Humboldt-Universität zu Berlin
Institut für Informatik
Wissensmanagement in der Bioinformatik

(a) 1968 Chicago Riots; Quelle: https://afrocityblog.wordpress.com/tag/chicago-police-reunion/

Pig
its natural Env
and _ Latin

Seminar »Large Scale Data Analysis«
Sommersemester 2012

Justin Freywald

3. Fassung vom 19. September 2013 v1.0

Inhaltsverzeichnis

1 Einleitung

[...] the map-reduce paradigm is too low-level and rigid, and leads to a great deal of custom user code that is hard to maintain, and reuse.

Olston, C. et al[6, p. 1099]

Pig soll die Analyse von *big data* erleichtern (in [3] geht Alan Gates ausführlich darauf ein). Unter *big data* sind Datenmengen zu verstehen, die auf Grund ihrer Größe im Tera- und Petabereich auf tausenden Servern verteilt vorliegen[7, p. 3] und deswegen innerhalb des sich bildenden Netzwerks sinnvollerweise parallel verarbeiten werden müssen. Ein solches Netzwerk wird auch durch das Modewort *Cloud computing* beschrieben, wobei hier in allgemeiner Bedeutung Ressourcen im Kontext eines Dienstes über ein Netzwerk – z.B. das Internet – verbunden sind. Nach außen bietet die Wolke (Cloud) transparente Dienste von Infrastruktur, Plattformen und Software an, die auf virtuellen Infrastrukturen laufen und – nach Bedarf – auf diesen skalieren.

Einer der Vorteile von Pig ist, daß die Analyse von z. B. Logdateien ohne einen Import in das Datenformat einer Applikation wie das eines *DBMS* (Datenbankmanagementsysteme) durchführbar ist: Pig arbeitet auf unstrukturierten, geschachtelten oder relationalen Daten.[3] Daten können ohne assoziierte *Schemata* geladen werden. Schemata definieren Struktur bei der Dateneinbindung, gehen dem Kompiler zur Hand und erleichtern syntaktisch den Zugriff auf das *geschachtelte Datenmodell* (Nested Data Modell), auf das später detailiert eingegangen wird. Eine

gewisse Strukturierung ist auch in Pig von Nöten, denn vollkommen unstrukturierte Daten haben allgemein keine gültige Interpretation und sind somit ohne jedwede Information. Weiter setzt Pig nur rein-lesbare Arbeitslast voraus und bietet weder transaktionale Sicherheit noch Integritätseinschränkungen.

Zuerst werde ich im 2. Kapitel auf die ”natürliche Umgebung eines Pig” eingehen. Dann werde ich mich im 3. Kapitel mit der Architektur von Pig befassen und das geschachtelte Datenmodell einführen. *Pig Latin* ist die relationale Datenflußsprache,[2] mit der Programme für Pig geschrieben werden. Zuvorderst konzentriert sich diese Arbeit auf Pig Latin. Im 4. Kapitel werde ich mich damit ausschließlich befassen und als Fundament die Operatoren, der Sprache Syntax und Semantik festigen und stets die Abbildung auf das MapReduce-Paradigma angeben.

Wem nützt es?[1]

- Nokia zur Analyse von Logs und Datenbankauszügen,
- LinkedIn zur Analyse von ” People You May Know” u.Ä.,
- AOL zur Analyse und Stapelverarbeitung für verschiedene Applikationen,
- Yahoo!: mehr als 40% der Jobs laufen unter Pig.

> Pig has been widely adopted in Yahoo [...] with thousands of jobs executed daily.[4, p. 1415]

Das 5. Kapitel schließt die Arbeit ab und geht kurz auf zwei Konkurrenten von Pig ein.

[1] https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=26120079

2 ”Die natürliche Umgebung”

In diesem Kapitel behandle ich das grundlegende MapReduce-Paradigma und Hadoop, welches das Paradigma verwendet und Teil der ”natürliche Umgebung” von Pig ist.

2.1 MapReduce

Das MapReduce-Paradigma wurde von Google 2004 durch das Papier [1] vorgestellt. MapReduce bietet genau zwei Funktionen zweiter Ordnung, namentlich Map und Reduce, die jeweils als Eingabe eine benutzerdefinierte Funktion erster Ordnung mit nur einer Eingabe erhalten. Die Abfolge der Schritte Map und Reduce bildet Phasen, die sequentiell ausgeführt werden und im folgenden MapReduce-Pläne genannt werden. Die Parallelisierung der MapReduce-Pläne erfolgt durch Partionierung der Daten und liegt in Verantwortung der *Ausführungsengine*; bei Pig gewährleistet dies momentan Hadoop. Die Entwickler von Pig stellen jedoch klar, daß als *Backend* auch andere Frameworks unterstützt werden können.

Die Signatur beider Funktionen ist definiert wie folgt: Map erhält ein Schlüssel-Wert-Paar und liefert eine Menge solcher Paare zurück: map m: $(K \times V)^+ \rightarrow (\dot{K} \times \dot{V})^*$. Reduce erhält alle Schlüssel-Wert-Paare des gleichen Schlüssels aller Rechner und liefert eine Menge von Ergebniswerten: reduce r: $(\dot{K} \times \dot{V}^*)^+ \rightarrow (\ddot{K} \times \ddot{V})^*$. Das MapReduce-Paradigma bietet eine ”natürliche” Parallelisierung der Map- und Reduce-Schritte durch Datenparallelität an.

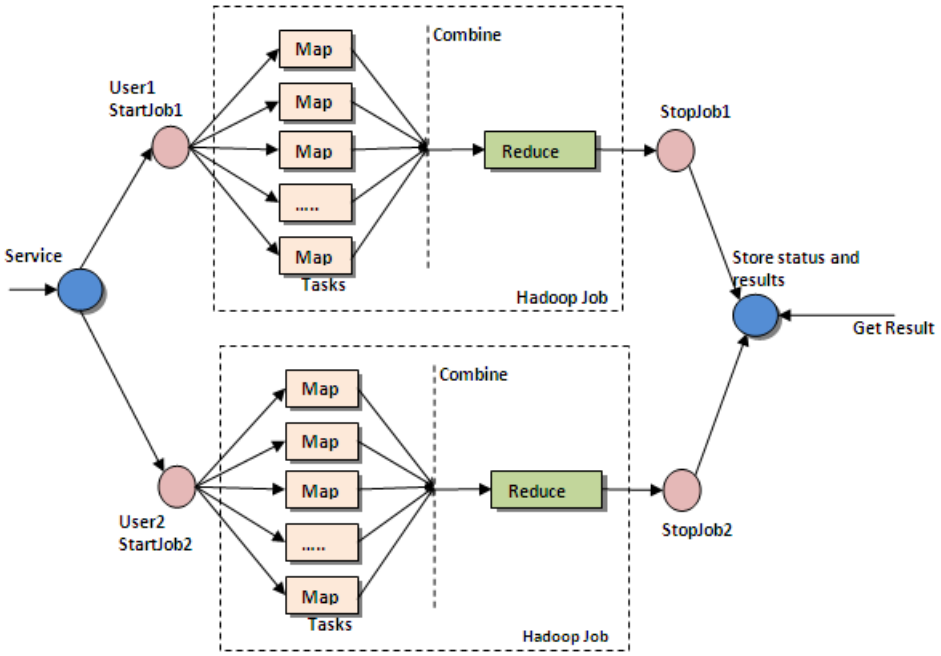

Abb. 2.1.: MapReduce-Ausführung[1]

Weiterhin ist es üblich, eine zusätzliche Funktion zwischen Map und Reduce einzuführen: Falls möglich, nimmt Combine eine folgende Aggregation im Reduce vorweg und führt eine solche Aggregation bereits lokal aus, noch bevor die Ergebnisse von Map mit den gleichen Schlüsseln insgesamt zu einer Reduce-Instanz versendet werden. Bei nicht-*algebraischen* Operationen ist Combine nicht möglich. Eine *algebraische* Operation auf Daten zeichnet sich dadurch aus, daß ihre Berechnung auf Partitionen der Gesamtdatenmenge rekursiv definiert werden kann: ”Count” ist algebraisch, ”Median” nicht.

Die Abbildung 2.1 zeigt exemplarisch eine Konfiguration, in der MapReduce-Programme laufen, ferner zwei Jobs, die aus einem einfachen MapReduce-Plan bestehen (eine MapReduce-Phase), möglicherweise liegt beiden Jobs der gleiche Plan zugrunde. Die Jobs der Benutzer werden isoliert ausgeführt und die Ergebnisse an

[1] http://map-reduce.wikispaces.asu.edu/

zentraler Stelle gesammelt (z. B. in einem verteilten Dateisystem). Jede MapReduce-Phase lässt sich aufteilen in verschiedene Funktionen: Map, Combine und Reduce.

Zu bemerken ist, daß ein Reduce eine globale Sortierung (oder Hashen) erfordert, um zu gewährleisten, daß alle gleichen Schlüssel genau in einem Reduce verarbeitet werden. Die Partionierung der Ausgangsdaten des Map-Schrittes, anschließende, aber nicht notwendige lokale Sortierung und Versendung über das Netzwerk heißt Shuffle.

Im Vergleich zu SQL fällt auf, daß Map auf einer Eingabe, JOIN jedoch zwei Eingaben verarbeitet. Daher ist die Abbildung von JOIN und ähnlichen Operationen auf das MapReduce-Paradigma unter Datenzunahme möglich, aber umständlich.

2.2 Hadoop

Hadoop ist ein Framework mit den Komponenten Hadoop Common, Hadoop MapReduce und *HDFS* (Hadoop Distributed File System). Mit den letzten beiden werde ich mich befassen.[2]

Hadoop MapReduce basiert auf dem MapReduce-Paradigma. HDFS ist ein *verteiltes Dateisystem*. Ein verteiltes Dateisystem verwaltet und repliziert Dateien und ist über die Speicher mehrerer Rechner gespannt. Die in der Architektur über Hadoop angesiedelte Software, z. B. (logische) MapReduce-Pläne, sind von der Dateiverwaltung losgelöst.

2.3 Hadoop's MapReduce

Jeder MapReduce-Cluster wird von einem *Jobtracker* koordiniert, der neue Jobs an Knoten bindet etc. Auf jedem solcher Knoten läuft ein *Tasktracker*, der in Eigenverantwortung Map- und Reduce-Aufgaben ausführt, die er vom Jobtracker zugewiesen bekommt, wie die Abbildung 2.2 verdeutlichen soll.[3]

[2] http://hadoop.apache.org/

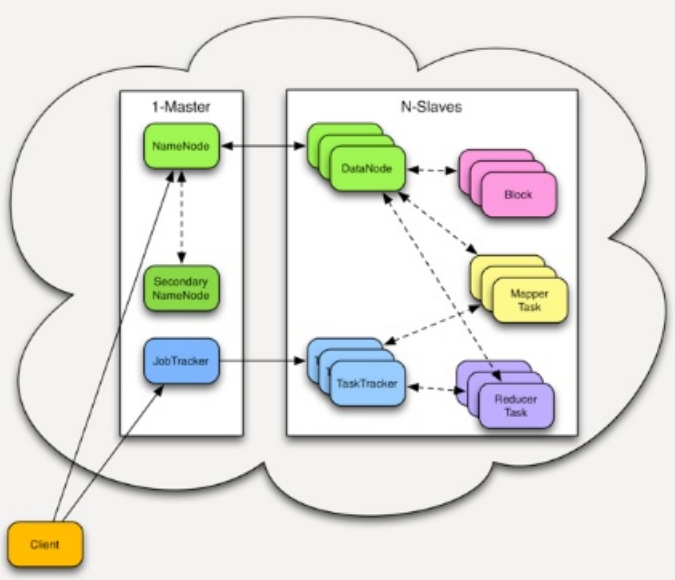

Abb. 2.2.: Hadoops Architektur[3]

Die Eingabe von Map sei eine im HDFS vorliegende Datei. Im Allgemeinen wird HDFS bei n Knoten die Datei auf möglichst n Knoten verteilen (mit fester Blockgröße), undzwar mindestens jeden Dateidatenblock zweifach redundant, d.h. jeder Block ist auf mindestens drei Instanzen verfügbar.[3] Jede Map-Funktion wird nun vorrangig auf dem Knoten ausgeführt, auf dem diese Datei physisch vorliegt (es wird der naheste, verfügbare Knoten gewählt).

Nun wird noch obigem Prinzip der *Lokalität* von Datenblöcken die Datei in *Input Splits* aufgeteilt, sodaß jede Map-Funktion genau einen Input Split verarbeitet und die Gesamtheit aller Input Splits alle Datenblöcke der Datei repräsentieren.

Der Parallelisierunsgrad eines Reduce-Schrittes wird durch die heterogene Verteilung der Schlüssel bestimmt und ist frei wählbar (auch Pig bietet diese Möglichkeit), die Anzahl der Map-Schritte wird nach unten durch die Anzahl der verfügbaren Knoten im HDFS beschränkt.

[3] http://www.slideshare.net/rawar/schweine-lateinvortrag/

3 Pig

3.1 Architektur

Abbildung 3.1 zeigt die Architektur und Umgebung von Pig, wobei ich Hive, HBase und HCatalog von Erläuterungen ausschliesse. Die Sprache Pig Latin gehört zu Pig. Pig kompiliert Pig-Latin-Programme in MapReduce-Pläne (logische und physikalische), welche von Hadoop auf beliebig vielen *Instanzen* ausgeführt und Daten auf HDFS gespeichert werden.[4, p. 1415] Jede Hadoop bekannte Maschine im Netzwerk ist eine solche Instanz.

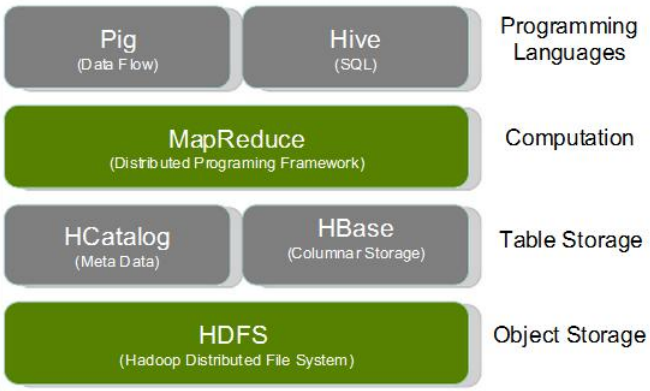

Abb. 3.1.: Architektur um Pig[1]

3.2 Pig

Pig ist eines der durch die Entwicklungsorganisation ”Yahoo! Research” entstandenes Projekt, lag 2007 für alle quelloffen vor und wurde später in das ”Apache Projekt” aufgenommen.[3]

Die Entwickler entwickelten auch Prinzipien,[2] die sie als Philosophie deuten, wohl aber eher ersteres darstellt, denn es verhält sich umgekehrt. Die Philosophie muß die Welt deuten, das ist ihre Aufgabe, die Gesamtheit der Welt zu erklären vermögen wohl folgende Prinzipien nicht:

”Pigs live anywhere”

Java-Threads[3] ersetzen Hadoop/MapReduce, wenn die Pig-Latin-Programme lokal ausgeführt werden, oder lassen sich in Hadoop als MapReduce-Plan ausgeführen. Weiterhin nutzt Pig derzeit Hadoop, kann aber auch für andere Frameworks implementiert werden.

”Pigs eat anything”

Pig verarbeitet jede Art von Daten: es arbeitet auf relationalen, geschachtelten oder unstrukturierten Daten. Die Ein-/Ausgabe ist nicht auf Dateien beschränkt, sondern auf z. B. Schlüssel-Werte-Speicher oder DBMS erweiterbar. Desweiteren erlaubt Pig *UDF* (benutzerdefinierte Funktionen derzeit in den Sprachen Java und Python). UDF können fast überall im Quellkode genutzt werden.[3]

Die Reihenfolge der Operatorenausführung eines Pig-Plans ist nicht zwingend sequentiell. Das heißt, daß der Kompiler in einigen Fällen optimieren kann – genug Informationen über die beteiligten Relationen vorausgesetzt.

Sei z.B. folgender Rumpf eines Programmes wie in Bsp. 3.1 gegeben und weiter isSpam eine UDF:

[1] http://www.analyticbridge.com/profiles/blogs/hadoop-technology-stack/, verändert

[2] http://pig.apache.org/philosophy.html

Bsp. 3.1: Optimierung von UDF

```
1 spam_urls=FILTER url BY isSpam(url);
2 culprit_urls=FILTER spam_urls BY pagerank>0.8;
```

”spam_urls” und ”culprit_urls” sind Relationen, deren Inhalt durch die rechte Seite bestimmt wird. Auf der rechten Seite ist *FILTER* eine Operation, die unter einer gegebenen Bedingung, die durch BY eingeleitet wird, Tupel einer Relation, hier ”url” aussortiert.

Der PageRank liegt hier bereits berechnet vor und die Operatorenreihenfolge beeinflusst daher das Ergebnis nicht.

Hier sollte der Optimierer zuerst die Bedingung ”pagerank>0.8” testen, falls einerseits isSpam eine aufwändige Operation ist, um somit bereits Tupel im Zwischenergebnis auszuschließen und andererseits durch die Bedingung genügend viele Tupel aussortiert werden, um die Menge verarbeiteter Daten zu reduzieren.

Leider ist diese Optimierungsproblem ungelöst, denn der Optimierer kann für allgemeine UDF – in Fällen defizitärer Information – kein Maß anlegen. In anderen Fällen ist die Optimierung möglich, aber teuer. Der Kompiler unterstützt die Umordnung der Operatoren – wie in diesem Beispiel gezeigt – bereits ohne UDF nicht, jedoch können UDF die Schnittstelle ”algebraic” implementieren, um dann vom Combiner zu profitieren.[3] Der manuelle Eingriff sei schneller als für das MapReduce-Paradigma typisch und notwendig, komplexere Map- und Reduce-Funktionen z. B. in Java für Hadoop anzupassen.[6, p. 1100] Der Optimierer ordnet in bestimmten Fällen einige Operationen im Plan um;[3] weiteres dazu blieb mir aber unbekannt.

Desweiteren bieten die Operatoren JOIN und GROUP die Möglichkeit, den Optimierer zu übergehen und spezialisierte Algorithmen zu erzwingen – angeboten werden derzeit vier JOIN- und zwei GROUP-Algorithmen.[2] Beide werden später vorgestellt.

[3] http://pig.apache.org/docs/r0.10.0/udf.html

3.3 Pig Latin

Pig Latin ist eine *Datenflußsprache*. Bekannte Konstrukte anderer Sprachen wie "if/for" fokussieren primär den Kontrollfluß mit dem Datenfluß als Seiteneffekt.[3] Umgekehrt stellen die Operatoren in einer Datenflußsprache den Datenfluß dar aus denen sich der Kontrollfluß als Seiteneffekt ergibt.

Intern werden die Operatoren in einen *DAG* (gerichteten, azyklischen Graph) angeordnet, wobei die Knoten Operatoren und Kanten Datenflüsse repräsentieren.[3] Die Operatoren geben an, wie Daten gelesen, verarbeitet und gespeichert werden.

Pig Latin bietet spezielle, hochsprachliche Operatoren, von manchen existieren Entsprechungen aus SQL. Sie ist *prozedural* – sequentielle Abarbeitung der Anweisungen – im Unterschied zu *deklarativen* SQL.[2] Deklarative Anweisungen erforderen für die Ergebnisberechnung nicht die Angabe eines Algorithmus, sondern geben das Ergebnis durch Struktur an (*was*), aber nicht, *wie* es ermittelt werden soll und es ist Aufgabe des Systems, das Ergebnis zu ermitteln.

Dadurch, daß Pig prozedural ist, komme es dem gemeinen Programmierer sehr gelegen. Weiterhin gibt es Relationen ("spam_urls" in Bsp. 3.1 wäre eine solche Relation), die keine Variablen[3], sondern *handles* zu einer Multimenge sind.[6, p. 1103, 3.2.]

Hiermit ist offensichtlich geworden, inwiefern die Hochsprache Pig Latin eine Bereicherung darstellt: in Hadoop erfordern gewöhnliche Operationen wie Projektion und Selektion eine selbstständig erfolgte Implementierung, Pig bietet dafür in seiner Datenflußsprache bereits "fertige" Operatoren.

3.3.1 Geschachteltes Datenmodell

In Pig werden Daten durch geschachtelte Datenstrukturen repräsentiert. Skalare (Atome) werden in Tupel, Multimengen (Relationen) oder assoziativen Feldern

verschachtelt (wie Bsp. 3.2 zeigt). Diese seien für den Programmierer natürlicher als die normalisierte Form von Daten in DBMS.[6, p. 1101, 2.3.]

Bsp. 3.2: Geschachtelte Datenstrukturen

```
'Und Alice war gut'; 23; 42.0 --Skalare (auch: Atome)
('a', 42.0) --Tupel
{(), ('Alice'), (22.99,42.01)} --Multimengen
['fan of' → { ('me'),('you') }, 'age' → 20 ] --Assoziative Felder
```

Bemerkung: die Kommatatrennung von Atomen ist nicht Teil der Darstellung von Pig.

Atome sind Zahlen, Zeichenketten und Blobs – mit den Typen int, long, float, double, chararray, bytearray. Zahlen werden wie in Java notiert, exemplarisch wird ein long durch 1000L angegeben. *Skalare* setzen sich aus Atomen und anderen Skalaren zusammen. *Tupel* sind geordnete Felder von Typen, bei denen Duplikate erlaubt und Typen frei wählbar sind.

Multimengen (ich werde sie und Relationen synonym verwenden) sind Mengen von Tupeln – mit den Freiheitsgraden der Tupel und dem Zusatz der Unbeschränktheit der Anzahl der Felder eines Tupels. Multimengen sind die einzigen Typen, die u.U. nicht vollständig im Hauptspeicher materialisiert werden, da sie leicht dessen Kapazität überschreiten.[3]

Assoziative Felder sind Schlüssel-Wert-Paare mit der Bedingung eines atomaren Schlüssels und derzeit weiteren Einschränkung auf chararray.[3]

Der Typ *Null* existiert mit der gleichen Semantik wie in SQL: der Wert ist unbestimmt. Weiters gibt es keine Einschränkungen auf nicht-Null, jedes Feld kann Null werden.

Ein Schema kann Strukturen, wie gerade vorgestellt, beschreiben. In Bsp. 3.3 wird beim Laden von einer Datei ein Schema zu einer Relation assoziiert.

Bsp. 3.3: Schema

```
B = LOAD 'file' AS (name: chararray, age: int, money: float);
```

Pig nutzt es zur Optimierung und zur frühen Typprüfung. Wird ein Schema angegeben, werden die nicht-beschriebenen, zusätzlichen Felder eines Tupels abgeschnitten oder fehlende mit Null initialisiert.

Ein solches Schema muß keineswegs vollständig definiert sein; der Inhalt eines Tupels muß nicht weiter beschrieben werden, es reicht die Angabe, daß es ein Tupel ist.[3] Es bieten sich weitreichende Möglichkeiten auf geschachtelte Elemente in Strukturen zuzugreifen, wenn die Struktur bekannt ist. Bsp. 3.4 zeigt eine Relation mit gebundenem Schema und ist Beispiel für einfach-verschachtelte Daten. Die folgende Tabelle zeigt die Zugriffsyntax, um innerhalb geschachtelter Daten Elemente zu extrahieren.[6, p. 1102, 3.]. Sei t ein Tupel.

Bsp. 3.4: Zugriff

```
1 t=('alice', { ('lakers',1),('iPod',2) }, ['age' → 20] ) und Tupel ↙
    ↳ t=(f1:,f2:,f3:)
```

Ausdruckstyp	Beispiel	Wert für t
Konstante	'bob'	unabhängig von t
Feld nach Position	$0	'alice'
Feld nach Bezeichnung	f3	['age' → 20]
Projektion	f2.$0	('lakers'),('iPod')
Feldnachschlagung	f3#'age'	20
Funktionale Auswertung	SUM(f2.$1)	1+2=3
Bedingter Ausdruck	f3#'age'>18?'adult':'minor'	'adult'
Einebnung	FLATTEN(f2)	'lakers',1 'iPod',2

4 Pig Latin

In diesem Kapitel werde ich die Operatoren von Pig Latin vorstellen, die Vorstellung an geeigneter Stelle für ein Beispiel unterbrechen, dann zeitnah fortführen mit der Abbildung von Operatoren auf MapReduce.

4.1 Einführung

Zur Einführung von Pig Latin verwende ich Bsp. 4.1: sei eine Relation ”urls” mit Schema ”(url, category, pagerank)” gegeben. Folgender Kode zeigt eine SQL-Anfrage und ihre Entsprechung in Pig Latin.

Bsp. 4.1: Pig Latin vs. SQL

```
1  --SQL
2  SELECT category, AVG(pagerank)
3  FROM urls WHERE pagerank>0.2
4  GROUP BY category HAVING COUNT(*)>1000000;
5
6  --PIG LATIN
7  good_urls=FILTER urls BY pagerank>0.2;
8  groups=GROUP good_urls BY category;
9  big_groups=FILTER groups BY COUNT(good_urls)>1000000;
10 output=FOREACH big_groups GENERATE category, ↙
     ↳ AVG(good_urls.pagerank);
```

Aus allen nach ”urls.category” gruppierten Tupeln mit mehr als 10^6 Einträgen aus der Eingaberelation ”urls”, die der Bedingung ”urls.pagerank>0.2” genügen, wird eine Ausgaberelation ”output” erzeugt, die aus der jeweiligen ”big_groups.category” und des für die Gruppe durchschnittlichen ”good_urls.pagerank” besteht.

4.2 Grundlegende Operatoren

4.2.1 LOAD

LOAD verknüpft die Datei ”file” als Eingabe mit einer Relation (in Bsp. 4.2 A). ”myLoad” ist eine eigene Serialisierungsfunktion, die die semi- oder unstrukturierte Datei ”file” für Pig aufbereitet. Die USING-Angabe ist ebenso wie die Angabe des Schema mittels AS optional. Ohne USING wird der Serialisierer PigStorage verwendet, der aus jeder Zeile der Eingabedatei Tupel erzeugt, wobei jedes durch Tabulatoren getrennte Token ein neues Feld im Tupel wird. Das Ergebnis der Operation auf der rechten Seite ist dann eine Multimenge, die A zugewiesen wird. Dieses Schema gibt Name (”id”) und Typ (Zeichenkette) des ersten Tupels und das zweite Tupel namens ”time” an.

Bsp. 4.2: LOAD

```
A=LOAD 'file' USING myLoad() AS (id:chararray,time);
```

Hier kommt eine weitere Eigenschaft von Pig zum Tragen: verzögerte Auswertung (”Lazy Evaluation”). Erst wenn die Relation A oder eine auf A ”aufbauende” Relation gespeichert wird (STORE), wird die gesamte Pipeline ausgeführt und die mit LOAD spezifizierte Datei geladen, genauer: der logische Plan in einen physikalischen Plan kompiliert und über Hadoop ausgeführt.[6, p. 1106] Ebenso werden Typen erst dann implizit umgewandelt, wenn die Operationen es erfordern.[4, p. 1417]

4.2.2 DESCRIBE

Ein *DESCRIBE*-Befehl zeigt das Schema einer Relation an. Wurde kein Schema assoziiert, werden die Datentypen mit Hilfe der Operationen und Typregeln inferiert. Aufschlußreich ist z. B. die Regel: nur Zahlen können addiert werden. Inferierung stellt sicher, daß jederzeit ein (unvollständiges) Schema existiert, gleichwegs, ob es explizit assoziiert worden ist. Pig nimmt an, daß alle durch LOAD geladene Felder vom Typ ”bytearray” sind, falls kein Typ im Schema genannt oder kein Schema assoziiert oder inferiert wurde.[4, p. 1416]

4.2.3 STORE

Analog zum LOAD-Befehl speichert *STORE* eine Relation A in eine Datei, die zu speichernden Daten werden durch einen Serialisierer verarbeitet (siehe Bsp. 4.3). Ohne USING wird der Serialisierer PigStorage verwendet. STORE kann jederzeit verwendet werden, um Sicherungspunkte (Checkpoints) zu erstellen – Pig generiert keine automatisch. Dies ist kein Nachteil, da im Gegensatz zu SQL keine temporären Tabellen notwendig sind und die Berechnung weiterlaufen kann. Ohnehin kann nur der Programmierer an den geeignetesten Abschnitten Sicherungspunkte erstellen, sodaß abgebrochene Programme manuell weiter ohne Neuberechnung der angefallenen Daten fortgeführt werden können.[2]

Bsp. 4.3: STORE

```
STORE A INTO 'file' USING myStore();
```

Dieser Befehl verursacht keine neuen Map- oder Reduce-Schritte.[2]

Hinweis: die Operationen LOAD, DESCRIBE, DUMP und STORE werden nicht auf MapReduce abgebildet.

4.2.4 FOREACH

FOREACH dient der tupelweisen Projektion: eine Menge von Ausdrücken wird auf jedes Tupel voneinander unabhängig angewendet. Sei das Schema ”queries (userID, queryString, timestamp)” in gegeben. Weiters habe die Relation ”queries” die Tupel aus Bsp. 4.4

Bsp. 4.4: Tupel von queries

```
1 (alice,A,1),(bob,B,3)
```

und das durch Bsp. 4.5 spezifizierte Programm mit expandQuery als eine UDF. Wir erkennen in der Ausgabe Bsp. 4.6, daß expandQuery eine Multimenge erzeugt. Bei Relation b wird zusätzlich eine Operation *FLATTEN* angewendet (es ist keine UDF), die aus ihrer Eingabe die tiefste Verschachtelung entfernt (jedes Tupel der Menge) und mit den dabei entstehenden Tupel und mit den äußeren Tupelelementen (hier ”alice”) der in der GENERATE-Klausel aufgeführten Tupel (hier ”userID”) ein Kreuzprodukt bildet. Ein leerer Bag erzeugt keine Tupel.[3]

Auch hier werden – soweit möglich – Namen und Typen inferiert. Auf Werte von Ausdrücken ohne expliziten Namen (alternativ gibt es wie in SQL eine AS-Klausel) greift man über die $i-Notation zu, die in Kapitel 3. vorgestellt worden ist.

Bsp. 4.5: FOREACH

```
1 a=FOREACH queries GENERATE userID,expandQuery(queryString);
2 b=FOREACH queries GENERATE ↙
    ↳ userID,FLATTEN(expandQuery(queryString));
3 DUMP a;
4 DUMP b;
```

Bsp. 4.6: Programmausgabe

```
1 (alice,{(A x),(A y)}),(bob,{(B u),(B v)})
2 (alice,A x),(alice,A y),(bob,B u),(bob,B v)
```

FOREACH kann entweder in der Map- oder in der Reduce-Phase stattfinden, vorzugsweise in der aktuellen Phase, in der sich die Pipeline befindet.

4.2.5 FILTER

FILTER sortiert unter einer gegebenen Bedingung, die durch ”BY” eingeleitet wird, Tupel einer Relation aus. Es entspricht der Selektion in SQL. Als Bedingung eignen sich die Operatoren neq, eq oder ”matches REGEXP” für reguläre Ausdrücke bei Zeichenketten; ==, !=, > usf. finden Anwendung für Zahlen (siehe Bsp. 4.7).

Bsp. 4.7: FILTER

```
1 a=FILTER queries BY userID neq 'bot';
2 b=FILTER queries BY userID not isBot(userID);
```

4.2.6 (CO)GROUP

Hier sind zwei Operatoren vorzustellen. *COGROUP* gruppiert Tupel von mehreren Datenquellen und *GROUP* von einer Datenquelle. Für jede Gruppe wird ein Tupel ausgegeben, dessen erstes Feld ”group” heißt und jedes der folgenden Felder eine Multimenge ist. Offensichtlich sind bei Pig im Gegensatz zu SQL Gruppierung und Aggregation getrennt.

GROUP auf einer Relation haben wir bereits in einem Beispiel gesehen. Sei also folgendes Programm Bsp. 4.8 mit COGROUP gegeben.

Bsp. 4.8: COGROUP und FOREACH

```
1 a = LOAD 'file1' AS (name, age, gpa);
2 b = LOAD 'file2' AS (name, age, registration, donation);
3 c = COGROUP a BY name, b BY name;
4 DESCRIBE c
5 DUMP c
6 d = FOREACH c GENERATE FLATTEN((IsEmpty(a) ? null : a)), ↙
    ↳ FLATTEN((IsEmpty(b) ? null : b));
7 DUMP d
```

```
e = FOREACH c GENERATE FLATTEN(a), FLATTEN(b);
DUMP e
```

Die Ausgabe ist in Bsp. 4.9, an dem die besondere Wirkung von Null bei FLATTEN deutlich wird.

Bsp. 4.9: Programmausgabe

```
c: {group,a: {(name,age,gpa)},b: {(name,age,registration,donation)}} --DESCRIBE c

(a,{(a,7,7.5)},      {(a,5,k,4.32)}) --DUMP c
(b,{(b,8,34.23)},    {(b,6,k,4.5)})
(c,{(c,9,35446.3)}, {(c,8,r,4.3)})
(q,{(q,6,666.6)},    {})
(x,{},               {(x,99,r,3.33)})

(a,7,7.5,a,5,k,4.32) --DUMP d
(b,8,34.23,b,6,k,4.5)
(c,9,35446.3,c,8,r,4.3)
(q,6,666.6,)
(,,,x,99,r,3.33)

(a,7,7.5,a,5,k,4.32)--DUMP e
(b,8,34.23,b,6,k,4.5)
(c,9,35446.3,c,8,r,4.3)
```

(CO)GROUP ist eine typische Reduce-Operation. Shuffle und Reduce impliziert (CO)GROUP. Aus diesem Grund erzwingt der Operator eine Reduce-Phase: wenn die Pipeline in einer Map-Phase ist, folgt Shuffle an Reduce; andernfalls ist die Pipeline in einer Map-Phase und erzwingt einen vollständigen Map-Shuffle-Reduce-Zyklus.[3]

4.2.7 ORDER BY

Syntax und Semantik von *ORDER BY* sind äquivalent zu denen in SQL.

Bsp. 4.10: ORDER BY

```
b=ORDER a BY date DESC
```

Diese Operation erzwingt eine Reduce-Phase und fügt zusätzlich einen kompletten MapReduce-Zyklus zwecks Optimierung der Shuffle-Phase hinzu, denn einige Reducer bekommen weniger Daten, andere viel zu viel. Zuerst wird die Schlüsselverteilung geschätzt, in dem die Eingabe abgetastet wird. Danach wird ein Partionierer erstellt, der eine balancierte, totale Ordnung produziert. Dieser Partitionierer kann denselben Schlüssel also an verschiedene Reduce-Instanzen binden, um eine Gleichverteilung zu erzielen. Offensichtlich ist dies eine ”notwendige” Abweichung von der ”reinen Lehre’. Falls die Datenanalyse die Einhaltung dieser Konvention erfordert, wird abgeraten, ORDER BY zu verwenden.[3]

4.3 Beispiel: Wordcount

Wir geben im Folgenden ein Beispiel, um bereits gegebenes Wissen zu tradieren. Die Interpretation des vollständigen Programmes Bsp. 4.11 überlasse ich dem geneigten Leser zuallererst selbst und zeige anschließend Schritt für Schritt, wie die Eingabe transformiert wird. Auf die rechte Seite der nummerierten Schritte wende ich DESCRIBE (Bildunterschrift) und DUMP (Kodebox) an.

Bsp. 4.11: Komplettes Programm ”Wordcount”

```
1 input_lines = LOAD './Ipsum.txt' AS (line:chararray);
2 words = FOREACH input_lines GENERATE FLATTEN(TOKENIZE(line)) AS ↙
    ↳ word;
3 filtered_words = FILTER words BY word MATCHES '\\w+';
4 word_groups = GROUP filtered_words BY word;
5 word_count = FOREACH word_groups GENERATE COUNT(filtered_words) ↙
    ↳ AS count, group AS word;
6 ordered_word_count = ORDER word_count BY count DESC;
7 STORE ordered_word_count INTO '/tmp/result';
```

Dies ist die Eingabe ./Ipsum.txt des Programmes.

Bsp. 4.12: Eingabe für Wordcount

```
Used since the 1500s. Written by Cicero.

1914 translation by H. Rackham
```

Die Eingabe wurde eingelesen über den Serialisierer PigStorage. Jede Zeile der Eingabe ist ein Tupel.

Bsp. 4.13: 1) input_lines: {line: chararray}

```
(Used since the 1500s. Written by Cicero.)
()
(1914 translation by H. Rackham)
```

In diesem Abschnitt wurden die Sätze tokenisiert...

Bsp. 4.14: 2) words: {word: chararray}

```
(Used), (since), (the), (1500s.), (Written), (by), (Cicero.),
(), (1914), (translation), (by), (H.), (Rackham)
```

...und hier Weißzeichen aussortiert (Der reguläre Ausdrück passt aus mir nicht bekannten Gründen auch auf das Wort ”1500s.”).

Bsp. 4.15: 3) filtered_words: {word: chararray}

```
(Used), (since), (the), (Written),
(by), (1914), (translation), (by), (Rackham)
```

Das ist die Ausgabe, wenn nach den Worten gruppiert worden ist.

Bsp. 4.16: 4) word_groups: {group: chararray,filtered_words: {(word: chararray)}}

```
(by,{(by),(by)}), (the,{(the)}), (1914,{(1914)}), (Used,{(Used)}),
(since,{(since)}), (Rackham,{(Rackham)}), (Written,{(Written)}), ↙
    ↳ (translation,{(translation)})
```

Auf die Gruppe wurde die Aggregation angewendet.

Bsp. 4.17: 5) word_count: {count: long,word: chararray}

```
(2,by)
(1,the), (1,1914), (1,Used), (1,since),
(1,Rackham), (1,Written), (1,translation)
```

Die absteigend sortierte Relation entspricht dem Erwünschten.

Bsp. 4.18: 6) ordered_word_count: {count: long,word: chararray}

```
(2,by)
(1,the), (1,1914), (1,Used), (1,since),
(1,Rackham), (1,Written), (1,translation)
```

4.4 Weitere Operatoren

Der letzte der hier ausführlich vorgestellten Operatoren ist der JOIN. Später werde ich noch kurz weitere Operatoren benennen.

4.4.1 JOIN

Auch der *JOIN* verhält sich wie in SQL, jedoch werden nur Equi-Join unterstützt.[3] Ohne weitere Angabe ist jeder JOIN ein Inner-Join, aber auch LEFT-, RIGHT- und FULL-OUTER-JOIN sind möglich, wie folgendes Bsp. 4.19 zeigt.

Bsp. 4.19: JOIN

```
a=JOIN daily BY (symbol, date) LEFT OUTER, divs BY (symbol, date);
```

Auch der JOIN erzwingt wie (CO)GROUP eine Reduce-Phase. Eine Anmerkung zu dem Problem mehrerer Eingaben in MapReduce: gemeinhin wird dies gelöst, indem in der Map-Phase der Ursprung des Tupels hinzugefügt wird (Datenzunahme). Der JOIN-Schlüssel ist der Schlüssel, auf dem Shuffle arbeitet. Und zuletzt wird beim Reduce ein Kreuzprodukt von beiden Eingaben gebildet wie folgt: die linke Quelle wird im Hauptspeicher gehalten, die rechte folgt partioniert. Von jedem Tupel rechts

wird schrittweise das Kreuzprodukt mit den Tupeln links gebildet, die der JOIN-Bedingung genügen.[3]

Zum Abschließen dieses Kapitels folgt ein komplexes Beispiel. Gegeben seien die Daten Bsp. 4.20, Bsp. 4.21 und das Programm Bsp. 4.22: zwei Relationen, die eine revenue mit Schema "(queryString, amount)", die andere results mit Schema "(queryString,url)".

Bsp. 4.20: Tupel von revenue

```
(lakers,50), (lakers,20), (kings,30), (kings,10)
```

Bsp. 4.21: Tupel von results

```
(lakers,nba.com), (lakers,espn.com), (kings,nhl.com), ↙
    ↳ (kings,nba.com)
```

Das Programm Bsp. 4.22 zeigt die Äquivalenz von COGROUP mit FLATTEN und einem JOIN. Dies gilt, bis es Null-Werte in den Schlüsseln gibt.[3] Hiermit klärt sich auch, wie JOIN auf MapReduce abgebildet werden kann.

Bsp. 4.22: Äquivalenz von JOIN und COGROUP mit FLATTEN

```
1 a = JOIN results BY queryString, revenue BY queryString;
2 tmp = COGROUP results BY queryString, revenue BY queryString;
3 b = FOREACH tmp GENERATE FLATTEN(results), FLATTEN(revenue);
```

Es gilt auf Grund der Äquivalenz a=b. Der Befehl "DUMP a" gibt Bsp. 4.23 aus.

Bsp. 4.23: {results::queryString,results::url,revenue::queryString,revenue::amount}

```
1 (kings,nhl.com,kings,30)
2 (kings,nhl.com,kings,10)
3 (kings,nba.com,kings,30)
4 (kings,nba.com,kings,10)
5 (lakers,espn.com,lakers,50)
6 (lakers,espn.com,lakers,20)
7 (lakers,nba.com,lakers,50)
8 (lakers,nba.com,lakers,20)
```

Der Zwischenschritt DUMP tmp wird in Bsp. 4.24 gezeigt.

Bsp. 4.24: {group, results:{(queryString,url)},revenue: {(queryString,amount)}}

```
(kings,{(kings,nhl.com),(kings,nba.com)},{(kings,30),(kings,10)})
(lakers, {(lakers,nba.com),(lakers,espn.com)}, ↙
    ↳ {(lakers,50),(lakers,20)})
```

4.4.2 UNION

UNION vereint zwei oder mehrere Multimengen.

4.4.3 CROSS

CROSS berechnet das Kreuzprodukt von zwei oder mehreren Multimengen.

4.4.4 DISTINCT

DISTINCT eliminiert Tupelduplikate und ist äquivalent zu: gruppiere die Multimenge nach allen Feldern, filtere die Gruppen aus. Es erzwingt eine Reduce-Phase.

4.4.5 SPLIT

SPLIT trennt Tupel nach einem Prädikat in zwei Bags. Die ”Pipeline” wird dadurch aufgeteilt. Obwohl wir bisher oft von Pipeline gesprochen haben, wie es auch bei SQL der Fall ist, gilt dies im Allgemeinen – wie angekündigt – für Pig nicht. Offensichtlich wird bei dem Operator SPLIT, daß wir es mit einem *DAG* (gerichteten, azyklischen Graph) zu tun haben.

4.5 Bemerkung

Nur wenige Operatoren wurden ausgelassen. Anschließend an die Anmerkung zu der Parallelisierung in Kapitel 2. läßt sich vervollständigen, daß der Parallelisierunggrad der Reduce-Phase steuerbar durch Schlüsselwörter bei derzeit GROUP, COGROUP, ORDER, DISTINCT, JOIN, CROSS und LIMIT ist. Jetzt am Ende des Kapitels möge sich der geneigte Leser mit den Datentransformationen in Bsp. 4.1 genauer befassen.

5 Schluß

Mit dieser Arbeit wird offenbar, daß Pig Latin das MapReduce-Paradigma bereichert: viele Operatoren sind aus SQL bekannt und obwohl GROUP, ORDER BY, FOREACH und FILTER natürlich im MapReduce-Paradigma auszudrücken sind, wird die Implementierung der JOIN in MapReduce durch die Limitierung des Paradigmas erschwert[4, p. 1414][3] und daher ist der Kode einer Hochsprache, die auf MapReduce aufsetzt, durchsichtiger als z.B. native MapReduce-Programme. Letztlich gilt gleichwohl: die Nutzung einer Hochsprache kompaktifiziert Quellkode von Anwendungen und versteckt dabei deren Komplexität mit der Folge, daß direkter Einfluß auf niedere Ebenen ("low-level") erschwert wird. Erkauft wird sich u.a. eine bessere Wart- und Wiederverwertbarkeit.

Weiterhin sollen Pig-Latin-Programme neben Hadoop auch auf andere Frameworks aufsetzen können, was hilfreich ist, falls sich (die lose Kopplung mit) Hadoop als zu unflexibel erweist.

Trotzdem Pig eine Hochsprache ist, wird dem vernünftigen Programmierer, der verantwortungsbewusst in niedere Ebenen (low-level) eingreift, die Möglichkeiten gegegeben, explizit geeignete Algorithmen verwenden, in die Parallelisierung eingreifen und selbstständig Sicherungspunkte setzen zu können.

Zwei Konkurrenten von Pig sind beispielsweise Jaql und Stratosphere. Jaql wird – trotzdem es quelloffen verfügbar ist – derzeit nicht mehr weiterentwickelt. Stratosphere ist ein laufendes Forschungsprojekt; ob es mit der Hochsprache Meteor, die sich in Entwicklung befindet, an Pig Latin anschliessen kann, wird sich zeigen.[5]

Anders als viele Konkurrenten hat sich – wie die Nutzung seitens großer IT-Firmen und die stete Weiterentwicklung zeigen – Pig bei der Analyse von big data mittels des MapReduce-Paradigmas bereits bewährt.

Kode

Abbildungen

Literaturverzeichnis

[1] Dean, J. and S. Ghemawat: *Mapreduce: Simplified data processing on large clusters.* OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

[2] Gates, A.: *Comparing pig latin and sql for constructing data processing pipelines.* none, 2010. http://developer.yahoo.com/blogs/hadoop/posts/2010/01/comparing_pig_latin_and_sql_fo/, visited on 2010-01-29.

[3] Gates, A.: *Programming Pig.* O'Reilly Media, 1st ed., 2011. http://ofps.oreilly.com/titles/9781449302641/index.html.

[4] Gates, A. *et al.*: *Building a high-level dataflow system on top of map-reduce: the pig experience.* Proc. of the VLDB Endowment 2(2), 2009.

[5] Heise, A., A. Rheinländer, *et al.*: *Meteor/sopremo: An extensible query language and operator model.* Int. Workshop on End-to-end Management of Big Data 2012 Istanbul,Turkey, 2012.

[6] Olston, C., B. Reed, *et al.*: *Pig latin: a not-so-foreign language for data processing.* SIGMOD Conference, Vancouver, CD, 2008.

[7] White, T.: *Hadoop: The Definitive Guide.* O'Reilly Media, 2012.