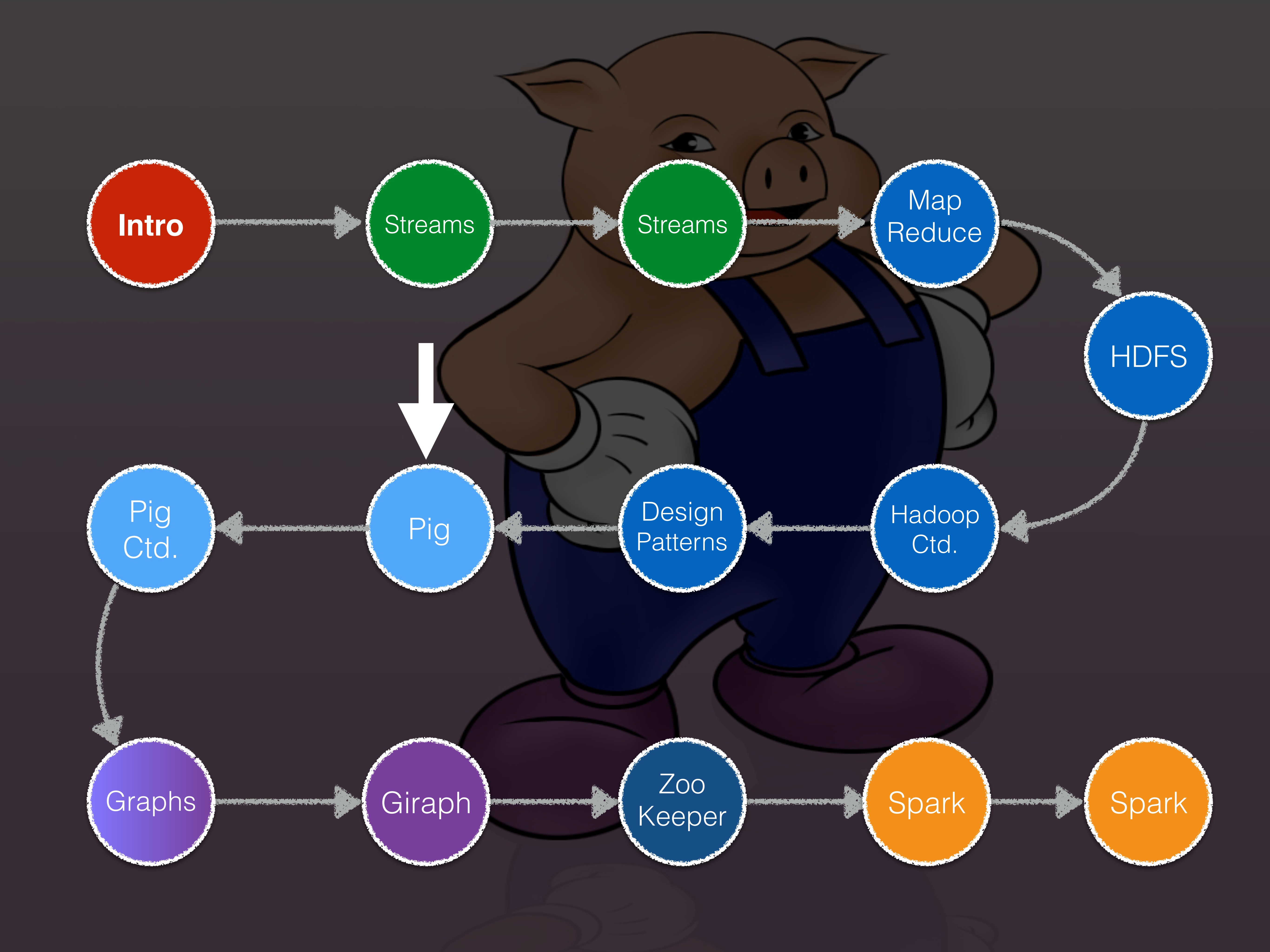
TI2736-B

Big Data Processing

Claudia Hauff
ti2736b-ewi@tudelft.nl

TU Delft


Intro
Streams
Streams
Map Reduce
HDFS
Hadoop Ctd.
Design Patterns
Pig
Pig Ctd.
Graphs
Giraph
Zoo Keeper
Spark
Spark

Learning objectives

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

- **Explain** the idea and mechanisms of UDFs

Introduction

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

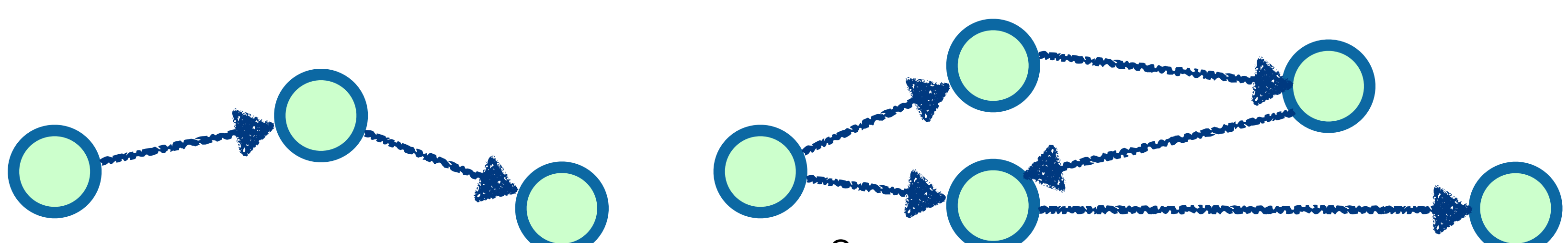
Pig 0.12 is part of the CDH

Pig 0.16 released 06/2016

Pig Latin

- A parallel **dataflow language**: users describe **how** data is (1) **read**, (2) **processed** and (3) **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 (DAG)**

directed graph, no directed cycles



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 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

PigMix: Pig 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

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

PigMix: Pig 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 ag

Test	Pig run time	Jav	
PigMix_1	218	133	[illegible]
PigMix_2	99.333	48	[illegible]
PigMix_3	272	[illegible]	[illegible]
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

			ltiplier
[illegible]	[illegible]	[illegible]	830985915493
[illegible]	[illegible]	[illegible]	4516129032258
[illegible]	[illegible]	[illegible]	89240506329139
PigM _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)

History of Pig

- Research project at **Yahoo! Research**
- Paper about Pig prototype published in **2008**


Pig Latin: A Not-So-Foreign Language for Data Processing

Christopher Olston * Yahoo! Research
Benjamin Reed † Yahoo! Research
Utkarsh Srivastava ‡ Yahoo! Research
Ravi Kumar § Yahoo! Research
Andrew Tomkins ¶ Yahoo! Research

>1800 citations

- **Data scientists** spent too much time writing Hadoop jobs and not enough time **analysing data**
- Most Hadoop users know **SQL** well
- **Apache top-level project** since 2010

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

First code examples

Pig’s version of WordCount

```
-- read file pg100.txt line by line, call each record line
shakespeare = load ‘pg100.txt’ as (line);

-- tokenize each line, each term is now a record called word
words = foreach shakespeare
            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 to screen
 */
dump cntd;
```

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

https://www.youtube.com/watch?v=s4Y-Yv5HY_A

Another example:

Top clicked URL by users aged18-25

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;

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';
```

9 lines of code in Pig vs. ~150 in plain Hadoop

Another example:

Top clicked URL by users aged18-25

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

field name

relation name (alias); not a variable

```
set io.sort.mb 5;

users = load 'users' as (name,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';
```

9 lines of code in Pig vs. ~150 in plain Hadoop

https://www.youtube.com/watch?v=76GbK8REmuo

cloudera

Alternative script

Top clicked URL by users aged18-25

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;
A = load ‘users’ as (name,age);
A = filter A by age>=18 and age<=25;
B = load ‘clicks’ as (user,url);
A = join A by name, clicks by user;
A = group A by url;
A = foreach A generate group, COUNT(A);
A = order A by $1 desc;
A = limit A 1;
Store A into 'A';
```

positional reference, starts from $0

This works too! Not recommended: hard to debug & lost relations!

Pig is a bit quirky

Case Sensitivity

The names (aliases) of relations and fields are case sensitive. The names of Pig Latin functions are case sensitive. The names of parameters (see [Parameter Substitution]) and all other Pig Latin keywords (see [Reserved Keywords]) are case insensitive.

In the example below, note the following:

- The names (aliases) of relations A, B, and C are case sensitive.
- The names (aliases) of fields f1, f2, and f3 are case sensitive.
- Function names PigStorage and COUNT are case sensitive.
- Keywords LOAD, USING, AS, GROUP, BY, FOREACH, GENERATE, and DUMP are case insensitive. They can also be written as load, using, as, group, by, etc.
- UDF names are also case-sensitive

https://pig.apache.org/docs/r0.12.1/basic.html#case-sensitivity

Pig is customisable

- All parts of the processing path are customizable
  - Loading
  - Storing
  - Filtering
  - Grouping
  - Joining
- Can be altered by **user-defined functions** (UDFs)
  - Not just restricted to Java, also possible in **Python, Jython, Ruby, JavaScript** and **Groovy**

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
  - 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 fea
- Useful for **prototyping**: scripts can
  - Basic syntax and semantic chec
  - Pig executes the commands (sta once **dump** or **store** are encoun

Other ways of running Pig Latin:

(1) `pig script.pig`

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

(3) In CDH from Hue

https://www.youtube.com/watch?v=aLrlOzTHtiI

Drawback: Hue does not run Pig scripts in local mode (takes time)

Advantage: Pig editor has Assist functionality

Data model & schema

Recall: Pig's data model

`java.lang.String`

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

  DataByteArray, wraps `byte[]`

null: value unknown (SQL-like)

- **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)}`

Extracting data from complex types

```
[cloudera@localhost ~]$ pig –x local
grunt> bball = load 'baseball' as (name:chararray,
               team:chararray,
               position:bag{t:(p:chararray)},
               bat:map[]);
grunt> avg = foreach bball generate
               bat#'batting_average';
```

Extraction from **maps**: use **#** followed by the name of the key as string.

```
[cloudera@localhost ~]$ pig –x local
grunt> A = load ‘data’ as (t:tuple(x:int, y:int));
grunt> B = foreach A generate t.x, t.$1;
```

Extraction from **tuple**: use the dot operator

Schemas

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

```
[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}
```

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,
                                  syea                         float);

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

as indicates the schema.

Schemas

- What about data with **hundreds 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, but it also makes complex scripts difficult to understand & debug

No need to work with unwanted content

We can select which file content we want to process

Read only the first column

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

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 does not match the file content
- Necessary for large-scale data where **corrupted** and **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)
…
```

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

Pig: loading & storing

tab separated text file

```
[cloudera@localhos  ~]$ pig –x local
grunt> records = load 'table1' as (name:chararray,
                         sy ar:chararray, grade:float);
grunt> describe records;
records: {name: chararray,                ,grade: float}
grunt> dump records;
```

local file (URI)

```
(bob,1st_year,8.5)
(jim,2nd_year,7.0)
(tom,3rd_year,5.5)
…
```

dump runs a Hadoop job and writes output to screen

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 file

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

bob	1st_year	8.5	9.0
jim	2nd_year	7.0	5.5
tom	3rd_year	5.5	3.5
andy	2nd_year	6.0	7.0
bob2	1st_year	7.5	4.5
tim	2nd_year	8.0	9.0
cindy	1st_year	8.5	9.5
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		9.5	
ralf		2.5	

- 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

bob	1st_year	8.5	9.0
jim	2nd_year	7.0	5.5
tom	3rd_year	5.5	3.5
andy	2nd_year	6.0	7.0
bob2	1st_year	7.5	4.5
tim	2nd_year	8.0	9.0
cindy	1st_year	8.5	9.5
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		9.5	
ralf		2.5	

- Applies a set of expressions **to ever** the pipeline
- 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>
```

```
grunt> re                'table2' as (na                ;
grunt> gr           rds = foreach records generate name,year;
grunt> gr           rds = foreach records generate ..year;

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

range of fields (useful when #fields is large)

fields can be accessed by their position

foreach

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		9.5
ralf		2.5

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

```
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

```
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 (int)

```
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;
```

inferred (float)

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

defined (float)

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: {group: chararray,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** the 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> gr
grunt> du
```

```
((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);
```

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

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 (ascending sort).

`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.

Always enforces a reduce phase

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 workhorse of data processing

```
grunt> records1 = load ‘table1’ as (name,year,grade);
grunt> records2 =
grunt> join_up =

grunt> dump join_
(jim,2nd_year,7.
(tim,2nd_year,8.
…
```

```
(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>
```

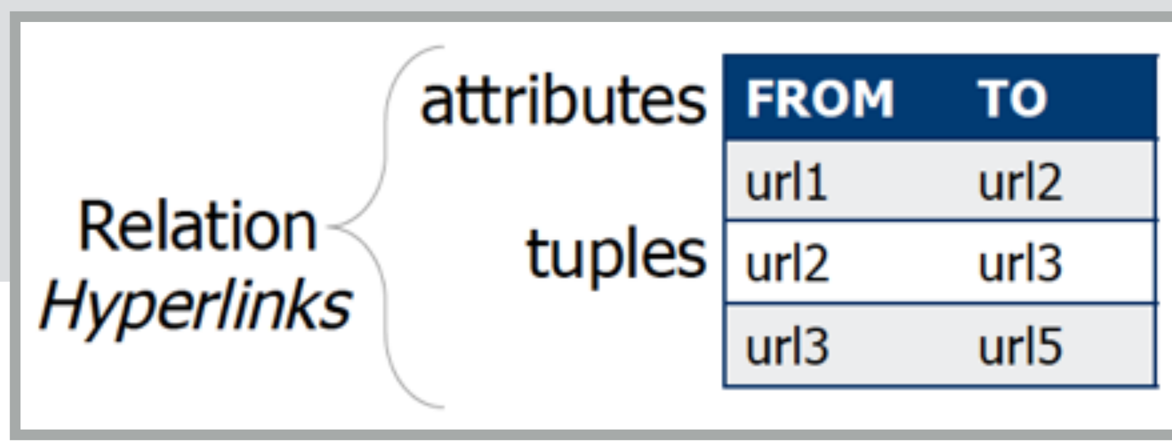
- Pig also supp
  have a match
  left/right/full

```
grunt> join_up =
```

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

Summary

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

THE END