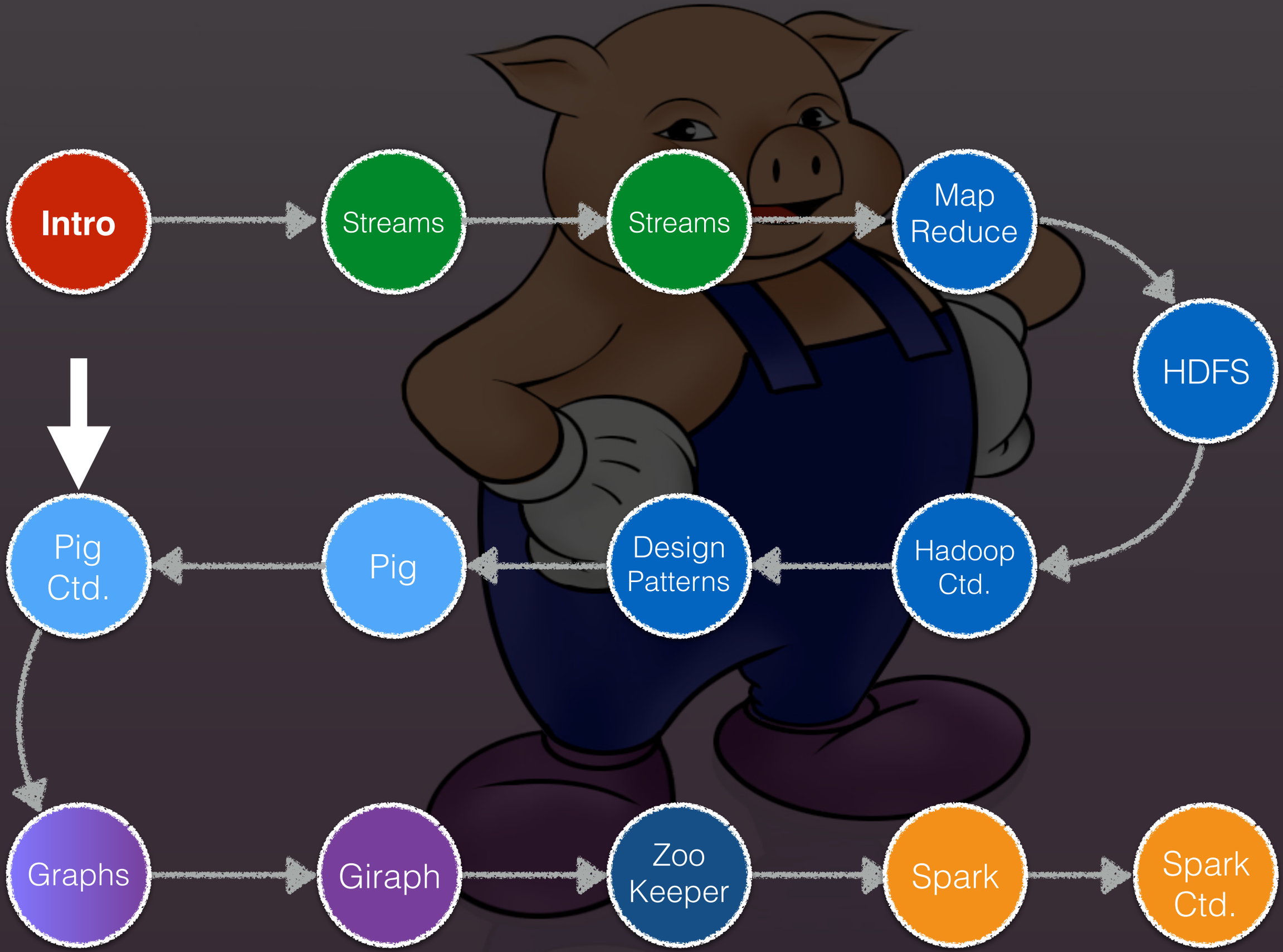
TI2736-B

Big Data Processing

Claudia Hauff
ti2736b-ewi@tudelft.nl

TU Delft


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

Learning objectives

- **Implement** Pig scripts that make use of complex data types and advanced Pig operators
- **Implement** simple UDFs and deploy them
- **Use** and **explain** Pig's different join implementations
- **Exploit** capabilities of Pig's preprocessor

Complex schemas

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

Recall: 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}
```

More about 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**

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

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, but **not easy**: `int` from ASCII string, hex value, etc.?

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

similar instructions hold for the other supported languages

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)
…
```

- Steps:
  1. Write a Java class that extends `org.apache.pig.FilterFunc`, e.g. `GoodStudent`
  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.GoodStudent(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 GoodStudent 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);
            }
      }
}
```

UDF: FilterFunc

```
import pdb;
i
i                                    xtend.executionengine.execexception
i                                    n.tuple;
i                                    terfunc;

public class GoodStudent 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
```

UDF: EvalFunc

type of return value

```
public class Trim extends EvalFunc<String> {
  @Override
  public String exec(Tuple input) throws IOException {
    if (input == null || input.size() == 0)
      return null;

    try {
      Object object = input.get(0);
      if (object == null)
        return null;
      return ((String) object).trim();
    }
    catch (ExecException e) {
      throw new IOException(e);
    }
  }
  @Override
  public List<FuncSpec> getArgToFuncMapping()
          throws FrontendException {
    //returns null, if the input is not
    //of DataType.CHARARRAY
  }
}
```

UDF: EvalFunc

type of return value

```
public class Trim extends EvalFunc<String> {
  @Override
  public String exec(Tuple input) throws IOException {
    if (input == null || input.size() == 0)
      return null;

    try {
      Object object = in
      if (object == null
        return null;
      return ((String) o
    }
    catch (ExecException
      throw new IOExcept
    }
  }
  @Override
  public List<FuncSpec>
        throws Frontend
    //returns null, if t
    //of DataType.CHARARRAY
  }
}
```

```
grunt> fruit = load 'fruit';
grunt> dump fruit;
(  apple)
( banana    )
( pear   )
grunt> REGISTER bdp.jar;
grunt> fruit_trimmed = foreach fruit
generate bdp.Trim($0);
grunt> dump fruit_trimmed;
(apple)
(banana)
(pear)
```

Not everything is straight-forward

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

Doesn't Pig have a COUNT function?

COUNT

Computes the number of elements in a bag.

Syntax

COUNT(expression)

Terms

expression	An expression with data type bag.

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!

foreach: Extracting data from complex types

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

- **Remember**: numeric operators are not defined for arbitrary bags
- **Example**: sum up the total number of students at each university

Doesn't Pig have a SUM function?

SUM

Computes the sum of the numeric values in a single-column bag. SUM requires a preceding GROUP ALL statement for global sums and a GROUP BY statement for group sums.

Syntax

```
SUM(expression)
```

foreach: Extracting data from complex types

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

- **Remember**: numeric operators are not defined for arbitrary bags
- **Example**: sum up the total number of students at each university

```
grunt> A = load 'data' as (x:chararray, d, y:int, z:int);
grunt> B = group A by x; --produces bag A containing all vals for x

grunt> C = foreach B generate group, SUM(A.y + A.z);

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 bags

More Pig Latin operators

foreach flatten

- **Removing** levels of **nesting**
- **Example**: 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)});
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

NYSE	CLI	2009-12-3135.39	35.70	34.50	34.57	890100	34.12
NYSE	CLI	2009-12-3035.22	35.46	34.96	35.40	516900	34.94
NYSE	CPE	2009-02-181.75	1.80	1.56	1.56	452200	1.56
NYSE	CPE	2009-02-171.91	1.95	1.75	1.75	361300	1.75
NYSE	CLB	2009-04-0775.17	75.92	73.75	75.52	339600	74.55
NYSE	CVA	2009-01-1220.46	20.88	19.91	20.24	700800	20.24

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);
       };
-(NYSE,237)
```

indicates 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

NYSE	CLI	2009-12-3135.39	35.70	34.50	34.57	890100	34.12
NYSE	CLI	2009-12-3035.22	35.46	34.96	35.40	516900	34.94
NYSE	CPE	2009-02-181.75	1.80	1.56	1.56	452200	1.56
NYSE	CPE	2009-02-171.91	1.95	1.75	1.75	361300	1.75
NYSE	CLB	2009-04-0775.17	75.92	73.75	75.52	339600	74.55
NYSE	CVA	2009-01-1220.46	20.88	19.91	20.24	700800	20.24

nested foreach

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

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

NYSE	CLI	2009-12-3135.39	35.70	34.50	34.57	890100	34.12
NYSE	CLI	2009-12-3035.22	35.46	34.96	35.40	516900	34.94
NYSE	CPE	2009-02-181.75	1.80	1.56	1.56	452200	1.56
NYSE	CPE	2009-02-171.91	1.95	1.75	1.75	361300	1.75
NYSE	CLB	2009-04-0775.17	75.92	73.75	75.52	339600	74.55
NYSE	CVA	2009-01-1220.46	20.88	19.91	20.24	700800	20.24

nested foreach

Example: finding the top-k elements in a group

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

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 `partition`

- Operators that have a **reduce phase** can take the partition clause

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

union

A	2
B	3
C	4
D	5

file 1

A	2
B	22
C	33
D	44

file 2

- **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

A	2	
B	3	
C	4	
D	5	

file 1

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

file 2

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,
                                name:chararray)
grunt> C = union data1, data2; dump C;
(A,2,John)          (A,2.0,John)
(B,22,Mary)         (B,22.0,Mary)
(C,33,Cindy)        (C,33.0,Cindy)
(D,44,Bob)          (D,44.0,Bob)
(A,2.0)             (A,2.0,)
(B,3.0)             (B,3.0,)
(C,4.0)             (C,4.0,)
(D,5.0)             (D,5.0,)
grunt> describe C;
Schema for C unknown.
grunt> D = union onschema data1, data2;
grunt> dump D; describe D;
D: {id: chararray, val: float,name: chararray}
```

inputs **must** have schemas

shared schema: generated by adding fields and casts

cross

A	2
B	3
C	4
D	5

file 1

A	11
B	22
C	33
D	44

file 2

- 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`

- 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
- Used to review how data is transformed through a sequence of Pig Latin statements
- Output is easy to follow, allows programmers to gain insights into what the query is doing

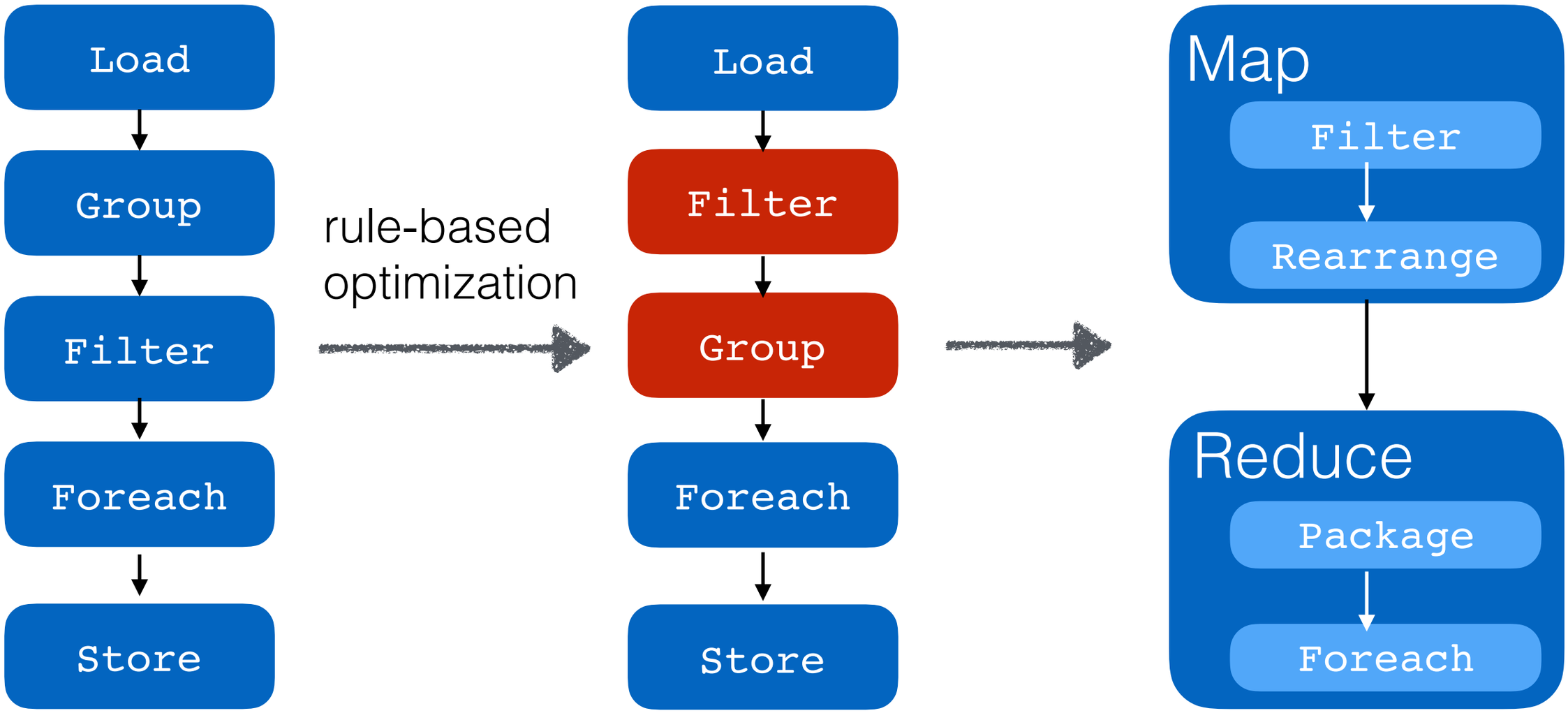
```
grunt> illustrate path;
```

`explain`

- Displays **execution plans**
- **Logical plan**: shows a pipeline of operators to be executed to build the relation.
- **Physical plan**: shows how the logical operators are translated to backend-specific physical operators
- **Mapreduce plan**: shows how the physical operators are grouped into map reduce jobs.

```
grunt> explain path;
```

explain



Slide reproduced from “Pig Optimization and Execution” by Alan F. Gates
https://www.cs.duke.edu/courses/fall11/cps216/Lectures/gates.pdf

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 fits into memory, i.e. **reduce phase is unnecessary**

- More efficient to send the smaller data set 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` (large relation followed by small relation(s))

`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)

- 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 to one reducer)
     2. Skewed keys distributed over reducers

join implementations

skewed data

```
--users(name,city)
--city_info(city,population)
grunt> jnd = join city_info by city,
                  users by city using ‘skewed’
```

- **Example scenario**:
  - A data set contains
    - 20 users from Delft,
    - 100,000 users from New York, and,
    - 350 users from Amsterdam.
  - A reducer can deal with 75,000 records in memory.

- **Outcome**: 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 only take **two inputs**; multiway joins need to be broken up

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

join implementations

sorted data

- **Sort-merge join**: 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

THE END