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

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).

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
  - 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

Pig Latin Example

Table urls: (url,category, pagerank)

Find for each suffciently large category, the average pagerank of high-pagerank urls in that category

SQL:

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

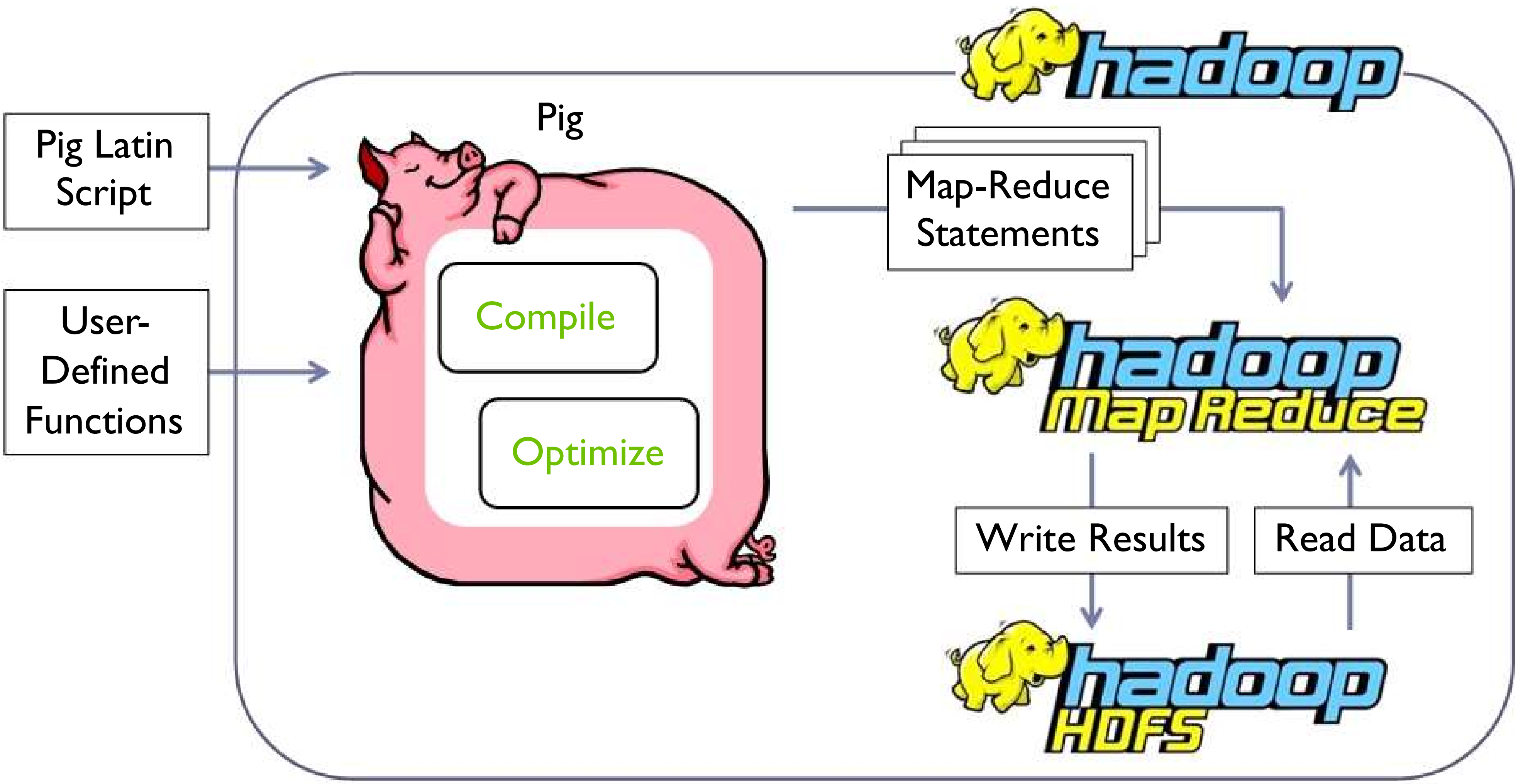
Pig Latin:

```
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);
```

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



Pig Latin Script
User-Defined Functions
Pig
Compile
Optimize
hadoop
Map-Reduce Statements
hadoop
MapReduce
Write Results
Read Data
hadoop
HDFS

Data Model

- Atom - simple atomic value (ie: number or string)
- Tuple
- Bag
- Map

$$\left(\text{'alice'}, \left\{ \begin{array}{c} (\text{'lakers'}, 1) \\ (\text{'iPod'}, 2) \end{array} \right\}, [\text{'age'} \rightarrow 20] \right)$$

Data Model

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

$$\left(\text{'alice'}, \left\{ \begin{array}{c} (\text{'lakers'}, 1) \\ (\text{'iPod'}, 2) \end{array} \right\}, [\text{'age'} \rightarrow 20] \right)$$

Data Model

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

$$\left(\text{'alice'}, \left\{\begin{array}{c}(\text{'lakers'}, 1)\\(\text{'iPod'}, 2)\end{array}\right\}, [\text{'age'} \rightarrow 20]\right)$$

Data Model

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

$$\left(\text{'alice'}, \left\{ \begin{array}{l} (\text{'lakers'}, 1) \\ (\text{'iPod'}, 2) \end{array} \right\}, [\text{'age'} \rightarrow 20] \right)$$

Data Model

- Control over dataflow

  Ex 1 (less efficient)

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

  Ex 2 (most efficient)

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

- 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)
  - Ex: spam_urls = FILTER urls BY isSpam(url);
  - 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 deserializer with "USING"
  - Can provide a schema with "AS"

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

```
Queries = LOAD 'query_log.txt'
          USING myLoad()
          AS (userID,queryString, timeStamp)
```

Speaking Pig Latin

- 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))

```
newBag =
   FOREACH bagName
   GENERATE field1, field2, ...;
```

Speaking Pig Latin

- FILTER
  - Select a subset of the tuples in a bag

```
newBag = FILTER bagName
              BY   expression;
```

  - Expression uses simple comparison operators (==, !=, <, >, …) and Logical connectors (AND, NOT, OR)

```
some_apples =
    FILTER apples BY colour != 'red';
```

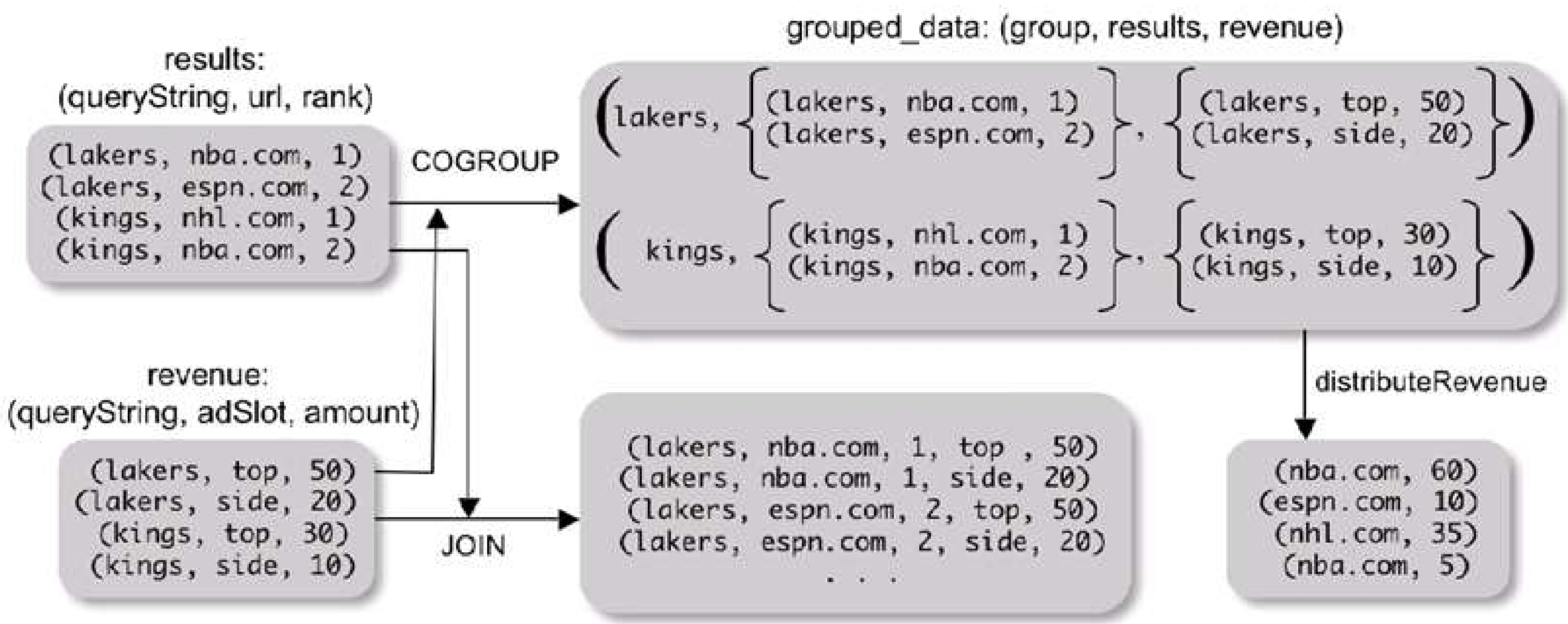
  - Can use UDFs

```
some_apples =
    FILTER apples BY NOT isRed(colour);
```

Speaking Pig Latin

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

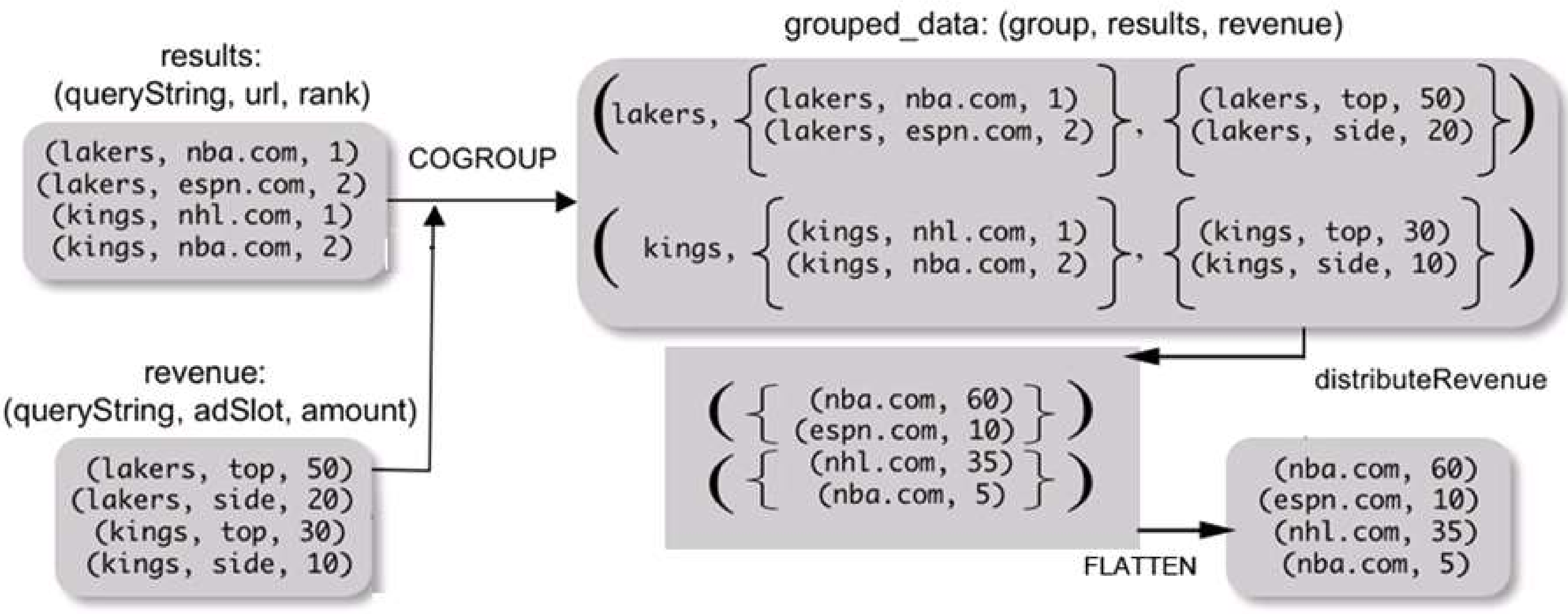
```
grouped_data = COGROUP results BY queryString,
                       revenue BY queryString;
```



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:

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

Equivalent

```
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(variable) defined

Compilation

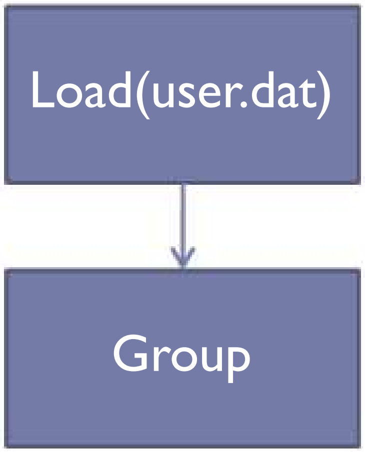
- Building a Logical Plan Example

```
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';
```

Compilation

- Building a Logical Plan Example

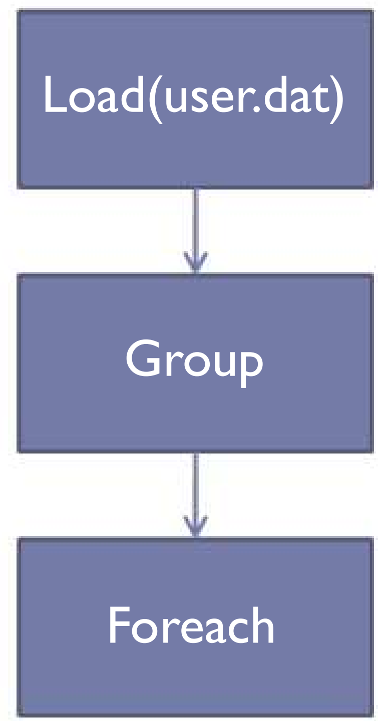
```
A = LOAD 'user.dat' AS (name, age, city);
B = GROUP A BY city;
C = FOREACH B GENERATE group AS city,
        COUNT(A);
D = FILTER C BY city IS 'kitchener'
        OR city IS 'waterloo';
STORE D INTO 'local_user_count.dat';
```



Compilation

Building a Logical Plan Example

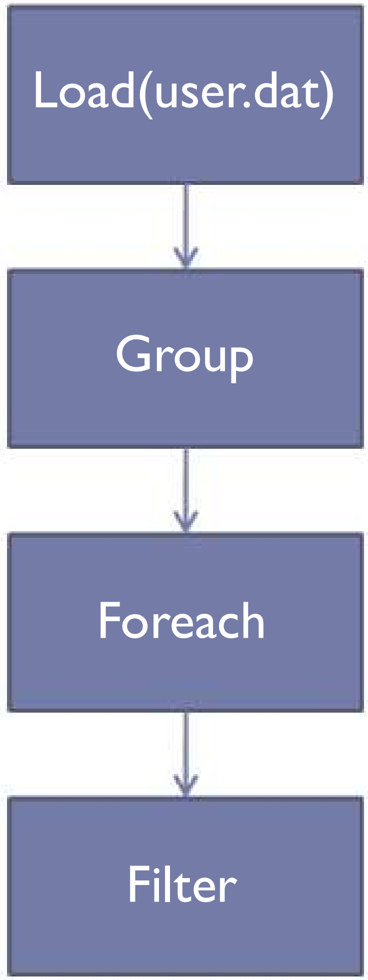
```
A = LOAD 'user.dat' AS (name, age, city);
B = GROUP A BY city;
C = FOREACH B GENERATE group AS city,
       COUNT(A);
D = FILTER C BY city IS 'kitchener'
       OR city IS 'waterloo';
STORE D INTO 'local_user_count.dat';
```



Compilation

Building a Logical Plan Example

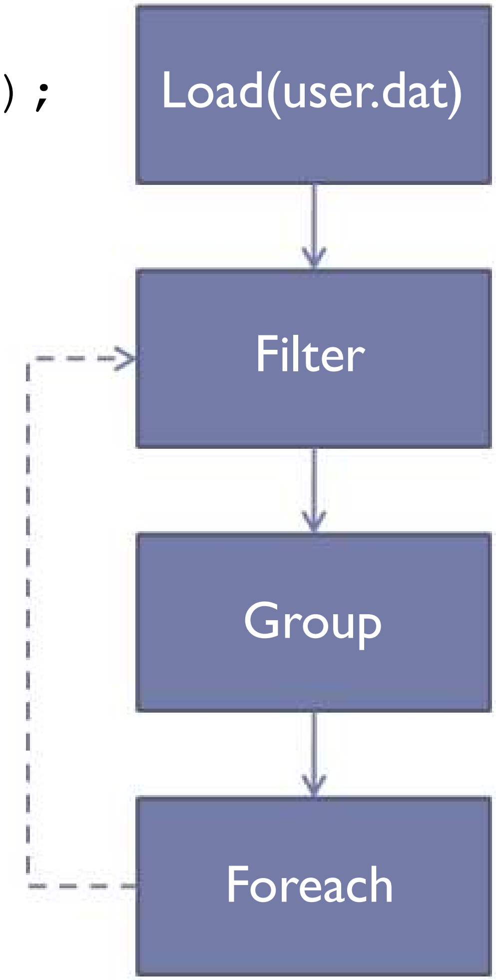
```
A = LOAD 'user.dat' AS (name, age, city);
B = GROUP A BY city;
C = FOREACH B GENERATE group AS city,
          COUNT(A);
D = FILTER C BY city IS 'kitchener'
          OR city IS 'waterloo';
STORE D INTO 'local_user_count.dat';
```



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';
```

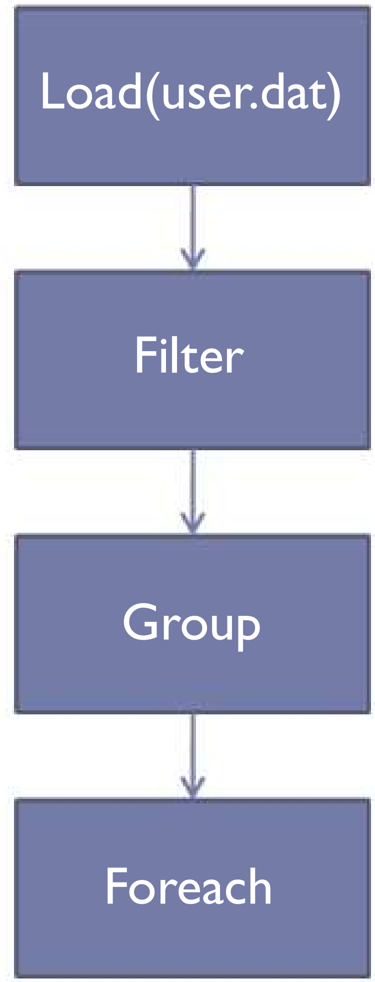


Compilation

Building a Physical Plan

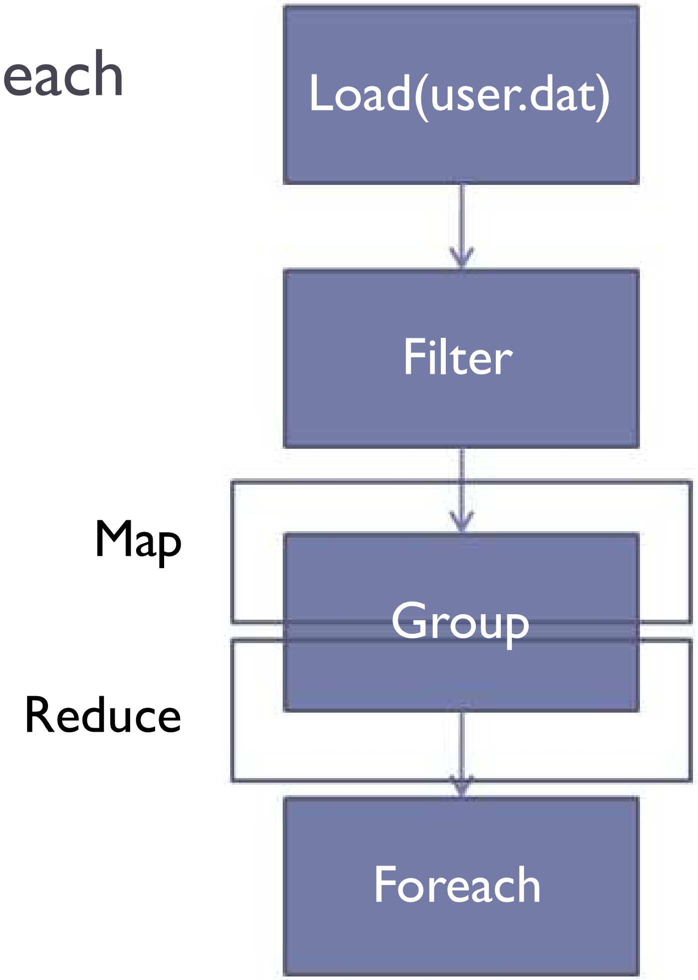
```
A = LOAD 'user.dat' AS (name, age, city);
B = GROUP A BY city;
C = FOREACH B GENERATE group AS city,
        COUNT(A);
D = FILTER C BY city IS 'kitchener'
        OR city IS 'waterloo';
STORE D INTO 'local_user_count.dat';
```

Only happens when output is specified by STORE or DUMP



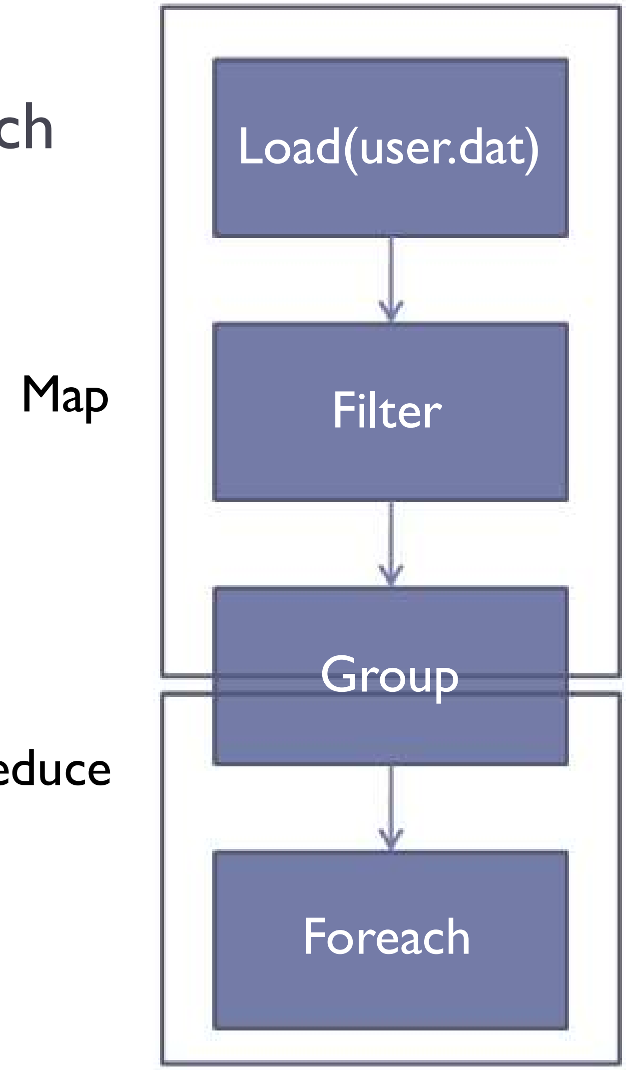
Compilation

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



Compilation

- 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 separate map-reduce jobs)



Compilation

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

Compilation

- 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)

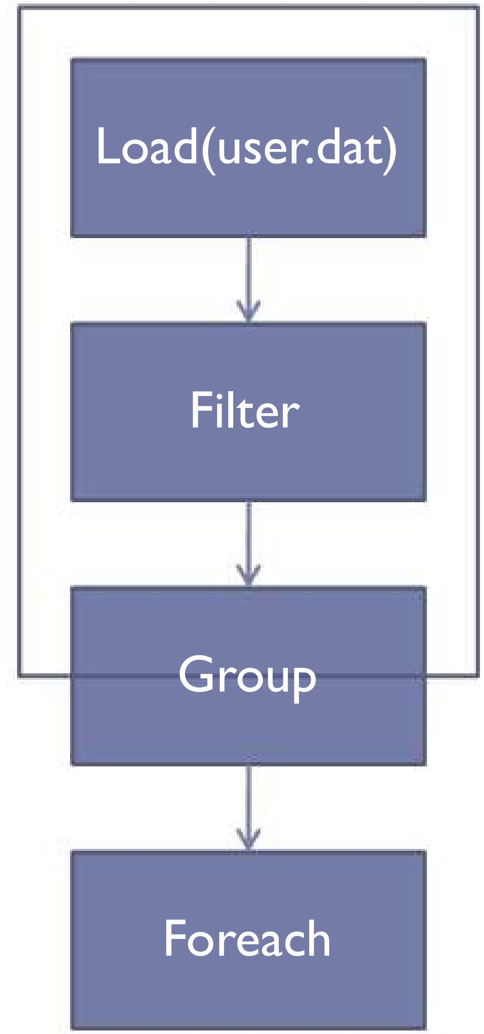
Compilation

- Efficiency with Nested Bags

$$\begin{bmatrix} \text{'waterloo'}, (\text{'Alice'}, 21, \text{'waterloo'}) \\ \text{'kitchener'}, (\text{'Charles'}, 36, \text{'kitchener'}) \end{bmatrix}$$

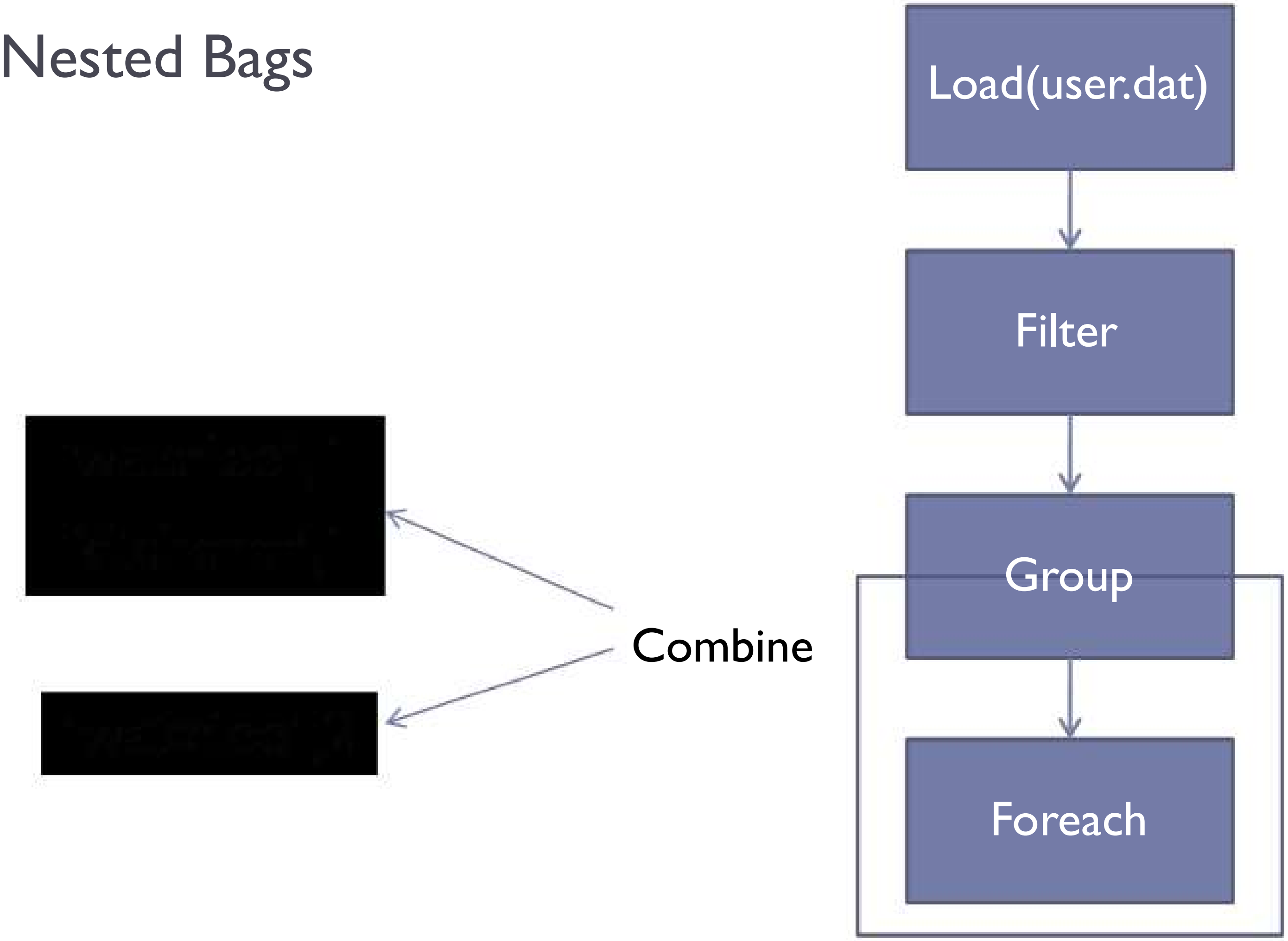
Map

$$\begin{bmatrix} \text{'waterloo'}, (\text{'Bob'}, 18, \text{'waterloo'}) \\ \text{'waterloo'}, (\text{'Pete'}, 39, \text{'waterloo'}) \end{bmatrix}$$



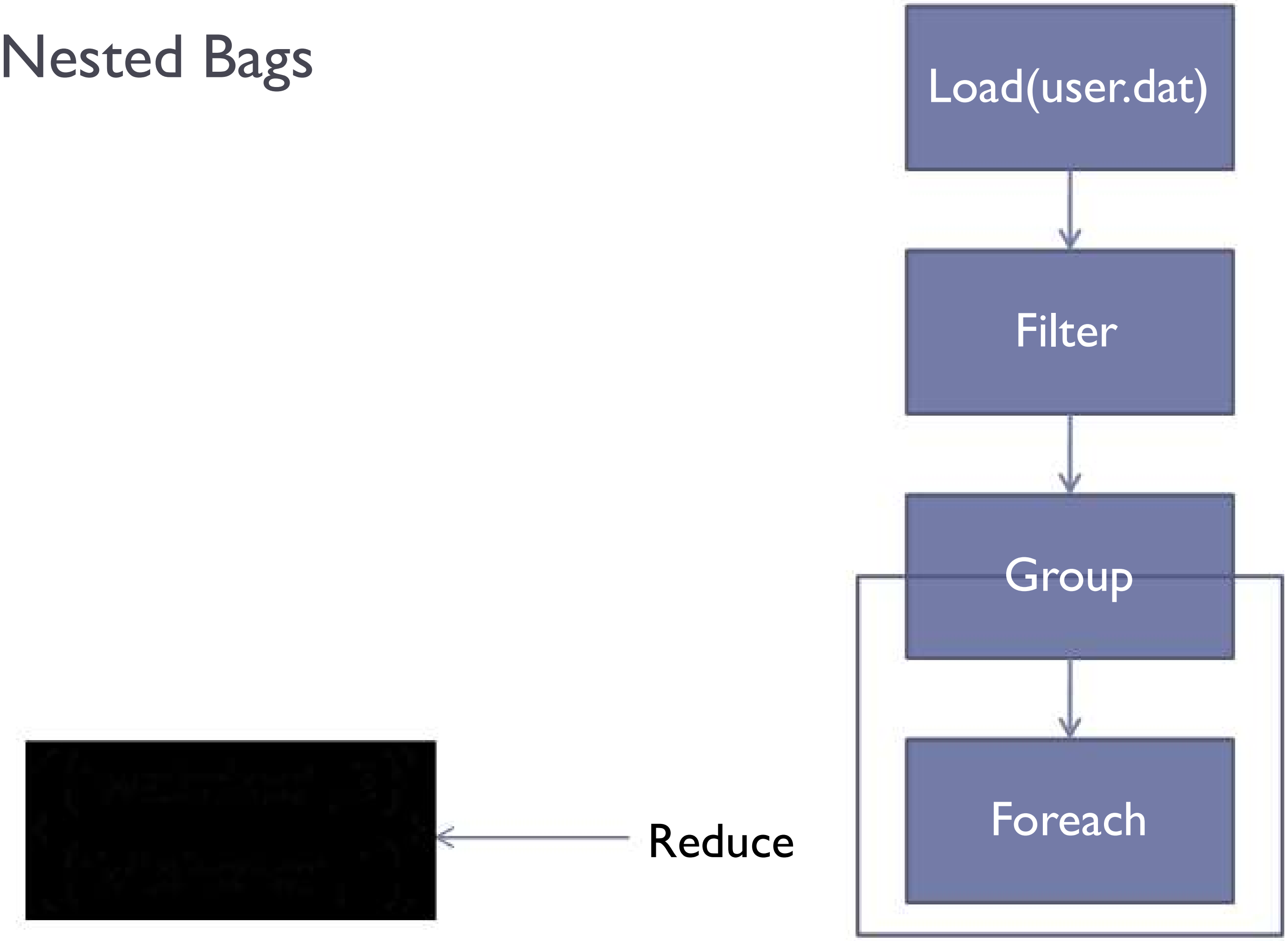
Compilation

- Efficiency with Nested Bags



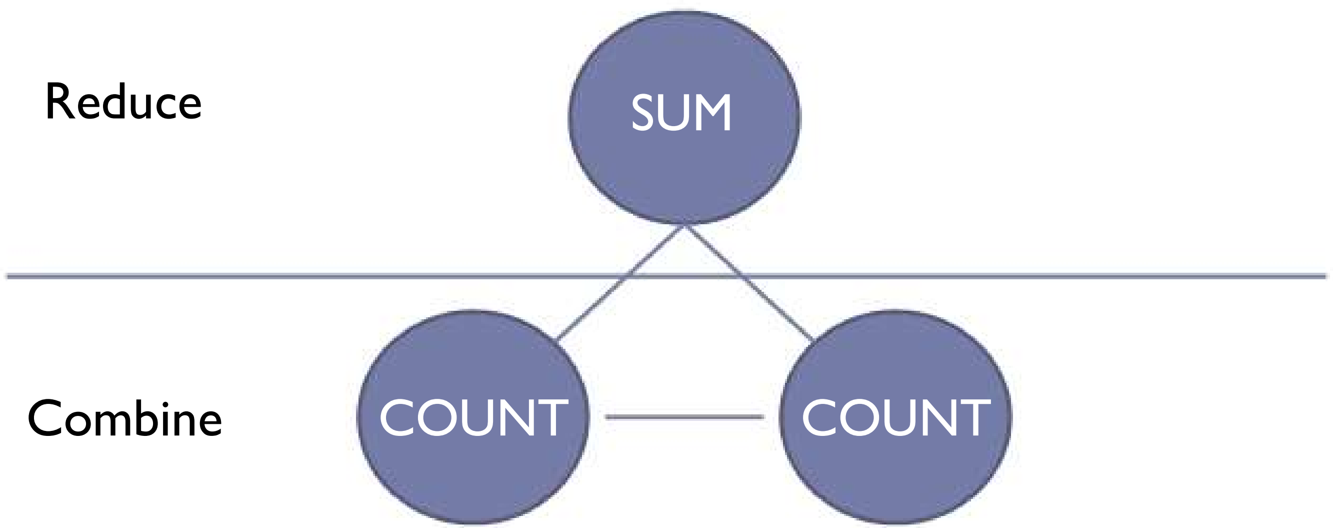
Compilation

- Efficiency with Nested Bags



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



Compilation

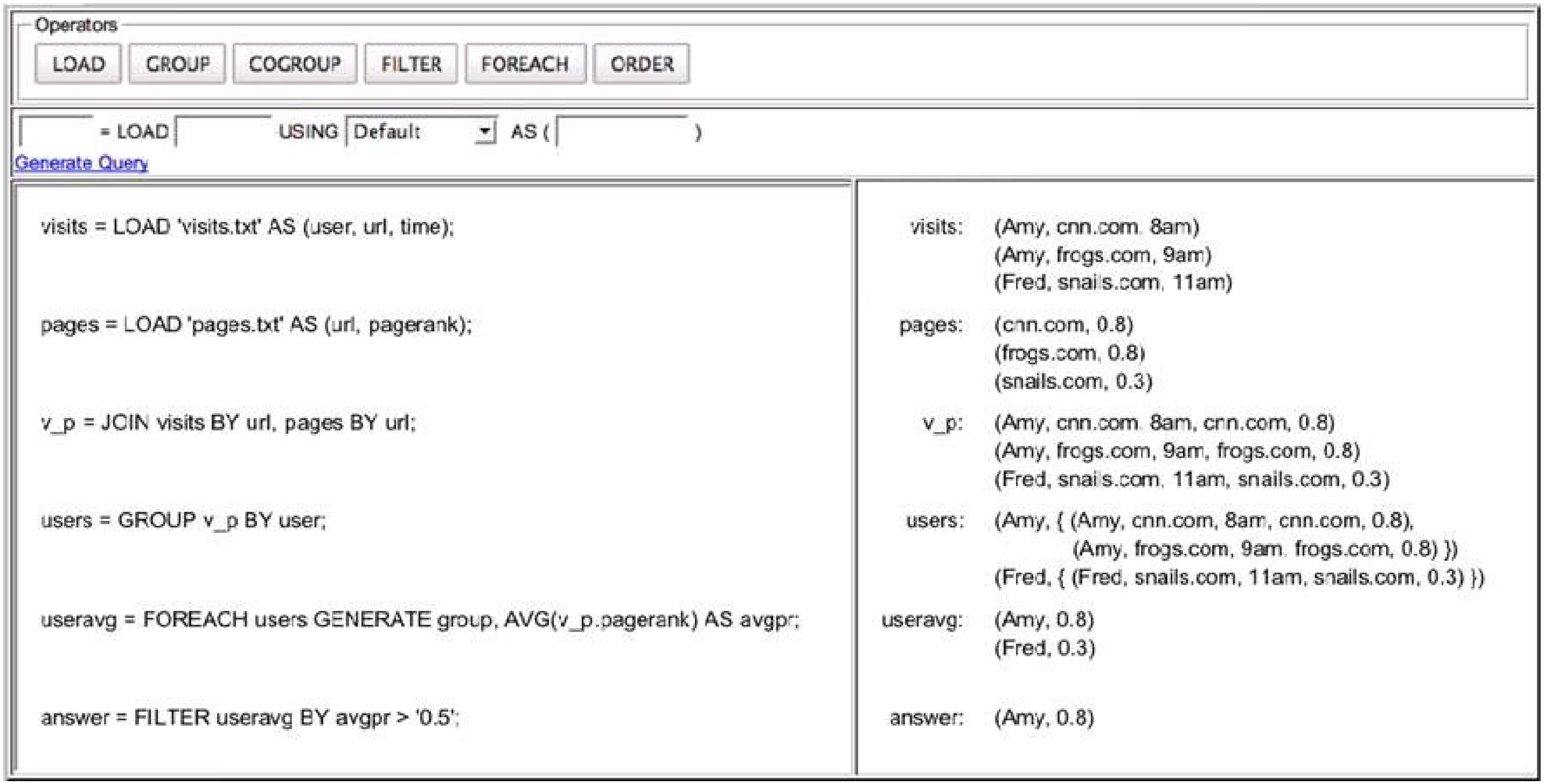
- Efficiency with Nested Bags
  - Pig provides an interface for writing algebraic UDFs so they can take advantage of this optimization as well.
- Inefficiencies
  - Non-algebraic aggregate functions (ie: MEDIAN) need entire bag to materialize; may cause a very large bag to spill to disk if it doesn‟t fit in memory
  - Every map-reduce job requires data be written and replicated to the HDFS (although this is offset by parallelism achieved)

Debugging

- 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

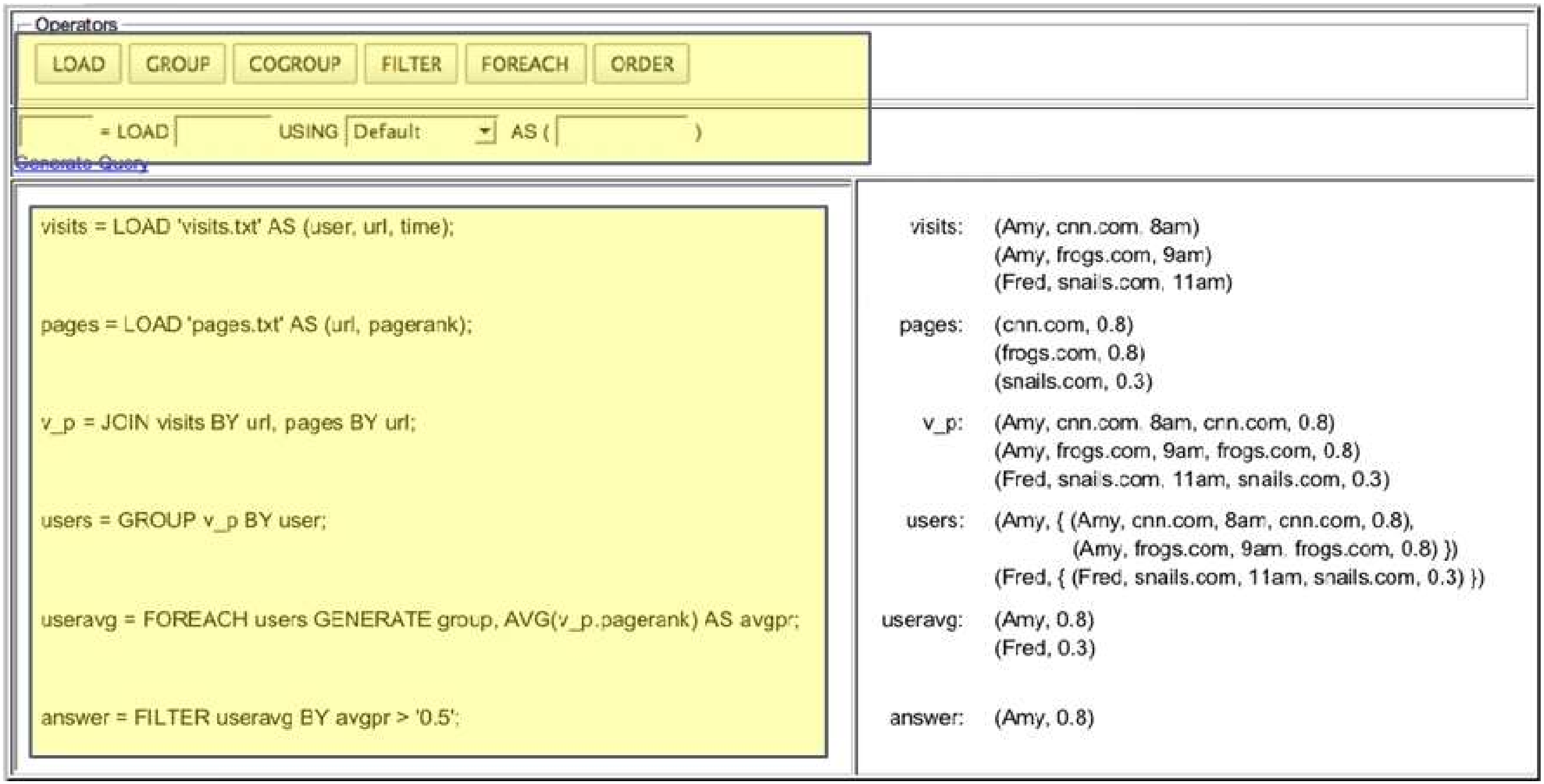
Debugging

- Pig-Pen



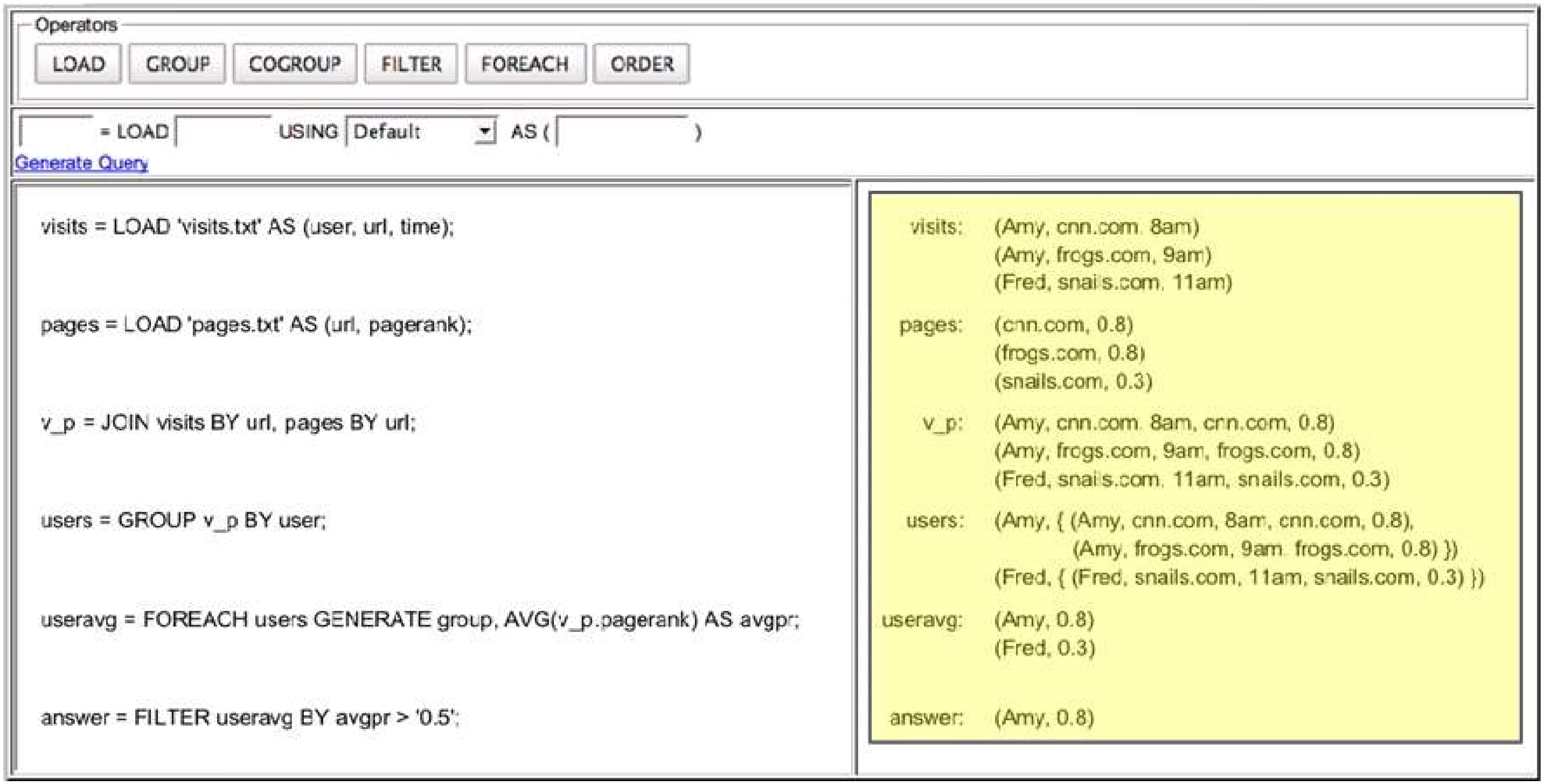
Debugging

- Pig-Latin command window and command generator



Debugging

- Sand Box Dataset (generated automatically!)



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

Conclusion

- 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

- Pig, http://hadoop.apache.org/pig/
- Hadoop, http://hadoop.apache.org