mit graph-paralleler Berechnung

S2X *(SPARQL on Spark with GraphX)* [Sc15a] nutzt einen anderen Ansatz als alle anderen vorgestellten Systeme. Mithilfe der Apache-Spark-API „GraphX“, welche Werkzeuge für Graphoperationen bereitstellt, werden graph-parallele Berechnungen auf RDF-Daten ausgeführt. Diese werden dafür auf die „Property Graph“-Struktur von GraphX abgebildet, welcher im HDFS persistiert werden kann.

3 Evaluation

Die Evaluation wurde auf einem Cluster von 16 Dell PowerEdge R320 Servern durchgeführt (ein Namenode, ein Zookeeper und 14 Worker), die jeweils mit einem Intel Xeon E5-2430 v2 (2,5GHz mit 6 Cores) und 48GB RAM ausgestattet und durch 1GBit/s Ethernet mit dem Netzwerk verbunden waren. Auf den einzelnen Rechnern war jeweils openSUSE 13.2 mit Hadoop 2.6.0, Pig 0.15, Accumulo 1.7.0 und Tomcat 7 installiert.

Es wurde exemplarisch die Leistung von hybriden Ansätzen getestet, da diese die neueren Entwicklungen in den RDF-on-Hadoop-Lösungen widerspiegeln. Die Wahl fiel auf die beiden Systeme PigSPARQL (basierend auf Query-Übersetzungen) und Rya (zentralisierte

Skalierungsfaktor	25000	50000	100000
urspr. Dateigröße	142,6 GB	287,2 GB	560,5 GB
Datenbankgröße PigSPARQL (VP)	238,4 GB	480,9 GB	939,4 GB
Datenbankgröße Rya	42,3 GB	84,1 GB	166,3 GB
Ladezeit PigSPARQL (VP)	01:17:52	02:46:27	05:56:59
Ladezeit Rya	04:16:02	07:52:23	15:13:44

Tab. 1: Ladezeiten für die Datensätze des WatDiv-Benchmarks.

Queryausführung) – auf weitere Systeme wurde verzichtet, da die Quelltexte aller Ansätze mit zentralisierten Datenbanken nicht öffentlich verfügbar sind und sich S2X im Laufe der Arbeit aufgrund großem Informationsoverhead bei der Queryausführung als noch nicht geeignet für den Big-Data-Anwendungsfall herausgestellt hat. Zur Evaluation wurde die *Waterloo SPARQL Diversity Test Suite* (WatDiv) [Al14] genutzt. Aus den 17 mitgelieferten Vorlagen für lineare, sternförmige und schneeflockenförmige Anfragen wurden jeweils fünf konkrete Anfragen erstellt, wodurch sich 85 verschiedene Queries pro Datensatz ergaben – jede davon wurde einmal auf den beiden Systemen ausgeführt und aus den Laufzeitergebnissen der Median gebildet. Beim Generieren der Daten wurden sich verdoppelnde Skalierungsfaktoren von 25000, 50000 und 100000 verwendet, was in Datensätzen von ca. 2,6 bis 10,5 Milliarden Tripeln resultierte. In Tabelle 1 sind die Dateigrößen der ursprünglichen Datensätze und die sich daraus ergebenen Ladezeiten und Datenbankgrößen dargestellt. Durch die vertikale Partitionierung muss PigSPARQL einige Daten redundant abspeichern und verbraucht so mehr Speicherplatz als der ursprüngliche Datensatz. Prinizpiell doppelt auch Rya durch seinen Three Tables Index viele Tripel – durch die in Accumulo standardmäßig integrierte gzip-Kompression wird dies aber ausgeglichen und dadurch lediglich 30% der Ausgangsgröße verbraucht. Dies kommt aber zum Preis einer sehr viel längeren Vorverarbeitungszeit: Schon beim kleinsten Datensatz braucht Rya dreimal so lang zum Laden der Daten wie PigSPARQL.

3.1 Vergleich der Laufzeiten

L1–L5 Bei PigSPARQL spiegelt sich der niedrige Selektivitätsgrad der fünf linearen Anfragen direkt in den Laufzeiten nieder: Sie liegen konstant sehr niedrig, selbst beim größten Datensatz war die rechenaufwändigste Query L1 in 2:40 Minuten zu beantworten (siehe Abbildung 1). Bei den Tests mit Rya war eine deutlich langsamere Anfragezeit zu beobachten – L3 brauchte beim kleinsten Datensatz 12 Stunden und 20 Minuten (im Vergleich zu 59 Sekunden bei PigSPARQL), mit Skalierungsfaktor 50000 lag die Bearbeitungszeit bereits über der gesetzten Maximalzeit von 24 Stunden. Der Grund hierfür ist die fehlende algebraische Optimierung der SPARQL-Anfragen durch Rya. Diese wird nur durchgeführt, wenn ein zeit- und speicherintensiver Prozess namens *Prospector* aufgerufen wird, welcher die Selektivität von Tripelmustern eines Datensatzes abzählt. Die Optimierung wäre auch ohne diesen Prozess möglich, was besonders bei Anfrage L3 gut zu beobachten ist: PigSPARQL implementiert zur Optimierung u. a. eine sehr einfache Strategie zur

Laufzeit namens *Variable-Counting*-Heuristik, welche Tripelmuster mit weniger ungebundenen Bestandteilen in einer Anfrage möglichst weit nach oben verschiebt, um früh Zwischenergebnisse zu verkleinern. Bei Eingabe dieser durch PigSPARQL optimierten Tripelanordnung in Rya kann man eine starke Verringerung der Laufzeit beobachten: Mit 0,4 Sekunden Bearbeitungszeit beim größten Datensatz liegt Rya plötzlich deutlich vor PigSPARQL.

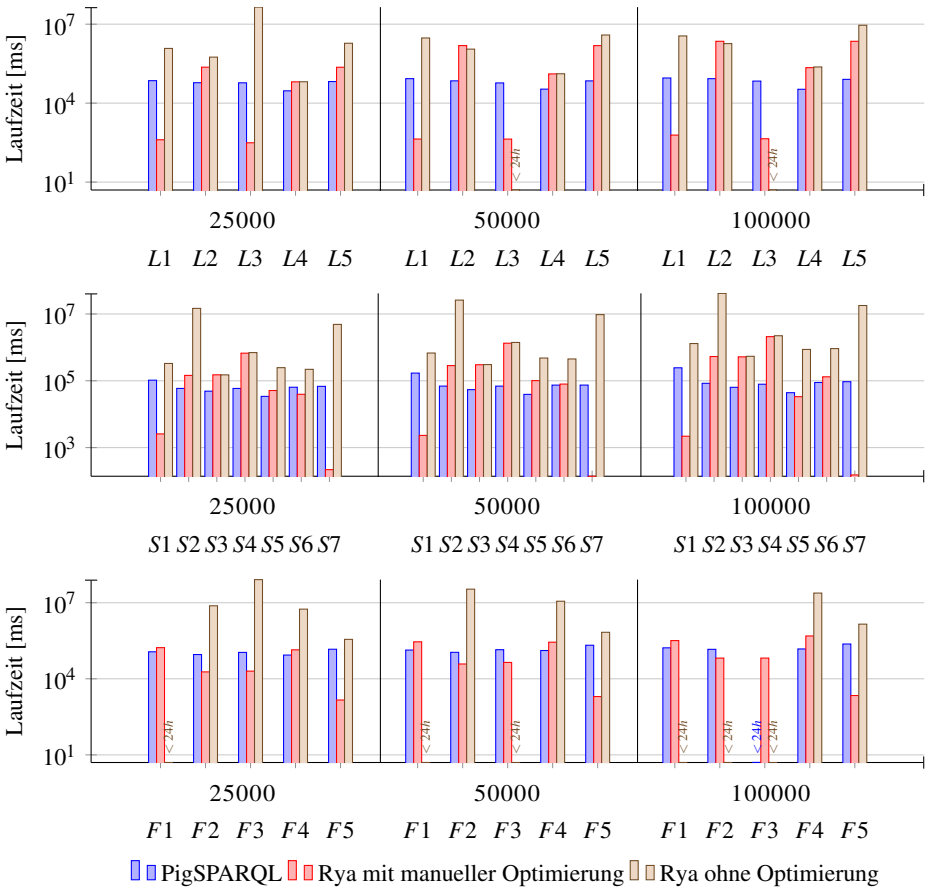

Abb. 1: Laufzeiten der getesteten WatDiv-Anfragen mit Skalierung.

F1–F5 Durch die Kombination von linearen mit sternförmigen Anfragen kommt Rya bei den schneeflockenförmigen Queries an seine Grenzen: Bereits beim kleinsten Datensatz benötigt das System mehr als 24 Stunden zur Beantwortung von F1. Hier macht sich die Vermeidung von Kreuzprodukten durch die Optimierung von PigSPARQL gemeinsam mit der Mehrwege-Join-Unterstützung von Pig stark bemerkbar (zwei Joins im Gegensatz zu ursprünglich fünf). F4 gestaltet sich ähnlich: PigSPARQL benötigt zur Bearbeitung des schneeflockenförmigen Musters zwei im Vergleich zu acht Joins bei Rya. Durch eine manuelle Neuordnung der Anfragen im Stile der Variable-Counting-Heuristik kann man die Laufzeiten der Anfragen durch Vermeidung von Kreuzprodukten nah an das Niveau

von PigSPARQL heran bringen, welches aber nicht ganz erreicht werden kann. F2, F3 und F5 werden von Rya ebenfalls langsamer bearbeitet als von PigSPARQL. Die drei Anfragen können im Gegensatz zu F1 und F2 aber so neu geordnet werden, sodass Rya schnellere Laufzeiten erreichen kann als PigSPARQL. Obwohl PigSPARQL hier wieder sieben Joins zu lediglich zwei Joins zusammenfassen kann, reicht die Neuordnung aus, dass Rya bei allen Datensatzgrößen schneller als PigSPARQL rechnet. Bei F3 und F5 erhält man durch ähnliche Verschiebungen ebenfalls bessere Laufzeiten als PigSPARQL.

S1–S7 Ohne manuelle Optimierung der Anfragen generiert Rya bei den sternförmigen Anfragen viel längere Laufzeiten als PigSPARQL. Vor allem bei S1 und S7 können durch aufsteigende Sortierung der Tripelmuster nach ungebundenen Elementen und durch Vermeidung von Kreuzprodukten ähnlich wie bei L1 und L3 stark beschleunigte Ergebnisse von Rya erreicht werden. Mit diesen kleinen Zwischenergebnissen kann Rya dank seines Three Tables Index die Ergebnisse in nahezu konstant bleibender Zeit berechnen. Bei den restlichen Queries ist PigSPARQL durch seine Integration eines Mehrwege-Joins eindeutig überlegen. Bei S2, S3 (ohne weitere statistische Informationen nicht optimierbar), S5 und S6 benötigt das System dadurch lediglich einen Join im Gegensatz zu jeweils einem Join pro Tripelmuster bei Rya und kann dadurch viel schnellere Laufzeiten erzielen.

4 Fazit

Mit der steigenden Beliebtheit von RDF geht auch ein größeres Interesse der akademischen Forschung einher. Während 2008/09 noch sehr wenige Paper zu RDF-on-Hadoop-Systemen veröffentlicht wurden, nimmt die Rate seit sechs Jahren zu. Bei der Evaluation der zwei exemplarisch ausgewählten Systeme PigSPARQL und Rya erlangte ersteres durch seine effiziente physische und logische Optimierung durchweg sehr gute Laufzeiten bei WatDiv, besonders bei komplexeren Queries, welche große Zwischenergebnisse mit sich bringen. Sobald Queries jedoch sehr selektiv sind, ist PigSPARQL sehr viel langsamer als der Konkurrent Rya – denn der größte geschwindigkeitshemmendere Faktor ist nicht die zentralisierte Anfrageauswertung, sondern die sehr rudimentäre SPARQL-Optimierung durch Rya. Das System verlässt sich vollkommen auf den sehr zeit- und speicherintensiven Prozess namens Prospector. Während dies zwar zur präzisesten Neuordnung der Tripelmuster einer SPARQL-Anfrage führt, reichen bereits sehr einfach zur Laufzeit ausführbare Strategien wie die Variable-Count-Heuristik aus, um sehr viel schnellere Ergebnisse zu erzielen. Daneben ist die Prospector-Optimierung nicht ausreichend zur vollständigen Optimierung von Anfragen – grundlegende Maßnahmen wie Filter-Substitutionen werden komplett ignoriert. Wie die Entwickler von Rya angekündigt haben, ist geplant, einen MapReduce-Job zur Bearbeitung von Queries zu implementieren – hier würde es sich anbieten, ähnlich wie H_2RDF+ auf Grundlage der durch den Prospector evaluierten Selektivitäten abzuschätzen, ob eine Anfrage schneller zentralisiert oder über den Cluster verteilt beantwortet werden kann. Als abschließende Erkenntnis dieser Arbeit bleibt die beeindruckende Fülle an angebotenen Kombinationen der RDF-on-Hadoop-Lösungen. Fast jede mögliche Kombination von Zentralisierung und Verteilung der Datenspeicherung und Anfrageausführung wurde bereits behandelt. Nach zwölf Jahren RDF sind die RDF-on-Hadoop-Anwendungen aber immer noch weit davon entfernt, für produktive Anwendungen geeignet zu sein.

5 Danksagung

Die vorliegende Arbeit wurde betreut durch Dr. Eric Peukert und gefördert durch das Bundesministerium für Bildung und Forschung innerhalb des *Competence Center for Scalable Data Services and Solutions* (ScaDS) Dresden/Leipzig (BMBF 01IS14014B).

Literatur

[Al14] Aluç, G.; Hartig, O.; Özsu, T.; Daudjee, K.: Diversified Stress Testing of RDF Data Management Systems. In: Proceedings of the 13th ISWC. Springer, Cham, S. 197–212, 2014.

[Ch09] Choi, H.; Son, J.; Cho, Y.; Sung, M. K.; Chung, Y. D.: SPIDER: A System for Scalable, Parallel / Distributed Evaluation of large-scale RDF Data. In: Proceedings of the 18th CIKM. ACM, New York, 2009.

[CJL13] Choi, P.; Jung, J.; Lee, K.-H.: RDFChain: Chain Centric Storage for Scalable Join Processing of RDF Graphs using MapReduce and HBase. In: Proceedings of the 2013 ISWC - P&D Track. 2013.

[Du12] Du, J.-H.; Wang, H.-F.; Ni, Y.; Yu, Y.: HadoopRDF: A Scalable Semantic Data Analytical Engine. In: Intelligent Computing Theories and Applications. Berlin: Springer, 2012.

[Fr11] Franke, C.; Morin, S.; Chebotko, A.; Abraham, J.; Brazier, P.: Distributed semantic web data management in HBase and MySQL cluster. In: Proceedings of the 2011 CloudCom. S. 105–112, 2011.

[GFM14] Giménez-García, J. M.; Fernández, J. D.; Martínez-Prieto, M. A.: MapReduce-based Solutions for Scalable SPARQL Querying. Open Journal of Semantic Web 1/, 2014.

[Go13] Goasdoué, F.; Kaoudi, Z.; Manolescu, I.; Quiané-Ruiz, J.; Zampetakis, S.: CliqueSquare: efficient Hadoop-based RDF query processing. In: Proceedings of the 2013 BDA. Nantes, 2013.

[Ha13] Haque, A.: A MapReduce Approach to NoSQL RDF Databases, Bachelorarbeit, University of Texas, 2013.

[HAR11] Huang, J.; Abadi, D. J.; Ren, K.: Scalable SPARQL Querying of Large RDF Graphs. Proceedings of the VLDB Endowment 4/11, S. 1123–1134, 2011.

[Hu09] Husain, M. F.; Doshi, P.; Khan, L.; Thuraisingham, B.: Storage and Retrieval of Large RDF Graph Using Hadoop and MapReduce. In: Cloud Computing. Springer, Berlin, S. 680–689, 2009.

[Kh12] Khadilkar, V.; Kantarcioglu, M.; Thuraisingham, B.; Castagna, P.: Jena-HBase: A Distributed, Scalable and Efficient RDF Triple Store. In: ISWC-PD 2012. CEUR-WS.org, Aachen, S. 85–88, 2012.

[Ko12] Kotoulas, S.; Urbani, J.; Boncz, P.; Mika, P.: Robust Runtime Optimization and Skew-resistant Execution of Analytical SPARQL Queries on Pig. In: Proceedings of the 11th ISWC. Springer, Berlin, S. 247–262, 2012.

[LL13] Lee, K.; Liu, L.: Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. Proceedings of the VLDB Endowment 6/14, S. 1894–1905, 2013.

[LYG13] Liu, L.; Yin, J.; Gao, L.: Efficient Social Network Data Query Processing on MapReduce. In: Proceedings of the 5th ACM Workshop on HotPlanet. ACM, New York, S. 27–32, 2013.

[MYL10] Myung, J.; Yeon, J.; Lee, S.: SPARQL Basic Graph Pattern Processing with Iterative MapReduce. In: Proceedings of the 2010 MDAC. ACM, New York, 6:1–6:6, 2010.

[NLH08] Newman, A.; Li, Y.-F.; Hunter, J.: Scalable Semantics - The Silver Lining of Cloud Computing. In: Proceedings of the 4th eScience. S. 111–118, 2008.

[Pa13] Papailiou, N.; Konstantinou, I.; Tsoumakos, D.; Karras, P.; Koziris, N.: H_2RDF+: High-performance distributed joins over large-scale RDF graphs. In: Proceedings of the 2013 IEEE Big Data Conference. S. 255–263, 2013.

[PCR12] Punnoose, R.; Crainiceanu, A.; Rapp, D.: Rya: A Scalable RDF Triple Store for the Clouds. In: Proceedings of the 1st Cloud-I. ACM, New York, 4:1–4:8, 2012.

[Pr13] Przyjaciel-Zablocki, M.; Schätzle, A.; Skaley, E.; Hornung, T.; Lausen, G.: Map-Side Merge Joins for Scalable SPARQL BGP Processing. In: Proceedings of the 5th CloudCom. 2013.

[RKA11] Ravindra, P.; Kim, H.; Anyanwu, K.: An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. In: Proceedings of the 8th ESWC 2011. S. 46–61, 2011.

[RS10] Rohloff, K.; Schantz, R. E.: High-performance, Massively Scalable Distributed Systems Using the MapReduce Software Framework: The SHARD Triple-store. In: PSI EtA 2010. ACM, New York, 4:1–4:5, 2010.

[Sc12] Schätzle, A.; Przyjaciel-Zablocki, M.; Dorner, C.; Hornung, T.; Lausen, G.: Cascading Map-Side Joins over HBase for Scalable Join Processing, 2012, URL: https://arxiv.org/pdf/1206.6293v1, Stand: 12. 12. 2016.

[Sc13] Schätzle, A.; Przyjaciel-Zablocki, M.; Hornung, T.; Lausen, G.: PigSPARQL: A SPARQL Query Processing Baseline for Big Data. In: Proceedings of the 2013 ISWC. CEUR-WS.org, Aachen, S. 241–244, 2013.

[Sc14] Schätzle, A.; Przyjaciel-Zablocki, M.; Neu, A.; Lausen, G.: Sempala: Interactive SPARQL Query Processing on Hadoop. In: Proceedings of the 13th ISWC. Springer, New York, S. 164–179, 2014.

[Sc15a] Schätzle, A.; Przyjaciel-Zablocki, M.; Berberich, T.; Lausen, G.: S2X: Graph-Parallel Querying of RDF with GraphX. In: Big-O(Q) 2015. 2015.

[Sc15b] Schätzle, A.; Przyjaciel-Zablocki, M.; Skilevic, S.; Lausen, G.: S2RDF: RDF Querying with SPARQL on Spark, 2015, URL: http://arxiv.org/abs/1512.07021, Stand: 12. 12. 2016.

[SJ10] Sun, J.; Jin, Q.: Scalable RDF store based on HBase and MapReduce. In: Proceedings of the 3rd ICACTE. S. 633–636, 2010.