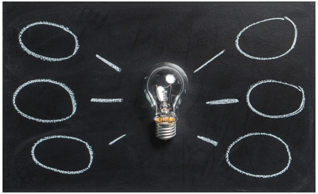
Julian Kunkel

Dataflow Computation



Outline

Learning Objectives

- Create a pipe diagram for pseudocode
- Illustrate the dataflow programming paradigm using examples
- Describe the concept of lazy evaluation
- Sketch a Pig Latin example program

General Data Model for Dataflow Languages

Data

- Tuple $t = (x_1, ..., x_n)$ where x_i may be of a given type
- Input/Output = list of tuples (like a table)

Typical Operators for Data-Flow Processing

- Operations process individual tuples
  - Map/Foreach: process or transform data of **individual tuples or group**
    - transform a tuple: student.Map((matrikel, name) ⇒ (matrikel + 4, name))
    - count members for each group: groupedStudents.Map((year) ⇒ count())
  - Filter tuples by comparing a key to a value
- Operations that require the complete input data
  - Group tuples by a key
  - Sort data according to a key
  - Join multiple relations together
  - Split tuples of a relation into multiple relations (based on a condition)

Data Flow Programming Paradigm [68]

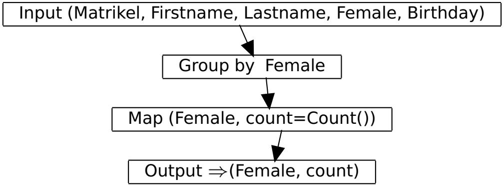
- Focus: data movement and transformation
  - Compare to imperative programming: sequence of commands
- Models program as directed graph of data flowing between operations
  - Input/output is illustrated as a node
  - Node is an operation, edges are dependencies
- Operation is run once all inputs become valid
  - An operation might work on a single data element or on the complete data
  - Parallelism is inherently supported by data flow languages
- States (in the program)
  - Dataflow works best with **stateless** programs
  - Stateful dataflow graphs support mutable states
  - Data related states, e.g., reductions, may be encoded as data
- Programming
  - Example: *read*("file.csv").*filter*("word" == "big data").*reduce*(count)
  - Functional declarative programming model is optimal

Pipe Diagrams[30]

- Goal: Visualize the processing pipeline of data-flows with a schema
  - Optional: Add examples to illustrate processing

Elements and diagram concepts

- Box: Operation
  - e.g., functions, filter, grouping, aggregating, mapping
  - Indicate also changes in schema
- Arrows show processing order (DAG), joins have two inputs



30 We will use a variant from [11]

Pipe Diagram with Examples

Matrikel	Firstname	Lastname	Female	Birthdate
22	"Fritz"	"Musterman M."	false	2000-01-01
23	"Nina"	"Musterfrau F."	true	2000-01-01
24	"Hans"	"Im Glück"	false	2001-01-01

Group by Female

Matrikel	Firstname	Lastname	Female	Birthdate
22	"Fritz"	"Musterman M."	false	2000-01-01
24	"Hans"	"Im Glück"	false	2001-01-01

23	"Nina"	"Musterfrau F."	true	2000-01-01

Map (Female, count=Count())

Female	count
false	2
true	1

Apache Pig [60, 61, 62]

- Pig: Infrastructure (language, compiler) for executing big data programs
  - No server (services) required
  - Data is stored on HDFS
  - Uses MapReduce or TEZ execution engine
- High-level scripting language Pig Latin
  - Describes processing as data flow
  - Compiler parallelizes data flow (into MapReduce / TEZ job)
- Batch mode and interactive shell (`pig`)

Data Model for Apache Pig [62]

- Tuple: An ordered set of named fields (our data)
  - A field can be a simple type or complex (tuple, bag or map)
  - Fields are referred by name or position ($0 to $n)
- Bag: Collection of tuples (evtl. with duplicates)
- Relation: Is a bag (like a table)
  - Data types of fields can be assigned with a schema
  - Not necessarily with a fixed schema
    - Each tuple may have different fields
    - Without defined type, data will be converted if necessary
  - Relations are referred to by name or alias (variable)

Example: Loading data with a schema

```
# table with student basic information
S = LOAD 'stud.csv' as (matrikel:int, semester:int, feminine:boolean, name:chararray, birthday:datetime);
```

stud.csv

```
4711  5 false "Max Mustermann"  2000-01-01
4712  4 true  "Nina Musterfrau F."  2000-01-01
```

Outline

Scripting Language Pig Latin [62]

- Data-flow oriented imperative programming language
  - Declare execution plan vs. SQL (declare results)
- Datatypes: basic types, tuples, bags and maps
- Statement: operator with a named relation as input and output
  - LOAD and STORE operations are exceptions
  - Relations are referred to by name or alias (variable)
- For computation, additional (arithmetic) operators are provided
  - They are applied to each tuple
- Preprocessor with parameter substitution and macros (functions)
- Lazy evaluation for interactive shell
  - Run commands only when output is requested by the user
- Note: Intermediate relations are stored on tmp files on HDFS

Relational Operators [62]

Input/Output

- DUMP: Output results on stdout
- LOAD/STORE: Input/output relations to/from HDFS

Subsetting tuples from relations

- DISTINCT: Removes duplicated tuples
- FILTER: Select tuples by a a condition
- SAMPLE: Select random tuples from the relation
- LIMIT: Limit the number of tuples
- SPLIT: Partition the relation into relations based on conditions
- UNION: Merge multiple relations

Relational Operators [62]

Rearrange tuples

- GROUP: Group the data based on the values
- COGROUP: Like group but involves multiple relations
- ORDER BY: Sort the relation based on fields
- RANK: To each tuple add the position in the relation (can also apply sort before ranking)

Relational Operators [62]

Data manipulation

- FOREACH: Transform tuples of an relation
  - Supports nesting for processing of collections
- JOIN: Join of multiple relations based on identical field keys
- CROSS: Cross product of two or more relations
- CUBE: Aggregates for all combinations of specified groups
  - For n dimensions, this creates 2^n aggregates
  - ROLLUP creates $n + 1$ aggregates based on the hierarchical order

Execution of external functions

- MAPREDUCE: Run MapReduce jobs inside pig
- STREAM: Send data to an external script
- DEFINE: Create user defined functions
- REGISTER: Register UDFs of a JAR

Non-relational Operators[62]

- Arithmetic: +,-,*,/,%, ?:, CASE
- Boolean: AND, OR, NOT, IN (for collections)
- Casting: Conversion between data types
- Comparison (includes regex support)
- Flatten: Convert tuple elements and bags into tuples
- Disambiguate: Specifies the relation field, e.g., RELATION::f

Functions

- Evaluation functions (reduction):
  - AVG, MIN, MAX, SUM, COUNT, COUNT_STAR (also counts NULL)
  - CONCAT: concatenation
  - TOKENIZE: split string and returns bag
- String, datetime handling
- Conversion of strings to types
- Math functions

Outline

Accessing and Manipulating Data with Pig

- The pig shell is convenient for interactive usage
  - Checks schema and certain language/programming errors
- Invoke code in other languages via user-defined functions (UDF)
- Pig Latin can be embedded into, e.g., Python, JavaScript, Java

Debugging [62]

- For testing, run in local mode (`pig -x local`)
- For performance analysis, some run statistics are provided
- Add file names to tuples (e.g., using `PigStorage(',', '-tagsource')`)
- Some operators (with shortcuts) are provided to help debugging

Useful operators for debugging

- ASSERT: Ensure a condition on data (or abort)
- DUMP (\d): output results on stdout
- DESCRIBE (\de): show the schema of a relation
- EXPLAIN (\e): view the execution plans for computation
- ILLUSTRATE (\i): step-by-step execution of statements

Pig Examples: Our Student/Lecture Example

stud.csv

```
22,"Fritz","Musterman M.",false,2000-01-01
23,"Nina","Musterfrau F.",true,2000-01-01
```

lecture.csv

```
1;"Big Data";{(22),(23)}
2;"Hochleistungsrechnen";{(22)}
```

Pig schema and data loading

```
s = LOAD 'stud.csv' USING PigStorage(',') AS (matrikel:int, name:chararray,  firstname:chararray,
    ↪ feminine:boolean, birthday:datetime);
l = LOAD 'lecture.csv' USING PigStorage(';') AS (id:int, name:chararray, students:bag{T: (matrikel:int)});
ASSERT s BY matrikel > 0, 'matrikel must be bigger than 1';
describe s;
-- s: {matrikel: int, name: chararray, firstname: chararray, feminine: boolean, birthday: datetime}
DUMP l; STORE l INTO 'result' USING PigStorage (';');
```

Pig Examples: Our Student/Lecture Example

Goal: Identify student names participating in the lecture

```
-- unroll the bag for a join
lflat = FOREACH l GENERATE id,name,FLATTEN(students) as matrikel;
spart = JOIN lflat by matrikel, s by matrikel;
describe spart;
-- spart: {lflat::id: int,lflat::name: chararray,lflat::matrikel: int,s::matrikel: int,s::name:
    ↪ chararray,s::firstname: chararray,s::feminine: boolean,s::birthday: datetime}
dump spart;
--(2,"Hochleistungsrechnen",22,22,"Fritz","Musterman M.",false, 2000-01-01T00:00:00.000+01:00)
--(1,"Big Data",22,22,"Fritz","Musterman M.",false,2000-01-01T00:00:00.000+01:00)
--(1,"Big Data",23,23,"Nina","Musterfrau F.",true,2000-01-01T00:00:00.000+01:00)
```

Pig Examples: Our Student/Lecture Example

Goal: Determine the number of students

```
t = GROUP s ALL; -- we generate only one group containing all tuples
c = FOREACH t GENERATE COUNT(s); -- we compute the count for each group
-- (2)
```

Goal: Determine the number of participants per lecture

```
c = FOREACH l GENERATE id,COUNT(students) AS participants;
-- (1,2)
-- (2,1)

-- alternatively on our flattened table:
z = GROUP spart BY id;
c = FOREACH z GENERATE group AS id, COUNT(p) AS participants;
```

Pig Examples: Our Student/Lecture Example

Goal: Identify female participants in lectures starting with “Big”

```
sf = FILTER s BY (feminine == true);
-- Filter the lectures
lf = FILTER l BY (name == 'Big.*');
-- Flatten the filtered lectures
lfflat = FOREACH lf GENERATE name,FLATTEN(students) as matrikel;

-- Now join them
fp = JOIN lfflat by matrikel, sf by matrikel;
-- ("Big Data",23,23,"Nina","Musterfrau F.",true, 2000-01-01T00:00:00.000+01:00)
-- only print the name
fpn = FOREACH fp GENERATE sf::name;
-- ("Nina")
```

Pig Examples: Our Student/Lecture Example

Goal: determine the average student age per lecture

```
sf = FOREACH s GENERATE name, birthday, matrikel;
spart = JOIN lflat by matrikel, sf by matrikel;
-- filter name of the lecture and birthday, we can also embed multiple operations here
f = FOREACH spart GENERATE lflat::name AS lecture, birthday;
-- group for the lecture name

z = GROUP f BY lecture;
-- ("Big Data",{("Big Data",2000-01-01T00:00:00.000+01:00),("Big Data", 2000-01-01T00:00:00.000+01:00)})
-- ("Hochleistungsrechnen",{("Hochleistungsrechnen", 2000-01-01T00:00:00.000+01:00)})

-- Now we iterate over the bag f that is the result of the grouping
alj = FOREACH z {
   tmp = FOREACH f GENERATE WeeksBetween(CurrentTime(), birthday);
   GENERATE group as lecture, AVG(tmp)/52 as avgAge, COUNT(tmp) as students;
}
-- ("Big Data",15.75,2)
-- ("Hochleistungsrechnen",15.75,1)
```

Pig Examples: Our Student/Lecture Example

Goal: for each student, identify the lectures s/he participates

```
sf = FOREACH s GENERATE name, matrikel;
lflat = FOREACH l GENERATE id,name,FLATTEN(students) as matrikel;
spart = JOIN lflat by matrikel, sf by matrikel;
z = GROUP spart BY sf::matrikel;
-- (22,{(1,"Big Data",22,"Fritz",22), (2,"Hochleistungsrechnen",22, "Fritz",22)})
-- (23,{(1,"Big Data",23,"Nina",23)})
al = FOREACH z {
   lectures = FOREACH spart GENERATE lflat::name;
   tmp = LIMIT spart 1;
   name = FOREACH tmp GENERATE sf::name;
   -- Apply flatten to remove the unneeded grouping of name
   GENERATE group as matrikel, FLATTEN(name), lectures;
}
-- (22,"Fritz",{("Big Data"),("Hochleistungsrechnen")})
-- (23,"Nina",{("Big Data")})
```

Preprocessor [67]

Parameter substitution

- Substitute variables in a script with Pig command line arguments
- Example: Use the matrikel as argument

```
-- in the pig script
%default MATRIKEL 23
s = FILTER students by matrikel = '$MATRIKEL'
-- on the command line:
pig -p MATRIKEL=4711 studentLecture.pig
```

Macros (Modularize the Pig scripts)

```
%declare searchMatrikel 23  -- define a constant

define studAttends (myMatrikel) returns attendedLectures {
  s = LOAD 'stud.csv' USING PigStorage(',') AS (matrikel:int, name:chararray,  firstname:chararray);
  l = LOAD 'lecture.csv' USING PigStorage(';') AS (id:int, name:chararray, students:bag{T: (matrikel:int)});
  i = FOREACH l {
    S = FILTER students BY (matrikel == $myMatrikel);
    GENERATE ( IsEmpty(S.$0) ? NULL: id ) AS lectureId;
  }
  $attendedLectures = FILTER i BY lectureId is not NULL;
}
dump studAttends($searchMatrikel);
-- Returns: (1)
```

Embedding Pig into Python [62]

```python
#!/usr/bin/python
# import the Pig class
from org.apache.pig.scripting import Pig

# Execution consists of three steps, compile, bind and run
# Compile returns a Pig object representing the data flow pipeline
# Variables can be used here and bind later
P = Pig.compile("""
  a = load '$in';
  store a into '$out';
  """)

input = 'stud.csv'
output = 'out.csv'

# bind variables and run the script, output is stored on HDFS
result = P.bind({'in':input, 'out':output}).runSingle()

if result.isSuccessful() : # Check if the job runs successful
    print 'Pig job succeeded'
else :
    raise 'Pig job failed'
```

To run the python script type `pig testpy.py`

Writing UDFs in Python [62]

Definition of the Python UDF

```python
import md5

@outputSchema("as:int")
def square(num):
    if num == None:
        return None
    return ((num) * (num))

@outputSchema("word:chararray")
def concat(word):
    return word + word

@outputSchema("anonym:chararray")
def anonymize(word):
    m = md5.new()
    m.update(str(word))
    return m.hexdigest()
```

Using the UDF in Pig

```
Register 'test.py' using jython as my;
-- Alternatively: streaming_python is another method, but code is different
b = FOREACH s GENERATE my.anonymize(matrikel),my.concat('test'),my.square(2);
-- (b6d767d2f8ed5d21a44b0e5886680cb9,testtest,4)
```

Outline

File Formats

- Support for Avro, CSV, RCFile, SequenceFile, JSONStorage, Binary
- Support for Hive's tables via HCatalog using the HCatLoader
- Internally BinStorage formats is used for intermediate files
- The schema can be part of the file to be loaded or explicitly given
- External schema can be written/read to/from `.pig-schema` file [65]

CSV (the default) via PigStorage class

- Supports compression bzip2, gzip, lzo
  - Automatically de/compressed if directory ends with .bz2/.gz

Examples

```
A = LOAD 'stud.gz' USING PigStorage('\t','-schema'); -- load the external schema
A = LOAD 'stud.gz' USING PigStorage('\t') AS (matrikel:int, ...);
A = LOAD 'stud.bin' USING BinStorage();
A = LOAD 'stud.json' USING JsonLoader();
A = LOAD 'data.txt' USING TextLoader(); -- load unstructured text as it is
A = LOAD 'stud.avro' USING AvroStorage (); -- contains elements, see [64]
```

Execution of Pig Queries on MapReduce and TEZ

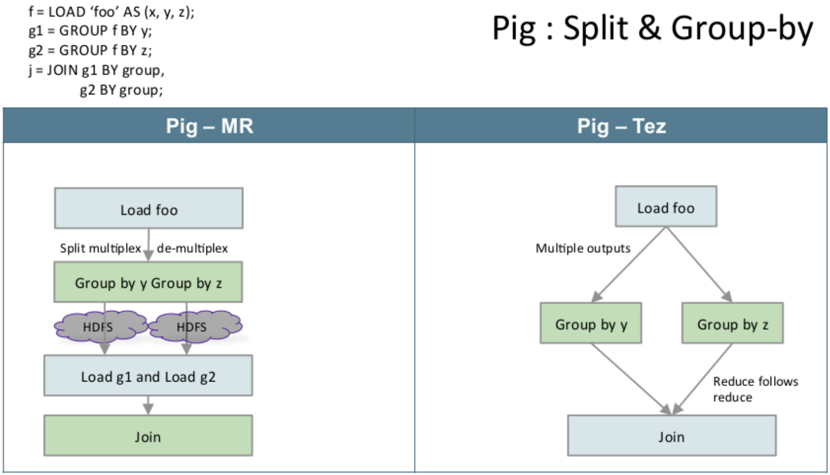


Figure: Source: H. Shah [20]

Performance Advises and Parallelism [62]

- Lazy evaluation applies several optimizations automatically
  - Rearrange work (run filters first) and merge operations if possible
  - Filter early in the pipeline
- Flexible number of reducers for the parallelism
  - By default a heuristics sets them based on the size of input data
  - The default number of reducers can be set

```
SET default_parallel 10;  -- 10 reducers
```

  - PARALLEL clause can be used to set reducers for an operator

```
O = GROUP input BY key PARALLEL 10;
```

- Use TEZ instead of MapReduce (start shell via `pig -x tez`)
- Use schemas for numeric data (otherwise floating point (double) is used)

Performance Advises and Parallelism [62]

- Choose the key for the Hadoop partitioner [66]
  - Maps keys to reducers
  - By default a HashPartitioner is used on the group

```
O = GROUP input BY key PARTITION BY org.apache.hadoop.mapred.lib.BinaryPartitioner;
```

- Intermediate relations can be compressed via properties:

```
SET pig.tmpfilecompression (true, false)
SET pig.tmpfilecompression.codec (gz, lzo)
```

- If you have many small input files: aggregate them before using Pig
- A cache is used (automatically) for storing JARs of user-defined functions

Optimization of Joins [62]

- Drop NULL keys before join
  - NULL keys are sent to a single reducer and may be overwhelming
- The last relation in a join operator is streamed by Pig
  - The largest relation should be listed last
- There are join strategies for optimization that have to be chosen [69]
  - **replicated** joins multiple small relations
  - **merge** joins relations already sorted by key
  - **merge-sparse** joins when the output is expected to be sparse
  - **skewed** distributes popular items across several reducers

Example

Assume input is small and input2 is a large relation

```
f = FILTER input BY $0 is not null;
f2 = FILTER input2 BY $0 is not null;
O = JOIN f BY $0, f2 BY $0 USING 'merge-sparse';
```

Summary

- Data flow programming paradigm is easy parallelizable
- Pipe diagrams visualize data flow programs
- Pig provides a data flow oriented programming infrastructure
  - Input/Output from/to HDFS
  - Utilizes MapReduce and Tez
  - No additional server(s) needed
- PigLatin is a domain-specific programming language
  - Only a few basic operations are necessary
  - FOREACH: Iteration over tuples and nested attributes
  - Beware: PigLatin details are complex; may indroduce complex errors
- Pig can be called from Python to script complex workflows
- User-defined functions can be integrated into PigLatin

Bibliography

11 Book: N. Marz, J. Warren. Big Data Principles and best practices of scalable real-time data systems.

60 https://pig.apache.org/

61 http://hortonworks.com/hadoop-tutorial/how-to-process-data-with-apache-pig/

62 http://pig.apache.org/docs/r0.15.0

63 https://www.qubole.com/resources/cheatsheet/pig-function-cheat-sheet/

64 https://cwiki.apache.org/confluence/display/PIG/AvroStorage

65 https://hadoopified.wordpress.com/2012/04/22/pigstorage-options-schema-and-source-tagging/

66 https://hadoop.apache.org/docs/r2.4.1/api/org/apache/hadoop/mapred/Partitioner.html

67 http://chimera.labs.oreilly.com/books/1234000001811/ch06.html

68 https://en.wikipedia.org/wiki/Dataflow_programming

69 https://vkalavri.wordpress.com/2012/04/10/join-types-in-pig/