Apache Pig

Haridimos Kondylakis

CS562 Fall 2017
Motivation

- You’re a procedural programmer
- You have huge data
- You want to analyze it
Motivation

- As a procedural programmer...
- May find writing queries in SQL unnatural and too restrictive
- More comfortable with writing code; a series of statements as opposed to a long query. (Ex: MapReduce is so successful).
- Parallel database products expensive
- Rigid schemas
- Processing requires declarative SQL query construction
Motivation

- Map Reduce is very powerful, but:
  - It requires a Java programmer.
  - User has to re-invent common functionality (join, filter, etc.)
Pig

- Started at Yahoo! Research
- Runs over 70% of Yahoo!‘s jobs
- Features
  - Expresses sequences of MapReduce jobs
  - Data model: nested “bags” of items
  - Provides relational (SQL) operators (JOIN, GROUP BY, etc.)
  - Easy to plug in Java functions

† Sweet Spot: Take the best of both SQL and Map-Reduce; combine high-level declarative querying with low-level procedural programming…Pig Latin!
Big Picture

Pig Latin Script

User-Defined Functions

Pig

Compile

Optimize

Map-Reduce Statements

Write Results

Read Data

Hadoop

Hadoop MapReduce

HDFS
The Language
Pig Latin

- A script in Pig allows to define flows of data manipulation over datasets stored in HDFS
  - Sequence of statements
- High-level scripting language
  - Describes processing as data flow
  - Compiler parallelizes data flow
  - Uses MapReduce or TEZ execution engine
Data Model

- Tuple: an ordered set of named fields (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
```sql
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
```
Example Data Analysis Task
Find the top 10 most visited pages in each category

<table>
<thead>
<tr>
<th>User</th>
<th>Url</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>cnn.com</td>
<td>8:00</td>
</tr>
<tr>
<td>Amy</td>
<td>bbc.com</td>
<td>10:00</td>
</tr>
<tr>
<td>Amy</td>
<td>flickr.com</td>
<td>10:05</td>
</tr>
<tr>
<td>Fred</td>
<td>cnn.com</td>
<td>12:00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Url</th>
<th>Category</th>
<th>PageRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnn.com</td>
<td>News</td>
<td>0.9</td>
</tr>
<tr>
<td>bbc.com</td>
<td>News</td>
<td>0.8</td>
</tr>
<tr>
<td>flickr.com</td>
<td>Photos</td>
<td>0.7</td>
</tr>
<tr>
<td>espn.com</td>
<td>Sports</td>
<td>0.9</td>
</tr>
</tbody>
</table>
Data Flow

1. Load Visits
2. Group by url
3. Foreach url generate count
4. Load Url Info
5. Join on url
6. Group by category
7. Foreach category generate top10 urls
In MapReduce

```java
public static class Reader { // Reader is a single MapReduce InputSource. 
  public void configure(Context context) { 
    // Configure the Reader. 
  } 
} 

public static class LocalReader extends Reader { // Reader is a single MapReduce InputSource. 
  public void configure(Context context) { 
    // Configure the Reader. 
  } 
}
```

In Pig Latin

visits = load '/data/visits' as (user, url, time);
gVisits = group visits by url;
visitCounts = foreach gVisits generate url, count(visits);

urlInfo = load '/data/urlInfo' as (url, category, pRank);
visitCounts = join visitCounts by url, urlInfo by url;

gCategories = group visitCounts by category;
topUrls = foreach gCategories generate top(visitCounts,10);

store topUrls into '/data/topUrls';;
Built-in Functions and Operators

- General operators
  - Arithmetic, comparison, dereference, FLATTEN…
- Relational operators
  - GROUP, JOIN, COGROUP, CROSS, SPLIT…
- Eval functions (aggregations)
  - AVG, SUM, COUNT, CONCAT…
- Math functions
- String functions
- Bag and Tuple functions
  - TOBAG, TOP, TOTUPLE
User-Defined Functions

- Custom processing can be achieved by extending Pig with User-defined functions (UDFs)
- UDFs can be developed in different programming languages
  - Java, Jython, Python, JavaScript, Ruby and Groovy
- Functions written in Java have the most extensive support, allowing to customize all parts of the processing
Quick Start and Interoperability

visits = load ‘/data/visits’ as (user, url, time);
gVisits = group visits by url;
visitCounts = foreach gVisits generate url, count(urlVisits);

urlInfo = load ‘/data/urlInfo’ as (url, category, pRank);

visitCounts = join visitCounts by url, urlInfo by url;

gCategories = group visitCounts by category;
topUrls = foreach gCategories generate top(visitCounts, 10);

store topUrls into ‘/data/topUrls’;

Operates directly over files
Quick Start and Interoperability

visits = load ‘/data/visits’ as (user, url, time);
gVisits = group visits by url;
visitCounts = foreach gVisits generate url, count(urlVisits);

urlInfo = load ‘/data/urlInfo’ as (url, category, pRank);

visitCounts = join visitCounts by url, urlInfo by url;
gCategories = group visitCounts by category;
topUrls = foreach gCategories generate top(visitCounts, 10);

store topUrls into ‘/data/topUrls’;

Schemas optional; Can be assigned dynamically
User-Code as a First-Class Citizen

User-defined functions (UDFs) can be used in every construct

- Load, Store
- Group, Filter, Foreach

visits = load '/data/visits' as (user, url, time);
gVisits = group visits by url;
visitCounts = foreach gVisits generate url, count(urlVisits);
urlInfo = load '/data/urlInfo' as (url, category, pRank);
visitCounts = join visitCounts by url, urlInfo by url;
gCategories = group visitCounts by category;
topUrls = foreach gCategories generate top(visitCounts, 10);
store topUrls into '/data/topUrls';
Nested Data Model

- Pig Latin has a fully-nestable data model with:
  - Atomic values, tuples, bags (lists), and maps

- More natural to programmers than flat tuples
- Avoids expensive joins
Common case: aggregation on these nested sets

Power users: sophisticated UDFs, e.g., sequence analysis

Efficient Implementation
CoGroup

<table>
<thead>
<tr>
<th>query</th>
<th>url</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lakers</td>
<td>nba.com</td>
<td>1</td>
</tr>
<tr>
<td>Lakers</td>
<td>espn.com</td>
<td>2</td>
</tr>
<tr>
<td>Kings</td>
<td>nhl.com</td>
<td>1</td>
</tr>
<tr>
<td>Kings</td>
<td>nba.com</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>query</th>
<th>adSlot</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lakers</td>
<td>top</td>
<td>50</td>
</tr>
<tr>
<td>Lakers</td>
<td>side</td>
<td>20</td>
</tr>
<tr>
<td>Kings</td>
<td>top</td>
<td>30</td>
</tr>
<tr>
<td>Kings</td>
<td>side</td>
<td>10</td>
</tr>
</tbody>
</table>

Cross-product of the 2 bags would give natural join
The Compiler
Implementation

- ~70% of Hadoop jobs at Yahoo! are Pig
- 1000s of jobs per day
Compilation

Pig system does two tasks:

- Builds a Logical Plan from a Pig Latin script
  - Supports execution platform independence
  - No processing of data performed at this stage

- Compiles the Logical Plan to a Physical Plan and Executes
  - Convert the Logical Plan into a series of Map-Reduce statements to be executed (in this case) by Hadoop Map-Reduce
Compilation into Map-Reduce

Every group or join operation forms a map-reduce boundary

Other operations pipelined into map and reduce phases
Pig optimization principles

- vs RDBMS: There is absence of accurate models for data, operators and execution environments
- Use available reliable info. Trust user choice.
- Use rules that help in most cases
- Rules based on runtime information
Logical Optimizations

Restructure given logical dataflow graph
- Apply filter, project, limit early
- Merge foreach, filter statements
- Operator rewrites
Physical Optimizations

Physical plan: sequence of MR jobs having physical operators.

- Built-in rules. eg. use of combiner
- Specified in query - eg. join type
Hash Join

Users = load ‘users’ as (name, age);
Pages = load ‘pages’ as (user, url);
Jnd = join Users by name, Pages by user;

Map 1
Pages block n

Map 2
Users block m

Reducer 1
(1, user)
(1, fred)
(2, fred)
(2, fred)

Reducer 2
(1, name)
(2, jane)
(2, jane)
Skew Join

Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using 'skewed';
Users = \texttt{load 'users' as (name, age)};
Pages = \texttt{load 'pages' as (user, url)};
\texttt{Jnd = join Pages by user, Users by name using 'merge';}
Replicated Join

Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using 'replicated';
Group/cogroup optimizations

• On sorted and ‘collected’ data

\[
grp = \text{group Users by name using ‘collected’}.
\]

<table>
<thead>
<tr>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>aaron</td>
</tr>
<tr>
<td>aaron</td>
</tr>
<tr>
<td>barney</td>
</tr>
<tr>
<td>carol</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>zach</td>
</tr>
</tbody>
</table>

Map 1

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>aaron</td>
</tr>
<tr>
<td>aaron</td>
</tr>
<tr>
<td>barney</td>
</tr>
</tbody>
</table>

Map 2

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>carol</td>
</tr>
<tr>
<td>.</td>
</tr>
</tbody>
</table>
Multi-store script

A = load 'users' as (name, age, gender, city, state);
B = filter A by name is not null;
C1 = group B by age, gender;
D1 = foreach C1 generate group, COUNT(B);
store D into 'bydemo';
C2 = group B by state;
D2 = foreach C2 generate group, COUNT(B);
store D2 into 'bystate';
Multi-Store Map-Reduce Plan

map
  └── filter
      └── split
          ├── local rearrange
          └── local rearrange

reduce
  └── package
      └── package
          ├── foreach
          └── foreach
Memory Management

Use disk if large objects don’t fit into memory

- JVM limit > phy mem - Very poor performance
- Spill on memory threshold notification from JVM - unreliable
- Pre-set limit for large bags. Custom spill logic for different bags - eg distinct bag.
Other optimizations

- Aggressive use of combiner, secondary sort
- Lazy deserialization in loaders
- Better serialization format
- Faster regex lib, compiled pattern
- Carry data as byte arrays as far as possible
- Using binary comparator for sorting
- “Streaming” data through external executables
Tools (pig-pen & grunt)
Debugging

- How to verify the semantics of an analysis program
  - Run the program against whole data set. Might take hours!
  - Generate sample dataset
    - Empty result set may occur on few operations like join, filter
    - Generally, testing with sample dataset is difficult
  - Pig-Pen
    - Samples data from large dataset for Pig statements
    - Apply individual Pig-Latin commands against the dataset
    - In case of empty result, pig system resamples
    - Remove redundant samples
Debugging – Pig-Pen

visits = LOAD 'visits.txt' AS (user, url, time);

pages = LOAD 'pages.txt' AS (url, pagerank);

v_p = JOIN visits BY url, pages BY url;

users = GROUP v_p BY user;

useravg = FOREACH users GENERATE group, AVG(v_p.pagerank) AS avgpr;

answer = FILTER useravg BY avgpr > '0.5';
Debugging

- Pig-Latin command window and command generator
Debugging

- Sand Box Dataset (generated automatically!)

```sql
-- Operator usage
LOAD 'visits.txt' AS (user, url, time);
LOAD 'pages.txt' AS (url, pagerank);
JOIN visits BY url, pages BY url;
GROUP v_p BY user;
FOREACH users GENERATE group, AVG(v_p.pagerank) AS avgpr;
FILTER useravg > 0.5;

-- Query results
visits:
- (Amy, cnn.com, 8am)
- (Amy, frogs.com, 9am)
- (Fred, snails.com, 11am)

pages:
- (cnn.com, 0.8)
- (frogs.com, 0.8)
- (snails.com, 0.3)

v_p:
- (Amy, cnn.com, 8am, cnn.com, 0.8)
- (Amy, frogs.com, 9am, frogs.com, 0.8)
- (Fred, snails.com, 11am, snails.com, 0.3)

users:
- (Amy, { (Amy, cnn.com, 8am, cnn.com, 0.8),
  (Amy, frogs.com, 9am, frogs.com, 0.8) })
- (Fred, { (Fred, snails.com, 11am, snails.com, 0.3) })

useravg:
- (Amy, 0.8)
- (Fred, 0.3)

answer:
- (Amy, 0.8)
```
Debugging – Pig-Pen

- Provides sample data that is:
  - Real - taken from actual data
  - Concise - as small as possible
  - Complete - collectively illustrate the key semantics of each command
- Helps with schema definition
- Facilitates incremental program writing
Grunt

- Interactive shell for typing and executing PigLatin statements
- Each statement is interpreted as it is typed
- Execution is delayed until output is requested
- Useful for debugging and ad hoc data inspection

- **DESCRIBE**: reviews the schema of a bag
- **ILLUSTRATE**: displays the result of a step-by-step executions of statements using a tiny subset of data
- **EXPLAIN**: displays the execution plan
Some results from Yahoo – Comparison with MR in Java

1/20 the lines of code

What about Performance?

1/16 the development time
Pig performance

- Pigmix: pig vs mapreduce

![Graph showing Pigmix ratio over time from 1-Jun to 1-Oct. The ratio decreases from 1.6 to 0.8 over this period.]
Pig Compared to Map Reduce

- Faster development time
- Data flow versus programming logic
- Many standard data operations (e.g. join) included
- Manages all the details of connecting jobs and data flow
- Copes with Hadoop version change issues
And, You Don’t Lose Power

- UDFs can be used to load, evaluate, aggregate, and store data
- External binaries can be invoked
- Metadata is optional
- Flexible data model
- Nested data types
- Explicit data flow programming
Summary

- Big demand for parallel data processing
  - Emerging tools that do not look like SQL DBMS
  - Programmers like dataflow pipes over static files

- Hence the excitement about Map-Reduce

- But, Map-Reduce is too low-level and rigid

Pig Latin
Sweet spot between map-reduce and SQL
Choosing the Right Tool

- Choose the best solution for the given task – Mix and match as needed
- MapReduce
  - Lowlevel approach offers flexibility, control, and performance
  - More time-consuming and error-prone to write
  - Choose when control and performance are most important
- Pig and Hive
  - Faster to write, test, and deploy than MapReduce
  - Better choice for most analysis and processing tasks
Choosing the Right Tool

- Use Hive or Pig when...
  - You need support for custom file types, or complex data types
- Use Pig when...
  - You have developers experienced with writing scripts
  - You need complex processing flows
  - Your data is unstructured-semi/structured
- Use Hive When...
  - You have very complex long/running queries
Pig/HIVE vs. RDBMS

- Not intended to replace a RDBMS
- Relational databases are optimized for:
  - small to medium amounts of data
  - Immediate results
  - In/place modification of data
- Pig and Hive are optimized for:
  - Large amounts of read-only data
  - Extensive scalability at low cost
- Pig and Hive are better suited for batch processing
- RDBMSs are better for interactive use
References

- Cloudera, Introduction to PIG
- Utkarsh Srivastava, Pig: Building High-Level Dataflows over Map-Reduce
- Thejas Nair, How to make your map-reduce jobs perform as well as pig: Lessons from pig optimizations
- https://pig.apache.org/
- https://cwiki.apache.org/confluence/display/PIG/AvroStorage
- 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