# Introduction of the Design of A High-level Language over MapReduce

-- The Pig Latin

Yu LIU

NII 2010/04/17

# Papers about Pig - A Dataflow System

- Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience

  Alan F. Gates, et al, VLDB'09

- Generating Example Data for Dataflow Programs

  O. Christopher, et al, SIGMOD'09

- Automatic Optimization of Parallel Dataflow Programs

  O. Christopher, et al, USENIX'08

# Background: Pig System and Pig Latin

- Pig is a platform for analyzing large data sets (invented by Yahoo!)
- Pig Latin is a high-level language for expressing data analysis (dataflow) programs
- Pig Latin programs can be complied into Map/Reduce jobs and executed using Hadoop
  - Pig system is implemented by Java and support other languages UDFs (Python, and JavaScript)

# Similar Languages of Pig Latin

- LINQ the .NET Language Integrated Query
- HiveQL a SQL-like language
- Jaql a query language designed for Javascript Object Notation (JSON)
- Sawzall google's language for wrapping MapReduce
- Scope a parallel query language

# Background: Pig Latin

The syntax of Pig Latin:

```
A = LOAD 'student' USING PigStorage() AS (name:chararray, age:int, gpa:float);
X = FOREACH A GENERATE name,$2;
DUMP X;
(John,4.0F)
(Mary,3.8F)
(Bill,3.9F)
(Joe,3.8F)
```

Statements

| | |
|---|---|
| Rich Data Types | [Scalars], [Arrays], bag, tuple, map … |
| Relational Operators | LOAD, GROUP, FOREACH, IMPORT, JOIN, GENERATE, MAPREDUCE, STORE… |
| Arithmetic Operators | +, -, *, / , %, ? : |
| UDFs | User defined funtions |

# Practical Problems of MapReduce programming model

- it does not directly support complex N -step dataflows, which often arise in practice.
- lacks explicit support for combined processing of multiple data sets (such as *join*)
- frequently-needed data manipulation primitives must be coded by hand (like *filtering*, *aggregation* and *top-k thresholding* )

# Salient Features of Pig Latin

test.pig

```
1 urls = LOAD 'dataset' AS (url, category, pagerank);
2 groups = GROUP urls BY category;
3 bigGroups = FILTER groups BY COUNT(urls)>1000000;
4 result = FOREACH bigGroups GENERATE
5 group, top10(urls);
6 STORE result INTO 'myOutput';
```

1. Composable high-level data manipulation constructs in the spirit of SQL
   - *Pig programs encode explicit dataflow graphs, as opposed to implicit dataflow as in SQL*
   - *It is a step-by-step dataflow language , computation steps are chained together through the use of variables*

# Salient Features of Pig (-con)

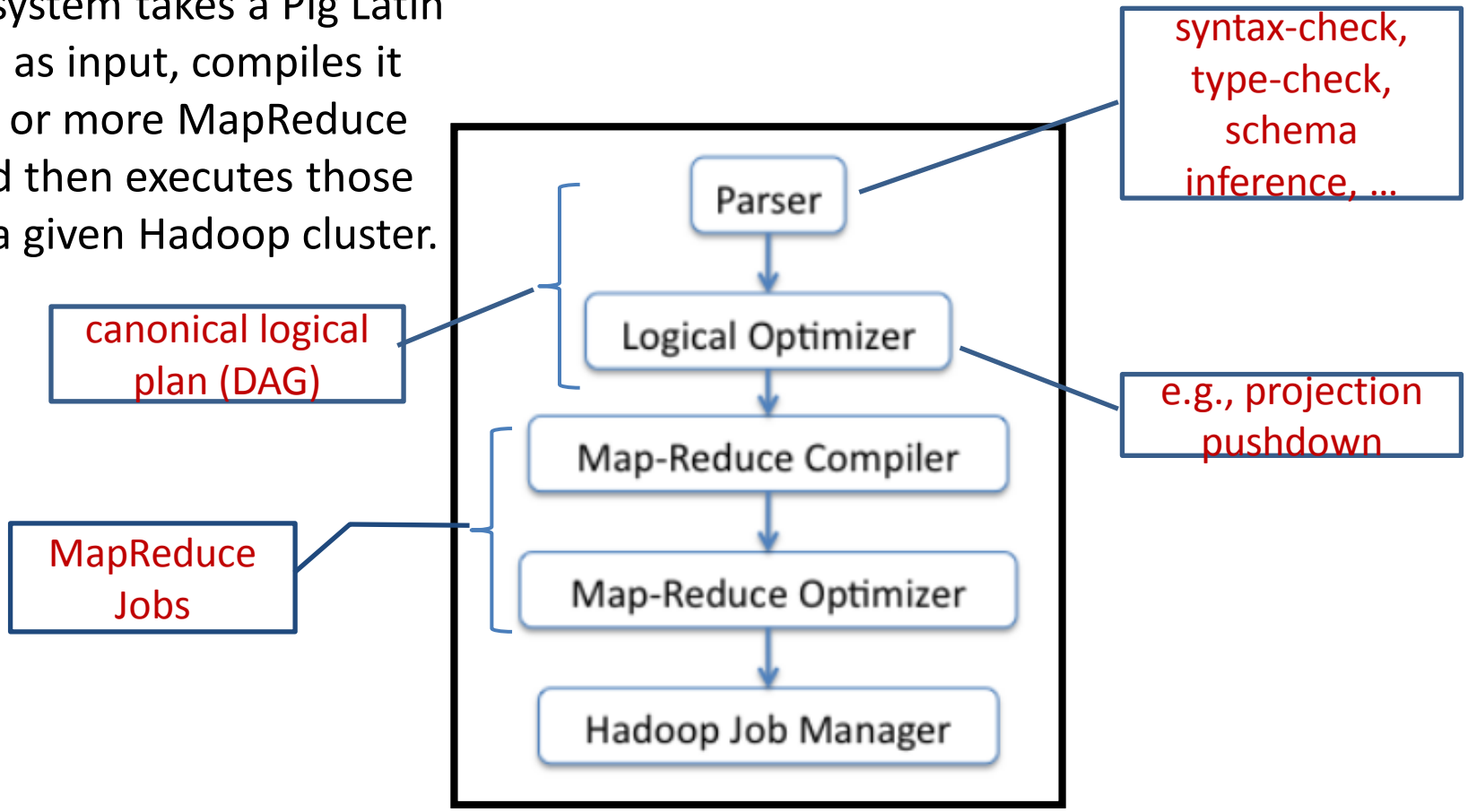
2. Pig compiles *Pig Latin* programs into sets of Hadoop MapReduce jobs, and coordinates their execution
3. High-level transformations (e.g., *GROUP, FILTER*)
4. Can specify schemas as part of issuing a program

# Salient features of Pig (-con)

5. UDFs as first-class citizens

# System Overview

The Pig system takes a Pig Latin program as input, compiles it into one or more MapReduce jobs, and then executes those jobs on a given Hadoop cluster.




*Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience*

# Type System of Pig

- Pig has a nested data model, can support complex, non-normalized data
  - scalar types: int, float, double, chararray(string)
  - complex types: map, tuple, and bag

| Type | |
|---|---|
| map | An associative array, {key:chararray, vla:any} |
| tuple | An ordered list of data elements |
| bag | A collection of tuples |

# Type System of Pig

- Type Declaration
  - by default is to treat all fields as *bytearray*

```
1 a = LOAD 'data' USING BinStorage AS (user);
2 b = GROUP a BY user;
3 c = FOREACH b GENERATE COUNT(a) AS cnt;
4 d = ORDER c BY cnt;
```

  - Pig is able to know the type of a field even it is not declared
  - type can be declared explicitly as part of the AS clause

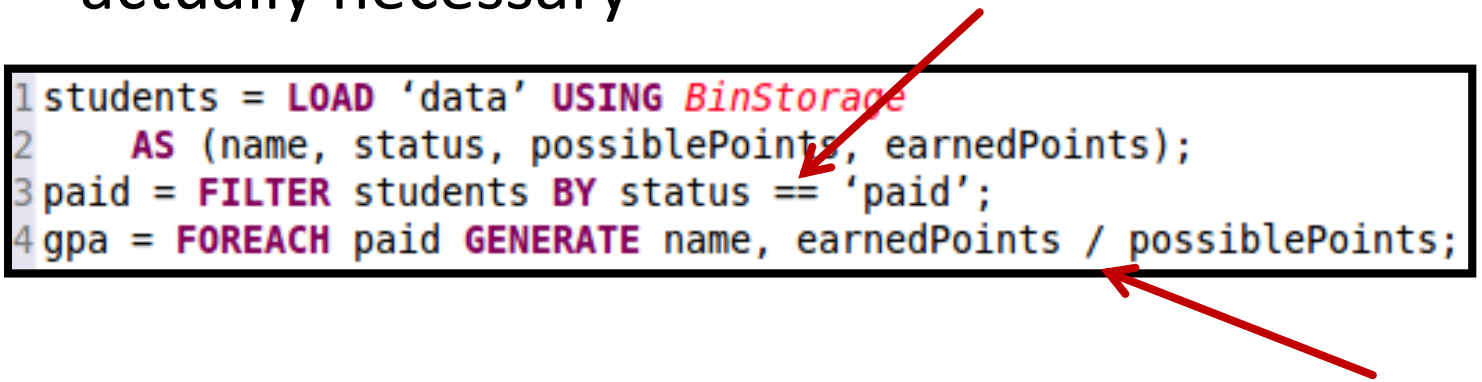
```
1 a = LOAD 'data' USING BinStorage
2             AS (user:chararray);
3 b = ORDER a BY user;
```

# Con - **Type System of Pig**
## Type Declaration

- Type information  can be defined in the schema (for self-describing data formats or non- self-describing data formats)

# Type System of Pig

- Lazy Conversion of Types
  - Type casting is delayed to the point where it is actually necessary



```
1 students = LOAD 'data' USING BinStorage
2     AS (name, status, possiblePoints, earnedPoints);
3 paid = FILTER students BY status == 'paid';
4 gpa = FOREACH paid GENERATE name, earnedPoints / possiblePoints;
```

# Generation, Optimization and Execution of Query Plans

- logical optimizations
  - Currently, IBM System R-style heuristics like *filter pushdown*

# Generation, Optimization and Execution of Query Plans

- Pig Latin program is translated in a one-to-one fashion to a logical plan

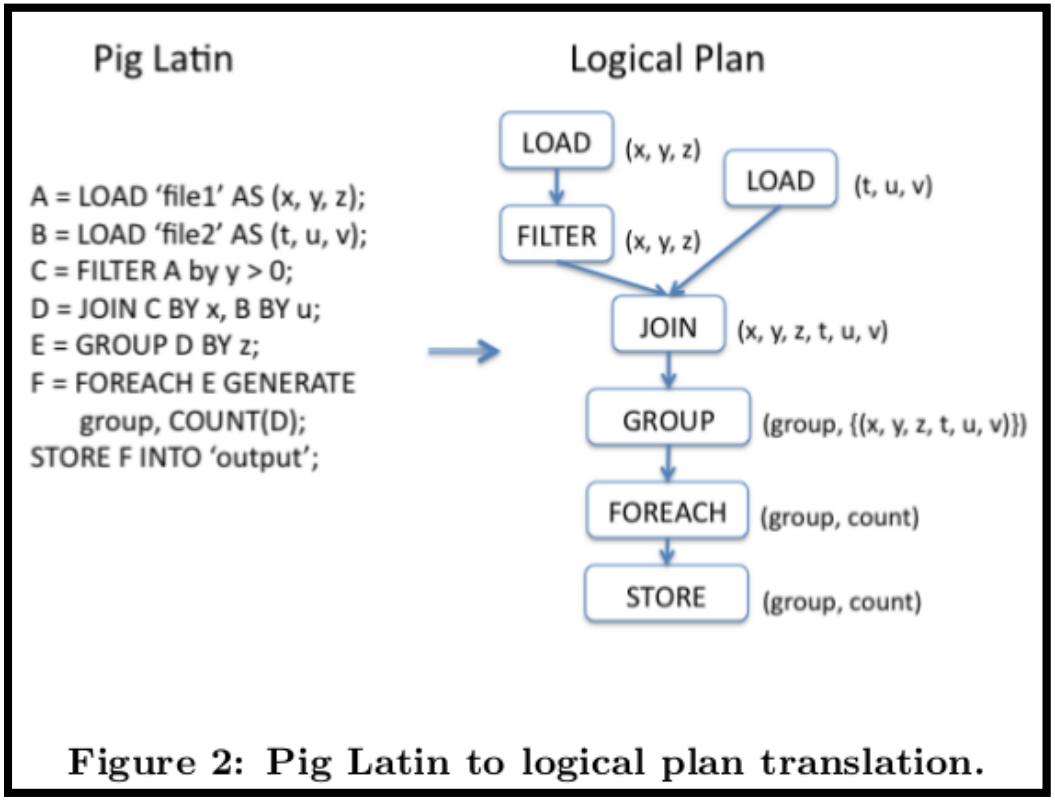



Figure 2: Pig Latin to logical plan translation.

*Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience*

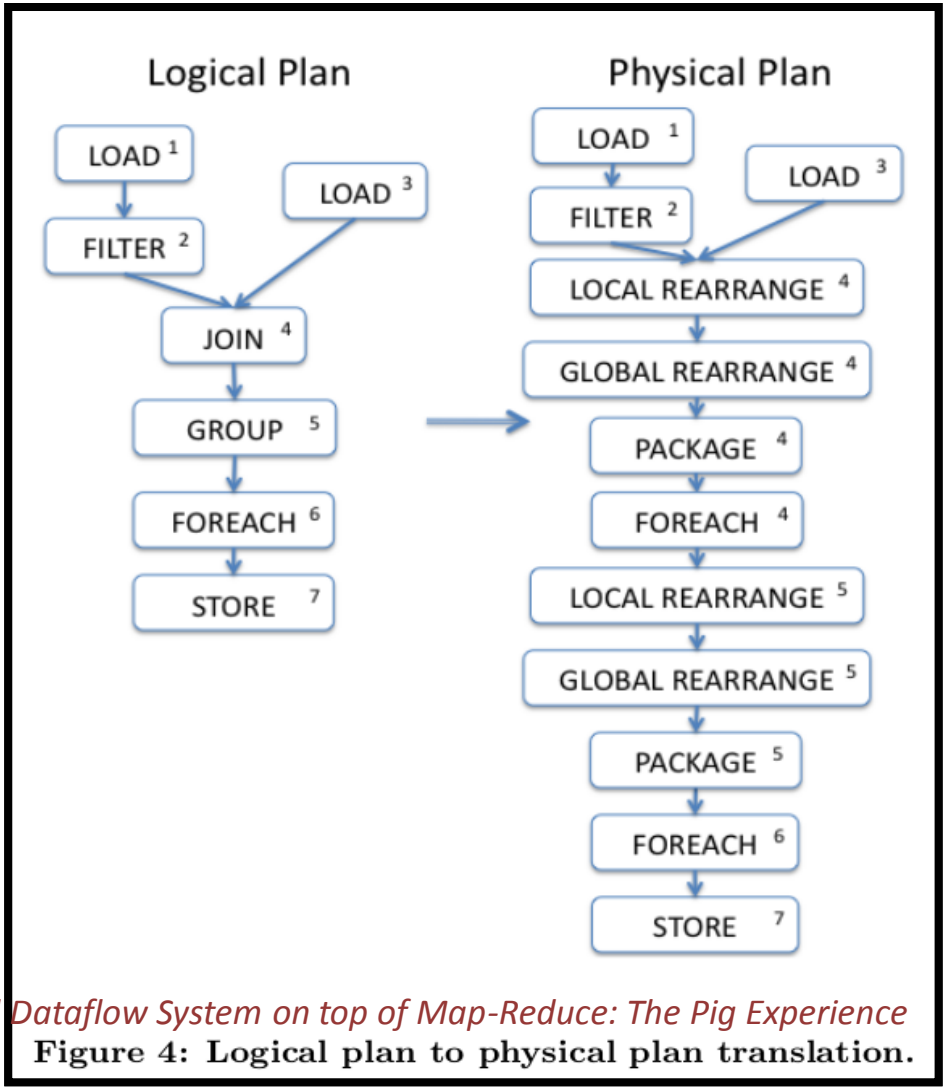
# Generation, Optimization and Execution of Query Plans

## -- logical plan into a physical plan

-The logical *GROUP* operator becomes: *local rearrange, global rearrange, package*

-The logical *JOIN* operator becomes: (1) *GROUP* followed by a *FOREACH*, (2) fragment replicate join

*(the syntax 'replicated', 'skewed', 'merge' deside which is performed)*

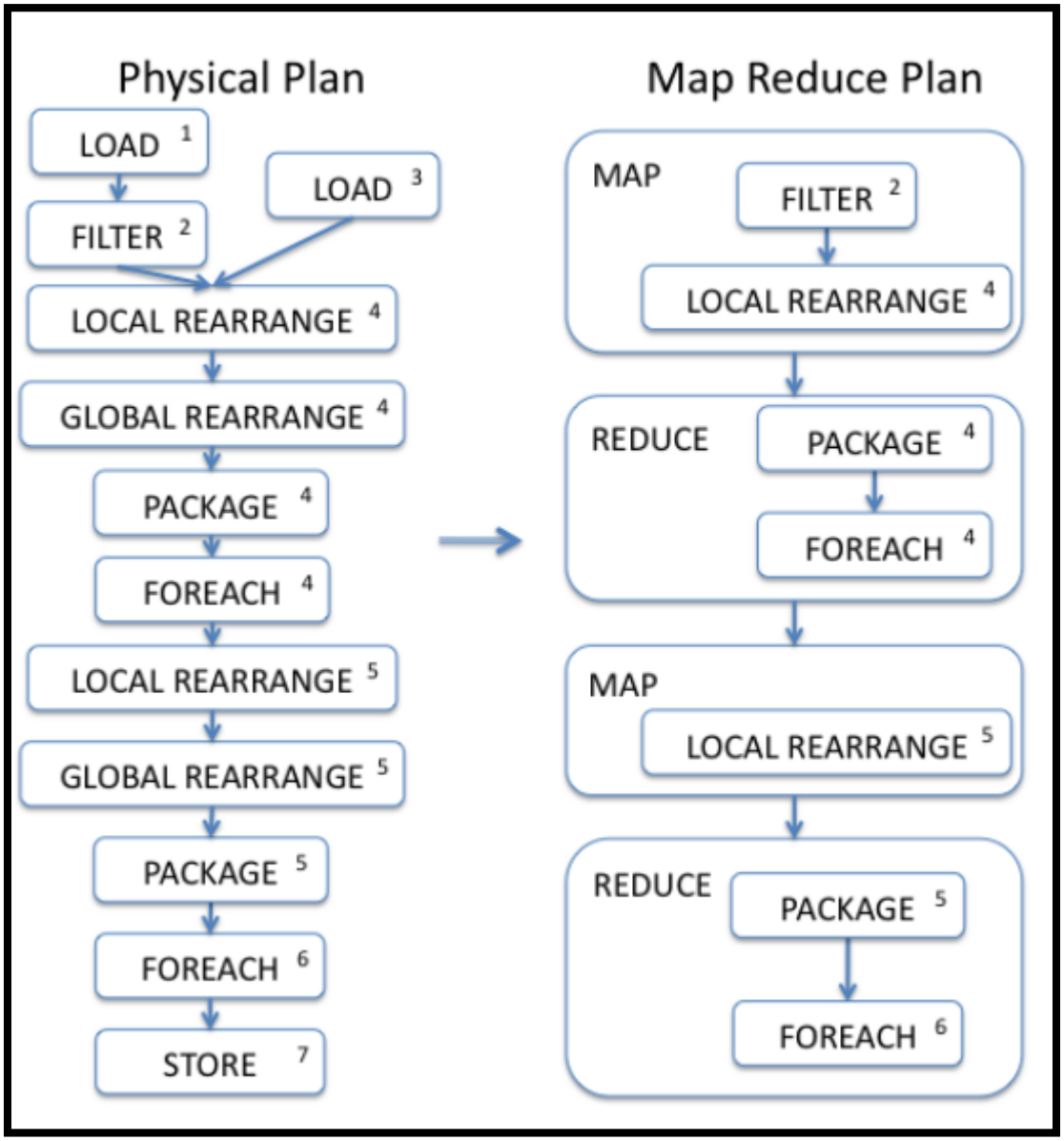



*Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience*

**Figure 4: Logical plan to physical plan translation.**

then embeds each *physical operator* inside a Map-Reduce stage to arrive at a *MapReduce plan*

*- Global rearrange operators are removed*




*Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience*

# Generation, Optimization and Execution of Query Plans

- MapReduce Optimization
  - Only one optimization is performed:

For distributive and algebraic aggregation functions, are break to *initial*, *intermediate*, *final* three steps, then assigned to the *map*, *combine*, and *reduce* stages respectively.

E.g., *AVERAGE* :

*initial*: generate (sum, count) ->

*intermediate*: combine [(sum,count) ] ->

*final*: reduce [(sum,count)]

# Generation, Optimization and Execution of Query Plans

- Compilation process generates a Java jar file that contains the Map and Reduce implementation classes
- The Map and Reduce classes contain general purpose dataflow execution engines

# Generation, Optimization and Execution of Query Plans

- The flow control is in an extended *iterator model* (pull model):

  When an operator is asked to produce a tuple, it can respond in one of three ways:
  - (1) return a tuple,
  - (2) declare itself finished, or
  - (3) return a pause signal to indicate that it is not finished but also not able to produce an output tuple at this time.

# MapReduce Execution Stages

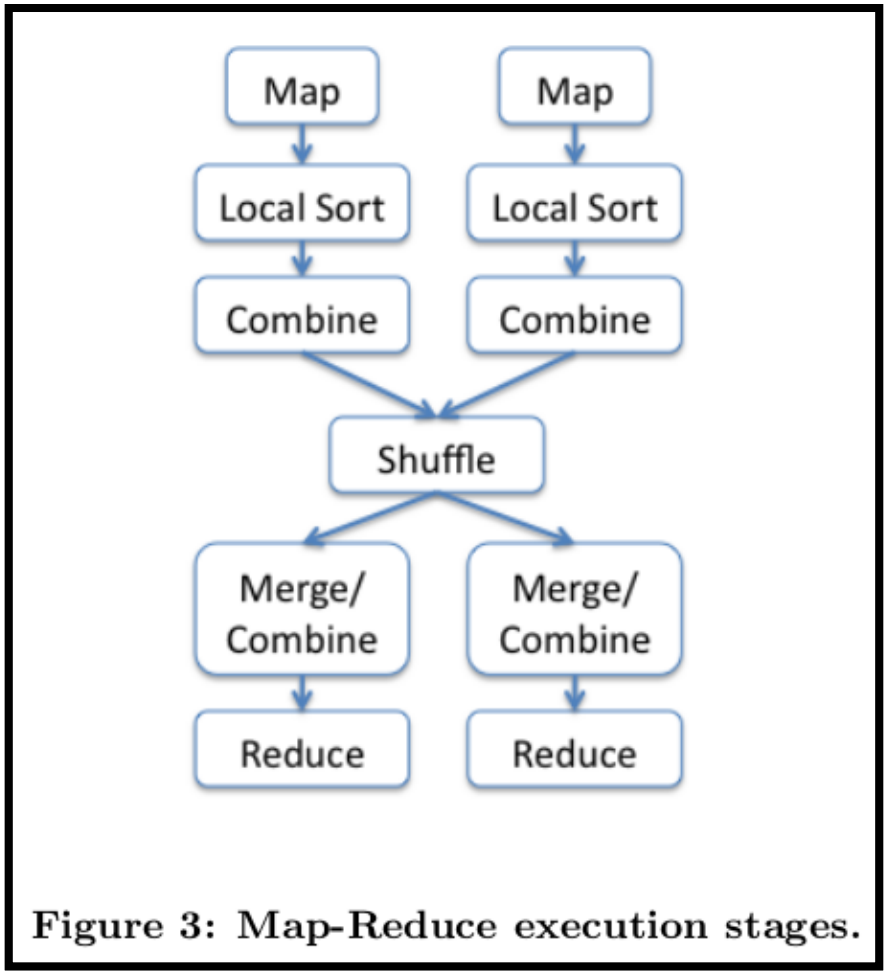



Figure 3: Map-Reduce execution stages.

*Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience*

# Generation, Optimization and Execution of Query Plans

- Pig permits a limited form of nested programming

```
1 clicks = LOAD 'clicks'
2          AS (userid, pageid, linkid, viewedat);
3 byuser = GROUP clicks BY userid;
4 result = FOREACH byuser {
5              uniqPages = DISTINCT clicks.pageid;
6              uniqLinks = DISTINCT clicks.linkid;
7              GENERATE group, COUNT(uniqPages), COUNT(uniqLinks);
8 };
```

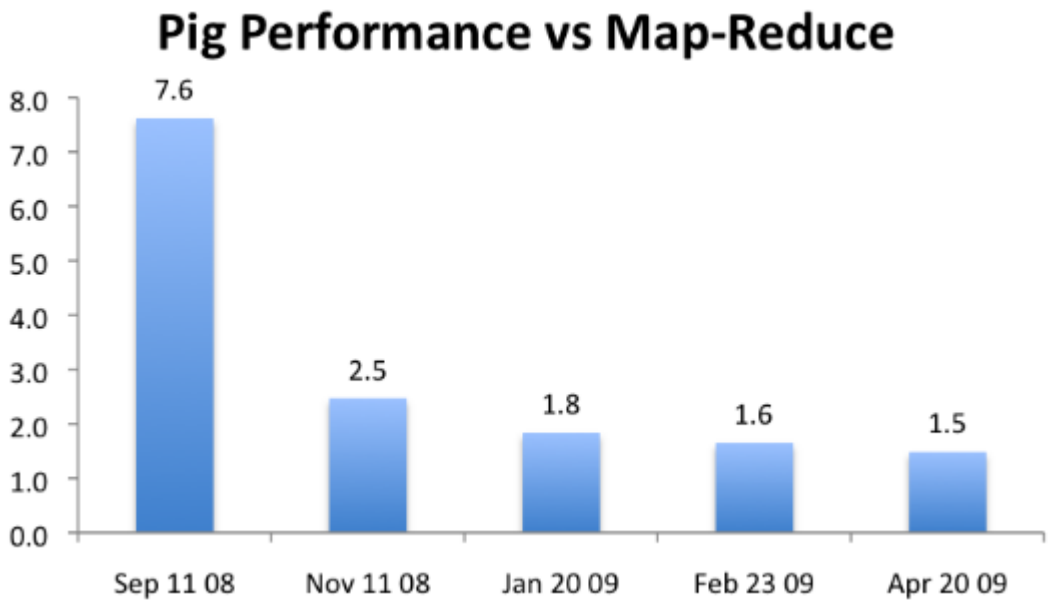
Two pipelines will be generated in this case

Some certain Pig operators can be invoked

# Generation, Optimization and Execution of Query Plans

- Memory Management
- UDF and Streaming
  - Pig allows users to incorporate custom code wherever necessary in the data processing pipeline
  - Streaming allows data to be pushed through external executables as part of a Pig data processing pipeline.

# Performance




*Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience*

By 2011 (r0.8.0), this score is around 0.9

# Future Work of Pig (VLDB09)

- Both rule and cost-based optimizations
- Non-Java UDFs
  - (already implemented: Python, JavaScript)
- SQL interface
- Grouping and joining of pre-partitioned/sorted data.
- Code generation
- Skew handling

# Other Optimization Approaches
(USENIX 2008)

- Logical optimization:
  - Early projection, early filtering, Operator rewrite
- Physical Optimizations
  - Optimization about *JOIN*
- Cross-program optimization

# Other Optimization Approaches

- Automatic Optimization for MapReduce Programs, J. Eaman, et.al, VLDB'11
  - Indexing, Selection, Projection, Data compression