

High-level Programming Languages

Apache Pig and Pig Latin

Pietro Michiardi

Eurecom

Apache Pig

Introduction

- **Collection and analysis of enormous datasets is at the heart of innovation in many organizations**
 - ▶ E.g.: web crawls, search logs, click streams
- **Manual inspection before batch processing**
 - ▶ Very often engineers look for exploitable trends in their data to drive the design of more sophisticated techniques
 - ▶ This is difficult to do in practice, given the sheer size of the datasets
- **The MapReduce model has its own limitations**
 - ▶ One input
 - ▶ Two-stage, two operators
 - ▶ Rigid data-flow

MapReduce limitations

- **Very often tricky workarounds are required¹**

- ▶ This is very often exemplified by the difficulty in performing `JOIN` operations

- **Custom code required even for basic operations**

- ▶ Projection and Filtering need to be “rewritten” for each job

→ Code is difficult to reuse and maintain

→ Semantics of the analysis task are obscured

→ Optimizations are difficult due to opacity of `Map` and `Reduce`

¹The term workaround should not only be intended as negative.

Use Cases

Rollup aggregates

- **Compute aggregates against user activity logs, web crawls, etc.**
 - ▶ Example: compute the frequency of search terms aggregated over days, weeks, month
 - ▶ Example: compute frequency of search terms aggregated over geographical location, based on IP addresses
- **Requirements**
 - ▶ Successive aggregations
 - ▶ Joins followed by aggregations
- **Pig vs. OLAP systems**
 - ▶ Datasets are too big
 - ▶ **Data curation** is too costly

Use Cases

Temporal Analysis

- **Study how search query distributions change over time**
 - ▶ Correlation of search queries from two distinct time periods (groups)
 - ▶ Custom processing of the queries in each correlation group
- **Pig supports operators that minimize memory footprint**
 - ▶ Instead, in a RDBMS such operations typically involve `JOINS` over very large datasets that do not fit in memory and thus become slow

Use Cases

Session Analysis

- **Study sequences of page views and clicks**
- **Example of typical aggregates**
 - ▶ Average length of user session
 - ▶ Number of links clicked by a user before leaving a website
 - ▶ Click pattern variations in time
- **Pig supports advanced data structures, and UDFs**

Pig Latin

- **Pig Latin, a high-level programming language initially developed at Yahoo!, now at HortonWorks**
 - ▶ Combines the best of both declarative and imperative worlds
 - ★ High-level declarative querying in the spirit of SQL
 - ★ Low-level, procedural programming á la MapReduce
- **Pig Latin features**
 - ▶ Multi-valued, nested data structures instead of flat tables
 - ▶ Powerful data transformations primitives, including joins
- **Pig Latin program**
 - ▶ Made up of a series of operations (or transformations)
 - ▶ Each operation is applied to input data and produce output data
 - A Pig Latin program describes a data flow

Example 1

Pig Latin premiere

- **Assume we have the following table:**

`urls: (url, category, pagerank)`

- **Where:**

- ▶ `url`: is the url of a web page
- ▶ `category`: corresponds to a pre-defined category for the web page
- ▶ `pagerank`: is the numerical value of the pagerank associated to a web page

→ *Find, for each sufficiently large category, the average page rank of high-pagerank urls in that category*

Example 1

SQL

```
SELECT category, AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > 106
```

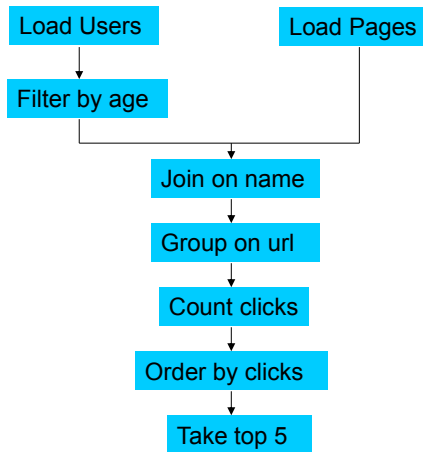
Example 1

Pig Latin

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls) > 106;
output = FOREACH big_groups GENERATE
category, AVG(good_urls.pagerank);
```

Example 2

- User data in one file, website data in another
- Find the top 5 most visited sites
- Group by users aged in the range (18,25)



Example 2: in MapReduce

[illegible]

- Hundreds lines of code; hours to write

Example 2: in Pig

```
Users = load 'users' as (name, age);  
Fltrd = filter Users by age >= 18 and age <= 25;  
Pages = load 'pages' as (user, url);  
Jnd = join Fltrd by name, Pages by user; Grpd = group Jnd by  
url;  
Smmd = foreach Grpd generate group, COUNT(Jnd) as clicks;  
Srted = order Smmd by clicks desc; Top5 = limit Srted 5;  
store Top5 into 'top5sites';
```

- Few lines of code; few minutes to write

Pig Execution environment

● How do we go from Pig Latin to MapReduce?

- ▶ The Pig system is in charge of this
- ▶ Complex execution environment that interacts with Hadoop MapReduce
- The programmer focuses on the data and analysis

● Pig Compiler

- ▶ Pig Latin operators are translated into MapReduce code
- ▶ **NOTE**: in some cases, hand-written MapReduce code performs better

● Pig Optimizer²

- ▶ Pig Latin data flows undergo an (automatic) optimization phase³
- ▶ These optimizations are borrowed from the RDBMS community

²Currently, rule-based optimization only.

³Optimizations can be selectively disabled.

Pig and Pig Latin

- **Pig is not a RDBMS!**

- ▶ This means it is not suitable for all data processing tasks

- **Designed for batch processing**

- ▶ Of course, since it compiles to MapReduce
- ▶ Of course, since data is materialized as files on HDFS

- **NOT designed for random access**

- ▶ Query selectivity does not match that of a RDBMS
- ▶ Full-scans oriented!

Comparison with RDBMS

- **It may seem that Pig Latin is similar to SQL**

- ▶ We'll see several examples, operators, etc. that resemble SQL statements

- **Data-flow vs. declarative programming language**

- ▶ Data-flow:
 - ★ Step-by-step set of operations
 - ★ Each operation is a **single transformation**
- ▶ Declarative:
 - ★ Set of constraints
 - ★ Applied together to an input to generate output

→ **With Pig Latin it's like working at the query planner**

Comparison with RDBMS

- **RDBMS store data in tables**

- ▶ Schema are predefined and strict
- ▶ Tables are flat

- **Pig and Pig Latin work on more complex data structures**

- ▶ Schema can be defined at run-time for readability
- ▶ *Pigs eat anything!*
- ▶ UDF and streaming together with nested data structures make Pig and Pig Latin more flexible

Dataflow Language

- **A Pig Latin program specifies a series of steps**

- ▶ Each step is a **single**, high level data transformation
- ▶ Stylistically different from SQL

- **With reference to Example 1**

- ▶ The programmer supply an order in which each operation will be done

- **Consider the following snippet**

```
spam_urls = FILTER urls BY isSpam(url);  
culprit_urls = FILTER spam_urls BY pagerank > 0.8;
```

Dataflow Language

- **Data flow optimizations**

- ▶ Explicit sequences of operations can be overridden
- ▶ Use of high-level, relational-algebra-style primitives (`GROUP`, `FILTER`,...) allows using traditional RDBMS optimization techniques

→ **NOTE: it is necessary to check whether such optimizations are beneficial or not, by hand**

- **Pig Latin allows Pig to perform optimizations that would otherwise be a tedious manual exercise if done at the MapReduce level**

Quick Start and Interoperability

- **Data I/O is greatly simplified in Pig**

- ▶ No need to curate, bulk import, parse, apply schema, create indexes that traditional RDBMS require
- ▶ Standard and ad-hoc “readers” and “writers” facilitate the task of ingesting and producing data in arbitrary formats

- **Pig can work with a wide range of other tools**

- **Why RDBMS have stringent requirements?**

- ▶ To enable transactional consistency guarantees
- ▶ To enable efficient point lookup (using physical indexes)
- ▶ To enable data curation on behalf of the user
- ▶ To enable other users figuring out what the data is, by studying the schema

Quick Start and Interoperability

● Why is Pig so flexible?

- ▶ Supports **read-only workloads**
- ▶ Supports **scan-only workloads** (no lookups)
- No need for transactions nor indexes

● Why data curation is not required?

- ▶ Very often, Pig is used for ad-hoc data analysis
- ▶ Work on temporary datasets, then throw them out!
- Curation is an overkill

● Schemas are optional

- ▶ Can apply one on the fly, at runtime
- ▶ Can refer to fields using positional notation
- ▶ E.g.: `good_urls = FILTER urls BY $2 > 0.2`

Nested Data Model

- **Easier for “programmers” to think of nested data structures**

- ▶ E.g.: capture information about positional occurrences of terms in a collection of documents
- ▶ `Map<documnetId, Set<positions> >`

- **Instead, RDBMS allows only fat tables**

- ▶ Only atomic fields as columns
- ▶ Require **normalization**
- ▶ From the example above: need to create two tables
- ▶ `term_info: (termId, termString, ...)`
- ▶ `position_info: (termId, documentId, position)`
- Occurrence information obtained by joining on `termId`, and grouping on `termId, documentId`

Nested Data Model

- **Fully nested data model (see also later in the presentation)**
 - ▶ Allows complex, non-atomic data types
 - ▶ E.g.: set, map, tuple
- **Advantages of a nested data model**
 - ▶ More natural than normalization
 - ▶ Data is often already stored in a nested fashion on disk
 - ★ E.g.: a web crawler outputs for each crawled url, the set of outlinks
 - ★ Separating this in normalized form imply use of joins, which is an overkill for web-scale data
 - ▶ Nested data allows to have an **algebraic language**
 - ★ E.g.: each tuple output by `GROUP` has one non-atomic field, a nested set of tuples from the same group
 - ▶ Nested data makes life easy when writing UDFs

User Defined Functions

- **Custom processing is often predominant**
 - ▶ E.g.: users may be interested in performing natural language stemming of a search term, or tagging urls as spam
- **All commands of Pig Latin can be customized**
 - ▶ Grouping, filtering, joining, per-tuple processing
- **UDFs support the nested data model**
 - ▶ Input and output can be non-atomic

Example 3

• Continues from Example 1

- ▶ Assume we want to find for each category, the top 10 urls according to pagerank

```
groups = GROUP urls BY category;  
output = FOREACH groups GENERATE category,  
top10(urls);
```

- `top10()` is a UDF that accepts a set of urls (for each group at a time)
- it outputs a set containing the top 10 urls by pagerank for that group
- final output contains non-atomic fields

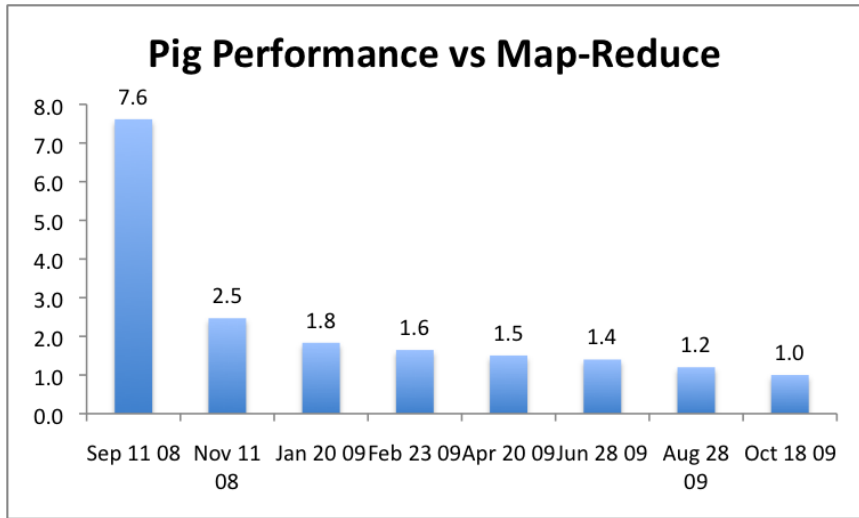
User Defined Functions

- **UDFs can be used in all Pig Latin constructs**
- **Instead, in SQL, there are restrictions**
 - ▶ Only scalar functions can be used in `SELECT` clauses
 - ▶ Only set-valued functions can appear in the `FROM` clause
 - ▶ Aggregation functions can only be applied to `GROUP BY` or `PARTITION BY`
- **UDFs can be written in Java, Python and Javascript**
 - ▶ With streaming, we can use also C/C++, Python, ...

Handling parallel execution

- **Pig and Pig Latin are geared towards parallel processing**
 - ▶ Of course, the underlying execution engine is MapReduce
 - ▶ SPORK = Pig on Spark → the execution engine need not be MapReduce
- **Pig Latin primitives are chosen such that they can be easily parallelized**
 - ▶ Non-equi joins, correlated sub-queries,... are not directly supported
- **Users may specify parallelization parameters at run time**
 - ▶ **Question:** Can you specify the number of maps?
 - ▶ **Question:** Can you specify the number of reducers?

A note on Performance



Pig Latin

Introduction

- **Not a complete reference to the Pig Latin language:** refer to [?]
 - ▶ Here we cover some interesting/useful aspects
- **The focus here is on some language primitives**
 - ▶ Optimizations are treated separately
 - ▶ How they can be implemented (in the underlying engine) is not covered
- **Examples are taken from [?, ?]**

Data Model

- **Supports four types**

- ▶ *Atom*: contains a simple atomic value as a string or a number, e.g. `'alice'`
- ▶ *Tuple*: sequence of *fields*, each can be of any data type, e.g., `('alice', 'lakers')`
- ▶ *Bag*: collection of tuples with possible duplicates. Flexible schema, no need to have the same number and type of fields
$$\left\{ \begin{array}{l} ('alice', 'lakers') \\ ('alice', ('iPod', 'apple')) \end{array} \right\}$$

The example shows that tuples can be nested

Data Model

• Supports four types

- ▶ *Map*: collection of data items, where each item has an associated key for lookup. The schema, as with bags, is flexible.
 - ★ **NOTE**: keys are required to be data atoms, for efficient lookup.

$$\left[\begin{array}{l} \text{'fan of'} \rightarrow \left\{ \begin{array}{l} (\text{'lakers'}) \\ (\text{'iPod'}) \end{array} \right\} \\ (\text{'age'} \rightarrow 20) \end{array} \right]$$

- ★ The key `'fan of'` is mapped to a bag containing two tuples
- ★ The key `'age'` is mapped to an atom
- ▶ Maps are useful to model datasets in which schema may be dynamic (over time)

Structure

- **Pig Latin programs are a sequence of steps**

- ▶ Can use an interactive shell (called `grunt`)
- ▶ Can feed them as a “script”

- **Comments**

- ▶ In line: with double hyphens (`--`)
- ▶ C-style for longer comments (`/* ... */`)

- **Reserved keywords**

- ▶ List of keywords that can't be used as identifiers
- ▶ Same old story as for any language

Statements

- **As a Pig Latin program is executed, each statement is *parsed***
 - ▶ The interpreter builds a **logical plan** for every relational operation
 - ▶ The logical plan of each statement is added to that of the program so far
 - ▶ Then the interpreter moves on to the next statement
- **IMPORTANT: No data processing takes place during construction of logical plan → Lazy Evaluation**
 - ▶ When the interpreter sees the first line of a program, it confirms that it is syntactically and semantically correct
 - ▶ Then it adds it to the logical plan
 - ▶ It does not even check the existence of files, for data load operations

Statements

- **It makes no sense to start any processing until the whole flow is defined**
 - ▶ Indeed, there are several optimizations that could make a program more efficient (e.g., by avoiding to operate on some data that later on is going to be filtered)
- **The trigger for Pig to start execution are the DUMP and STORE statements**
 - ▶ It is only at this point that the logical plan is **compiled** into a **physical plan**
- **How the physical plan is built**
 - ▶ Pig prepares a series of MapReduce jobs
 - ★ In *Local mode*, these are run locally on the JVM
 - ★ In *MapReduce mode*, the jobs are sent to the Hadoop Cluster
 - ▶ **IMPORTANT:** The command `EXPLAIN` can be used to show the MapReduce plan

Statements

Multi-query execution

- **There is a difference between DUMP and STORE**

- ▶ Apart from diagnosis, and interactive mode, in batch mode `STORE` allows for program/job optimizations

- **Main optimization objective: minimize I/O**

- ▶ Consider the following example:

```
A = LOAD 'input/pig/multiquery/A';  
B = FILTER A BY $1 == 'banana';  
C = FILTER A BY $1 != 'banana';  
STORE B INTO 'output/b';  
STORE C INTO 'output/c';
```

Statements

Multi-query execution

- **In the example, relations *B* and *C* are both derived from *A***
 - ▶ Naively, this means that at the first `STORE` operator the input should be read
 - ▶ Then, at the second `STORE` operator, the input should be read again
- **Pig will run this as a single MapReduce job**
 - ▶ Relation *A* is going to be read only once
 - ▶ Then, each relation *B* and *C* will be written to the output

Expressions

- **An expression is something that is evaluated to yield a value**
 - ▶ Lookup on [?] for documentation

$$t = \left(\text{'alice'}, \left\{ \begin{array}{l} (\text{'lakers'}, 1) \\ (\text{'iPod'}, 2) \end{array} \right\}, [\text{'age'} \rightarrow 20] \right)$$

Let fields of tuple t be called $f1$, $f2$, $f3$

Expression Type	Example	Value for t
Constant	'bob'	Independent of t
Field by position	$\$0$	'alice'
Field by name	$f3$	'age' \rightarrow 20
Projection	$f2.\$0$	$\left\{ \begin{array}{l} (\text{'lakers'}) \\ (\text{'iPod'}) \end{array} \right\}$
Map Lookup	$f3\#\text{'age'}$	20
Function Evaluation	$SUM(f2.\$1)$	$1 + 2 = 3$
Conditional Expression	$f3\#\text{'age'} > 18?$ 'adult': 'minor'	'adult'
Flattening	$FLATTEN(f2)$	'lakers', 1 'iPod', 2

Schemas

- **A relation in Pig may have an associated schema**

- ▶ This is optional
- ▶ A schema gives the fields in the relations names and types
- ▶ Use the command `DESCRIBE` to reveal the schema in use for a relation

- **Schema declaration is flexible but reuse is awkward⁴**

- ▶ A set of queries over the same input data will often have the same schema
- ▶ This is sometimes hard to maintain (unlike HIVE) as there is no external components to maintain this association

HINT:: You can write a UDF function to perform a personalized load operation which encapsulates the schema

⁴Current developments solve this problem: HCatalogs. We will not cover this in this course.

Validation and nulls

- **Pig does not have the same power to enforce constraints on schema at load time as a RDBMS**
 - ▶ If a value cannot be cast to a type declared in the schema, then it will be set to a `null` value
 - ▶ This also happens for corrupt files
- **A useful technique to partition input data to discern good and bad records**
 - ▶ Use the `SPLIT` operator
`SPLIT records INTO good_records IF temperature is not null, bad _records IF temperature is NULL;`

Other relevant information

- **Schema propagation and merging**

- ▶ How schema are propagated to new relations?
- ▶ Advanced, but important topic

- **User-Defined Functions**

- ▶ Use [?] for an introduction to designing UDFs

Data Processing Operators

Loading and storing data

- **The first step in a Pig Latin program is to load data**
 - ▶ Accounts for what input files are (e.g. csv files)
 - ▶ How the file contents are to be deserialized
 - ▶ An input file is assumed to contain a sequence of tuples

- Data loading is done with the `LOAD` command

```
queries = LOAD 'query_log.txt'  
USING myLoad()  
AS (userId, queryString, timestamp);
```

Data Processing Operators

Loading and storing data

- **The example above specifies the following:**

- ▶ The input file is `query_log.txt`
- ▶ The input file should be converted into tuples using the custom `myLoad` deserializer
- ▶ The loaded tuples have three fields, specified by the schema

- **Optional parts**

- ▶ `USING` clause is optional: if not specified, the input file is assumed to be plain text, tab-delimited
- ▶ `AS` clause is optional: if not specified, must refer to fields by position instead of by name

Data Processing Operators

Loading and storing data

- Return value of the `LOAD` command
 - ▶ Handle to a bag
 - ▶ This can be used by subsequent commands
 - bag handles are only logical
 - no file is actually read!
- The command to write output to disk is `STORE`
 - ▶ It has similar semantics to the `LOAD` command

Data Processing Operators

Per-tuple processing: Filtering data

- **Once you have some data loaded into a relation, a possible next step is, e.g., to filter it**
 - ▶ This is done, e.g., to remove unwanted data
 - ▶ **HINT:** By filtering early in the processing pipeline, you minimize the amount of data flowing through the system
- **A basic operation is to apply some processing over every tuple of a data set**
 - ▶ This is achieved with the `FOREACH` command

```
expanded_queries = FOREACH queries GENERATE
userId, expandQuery(queryString);
```

Data Processing Operators

Per-tuple processing: Filtering data

- **Comments on the example above:**

- ▶ Each tuple of the bag `queries` should be processed **independently**
- ▶ The second field of the output is the result of a UDF

- **Semantics of the `FOREACH` command**

- ▶ There can be no dependence between the processing of different input tuples
- This allows for an efficient parallel implementation

- **Semantics of the `GENERATE` clause**

- ▶ Followed by a list of expressions
- ▶ Also *flattering* is allowed
 - ★ This is done to eliminate nesting in data
 - Allows to make output data independent for further parallel processing
 - Useful to store data on disk

Data Processing Operators

Per-tuple processing: Discarding unwanted data

- **A common operation is to retain a portion of the input data**

- ▶ This is done with the `FILTER` command

```
real_queries = FILTER queries BY userId neq  
  'bot' ;
```

- **Filtering conditions involve a combination of expressions**

- ▶ Comparison operators
- ▶ Logical connectors
- ▶ UDF

Data Processing Operators

Per-tuple processing: Streaming data

- **The `STREAM` operator allows transforming data in a relation using an external program or script**
 - ▶ This is possible because Hadoop MapReduce supports “streaming”
 - ▶ Example:

```
C = STREAM A THROUGH 'cut -f 2';
```

which use the Unix `cut` command to extract the second field of each tuple in `A`
- **The `STREAM` operator uses `PigStorage` to serialize and deserialize relations to and from `stdin/stdout`**
 - ▶ Can also provide a custom serializer/deserializer
 - ▶ Works well with python

Data Processing Operators

Getting related data together

- **It is often necessary to *group* together tuples from one or more data sets**
 - ▶ We will explore several nuances of “grouping”
- **The first grouping operation we study is given by the **COGROUP** command**

Example: Assume we have loaded two relations from a Search Engine log file

`results: (queryString, url, position)`

`revenue: (queryString, adSlot, amount)`

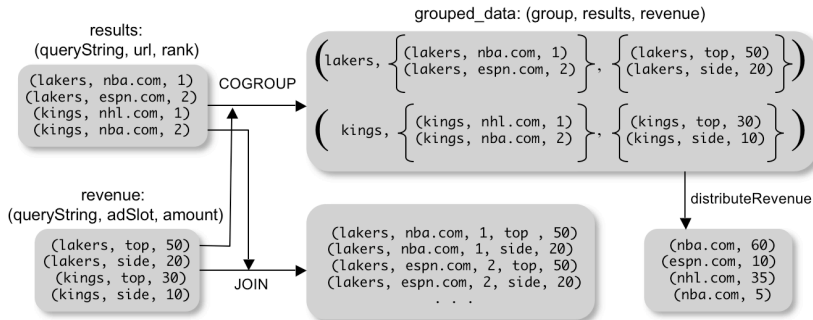
- ▶ `results` contains, for different query strings, the urls shown as search results, and the positions at which they were shown
- ▶ `revenue` contains, for different query strings, and different advertisement slots, the average amount of revenue

Data Processing Operators

Getting related data together

- Suppose we want to group together all search results data and revenue data for the same query string

grouped_data = COGROUP results BY queryString,
revenue BY queryString;



Data Processing Operators

The COGROUP command

- **Output of a COGROUP contains one tuple for each group**
 - ▶ First field (`group`) is the group identifier (the value of the `queryString`)
 - ▶ Each of the next fields is a bag, one for each group being co-grouped
- **Grouping can be performed according to UDFs**
- **Next: why COGROUP when you can use JOINS?**

Data Processing Operators

COGROUP VS JOIN

- JOIN VS. COGROUP

- ▶ Their are equivalent: JOIN == COGROUP followed by a cross product of the tuples in the nested bags

- **Example 3: Suppose we try to attribute search revenue to search-results urls → compute monetary worth of each url**

```
grouped_data = COGROUP results BY queryString,  
revenue BY queryString;  
url_revenues = FOREACH grouped_data GENERATE  
FLATTEN(distribteRevenue(results, revenue));
```

- ▶ Where `distribteRevenue` is a UDF that accepts search results and revenue information for each query string, and outputs a bag of urls and revenue attributed to them

Data Processing Operators

COGROUP VS JOIN

- **More details on the UDF distribute Revenue**

- ▶ Attributes revenue from the top slot entirely to the first search result
- ▶ The revenue from the side slot may be equally split among all results

- **Let's see how to do the same with a JOIN**

- ▶ JOIN the tables `results` and `revenues` by `queryString`
- ▶ GROUP BY `queryString`
- ▶ Apply a custom aggregation function

- **What happens behind the scenes**

- ▶ During the JOIN, the system computes the cross product of the search and revenue information
- ▶ Then the custom aggregation needs to undo this cross product, because the UDF specifically requires so

Data Processing Operators

COGROUP in details

- **The COGROUP statement conforms to an algebraic language**
 - ▶ The operator carries out only the operation of grouping together tuples into nested bags
 - ▶ The user can decide whether to apply a (custom) aggregation on those tuples or to cross-product them and obtain a JOIN
- **It is thanks to the nested data model that COGROUP is an independent operation**
 - ▶ Implementation details are tricky
 - ▶ Groups can be very large (and are redundant)

Data Processing Operators

A special case of COGROUP: the GROUP operator

- **Sometimes, we want to operate on a single dataset**

- ▶ This is when you use the GROUP operator

- **Let's continue from Example 3:**

- ▶ Assume we want to find the total revenue for each query string.
This writes as:

```
grouped_revenue = GROUP revenue BY queryString;  
query_revenue = FOREACH grouped_revenue GENERATE  
queryString, SUM(revenue.amount) AS totalRevenue;
```

- ▶ Note that `revenue.amount` refers to a projection of the nested bag in the tuples of `grouped_revenue`

Data Processing Operators

JOIN in Pig Latin

- In many cases, the typical operation on two or more datasets amounts to an equi-join
 - ▶ **IMPORTANT NOTE:** large datasets that are suitable to be analyzed with Pig (and MapReduce) are generally **not normalized**
 - JOINS are used more infrequently in Pig Latin than they are in SQL
- The syntax of a JOIN

```
join_result = JOIN results BY queryString,  
revenue BY queryString;
```

 - ▶ This is a classic inner join (actually an equi-join), where each match between the two relations corresponds to a row in the `join_result`

Data Processing Operators

JOIN in Pig Latin

- **JOINS lend themselves to optimization opportunities**
 - ▶ Active development of several join flavors is on-going
- **Assume we join two datasets, one of which is considerably smaller than the other**
 - ▶ For instance, suppose a dataset fits in memory
- **Fragment replicate join**
 - ▶ Syntax: append the clause `USING "replicated"` to a `JOIN` statement
 - ▶ Uses a distributed cache available in Hadoop
 - ▶ All mappers will have a copy of the small input
 - This is a Map-side join

Data Processing Operators

MapReduce in Pig Latin

- **It is trivial to express MapReduce programs in Pig Latin**

- ▶ This is achieved using `GROUP` and `FOREACH` statements
- ▶ A map function operates on one input tuple at a time and outputs a bag of key-value pairs
- ▶ The reduce function operates on all values for a key at a time to produce the final result

- **Example**

```
map_result = FOREACH input GENERATE  
FLATTEN(map(*));  
key_groups = GROUP map_results BY $0;  
output = FOREACH key_groups GENERATE reduce(*);
```

- ▶ where `map()` and `reduce()` are UDFs

The Pig Execution Engine

Pig Execution Engine

- **Pig Latin Programs are compiled into MapReduce jobs, and executed using Hadoop⁵**
- **Overview**
 - ▶ How to build a **logical plan** for a Pig Latin program
 - ▶ How to compile the logical plan into a **physical plan** of MapReduce jobs
- **Optimizations**

⁵Other execution engines are allowed, but require a lot of implementation effort.

Building a Logical Plan

- **As clients issue Pig Latin commands (interactive or batch mode)**
 - ▶ The Pig interpreter parses the commands
 - ▶ Then it verifies validity of input files and bags (variables)
 - ★ E.g.: if the command is `c = COGROUP a BY ..., b BY ...;`, it verifies if `a` and `b` have already been defined
- **Pig builds a **logical plan** for every bag**
 - ▶ When a new bag is defined by a command, the new logical plan is a combination of the plans for the input and that of the current command

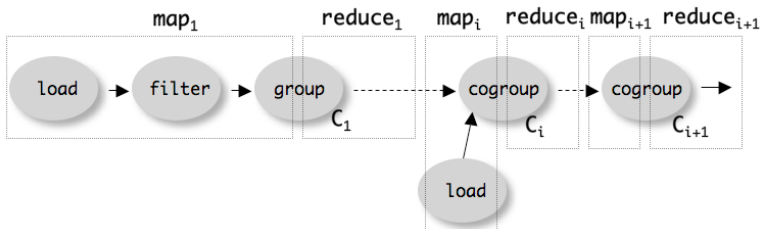
Building a Logical Plan

- **No processing is carried out when constructing the logical plans**
 - ▶ Processing is triggered only by `STORE` or `DUMP`
 - ▶ At that point, the logical plan is compiled to a physical plan
- **Lazy execution model**
 - ▶ Allows in-memory pipelining
 - ▶ File reordering
 - ▶ Various optimizations from the traditional RDBMS world
- **Pig is (potentially) platform independent**
 - ▶ Parsing and logical plan construction are platform oblivious
 - ▶ Only the compiler is specific to Hadoop

Building the Physical Plan

- **Compilation of a logical plan into a physical plan is “simple”**
 - ▶ MapReduce primitives allow a parallel `GROUP BY`
 - ★ Map assigns keys for grouping
 - ★ Reduce process a group at a time (actually in parallel)
- **How the compiler works**
 - ▶ Converts each `(CO) GROUP` command in the logical plan into **distinct** MapReduce jobs
 - ▶ *Map function* for `(CO) GROUP` command *C* initially assigns keys to tuples based on the `BY` clause(s) of *C*
 - ▶ *Reduce function* is initially a `no-op`

Building the Physical Plan



● MapReduce boundary is the COGROUP command

- ▶ The sequence of `FILTER` and `FOREACH` from the `LOAD` to the first `COGROUP` C_1 are pushed in the Map function
- ▶ The commands in later `COGROUP` commands C_i and C_{i+1} can be pushed into:
 - ★ the Reduce function of C_i
 - ★ the Map function of C_{i+1}

Building the Physical Plan

- **Pig optimization for the physical plan**

- ▶ Among the two options outlined above, the first is preferred
- ▶ Indeed, grouping is often followed by aggregation
- reduces the amount of data to be materialized between jobs

- **COGROUP command with more than one input dataset**

- ▶ Map function appends an extra field to each tuple to identify the dataset
- ▶ Reduce function decodes this information and inserts tuple in the appropriate nested bags for each group

Building the Physical Plan

- **How parallelism is achieved**

- ▶ For `LOAD` this is inherited by operating over HDFS
- ▶ For `FILTER` and `FOREACH`, this is automatic thanks to MapReduce framework
- ▶ For `(CO) GROUP` uses the `SHUFFLE` phase

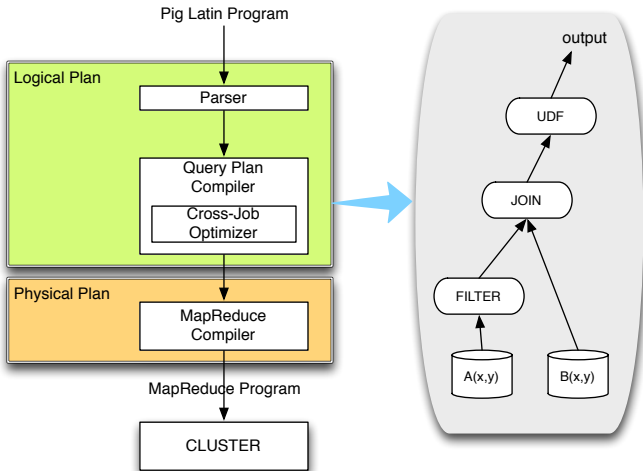
- **A note on the `ORDER` command**

- ▶ Translated in two MapReduce jobs
- ▶ First job: **Samples the input** to determine quantiles of the sort key
- ▶ Second job: Range partitions the input according to quantiles, followed by sorting in the reduce phase

- **Known overheads due to MapReduce inflexibility**

- ▶ Data materialization between jobs
- ▶ Multiple inputs are not supported well

Summary



Single-program Optimizations

- **Logical optimizations: query plan**

- ▶ Early projection
- ▶ Early filtering
- ▶ Operator rewrites

- **Physical optimization: execution plan**

- ▶ Mapping of logical operations to MapReduce
- ▶ Splitting logical operations in multiple physical ones
- ▶ Join execution strategies

Efficiency measures

- **(CO) GROUP command places tuples of the same group in nested bags**
 - ▶ Bag materialization (I/O) can be avoided
 - ▶ This is important also due to memory constraints
 - ▶ **Distributive** or **algebraic** aggregation facilitate this task
- **What is an algebraic function?**
 - ▶ Function that can be structured as a tree of sub-functions
 - ▶ Each leaf sub-function operates over a subset of the input data
 - If nodes in the tree achieve data reduction, then the system can reduce materialization
 - ▶ Examples: COUNT, SUM, MIN, MAX, AVERAGE, ...

Efficiency measures

- **Pig compiler uses the **combiner** function of Hadoop**
 - ▶ A special API for algebraic UDF is available
- **There are cases in which (CO) GROUP is inefficient**
 - ▶ This happens with non-algebraic functions
 - ▶ Nested bags can be spilled to disk
 - ▶ Pig provides a **disk-resident bag implementation**
 - ★ Features external sort algorithms
 - ★ Features duplicates elimination

References

References I