Big Data Processing, 2014/15

Lecture 8: Pig Latin

Claudia Hauff (Web Information Systems)
ti2736b-ewi@tudelft.nl

Start up CDH to follow along the Pig script examples. The example data used in the lecture is available on BB: **[TI2736-B]/Lectures/Lecture8/example_data_lecture8**

Course content

- Introduction
- Data streams 1 & 2
- The MapReduce paradigm
- Looking behind the scenes of MapReduce: HDFS & Scheduling
- Algorithm design for MapReduce
- **A high-level language for MapReduce: Pig Latin 1** & 2
- MapReduce is not a database, but HBase nearly is
- Lets iterate a bit: Graph algorithms & Giraph
- How does all of this work together? ZooKeeper/Yarn

Learning objective

- **Translate** basic problems (suitable for MapReduce) into Pig Latin based on built-in operators

Last time …

Relation
Hyperlinks

attributes

tuples

FROM	TO
url1	url2
url2	url3
url3	url5

- Database tables can be written out to file, one tuple per line
- MapReduce jobs can perform standard database operations
    - Most useful for operations that pass over most (all) tuples
- Joins are particularly tedious to implement in “plain” Hadoop

We don't just have joins ….

No Pig Latin yet

Selections

Web_pages

Url	Category	Last_crawl_ date	Page_length	Lng
news.yahoo.de	news	03-12-2013 07:08:45	765443	GER
nu.nl	news	03-12-2013 11:45:00	64435	NL
chess.com	game	23-10-2013 19:34:01	1264	EN
www.bbc.com/ sport/0/football/	sports	03-12-2013 14:13:22	6324	EN

Question: how can you do a selection in Hadoop?

Projections

Web_pages

Url	Category	Last_crawl_date	Page_length	Lng
news.yahoo.de	news	03-12-2013 07:08:45	765443	GER
nu.nl	news	03-12-2013 11:45:00	64435	NL
chess.com	game	23-10-2013 19:34:01	1264	EN
www.bbc.com/sport/0/football/	sports	03-12-2013 14:13:22	6324	EN

Question: how can you do a projection in Hadoop?

Union

Web_pages_crawler1

Url	Category	Page_length	Lng
news.yahoo.de	news	765443	GER
nu.nl	news	64435	
chess.com	game	1264	
www.bbc.com/ sport/0/football/	sports	6324	

Web_pages_crawler2

Url	Category	Page_length	Lng
news.yahoo.de	news	765443	GER
volkskrant.nl	news	234445	NL
chessbase.com	game	1264	EN
www.bbc.com/ sport/0/football/	sports	6324	EN

Question: how can you do a union in Hadoop?

Intersection

Web_pages_crawler1

Url	Category	Page_length	Lng
news.yahoo.de	news	765443	GER
nu.nl	news	64435	
chess.com	game	1264	
www.bbc.com/ sport/0/football/	sports	6324	

Web_pages_crawler2

Url	Category	Page_length	Lng
news.yahoo.de	news	765443	GER
volkskrant.nl	news	234445	NL
chessbase.com	game	1264	EN
www.bbc.com/ sport/0/football/	sports	6324	EN

Question: how can you do an intersection in Hadoop?

And now …. Pig Latin

Pig vs. Pig Latin

- Pig: an **engine** for executing **data flows** in parallel on Hadoop.
- **Pig Latin**: the language for expressing data flows
- Pig Latin contains common data processing operators (**join**, **sort**, **filter**, …)
- **User defined functions** (UDFs): developers can write their own functions to read/process/store the data

Pig is part of the CDH!!

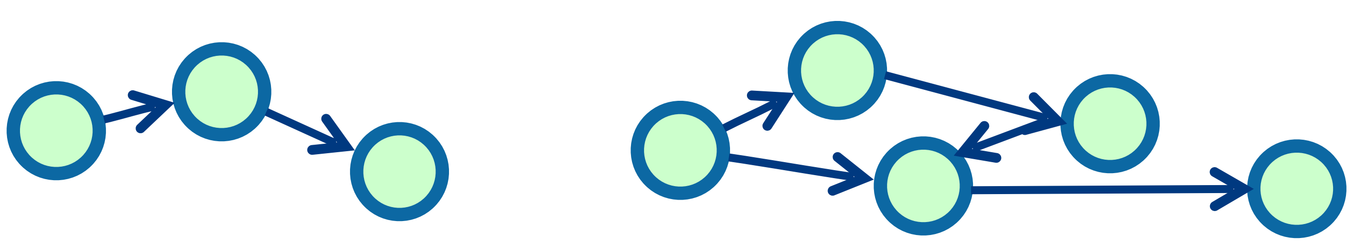
Pig on Hadoop

- Makes use of **HDFS** and the **MapReduce core** of Hadoop
  - By default, reads input from & writes output to HDFS
- Pig Latin scripts are **compiled** into **one or more** Hadoop jobs which are executed in order
- Pig Latin users need **not** to be aware of the algorithmic details in the map/shuffle/reduce phases
  - Pig **decomposes** operations into the appropriate map and/or map/reduce phases **automatically**

Pig Latin

- A parallel **dataflow language**: users describe **how** data is read, processed and stored
- Dataflows can be simple (e.g. “counting words”) or complex (multiple inputs are joined, data is split up into streams and processed separately)
- Formally: a Pig Latin script describes a **Directed Acyclic Graph**

directed graph, no directed cycles



Pig vs. OO & SQL

- OO programming languages describe control flow with data flow as side effect, Pig Latin describes data flow (**no control constructs** such as `if`)

Pig	SQL
Procedural: script describes **how** to process the data	Descriptive: query describes **what** the output should be
Workflows can contain many data processing operations	One query answers one question (*subqueries)
Schemas may be unknown or inconsistent	RDBMSs have defined schemas
Reads files from HDFS (and other sources)	Data is read from database tables

Pig vs. Hadoop

Pig	**Hadoop**
Standard data-processing operations are built-in (filter, join, group-by, order-by, …)	Group-by and order-by exist. Filtering and projection are easy to implement. Joins are hard work
Contains non-trivial implementations of data operators (e.g. for skewed key distributions reducer load can be rebalanced)	Load re-balancing based on key/value distributions not available
Error checking and optimization	Code within `map` & `reduce` is executed as-is
Pig Latin scripts are easy to understand, maintain and extend	Relatively opaque code with a lot of (ever changing) boilerplate
Few lines of code and a short development time	A large amount of boilerplate

Pig vs. Hadoop

Pig	Hadoop
Standard data-processing operations are built-in (filter, join, group-by, order-by, …)	Group-by and order-by exist. Filtering and projection are easy to implement. Joins are hard work
Contains non-trivial implementations of data operators (e.g. for skewed key distributions reducer load can be rebalanced)	Load re-balancing based on key/value distributions not available

Why then use Hadoop at all?

Pig heavily optimises **standard data operations**. Less common operations can be difficult to implement as Pig Latin is more restrictive than Hadoop.

https://cwiki.apache.org/confluence/display/PIG/PigMix

PigMix: Pig script benchmarks

A set of queries to test Pig's performance: how well does a Pig script perform compared to a direct Hadoop implementation?

Run date: August 27, 2009, run against top of trunk as of that day.

Test	Pig run time	Java run time	Multiplier
PigMix_1	218	133.33	1.635
PigMix_2	99.333	48	2.07
PigMix_3	272	127.67	2.13
PigMix_4	142.33	76.333	1.87
PigMix_5	127.33	107.33	1.19
PigMix_6	135.67	73	1.86
PigMix_7	124.67	78.333	1.59
PigMix_8	117.33	68	1.73

Pig 0.12 (4/4/2013)

Test	Pig run time	Java run time	Multiplier
PigMix_1	168	142	1.1830985915493
PigMix_2	71	62	1.14516129032258
PigMix_3	141	158	0.892405063291139
PigMix_4	93	87	1.06896551724138
PigMix_5	87	158	0.550632911392405
PigMix_6	93	81	1.14814814814815
PigMix_7	77	87	0.885057471264368
PigMix_8	62	57	1.08771929824561

PigMix: Pig script benchmarks

A set of queries to test Pig's performance: how well does a Pig script perform compared to a direct Hadoop implementation?

anti-join:

```
SELECT
*
FROM table1 t1
LEFT JOIN table2 t2 ON t1.id = t2.id
WHERE t2.id IS NULL
```

Run date: August 27, 2009, run agai

Test	Pig run time	Java	
PigMix_1	218	133.3	
PigMix_2	99.333	48	
PigMix_3	272	127.67	2.13
PigMix_4	142.33	76.333	1.87
PigMix_5	127.33	107.33	1.19
PigMix_6	135.67	73	1.86
PigMix_7	124.67	78.333	1.59
PigMix_8	117.33	68	1.73

			iplier
			30985915493
			516129032258
PigM	141	158	0.892405063291139
PigMix_4	93	87	1.06896551724138
PigMix_5	87	158	0.550632911392405
PigMix_6	93	81	1.14814814814815
PigMix_7	77	87	0.885057471264368
PigMix_8	62	57	1.08771929824561

Pig is useful for

- **ETL** (extract transform load) data pipelines
    - Example: web server logs that need to be cleaned before being stored in a data warehouse

- Research on raw data
    - Pig **handles erroneous**/corrupt **data** entries gracefully (cleaning step can be skipped)
    - Schema can be **inconsistent** or missing
    - Exploratory analysis can be performed **quickly**

- Batch processing
    - Pig Latin scripts are internally **converted to Hadoop jobs** (the same advantages/disadvantages apply)

http://pig.apache.org/philosophy.html

Pig philosophy

- **Pigs eat anything**
  - Pig operates on any data (schema or not, files or not, nested or not)
- **Pigs live anywhere**
  - Parallel data processing language; implemented on Hadoop but not tied to it
- **Pigs are domestic animals**
  - Easily controlled and modified
- **Pigs fly**
  - Fast processing

History of Pig

- Research project at **Yahoo! Research**
- Paper about Pig prototype published in **2008**
- **Motivation**:
  - Data scientists spent too much time writing Hadoop jobs and not enough time analysing the data
  - Most Hadoop users know SQL well
- **Apache top-level project** in 2010

Pig's version of WordCount

A screencast explaining the code line by line is available on Blackboard!
TI2736-B/Lectures/Lecture8/Screencast: first Pig example

```
-- read the file pg46.txt line by line, call each record line
cur = load 'pg46.txt' as (line);

-- tokenize each line, each term is now a record called word
words = foreach cur generate flatten(TOKENIZE(line)) as word;

-- group all words together by word
grpd  = group words by word;

-- count the words
cntd  = foreach grpd generate group, COUNT(words);

/*
 * start the Hadoop job and print results
 */
dump cntd;
```

5 lines of code in Pig vs. 50 in plain Hadoop

Pig's version of W

```
-- read the file pg46.txt line by line,
christmas_book = load ‘pg46.txt’ as (li

-- tokenize each line, each term is now
words = foreach input generate flatten(

-- group all words together by word
grpd  = group words by word;

-- count the words
cntd  = foreach grpd generate group, CO

/*
 * start the Hadoop job and print resul
 */
dump cntd;
```

5 lines of code i

```
(well-remembered,1)
(wine-merchant's,2)
(blindman's-buff.,1)
(entered--flushed,1)
(extinguisher-cap,1)
(highly-decorated,1)
(http://pglaf.org,2)
(including--which,1)
(knocker!--Here's,1)
(notwithstanding.,2)
(self-accusatory.,1)
(stagnant-blooded,1)
(unenforceability,1)
(fellow-'prentice.,1)
(fellow-passengers,1)
(five-and-sixpence,1)
(gbnewby@pglaf.org,1)
(sticking-plaister,1)
(strait-waistcoat.,1)
(surprised-looking,1)
(thread-the-needle,1)
(www.gutenberg.net,3)
(pleasantest-spoken,1)
(shabby--compounded,1)
(business@pglaf.org.,1)
(trademark/copyright,1)
(counting-house--mark,1)
(http://www.pglaf.org.,1)
(weathercock-surmounted,1)
(http://pglaf.org/donate,1)
(http://www.gutenberg.net,1)
(snowball--better-natured,1)
(http://gutenberg.net/license,1)
(http://pglaf.org/fundraising.,1)
(http://www.gutenberg.net/4/46/,1)
(,0)
grunt>
```

Another example:

Top clicked URL by users age 18-25

John	18
Tom	24
Alfie	45
Ralf	56
Sara	19
Marge	27

users: name & age

John	url1
John	url2
Tom	url1
John	url2
Ralf	url4
Sara	url3
Sara	url2
Marge	url1

clicks: name & url

```
set io.sort.mb 5;
-- the top URL clicked by users age 18-25
users = load ‘users’ as (name,age);
filtered = filter users by age>=18 and age<=25;
clicks = load ‘clicks’ as (user,url);
joined = join filtered by name, clicks by user;

grouped = group joined by url;
summarized = foreach grouped generate group, COUNT(joined) as
         amount_clicked;
sorted = order summarized by amount_clicked desc;

top1 = limit sorted 1;
Store top1 into 'top1site';
```

A screencast explaining the code line by line is available on Blackboard!
TI2736-B/Lectures/Lecture8/Screencast: top clicked URL

Pig is customisable

- All parts of the processing path are customizable
  - Loading
  - Storing
  - Filtering
  - Grouping
  - Joining
- Can be altered by **user-defined functions** (UDFs)

Grunt: running Pig

- Pig's interactive shell

testing: local file system

real analysis: HDFS

- Grunt can be started in **local** and **MapReduce mode**

```
pig –x local        pig
```

Errors do not kill the chain of commands

- Useful for sampling data (a pig feature)
- Useful for prototyping: scripts can be entered interactively
  - Basic syntax and semantic checks (errors do not kill the chain of commands)
  - Pig executes the commands (starts a chain of Hadoop jobs) once **`dump`** or **`store`** are encountered

Grunt: running Pig

- Pig's interactive shell

testing: local file system

real analysis: HDFS

- Grunt can be started in **local** and **MapReduce mode**

`pig –x local` `pig`

Errors do not kill the chain of commands

- Useful for sampling data (a pig feature)

- Useful for prototyping: scripts can be entered interactively
  - Basic syntax and semantic checks (errors do not kill the chain of commands)
  - Pig executes the command… jobs) once **`dump`** or **`store`**

Other ways of running Pig Latin:

(1) `pig script.pig`

(2) Embedded in Java programs (`PigServer` class)

Pig's data model

null: value unknown (SQL-like)

java.lang.String

- Scalar types: `int, long, float, double, chararray, bytearray`

DataByteArray, wraps `byte[]`

- Three complex types that can contain data of any type (nested)
  - **Maps**: chararray to data element mapping (values can be of different types) `[name#John,phone#5551212]`
  - **Tuples**: ordered collection of Pig data elements; tuples are divided into fields; analogous to rows (tuples) and columns (fields) in database tables `(John,18,4.0F)`
  - **Bags**: unordered collection of tuples (tuples cannot be referenced by position) `{(bob,21),(tim,19),(marge,21)}`

Schemas

- Remember: pigs eat anything
- Runtime declaration of schemas
- Available schemas used for error-checking and optimization

Pig reads three fields per line, **truncates** the rest; **adds null** values for missing fields

```
[cloudera@localhost ~]$ pig –x local
grunt> records = load 'table1' as (name:chararray, syear:chararray,
>>grade:float);

grunt> describe records;
records: {name: chararray,syear: chararray,grade: float}
```

as indicates the schema.

Schemas

- What about data with 100s of columns of known type?
  - Painful to add by hand every time
  - Solution: store schema in metadata repository Apache HCatalog – Pig can communicate with it

table and storage management layer - offers a relational view of data in HDFS.

- Schemas are not necessary (but useful)

A guessing game

column names, no types

```
[cloudera@localhost ~]$ pig –x local
grunt> records = load 'table1' as (name,syear,grade);
grunt> describe records;
records: {name: bytearray,syear: bytearray,grade: bytearray}
```

- Pig makes intelligent type guesses based on data usage (remember: nothing happens before we use the **`dump/store`** commands)

- If it is not possible to make a good guess, Pig uses the bytearray type (default type)

Default names

column types, no names

```
grunt> records2 = load 'table1' as(chararray,chararray,float);
grunt> describe records2;
records2: {val_0: chararray, val_1: chararray,val_2: float}
```

- Pig assigns default names if none are provided
- Saves typing effort, makes complex programs difficult to understand …

No need to work with unwanted content

Read only the first column

```
grunt> records3 = load 'table1' as(name);
grunt> dump records3;
(bob)
(jim)
. . .
```

- We can select which file content we want to process

More columns than data

```
grunt> records4 = load 'table1' as(name,syear,grade,city,bsn);
grunt> dump records4;
(bob,1st_year,8.5,,)
(jim,2nd_year,7.0,,)
(tom,3rd_year,5.5,,)
..
```

The file contains 3 "columns" – the remaining two columns are set to null

- Pig **does not throw an error** if the schema das not match the file content!
- Necessary for large-scale data where corrupted/incompatible entries are common
- Not so great for debugging purposes

Pig: loading & storing

```
[cloudera@localhost ~]$ pig —x local
grunt> records = load ‘table1’ as (name:chararray,
>> syear:chararray, grade:float);
grunt> describe records;
records: {name: chararray,syear: chararray,grade: float}
grunt> dump records;
(bob,1st_year,8.5)
(jim,2nd_year,7.0)
(tom,3rd_year,5.5)
…
```

relation consisting of tuples

```
grunt> store records into ‘stored_records’ using PigStorage(‘,’);
grunt> store records into ‘stored_records2’;
```

Pig: loading & storing

tab separated text file

```
[cloudera@localhost ~]$ pig –x local
grunt> records = load ‘table1’ as (name:chararray,
>> syear:chararray, grade:float);
grunt> describe records;
records: {name: chararray, ,grade: float}
grunt> dump records;
(bob,1st_year,8.5)
(jim,2nd_year,7.0)
(tom,3rd_year,5.5)
…
```

local file (URI)

dump runs a Hadoop job and writes output to screen

relation consisting of tuples

delimiter

```
grunt> store records into ‘stored_records’ using PigStorage(‘,’);
grunt> store records into ‘stored_records2’;
```

default output is tab delimited

store runs a Hadoop job and writes output to

Pig: loading and storing

```
[cloudera@localhost ~]$ ls stored_records/
part-m-00000  _SUCCESS
[cloudera@localhost ~]$ more stored_records/part-m-00000
bob,1st_year,8.5
jim,2nd_year,7.0
tom,3rd_year,5.5
andy,2nd_year,6.0
bob2,1st_year,7.5
tim,2nd_year,8.0
cindy,1st_year,8.5
arie,2nd_year,6.5
jane,1st_year,9.5
tijs,1st_year,8.0
claudia,2nd_year,7.5
mary,3rd_year,9.5
mark,3rd_year,8.5
john,,
ralf,,
[cloudera@localhost ~]$
```

`store` is a Hadoop job with only a map phase: part-m-*** (reducers output part-r-****)**

Relational operations

Transform the data by sorting, grouping,
joining, projecting, and filtering.

foreach

- Applies a set of expressions to every record in the pipeline
- Generates new records
- Equivalent to the projection operation in SQL

```
grunt> records = load 'table2' as (name,year,grade_1,grade_2);
grunt> gradeless_records = foreach records generate name,year;
grunt> gradeless_records = foreach records generate ..year;

grunt> diff_records = foreach records generate $3-$2,name;
```

foreach

- Applies a set of expressions to every record in the pipel
- Gene records
- Equiv e projection operation in SQL

```
(0.5,bob)
(-1.5,jim)
(-2.0,tom)
(1.0,andy)
(-3.0,bob2)
(1.0,tim)
(1.0,cindy)
(,arie)
(,jane)
(,tijs)
(,claudia)
(,mary)
(,mark)
(,john)
(,ralf)
grunt>
```

range of fields (useful when #fields is large)

```
grunt> r d 'table2' as (name
grunt> g ords = foreach records generate name,year;
grunt> g ords = foreach records generate ..year;

grunt> diff_records = foreach records generate $3-$2,name;
```

fields can be accessed by their position

foreach

- Evaluation function UDFs: take as input one record at a time and produce one output; Generates new records

```
grunt> records = load 'table1' as
                (name:chararray,year:chararray,grade:float);
grunt> grpd= group records by year;
grunt> avgs = foreach grpd generate group, AVG(records.grade);
grunt> dump avgs;
(1st_year,8.4)
(2nd_year,7.0)
(3rd_year,7.83333333)
(,)
```

Average: a built-in UDF

filter

- Select records to keep in the data pipeline

```
grunt> filtered_records = FILTER records BY grade>6.5;
grunt> dump filtered_records;
(bob,1st_year,8.5)
(jim,2nd_year,7.0)
…
grunt> filtered_records = FILTER records BY grade>8 AND
               (year=='1st_year' OR year=='2nd_year');
grunt> dump filtered_records;
(bob,1st_year,8.5)
(cindy,1st_year,8.5)
…
grunt> notbob_records = FILTER records
                  BY NOT name matches 'bob.*';
```

conditions can be combined

negation

regular expression

filter
inferred vs. defined data types

inferred

```
grunt> records = load ‘table1’ as (name,year,grade);
grunt> filtered_records = FILTER records BY grade>8
               AND (year==‘1st_year’ OR year==‘2nd_year’);
grunt> dump filtered_records;
```

inferred

```
grunt> records = load ‘table1’ as (name,year,grade);
grunt> filtered_records = FILTER records BY grade>8.0
               AND (year==‘1st_year’ OR year==‘2nd_year’);
grunt> dump filtered_records;
```

defined

```
grunt> records = load ‘table1’ as
               (name:chararray,year:chararray,grade:f
grunt> filtered_records = FILTER records BY grade>8
               AND (year==‘1st_year’ OR year==‘2nd_year’);
grunt> dump filtered_records;
```

A screencast explaining the code line by line is available on Blackboard!
TI2736-B/Lectures/Lecture8/Screencast: inferred vs. defined

group

- Collect records together that have the **same key**

```
grunt> grouped_records = GROUP filtered_records BY syear;
grunt> dump grouped_records;
(1st_year,{(bob,1st_year,8.5),(bob2,1st_year,7.5),(cindy,
1st_year,8.5),(jane,1st_year,9.5),(tijs,1st_year,8.0)})
(2nd_year,{(tim,2nd_year,8.0),(claudia,2nd_year,7.5)})
```

two tuples, grouped together by the first field

bag of tuples, indicated by {}

```
grunt> describe grouped_records;
grunt> grouped_records: {                    filtered_records:
{(name: chararray, syear: charray,grade: float)}}
```

name of grouping field

group

Question: if the pipeline is in the map phase, what has to happen?

```
grunt> grouped_records = GROUP filtered_records BY syear;
grunt> dump grouped_records;
```

Question: if the pipeline is in the reduce phase, what has to happen?

two tuples, grouped together by the first field

bag of tuples, indicated by {}

```
grunt> describe grouped_records;
grunt> grouped_records: {                    filtered_records:
{(name: chararray, syear: charray,grade: float)}}
```

name of grouping field

group

- There is no restriction on how many keys to group by

- All records with null keys end up in the same group

```
grunt> grouped_twice = GROUP records BY (year,grade);
grunt> dump grouped_twice;
```

- In the underlying Hadoop job effects depend on phase:
  - Map phase: a reduce phase is enforced
  - Reduce phase: a map/shuffle/reduce is enforced

group

- There is no restriction on how many keys to group by
- All record

```
grunt> gro
grunt> dum
```

- In the un phase:
  - Map p
  - Reduce phase: a map/shuffle/reduce is enforced

```
((1st_year,7.5),{(bob2,1st_year,7.5)})
((1st_year,8.0),{(tijs,1st_year,8.0)})
((1st_year,8.5),{(cindy,1st_year,8.5),(bob,1st_year,8.5)})
((1st_year,9.5),{(jane,1st_year,9.5)})
((2nd_year,6.0),{(andy,2nd_year,6.0)})
((2nd_year,6.5),{(arie,2nd_year,6.5)})
((2nd_year,7.0),{(jim,2nd_year,7.0)})
((2nd_year,7.5),{(claudia,2nd_year,7.5)})
((2nd_year,8.0),{(tim,2nd_year,8.0)})
((3rd_year,5.5),{(tom,3rd_year,5.5)})
((3rd_year,8.5),{(mark,3rd_year,8.5)})
((3rd_year,9.5),{(mary,3rd_year,9.5)})
((,),{(john,,),(ralf,,)})
grunt> grouped_twice = group records by (year,grade);
```

order by

- Total ordering of the output data (including across partitions)
- Sorting according to the natural order of data types
- Sorting by maps, tuples or bags is not possible

```
grunt> records = load 'table1' as (name,year,grade);
grunt> graded = ORDER records BY grade,year;
grunt> dump graded;
(ralf,,)
(john,,)
. .
(tijs,1st_year,8.0)
(tim,2nd_year,8.0)
. .
```

The results are first ordered by grade and within tuples of the same grade also by year.
Null values are ranked first.

`order by`

- Pig balances the output across reducers
  1. Samples from the input of the order statement
  2. Based on the sample of the key distribution a “fair” partitioner is built

An additional Hadoop job for the sampling procedure is required.

Same key to different reducers!

- Example of sampled keys (3 reducers available):

`a a a a c d x y z` {a, (a,c,d), (x,y,z)}

distinct

- Removes duplicate **records**

```
grunt> year_only = foreach records generate year;
grunt> uniq_years = distinct year_only;
(1st_year)
(2nd_year)
(3rd_year)
()
```

Works on entire records only, thus first a projection (line 1) is necessary.

Question: do we need a map and/or reduce phase here?

join

- THE workhorse of data processing

```
grunt> records1 = load ‘table1’ as (name,year,grade);
grunt> records2 = load ‘table3’ as (name,year,country,km);
grunt> join_up = join records1 by (name,year),
                      records2 by (name,year);
grunt> dump join_up;
(jim,2nd_year,7.0,jim,2nd_year,Canada,164)
(tim,2nd_year,8.0,tim,2nd_year,Netherlands,)
. . .
```

- Pig also supports outer joins (values that do not have a match on the other side are included): left/right/full

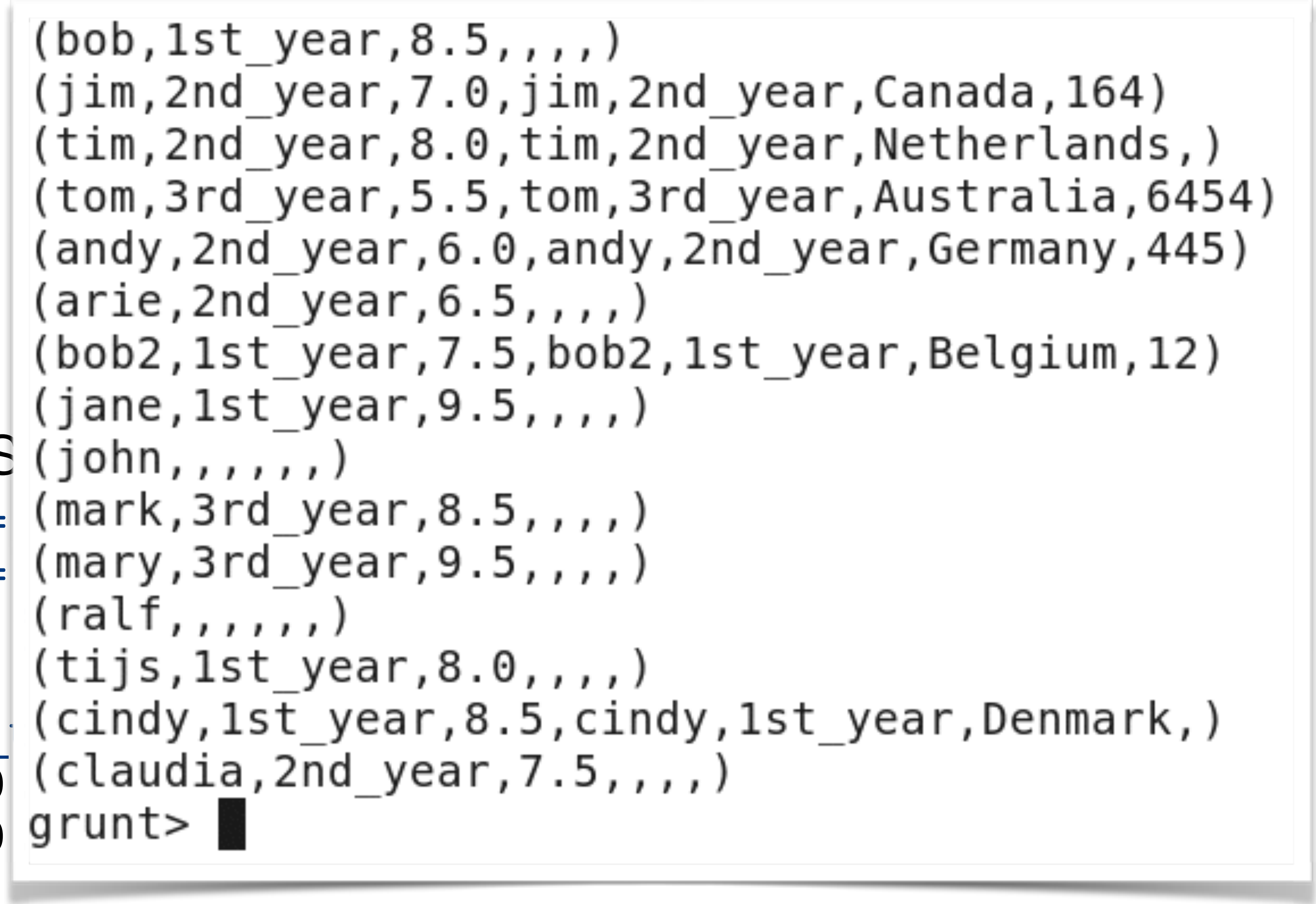
```
grunt> join_up = join records1 by (name,year) left outer,
                      records2 by (name,year);
```

join

- THE workhors

```
grunt> records1 =
grunt> records2 =
grunt> join_up =

grunt> dump join_
(jim,2nd_year,7.0
(tim,2nd_year,8.0
. . .
```



```
(bob,1st_year,8.5,,,,)
(jim,2nd_year,7.0,jim,2nd_year,Canada,164)
(tim,2nd_year,8.0,tim,2nd_year,Netherlands,)
(tom,3rd_year,5.5,tom,3rd_year,Australia,6454)
(andy,2nd_year,6.0,andy,2nd_year,Germany,445)
(arie,2nd_year,6.5,,,,)
(bob2,1st_year,7.5,bob2,1st_year,Belgium,12)
(jane,1st_year,9.5,,,,)
(john,,,,,,)
(mark,3rd_year,8.5,,,,)
(mary,3rd_year,9.5,,,,)
(ralf,,,,,,)
(tijs,1st_year,8.0,,,,)
(cindy,1st_year,8.5,cindy,1st_year,Denmark,)
(claudia,2nd_year,7.5,,,,)
grunt>
```

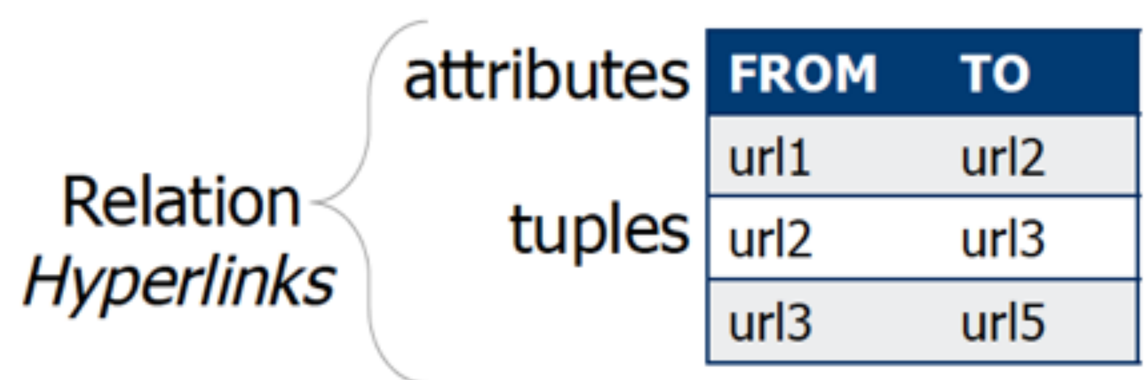
- Pig also supports outer joins (values that do not have a match on the other side are included): left/ right/full

```
grunt> join_up = join records1 by (name,year) left outer,
                      records2 by (name,year);
```

join

- Self-joins are supported, though data needs to be loaded twice - very useful for graph processing problems

```
grunt> urls1 = load ‘urls’ as (A,B);
grunt> urls2 = load ‘urls’ as (C,D);
grunt> path_2 = join urls1 by B, urls2 by C;
grunt> dump path_2;
(url2,url1,url1,url2)
(url2,url,url1,url4)
(url2,url1,url1,url3)
. . .
```



- Pig **assumes** that the **left** part of the join is the **smaller** data set

limit

- Returns a limited number of records
- Requires a reduce phase to count together the number of records that need to be returned

```
grunt> urls1 = load ‘urls’ as (A,B);
grunt> urls2 = load ‘urls’ as (C,D);
grunt> path_2 = join urls1 by B, urls2 by C;
grunt> first = limit path_2 1;
grunt> dump first;
(url2,url1,url1,url2)
```

- No ordering guarantees: every time limit is called it may return a different ordering

illustrate

A sample command exists, e.g.

```
some = sample path 0.5;
```

to sample 50% of the data.

- Creating a sample data set from the complete one
  - Concise: small enough to be understandable to the developer
  - Complete: rich enough to cover all (or at least most) cases
- Random sample can be problematic for filter & join operations
- Output is easy to follow, allows programmers to gain insights into what the query is doing

```
grunt> illustrate path;
```

illustrate

```
---------------------------------------------------
| urls1      | A:bytearray    | B:bytearray      |
---------------------------------------------------
|            | url2           | url1             |
|            | url1           | url2             |
|            | url5           | url1             |
|            | url1           | url4             |
---------------------------------------------------
---------------------------------------------------
| urls2      | C:bytearray    | D:bytearray      |
---------------------------------------------------
|            | url2           | url1             |
|            | url1           | url2             |
|            | url5           | url1             |
|            | url1           | url4             |
---------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
| path      | urls1::A:bytearray    | urls1::B:bytearray    | urls2::C:bytearray    | urls2::D:bytearray    |
------------------------------------------------------------------------------------------------------------------------
|           | url2                  | url1                  | url1                  | url2                  |
|           | url2                  | url1                  | url1                  | url4                  |
|           | url5                  | url1                  | url1                  | url2                  |
|           | url5                  | url1                  | url1                  | url4                  |
------------------------------------------------------------------------------------------------------------------------
```

Summary

- Simple database operations translated to Hadoop jobs
- Introduction to Pig

THE END