Master Projekt Bericht

Erweiterung des Projektes Sempala
um das Datenformat Single Table

Manuel Schneider

6. September 2016

Inhaltsverzeichnis

1. Einleitung

Diese Projektarbeit beschäftigt sich mit der Erweiterung der SPARQL-auf-Hadoop Lösung Sempala. Sempala baut auf SQL auf und verwendet Impala als SQL Backend. Impala ist eine MPP SQL Query Engine auf Hadoop und erlaubt es SQL Anfragen nahezu in Echtzeit zu bearbeiten. Sempala übersetzt SPARQL Anfragen in SQL und profitiert von der geringen Latenzzeit der Impala Anfragen.

Im Ausblick seiner Masterthesis [5] schlägt Simon Skilevic ein Datenmodell vor, das auch, wie im ursprünglichen Sempala Datenformat „Unified Property Table“ den kompletten Datensatz in einer Tabelle hält. Auch wenn die Idee als Alternative zum S2RDF Extended Vertical Partitioning, welches auf Spark basiert, gedacht war, ist das Datenmodell auch auf Impala umsetzbar. In der Masterthesis wurde das Konzept Big Table genannt. Um Namenskonflikte zu vermeiden, wird das Konzept im Rahmen von Sempala *Single Table* genannt.

Wie beim S2RDF ExtVP ist die Idee hinter der Single Table, die Größe der Eingabemenge der notwendigen Joins für die Anfragen zu verringern, indem nur wirklich notwendige Daten über das Netzwerk übertragen und im Join bearbeitet werden müssen.

Im Rahmen dieses Master Projekts soll Sempala um das Datenmodell Single Table erweitert werden. Das bedeutet, dass Sempala die Singletable anlegen und anfragen können soll. Anschließend soll das neue Datenmodell getestet und evaluiert werden.

In Abschnitt 2 wird das Datenmodell Single Table im Detail erklärt. In Abschnitt 3 wird auf die technischen Details und die praktische Umsetzung eingegangen. In Abschnitt 4 werden die Tests erklärt und die Ergebnisse diskutiert. Abschließend werden in Abschnitt 5 die Erkenntnisse zusammengefasst.

2. Das Datenmodell

Das Ziel der Single Table ist die Verbesserung der Laufzeiten von Verbundanfragen. Da die Daten dezentral gelagert sind, müssen bei Verbundanfragen die für den Verbund notwendigen Daten der rechten Tabelle über das Netzwerk von jedem Knoten an jeden Knoten verteilt werden. Dieser Vorgang wird Broadcast genannt. Wenn die Menge der zu übertragenden Daten wächst, kann dieser Vorgang einen nicht unerheblichen Zeitaufwand mit sich bringen. Daher wird versucht die Daten, die über das Netzwerk gesendet werden, zu reduzieren. Selektion wird daher durch Impala vor dem Broadcast ausgeführt, um die Daten auf die gewünschten Attribute zu reduzieren.

Doch nicht allein der Engpass im Netzwerk spielt eine Rolle. Auch die Selektion des kompletten Datensatzes kann zu Performanceeinbußen führen. Eine erste Besserung bringt Impalas implizite Partitionierung der Tripel Tabelle nach Prädikaten. Impala muss dadurch zur Selektion nach Prädikaten nicht die komplette Tabelle nach Tripeln mit bestimmten Prädikaten durchsuchen, sondern legt intern für jedes Prädikat eine eigene Datei an, die direkten Zugriff auf Tripel mit jenem Prädikat zulässt.

Abgesehen von der Selektion der Daten und dem potentiellen Engpass im Netzwerk stellt die Komplexität des Verbundes eines der größten Laufzeitprobleme dar. Für den Verbund muss jedes Tripel des linken Verbundpartners mit jedem Tripel des rechten Verbundpartners verglichen werden. Angenommen die beiden Verbundmengen sind im Mittel gleich groß, steigt die Komplexität des Verbundes quadratisch mit der Eingabemenge. Diese Komplexität kann sich rasch zum dominierenden Faktor der Laufzeit entwickeln, wie später in der Evaluation zu sehen ist.

Das Datenmodell der Sempala Single Table versucht dem entgegenzuwirken, indem die Eingabemenge weiter reduziert wird. Erreicht wird das dadurch, dass der Tripel Tabelle zusätzliche Informationen angehängt werden, die durch Selektion helfen die Menge der Daten, die zum Verbund gebroadcastet werden, zu reduzieren. Das hilft einerseits das Netzwerk zu entlasten und andererseits die Eingabemenge zu reduzieren.

Vereinfacht gesagt wird das Tripel als einzelnes betrachtet und beurteilt zu welcher Relation es zu anderen Prädikaten steht. Abbildung 1 zeigt einen minimalen Graphen um die Idee zu illustrieren. Im RDF Graph sind fünf Tripel zu sehen. Das Tripel (A, P_1, B)

steht in Verbindung zu den Prädikaten P_2, P_3, P_4 und P_5. Umgekehrt steht zum Beispiel Tripel (D, P_3, A) nur zu P_1 und P_2 in Verbindung.

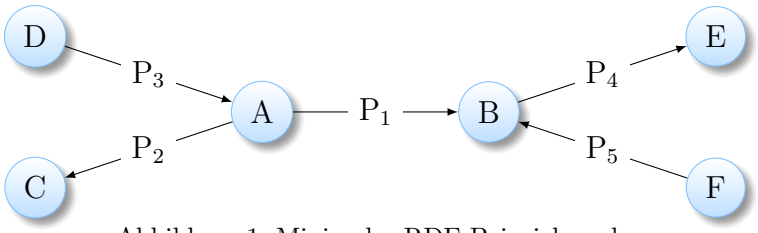

Abbildung 1: Minimaler RDF Beispielgraph.

Das verwendete Datenmodell geht noch einen Schritt weiter und bestimmt genau wie das Tripel zu anderen Prädikaten in Verbindung steht. Dazu wird angegeben, welche Seiten der anliegenden Tripel, also Subjekt oder Objekt, verbunden sind. Daraus folgt, dass es vier Gruppen gibt: Tripel die in Subjekt-Subjekt, Subjekt-Objekt, Objekt-Subjekt und Objekt-Objekt Verbindung stehen. Im Folgenden werden nur noch die Abkürzungen SS, SO, OS, und OO verwendet. Da OO Beziehungen in Anfragegraphen nur sehr selten vorkommen, werden sie in der Single Table nicht verwendet.

Im Beispiel in Abbildung 1 steht das Tripel (A, P_1, B) in SS-Verbindung zu P_2, in SO-Verbindung zu P_3, in OS-Verbindung zu P_4 und in OO-Verbindung zu P_5, welche aber wie erwähnt ignoriert wird. (D, P_3, A) hingegen steht in OS-Verbindung zu P_1 und in SS-Verbindung zu P_2.

Die Relationen werden in der Single Table in Form von Spalten mit booleschen Werten gespeichert. Für jedes Prädikat werden drei Spalten angelegt. Für jede Form der Verbindung eine. Das Datenbankschema für den RDF Graphen in Abbildung 1 würden dann wie folgt aussehen:

$$|S|P|O|SS_{P_1}|SO_{P_1}|OS_{P_1}|SS_{P_2}|SO_{P_2}|OS_{P_2}|SS_{P_3}|SO_{P_3}|OS_{P_3}|SS_{P_4}|SO_{P_4}|OS_{P_4}|SS_{P_5}|SO_{P_5}|OS_{P_5}|$$

Wenn das Tripel in einer bestimmten Form in Verbindung zu einem Prädikat steht, wird das Feld der entsprechenden Spalte auf `true` gesetzt, wenn nicht dann auf `false`. Die Single Table des minimalen Beispielgraphen sieht dementsprechend wie in Tabelle 1 dargestellt aus.

Wie in Tabelle 1 zu sehen ist, ist die Matrix dünnbesetzt. Je geringer die Dichte des Graphen ist, desto dünnbesetzer ist diese Matrix. RDF Graphen haben üblicherweise einen sehr geringe Dichte. Daraus folgt, dass die Single Table für gewöhnliche RDF Graphen sehr schwach besetzt ist. Spaltenorientierte Datenformate wie Parquet[1], die fähig sind sich wiederholende Werte speichereffizient zu kodieren, können von diesem Format profitieren und die Single Table mit wenig zusätzlichem Speicherplatz umsetzen.

Tabelle 1: Single Table zum minimalen Beispielgraph in Abbildung 1.

			SS					SO					OS				
S	P	O	P_1	P_2	P_3	P_4	P_5	P_1	P_2	P_3	P_4	P_5	P_1	P_2	P_3	P_4	P_5
A	P_1	B	·	✓	·	·	·	·	·	✓	·	·	·	·	·	✓	·
A	P_2	C	✓	·	·	·	·	·	·	✓	·	·	·	·	·	·	·
D	P_3	A	·	·	·	·	·	·	·	·	·	·	✓	✓	·	·	·
B	P_4	E	·	·	·	·	·	✓	·	·	·	✓	·	·	·	·	·
F	P_5	B	·	·	·	·	·	·	·	·	·	·	·	·	·	✓	·

3. Implementierung

Sempala setzt sich aus zwei Komponenten zusammen. Der Sempala Loader ist für das erstellen der Single Table zuständig. Dafür wird ein Hadoop Cluster wie zum Beispiel die Cloudera open-source Apache Hadoop Distribution benötigt, auf dem HDFS und Impala aktiviert ist. Rohdaten, die als Quelle für die zu erstellende Tabelle dienen, werden auch benötigt und müssen im HDFS gelagert sein. Der Sempala Translator ist für das Übersetzen von SPAQRL Anfragen in den Imapala SQL Dialekt zuständig.

Im Rahmen dieser Master Projekt Arbeit wurde der ursprüngliche Sempala Loader, der die Datenbanken noch mit dem MapReduce Framework erstellt hat, komplett neu geschrieben. Ziel war es die Datenbanken alleine mit Java und Impala SQL anstatt mit MapReduce zu erstellen. Zusätzlich wurde der Sempala Loader um das Datenformat Single Table erweitert.

Der Sempala Translator wurde ebenfalls um das Datenformat Single Table erweitert. Die ursprüngliche Variante des Semapla Translators diente lediglich dem Übersetzen

[1] Apache Parquet. http://parquet.apache.org/

der übergebenen Anfragen. Der Sempala Translator wurde zusätzlich um die Fähigkeit erweitert, die übergebenen und übersetzten Anfragen direkt auf den Impala Cluster auszuführen.

Des Weiteren wurde das ursprüngliche Build System durch Apache Maven ersetzt, welches das Dependency Mangement wesentlich vereinfacht. Das Projekt ist nun in drei Module gegliedert. Sempala Loader und Translator bilden die ersten zwei Module. Beide sind eigenständige Projekte, die ausführbare Programme erstellen. Da der Loader und Translator häufig gemeinsam gebraucht werden und die selben Abhängigkeiten teilen, vereint das dritte Modul letztere in eine ausführbare Datei. Ob der Loader oder Translator verwendet werden soll, kann in Form von High Level Commands angegegeben werden.

Um direkten JDBC Anschluss an den Impala Daemon, der auf dem CDH Cluster läuft, zu bekommen, wird der *Cloudera JDBC Driver for Impala*[2] verwendet. Der Cloudera JDBC Treiber selbst hat viele Abhängigkeiten, die sorgfältig in das Maven Dependency Management eingepflegt wurden. Lediglich der Impala JDBC Treiber selbst ist nicht in öffentlichen Repositories erhältlich. Daher muss er mit dem Maven Install Plugin in ein lokales Repository installiert werden, welches Maven zum Auflösen der Dependecies verwenden kann. Mehr Informationen dazu befinden sich in den einschlägigen Dateinen und Readmes im Quelltext.

3.1. Sempala Loader

Wie erwähnt ist die Aufgabe des Sempala Loaders die Erstellung der RDF Datenbanken in verschiedenen Formaten. Dem Programm müssen beim Start die folgenden Informationen übergeben werden: die Adresse des Koordinatorknotens, der Name der Datenbank, der Name des Datenmodells und der HDFS-Pfad der RDF Daten. Die Kommandozeilenschnittstelle erlaubt zusätzlich das Programm zu instruieren eine Präfix Datei zu verwenden um Namensräume in den Rohdaten zu ersetzen, einen eventuellen Punkt am Ende der Zeilen der Rohdaten zu ignorieren, Duplikate in der Eingabemenge zu entfernen und das Löschen der temporären Tabellen auszulassen. Des Weiteren kann man folgende

[2] http://www.cloudera.com/downloads.html

Standardwerte überschreiben: Standardport '21050' des Impala Dämons, Subjektspaltenname 'subject', Prädikatspaltenname 'predicate', Objektspaltenname 'object', Feldendmarke der Rohdaten '\t', Zeilenendmarke der Rohdaten '\n', Name der Ausgabetabelle 'singletable' beziehungsweise 'propertytable' und das Standard Verbundverhalten 'BROADCAST'.

Der Sempala Loader birgt keine komplizierte Architektur. Strukurell ist der Loader ein eher imperatives Programm das SQL Anfragen an den Impala Cluster sendet. Die essentielle Arbeit wird per SQL erledigt.

Da Impala SQL in keinem der populären Java SQL Query Builder, wie zum Beispiel QueryDsl oder jOOQ, vertreten ist, wurde ein unvollständiger Java SQL Query Builder entwickelt, der alle nötigen SQL Statements abdeckt. Er dient vorrangig der Reduktion der Fehleranfälligkeit und Lesbarkeit des Quelltextes.

Für die Erstellung der Property Table als auch der Single Table wird eine Triple Table benötigt. Um diese zu erstellen, wird mit dem HDFS Pfad, der als Parameter übergeben wurde, eine externe Tabelle erstellt. Externe Tabellen bieten die Möglichkeit Klartext Daten im HDFS durch Angabe des Formates der Daten als Tabelle zu verwenden. Mit dieser externen Tabelle wird eine schematisch identische interne Tabelle erstellt. Dadurch kann die Tabelle nach Prädikaten partitioniert, mit Parquet formatiert und mit Snappy komprimiert werden. In diesem Schritt werden auch, falls verlangt, die Namespaces durch Präfixe ersetzt.

3.1.1. Property Table

Die Property Table ist eine subjektorientierte Tabelle. Das Datenbankschema setzt sich zusammen aus dem Subjekt und den Prädikaten. In den Feldern der Prädikatspalten befinden sich die Objekte.

Aus der Triple Table wird eine Ausgangstabelle mit einer einzigen Spalte mit eindeutigen Subjekten erstellt. Ebenso wird eine leere Ergebnistabelle erstellt, deren Schema sich aus einer Subjektspalte und je einer Spalte für jedes Prädikat zusammensetzt.

Die Daten für die Property Table werden in einer einzigen SELECT Anfrage erlangt. Für jedes Prädikat wird ein Left Join der Ausgangstabelle mit den Tripeln, die jenes Prädikat

enthalten, über die Subjektspalte ausgeführt. Der Left Outer Join ist notwendig, da Subjekte nicht verloren gehen sollen, wenn sie in den Tripeln nicht vorkommen. Die Projektion der Objektspalte ergibt nach Umbenennung die Propertyspalte des Prädikats, das gejoint wurde. Schließlich wird diese Select Anfrage im Rahmen einer INSERT-AS-SELECT Statements in die Ergebnistabelle geschrieben.

S	P	O
S_1	P_1	O_1
S_1	P_1	O_2
S_1	P_2	O_3
S_1	P_2	O_4

S	P_1	P_2
S_1	O_1	O_3
S_1	O_2	O_4
S_1	O_1	O_3
S_1	O_2	O_4

Abbildung 2: Beispiel Triple Table und zugehörige Property Table.

Abbildung 2 zeigt eine Beispiel Triple Table und die Property Table, die daraus resultieren würde. In diesem Beispiel wird ersichtlich, dass die Daten dupliziert werden. O_1 und O_2 aus den ersten beiden Tripel der Triple Table werden durch den ersten Left Join in Spalte P_1 untergebracht. Beim zweiten Left Join mit den letzten zwei Tripeln werden die Daten vervielfacht, wie an den sich wiederholenden Objekten in der Property Table zu sehen ist.

In diesem minimalen Beispiel ist die Problematik des Speicherbedarfs nicht direkt ersichtlich, aber wenn nur zwei weitere Tripel mit einem dritten Prädikat hinzugefügt werden, steigt die Zeilenzahl der Property Table auf acht. Somit kann man die Zeilenzahl exponentiell steigen lassen. Glücklicherweise ist es in einem RDF Graphen eher die Ausnahme als die Regel, dass ein Subjekt viele ausgehende Kanten des selben Prädikats hat. Wenn es auch ein künstlicher Worst Case ist, das theoretische Problem existiert.

3.1.2. Single Table

Im Gegensatz zur Property Table ist die Single Table tripelorientiert. Wie die Ausgabe des Loaders für Single Table auszusehen hat, wurde in Abschnitt 2 detailliert erklärt.

Um die Single Table zu erstellen, werden, wie bei der Propterty Table, die zusätzlichen Informationen mit Left Joins angehängt. Da die Single Table tripelorientiert ist, dient hier die Triple Table als Ausgangstabelle. Auch hier wird ein Left Join verwendet, da die

Triple Table komplett erhalten bleiben muss, auch wenn kein Verbundpartner gefunden wird.

Da für die Erstellung der Spalten durch den Left Join nur die Kanten, also Prädikate, und die anliegenden Knoten benötigt werden, werden zwei temporäre Tabellen erstellt, die lediglich alle eindeutigen Objekt-Prädikat beziehungsweise Subjekt-Prädikat Relationen enthalten. Im Folgenden werden diese Tabellen OP und SP genannt.

Für eine SS_P Verbindung wird der Left Join der Triple Table mit SP über die Subjektspalten beider Tabellen ausgeführt. Für eine SO_P Verbindung wird der Left Join der Triple Table mit OP über die Subjektspalten der Triple Table und Objektspalte der OP ausgeführt. Für eine OS_P Verbindung wird der Left Join der Triple Table mit SP über die Objektspalte der Triple Table und Subjektspalte der SP ausgeführt. Die zusätzliche Joinbedingung, dass das Prädikat der SP beziehungsweise OP das Prädikat P der gerade erstellten Spalte sein muss, sorgt dafür, dass das Feld `NULL` ist, wenn das Tripel der Zeile nicht in der Relation steht, die die Spalte darstellt.

Was genau in den Feldern steht, wenn sie nicht `NULL` sind, ist unerheblich. Relevant ist nur, ob die Tripel in der Relation stehen, für die jene Spalte steht. Deshalb wird, wenn das Feld `NULL` ist, zur Selektion `false` ausgegeben und sonst `true`.

Listing 1 zeigt eine mögliche SQL Anfrage, die die Daten der Single Table berechnet. Sie zeigt, dass die Anfragen abhängig von der Anzahl der Prädikate sehr umfangreich werden kann. Jedoch sind Prädikate üblicherweise endlich und relativ klein. Zum Beispiel hat der Datensatz, der zum Testen der Single Table verwendet wird, 85 Prädikate.

Listing 1: Beispiel Anfrage zur Erstellung der Single Table

```
SELECT
  tt.s, tt.p, tt.o,
  CASE WHEN tss_p1.s IS NULL THEN false ELSE true END AS ss_p1,
  CASE WHEN tso_p1.o IS NULL THEN false ELSE true END AS so_p1,
  CASE WHEN tos_p1.s IS NULL THEN false ELSE true END AS os_p1,
  CASE WHEN tss_p2.s IS NULL THEN false ELSE true END AS ss_p2,
  CASE WHEN tso_p2.o IS NULL THEN false ELSE true END AS so_p2,
  CASE WHEN tos_p2.s IS NULL THEN false ELSE true END AS os_p2,
  CASE WHEN tss_p3.s IS NULL THEN false ELSE true END AS ss_p3,
  CASE WHEN tso_p3.o IS NULL THEN false ELSE true END AS so_p3,
  CASE WHEN tos_p3.s IS NULL THEN false ELSE true END AS os_p3,
  CASE WHEN tss_p4.s IS NULL THEN false ELSE true END AS ss_p4,
  CASE WHEN tso_p4.o IS NULL THEN false ELSE true END AS so_p4,
  CASE WHEN tos_p4.s IS NULL THEN false ELSE true END AS os_p4,
  CASE WHEN tss_p5.s IS NULL THEN false ELSE true END AS ss_p5,
  CASE WHEN tso_p5.o IS NULL THEN false ELSE true END AS so_p5,
  CASE WHEN tos_p5.s IS NULL THEN false ELSE true END AS os_p5,
FROM tripletable tt
  LEFT JOIN sp_relations tss_p1 ON tt.s=tss_p1.s AND tss_p1.p='p1'
  LEFT JOIN op_relations tso_p1 ON tt.s=tso_p1.o AND tso_p1.p='p1'
  LEFT JOIN sp_relations tos_p1 ON tt.o=tss_p1.s AND tos_p1.p='p1'
  LEFT JOIN sp_relations tss_p2 ON tt.s=tss_p2.s AND tss_p2.p='p2'
  LEFT JOIN op_relations tso_p2 ON tt.s=tso_p2.o AND tso_p2.p='p2'
  LEFT JOIN sp_relations tos_p2 ON tt.o=tss_p2.s AND tos_p2.p='p2'
  LEFT JOIN sp_relations tss_p3 ON tt.s=tss_p3.s AND tss_p3.p='p3'
  LEFT JOIN op_relations tso_p3 ON tt.s=tso_p3.o AND tso_p3.p='p3'
  LEFT JOIN sp_relations tos_p3 ON tt.o=tss_p3.s AND tos_p3.p='p3'
  LEFT JOIN sp_relations tss_p4 ON tt.s=tss_p4.s AND tss_p4.p='p4'
  LEFT JOIN op_relations tso_p4 ON tt.s=tso_p4.o AND tso_p4.p='p4'
  LEFT JOIN sp_relations tos_p4 ON tt.o=tss_p4.s AND tos_p4.p='p4'
  LEFT JOIN sp_relations tss_p5 ON tt.s=tss_p5.s AND tss_p5.p='p5'
  LEFT JOIN op_relations tso_p5 ON tt.s=tso_p5.o AND tso_p5.p='p5'
  LEFT JOIN sp_relations tos_p5 ON tt.o=tss_p5.s AND tos_p5.p='p5'
```

Da Anfragen diesen Umfangs sehr speicherintensiv sind, wird die Single Table inkrementell erstellt. Die Ergebnistabelle wird im Voraus erstellt. Die Daten der Single Table werden dann partitionsweise erstellt und mit einem INSERT-AS-SELECT Statement in die Ergebnistabelle eingefügt.

3.2. Sempala Translator

Die Aufgabe des Sempala Loaders ist das Übersetzen und das eventuelle Ausführen einer Menge übergebener SPARQL Anfragen. Dem Programm werden beim Start mindestens der Name des Datenmodells und der Pfad zu einer Datei oder einem Order mit Dateien, die SPARQL Anfragen enthalten, übergeben. Zusätzlich kann die Adresse und Port eines Koordinatorknotens und der Name einer Datenbank angegeben werden. Sempala versucht dann die angegebenen SPARQL Anfragen auf dieser Datenbank auszuführen. Des Weiteren kann der Translator, wie der Loader, instruiert werden die Namensräume

durch Präfixe zu ersetzen, die Ergebnistabellen zu Benchmarkingzwecken direkt nach dem Erstellen zu löschen oder die SPARQL Algebra Optimierung zu aktivieren.

Der Sempala Translator durchläuft für jede Anfrage, exklusive der Ausführung der Anfrage, vier Phasen. Die SPARQL Anfrage wird geparst und in eine interne Objektstruktur überführt. Aus dieser internen Repräsentation wird ein SPARQL Algebra Baum erstellt, der die Anfrage repräsentiert. Dieser SPARQL Algebra Baum wird daraufhin in einen Impala SQL Algebra Baum übersetzt und anschließend wird aus dem Impala SQL Algebra Baum eine SQL Anfrage erstellt, welche nach Bedarf ausgeführt werden kann.

Phase eins und zwei wird komplett vom Apache Jena Framework übernommen. Jena gibt einen Algebra Baum zurück, der nach dem Visitor Pattern traversiert werden kann. Dazu bietet Jena das Interface eines zu besuchenden Objekts `Op` und das dazugehörige Interface des Besuchers `OpVisitor`. Der SPARQL Algebra Baum besteht aus Klassen, die das Interface `Op` realisieren.

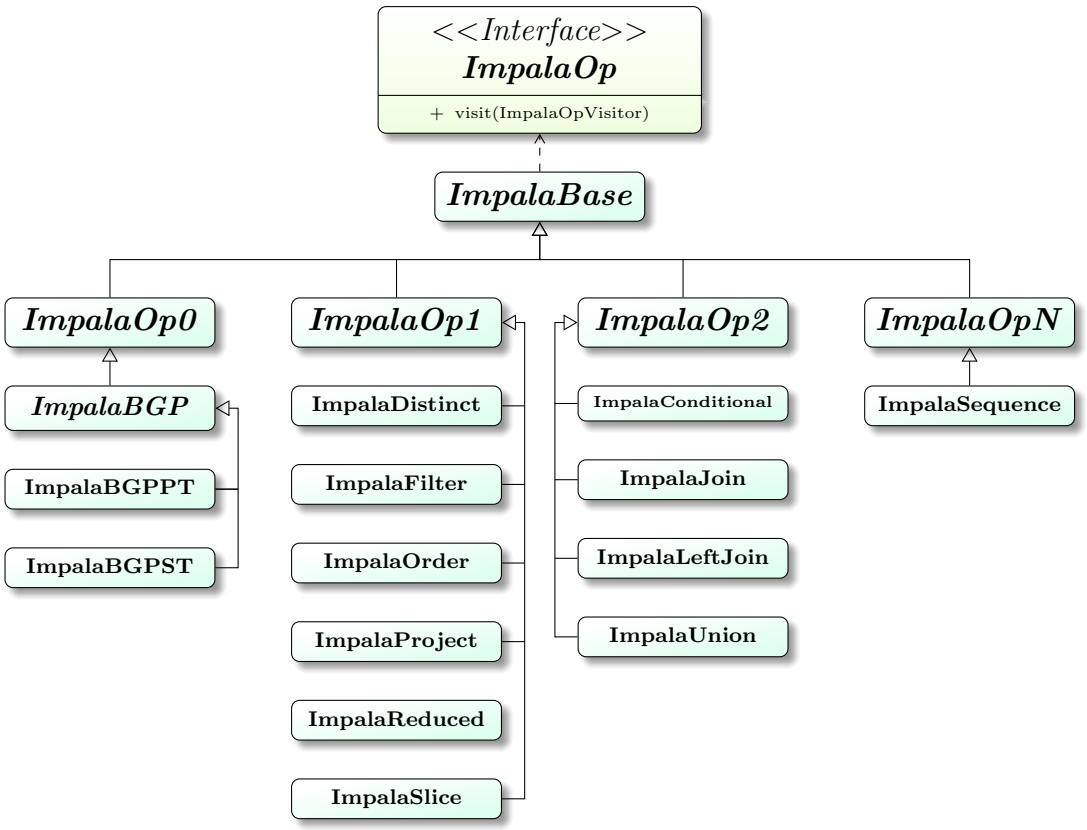

Abbildung 3: Die Impala Algebra Klassen.

Die Aufgabe des `AlgebraTransformers`, ist es den SPARQL Algebra Baum in einen Impala SQL Algebra Baum zu transformieren. Er realisiert das `OpVisitor` Interface und kann daher mit Hilfe des `AlgebraWalkers` den SPARQL Algebra Baum traversieren. Während der Traversierung des Baums erstellt der `AlgebraTransformer` einen äquivalenten Impala SQL Algebra Baum. Dazu werden eigene Klassen verwendet, die sich strukturell an den Jena Algebra Klassen orientieren. Abbildung 3 zeigt die Klassen, aus denen der Impala Algebra Baum erstellt wird.

In der Abbildung ist zu erkennen, dass ImpalaBase das zu `Op` äquivalente `ImapalaOp` realisiert. Die vier Operationen `ImpalaOpX` bilden Abstraktionen über Operationen, die eine bestimmte Anzahl an Suboperationen halten. Die atomaren Operationen ohne Parameter bilden immer `ImpalaBGP`s. Sie sind im Algebra Baum immer die Blätter.

Als Gegenstück zur `ImpalaOp` gibt es auch ein zum `OpVisitor` äquivalentes `ImpalaOpVisitor` Interface. Der `ImpalaOpTransformer` realisiert das `ImpalaOpVisitor` Interface und kann mit dem `ImpalaOpWalker` den erstelltem Impala Algebra Baum traversieren und eine entsprechende SQL Anfrage erstellen.

Bei der Übersetzung einer SPARQL Anfrage zur SQL Anfrage unterscheiden sich die Impala Single Table von der Property Table nur durch die Übersetzung des Basic Graph Pattern. Daher wurde für die Einführung der Single Table die Klasse `ImpalaBGP` zu einer abstrakten Klasse gemacht und zwei neue Subklassen eingeführt. `ImpalaBGPPropertyTable` erledigt die Arbeit, die zuvor das `ImpalaBGP` erledigt hat, während `ImpalaBGPSingleTable` für die Übersetzung eines Basic Graph Patterns in einen SQL String, der mit der Single Table kompatibel ist, zuständig ist.

Welche der beiden Klassen instantiiert und Teil des Impala Algebrabaumes wird, entscheidet der `AlgebraTransformer` während der Transformation des SPARQL Algebra Baumes.

Im Folgenden wird beschrieben, wie die SPAQRL Anfrage in eine SQL Anfrage, die zum Datenmodell der Single Table passt, übersetzt wird. SPARQL stellt keine Anforderungen an die Ordnung der Tripel im Basic Graph Pattern. Für die Joins der Tripel in der Single Table ist die Reihenfolge aber wichtig, denn für einen Join wird eine Spalte benötigt, die auf Gleichheit geprüft werden kann. Anschaulich bedeutet das, dass die Tripel im BGP

anliegend sein müssen. Des Weiteren sollen Partitionen im Graph erkannt werden, die durch das Kreuzprodukt verbunden werden.

Um Partitionen und Reihenfolge zu bestimmen, kommt ein Nachbarschafts Algorithmus zum Einsatz. Dazu beginnt man bei einem zufälligen Tripel im Graph und findet anliegende Tripel. Das wird so oft wiederholt, bis es keine anliegende Tripel mehr gibt. Sind noch Tripel übrig ist der Anfragegraph partitioniert. Die bisherigen Tripel gehören zur ersten Partition. Man fährt mit den restlichen Tripeln wie eben beschrieben fort, bis alle Tripel behandelt wurden. Nun sind alle Partitionen bekannt und die Reihenfolge, in der die Tripel bearbeitet wurden ist die Join Reihenfolge. Nähere Details können im Quelltext dieses Projektes eingesehen werden.

Die einzelnen Tripel in einer Partition werden zu eigenständigen Subqueries verarbeitet. In diesem Schritt werden die zusätzlichen Informationen der Singletable verwendet. Mit einem zuvor angelegten invertiertem Index, der Variablen auf Tripel abbildet, werden die Kriterien, die die Daten der Subquery zu erfüllen haben, ermittelt und zur Selektion der Subquery hinzugefügt. Hier wird der eingangs im Abschnitt 2 beschriebene Performancegewinn erzeugt, denn die strengere Selektion liefert weniger Daten, die das Netzwerk oder den Joinvorgang belasten. Abschließend werden die Subqueries über die anliegenden Variablen gejoint und die Partitionen einem Cross Join unterzogen.

4. Evaluation

Die Testumgebung in der die Anfragen ausgeführt wurden besteht aus einem Cluster aus zehn Rechnern. Alle Rechner besitzen einen Intel Xeon E5-2420 Prozessor, der mit einer Grundfrequenz von 1,9 GHz taktet und eine maximale TurboBoost-Frequenz[3] von 2,4 GHz hat. Jede Maschine hat 32 GB Arbeitsspeicher und zwei 2 TB Festplatten. Verbunden sind die Rechner über eine Gigabit Netzwerkverbindung. Impala läuft auf Cluster im Rahmen der Cloudera open-source Apache Hadoop Distribution CDH Version 5.7.0, die Hadoop (HDFS) 2.6.0 und Impala in der Version 2.5.0 liefert.

[3]Die Intel Turbo-Boost-Technik erhöht dynamisch die Frequenz eines Prozessors nach Bedarf, indem die Temperatur- und Leistungsreserven ausgenutzt werden, um bei Bedarf mehr Geschwindigkeit und andernfalls mehr Energieeffizienz zu bieten.

Die *Waterloo SPARQL Diversity Test Suite* liefert einen Datengenerator mit dem die Testdaten generiert wurden. Der Generator erlaubt die ausgegebenen Datenmengen zu skalieren. Die generierten Datensätze enthalten Vielfache von etwa 105 Tausend N-Tripel. Für das Testen des Datenformates Single Table wurden Datensätze des Skalierungsfaktors (von nun an *SF*) 10, 100, 1000 und 10000 verwendet. Umgerechnet sind das in etwa eine Million bis zu einer Milliarde Datensätze.

Die Testergebnisse der Sempala Single Table werden mit ähnlichen SPARQL-auf-Hadoop Systemen verglichen. Zum einen wird die ursprüngliche Version von Sempala als bisher einziger Impala SPARQL Query Processor als Referenz hergezogen, zum anderen S2RDF, welches auf Apache Spark basiert und aktuell eines der schnellsten SPARQL-auf-Hadoop Systeme ist [4].

Sempala in der ursprünglichen Version basiert auf dem Datenmodell Property Table. Die Property Table basiert auf der Idee RDF Daten subjektorientiert zu speichern, indem das Datenbankschema für das Subjekt und jedes Prädikat eine Spalte enthält. Die Objekte werden dann in den einzelnen Feldern gespeichert. Das erfordert die Möglichkeit verschachtelte Daten zu speichern[4]. Mehr dazu in [3].

S2RDF bietet verschiedene Datenmodelle. Die zuvor erwähnte Performance wird mit dem Datenmodell ExtVP erreicht, welches auf der Idee basiert für alle Partitionen beziehungsweise Prädikate die für einen Join notwendigen Tripel im voraus zu berechnen. Mehr dazu in [4]. Ext VP wird als Refernz für aktuelle Systeme dieser Art verwendet. S2RDF Big Table basiert auf dem selben Datenmodell wie die Single Table und wird als Referenz für dieses Datenmodell auf einer anderen Engine verwendet. Somit kann beurteilt werden ob Laufzeitunterschiede vom Datenmodell oder der Engine her rühren.

Die Ladezeiten und HDFS-Speicherbedarf der Single Table sind in Tabelle 2 gelistet. Zum Vergleich wurden auch die Ladezeiten von S2RDF ExtVP und Sempala Property Table, sowie der Speicherbedarf von S2RDF ExtVP, S2RDF Big Table und Sempala Property Table gelistet.

[4]Das Datenformat, das Impala zugrunde liegt, unterstützt zwar verschachtelte Daten, aber Impala unterstützt letztere erst seit Impala 2.3. Sempala Property Table wurde vor diesem Release konzipiert und implementiert und verwendet daher einen Workaround [3].

Tabelle 2: Ladezeiten und HDFS-Speicherbedarf im Vergleich mit ähnlichen Systemen.

	SF	10	100	1000	10000
Ladezeit	Single Table	564 s	855 s	5179 s	67080 s
	Property Table	26 s	56 s	333 s	2782 s
	ExtVP	1430 s	2418 s	9497 s	60572 s
Speicherbedarf	Klartext	49 MB	507 MB	5,3 GB	54,9 GB
	Triple Table	6,9 MB	103 MB	1,2 MB	13,2 GB
	Single Table	13,5 MB	141 MB	1,6 GB	22,7 GB
	Property Table	13 MB	249 MB	3,5 GB	40,4 GB
	ExtVP	231 MB	614 MB	6,2 GB	63,7 GB
	Big Table	240 MB	419 MB	1,6 GB	13,6 GB

Die Laufzeiten der Single Table scheinen mit der Eingabemenge ungefähr gleich schnell zu wachsen wie die der ExtVP. Der Grund ist wie auch bei der ExtVP die relativ teure Erstellung der Tabelle, bei der viele LeftJoins ausgeführt werden müssen.

Der Speicherbedarf der Single Table ist relativ gering. Das liegt zum einen daran, dass die Single Table den Datensatz gemessen an der Zeilenanzahl nicht vergrößert, sondern lediglich eine konstante Anzahl an Spalten zu der Triple Table hinzufügt, die boolesche Werte enthalten. Zum anderen ist das zugrundeliegende Datenformat Parquet[5] wie geschaffen für das Datenmodell der Single Table, da es fähig ist sich wiederholende Werte durch Repetition Levels speichereffizient zu kodieren [2] und Wiederholungen, bedingt durch die kleine Grundmenge der Booleschen Algebra, die Regel sind. Der Vergleich mit der Property Table und Big Table zeigt, dass unabhängig der Kodierung und Komprimierung das Datenmodell eine große Rolle zu spielen scheint.

Im Folgenden werden die Ergebnisse der Sempala Single Table Testläufe präsentiert. In Abschnitt 4.1 wird die Single Table mit den Anfragen getestet, die mit der Waterloo SPARQL Diversity Test Suite geliefert werden. In Abschnitt 4.2 wird das Datenmodell mit der komplementären Testsuite Incremental Linear Testing getestet, welche sich auf lineare Anfragegraphen höheren Durchmessers konzentriert.

Tabelle 3: Laufzeiten und Mittel der WatDiv Basic Anfragen im Vergleich zu ähnlichen Systemen [ms].

	Query	L1	L2	L3	L4	L5	AM_L	S1	S2	S3	S4	S5	S6	S7	AM_S
SF10	Single Table	591	590	527	534	590	567	1000	634	618	629	617	582	572	665
	Property Table	786	748	642	642	758	715	1120	822	764	790	866	714	652	818
	Big Table	262	206	157	158	211	199	844	343	364	297	354	248	277	390
	ExtVP	164	145	97	95	140	128	478	204	180	190	211	138	141	220
SF100	Single Table	640	621	570	575	640	609	1019	730	661	663	668	617	606	709
	Property Table	750	748	636	646	748	706	1340	854	756	844	940	764	754	893
	Big Table	265	219	171	166	203	205	893	355	369	331	379	285	289	414
	ExtVP	168	193	126	107	173	153	562	216	214	221	193	146	164	245
SF1000	Single Table	947	706	866	697	670	777	1792	910	919	831	858	886	908	1015
	Property Table	904	746	740	664	752	761	3130	1058	862	876	960	848	870	1229
	Big Table	353	281	292	212	224	272	1032	454	439	372	429	326	426	497
	ExtVP	202	196	196	132	162	178	735	294	219	209	199	209	191	294
SF10000	Single Table	4775	2372	4346	2340	1110	2989	7977	4218	2945	2522	2968	3735	4581	4135
	Single Table$_M$	4552	2347	3484	2329	1068	2756	6781	3996	2975	2243	3004	3252	3796	3721
	Single Table$_C$	4384	2208	4021	2359	1050	2804	7794	4156	2875	2503	2858	3741	3969	4019
	Single Table$_C M$	4409	2394	3425	2364	1096	2738	6577	3982	2972	2197	2950	3167	3779	3661
	Property Table	3938	2140	3630	2616	1914	2848	17386	5368	2816	2442	3142	2260	3476	5270
	Big Table	728	661	776	502	364	606	2169	886	734	564	743	530	812	920
	ExtVP	471	498	549	209	270	399	2208	607	311	329	260	235	420	624

	Query	F1	F2	F3	F4	F5	AM_F	C1	C2	C3	AM_C	AM_T
SF10	Single Table	742	861	767	941	767	816	912	1009	828	916	716
	Property Table	866	1100	1180	1172	988	1061	1184	1354	1066	1201	911
	Big Table	642	954	618	985	541	748	1277	1332	704	1104	539
	ExtVP	370	451	376	461	334	398	535	472	450	486	282
SF100	Single Table	784	918	804	990	814	862	1007	1002	907	972	762
	Property Table	928	1174	1150	1202	1072	1105	1292	1656	1702	1550	998
	Big Table	696	961	624	1055	591	785	1524	1359	865	1249	580
	ExtVP	393	539	385	579	398	459	577	689	688	651	337
SF1000	Single Table	1101	1220	1414	1575	1436	1349	1689	2435	3900	2675	1288
	Property Table	1068	1704	1538	1950	2545	1761	2828	5992	6040	4953	1804
	Big Table	791	1114	854	1202	870	966	1655	1620	2322	1866	763
	ExtVP	433	642	638	692	672	615	923	1460	2929	1771	567
SF10000	Single Table	3569	4043	8306	10184	8331	6887	4247	16770	7899	9639	5362
	Single Table$_M$	3207	4183	7861	7307	6661	5844	4223	15896	5451	8523	4731
	Single Table$_C$	3614	4102	7713	9692	7934	6611	4753	16212	8458	9808	5231
	Single Table$_C M$	3173	4101	7665	7303	6593	5767	4345	15567	5460	8457	4676
	Property Table	4420	9316	12090	11668	19516	11402	23136	39710	37462	33436	10422
	Big Table	1206	1619	2247	1936	2512	1904	3224	3410	12477	6370	1905
	ExtVP	590	1226	1969	1265	2254	1461	2508	2740	16407	7218	1766

4.1. WatDiv Basic

Die Waterloo SPARQL Diversity Test Suite liefert zwanzig Anfragevorlagen verschiedener Typen. *Linear Queries* sind gerade Pfade im Graph, während *Star Queries* Anfragen sind, die der Form eines Sternes gleichen. *Snowflake Queries* sind verbundene Sternanfragen und *Complex Queries* eine Kombination aus allen drei Kategorien. Die verwendeten Anfragevorlagen können in Anhang A eingesehen werden.

Die Linear-, Star- und Snowflake Anfragen sind variabel. Die Vorlagen enthalten eine Variable die durch einen zufällig gewählten Internationalized Resource Identifiers einer angegebenen Klasse gewählt werden muss. Beispielsweise muss für die Anfrage L1 (Vgl.

[5] Apache Parquet. http://parquet.apache.org/

Anhang A.1) die Variable %v1% mit einer IRI der Klasse wsdbm:Website ersetzt werden.

Die Anfragen wurden auf den vier Datensätzen mit dem Skalierungsfaktor 10, 100, 1000 und 10000 ausgeführt. Da Sempala ein RDF-auf-SQL System ist, dessen eigentlicher Zweck die Ausführung von SPARQL Anfragen ist, werden die Ergebnisse zur Weiterverarbeitung in einer Ergebnistabelle gespeichert.

Um wirklich nur die Laufzeiten zu messen, die das System braucht, um die Daten zu erlangen, wird dieser Schritt üblicherweise ausgelassen. Deshalb wurde für den Skalierungsfaktor 10000 eine weitere Testreihe mit einer modifizierten Version von Sempala ausgeführt, die die Ergebnisse zählt anstatt sie zu speichern. Diese Modifikation sorgt dafür, dass die Ergebnisse von jedem Host lokal aggregiert werden und das Ergebnis von nur einem Record an den Koordinatorknoten gesendet wird. Da die Ergebnisse je nach Anfrage sehr groß werden können, erspart diese Maßnahme das potentiell sehr lange dauernde Verteilen der Ergebnisse im Cluster. Zusätzlich entfällt die zeitaufwändige Festplatten Ein- und Ausgabe, die notwendig für das dezentrale Speichern der Ergebnisse ist. Im Folgenden wird die modifizierte Version mit einem M im Index referenziert: Single Table$_{\mathrm{M}}$.

Impala bietet die Möglichkeit durch HDFS Caching Partitionen oder ganze Tabellen zwischenzuspeichern. Somit kann Impala Daten mit der Geschwindigkeit des Speicherbusses lesen und schreiben [1]. Zum Vergleich wurden weitere Testläufe ausgeführt, die auf eine Tabelle im Cache zugreifen. Die Testläufe mit einer gecachten Tabelle wurden jeweils mit der normalen und der modifizierten Version von Sempala ausgeführt. Die Ergebnisse aller Testläufe mit WatDiv Basic Anfragen sind in Tabelle 3 gelistet. Gecachte Testläufe sind in der Tabelle mit einem C im Index gekennzeichnet.

Die mittleren Laufzeiten der jeweiligen Anfragenkategorie sind in Abbildung 4 gegenüber gestellt. Da sich die Laufzeiten teils um einige Größenordungen unterscheiden, wurde eine logarithmische Skala gewählt. Bei Betrachtung fällt direkt die Dominanz des Extended Vertical Partitioning und der zu der Single Table äquivalenten Implementierung S2RDF Big Table ins Auge. Diese Dominanz ist auch konsistent in allen Kategorien zu beobachten. Die Ordnung der Ergebnisse legt die Vermutung nahe, dass Spark die schnellere

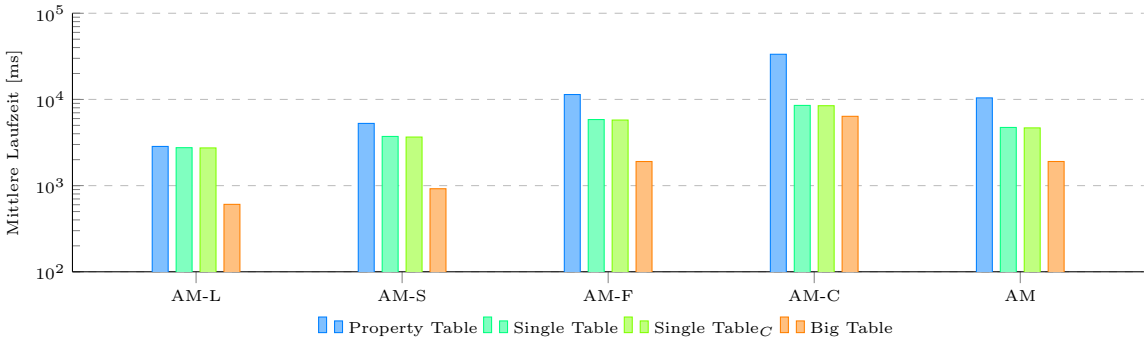

Abbildung 4: Mittlere Laufzeiten der WatDiv Basic Anfragen.

MPP Engine ist. Diese Vermutung deckt sich auch mit den Big Data Benchmarks[6] der Berkeley Universität von Californien. Um Impala und Spark im Rahmen dieser Evaluation zu vergleichen, müssen Konfigurationen gewählt werden, die vergleichbar sind. Um Spark mit der Sempala Single Table zu vergleichen, muss S2RDF Big Table verwendet werden, da das S2RDF ExtVP ein anderes Datenmodell verwendet. Da Spark ein In-Memory Sytem ist und zur Evaluation der S2RDF Big Table die Ergebnisse nach der Anfrage verworfen wurden, muss mit der gecachten und modifizierten Single Table Count Variante verglichen werden. Die Laufzeiten zeigen: Spark kann die WatDiv Basic Anfragen etwa zwei bis fünf mal schneller verarbeiten.

Beim Vergleich der beiden Impala Systeme Sempala Property Table und Sempala Single Table stellt sich heraus, dass die Single Table im Mittel die schnellere Alternative ist. Auffällig ist, dass die relative Performance der Property Table bei den Linear Queries verglichen mit den anderen Kategorien etwas höher, während die der Single Table etwas niedriger ist. Erstaunlich ist dabei, dass dieses Verhalten eigentlich bei den Star Queries zu erwarten wäre, weil das Datenformat Property Table bei einzelnen Stern-Anfragen keine Joins benötigt[7]. Des Weiteren wäre auch zu erwarten gewesen, dass Sempala Single Table in den Linear Queries relativ gut abschneidet, weil die Architektur der Single Table mit dem Gedanken Linear Queries zu optimieren entwickelt wurde.

Ein möglicher Faktor für diese Erwartungsuntreue ist die Gestalt der Linear Queries. Die subjektorientierte Property Table profitiert von den Linear Queries die aus einer degenerieten Star Query bestehen oder letztere enthalten. Die Property Table benötigt

[6] https://amplab.cs.berkeley.edu/benchmark/

[7] Das gilt aktuell allerdings nur solange Prädikate in den BGPs einmalig vorkommen [3], was in den WatDiv Basic Star Queries der Fall ist (Vgl. Anhang A).

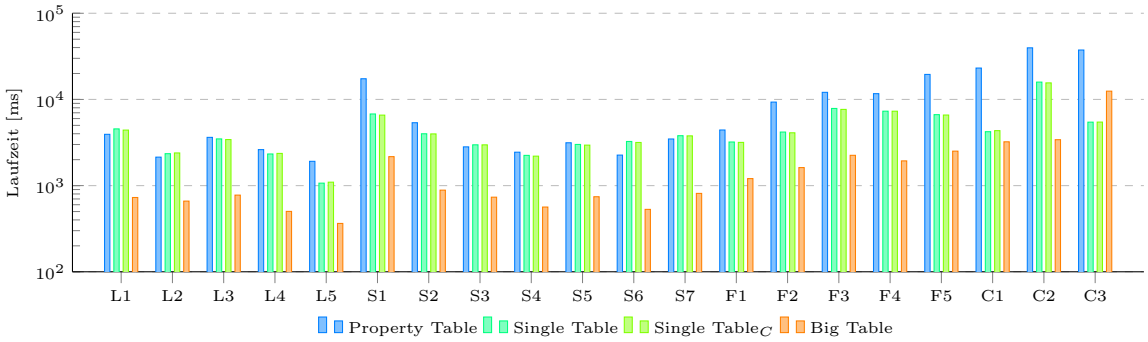

Abbildung 5: Laufzeiten der WatDiv Basic Anfragen.

maximal einen Join für die Verarbeitung der Linear Queries, da der Anfragegraph nicht gerichtet ist und somit Subjekte mit auschließlich ausgehenden Kanten enthält. Für L3 und L4 sind Joins nicht notwendig, da die Anfrage aus einem Subjekt mit zwei ausgehenden Kanten besteht. Währendessen muss Sempala Single Table zwei beziehungsweise drei Joins aufwenden.

In Abbildung 5 sind die Laufzeiten der einzelnen Anfragen gegenübergestellt. Mit einer Außnahme zeigt sich auch hier die selbe Ordnung der Mittelwerte wie in Abbildung 4. Die Ergebnisse der Anfrage C3 zeigen, dass Sempala Single Table die Anfrage etwa doppelt so schnell wie S2RDF Big Table verarbeiten kann. Der einzig ersichtliche Grund hierfür ist die starke Korrelation zur Ergebnismenge. Eindeutigere Anzeichen hierfür wird die folgende Anfrageklasse, die Incremental Linear Queries, liefern, welche teils erheblich größere Ergebnismengen ergeben.

Generell kann auch beobachtet werden, dass der Overhead bei kleineren Skalierungsfaktoren eine große Rolle spielt. So liegen die Laufzeiten der Anfragen auf dem Datensatz SF10 häufig im Rahmen der Standardabweichnung oder gar über den mittleren Laufzeiten der Anfragen auf dem Datensatz SF100. Anschaulich überlappt ein erheblicher Teil der Wahrscheinlichkeitsverteilungen.

4.2. WatDiv Incremental Linear

Die Incremental Linear Testing Anfragen sind ein Menge von Vorlagen die komplementär zu den WatDiv Basic Anfragen, welche größtenteils Anfragen mit einem maxi-

Tabelle 4: Laufzeiten und Mittel der WatDiv IL Anfragen der Pfadlängen 5 bis 10 [ms].

		IL-1-5	IL-1-6	IL-1-7	IL-1-8	IL-1-9	IL-1-10	AM_{IL-1}	IL-2-5	IL-2-6	IL-2-7	IL-2-8	IL-2-9	IL-2-10	AM_{IL-2}
SF10	Single Table	888	1074	1165	994	1037	1088	1041	782	858	900	966	1027	1078	935
	Property Table	1123	1064	1174	1291	1361	1444	1243	1054	1050	1062	1079	1081	1191	1086
	Big Table	543	628	759	888	995	1420	872	556	591	702	839	969	1171	805
	ExtVP	307	360	390	493	503	701	459	453	334	388	466	495	660	466
SF100	Single Table	2953	4067	4709	2417	2199	2312	3110	1493	1723	1698	1798	1845	1901	1743
	Property Table	3643	3753	3844	3919	4042	4126	3888	2164	2257	2290	2450	2527	2644	2389
	Big Table	845	928	1126	1275	1443	1804	1237	1317	947	1093	1198	1390	1715	1276
	ExtVP	558	604	711	834	875	1299	814	969	594	654	795	911	1047	828
SF1000	Single Table	14118	18865	22838	11773	10596	10654	14807	10913	8888	8755	8499	8974	8849	9146
	Property Table	29321	29684	29595	29696	29658	29663	29603	19357	19388	19496	19867	20162	20152	19737
	Big Table	2571	2308	2640	2827	3057	3293	2783	6034	3041	3006	3386	3845	3807	3853
	ExtVP	1724	1745	1965	2029	2185	2643	2048	4944	1869	1980	2114	2382	2413	2617
SF10000	Single Table	235865	148567	178891	148158	144950	176364	172133	59607	125678	84605	75862	113046	111570	95061
	Single Table$_M$	212930	125441	161425	154996	130530	150831	156025	45675	87538	73453	69693	79123	82707	73031
	Single Table$_C$	172769	123045	142636	115680	110874	114760	129961	49887	98169	65767	63897	79569	74608	71983
	Single Table$_{CM}$	193029	110812	139216	121499	118094	121926	134096	43566	79733	71752	65806	72256	75662	68129
	Property Table	128486	131304	152730	152169	153360	154272	145387	61843	63501	64487	76717	97933	96590	76845
	Big Table	21273	16238	19872	22364	24376	22195	21053	57251	25835	26848	28188	30787	29562	33078
	ExtVP	12543	12252	15062	15003	15478	16124	14410	41188	13276	14182	15261	16313	13922	19024

		IL-3-5	IL-3-6	IL-3-7	IL-3-8	IL-3-9	IL-3-10	AM_{IL-3}	AM-5	AM-6	AM-7	AM-8	AM-9	AM-10
SF10	Single Table	9465	12070	2696	67921	4431	4861	16907	3712	4667	1587	23294	2165	2342
	Property Table	2624	3155	1882	12548	3454	3620	4547	1600	1757	1373	4973	1965	2085
	Big Table	558	1020	937	7907	1390	1594	2234	552	746	799	3211	1118	1395
	ExtVP	382	783	732	10698	1118	1241	2492	381	492	503	3885	705	868
SF100	Single Table	96188	125219	22903	714894	37645	41267	173019	33545	43669	9770	239703	13896	15160
	Property Table	19214	26118	11818	111690	26654	28562	37343	8340	10709	5984	39353	11074	15092
	Big Table	1215	2893	1904	24739	3238	3473	6244	1126	1589	1374	9071	2024	2330
	ExtVP	855	2766	1978	36443	3608	3244	8149	794	1322	1114	12691	1798	1863
SF1000	Single Table	479772	620307	107386	3583029	213286	230228	872335	168268	216020	46326	1201100	77619	83243
	Property Table	155298	194758	93424	878232	217636	231430	295130	67992	81277	47505	309265	89152	93748
	Big Table	4509	12225	6855	108537	10335	10360	25470	4371	5858	4167	38250	5746	5820
	ExtVP	4474	12188	8552	178514	13411	13405	38424	3714	5267	4166	60886	5993	6154
SF10000	Single Table	2167746	2854454	761754	42602898	2997969	3178982	9093967	821073	1042900	341750	14275639	1085322	1155639
	Single Table$_M$	252653	144565	104940	728210	202999	208068	273573	170419	119181	113273	317633	137551	147202
	Single Table$_C$	2141407	2427551	669212	42454373	3006077	3221576	8986699	788021	882921	292538	14211317	1065507	1136981
	Single Table$_{CM}$	224795	125154	89350	733727	196108	205548	262447	153797	105233	100106	307011	128819	134379
	Property Table	493016	595152	365868	5649620	2026680	2462137	1932079	227782	263319	194362	1959502	759324	904333
	Big Table	33370	85228	64061	1656994	130310	99129	344849	37298	42434	36927	569182	61824	50295
	ExtVP	29590	87525	102971	2068100	158595	141940	431454	27774	37684	44072	699454	63462	57329

malen Durchmesser der Länge drei haben, mit einem Durchmesser von fünf bis zehn abdecken.

Die IL Anfragen bestehen aus drei Kategorien. Die Anfragen der Kategorie IL-1 und IL-2 sind gebunden, das heißt sie beginnen an einer bestimmten IRI, die, wie in den Watdiv Basic Anfragen, durch eine Variable bestimmt wird. Die IL-1 Anfragen beginnen bei einem wsdbm:User, während IL-2 Anfragen bei einem wsdbm:Retailer beginnen. IL-3 Anfragen sind komplett ungebunden und liefern daher enorm große Ergebnismengen.

Jede Anfragenkategorie besteht aus sechs Anfragen, welche aus einem linearen Basic Graph Pattern bestehen. Die Anfragen beginnen mit einem Basic Graph Pattern mit einem Durchmesser der Länge fünf. Bis zu einer Länge von zehn wird für jede weitere Anfrage ein weiteres Triple Pattern an die bestehende Anfrage angehängt. Die Namen

der IL Anfragen bilden sich wie folgt: IL-*Kategorie-Pfadlänge*. Die einzelnen Anfragen können in Anhang B eingesehen werden.

Die Konfigurationen der Testläufe sind die selben wie bei den WatDiv Basic Anfragen. Die Incremental Linear Anfragen wurden ebenfalls auf den vier Datensätzen mit dem Skalierungsfaktor 10, 100, 1000 und 10000 ausgeführt. Ebenso wurden für den Skalierungsfaktor 10000 wieder die normale und modifizierte Version mit ein- und ausgeschaltetem HDFS Cache ausgeführt. Die Ergebnisse der Testläufe befinden sich in Tabelle 4.

Die mittleren Laufzeiten der Anfragen der selben Kategorie und Durchmesser sind in Abbildung 6 dargestellt. Da die Ergebnisse sich teils um ein bis zwei Größenordnungen unterscheiden, wird auch hier eine logarithmische Skala verwendet.

Die Ordnung der mittleren Laufzeiten der Kategorie IL-1 gleicht den mittleren Laufzeiten der Watdiv Basic Linear Anfragen. S2RDF Big Table dominiert und Sempala Property Table und Single Table sind ungefähr gleich schnell. Ebenso verhält es sich mit der Kategorie IL-2. Allerdings zeigt die Single Table in IL-3 eine erstaunliche relative Performance. Auch zeigen die mittleren Laufzeiten über die Pfadlänge, mit Ausnahme der Pfadlänge acht, kein auffälliges Verhalten. Einzig das Mittel der Pfadlänge acht AM-8 zeigt die selbe Ordnung wie AM-IL-3.

Ein Blick in die detailliertere Abbildung 7 der mittleren Laufzeiten der einzelnen Anfragen zeigt die außerordentlich schlechte relative Performance der Big Table bei IL-3-8. Das Problem scheint die Menge der Daten zu sein [4]. IL-3-8 ergibt mit etwa 25 Milliarden Ergebnissen den maximalen Betrag aller Testläufe. Die Laufzeiten aller Systeme

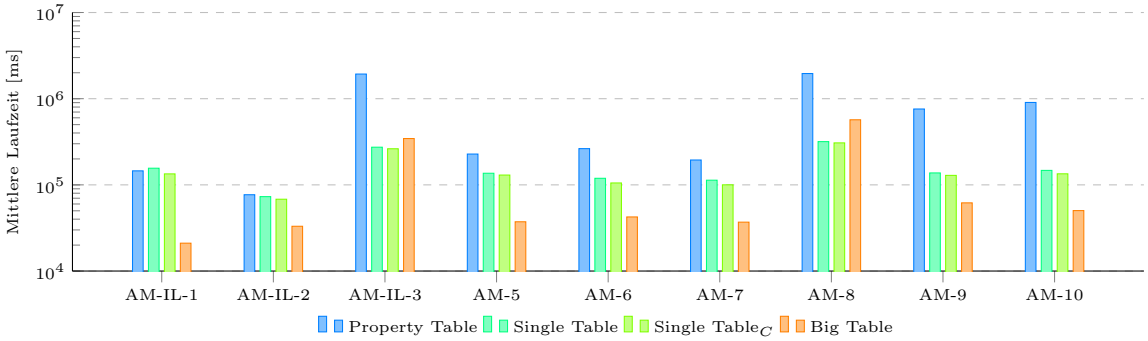

Abbildung 6: Mittlere Laufzeiten der WatDiv IL Anfragen.

korrelieren mit dem Betrag der Ergebnismenge, jedoch scheint die Menge der Daten auf Spark einen größeren Einfluss zu haben als auf Impala.

Interessant ist auch der Vergleich der Systeme innerhalb der Spark Engine. S2RDF Big Table schneidet im Vergleich zum S2RDF ExtVP besser ab, je größer die Ergebnismenge ist. Das zeigt, dass für große Ergebnismengen Impala nicht nur die besser geeignete Engine ist, sondern auch, dass das tripelorientierte Datenmodell der Impala Single Table beziehungsweise S2RDF Big Table mit großen Datenmengen performanter arbeitet als S2RDF ExtVP.

Eine weiter Auffälligkeit ist die Anfrage IL-2-5. Auch hier ist die relative Performance der Big Table ungewöhnlich schlecht. Bezüglich ExtVP wird in [4] erklärt, dass das darauf zurückzuführen ist, dass ExtVP bei den aufeinanderfolgenden, identischen Prädikaten der letzten zwei Tripel der Anfrage IL-2-5 keinen Gewinn durch Selektivität der Joins machen kann. Dieses Problem hat die Single Table nicht, da auch SO, OS und SS Relationen zu dem Prädikat selbst gehalten werden und somit auch bei aufeinanderfolgenden, identischen Prädikaten im BGP eine optimale Selektivität erreicht werden kann. Da dies auch für die Big Table gilt, scheint der Betrag der Zwischenergebnisse für Spark einen großen Einfluss zu haben.

Bei grober Betrachtung der Laufzeiten aller IL-3 Anfragen auf SF10000 in Tabelle 4 wird der Einfluss der Modifikation der Sempala Single Table deutlich. Mit bloßem Auge kann man beim Querlesen die Unterschiede der Größenordnungen erkennen, durch die sich Laufzeiten der Single Table und die der modifizierten Single Table$_M$ unterscheiden.

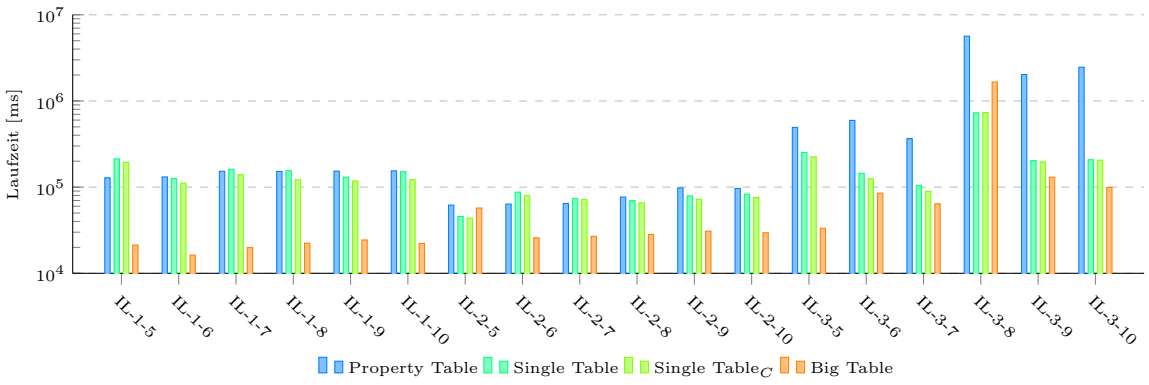

Abbildung 7: Laufzeiten der WatDiv IL Anfragen.

Hier wird noch einmal klar, dass der direkte Vergleich der anderen Systeme mit den unmodifizierten Sempala Single Table CTAS Varianten nicht sinnvoll ist.

Dieser Effekt kann man auch bei der WatDiv Basic C3 Anfrage mit, im Vergleich zu den IL-3, relativ kleinen Ergebnismengen beobachtet werden. Die ungefähr 42 Millionen Ergebnisse heben die Abweichung der Laufzeit der CTAS Variante im Vergleich zur COUNT Variante erheblich. Im Mittel sind die Laufzeiten der Single Table 13% höher als die der Single Table$_M$. Bei der Anfrage C3 steigt dieses Verhältnis auf 45% an.

Über alle Testreihen hinweg war der Einsatz des HDFS Caching kaum zu bemerken. Das könnte daran liegen, dass die komplette Tabelle in den Kernel Cache passt und die Daten auch ohne das HDFS Caching durch das Betriebssystem bereit gehalten werden. Die Vorzüge des HDFS Caching werden spürbarer, wenn die Datenmenge den freien Cache des Betriebssystems übersteigt und viele Anfragen parallel laufen. Unter diesen Umständen würde der Cache ständig mit verschiedenen Daten gefüllt und der Effekt des Caches geht verloren. In diesem Fall kann mit dem HFDS Cache ein statischer Cache einer häufig gebrauchten Partiton erzwungen werden.

5. Fazit

Die Single Table ist sehr sparsam im Speicherverbrauch. Im Vergleich zur Property Table wird nur die Hälfte des Speichers verbraucht. Ext VP benötigt gleich drei mal so viel Speicher. Spielt Speicher eine Rolle, zum Beispiel weil die komplette Tabelle in den Cache soll, dann ist die Singletable eine gute Wahl.

Was die Laufzeiten betrifft, ist die Single Table im Vergleich zur Property Table durchweg eine dominante Alternative. Im Mittel kann die Single Table die Laufzeit der Property Table schlagen und ist im Bereich sehr großer Ergebnismengen sogar eine ganze Größenordung schneller. ExtVP und ExtVP BigTable sind zwar im Mittel schneller, jedoch kann die Single Table im Bereich großer Ergebnismengen doppelt so schnell antworten wie die ExtVP Big Table und sogar drei mal so schnell wie das ExtVP.

Diese Situation könnte sich allerdings wieder zugunsten der Property Table ändern. Seit Impala 2.3 werden Complex Types unterstützt, welche der Property Table über einige

Probleme hinweghelfen können. Mit den verschachtelten Daten kann erreicht werden, dass der Join die Daten nicht mehr vervielfacht und die Zeilenanzahl konstant bleibt. Aber auch die Laufzeiten können damit in den Griff bekommen werden, denn kleinere Daten bedeuteuten kleinere Netzwerkbelastung und weniger Arbeit für den Join Prozess.

Abschließend kann man sagen, dass das Datenmodell Single Table sich als sehr guten Ersatz für die Property Table herausstellt. Ebenso kann es unter gewissen Umständen auch als Alternative für S2RDF in Betracht gezogen werden.

Literatur

[1] Kornacker, Marcel ; Behm, Alexander ; Bittorf, Victor ; Bobrovytsky, Taras ; Ching, Casey ; Choi, Alan ; Erickson, Justin ; Grund, Martin ; Hecht, Daniel ; Jacobs, Matthew u. a.: Impala: A Modern, Open-Source SQL Engine for Hadoop. In: *Biennial Conference on Innovative Data Systems Research*, 2015

[2] Melnik, Sergey ; Gubarev, Andrey ; Long, Jing J. ; Romer, Geoffrey ; Shivakumar, Shiva ; Tolton, Matt ; Vassilakis, Theo: Dremel: Interactive Analysis of Web-scale Datasets. In: *Proc. VLDB Endow.* 3 (2010), September, Nr. 1-2, S. 330–339. – URL `http://dx.doi.org/10.14778/1920841.1920886`. – ISSN 2150-8097

[3] Schätzle, Alexander ; Przyjaciel-Zablocki, Martin ; Neu, Anthony ; Lausen, Georg: Sempala: Interactive SPARQL Query Processing on Hadoop. In: *The Semantic Web – ISWC 2014: 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I* (2014), S. 164–179

[4] Schätzle, Alexander ; Przyjaciel-Zablocki, Martin ; Skilevic, Simon ; Lausen, Georg: S2RDF: RDF Querying with SPARQL on Spark. In: *Proceedings of the VLDB Endowment* (2016), S. 804–815

[5] Skilevic, Simon: *S2RDF: Distributed in-memory execution of SPARQL queries using Apache Spark SQL and Extended Vertical Partitioning*, Albert-Ludwigs-Universität Freiburg, Diplomarbeit, 2015

A. WatDiv Basic Queries

A.1. WatDiv Linear Queries

Listing 2: L1

```
#mapping v1 wsdbm:Website uniform
SELECT ?v0 ?v2 ?v3
WHERE {
 ?v0 wsdbm:subscribes %v1% .
 ?v2 sorg:caption ?v3 .
 ?v0 wsdbm:likes ?v2 .
}
```

Listing 3: L2

```
#mapping v0 wsdbm:City uniform
SELECT ?v1 ?v2
WHERE {
 %v0% gn:parentCountry ?v1 .
 ?v2 wsdbm:likes wsdbm:Product0 .
 ?v2 sorg:nationality ?v1 .
}
```

Listing 4: L3

```
#mapping v2 wsdbm:Website uniform
SELECT ?v0 ?v1
WHERE {
 ?v0 wsdbm:likes ?v1 .
 ?v0 wsdbm:subscribes %v2% .
}
```

Listing 5: L4

```
#mapping v1 wsdbm:Topic uniform
SELECT ?v0 ?v2
WHERE {
 ?v0 og:tag %v1% .
 ?v0 sorg:caption ?v2 .
}
```

Listing 6: L5

```
#mapping v2 wsdbm:City uniform
SELECT ?v0 ?v1 ?v3
WHERE {
 ?v0 sorg:jobTitle ?v1 .
 %v2% gn:parentCountry ?v3 .
 ?v0 sorg:nationality ?v3 .
}
```

A.2. WatDiv Star Queries

Listing 7: S1

```
#mapping v2 wsdbm:Retailer uniform
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 ?v9
WHERE {
 ?v0 gr:includes ?v1 .
 %v2% gr:offers ?v0 .
 ?v0 gr:price ?v3 .
 ?v0 gr:serialNumber ?v4 .
 ?v0 gr:validFrom ?v5 .
 ?v0 gr:validThrough ?v6 .
 ?v0 sorg:eligibleQuantity ?v7 .
 ?v0 sorg:eligibleRegion ?v8 .
 ?v0 sorg:priceValidUntil ?v9 .
}
```

Listing 8: S2

```
#mapping v2 wsdbm:Country uniform
SELECT ?v0 ?v1 ?v3
WHERE {
 ?v0 dc:Location ?v1 .
 ?v0 sorg:nationality %v2% .
 ?v0 wsdbm:gender ?v3 .
 ?v0 rdf:type wsdbm:Role2 .
}
```

Listing 9: S3

```
#mapping v1 wsdbm:ProductCategory uniform
SELECT ?v0 ?v2 ?v3 ?v4
WHERE {
 ?v0 rdf:type %v1% .
 ?v0 sorg:caption ?v2 .
 ?v0 wsdbm:hasGenre ?v3 .
 ?v0 sorg:publisher ?v4 .
}
```

Listing 10: S4

```
#mapping v1 wsdbm:AgeGroup uniform
SELECT ?v0 ?v2 ?v3
WHERE {
 ?v0 foaf:age %v1% .
 ?v0 foaf:familyName ?v2 .
 ?v3 mo:artist ?v0 .
 ?v0 sorg:nationality wsdbm:Country1 .
}
```

Listing 11: S5

```
#mapping v1 wsdbm:ProductCategory uniform
SELECT ?v0 ?v2 ?v3
WHERE {
 ?v0 rdf:type %v1% .
 ?v0 sorg:description ?v2 .
 ?v0 sorg:keywords ?v3 .
 ?v0 sorg:language wsdbm:Language0 .
}
```

Listing 12: S6

```
#mapping v3 wsdbm:SubGenre uniform
SELECT ?v0 ?v1 ?v2
WHERE {
 ?v0 mo:conductor ?v1 .
 ?v0 rdf:type ?v2 .
 ?v0 wsdbm:hasGenre %v3% .
}
```

Listing 13: S7

```
#mapping v3 wsdbm:User uniform
SELECT ?v0 ?v1 ?v2
WHERE {
 ?v0 rdf:type ?v1 .
 ?v0 sorg:text ?v2 .
 %v3% wsdbm:likes ?v0 .
}
```

A.3. WatDiv Snowflake Queries

Listing 14: F1

```
#mapping v1 wsdbm:Topic uniform
SELECT ?v0 ?v2 ?v3 ?v4 ?v5
WHERE {
 ?v0 og:tag %v1% .
 ?v0 rdf:type ?v2 .
 ?v3 sorg:trailer ?v4 .
 ?v3 sorg:keywords ?v5 .
 ?v3 wsdbm:hasGenre ?v0 .
 ?v3 rdf:type wsdbm:ProductCategory2 .
}
```

Listing 15: F2

```
#mapping v8 wsdbm:SubGenre uniform
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7
WHERE {
 ?v0 foaf:homepage ?v1 .
 ?v0 og:title ?v2 .
 ?v0 rdf:type ?v3 .
 ?v0 sorg:caption ?v4 .
 ?v0 sorg:description ?v5 .
 ?v1 sorg:url ?v6 .
 ?v1 wsdbm:hits ?v7 .
 ?v0 wsdbm:hasGenre %v8% .
}
```

Listing 16: F3

```
#mapping v3 wsdbm:SubGenre uniform
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6
WHERE {
 ?v0 sorg:contentRating ?v1 .
 ?v0 sorg:contentSize ?v2 .
 ?v0 wsdbm:hasGenre %v3% .
 ?v4 wsdbm:makesPurchase ?v5 .
 ?v5 wsdbm:purchaseDate ?v6 .
 ?v5 wsdbm:purchaseFor ?v0 .
}
```

Listing 17: F4

```
#mapping v3 wsdbm:Topic uniform
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7 ?v8
WHERE {
 ?v0 foaf:homepage ?v1 .
 ?v2 gr:includes ?v0 .
 ?v0 og:tag %v3% .
 ?v0 sorg:description ?v4 .
 ?v0 sorg:contentSize ?v8 .
 ?v1 sorg:url ?v5 .
 ?v1 wsdbm:hits ?v6 .
 ?v1 sorg:language wsdbm:Language0 .
 ?v7 wsdbm:likes ?v0 .
}
```

Listing 18: F5

```
#mapping v2 wsdbm:Retailer uniform
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6
WHERE {
 ?v0 gr:includes ?v1 .
 %v2% gr:offers ?v0 .
 ?v0 gr:price ?v3 .
 ?v0 gr:validThrough ?v4 .
 ?v1 og:title ?v5 .
 ?v1 rdf:type ?v6 .
}
```

A.4. WatDiv Complex Queries

Listing 19: C1

```
SELECT ?v0 ?v4 ?v6 ?v7
WHERE {
 ?v0 sorg:caption ?v1 .
 ?v0 sorg:text ?v2 .
 ?v0 sorg:contentRating ?v3 .
 ?v0 rev:hasReview ?v4 .
 ?v4 rev:title ?v5 .
 ?v4 rev:reviewer ?v6 .
 ?v7 sorg:actor ?v6 .
 ?v7 sorg:language ?v8 .
}
```

Listing 20: C2

```
SELECT ?v0 ?v3 ?v4 ?v8
WHERE {
 ?v0 sorg:legalName ?v1 .
 ?v0 gr:offers ?v2 .
 ?v2 sorg:eligibleRegion wsdbm:Country5 .
 ?v2 gr:includes ?v3 .
 ?v4 sorg:jobTitle ?v5 .
 ?v4 foaf:homepage ?v6 .
 ?v4 wsdbm:makesPurchase ?v7 .
 ?v7 wsdbm:purchaseFor ?v3 .
 ?v3 rev:hasReview ?v8 .
 ?v8 rev:totalVotes ?v9 .
}
```

Listing 21: C3

```
SELECT ?v0
WHERE {
 ?v0 wsdbm:likes ?v1 .
 ?v0 wsdbm:friendOf ?v2 .
 ?v0 dc:Location ?v3 .
 ?v0 foaf:age ?v4 .
 ?v0 wsdbm:gender ?v5 .
 ?v0 foaf:givenName ?v6 .
}
```

B. WatDiv Increasing Linear Queries

B.1. WatDiv IL-1 Queries

Listing 22: IL-1-5

```
#mapping v0 wsdbm:User uniform
SELECT ?v1 ?v2 ?v3 ?v4 ?v5
WHERE {
 %v0% wsdbm:follows ?v1 .
 ?v1 wsdbm:likes ?v2 .
 ?v2 rev:hasReview ?v3 .
 ?v3 rev:reviewer ?v4 .
 ?v4 wsdbm:friendOf ?v5 .
}
```

Listing 23: IL-1-7

```
#mapping v0 wsdbm:User uniform
SELECT ?v1 ?v2 ?v3 ?v4 ?v5 ?v6
WHERE {
 %v0% wsdbm:follows ?v1 .
 ?v1 wsdbm:likes ?v2 .
 ?v2 rev:hasReview ?v3 .
 ?v3 rev:reviewer ?v4 .
 ?v4 wsdbm:friendOf ?v5 .
 ?v5 wsdbm:makesPurchase ?v6 .
}
```

Listing 24: IL-1-7

```
#mapping v0 wsdbm:User uniform
SELECT ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7
WHERE {
 %v0% wsdbm:follows ?v1 .
 ?v1 wsdbm:likes ?v2 .
 ?v2 rev:hasReview ?v3 .
 ?v3 rev:reviewer ?v4 .
 ?v4 wsdbm:friendOf ?v5 .
 ?v5 wsdbm:makesPurchase ?v6 .
 ?v6 wsdbm:purchaseFor ?v7 .
}
```

Listing 25: IL-1-8

```
#mapping v0 wsdbm:User uniform
SELECT ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8
WHERE {
 %v0% wsdbm:follows ?v1 .
 ?v1 wsdbm:likes ?v2 .
 ?v2 rev:hasReview ?v3 .
 ?v3 rev:reviewer ?v4 .
 ?v4 wsdbm:friendOf ?v5 .
 ?v5 wsdbm:makesPurchase ?v6 .
 ?v6 wsdbm:purchaseFor ?v7 .
 ?v7 sorg:author ?v8 .
}
```

Listing 26: IL-1-9

```
#mapping v0 wsdbm:User uniform
SELECT ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 ?v9
WHERE {
 %v0% wsdbm:follows ?v1 .
 ?v1 wsdbm:likes ?v2 .
 ?v2 rev:hasReview ?v3 .
 ?v3 rev:reviewer ?v4 .
 ?v4 wsdbm:friendOf ?v5 .
 ?v5 wsdbm:makesPurchase ?v6 .
 ?v6 wsdbm:purchaseFor ?v7 .
 ?v7 sorg:author ?v8 .
 ?v8 dc:Location ?v9 .
}
```

Listing 27: IL-1-10

```
#mapping v0 wsdbm:User uniform
SELECT ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 ?v9
     ?v10
WHERE {
 %v0% wsdbm:follows ?v1 .
 ?v1 wsdbm:likes ?v2 .
 ?v2 rev:hasReview ?v3 .
 ?v3 rev:reviewer ?v4 .
 ?v4 wsdbm:friendOf ?v5 .
 ?v5 wsdbm:makesPurchase ?v6 .
 ?v6 wsdbm:purchaseFor ?v7 .
 ?v7 sorg:author ?v8 .
 ?v8 dc:Location ?v9 .
 ?v9 gn:parentCountry ?v10 .
}
```

B.2. WatDiv IL-2 Queries

Listing 28: IL-2-5

```
#mapping v0 wsdbm:Retailer uniform
SELECT ?v1 ?v2 ?v3 ?v4 ?v5
WHERE {
 %v0% gr:offers ?v1 .
 ?v1 gr:includes ?v2 .
 ?v2 sorg:director ?v3 .
 ?v3 wsdbm:friendOf ?v4 .
 ?v4 wsdbm:friendOf ?v5 .
}
```

Listing 29: IL-2-7

```
#mapping v0 wsdbm:Retailer uniform
SELECT ?v1 ?v2 ?v3 ?v4 ?v5 ?v6
WHERE {
 %v0% gr:offers ?v1 .
 ?v1 gr:includes ?v2 .
 ?v2 sorg:director ?v3 .
 ?v3 wsdbm:friendOf ?v4 .
 ?v4 wsdbm:friendOf ?v5 .
 ?v5 wsdbm:likes ?v6 .
}
```

Listing 30: IL-2-7

```
#mapping v0 wsdbm:Retailer uniform
SELECT ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7
WHERE {
 %v0% gr:offers ?v1 .
 ?v1 gr:includes ?v2 .
 ?v2 sorg:director ?v3 .
 ?v3 wsdbm:friendOf ?v4 .
```

```
 ?v4 wsdbm:friendOf ?v5 .
 ?v5 wsdbm:likes ?v6 .
 ?v6 sorg:editor ?v7 .
}
```

Listing 31: IL-2-8

```
#mapping v0 wsdbm:Retailer uniform
SELECT ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8
WHERE {
 %v0% gr:offers ?v1 .
 ?v1 gr:includes ?v2 .
 ?v2 sorg:director ?v3 .
 ?v3 wsdbm:friendOf ?v4 .
 ?v4 wsdbm:friendOf ?v5 .
 ?v5 wsdbm:likes ?v6 .
 ?v6 sorg:editor ?v7 .
 ?v7 wsdbm:makesPurchase ?v8 .
}
```

Listing 32: IL-2-9

```
#mapping v0 wsdbm:Retailer uniform
SELECT ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 ?v9
WHERE {
 %v0% gr:offers ?v1 .
 ?v1 gr:includes ?v2 .
 ?v2 sorg:director ?v3 .
 ?v3 wsdbm:friendOf ?v4 .
 ?v4 wsdbm:friendOf ?v5 .
 ?v5 wsdbm:likes ?v6 .
 ?v6 sorg:editor ?v7 .
 ?v7 wsdbm:makesPurchase ?v8 .
 ?v8 wsdbm:purchaseFor ?v9 .
}
```

Listing 33: IL-2-10

```
#mapping v0 wsdbm:Retailer uniform
SELECT ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 ?v9
     ?v10
WHERE {
 %v0% gr:offers ?v1 .
 ?v1 gr:includes ?v2 .
 ?v2 sorg:director ?v3 .
 ?v3 wsdbm:friendOf ?v4 .
 ?v4 wsdbm:friendOf ?v5 .
 ?v5 wsdbm:likes ?v6 .
 ?v6 sorg:editor ?v7 .
 ?v7 wsdbm:makesPurchase ?v8 .
 ?v8 wsdbm:purchaseFor ?v9 .
 ?v9 sorg:caption ?v10 .
}
```

B.3. WatDiv IL-3 Queries

Listing 34: IL-3-5

```
SELECT ?v0 ?v1 ?v2 ?v3 ?v4 ?v5
WHERE {
 ?v0 gr:offers ?v1 .
 ?v1 gr:includes ?v2 .
 ?v2 rev:hasReview ?v3 .
 ?v3 rev:reviewer ?v4 .
 ?v4 wsdbm:friendOf ?v5 .
}
```

Listing 35: IL-3-7

```
SELECT ?v0 ?v1 ?v2 ?v3 ?v4 ?v5 ?v6
WHERE {
 ?v0 gr:offers ?v1 .
 ?v1 gr:includes ?v2 .
 ?v2 rev:hasReview ?v3 .
 ?v3 rev:reviewer ?v4 .
 ?v4 wsdbm:friendOf ?v5 .
 ?v5 wsdbm:likes ?v6 .
}
```

Listing 36: IL-3-7

```
SELECT ?v0 ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7
WHERE {
 ?v0 gr:offers ?v1 .
 ?v1 gr:includes ?v2 .
 ?v2 rev:hasReview ?v3 .
 ?v3 rev:reviewer ?v4 .
 ?v4 wsdbm:friendOf ?v5 .
 ?v5 wsdbm:likes ?v6 .
 ?v6 sorg:author ?v7 .
}
```

Listing 37: IL-3-8

```
SELECT ?v0 ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8
WHERE {
 ?v0 gr:offers ?v1 .
 ?v1 gr:includes ?v2 .
 ?v2 rev:hasReview ?v3 .
 ?v3 rev:reviewer ?v4 .
 ?v4 wsdbm:friendOf ?v5 .
 ?v5 wsdbm:likes ?v6 .
 ?v6 sorg:author ?v7 .
 ?v7 wsdbm:follows ?v8 .
}
```

Listing 38: IL-3-9

```
SELECT ?v0 ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8
     ?v9
WHERE {
 ?v0 gr:offers ?v1 .
 ?v1 gr:includes ?v2 .
 ?v2 rev:hasReview ?v3 .
 ?v3 rev:reviewer ?v4 .
 ?v4 wsdbm:friendOf ?v5 .
 ?v5 wsdbm:likes ?v6 .
 ?v6 sorg:author ?v7 .
 ?v7 wsdbm:follows ?v8 .
 ?v8 foaf:homepage ?v9 .
}
```

Listing 39: IL-3-10

```
SELECT ?v0 ?v1 ?v2 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8
     ?v9 ?v10
WHERE {
 ?v0 gr:offers ?v1 .
 ?v1 gr:includes ?v2 .
 ?v2 rev:hasReview ?v3 .
 ?v3 rev:reviewer ?v4 .
 ?v4 wsdbm:friendOf ?v5 .
 ?v5 wsdbm:likes ?v6 .
 ?v6 sorg:author ?v7 .
 ?v7 wsdbm:follows ?v8 .
 ?v8 foaf:homepage ?v9 .
 ?v9 sorg:language ?v10 .
}
```