Pig Latin: A Not-So-Foreign Language for Data Processing

Christopher Olsten, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins

Presented by Dan Welch
Motivation

- You’re a procedural programmer
- You have some 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.
Motivation

- The Data
  - Could be from multiple sources and in different formats
  - Data sets are typically huge
  - Don’t need to alter the original data; just need to do reads
  - May be very temporary; could discard the data set after analysis
Motivation

Data analysis goals

Quick
- Exploit parallel processing power of a distributed system

Easy
- Be able to write a program or query without a huge learning curve
- Have some common analysis tasks predefined

Flexible
- Transform a data set(s) into a workable structure without much overhead
- Perform customized processing

Transparent
- Have a say in how the data processing is executed on the system
Motivation

- Relational Distributed Databases
  - Parallel database products expensive
  - Rigid schemas
  - Data has to be imported into system-managed tables
  - Processing requires declarative SQL query construction

- Map-Reduce
  - Relies on custom code for even common operations
  - Need to do workarounds for tasks that have different data flows other than the expected Map → Combine → Reduce
Motivation

- Relational Distributed Databases

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

- Map-Reduce
Outline

- System Overview
- Pig Latin (The Language)
  - Data Structures
  - Commands
- Pig (The Compiler)
  - Logical & Physical Plans
  - Optimization
  - Efficiency
- Pig Pen (The Debugger)
- Conclusion
Big Picture

- Avro
- Chukwa
- Hbase (Bigtable)
- HDFS (GFS)
- Hive
- Map-Reduce
- Pig
- Zookeeper (Chubby)
Big Picture

Pig Latin Script

User-Defined Functions

Compile

Optimize

Map-Reduce Statements

Write Results

Read Data

hadoop

Map Reduce

HDFS
Data Model

- Atom – simple atomic value (i.e.: number or string)
- Tuple
- Bag
- Map

\[
\left( \text{'alice'}, \left\{ \left( \text{'lakers'}, 1 \right), \left( \text{'iPod'}, 2 \right) \right\}, \left[ \text{'age' \rightarrow 20} \right] \right)
\]
Data Model

- Atom
- Tuple – sequence of fields; each field any type
- Bag
- Map

\[

data = ('alice', \{('lakers', 1), ('iPod', 2)\}, ['age' \rightarrow 20])
\]
Data Model

- Atom
- Tuple
- Bag – collection of tuples
  - Duplicates possible
  - Tuples in a bag can have different field lengths and field types
- Map

\[
\left( \text{'alice'}, \left\{ \left( \text{'lakers'}, 1 \right) \right\}, \left[ \text{'age' \rightarrow 20} \right] \right)
\]
Data Model

- Atom
- Tuple
- Bag
- Map – collection of key-value pairs
  - Key is an atom; value can be any type

\[
\left(\text{'}alice\text{', }\left\{(\text{'}lakers\text{'}, 1)\right\}, \left\{(\text{'}iPod\text{'}, 2)\right\}, [\text{'}age\text{' }\rightarrow \text{'}20}\right)\]
Data Model

- Use of data structures
  - Increased flexibility in data representation

- Fully nested
  - More natural for procedural programmers (target user) than normalization
  - Data is often stored on disk in a nested fashion
  - Facilitates ease of writing user-defined functions

- No schema required
Data Model

- **User-Defined Functions (UDFs)**
  - Can be used in many Pig Latin statements
  - Useful for custom processing tasks
  - Can use non-atomic values for input and output
  - Currently must be written in Java
Speaking Pig Latin

- **LOAD**
  - Input is assumed to be a bag (sequence of tuples)
  - Can specify a serializer with ‘USING’
  - Can provide a schema with ‘AS’

```
newBag = LOAD 'filename'
    <USING functionName()>
    <AS (fieldName1, fieldName2,...)>;
```
FOREACH

Apply some processing to each tuple in a bag

Each field can be:

- A fieldname of the bag
- A constant
- A simple expression (ie: f1+f2)
- A predefined function (ie: SUM, AVG, COUNT, FLATTEN)
- A UDF (ie: sumTaxes(gst, pst) )

```java
newBag = FOREACH bagName
          GENERATE field1, field2, ...;
```
Speaking Pig Latin

- **FILTER**
  - Select a subset of the tuples in a bag
    
    \[
    newBag = \text{FILTER}\ bagName \\
    \quad \text{BY}\ expression; \\
    \]

  - Expression uses simple comparison operators (==, !=, <, >, …) and Logical connectors (AND, NOT, OR)
    
    \[
    \text{some_apples} = \text{FILTER}\ apples\ \text{BY}\ \text{colour}\ !=\ 'red'; \\
    \]

  - Can use UDFs
    
    \[
    \text{some_apples} = \text{FILTER}\ apples\ \text{BY}\ \text{NOT}\ \text{isRed}(\text{colour}); \\
    \]
Speaking Pig Latin

- COGROUP
  - Group two datasets together by a common attribute
  - Groups data into nested bags

```
grouped_data = COGROUP results BY queryString, revenue BY queryString;
```
Speaking Pig Latin

- Why COGROUP and not JOIN?

```
url_revenues =
FOREACH grouped_data GENERATE
FLATTEN(distributeRev(results, revenue));
```
Speaking Pig Latin

- Why COGROUP and not JOIN?
  - May want to process nested bags of tuples before taking the cross product.
  - Keeps to the goal of a single high-level data transformation per pig-latin statement.
- However, JOIN keyword is still available:

```pig
JOIN results BY queryString,
   revenue BY queryString;
```

Equivalent

```pig
temp = COGROUP results BY queryString,
   revenue BY queryString;
join_result = FOREACH temp GENERATE
   FLATTEN(results), FLATTEN(revenue);
```
Speaking Pig Latin

- **STORE (& DUMP)**
  - Output data to a file (or screen)

    ```
    STORE bagName INTO 'filename'
    <USING deserializer()＞;
    ```

- **Other Commands (incomplete)**
  - **UNION** – return the union of two or more bags
  - **CROSS** – take the cross product of two or more bags
  - **ORDER** – order tuples by a specified field(s)
  - **DISTINCT** – eliminate duplicate tuples in a bag
  - **LIMIT** – Limit results to a subset
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

Building a Logical Plan

- Verify input files and bags referred to are valid
- Create a logical plan for each bag defined
Building a Logical Plan Example

\[ A = \text{LOAD} \ 'user.dat' \ AS \ (name, \ age, \ city); \]
\[ B = \text{GROUP} \ A \ \text{BY} \ \text{city}; \]
\[ C = \text{FOREACH} \ B \ \text{GENERATE} \ \text{GROUP} \ \text{AS} \ \text{city}, \]
\[ \quad \text{COUNT}(A); \]
\[ D = \text{FILTER} \ C \ \text{BY} \ \text{city} \ \text{IS} \ 'kitchener' \]
\[ \quad \text{OR} \ \text{city} \ \text{IS} \ 'waterloo'; \]
\[ \text{STORE} \ D \ \text{INTO} \ 'local_user_count.dat'; \]
Compilation

Building a Logical Plan Example

A = LOAD 'user.dat' AS (name, age, city);
B = GROUP A BY city;
C = FOREACH B GENERATE group AS city,
    COUNT(A);
D = FILTER C BY city IS 'kitchener'
    OR city IS 'waterloo';
STORE D INTO 'local_user_count.dat';
### Building a Logical Plan Example

A = LOAD 'user.dat' AS (name, age, city);
B = GROUP A BY city;
C = FOREACH B GENERATE group AS city, COUNT(A);
D = FILTER C BY city IS 'kitchener' OR city IS 'waterloo';
STORE D INTO 'local_user_count.dat';
Compilation

- Building a Logical Plan Example

A = LOAD 'user.dat' AS (name, age, city);
B = GROUP A BY city;
C = FOREACH B GENERATE group AS city,
   COUNT(A);
D = FILTER C BY city IS 'kitchener'
   OR city IS 'waterloo';
STORE D INTO 'local_user_count.dat';
Building a Logical Plan Example

\[
\begin{align*}
A &= \text{LOAD} \ 'user.dat' \ \text{AS} \ (\text{name}, \ \text{age}, \ \text{city}); \\
B &= \text{GROUP} \ A \ \text{BY} \ \text{city}; \\
C &= \text{FOREACH} \ B \ \text{GENERATE} \ \text{group} \ \text{AS} \ \text{city}, \\
    &\quad \quad \text{COUNT}(A); \\
D &= \text{FILTER} \ C \ \text{BY} \ \text{city} \ \text{IS} \ 'kitchener' \ \\
    &\quad \quad \text{OR} \ \text{city} \ \text{IS} \ 'waterloo'; \\
&\quad \quad \text{STORE} \ D \ \text{INTO} \ 'local_user_count.dat';
\end{align*}
\]
Compilation

- **Other Optimization Techniques**

  - Push Down Explodes – Perform FLATTEN operations after JOIN where possible.
  
  - Push Limits Up – Perform LIMIT operations as soon as possible to avoid unnecessary processing of intermediate data.
  
  - And a few others having to do with splitting output, avoiding reloading data sets, and type-casting.
  
  - Also a “cookbook” available online for tips and tricks on how to structure Pig Latin commands for better performance.
Building a Physical Plan

A = LOAD 'user.dat' AS (name, age, city);
B = GROUP A BY city;
C = FOREACH B GENERATE group AS city, COUNT(A);
D = FILTER C BY city IS 'kitchener' OR city IS 'waterloo';
STORE D INTO 'local_user_count.dat';

Only happens when output is specified by STORE or DUMP
Compilation

- Building a Physical Plan
  - Step 1: Create a map-reduce job for each COGROUP
Building a Physical Plan

Step 1: Create a map-reduce job for each COGROUP

Step 2: Push other commands into the map and reduce functions where possible

May be the case certain commands require their own map-reduce job (ie: ORDER needs two map-reduce jobs)
Compilation

- Efficiency in Execution
  - Parallelism
    - Loading data - Files are loaded from HDFS
    - Statements are compiled into map-reduce jobs
Efficiency with Nested Bags

- In many cases, the nested bags created in each tuple of a COGROUP statement never need to physically materialize.

- Generally perform aggregation after a COGROUP and the statements for said aggregation are pushed into the reduce function.

- Applies to algebraic functions (ie: COUNT, MAX, MIN, SUM, AVG)
Compilation

- Efficiency with Nested Bags

```plaintext
['waterloo', ('Alice', 21, 'waterloo')
'kitchener', ('Charles', 36, 'kitchener')]

['waterloo', ('Bob', 18, 'waterloo')
'waterloo', ('Pete', 39, 'waterloo')]
```

Diagram:
- Load(user.dat)
- Map
- Filter
- Group
- Foreach
Compilation

- Efficiency with Nested Bags

```
[ 'waterloo', 1 ]
[ 'kitchener', 1 ]
[ 'waterloo', 2 ]
```

Load(user.dat) → Filter → Group → Foreach → Combine
Compilation

- Efficiency with Nested Bags

\[
\{ ('waterloo', 3) \}, \{ ('kitchener', 1) \}
\]

```
Load(user.dat)
Filter
Group
Foreach
Reduce
```
Compilation

- Efficiency with Nested Bags

  - Why this works:
    - COUNT is an algebraic function; it can be structured as a tree of sub-functions with each leaf working on a subset of the data

```
  SUM
 /   ________________\
| \                |
|   Combine        |
|                 |
|  COUNT          |
|                 |
|                  |
| Reduce          |
|                 |
|  COUNT          |
|                 |
|                 |
|  COUNT          |
```

40
Compilation

- **Efficiency with Nested Bags**
  - Pig provides an interface for writing algebraic UDFs so they can take advantage of this optimization as well.

- **Inefficiencies**
  - Non-algebraic aggregate functions (i.e., MEDIAN) need entire bag to materialize; may cause a very large bag to spill to disk if it doesn’t fit in memory
  
  - Every map-reduce job requires data be written and replicated to the HDFS (although this is offset by parallelism achieved)
Debugging

Pig-Pen

```sql
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';

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-Latin command window and command generator

```sql
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';
```

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

Sand Box Dataset (generated automatically!)

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;

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
Pig version 0.5.0

- More support for JOINs (outer, left, right)
- Ability to stream data through an external program
- Generally faster performance
- Ability to add types to schemas (ie: int, boolean, etc.)
- Open project so development is ongoing…
Pig is a data processing environment in Hadoop that is specifically targeted towards procedural programmers who perform large-scale data analysis.

Pig-Latin offers high-level data manipulation in a procedural style.

Pig-Pen is a debugging environment for Pig-Latin commands that generates samples from real data.
More Info


Anks- Thay!