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

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

SIGMOD ACM 08'

James Ball, Zeyu Chang, Ryan Ding, Hongzhen Liang, Hima Varshini Parasa

Georgia Tech

What is Pig Latin?

Apache Pig

- A data processing language developed at Yahoo!
- Has both procedural and declarative aspects
- Compiled by the accompanying Pig system
- Uses the mapReduce system Hadoop in its execution
- Comes with a rich debugging environment

A Quick Example Case

Task: Given a table with the name url, and fields (url, category, and pagerank), find the average pagerank of each category for all urls with a pagerank greater than 0.2, and where the category has greater than 100,000 pages.

SQL Implementation:

```
SELECT category, AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT (*) > 10^6
```

A Quick Example Case

Task: Given a table with the Name url, and fields (url, category, and pagerank), find the average pagerank each category for all urls with a pagerank greater than 0.2, and where the category has greater than 100,000 pages.

Pig Latin Implementation:

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls) > 10^6;
output = FOREACH big_groups GENERATE category, AVG(good_urls.pagerank);
```

Pig System Features and Motivations

Dataflow Language

- Sequence of high-level data transformations
  - "Easier to work with"

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls)>10^6;
output = FOREACH big_groups GENERATE
            category, AVG(good_urls.pagerank);
```

- Order of execution is not necessarily fixed
  - System can optimize in most cases

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

Quick Start and Interoperability

- Support for Ad-Hoc Analysis
  - Can directly run Pig queries on data
    - No need for lengthy import processes
    - With function to parse the data into tuples
    - Schema is not necessary
  - Can format output of Pig queries
    - Ex. Convert the tuple output into bytes
  - Easy to integrate with other applications

```
good_urls = FILTER urls BY $2 > 0.2;
```

- Workload
  - Read Only
  - Scans (not much indexing, etc)
  - Data is discarded on normal basis

Nested Data Model

- Offers Nested Data Structures
  - As opposed to 1NF (atomic column values)
- Closer to how programmers conceptualize
- Data is often stored in a nested format on disks (read only)
  - Saves compute to break down into 1NF, recombine
- Better fits design paradigm
  - One data transformation per step
- Better support for complex user defined functions.

```
term_info:   (termId, termString, ...)
position_info:   (termId, documentId, position)
```

```
Map<documentId, Set<positions>>
```

UDFs as First-Class Citizens

- Lots of data analytics involves custom processing
  - Spam detection, Search analysis, etc.

- Extensive Support for User Defined Functions
  - Can customize the functionality of all processing steps
    - Grouping, filtering, joining, and per-tuple
    - One type of UDF which covers everything
      - SQL: Scalar for SELECT, Aggregation needs GROUP BY
    - Input and output can be non-scalar (nested)

Ryan Ding

UDFs as First-Class Citizens

- Consider the original example from the Introduction
  - Task: Given a table with the Name url, and fields (url, category, and pagerank), find the average pagerank each category for all urls with a pagerank greater than 0.2, and where the category has greater than 100,000 pages.

- Say we only want to average the top 10 urls for each category (based on pagerank)…

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

- top10(urls) takes a set of urls and returns a set of 10 urls
  - Implemented in Java (initially)

UDFs as First-Class Citizens

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls) > 10^6;
top_urls = FOREACH big_groups GENERATE category, top10(good_urls)
output = FOREACH big_groups GENERATE category, AVG(top_urls.pagerank);
```

Parallelism Required

- Lots of data
  - Parallelism is essentially required

- Has only small set of primitives that can be parallelized
  - LOAD, FOREACH, etc. (expanded in later sections)

- Does not natively support non-equal joins, correlated subqueries
  - Can still be manually implemented through UDFs, but has very limited performance

Ryan Ding

Debugging Environment

- Can take a long time to process queries
  - Has "novel" debugger
  - Prints out Example table after each step which shows structure of output
    - Expanded in future sections

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)

Pig Latin Language

Data Model

- 4 types of data
  - Atom: single atomic value;
  - Tuple: sequence of fields, each can be a different type;
  - Bag: A collection of tuples, can have duplicates, and each tuple can be a different structure;
  - Map: A collection of key -> value mappings. Key is required to be atomic.

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

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' → 20]
Projection	f2.$0	{ ('lakers') ('iPod') }
Map Lookup	f3#'age'	20
Function Evaluation	SUM(f2.$1)	1 + 2 = 3
Conditional Expression	f3#'age'>18? 'adult':'minor'	'adult'
Flattening	FLATTEN(f2)	'lakers', 1 'iPod', 2

Table 1: Expressions in Pig Latin.

A tuple with an atom, a bag, and a map.

Zeyu Chang

Command: LOAD

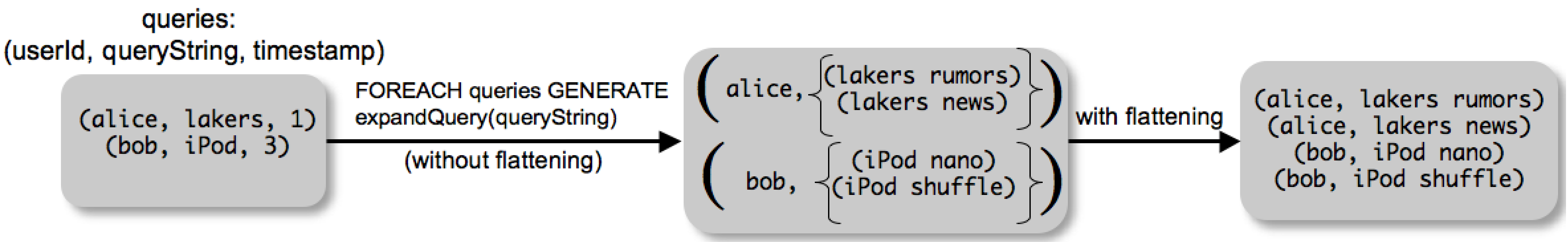
- Input:
  - The data file, e.g. "query_log.txt"
  - Deserialize function, e.g. "myLoad"
  - Tuple configuration

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

- Output:
  - A *handle* variable to the LOAD command

Command: FOREACH

- Apply processing to every tuple of a data set
- This could lead to nesting in the processed data
- "FLATTEN" is used to eliminate this nesting



Zeyu Chang

Command: FILTER

- Filters away some of the data base on some condition
- Condition can be arbitrary:
  - Comparison operators: ==, !=, eq, neq, >=, etc.
  - Logical operators: AND, OR, NOT
  - User defined functions

- The following two achieves the same objective:
  - `real_queries = FILTER queries BY userId neq 'bot';`
  - `real_queries = FILTER queries BY NOT isBot(userId);`

Command: COGROUP vs JOIN

- To group tuples from one or more data sets via some relation
- COGROUP: Generally, outputs a tuple for each group
- GROUP: A special case of COGROUP, where group is perform on only one data set
- JOIN: Equivalent to performing a cross product after executing COGROUP

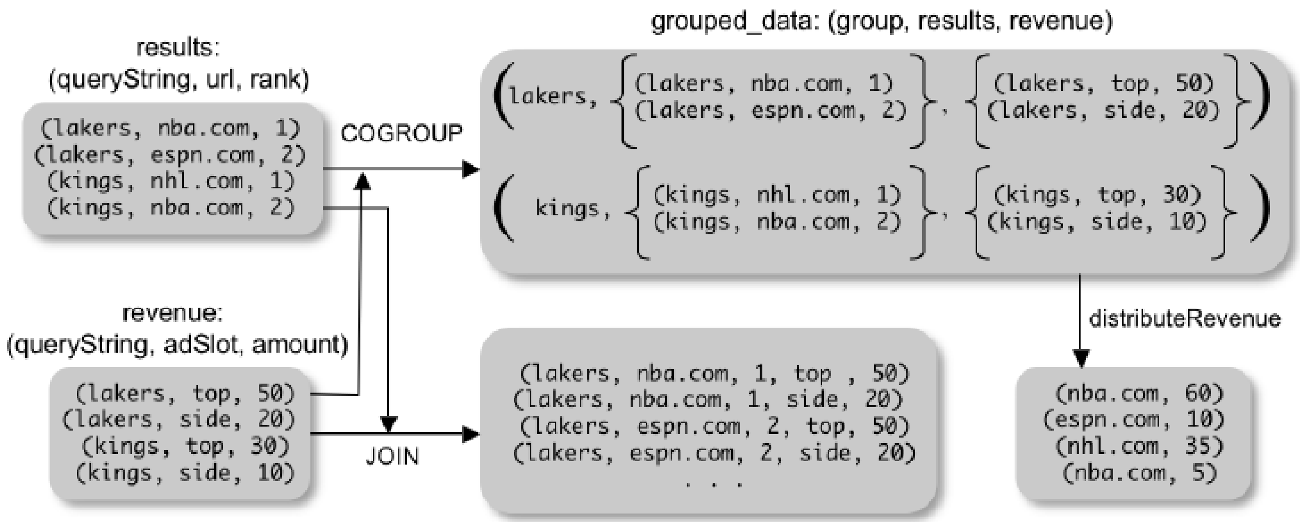


Figure 2: COGROUP versus JOIN.

Zeyu Chang

Other commands

- Similar commands to SQL
  - UNION: the union of two bags
  - CROSS: the cross product of two bags
  - ORDER (BY): order a bag by specific fields
  - DISTINCT: eliminate duplicates from the bag

- `ordered_result = ORDER query_revenues BY totalRevenue;`

Zeyu Chang

Nested operations

- Pig Latin allows some commands to be nested within a FOREACH command

```
grouped_revenue = GROUP revenue BY queryString;
query_revenues = FOREACH grouped_revenue{
                     top_slot = FILTER revenue BY
                                adSlot eq 'top';
                     GENERATE queryString,
                              SUM(top_slot.amount),
                              SUM(revenue.amount);
                 };
```

Command: STORE

- Store a Pig Latin expression sequence into a file

- `STORE query_revenues INTO 'myoutput' USING myStore();`
    - Serialize query_revenues using a custom serializer myStore
    - The serialized result is stored to "myoutput"

Pig Latin Implementation

IMPLEMENTATION

- **Pig System**: An open-source platform (Apache project) that implements Pig Latin.
- **Hadoop Integration**: Leverages Hadoop's scalable **MapReduce** framework.
- **Compilation & Execution Process:**
  - **Pig Latin scripts** are compiled into **logical plan**.
  - The **logical plans** are further compiled into **MapReduce jobs**.
  - Compiled **MapReduce jobs** are executed on a **Hadoop cluster**.

Building a Logical Plan

- **Pig Interpreter**:
  - **Parses** Pig Latin commands as they are issued by the user.
  - Handles **syntax checking** and provides error messages for issues.
  - **Validates** that **variables** (bags, relations) used are previously defined.
  - Builds a **logical plan for every bag.**
- **Logical Plans for Bags**:
  - **Definition**: An abstract representation of the data flow to produce bags.
  - **Example**:
    - Command: `c = COGROUP a BY queryString, b BY queryString`
    - Logical plan for `c` includes a cogroup operation using logical plans of `a` and `b`.
- **Lazy Execution**: Execution is deferred until a STORE command is invoked.
- **Platform Independence**: Logical plan construction is independent of the execution platform.

Hongzhen

Georgia Tech

Map-Reduce Plan Compilation

- Each COGROUP operation becomes its own MapReduce job.
- **Map Function**: Assigns keys based on the BY clause(s).
- **Reduce Function**: Forms grouped tuples and **creates nested bags** for cogroup data.

```
grouped_data  =  COGROUP results BY queryString,
                         revenue BY queryString;
```

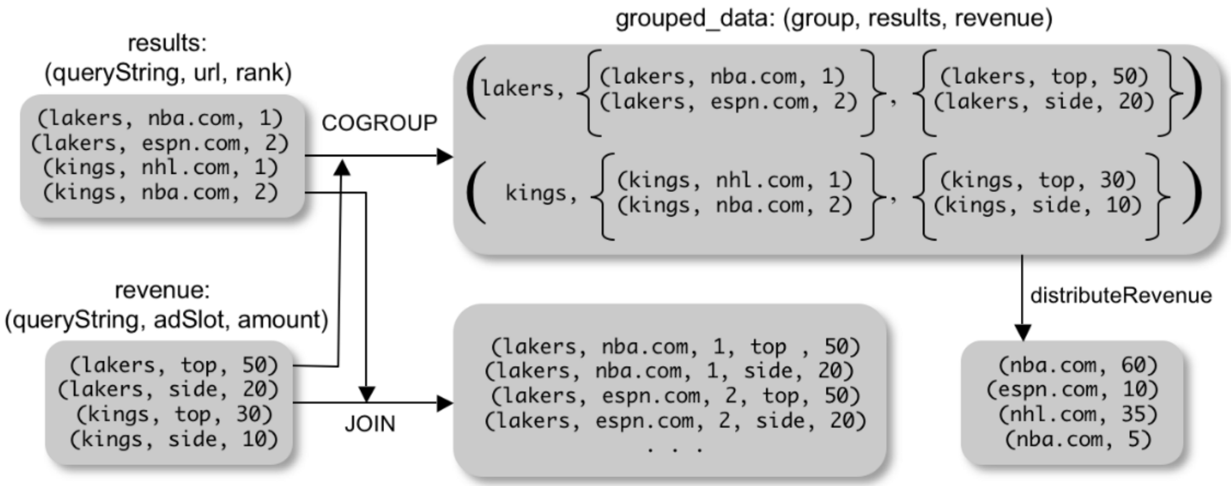


Figure 2: COGROUP versus JOIN.

Map-Reduce Plan Compilation

- **Operation Placement**:
  - **`FILTER`** and **`FOREACH`** commands from **`LOAD`** to the first `COGROUP` are incorporated into the **map** function.
  - Operations between subsequent `COGROUP` are pushed into the **reduce** function of the **preceding** `COGROUP`.
- **Limitations:**
  - **Intermediate data** must be **materialized** between jobs.
  - **Rigid** MapReduce model may not efficiently handle all Pig Latin operations.

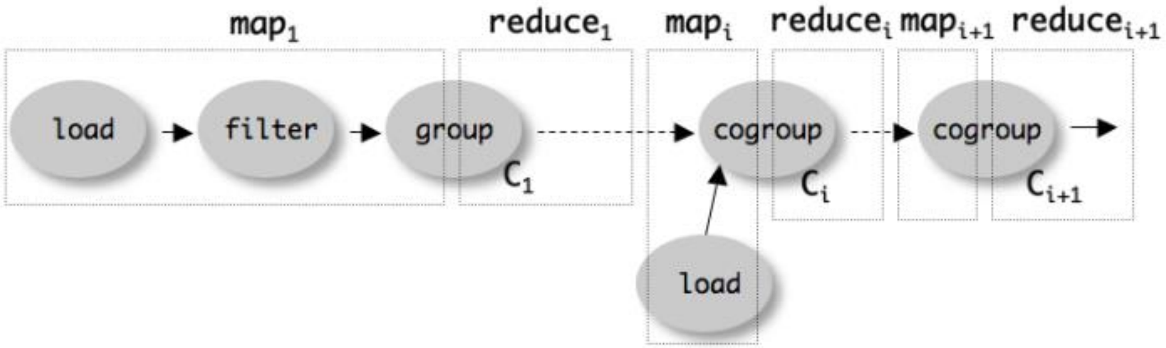


Figure 3: Map-reduce compilation of Pig Latin.

Efficiency With Nested Bags

- **Avoiding Materialization of Large Nested Bags**:
  - **Algebraic Functions for Aggregation**:
    - Functions **structured as a tree of subfunctions** operating on data subsets (e.g., `COUNT`, `SUM`, `MIN`, `MAX`, `AVERAGE`, `VARIANCE`).
    - Pig utilizes Hadoop's **combiner** feature for efficient aggregation.
  - **Handling Non-Algebraic Functions**:
    - Functions like **`MEDIAN` or custom UDFs** that are not algebraic require full materialization.
    - Pig spills large nested bags to disk when they can't fit in memory.

Other Features and Considerations

Debugging Environment – Pig Pen. But, why?

- Construction of a program in Pig Latin is repetitive, inefficient for large-scale data
- Traditional Debugging method: Sampling – Create a smaller subset of original data
- Limitations: Difficult to find sample data to test semantics of the program
- Example:

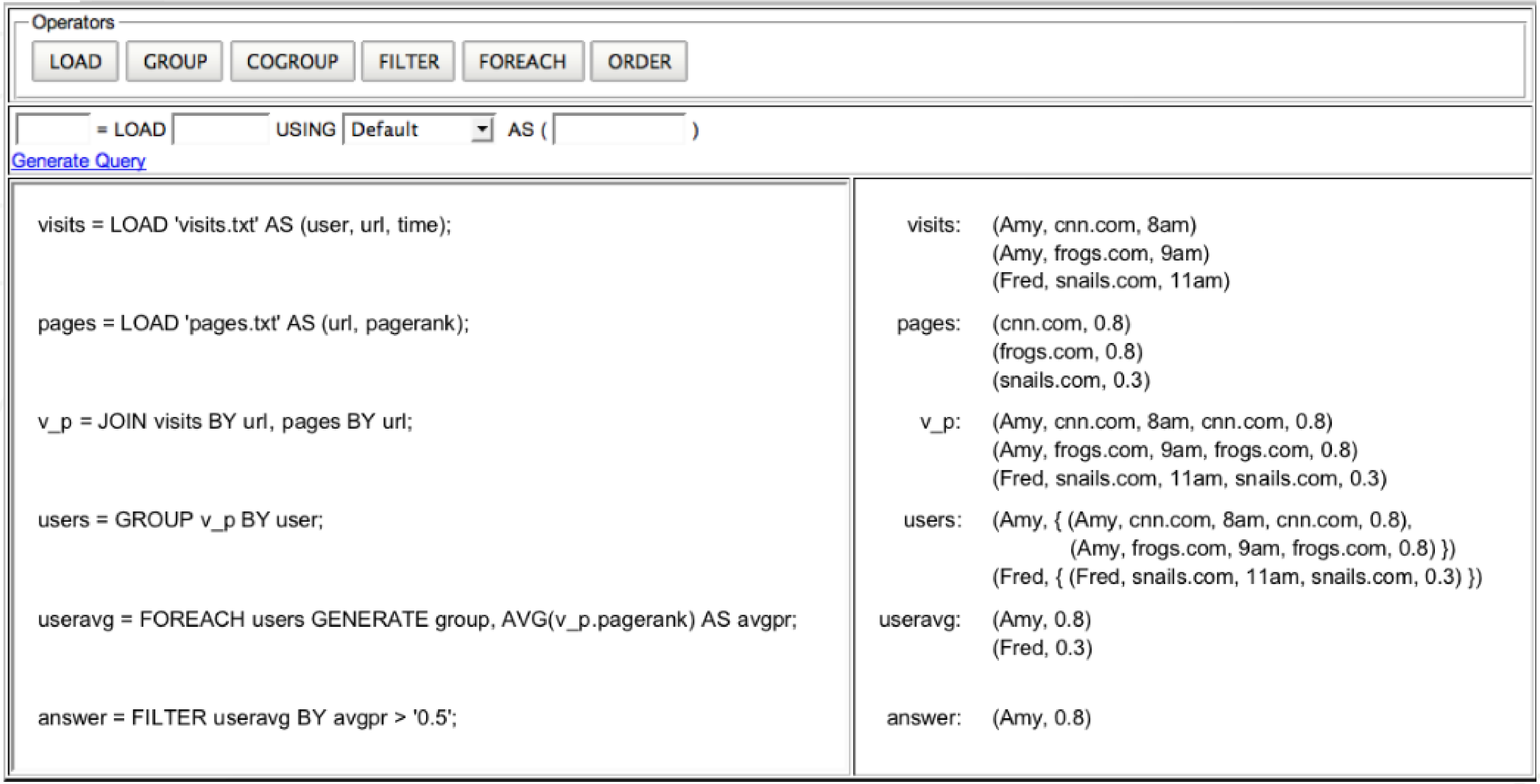
  R1 (x,y)

  R2 (x,z)

  R1 EQUI JOIN R2 might return empty set, if subset of datasets do not contain matching values for column 'x'.

- Hence, Pig Pen: Dynamically constructs side data set
- Advantages: Spot bugs, simplifies writing program incrementally

Pig Pen


Operators
LOAD
GROUP
COGROUP
FILTER
FOREACH
ORDER
= LOAD
USING
Default
AS (
)
Generate Query
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)

Generating Sandbox Dataset

- LOAD command: Sandbox dataset
- Primary Objectives:

1. **Realism**: Subset of actual dataset or synthesized from actual data
2. **Conciseness**: Small as possible, remove redundancies
3. **Completeness**: Illustrate semantics of command

Use-cases

- **Rollup Aggregates:** Calculating various aggregates
  Example: Counting the number of searches per user and computing the average per-user count in web crawls
- **Temporal Analysis:** COGROUP command in Pig is particularly useful for this task as it groups search queries from different time periods, facilitating custom processing.
  Example: Understanding how search query distributions change overtime
- **Session Analysis:** Natural abstraction and manipulation of sessions
  Example: Calculate metrics such as average session length, number of clicks before leaving a website, and variations in click patterns over time

Why is Pig better than OLAP?

- **Scalability and Flexibility**: Directly compute aggregates over large distributed files without the need for prior curation.
- **Ease of Incorporation of Custom Processing**: Easy integration of custom processing steps, such as IP-to-geo mapping and n-gram extraction.
- **Efficiency with Large Datasets:** Orchestrates a sequence of multiple map-reduce jobs.
- **Natural Data Representation**: Manipulate complex data structures/nested data like user sessions.

Related Work

Pig Latin vs. Other Platforms

- **Dynamo**: Focused on transactional key-value storage, not batch analytics like Pig.
- **Dryad**: More high-level than Dryad's low-level DAG model. Pig Latin could potentially compile to Dryad jobs.
- **DryadLINQ**: Similar high-level language, but Pig Latin has a more procedural style.
- **MapReduce**: More flexible than MapReduce's rigid two-step structure. Pig Latin allows chaining multiple operations.
- **Sawzall**: More flexible than Sawzall's fixed map+aggregate structure. Pig Latin supports arbitrary UDFs and operations like joins.
- **NESL**: Pig Latin adds data combination operations (e.g. join, cogroup) on top of nested data model.

Future Work

"Safe" Optimizer:

- Implement optimizations that guarantee performance benefits

Improved User Interfaces

- Develop a "boxes-and-arrows" GUI for visual program specification
- Enhance collaboration features (e.g., sharing program fragments, UDFs)

External Functions Support

- Enable UDFs in scripting languages (e.g., Perl, Python)
- Implement lightweight serialization/deserialization layer

Unified Development Environment

- Integrate control structures (loops, conditionals) into Pig Latin
- Embed Pig Latin into established languages (e.g., Perl, Python) for remote execution through packages
- Create a single environment for:
    - Main program development
    - Pig Latin commands
    - UDF writing

Summary

- Developed by Yahoo! in 2006, became Apache top-level project in 2010
- **Purpose**: High-level platform for analyzing large datasets, bridging SQL and MapReduce
- **Key Features**:
- Pig Latin: Simplified scripting language for data analysis
- Compiles to MapReduce jobs, runs on Hadoop
- Supports complex data types and user-defined functions
- **Current Status**:
- Latest stable release: v0.17.0 (May 2018)
- Open-source, relevant in Hadoop ecosystem
- Competitors: Google BigQuery, Amazon Redshift, Apache Hive, Apache Spark

Study Questions

1) Convert the following SQL query to a Pig Latin script. The SQL query retrieves the names and total sales of each product category from a `sales` table, grouped by category.

```
SELECT category, SUM(amount) as total_sales
FROM sales
GROUP BY category;
```

2) For the below web crawl data, where each record includes `url` and `visit_duration` (in seconds), write a Pig Latin script that categorizes each visit based on the `visit_duration` and outputs the `url`, `visit_duration`, and `category (Long, Medium, Short)` for each visit.

```
http://example.com, 450
http://example.org, 200
http://example.net, 50
http://example.edu, 600
```

Thank You!
Questions?

Georgia Tech