Big Data Processing, 2014/15

Lecture 9-10: Advanced Pig Latin

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

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 objectives

- **Implement** Pig scripts that make use of complex data types and advanced Pig operators
- **Exploit** capabilities of Pig's preprocessor
- **Explain** the idea and mechanisms of UDFs
- **Implement** simple UDFs and deploy them

Question: Do you remember Pig's data types?

```
Jorge Posada   Yankees        {(Catcher),(Designated_hitter)}      [games#1594,grand_slams#7]
Landon Powell  Oakland        {(Catcher),(First_baseman)}          [on_base_percentage#0.297]
Martin Prado   Atlanta        {(Second_baseman),(Infielder)}       [games#258,hit_by_pitch#3]
```

What is this?

What is this?

What is this?

and this?

Recall: Pig's data model

`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)}`

Recall: bags

- Bags are used to store collections **when grouping**
- Bags can become quite large
- Bags can be spilled to disk
- Size of a bag is **limited** to the amount of **local disk space**
- **Only data type that does not need to fit into memory**

Recall: schemas

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

```
[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}
```

John	2002	8.0
Mary	2010	7.5
Martin	2011	7.0
Sue	2011	3.0

Complex data types in schema definitions

File: baseball (tab delimited)

```
1:Jorge Posada  Yankees {(Catcher),(Designated_hitter)} [games#1594,hit_by_pitch#65,grand_slams#7]
2:Landon Powell Oakland {(Catcher),(First_baseman)}     [on_base_percentage#0.297,games#26,home_runs#7]
3:Martin Prado  Atlanta {(Second_baseman),(Infielder),(Left_fielder)}  [games#258,hit_by_pitch#3]
```

bag of tuples, one field each   Map with chararray key

```
[cloudera@localhost ~]$ pig –x local
grunt> player = load ‘baseball’ as
          ( name:chararray, team:chararray,
            pos:bag{(p:chararray)}, bat:map[]);

grunt> describe player;

player: {name: chararray,team: chararray,
         pos: {(p:chararray)},bat: map[]}
```

Bags & tuples are part of the file format!

Data source: https://github.com/alanfgates/programmingpig

Case sensitivity & comments

- Careful: Pig Latin **mixes** case sensitive with non-sensitive
  - Keywords are **not** case sensitive (LOAD == load)
  - Relations and field names are case sensitive (`A = load ‘foo’` is not the same as `a = load ‘foo’`)
  - UDF names are case sensitive (COUNT not the same as count)

- 2 types of comments
  - SQL-style single-line comments (`--`)
  - Java-style multiline comments (`/* … */`)
  - `A = load ‘foo’; --loading data`
  - `B = load /* loading data */’bar’;`

Inferred vs. defined type

```
grunt> records = load 'table4' as (name,year,grade1,grade2);
grunt> filtered_records = FILTER records BY grade2 > grade1;
grunt> dump filtered_records;
(bob,1st_year,8,99)
(tom,3rd_year,35,4)
(andy,2nd_year,60,9)
```

'>' is not enough to infer a type comparison byte by byte

```
grunt> records = load 'table4' as (name,year,grade1,grade2:int);
grunt> filtered_records = FILTER records BY grade2 > grade1;
grunt> dump filtered_records;
(bob,1st_year,8,99)
(jane,1st_year,7,52)
(mary,3rd_year,9,11)
```

```
grunt> records = load 'table4' as (name,year,grade1,grade2);
grunt> filtered_records = FILTER records BY (grade2+0) > grade1;
grunt> dump filtered_records;
(bob,1st_year,8,99)
(jane,1st_year,7,52)
(mary,3rd_year,9,11)
```

bob	**1st_year**	**8**	**99**
jim	**2nd_year**	**7**	**52**
tom	3rd_year	35	4
andy	2nd_year	60	9
bob2	**1st_year**	**9**	**11**

Casts

- Explicit casting is possible

```
grunt> records = load 'table5' as (  name, year,
                                      grade1, grade2);
grunt> filtered = FILTER records BY (int)grade2>grade1;
grunt> dump filtered;
```

- Pig's implicit casts are always widening
  - `int*float` becomes `(float)int*float`
- Casting between scalar types is allowed; not allowed from/to complex types
- Casts from bytearrays are allowed
  - Not easy: int from ASCII string, hex value, etc.?

Not everything is straight-forward

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
jane	1st_year	9.5
mary	3rd_year	9.5

Counting lines in Pig

1. **Loading** the data from file

2. **Grouping** all rows together into a **single group**

3. **Counting** the number of elements in each group (since there is only one, the output will be the number of lines in the file)

Counting lines in Pig

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
jane	1st_year	9.5
mary	3rd_year	9.5

```
[cloudera@localhost ~]$ pig –x local

grunt> records = load ‘table1’ as (name,year,grade);
grunt> describe records;

records: {name: bytearray,year: bytearray,grade: bytearray}

grunt> A = group records all;
grunt> describe A;

A: {group: chararray, records: {(name: bytearray,year:
bytearray,grade: bytearray)}}

grunt> dump A;

(all, {(bob,1st_year,8.5),(jim,2nd_year,7.0),….(mary,3rd_year,
9.5)})
grunt> B = foreach A generate COUNT(records);
grunt> dump B;
(7)
```

keyword!

`COUNT()`: evaluation function, input is an expression of data type bag

Advanced Pig Latin operators

```
A = load ‘input’ as (user,id,phone);
B = foreach A generate user,id;
```

foreach

- Applies a set of expressions to **every record** in the pipeline

- Map projection

```
grunt> bb = load ‘baseball’ as (name:chararray, team:chararray,
                    position:bag{t:(p:chararray)}, bat:map[]);
grunt> avg = foreach bb generate bat#’batting_average’;
```

What if we also want the player’s name?

- Tuple projection

```
grunt> A = load ‘input_file’ as (t:tuple(x:int, :int));
grunt> B = foreach A generate t.x, t.$1;
```

- Bag projection: new bag is created with only wanted fields

```
grunt> A = load ‘input_file’ as (b:bag{t:(x:int,y:int)});
grunt> B = foreach A generate b.(x,y);
```

`foreach:` Extracting data from complex types

- **Remember**: numeric operators are not defined for bags

- **Example**: sum up the total number of students at each university

```
grunt> A = load 'tmp' as (x:chararray, d, y:int, z:int);
grunt> B = group A by x; --produces bag A containing all vals for x
B: {group: chararray,A: {(x: chararray,d: bytearray,y: int,z: int)}}
grunt> C = foreach B generate group,SUM(A.y + A.z);
ERROR!
grunt> A = load 'tmp' as (x:chararray, d, y:int, z:int);
grunt> A1 = foreach A generate x, y+z as yz;
grunt> B = group A1 by x;
B: {group: chararray,A1: {(x: chararray,yz: int)}}
grunt> C = foreach B generate group,SUM(A1.yz);
(UT,410)
(TUD,541)
(UvA,568)
```

A.y, A.z are

TUD	EWI	200	123
UT	EWI	235	54
UT	BS	45	76
UvA	EWI	123	324
UvA	SMG	23	98
TUD	AE	98	12

foreach flatten

- Removing levels of nesting
    - E.g. input data has bags to ensure one entry per row

```
1:Jorge Posada  Yankees {(Catcher),(Designated_hitter)} [games#1594,hit_by_pitch#65,grand_slams#7]
2:Landon Powell Oakland {(Catcher),(First_baseman)}     [on_base_percentage#0.297,games#26,home_runs#7]
3:Martin Prado  Atlanta {(Second_baseman),(Infielder),(Left_fielder)}   [games#258,hit_by_pitch#3]
```

- Data pipeline might require the form

```
Catcher                  Jorge Posada
Designated_hitter        Jorge Posada
Catcher                  Landon Powell
First_baseman            Landon Powell
Second_baseman           Martin Prado
Infielder                Martin Prado
Left_field               Martin Prado
```

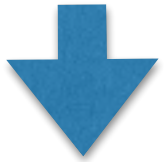
foreach flatten

- Flatten modifier in foreach

```
grunt> bb = load 'baseball' as (name:chararray, team:chararray,
position:bag{t:(p:chararray)}, bat:map[]);
grunt> pos = foreach bb generate flatten(position) as position, name;
grunt> grouped = group pos by position;
```

- Produces a **cross product** of every record in the bag with all other expressions in the generate statement

```
Jorge Posada,Yankees,{(Catcher),(Designated_hitter)}
```



```
Jorge Posada,Catcher
Jorge Posada,Designated_hitter
```

`foreach flatten`

- Flatten can also be applied to **tuples**
- Elevates each field in the tuple to a top-level field
- Empty tuples/empty bags will remove the entire record
- Names in bags and tuples are carried over after the flattening

nested foreach

- Foreach can apply a set of relational operations to **each record** in a pipeline

- Also called "*inner foreach*"

Inside foreach only some relational operators are (currently) supported: `distinct, filter, limit, order`

- **Example:** finding the number of unique stock symbols

```
grunt> daily = load 'NYSE_daily' as (exchange,symbol);
grunt> grpd = group daily by exchange;
grunt> uniqct = foreach grpd {
           sym = daily.symbol;
           uniq_sym = distinct sym;
           generate group, COUNT(uniq_sym);
       };
```

indicate nesting

Only valid inside foreach: take an expression and create a relation

Last line must generate!

each record passed is treated one at a time

nested foreach

- **Example**: sorting a bag before it is passed on to a UDF that requires sorted input (by timestamp, by value, etc.)

```
register 'acme.jar';
define analyze com.acme.financial.AnalyzeStock();
daily = load 'NYSE_daily' as (exchange:chararray,
      symbol:chararray, date:chararray,
      open:float, high:float, low:float, close:float,
      volume:int, adj_close:float);
grpd = group daily by symbol;
analy= foreach grpd {
              sorted = order daily by date;
              generate group, analyze(sorted);
};
dump analy;
```

nested foreach

- **Example**: finding the top-k elements in a group

```
divs = load 'NYSE_dividends' as (exchange:chararray,
symbol:chararray, date:chararray, dividends:float);
grpd = group divs by symbol;
top3 = foreach grpd {
           sorted = order divs by dividends desc;
           top = limit sorted 3;
           generate group, flatten(top);
};
dump top3;
```

`nested foreach`

- Nested code portions run **serially** for each record (though it may not be strictly necessary)
- Foreach itself runs in **multiple map or reduce tasks** but each instance of the foreach will not spawn subtasks to do the nested operations in parallel
- **Non-linear pipelines** are also possible

parallel

Without parallel:
Pig uses a heuristic: one reducer for every GB of input data.

- Reducer-side parallellism can be controlled
- Can be attached to any relational operator
- Only makes sense for operators forcing a reduce phase:
  - group, join, cogroup [most versions]
  - order, distinct, limit, cross

```
grunt> A = load 'tmp' as (x:chararray, d, y:int, z:int);
grunt> A1 = foreach A generate x, y+z as yz;
grunt> B = group A1 by x parallel 10;
grunt> averages = foreach B generate group, AVG(A1.yz) as avg;
grunt> sorted = order averages by avg desc parallel 2;
```

partition

- Pig uses **Hadoop's default Partitioner**, except for order and skew join
- A **custom partitioner** can be set via keyword partition
- Operators that have a reduce phase can take the partition clause
  - Cogroup, cross, distinct, group, etc.

```
grunt> register acme.jar; --contains partitioner
grunt> users = load 'users' as (id, age, zip);
grunt> grp = group users by id partition by com.acme.cparti
                parallel 100;
```

file1:		file2:	
A	2	A	2
B	3	B	22
C	4	C	33
D	5	D	44

union

- Concatenation of two data sets

```
grunt> data1 = load 'file1' as (id:chararray, val:int)
grunt> data2 = load 'file2' as (id:chararray, val:int)
grunt> C = union data1, data2;
(A,2)
(B,22)
(C,33)
(D,44)
(A,2)
(B,3)
(C,4)
(D,5)
```

- Not a mathematical union, **duplicates remain**

- Does not require a reduce phase

union

file1:

A	2
B	3
C	4
D	5

file2:

A	2	John
B	22	Mary
C	33	Cindy
D	44	Bob

- Also works if the schemas **differ** in the inputs (unlike SQL unions)

```
grunt> data1 = load 'file1' as (id:chararray, val:float)
grunt> data2 = load 'file2' as (id:chararray, val:int, n:chararray)
grunt> C = union data1, data2;
(A,2,John)
(B,22,Mary)
(C,33,Cindy)
(D,44,Bob)
(A,2.0)
(B,3.0)
(C,4.0)
(D,5.0)
grunt> describe C;
Schema for C unknown.
grunt> C = union onschema data1, data2; dump C; describe C;
C: {id: chararray,val: float,name: chararray}
```

inputs **must** have schemas

shared schema: generated by adding fields and casts

cross

file1:

A	2
B	3
C	4
D	5

file2:

A	2
B	22
C	33
D	44

- Takes two inputs and crosses each record with each other

```
grunt> C = cross data1,
data2;
(A,2,A,11)
(A,2,B,22)
(A,2,C,33)
(A,2,D,44)
(B,3,A,11)
…
```

- **Crosses are expensive** (internally implemented as joins), a lot of data is send over the network

- Necessary for advanced joins, e.g. approximate matching (fuzzy joins): first cross, then filter

mapreduce

- Pig: makes many operations simple
- Hadoop job: higher level of customization, legacy code
- Best of both worlds: combine the two!
- MapReduce job expects HDFS input/output; Pig stores the data, invokes job, reads the data back

```
grunt> crawl = load ‘webcrawl’ as (url,pageid);
grunt> normalized = foreach crawl generate normalize(url);
grunt> mapreduce ‘blacklistchecker.jar’
           store normalized into ‘input’
           load ‘output’ as (url,pageid)
           `com.name.BlacklistChecker –I input –o output`;
```

Hadoop job parameters

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                   |
------------------------------------------------------------------------------------------------------------------
```

joins

join implementations

- Pig's join starts up Pig's **default implementation** of join
- **Different implementations** of join are available, exploiting knowledge about the data
- In RDBMS systems, the SQL optimiser chooses the "right" join implementation automatically
- In Pig, the **user is expected to make the choice** (knows best how the data looks like)
- Keyword **using** to pick the implementation of choice

join implementations

small to large

- Scenario: lookup in a smaller input, e.g. translate postcode to place name
  - Germany has >80M people, but only ~30,000 postcodes
- Small data set usually fits into memory
  - Reduce phase is unnecessary
- More efficient to send the smaller data set (e.g. zipcode-town file) to every data node, load it to memory and join by streaming the large data set through the Mapper
- Called **fragment-replicate join** in Pig, keyword replicated

replicated can be used with more than two tables.
The first is used as input to the Mapper, the rest is in memory.

```
grunt> jnd = join X1 by (y1,z1), X2 by (y2,z2) using 'replicated'
```

join implementations

skewed data

- Skew in the **number of records per key** (e.g. words in a text, links on the Web)
  - Remember Zipf's law!

- Default join implementation is sensitive to skew, all records with the same key sent to the same reducer

- Pig's solution: **skew join**
  1. Input for the join is **sampled**
  2. Keys are identified with too many records attached
  3. Join happens in a second Hadoop job
     1. Standard join for all "normal" keys (a single key ends up in a reducer)
     2. Skewed keys distributed over reducers (split to achieve in-memory split)

join implementations

skewed data

```
users(name,city)
city_info(city,population)
join city_info by city, users by city using 'skewed';
```

- Data set contains:
  - 20 users from Delft
  - 100,000 users from New York
  - 350 users from Amsterdam
- A reducer can deal with 75,000 records in memory.
- User records with key ‘New York’ are split across 2 reducers. Records from city_info with key New York are duplicated and sent to both reducers.

join implementations

skewed data

- In general:
  - Pig samples from the second input to the join and splits records with the same key if necessary
  - The first input has records with those values replicated across reducers
- Implementation optimised for one of the inputs being skewed
- Skew join can be done on inner and outer joins.
- Skew join can only take two inputs; multiway joins need to be broken up

Same caveat as order: breaking MapReduce of one key = one reducer. Consequence: same key distributed across different part-r-** files.

join implementations

sorted data

- **Sort-merge join** (database join strategy): first sort both inputs and then walk through both inputs together
  - Not faster in MapReduce than default join, as a sort requires one MapReduce job (as does the join)
- Pig's merge join can be used if both inputs are already sorted on the join key
  - **No reduce phase required**
  - Keyword is merge

```
grunt> jnd = join sorted_X1 by y1, sorted_X2 by y2 using 'merge'
```

join implementations

sorted data

- But: **files are split into blocks**, distributed in the cluster

```
grunt> jnd = join sorted_X1 by y1, sorted_X2 by y2 using ‘merge’
```

- **First MapReduce job** to sample from sorted_X2: job builds an index of input split and the value of its first (sorted) record

- **Second MapReduce job** reads over sorted_X1: the first record of the input split is looked up in the index built in step (1) and sorted_X2 is opened at the right block

- No further lookups in the index, both “record pointers” are advanced alternatively

Non-linear data flows

Non-linear data flows

- Linear data flow: an input is **loaded**, **processed** and **stored**
- **Tree structures**: multiple inputs flow into a single output
  - join, union & cross
- Also possible in Pig:
  - **Splitting of data flows**, i.e. more than one output
  - **Diamond shaped workflows**: data flow is split and later joined back together
- Data-flow splits can be implicit and explicit
  - You have seen implicit ones already (e.g. two different group operations on the same input)

Non-linear data flows

explicit splits

- With **split operator** a data flow can be split arbitrarily

- **Example**: split log data into different files depending on the data in the log record

```
grunt> load 'weblogs' as (pageid,url,timestamp);
grunt> split wlogs into apr03 if timestamp < '20130404',
       apr02 if timestamp < '20130403' and timestamp > '20130401',
       apr01 if timestamp < '20130402' and timestamp > '20130331';
grunt> store apr03 into 'april_03';
grunt> store apr02 into 'april_02';
grunt> store apr01 into 'april_01';
```

- Split is not switch/case
- A single record can go into multiple outputs
- A record may go nowhere
- No default case

Non-linear data flows

execution

- **Best case scenario**: combine them into a single MapReduce job
- Split example on previous slide occurs with one map phase
- Group operators can also be combined into a single job

Non-linear data flows

execution

- **Example**: group baseball players by team and position

too many multi query inputs and shuffling becomes the bottleneck

If some jobs succeed Pig script will continue. Only an error in all jobs will lead to script failure.

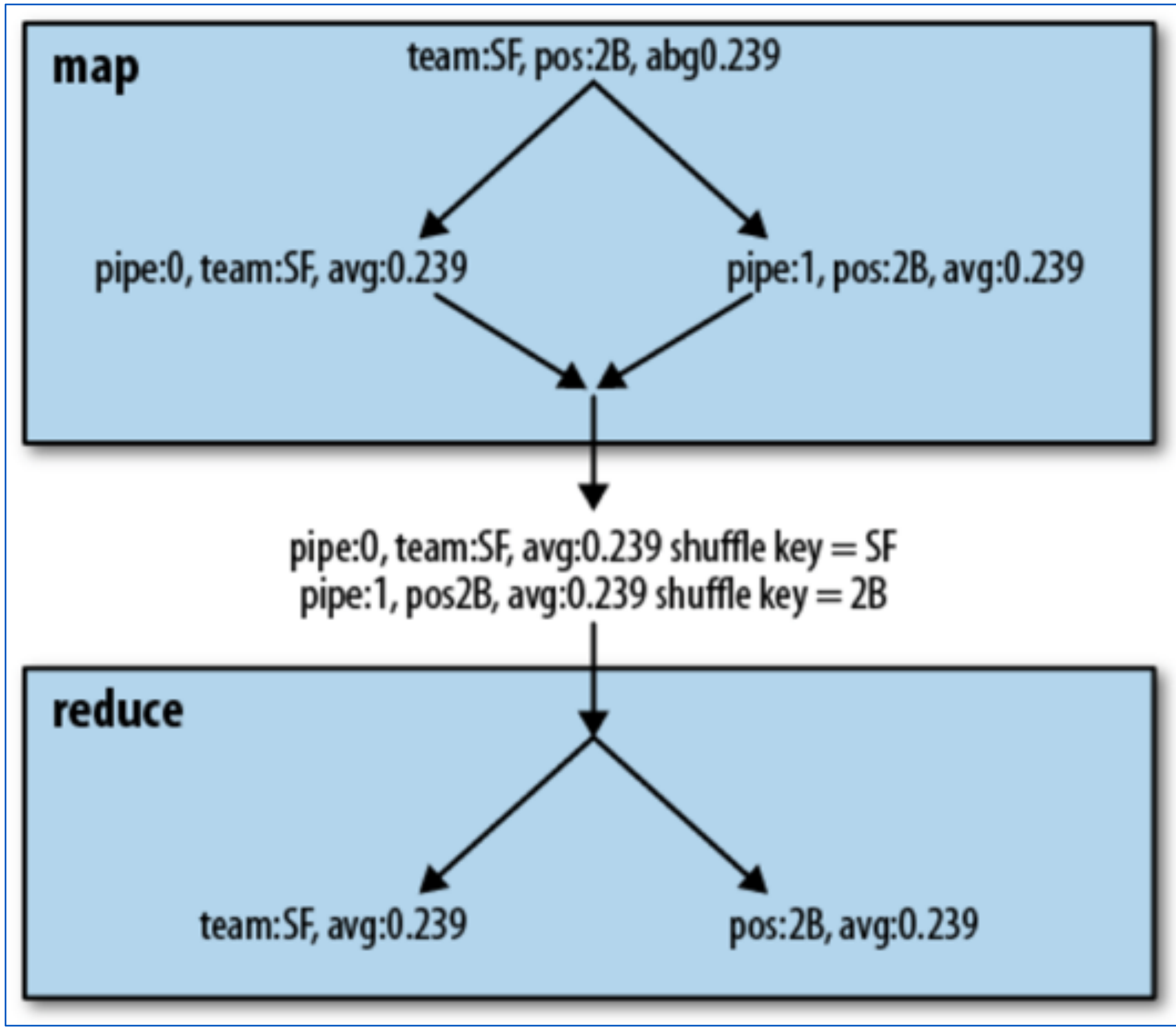


Image source: Programming Pig

How to script well

Scripting

- **Filter early and often**
  - Push filters as high as possible (Pig optimiser helps)
- **Make implicit filters explicit**
  - E.g. in a join with null values those records vanish; better to place a filter before the join
- **Project early and often**
  - Remove unneeded fields as soon as possible (network bandwidth….)
- **Choose the correct join** (understand input order)

Which join should you use?

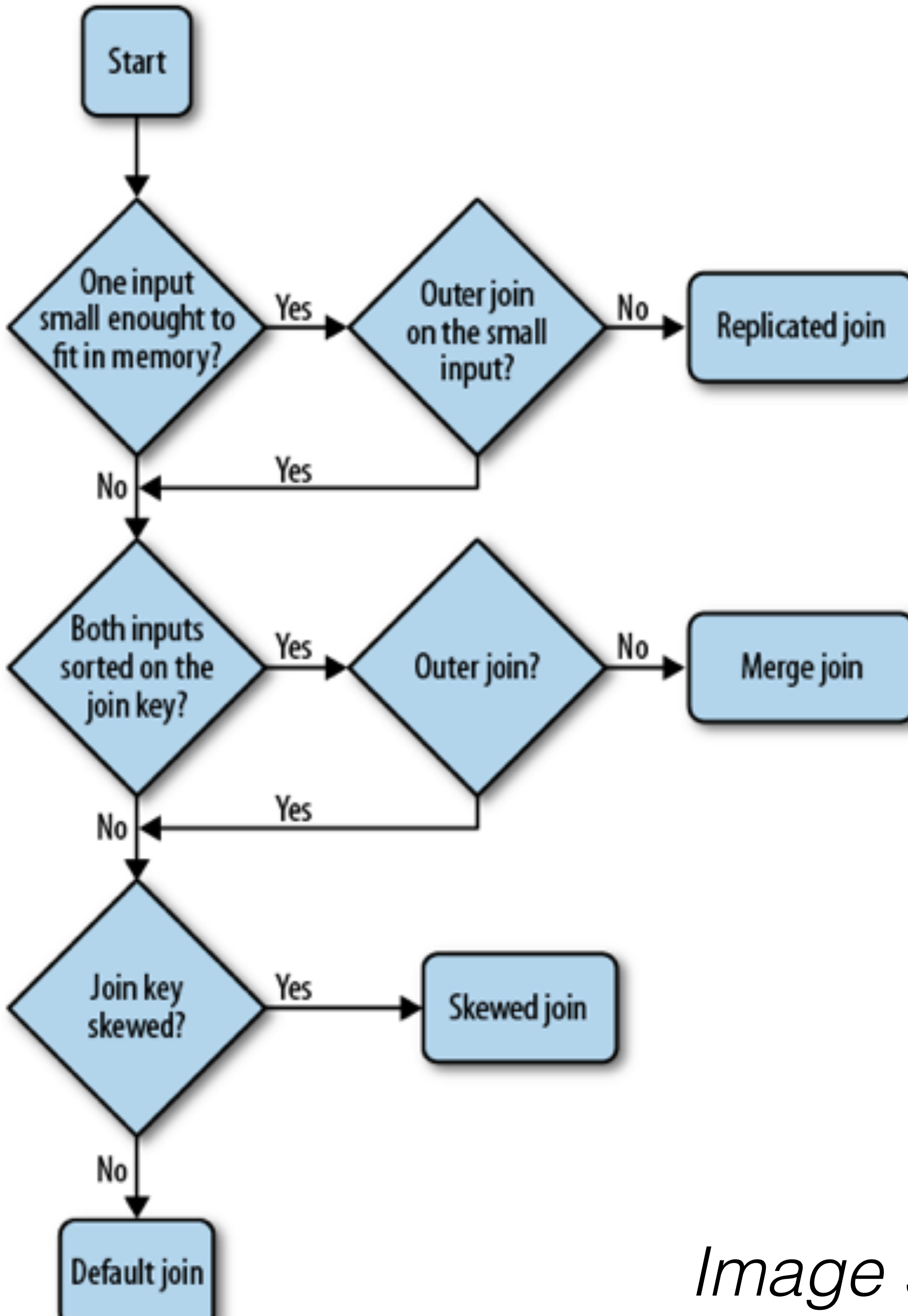


Image source: Programming Pig

Scripting

- **Choose the right data type**
  - Loading everything as byte array can have a cost (floating point arithmetics slower than integer arithmetics)

- **Select the right level of parallelism**

- **Select the right partitioner** (data skew)

Simplifying scripting

Pig Latin preprocessor

- Pig Latin preprocessor runs **before** the Pig Latin script is parsed
- Functionalities
    - Parameter substitution
    - Inclusion of other Pig scripts
    - Function-like macro definitions

Pig Latin preprocessor

parameter substitution

- **Example**: log files are aggregated once a day

```
--daily.pig
daily = load 'logfile' as (url,date,size);
yesterday = filter daily by dayte == '$DATE';
```

our input parameter

- Invocation: `pig –p DATE=2013-12-09 daily.pig`
- Multiple parameters possible via `-p -p -p`
- Parameter files can also be used as input

Pig Latin preprocessor

including other scripts

```
--helper.pig
define analysis(daily, date)
returns analyzed {
     $analyzed = …
};
```

imported file is written directly into the Pig script

```
--main.pig
import ' analysis.pig';
daily = load 'logfile' as (url,date,size);
results = analysis(daily,'2013-12-10');
```

Pig Latin preprocessor

define

- Using Java's full package names can be exhaustive

- **`define`** provides aliases

```
--define.pig
register '/usr/lib/pig/piggybank.jar';
define reverse org.apache.pig.piggybank.evaluation.string.Reverse();
unis = load 'tmp' as (name:chararray, num1: int, num2: int);
backward_uni_names = foreach unis generate reverse(name);
```

- Also the place for constructor arguments

```
define convert com.acme.financial.CurrencyConverter('dollar','euro');
```

Java static functions

- Java has a **rich collection of utilities and libraries**
- Invoker methods in Pig can make certain static Java functions appear as Pig UDFs
- **Java function requirements:**
  - Public static Java function
  - Zero input arguments or a combination of int, long, float, double, String (or arrays of this kind)
  - Returned is an int, long, float, double or String
- **Each return type has its invoker method:** InvokeForInt, InvokeForString, etc.

Java static functions

full package, class and method

parameters (types)

```
grunt> define hex InvokeForString('java.lang.Integer.toHexString','int');
grunt> nums = load 'numbers' as (n:int) --file with numbers, 1 per line
grunt> inhex = foreach nums generate hex(n);
(ff)
(9)
(11)
(9a)

grunt> define maxFloats InvokeForFloat('java.lang.Math.max','float float');
grunt> define imaginary InvokeForFloat('java.lang.Some.function','float[]');
```

User defined functions (UDF)

Three types of UDFs

- **Evaluation functions**: operate on single elements of data or collections of data
- **Filter functions**: can be used within `FILTER` statements
- **Load functions**: provide custom input formats
- Pig locates a UDF by looking for a Java class that exactly matches the UDF name in the script
- One UDF instance will be constructed and run in each map or reduce task (shared state within this context possible)

UDFs step-by-step

- **Example**: filter the student records by grade

```
grunt> filtered_records = FILTER records BY grade>7.5;
grunt> dump filtered_records;
(bob,1st_year,8.5)
…
```

> **not** a good class name choice; shortened for presentation purposes

- **Steps**

  1. Write a Java class that extends `org.apache.pig.FilterFunc`, e.g. `GoodStu`
  2. Package it into a JAR file, e.g. **`bdp.jar`**
  3. Register the JAR with Pig (in distributed mode, Pig automatically uploads the JAR to the cluster)

```
grunt> REGISTER bdp.jar;
grunt> filtered_records = FILTER records BY bdp.GoodStu(grade);
grunt> dump filtered_records;
```

UDF: FilterFunc

```
import pdb;
import java.io.exception;
import org.apache.pig.backend.executionengine.execexception
import org.apache.pig.data.tuple;
import org.apache.pig.filterfunc;

public class GoodStu extends FilterFunc {
        @Override
        public boolean exec(Tuple tuple) throws IOException {
                if(tuple==null || tuple.size()==0)
                        return false;
                try {
                        Object o = tuple.get(0);
                        float f = (Float)o;
                        if(f>7.5)
                                return true;
                        return false;
                }
                catch(ExecException e){
                        throw new IOException(e);
                }
        }
}
```

Use annotations if possible (they make debugging easier)

UDF: FilterFunc

Type definitions are not always optional anymore

```
grunt> records = load 'table' as (name,year,grade);
grunt> filtered_records = FILTER records
                 BY bdp.GoodStu(grade);
grunt> dump filtered_records;
java.lang.Exception: java.lang.ClassCastException:
Org.apache.pig.data.DataByteArray cannot be cast
to java.lang.Float
```

Summary

- Complex data types
- Advanced Pig operators
- Pig preprocessing
- How to script well
- UDFs

References

- **Hadoop: The Definite Guide** by Tom White. Available via TU Delft campus. **Chapter 11.**
  - http://proquest.safaribooksonline.com/book/software-engineering-and-development/9781449328917

- **Programming Pig** by Alan Gates. Available via TU Delft campus. **Chapters 6-9,10**.
  - http://proquest.safaribooksonline.com/book/-/9781449317881

THE END