Curso de extensão em Data Science

GERÊNCIA DE INFRAESTRUTURA PARA BIG DATA

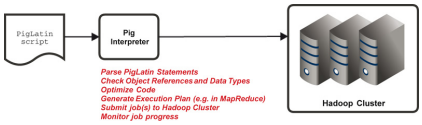
Prof. Tiago Ferreto – tiago.ferreto@pucrs.br

ESCOLA POLITÉCNICA

PIG

Pig Introduction

- Project initiated at Yahoo! in 2006 (became an ASF project in 2008)
- Goal: provide an alternative language interface to programming MapReduce using Java.
- Pig is abstracts the developer or analyst from the underlying MapReduce code
  - Provides an interpreted data flow language, which is converted to a series of map and reduce operations
- **Pig Latin** → dataflow scripting language used by Pig
- Interpreter (Grunt) receives PigLatin instructions, transforms it in MR jobs, submits to the cluster, monitors its progress, and returns the results to the console or saves it in HDFS



Parse PigLatin Statements
Check Object References and Data Types
Optimize Code
Generate Execution Plan (e.g. in MapReduce)
Submit job(s) to Hadoop Cluster
Monitor job progress

Grunt – The Pig Shell

- Interactive programming shell
- Uses lazy evaluation
  - Statements are parsed and interpreted, but execution only starts when output is requested
  - Enables optimization plans
- Can also be run in non-interactive or batch mode

Pig Latin Basics

- Data flow language
  - Not SQL or OO language
- A Pig Latin program usually uses a sequence of statements with a pattern
  1. Load data into a named dataset
  2. Create a new dataset by manipulating the input dataset
  3. Repeat step 2 n times (intermediate datasets are not necessarily physically stored)
  4. Output the final dataset to the console or output directory

Pig Data Structures

- Tuple
  - Ordered set of fields
  - Each field can be another data structure or atomic data (atom)
  - Example: ('Jeff', 47)
- Bag
  - Unordered collection of tuples
  - Example: {('Jeff', 47),('Paul', 44)}
- Map
  - Set of key value pairs
  - Example: [name#Jeff]

Object identifiers

- Used to identify data structures (bags, maps, tuples and fields)
- Rules
  - Case sensitive
  - Cannot match a Pig Latin language keyword
  - Must begin with a letter
  - Can only include letters, numbers, and underscores
- Example: settting a bag called ‘students’ with a specific schema with identifiers for each field
  - students = LOAD 'student' AS (name: chararray, age: int, gpa: float);

Pig Simple Datatypes (Atoms)

Type	Description	Example
`int`	32-bit signed integer	20
`long`	64-bit signed integer	20L
`float`	32-bit floating point	20.5F
`double`	64-bit floating point	20.5
`chararray`	UTF-8 string	‘Jeff’
`bytearray`	uninterpreted byte array	*<BLOB>*
`boolean`	True or False	true
`datetime`	datetime	1970-01-01T00:00:00.000+00:00
`biginteger`	Java BigInteger	200000000000
`bigdecimal`	Java BigDecimal	33.456783321323441233442

Pig Relational and Mathematical Operators

Operator	Notation
equal	`==`
not equal	`!=`
less than	`<`
greater than	`>`
less than or equal to	`<=`
greater than or equal to	`>=`
pattern matching	`matches`
Addition	`+`
Subtraction	`-`
Multiplication	`*`
Division	`/`
Modulo	`%`

Statements, Comments, and Conventions in Pig

- Statements are terminated by semicolon (;)
  - But can span multiple lines and include indentation
- Statements begin assigning a dataset to a bag
  - Loading data or manipulating a previously defined bag
  - Exception: when output is required (usually last line – using a DUMP or STORE statement)
- Keywords are capitalized by convention (e.g., LOAD, STORE, FOREACH), but are case insensitive
  - However, built-in functions (e.g., COUNT, SUM) are case sensitive
- Comments

```
-- Inline comment
/*
Block comment
*/
```


Loading Data into Pig

- Load functions → determines how a schema is extracted from data (similar to InputFormats)
  - Load function is defined through the USING clause
  - If USING is to used, data is assumed to be tab delimited text data
- Common Pig Load Functions
  - Other load functions include: HBaseStorage, AvroStorage, AccumuloStorage, OrcStorage, etc

Function	Syntax	Description
PigStorage (default)	`PigStorage([delim])`	Loads and stores data as structured text files
TextLoader	`TextLoader()`	Loads unstructured data in UTF-8 format
JsonLoader	`JsonLoader([schema])`	Loads JSON data

Example of LOAD statement

- Load statement for a new line terminated, comma delimited dataset

```
stations = LOAD 'stations' USING PigStorage(',') AS
           station_id,
           name,
           lat,
           long,
           dockcount,
           landmark,
           installation;
```

Pig Loading – schemas

```
--load data with no schema defined
stations = LOAD 'stations' USING PigStorage(',');
/* elements are accessed using their relative position, e.g. $0
all fields are typed as bytearray */

--load data with an untyped schema
stations = LOAD 'stations' USING PigStorage(',') AS
           (station_id,name,lat,long,dockcount,landmark,installation);
/* elements are accessed using their named identifier, e.g. name
all fields with an unassigned datatype are typed as bytearray */

--load data with a typed schema
stations = LOAD 'stations' USING PigStorage(',') AS
           (station_id:int, name:chararray, lat:float,
           long:float, dockcount:int, landmark:chararray,
           installation:chararray);
/* elements are accessed using their named identifier, e.g. name */
```


FILTER Statement

- Filters tuples from a bag
- Schema is not changed
  - Tuples matching the criteria keep in the bag, while others are dismissed
- Example

```
sj_stations = FILTER stations BY landmark == 'San Jose';
```


FOREACH Statement

- Equivalent to a map operation in a MR Job
- Each tuple in a given bag is iterated performing a requested operation
  - Examples of operations:
    - Remove a field or fields – project 3 fields from a bag with 4 fields
    - Adding a field or fields – add a computed field based on data in the bag
    - Transforming a field or fields – convert a char array to lowercase
    - Performing an aggregate function – COUNT of a field that is itself a bag
- Example
  - `station_ids_names = FOREACH stations GENERATE station_id, name;`

ORDER BY Statement

- Order tuples by a given field
- Example
  - `ordered = ORDER station_ids_names BY name;`

DESCRIBE Statement

- DESCRIBE - inspects the schema of a bag
  - Presents fields and types
  - It does not require or invoke execution
  - Example
    - `DESCRIBE stations;`

```
stations: {station_id: int,name: chararray,lat: float,long: float,
dockcount: int,landmark: chararray,installation: chararray}
```


ILLUSTRATE Statement

- ILLUSTRATE – besides presenting the schema, it also presents its predecessors and sample data (tuples)
  - Requires execution – takes longer to execute
  - Example
    - `ILLUSTRATE ordered;`

```
-------------------------------------------------------------------------------
| stations    | station_id:int  | name:chararray                          ...
-------------------------------------------------------------------------------
|             | 23              | San Mateo County Center
|             | 70              | San Francisco Caltrain (Townsend at
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
| station_ids_names   | station_id:int  | name:chararray                    ...
-------------------------------------------------------------------------------
|                     | 23              | San Mateo County Center
|                     | 70              | San Francisco Caltrain
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
| ordered     | station_id:int  | name:chararray                          ...
-------------------------------------------------------------------------------
|             | 70              | San Francisco Caltrain (Townsend at
|             | 23              | San Mateo County Center
-------------------------------------------------------------------------------
```

Built-in Functions

- Typically operate against a field and are used with the FOREACH operator
- Example
  - `lcase_stations = FOREACH stations GENERATE station_id, LOWER(name), lat, long, landmark;`
- Common Built-in Functions

```
--Eval Functions
AVG, COUNT, MAX, MIN, SIZE, SUM, TOKENIZE

--Math Functions
ABS, CEIL, EXP, FLOOR, LOG, RANDOM, ROUND

--String Functions
STARTSWITH, ENDSWITH, LOWER, UPPER, LTRIM, RTRIM, TRIM, REGEX_EXTRACT

--Datetime Functions
CurrentTime, DaysBetween, GetDay, GetHour, GetMinute, ToDate
```


GROUP Statement

- Enables grouping records by a specific field
- Results a structure with one record per unique value in the field being used to group
- Tuples in the structure have two fields
  - Group: same type of the "group field"
  - Field named after the relation being used to group by – bag of tuples from this relation that contain the element being grouped
- Usually used before performing an aggregate function (COUNT, SUM, AVG, etc)
- GROUP ALL Statement – groups all tuples into one structure
  - Can be counted using COUNT or COUNT_STAR (discards NULL values) or summed using SUM

```
--salespeople
--salespersonid, name, storeid
1,Henry,100
2,Karen,100
3,Paul,101
4,Jimmy,102
5,Janice,

--stores
--storeid, name
100,Hayward
101,Baumholder
102,Alexandria
103,Melbourne

--sales
--salespersonid, storeid, salesamt
1,100,38
1,100,84
2,100,26
2,100,75
3,101,55
3,101,46
4,102,12
4,102,67
```

```
grouped = GROUP sales BY salespersonid;
DESCRIBE grouped;
grouped: {group: int,
    sales: {(salespersonid: int,storeid: int,salesamt: int)}}
DUMP grouped;
(1,{(1,100,84),(1,100,38)})
(2,{(2,100,75),(2,100,26)})
(3,{(3,101,46),(3,101,55)})
(4,{(4,102,67),(4,102,12)})
salesbyid = FOREACH grouped GENERATE group AS salespersonid,
            SUM(sales.salesamt) AS total_sales;
DUMP salesbyid;
(1,122)
(2,101)
(3,101)
(4,79)
allsales = GROUP sales ALL;
DUMP allsales;
(all,{(4,102,67),(4,102,12),(3,101,46),(3,101,55),...})
--COUNT all tuples in the sales bag
sales_trans = FOREACH allsales GENERATE COUNT(sales);
DUMP sales_trans;
(8)
--SUM all salesamts in the sales bag
sales_total = FOREACH allsales GENERATE SUM(sales.salesamt);
DUMP sales_total;
(403)
```

<bag>.<field> notation = dereferencing operator

Nested FOREACH Statements

- Enables operating against the result of a GROUP operation
- Nest FOREACH iterates through every item in the nested bag
- Some restrictions apply
  - Only relational operators are allows within the nested operation (LIMIT, FILTER, ORDER)
  - GENERATE must be the last line in the nested code block

```
top_sales = FOREACH grouped {
            sorted = ORDER sales BY salesamt DESC;
            limited = LIMIT sorted 2;
            GENERATE group, limited.salesamt;};
DUMP top_sales;
(1,{(84),(38)})
(2,{(75),(26)})
(3,{(55),(46)})
(4,{(67),(12)})
```

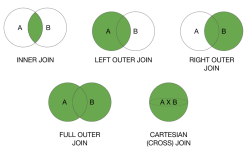

COGROUP Statement

- Enables grouping items from multiple bags
- Result contains one bag for each input bag to the COGROUP statement
- Example

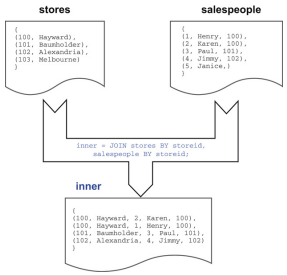
```
cogrouped = COGROUP stores BY storeid, salespeople BY storeid;
cogrouped: {group: int,stores: {(storeid: int,name: chararray)},
    salespeople: {(salespersonid: int,name: chararray,storeid: int)}}
DUMP cogrouped;
(100,{(100,Hayward)},{(2,Karen,100),(1,Henry,100)})
(101,{(101,Baumholder)},{(3,Paul,101)})
(102,{(102,Alexandria)},{(4,Jimmy,102)})
(103,{(103,Melbourne)},{})
(,{},{(5,Janice,)})
```


JOIN Statement

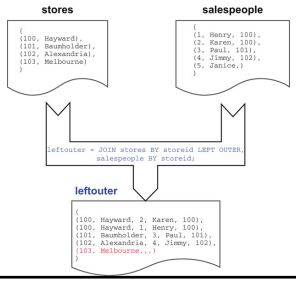
- Combine records from two bags based on a common field (join key)
- Join Types
  - Inner join (or simply join) – returns all records from both datasets where the key is present in both datasets
  - Outer join - does not require keys to match in both datasets. There are three different implementations
    - Left outer join – returns all records from the first dataset (left) along with matched records only from the right dataset (right)
    - Right outer join – returns all records from the second dataset (right) along with matched records only from the first dataset (left)
    - Full outer join – returns all records from both datasets whether there is a key match or not



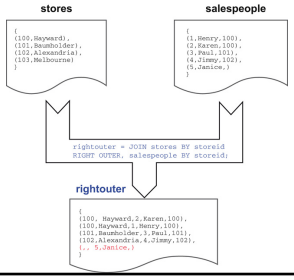
Inner Join in Pig



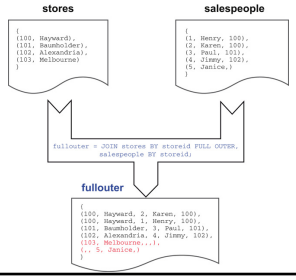
Left Outer Join in Pig



Right Outer Join in Pig



Full Outer Join in Pig

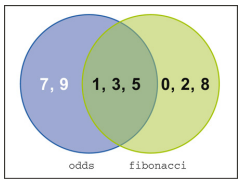


Recommendations

- 1st → Unlike in relational databases, there is no JOIN optimizations in Pig (no indexes or statistics)
  - The developer is responsible for manually optimizing the query
- Recommendation: "Join large by small." → reference the larger of the two input bags first followed by the smaller
- 2nd → Bags returned by GROUP, COGROUP, and JOIN operations will contain duplicate fields
  - Field in group, co-group or join will be duplicated in the resulting data structure
- Recommendation: "Filter early, filter often." → Following a COGROUP operation with a FOREACH operation to remove duplicate fields will assist the Pig optimizer

Set Operations

- Pig supports several common set operations: UNION, DISTINCT, SUBTRACT, CROSS, SPLIT
- Datasets using in the examples



UNION Statement

- Concatenates two datasets
- Union does not eliminate duplicates (requires DISTINCT) or preserve the order of the input bags (requires ORDER)

```
unioned = UNION odds, fibonacci;
DUMP unioned;
(0)
(1)
(2)
(3)
(5)
(8)
(1)
(3)
(5)
(7)
(9)
```


DISTINCT Statement

- Removes duplicate records or tuples from a bag

```
no_duplicates = DISTINCT unioned;
DUMP no_duplicates;
(0)
(1)
(2)
(3)
(5)
(7)
(8)
(9)
```


SUBTRACT Statement

- Returns tuples from one bag that are not present in another bag

```
cogrouped = COGROUP odds by $0, fibonacci by $0;
subtracted = FOREACH cogrouped GENERATE SUBTRACT(fibonacci, odds);
DUMP subtracted;
({(0)})
({(2)})
({(8)})
({})
({})
({})
({})
({})
```


CROSS Statement

- Produces a cross product of all tuples from one bag with all tuples from another bag
- **Caution**: Cross joins can create massive datasets!

```
crossed = CROSS odds, fibonacci;
DUMP crossed;
(9,8)
(9,5)
(9,3)
(9,2)
(9,1)
(9,0)
(7,8)
(7,5)
(7,3)
...
```


SPLIT Statement

- Splits one bag into multiple bags
- Resulting datasets do not have to be mutually exclusive

```
SPLIT unioned INTO evens IF ($0 % 2 == 0), odds IF ($0 % 2 != 0);
DUMP evens;
(0)
(2)
(8)
DUMP odds;
(1)
(3)
(5)
(1)
(3)
(5)
(7)
(9)
```


User-Defined Functions in Pig

- Pig can be extended through UDFs (User-defined functions)
- UDFs can be classified in
  - Load/Store functions
    - Pig equivalent to InputFormats, OutputFormats, and RecordReaders in MapReduce
    - Define a Custom input and output data sources and structures
  - Eval functions
    - Functions that can be used in Pig expressions and statements to return any simple or complex Pig Datatype
- UDFs can be written in multiple languages: Java, Python and Jython, JavaScript, Groovy, Jruby
  - JVM platforms are recommended

UDF Example

- Pig Eval UDF written in Jython to return a bag of n-grams

```
@outputSchema("n:bag(t:tuple(ngram:chararray))")
def return_ngrams(s,n):
 outBag = []
 if s is None: return None
 input_list = list(s)
 for i in range(len(input_list)-(n-1)):
  ngram_arr = input_list[i:i+n]
  if (all(isinstance(item, int) for item in ngram_arr)):
   ngram_str = "".join(chr(l).lower() for l in ngram_arr)
  else:
   return None
  outBag.append(ngram_str)
 return outBag
```

- Registering and Using a Pig UDF

```
REGISTER 'fuzzymatchingudf.py' USING jython as fuzzymatchingudf;
raw_data = LOAD 'test.data' USING PigStorage();
names_with_ngrams = FOREACH raw_data GENERATE $0,
                    fuzzymatchingudf.return_ngrams($0,3);
DUMP names_with_ngrams;
```


PiggyBank

- Community-contributed user-defined functions
  - https://cwiki.apache.org/confluence/display/PIG/PiggyBank

```
REGISTER 'lib/piggybank.jar';
DEFINE ISOToUnix
   org.apache.pig.piggybank.evaluation.datetime.convert.ISOToUnix();
...
toEpoch = FOREACH toISO GENERATE id,
          (long)ISOToUnix(ISOTime) as epoch:long;
...
```


Apache DataFu

- Library of Pig UDFs contributed by LinkedIn
- Includes statistical and set processing functions (not available natively or in PiggyBank)
- http://datafu.apache.org/

```
REGISTER 'datafu-1.2.0.jar';
DEFINE Quantile datafu.pig.stats.StreamingQuantile('5');
quantiles = FOREACH resp_time_only {
    sorted = ORDER resp_time_bag BY resp_time;
    GENERATE url_date, Quantile(sorted.resp_time) as quantile_bins;
    }
...
```


Stream Operator in Pig

- Similar to MapReduce Streaming API
- Enables writing functions in languages not supported for UDF development (e.g., Perl, BASH, etc)
- Records are read in as tab-delimited text on STDIN, and records are emitted as tab-delimited text using STDOUT

```
DEFINE MYFUNCTION `myfunction.pl` SHIP('myfunction.pl');
output = STREAM recs THROUGH MYFUNCTION AS (col1:chararray, col2:long);
...
```


Parametrizing Pig Programs

- Parameters improve code flexibility
- Pig implements parametrization by string substitution
  - Designated values are replaced by parameter values at runtime

```
...
longwords = FILTER words BY SIZE(word) > $WORDLENGTH;
...
```

```
$ bin/pig -p WORDLENGTH=10 wordcount.pig
```


Pig Macros

- Improve reusability of code
- Created using the DEFINE statement

```
define remove_stopwords (BAG_OF_IDS_AND_TERMS, STOPWORDS_TERMS_ONLY)
returns result {
/**** GET RID OF STOPWORDS HERE... ****/
stopwords = FOREACH $STOPWORDS_TERMS_ONLY GENERATE
              FLATTEN(TOKENIZE(stopword)) as stopword;
tokenized_stopwords_join = JOIN stopwords BY stopword RIGHT OUTER,
                           $BAG_OF_IDS_AND_TERMS by term PARALLEL 5;
meaningful_terms = FILTER tokenized_stopwords_join BY
                   (stopwords::stopword IS NULL);
$result = FOREACH meaningful_terms GENERATE
            $BAG_OF_IDS_AND_TERMS::id AS id,
            $BAG_OF_IDS_AND_TERMS::term AS term;
};
...
terms_final = remove_stopwords(tokens, known_stop_words);
...
```

Dryrun

- Pig option to substitute macros and parameters in the code without running it

```
$ bin/pig -p WORDLENGTH=10 -dryrun wordcount.pig
```

- Creates a file called <program>.substituted with all substitutions

IMPORT Statement

- Enables important a file with multiple macros in the program

```
IMPORT 'functions.pig';
```

PIG – HANDS-ON