


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

Christopher Olston, Benjamin Reed,
Utkarsh Srivastava, Ravi Kumar,
Andrew Tomkins (Yahoo! Research)

Presented by Aaron Moss (University
of Waterloo)



“We have this problem ...”

- > Developers at top web properties trying to make their product better
 - > They have massive, un-indexable datasets to work with
 - > And they want to do ad-hoc analysis on them
- 

“... and we have this tool ...”

- > Map-reduce over a parallel cluster solves the problem quite efficiently
- > Takes advantage of the massive parallelism inherent in the data analysis problem




“... how do we make it simple?”

- > Map-reduce is hard to use
 - doesn't support common operations like join in a reusable fashion
- > SQL is unfamiliar
 - Users are developers who are more comfortable with procedural than declarative code



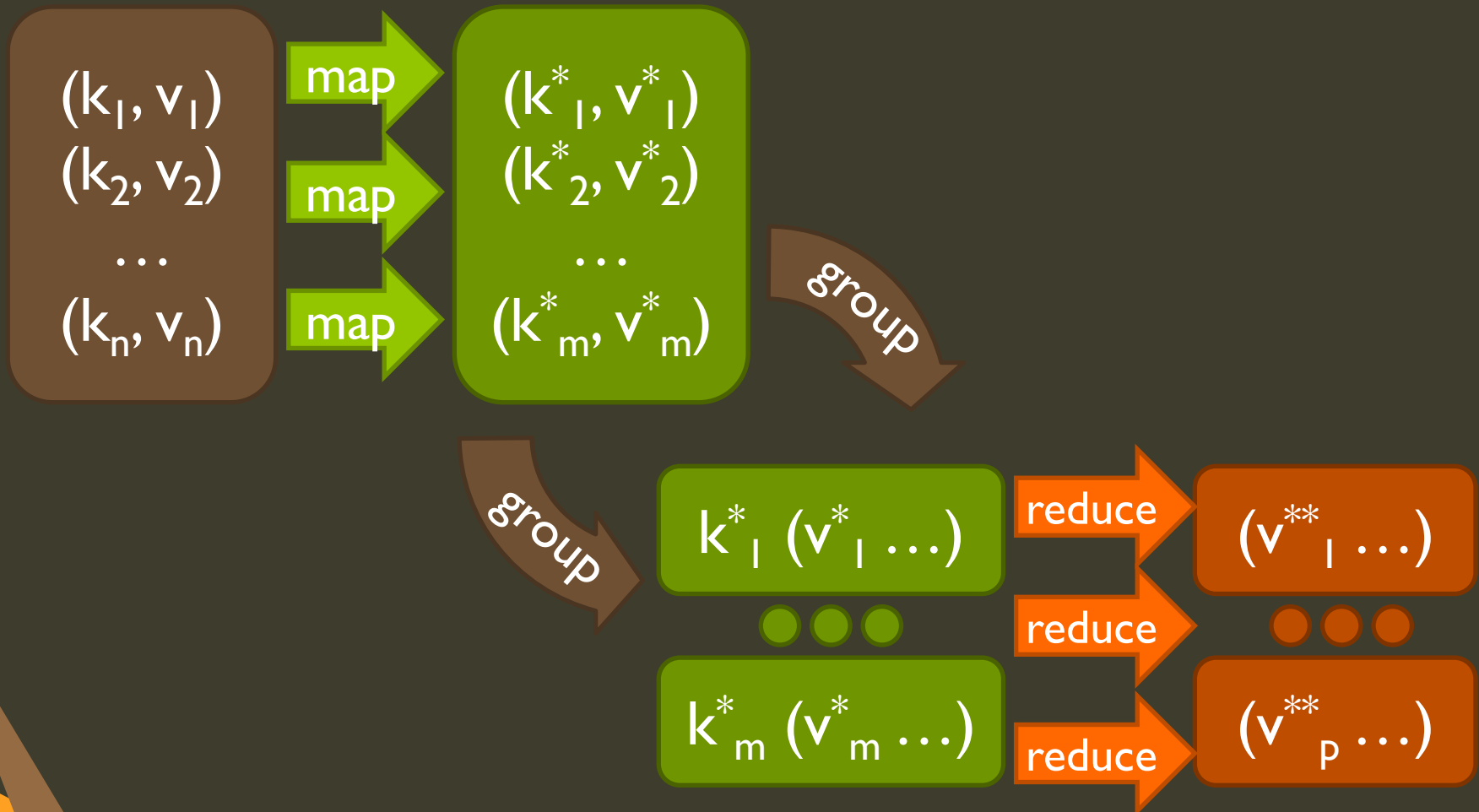
A different kind of question

- > This class has largely been concerned with implementing relational database constructs efficiently in a distributed setting
 - > This paper is concerned with building easy to use constructs on top of an efficient distributed implementation
- 

Introducing Pig Latin

- > A language designed to provide abstraction over procedural map-reduce without being so declarative as to be opaque
- > The authors have built a query translator called *Pig* for Pig Latin
 - Written in Java for the Hadoop map-reduce environment
 - Open source and available at <http://pig.apache.org>

Quick Overview of Map-Reduce



Disadvantages of SQL


- > Not as inherently parallelizable
- > Declarative style uncomfortable for procedural programmers
- > Many primitives with complex interaction
 - Hard for the programmer to know where the performance bottlenecks are

Advantages of Map-Reduce

- > Inherently Parallel
- > Procedural model
- > Only two primitives (map and reduce)
 - Makes it clear what the system is doing



Disadvantages of Map-Reduce

- > Can be difficult to make queries fit into the two-stage “map then reduce” model
 - > Can't optimize across the abstraction
 - > No primitives for common operations
 - Projection
 - Selection
- 

An Example

> SQL

```
SELECT category, AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > 1000000
```

> Pig Latin

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups
              BY COUNT(good_urls) > 1000000;
output = FOREACH big_groups GENERATE
          category, AVG(good_urls.pagerank);
```

Dataflow Language

- > Each line defines a single manipulation
- > Similar to a query execution plan
 - Lower level than SQL
 - This can aid optimization
- > User-defined I/O allows Pig to work with data stores that aren't databases
 - Queries are read-only (no transactions)
 - Queries are ad-hoc (less value in pre-built indices)

Data Model

- > **Atom**: single atomic value, e.g. string, number
- > **Tuple**: sequence of fields of any data type
- > **Bag**: collection of tuples
 - Possible duplicates
 - Tuples need not have consistent schema
- > **Map**: collection of data items which can be looked up by an atomic key
 - Keys must have same type, data items may not

Pig Expressions

```
t = ('quux', {( 'foo',1)( 'bar',2)}, [ 'baz' → 20])
```

Expression Type	Example	Value for t
Constant	'foo'	'foo'
Field (by position)	\$0	'quux'
Field (by name)	f3	['age' → 20]
Projection	f2.\$0	{('foo')('bar')}
Map Lookup	f3#'baz'	20
Function Evaluation	SUM(f2.\$1)	3
Conditional	f3#'baz'>42 ? 1:0	0
Flattening	f1, FLATTEN(f2)	('quux', 'foo', 1) ('quux', 'bar', 2)

Pig Latin Primitives - Input

- > `queries = LOAD 'query_log.txt'`
`USING myLoad() AS`
`(userID, queryString, timestamp);`
- > Loads from a file
- > Using a de-serializer
 - Default is plain-text tab-separated values
- > Given an optional schema
 - If not given, refer to data by position

Pig Latin Primitives - Output

- > **STORE** queries **INTO** 'q_output.txt'
USING myStore();
- > Stores bag to file
- > Using a serializer
 - Default is plain-text tab-separated values

Pig Latin Primitives - Per-tuple

- > `f_queries = FOREACH queries
GENERATE userId, f(queryString);`
- > For each tuple in the bag, generate the tuple described by the GENERATE expression
- > May use FLATTEN in the GENERATE expression to generate more than one tuple

Pig Latin Primitives - Per-tuple

- > `real_queries = FILTER queries
BY userId neq 'bot';`
- > Output each tuple in the bag only if it satisfies the BY expression
- > AND, OR, and NOT logical operators
- > `==`, `!=` operate on numbers, `eq`, `neq` on strings


Pig Latin Primitives - Grouping

- > `grouped_revenue = GROUP revenue BY queryString;`
- > Outputs a list of tuples, one for each unique value of the BY expression, where the first element is the value of that expression and the second is a bag of tuples matching that expression


Pig Latin Primitives - Grouping

- > `grouped_data = COGROUP
results BY queryString
revenue BY queryString;`
- > Outputs a list of tuples, one for each unique value of the BY expressions; the first element is that expression and the rest are bags of tuples from each BY which match that expression
- > Can generate a join by flattening the bags

Pig Latin Primitives - Sets

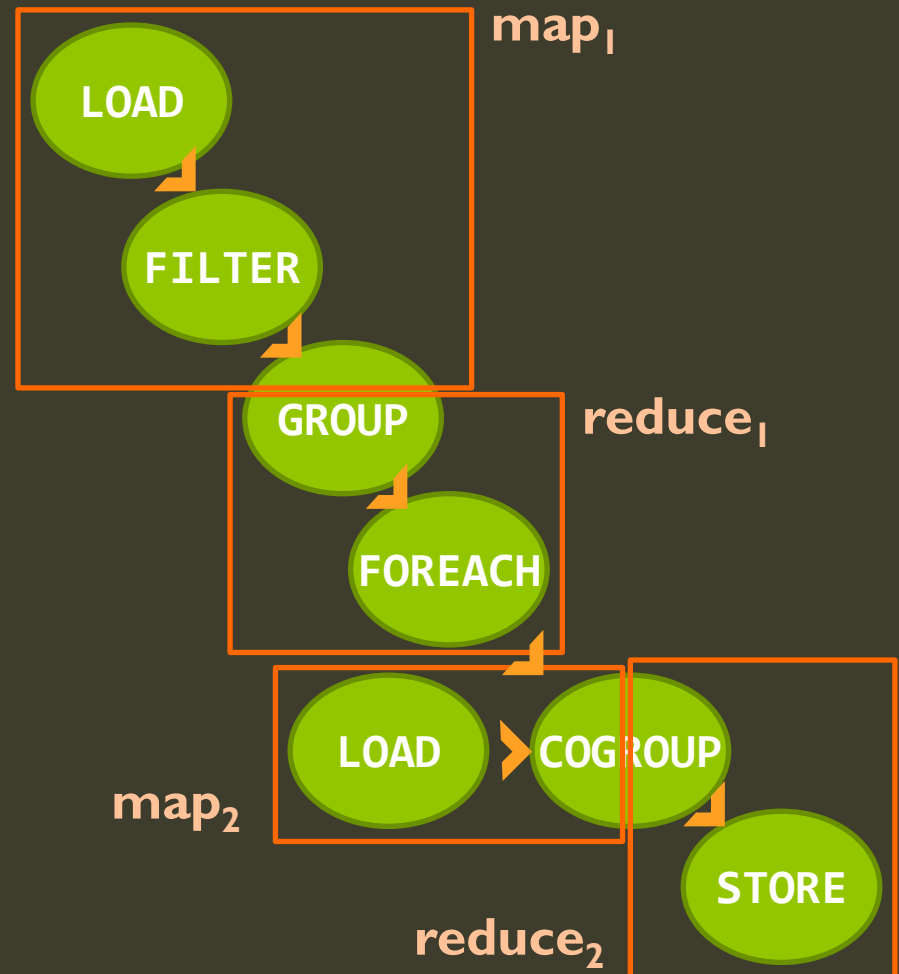
- > **UNION** returns the union of two bags
 - > **CROSS** returns the cross product of two bags
 - > **ORDER** sorts a bag by the given field(s)
 - > **DISTINCT** eliminates duplicate tuples in a bag
- 

Pig, the Pig Latin Compiler

- > Lazily builds execution plan as it sees each command
 - Allows optimizations like combining or re-ordering filters
 - > Processing is only triggered on STORE
 - > Logical plan construction is platform-independent, but designed for Hadoop map-reduce
- 

Implementation


- > Each (CO)GROUP command is converted into a map-reduce job
- > Map assigns keys to tuples by BY clauses
 - Also initial per-tuple processing
- > Reduce handles per-tuple processing up to the next (CO)GROUP




Efficiency

- > Moving/replicating data between successive map-reduce jobs results in overhead
- > When a nested bag is created, then aggregated using a parallelizable operation, can just perform the reduction as tuples are seen instead of materializing the bag
 - If this can't be done, spill to disk, use database-style sorting

Debugging

- > The **Pig Pen** environment dynamically creates a sandbox dataset to illustrate query flow
 - > Allows incremental query development without long execution times of full queries
 - > Takes random samples of data, attempts to provide full query coverage, synthesizes data where needed.
- 

Critique - Strengths

- > Approach of making language features translate clearly to low-level constructs
 - Allows the programmer to better optimize their work
 - > Simple, regular data model
 - > Ability to fit a full language overview in a short paper
- 

Critique - Weaknesses

- > Only anecdotal evidence for Pig Latin being easier to use than SQL
 - > Inadequate coverage of Pig Pen debugger
 - Algorithms are opaque, not apparently published
 - > No data to back up optimization claims
 - Can't prove programmers can optimize better than automated SQL optimizer
 - No comparison of Pig to optimized SQL
- 