Vorlesung

Cloud- und Web-Technologien

(CS3140)

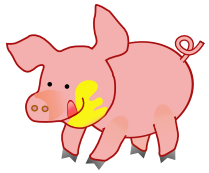
Datenverarbeitung mit Pig

Professor Dr. rer. nat. habil. Sven Groppe
https://www.ifis.uni-luebeck.de/index.php?id=groppe

Chronologische Übersicht über die Themen

Nr	Thema	
1	Einleitung	
2	Einführung in das Semantic Web, RDF und SPARQL	 Datenmodell
3	Die Semantic Web-Ontologiesprachen RDFS und OWL	
4	Multiplattform-Entwicklung mit Kotlin	 Multiplattform
5	Fortgeschrittene Themen mit Kotlin	
6	Einstieg in Cloud Computing, Hadoop	 Backend
7	Operatoren der relationalen Algebra in Hadoop	
8	**Datenverarbeitung mit Pig**	
9	Einführung in Spark und Flink	
10	Stromverarbeitung mit Flink	
11	Knotenzentrische Algorithmen mit Flink	
12	HTML und CSS	 Web
13	Browserprogrammierung mit JS/JQuery und Serverprogrammierung mit PHP Hypertext Preprocessor	
14	Zusammenfassung und Ausblick	

Pig

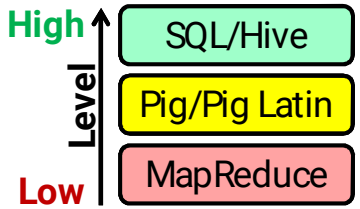


- Apache Open Source Projekt
  - ursprünglich von Yahoo entwickelt
- Engine
  - zum Ausführen von Programmen mit Hadoop
  - Seit R0.17 (2017) auch Spark-Unterstützung
- High-Level Sprache Pig Latin
  - zum Spezifizieren der Programme

Motivation von Pig 1/5

- Yahoo: >40% der Hadoop Jobs sind Pig Jobs*
- Unternehmen, die Pig einsetzen*:
  Twitter, LinkedIn, Nokia, PayPal, Salesforce.com, …
- Teil von Amazon EMR Web Service und
  Cloudera Hadoop Distribution
- Anwendungsgebiete von Pig:
  - Infrastruktur zur Bereitstellung von Suche-Funktionalität
  - Analyse der Relevanz von Werbung
  - Analyse von Benutzerabsichten (User Intent Analysis)
  - Verarbeitung von Weblogs
  - Inkrementelle Verarbeitung von großen Datensätzen

Motivation von Pig 2/5



- (Imperative) Higher-Level Sprache Pig Latin
- Flexibler als noch höhere Anfragesprache wie etwa SQL (bzw. Hive-Anfragesprache)
- Support von Standard-Datenoperationen (z.B. Projektion, Selektion, Join)
  - *„Anfragen stellen durch Angabe eines Anfrageplanes zur Ausführung“*
- Komplexe Berechnungen einfacher und fehlerunanfälliger ausdrückbar als in MapReduce
  - ➡ Erhöhung der Produktivität von Entwicklern
- Erniedrigt Aufwand durch Mehrfachentwicklungen

Motivation von Pig 3/5

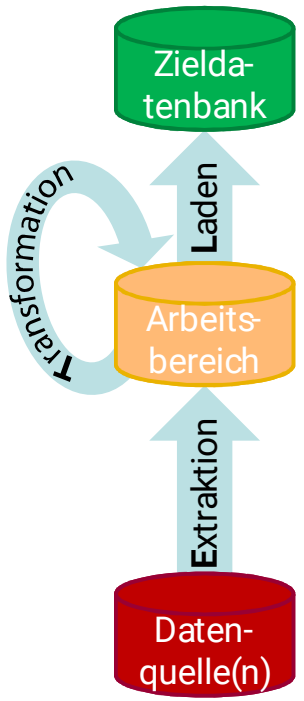
- Entwickelt für Ad-Hoc Datenanalyse
  - Anfragen können direkt über Datendateien gestellt werden
  - Falls Datenformat keinem (unterstütztem) Format folgt, kann der Benutzer eine Funktion zum Parsen des Dateiinhaltes zur Verfügung stellen
    - Einbindung von beliebigen Datenquellen (u.a. HBase) so möglich
    - Analog bei Ausgabe
  - Angabe von **Schemata** ist **optional**

Motivation von Pig 4/5

- Kapselung der Hadoop-Komplexität
  - Hadoop Version Upgrades mit API-Modifikationen
    $\leftrightarrow$ Pig Latin abwärtskompatibel
  - Tuning der Job-Konfiguration
  - Verkettung von Jobs und deren Datenflüsse

Motivation von Pig 5/5: Unterstützung des ETL-Prozesses durch Pig

- **ETL**-Prozess
  - **E**xtraktion der relevanten Daten aus verschiedenen Quellen
  - **T**ransformation der Daten in das Schema und Format der Zieldatenbank
  - **L**aden der Daten in die Zieldatenbank
- Anwendung des ETL-Prozesses z.B. beim Betrieb von Data-Warehouses
  - Konsolidierung von großen Datenmengen aus mehreren operationalen Datenbanken mit anschließender Speicherung im Data-Warehouse



Nachteile von Pig

- Benutzer
  - Erlernen einer neuen Programmiersprache notwendig
- Latenz
  - Ziel ist 10-20% Overhead im Vergleich zu Hadoop, öfter 50% Overhead

Beispiel: WordCount

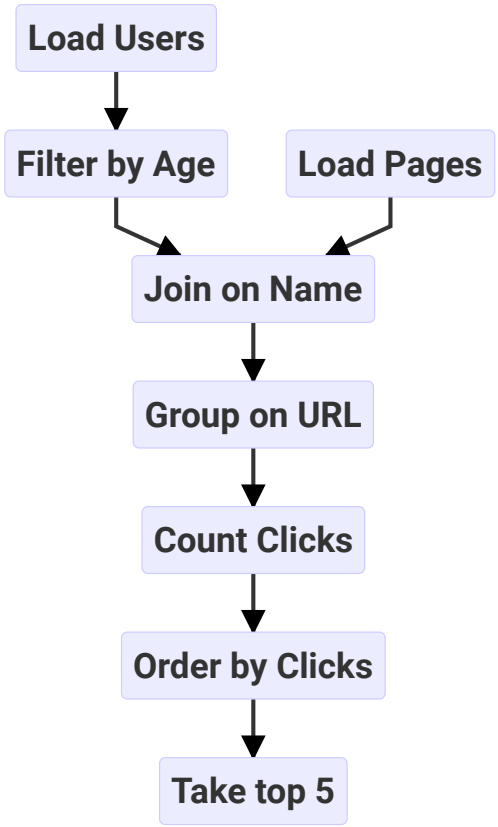
- MapReduce-Programm aus vorletzter Vorlesungseinheit: 37 Zeilen Code (ohne import)
- Als Pig Latin Skript in 4 Zeilen:

```
A = LOAD 'wordcount/input' as (token:chararray);

B = GROUP A BY token;

C = FOREACH B GENERATE group, COUNT(A) as numberOfWords;

DUMP C;
```

Beispiel: Analyse von Webseitenbesuche*

- Annahmen:
  - Datei mit Daten über Benutzer
  - Daten über Webseitenbesuche
- Gesucht:
  - Die 5 meistbesuchten Seiten von Benutzern zwischen 18 und 25 Jahren

Ablauf:



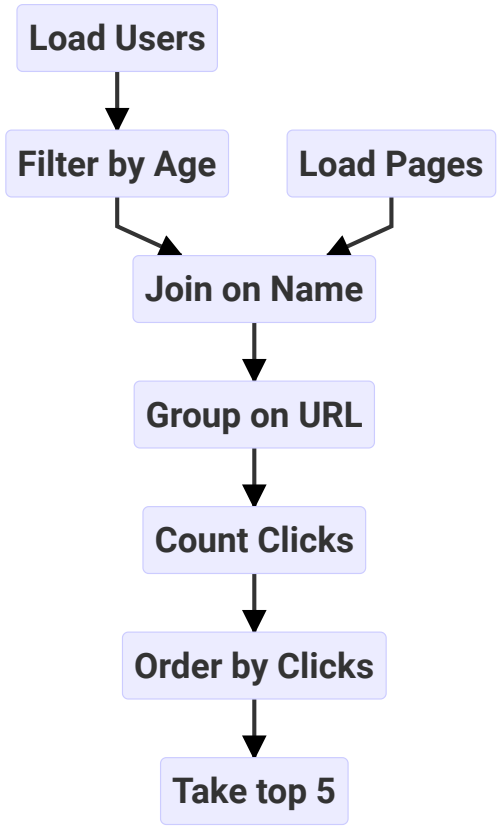
* Beispiel von Alan Gates, Yahoo!Research

Beispiel: Analyse von Webseitenbesuche*

Formulieren als Pig Latin-Skript:

```
Users    = load 'users' as (name, age);
Filtered = filter Users by
                  age >= 18 and age <= 25;
Pages    = load 'pages' as (user, url);


Joined   = join Filtered by name, Pages by user;


Grouped  = group Joined by url;


Summed   = foreach Grouped generate group,
                  count(Joined) as clicks;

Sorted   = order Summed by clicks desc;


Top5     = limit Sorted 5;
store Top5 into 'top5sites';
```

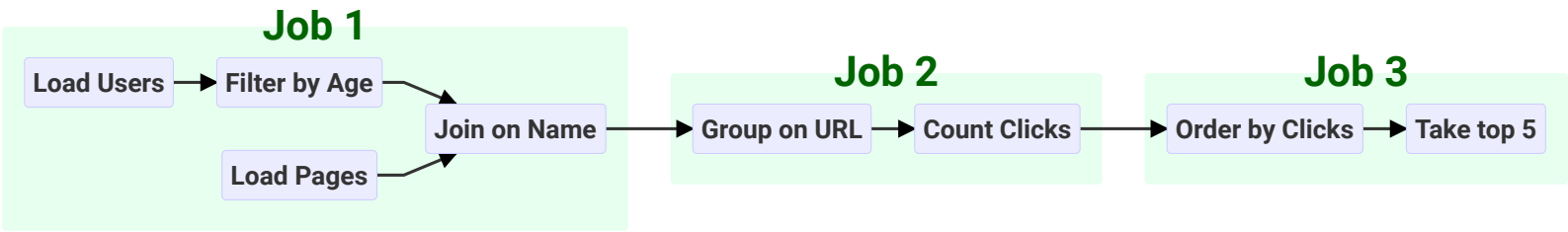
Ablauf:



* Beispiel von Alan Gates, Yahoo!Research

Beispiel: Analyse von Webseitenbesuche*

Pig kreiert automatisch MapReduce-Jobs:

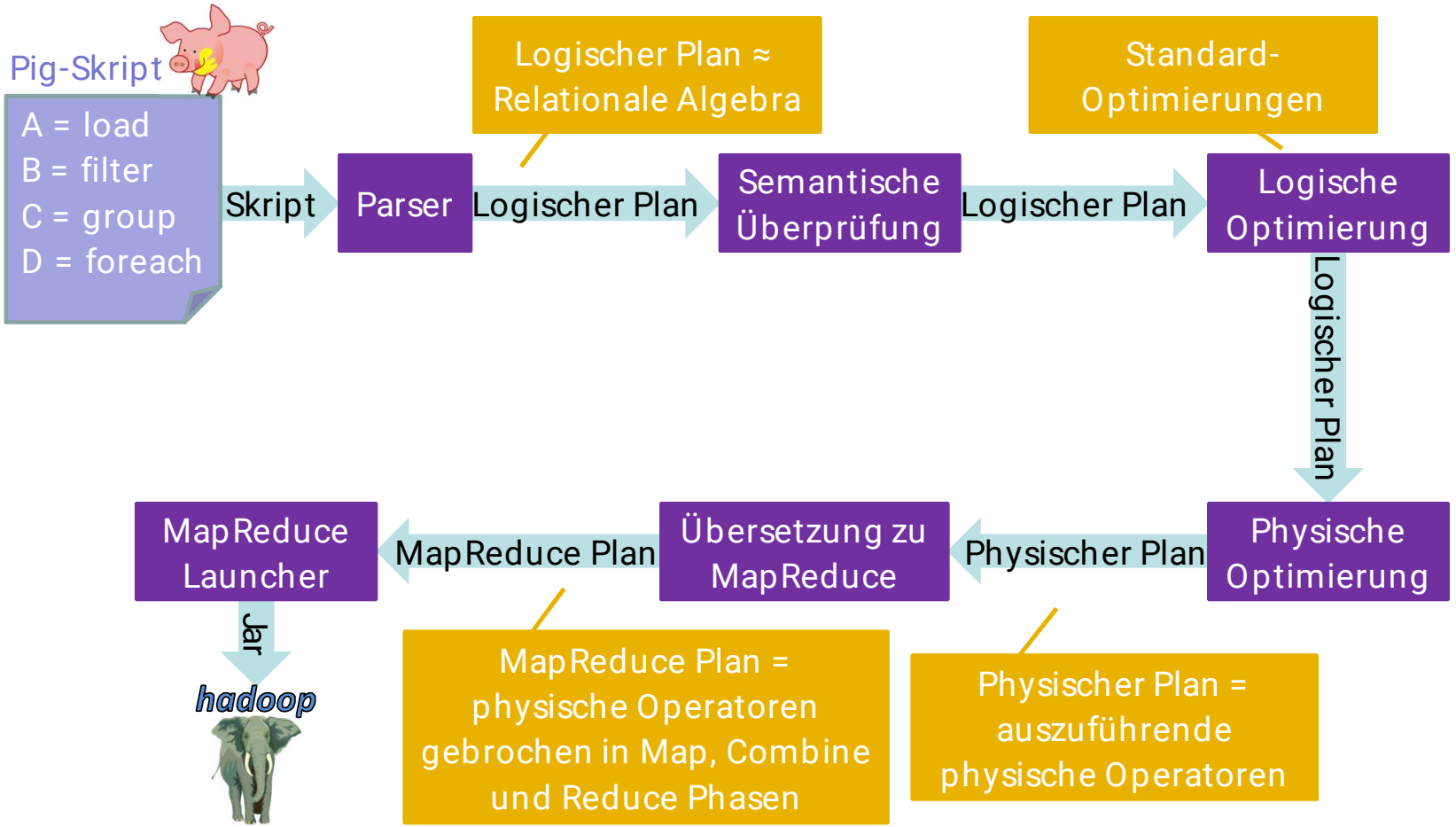


* Beispiel von Alan Gates, Yahoo!Research

Flexibilität von Pig

- Mehrere Ausgaben durch ein Pig Latin-Skript möglich
  - Vermeiden doppelter Berechnungen
- Support von User Defined Functions (UDF)
  - Entwicklung in Java, Python, JavaScript, Ruby oder Groovy
  - UDFs zum Laden/Speichern von Daten, Evaluieren von Tupeln (z.B. Transformation eines Tupels in FOREACH), Aggregieren und Filtern
- Ausführung von ganzen benutzerdefinierten (und in einer anderen Programmiersprache entwickelten) MapReduce-Jobs durch Pig Latin-Befehl

Übersetzung zu MapReduce



Datenmodell von Pig Latin

- Basistypen

Typ	Beschreibung	Notation für Konstante
Atom	Einfacher Wert - für bytearray gibt es keine Notation für Konstanten - Boolean-Konstanten sind Case Insensitive	long: `42` `42L` double: `42.0` `4.2e1` float: `42.0F` `4.2e1f` chararray: `'XLII'` boolean: `true` `false`
Tuple	Sequenz von Feldern beliebigen Typs	`(42, 'XLII', 'zweiundvierzig')`
Bag	Kollektion von Tupeln	`{ (41, 'XLI'), (42, 'XLII') }`
Map	Datenstruktur zur Verwaltung von Schlüssel-Wert-Paaren	`[ 41 # 'XLI', 42 # 'XLII' ]`

	Schlüssel	Wert
Datentyp	Atom	beliebig

- Beliebige Schachtelung von Datentypen möglich

Zugriff auf einzelne Komponenten eines Tupels

- Tupel `t= ('Alice',{('lakers',1),('iPod',2)},['age'# 20])`
  - `f1` `f2` `f3`
  - Felder von `t` seien benannt mit `f1`, `f2` und `f3`

Typ des Ausdrucks	Beispiel	Wert
Angabe der Feldposition	`$0`	`'Alice'`
Angabe des Feldnamens	`f3`	`[‘age‘ # 20]`
Projektion	`f2.$0`	`{(‘lakers‘), (‘iPod‘)}`
Map Lookup	`f3#'age'`	`20`
Funktionsevaluation	`SUM(f2.$1)`	`1 + 2 = 3`
Bedingter Ausdruck	`f3#'age'>18? 'adult':'minor'`	`'adult'`
Entschachteln	`FLATTEN(f2)`	`('lakers',1), ('iPod',2)` (2 Tupel anstatt 1 Bag)
Wildcard (für alle Felder)	`*`	`(‘Alice‘,{(‘lakers‘,1), (‘iPod‘,2)}, [‘age‘#20])`

Beachtung der Groß- und Kleinschreibung

- Beachtung bei
  - Namen/Aliase von
    - Relationen
    - Feldern
    - Funktionen
- Keine Beachtung bei
  - Schlüsselwörtern
    - Beispiele:
      `LOAD`, `USING`, `AS`, `GROUP`, `BY`, `FOREACH`, `GENERATE`
      gleichbedeutend mit
      `load`, `using`, `as`, `group`, `by`, `foreach`, `generate`

Notation von Schemata

Schema	Notation	Beispiele
Tupel	[alias:][tuple] (alias[:*type*] [, alias[:*type*] ...])	T:tuple(f1:int,f2:int,f3:int) ⇔ T:(f1:int,f2:int,f3:int)
Bag	[alias:][bag] {*tuple*}	B:bag{T:tuple(t1:int,t2:int,t3:int)} ⇔ B:{T:(t1:int,t2:int,t3:int)}
Map	[alias:][map] '[' [*type*] ']'	M:map[] ⇔ M:[] map[int] Wert der Map ist int, Mapfeld ist unbenannt map[(i:int)] Map-Wert ist ein (unbenanntes) Tupel mit einem Feld i
verschachtelt		T1: tuple(f1:int, M: map[]) B: bag{T2: tuple(t1:float, t2:float)}

[...]: optional alias: Bezeichner *type*: Datenschema *tuple*: Tupelschema

Daten laden

- `LOAD 'data' [USING function] [AS schema];`
  - `'data'`: HDFS Dateiname oder Verzeichnis (alle Dateien des Verzeichnisses werden geladen)
  - Optionale (vorher zu registrierende benutzerdefinierte oder Built-In (z.B. JSON, HBase)) Ladefunktion zum Parsen der Datei(en)
    - Fehlen der Funktion äquivalent zu `USING PigStorage('\t')`
      - Jede Zeile in Datei ergibt ein Tupel
      - Jedes Tab in einer Zeile trennt ein Feld im Tupel
  - Angabe eines Schemas optional
    - Bei Fehlen sind die Felder unbenannt und vom Typ Bytearray
- Beispiel:

myfile.txt

1	2	3
4	2	1
8	3	4

→

```
A = LOAD 'myfile.txt'
    AS (f1:int, f2:int, f3:int);
```

→

A

f1:int	f2:int	f3:int
1	2	3
4	2	1
8	3	4

Daten speichern

- `STORE alias INTO 'directory' [USING function];`
  - `alias`: Name der zu speichernden Relation
  - `'directory'`: Verzeichnis, in dem die Dateien (mit Namen *part-nnnnn*) geschrieben werden
    - Verzeichnis darf vorher nicht existieren!
  - Optionale (benutzerdefinierte oder Built-In (z.B. JSON, HBase)) Speicherfunktion
    - Default ist `USING PigStorage('\t')`
- Beispiel:

input.txt

1	2	3
4	2	1
8	3	4

→

```
A = LOAD 'input.txt'
    AS (f1:int, f2:int, f3:int);
STORE A INTO 'output'
    USING PigStorage('*');
```

→ output.txt

```
1*2*3
4*2*1
8*3*4
```

Selektion

- `alias = FILTER alias BY` *`expression`*`;`
- Funktionen über Wahrheitswerte
  - Vergleichsoperatoren: `==`, `!=`, `<`, `<=`, `>`, `>=`
  - Mustererkennung: *`expression`* `matches` *`string`*
    - *`expression`* muss dem Java-Format für reguläre Ausdrücke entsprechen
  - Verknüpfungen: `AND`, `OR`, `NOT`
- Beispiel:

a.txt

```
8 3 4
1 2 3
4 2 1
4 2 1
2 2 3
```

→

```
A = LOAD 'a.txt' AS (a1:int,a2:int,a3:int);
X = FILTER A BY (a3 == 3) OR (a3 < 2);
```

→ X

a1:int	a2:int	a3:int
1	2	3
4	2	1
4	2	1
2	2	3

Vereinigung

- `alias = UNION [ONSCHEMA] alias, alias [, alias …];`
  - `ONSCHEMA`: Vereinigung basierend auf Feldnamen anstatt bzgl. der Feldpositionen
  - Umfangreiche Regeln zur Bestimmung des Schemas der Ausgabe
    - Schemata mit unterschiedlicher Feldanzahl → Kein Ausgabeschema
    - Inkompatible Feldschemata → `bytearray`
    - Kompatible Feldschemata
      - `double` > `float` > `long` > `int` > `bytearray`
      - `tuple` | `bag` | `map` | `chararray` > `bytearray`
    - …
- Beispiel:

a.txt

1	2
4	2

b.txt

6	7
8	9

→

```
A = LOAD 'a.txt' AS (a1:int, a2:int);
B = LOAD 'b.txt' AS (a2:int, a1:int);
X = UNION A, B;
Y = UNION ONSCHEMA A, B;
```

→

X

:int	:int
1	2
4	2
6	7
8	9

Y

a1:int	a2:int
1	2
4	2
7	6
9	8

Duplikateliminierung

- `alias = DISTINCT alias [PARTITION BY partitioner] [PARALLEL n];`
  - `PARTITION BY partitioner`: Angabe eines benutzerdefinierten Hadoop-Partitioner, der die Partitionierung der Schlüssel der Map-Ausgabe kontrolliert
    (z.B. durch Anwendung einer Hash-Funktion)
  - `PARALLEL n`: Angabe des Parallelisierungsgrad
    (= Anzahl von Reduce-Jobs)
- Beispiel:

a.txt

```
8 3 4
1 2 3
4 2 1
4 2 1
1 2 3
```

→

```
A = LOAD 'a.txt' AS (a1:int,a2:int,a3:int);
X = DISTINCT A;
```

→ X

a1:int	a2:int	a3:int
8	3	4
1	2	3
4	2	1

Gruppieren

- alias = **GROUP** alias { **ALL** | **BY** *expression*}
  [, alias **ALL** | **BY** *expression* …] […]
  [**PARTITION BY** *partitioner*] [**PARALLEL** n];
  - **COGROUP** Synonym von **GROUP** (Verwendung üblicherweise beim Gruppieren über mehrere Relationen)
- Beispiele:

a.txt

table	5	1
pc	2	1
chair	2	1
chair	3	2
pc	1	6

b.txt

P1	table
P2	pc

→

```
A = LOAD 'a.txt' AS
    (product:chararray,
     weight:int,
     units:int);
B = GROUP A ALL;
C = GROUP A BY units*weight;
D = LOAD 'b.txt' AS
    (owner:chararray,
     owned:chararray);
E = COGROUP A BY product,
            D BY owned;
```

→

B

A:{(product:chararray,weight:int, units:int)}
{(table,5,1), (pc,2,1), (chair,2,1), (chair,3,2), (pc,1,6)}

C

group:int	A:{…}
5	{(table,5,1)}
2	{(pc,2,1), (chair,2,1)}
6	{(chair,3,2), (pc,1,6)}

E

group:chararray	A:{…}	D:{…}
table	{(table,5,1)}	{(P1,table)}
pc	{(pc,2,1), (pc,1,6)}	{(P2,pc)}
chair	{(chair,2,1), (chair,3,2)}	{}

Spaltenbasierte Verarbeitung

- `alias = FOREACH alias GENERATE` *expression* `[AS` *schema*`]` `[,` *expression* `[AS` *schema*`]` … `];`
- Beispiele:

a.txt

8 3 4
1 2 3
4 2 1
4 2 1
1 3 4

→

```
A = LOAD 'a.txt' AS
    (a1:int,a2:int,a3:int);
-- Projektion:
X = FOREACH A
    GENERATE a1, a2;
-- Add/Schema:
Y = FOREACH A
    GENERATE a1+a2 AS f1:int;
-- Aggregation:
B = GROUP A BY a1;
Z = FOREACH B
    GENERATE group,SUM (A.a2);
```

→

X

a1:int	a2:int
8	3
1	2
4	2
4	2
1	3

Y

f1:int
11
3
6
6
4

Z

group:int	:int
8	3
1	5
4	4

Verbund/Join

- `alias = JOIN alias BY {expression | (expression [, expression ...] )} (, alias BY {expression | (expression [, expression ...] )} ...) [USING 'replicated' | 'skewed' | 'merge' | 'merge-sparse'] [PARTITION BY partitioner] [PARALLEL n];`
- Beispiel:

a.txt

1	2	3
4	2	1
8	3	4
4	3	3
7	2	5
8	4	3

b.txt

2	4
8	9
1	3
2	7
2	9
4	6
4	9

→

```
A = LOAD 'a.txt' AS
    (a1:int, a2:int, a3:int);
B = LOAD 'b.txt' AS
    (b1:int, b2:int);
X = JOIN A BY a1,
         B BY b1;
```

→

X

a1:int	a2:int	a3:int	b1:int	b2:int
1	2	3	1	3
4	2	1	4	6
4	2	1	4	9
4	3	3	4	6
4	3	3	4	9
8	3	4	8	9
8	4	3	8	9

Sortieren

- `alias = ORDER alias BY { * [ASC|DESC] | field [ASC|DESC] [, field [ASC|DESC] ...] } [PARALLEL n];`
  - `ASC`/`DESC`: Auf-/Absteigend
- Beispiel:

myfile.txt

1	2	3
4	2	1
8	3	4

→

```
A = LOAD 'myfile.txt'
    AS (f1:int, f2:int, f3:int);
X = ORDER A BY f3 ASC;
```

→ X

f1:int	f2:int	f3:int
4	2	1
1	2	3
8	3	4

Weitere Operationen (unvollständige Liste)

- `REGISTER`: Registrieren von benutzerdefinierten Funktionen
- `MAPREDUCE`: Ausführen eines nativen MapReduce-Jobs
- `RANK`: Hinzufügen eines Ranges
  (Position innerhalb der (optional sortierten) Relation)
- `SAMPLE`: Stichprobe (spezifizierbarer Größe)
- `SPLIT`: Horizontales Aufteilen einer Relation
- `STREAM`: Daten von einem externen Programm verarbeiten
- `DEFINE`: Definieren von Makros
- `IMPORT`: Importieren von Makros
- Äußere Joins
- `CROSS`: Berechnen des Kreuzproduktes
- `LIMIT`: Ausgabetripel auf konstante Anzahl beschränken
- `DUMP`: Menschenlesbare Ausgabe einer Relation

Formulierung einer SPARQL-Anfrage als PigLatin-Skript


```
PREFIX ….
SELECT ?person ?name
WHERE {
  ?article rdf:type bench:Article .
  ?article dc:creator ?person .
  ?person foaf:name ?name .
}
ORDER BY ?name
```

Annahme:
Die Ergebnisse der Tripelmuster seien in den Dateien `tp1`, `tp2` und `tp3` abgelegt…

Zum Weiterlesen

- Übersicht über die Operationen
  - Link
- Hinweise zur Performance von Pig Latin-Skripten
  - Link
- Hauptpublikation
  - C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins: Pig latin: a not-so-foreign language for data processing. *In Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08)*, Vancouver, Canada, 2008.
    DOI

Zusammenfassung

- Einsatz in vielen Unternehmen
  insbesondere Yahoo
- Höhere Abstraktion als bei MapReduce
  - Operatoren entsprechen weitestgehend denen der relationalen Anfragesprache
  - ➡ Angabe einer Art Anfrageplan
- Niedrigere Abstraktion als bei Anfragesprachen, dafür wesentlich flexibler:
  - Einlesen beliebiger Eingaben und Ausgabe in beliebigen Formaten durch benutzerdefinierte Funktionen möglich
  - Ausführung von benutzerdefinierten MapReduce-Jobs