DISTRIBUTED INFORMATION SYSTEMS

The Pig Experience: Building High-Level Data flows on top of Map-Reduce

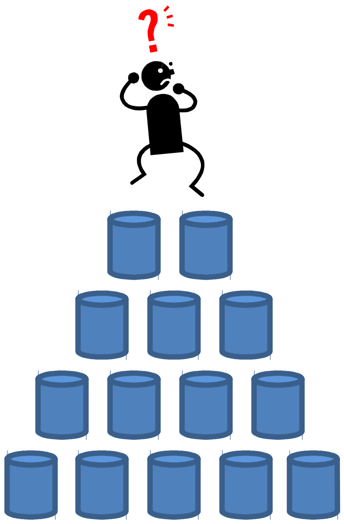
VLDB paper
Source: Javeria Iqbal, Martin Theobald

Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, Utkarsh Srivastava,
Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience, VLDB 2009.

Outline

- Map-Reduce and the need for Pig Latin
- Pig Latin
- Compilation into Map-Reduce
- Optimization
- Future Work

Data Processing Renaissance



- Internet companies swimming in data
  - TBs/day for Yahoo! Or Google!
  - PBs/day for FaceBook!
- Data analysis is “inner loop” of product innovation

Data Warehousing ...?

Scale

- High level declarative approach
- Little control over execution method

Price

Prohibitively expensive at web scale

- Up to $200K/TB

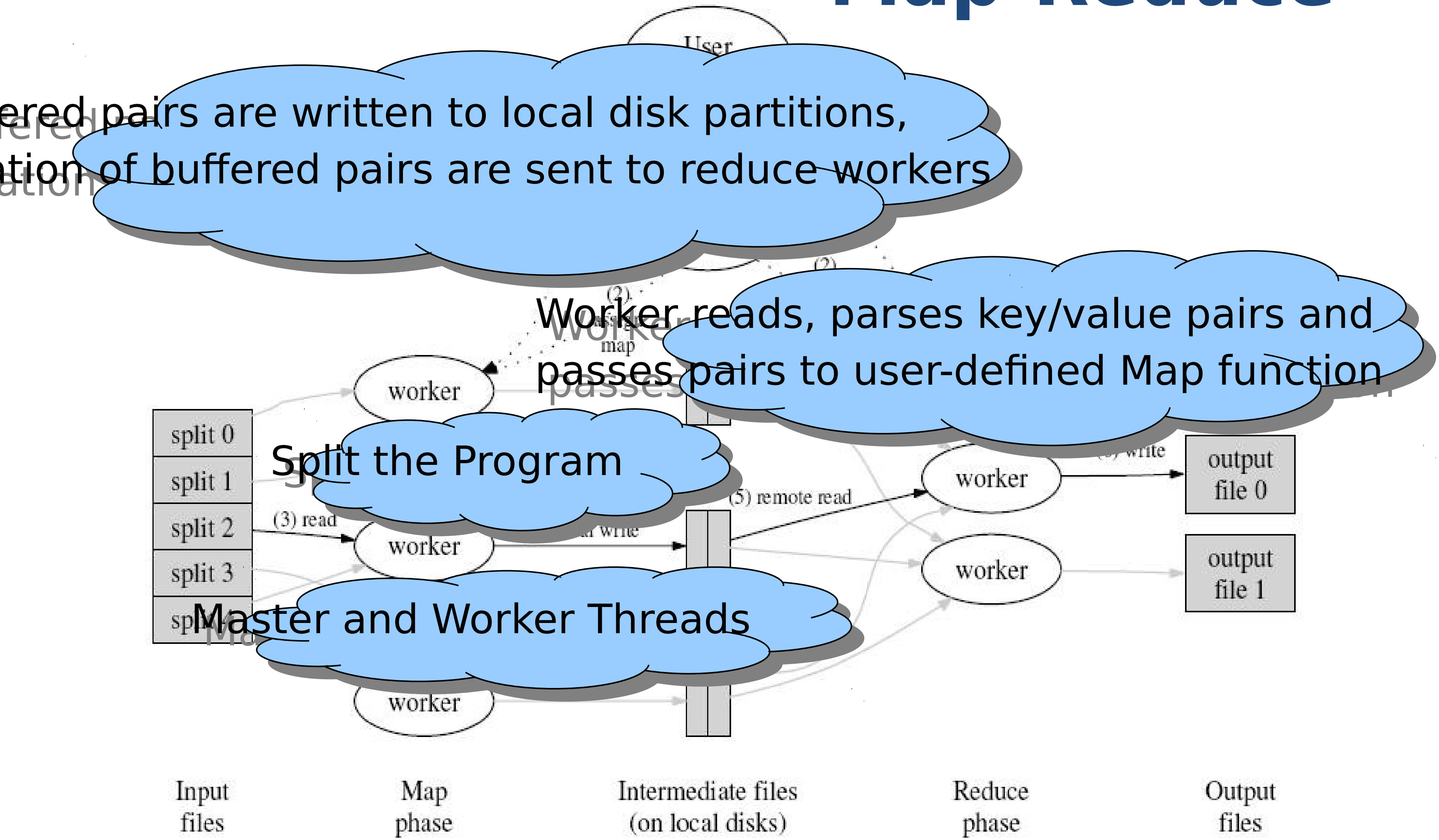
SQL

Often not scalable enough

Map-Reduce

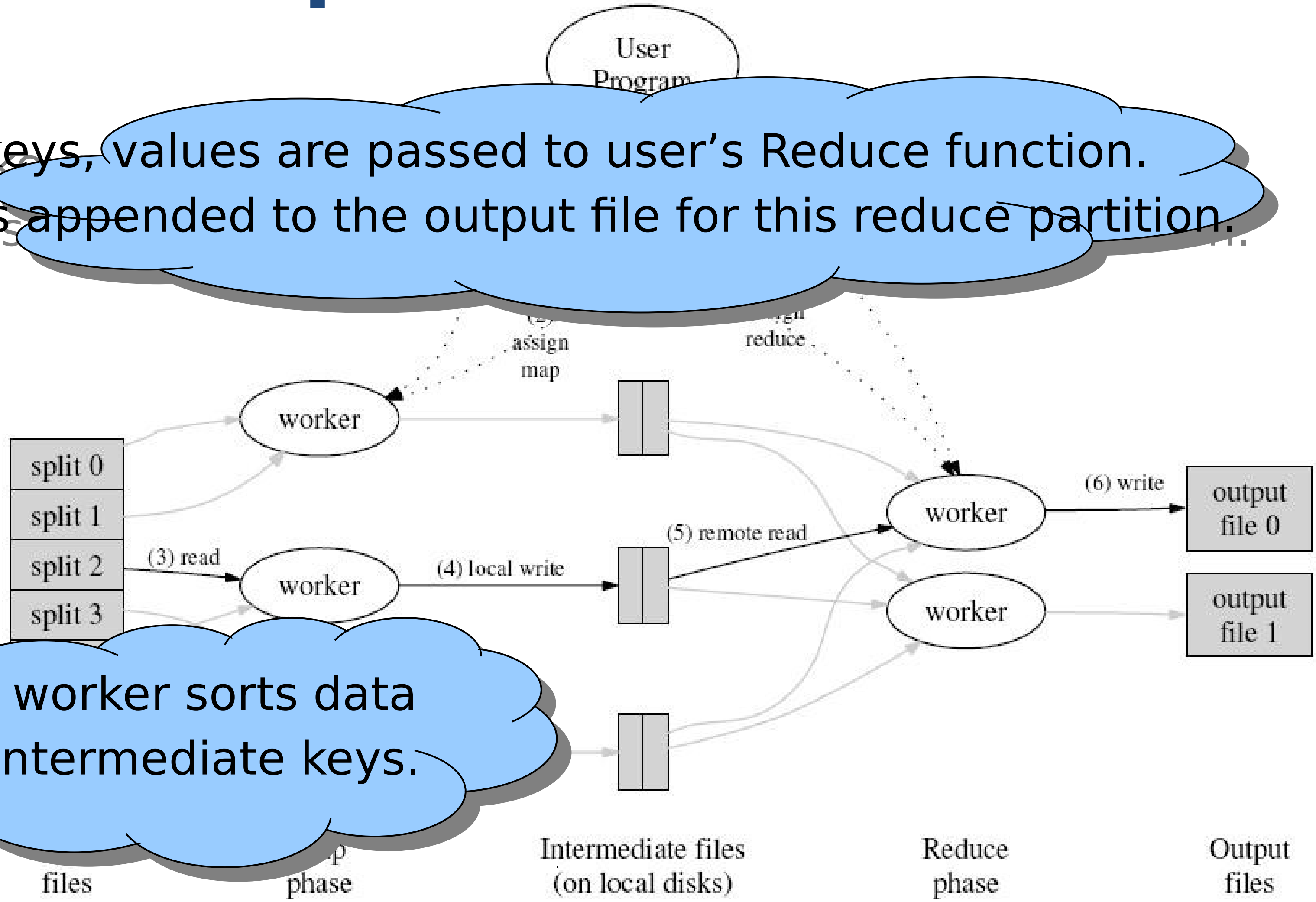
- Map : Performs filtering
- Reduce : Performs the aggregation
- These are two high level declarative primitives to enable parallel processing
- BUT no complex Database Operations
  e.g. Joins

Execution Overview of Map-Reduce



Execution Overview of Map-Reduce

nique keys, values are passed to user’s Reduce function.
utput is appended to the output file for this reduce partition.



Reduce worker sorts data
by the intermediate keys.

The Map-Reduce Appeal

Scale

- Scalable due to simpler design
- Explicit programming model
- Only parallelizable operations

Price

Runs on cheap commodity hardware

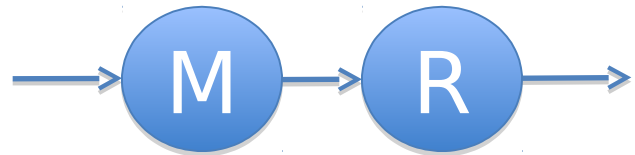
Less Administration

~~SQL~~

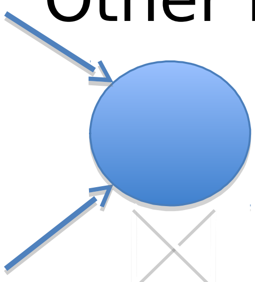
Procedural Control- a processing "pipe"

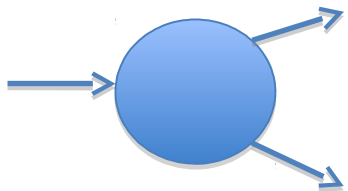
Disadvantages

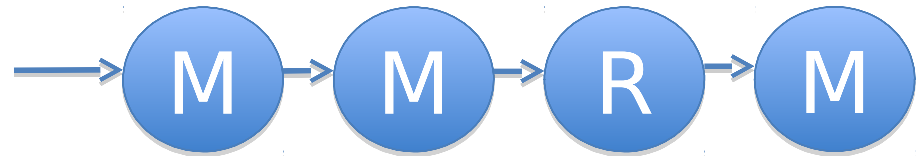
1. Extremely rigid data flow



Other flows hacked in







Join, Union

Spl it

Chains

Common operations must be coded by hand

Join, filter, projection, aggregates, sorting, distinct

Semantics hidden inside map-reduce functions

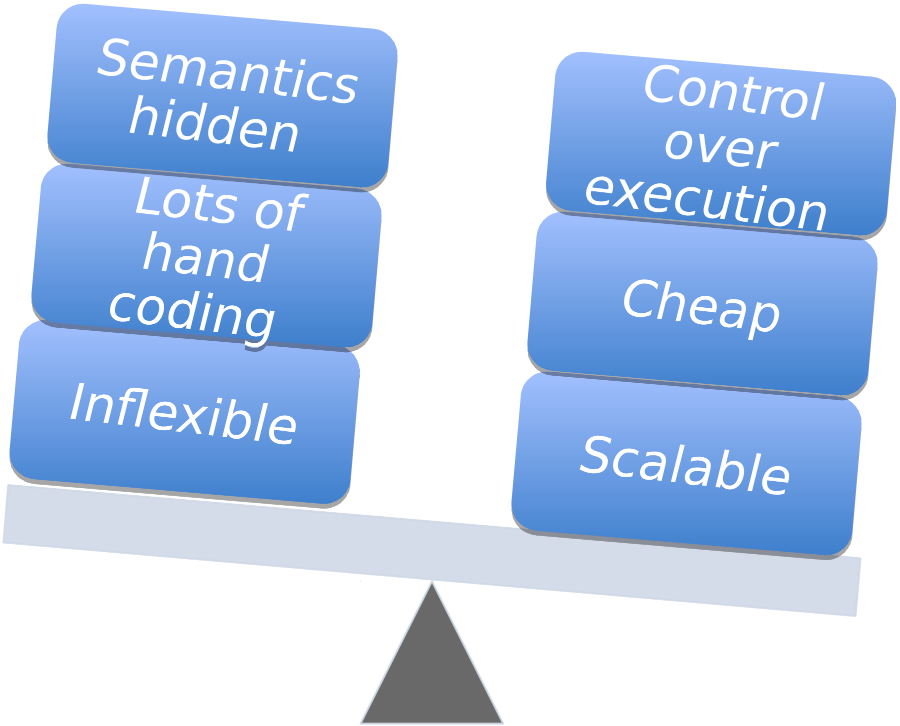
Difficult to maintain, extend, and optimize

No combined processing of multiple Datasets

Joins and other data processing operations

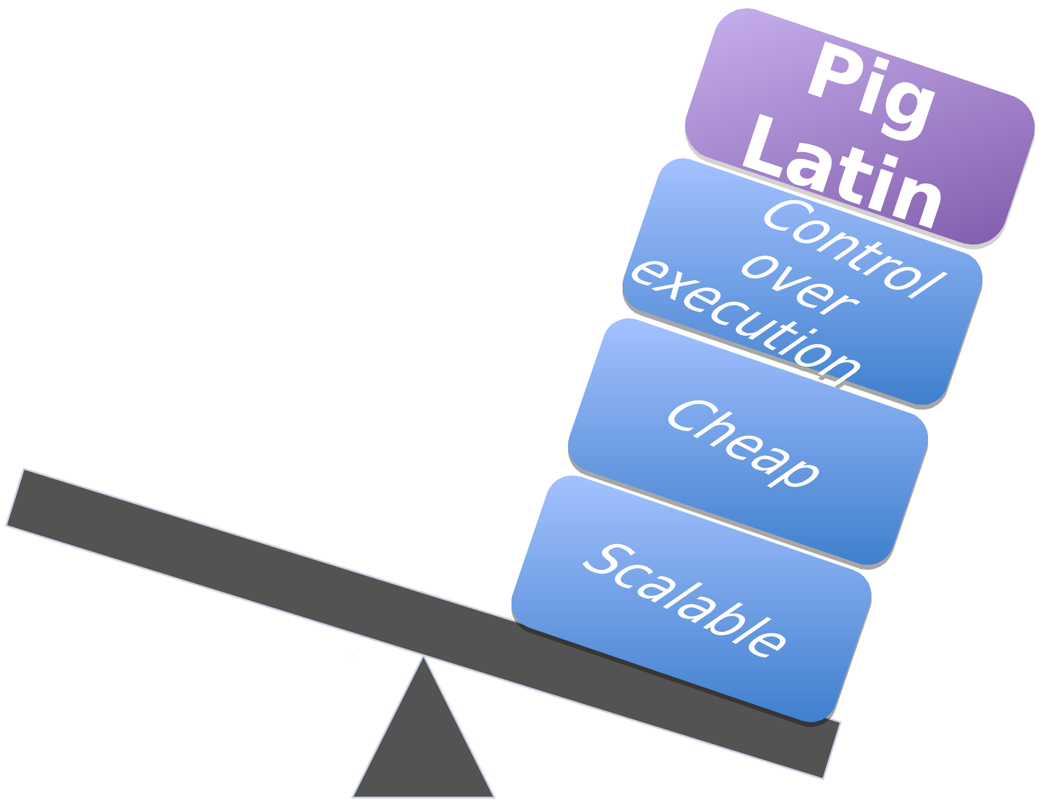
Motivation

Need a high-level, general data flow language



Enter Pig Latin

Need a high-level, general data flow language



Outline

- Map-Reduce and the need for Pig Latin
- Pig Latin
- Compilation into Map-Reduce
- Optimization
- Future Work

Pig Latin: Data Types

- Rich and Simple Data Model

Simple Types:

int, long, double, chararray, bytearray

Complex Types:

- Atom: String or Number e.g. (‘apple’)
- Tuple: Collection of fields e.g. (ápple’, ‘mango’)
- Bag: Collection of tuples

{ (‘apple’ , ‘mango’)

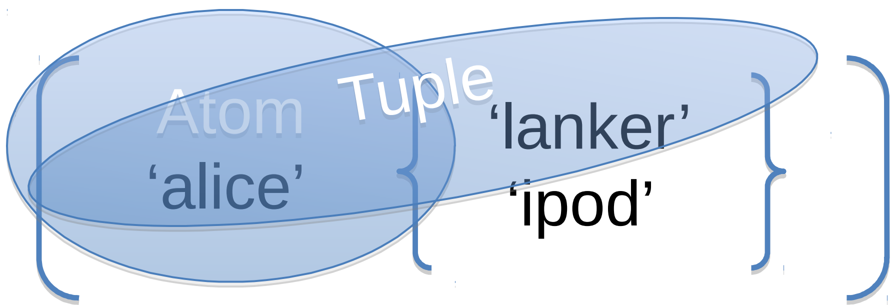
(ápple’, (‘red’ , ‘yellow’))

}

- Map: Key, Value Pair

Example: Data Model

- **Atom:** contains Single atomic value
- **Tuple:** sequence of fields
- **Bag:** collection of tuple with possible duplicates



Pig Latin: Input/Output Data

Input:

queries = LOAD `query_log.txt'

USING myLoad()

AS (userId, queryString, timestamp);

Output:

STORE query_revenues INTO `myoutput'

USING myStore();

Pig Latin: General Syntax

- Discarding Unwanted Data: FILTER
- Comparison operators such as ==, eq, !=, neq
- Logical connectors AND, OR, NOT

Pig Latin: Expression Table

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

Let fields of tuple t be called f1, f2, f3

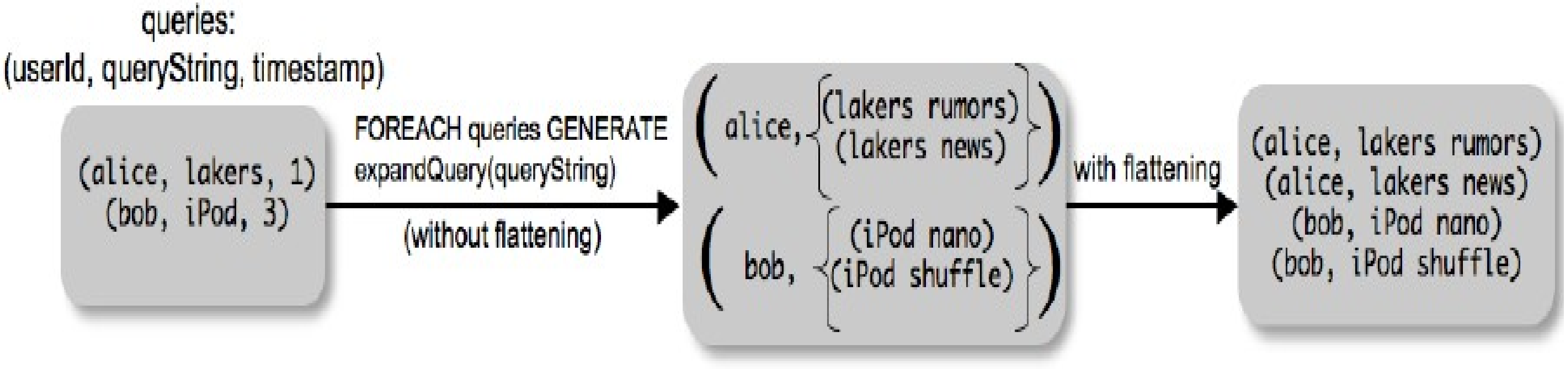
Expression Type	Example	Value for t
Constant	'bob'	Independent of t
Field by position	$0	'alice'
Field by name	f3	['age' → 20]
Projection	f2.$0	{ ('lakers') ('iPod') }
Map Lookup	f3#'age'	20
Function Evaluation	SUM(f2.$1)	1 + 2 = 3
Conditional Expression	f3#'age'>18? 'adult':'minor'	'adult'
Flattening	FLATTEN(f2)	'lakers', 1 'iPod', 2

Pig Latin: FOREACH with Flatten

expanded_queries = FOREACH queries GENERATE userId, expandQuery(queryString);

----

expanded_queries = FOREACH queries GENERATE userId,

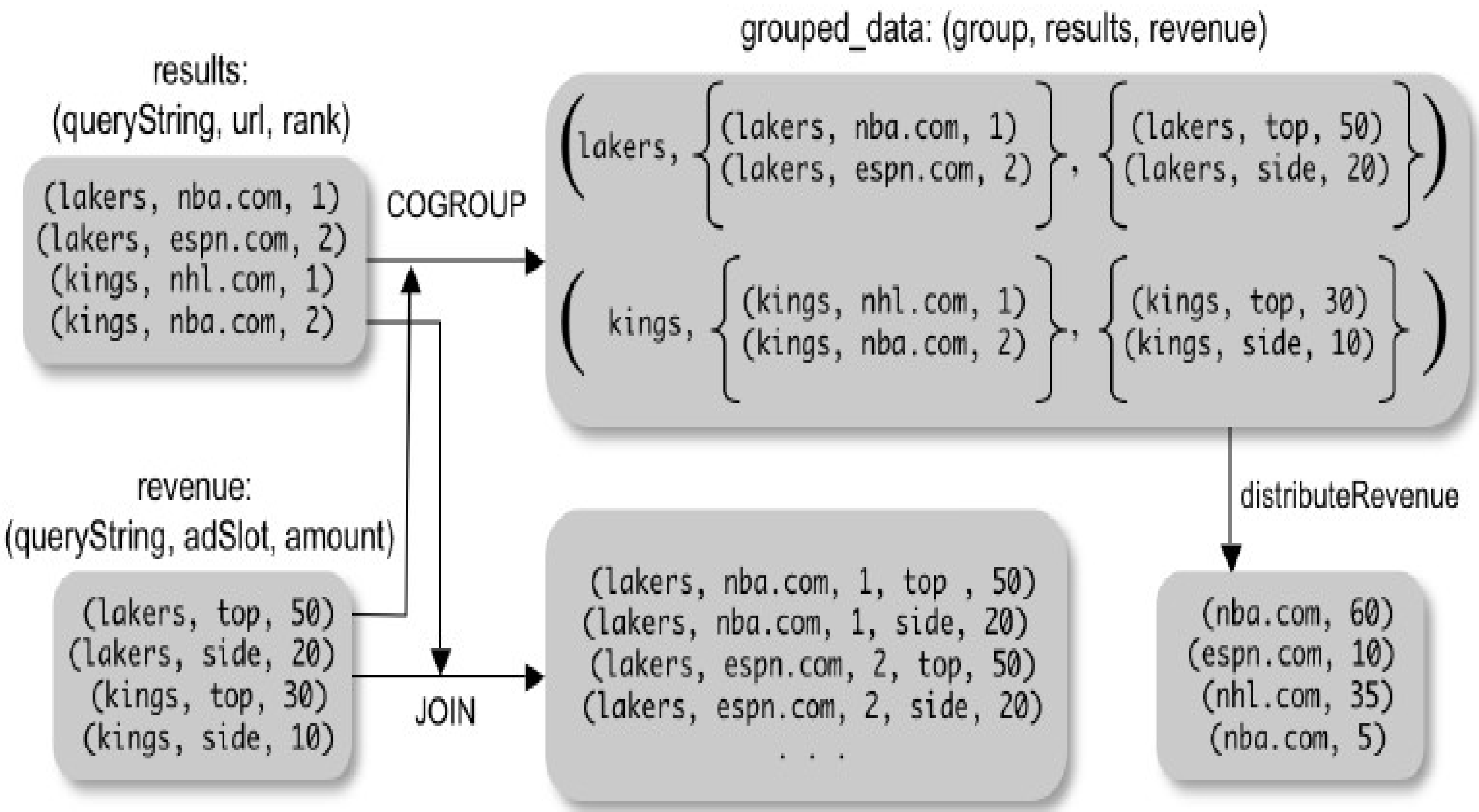


Pig Latin: COGROUP

- **Getting Related Data Together:**
  COGROUP
  Suppose we have two data sets
  result: (queryString, url, position)
  revenue: (queryString, adSlot, amount)

  grouped_data = COGROUP result BY queryString, revenue BY queryString;

Pig Latin: COGROUP vs. JOIN



Pig Latin: Map-Reduce

- Map-Reduce in Pig Latin

```
map_result = FOREACH input GENERATE
                 FLATTEN(map(*));

key_group =  GROUP map_result BY $0;

output = FOREACH key_group GENERATE
reduce(*);
```

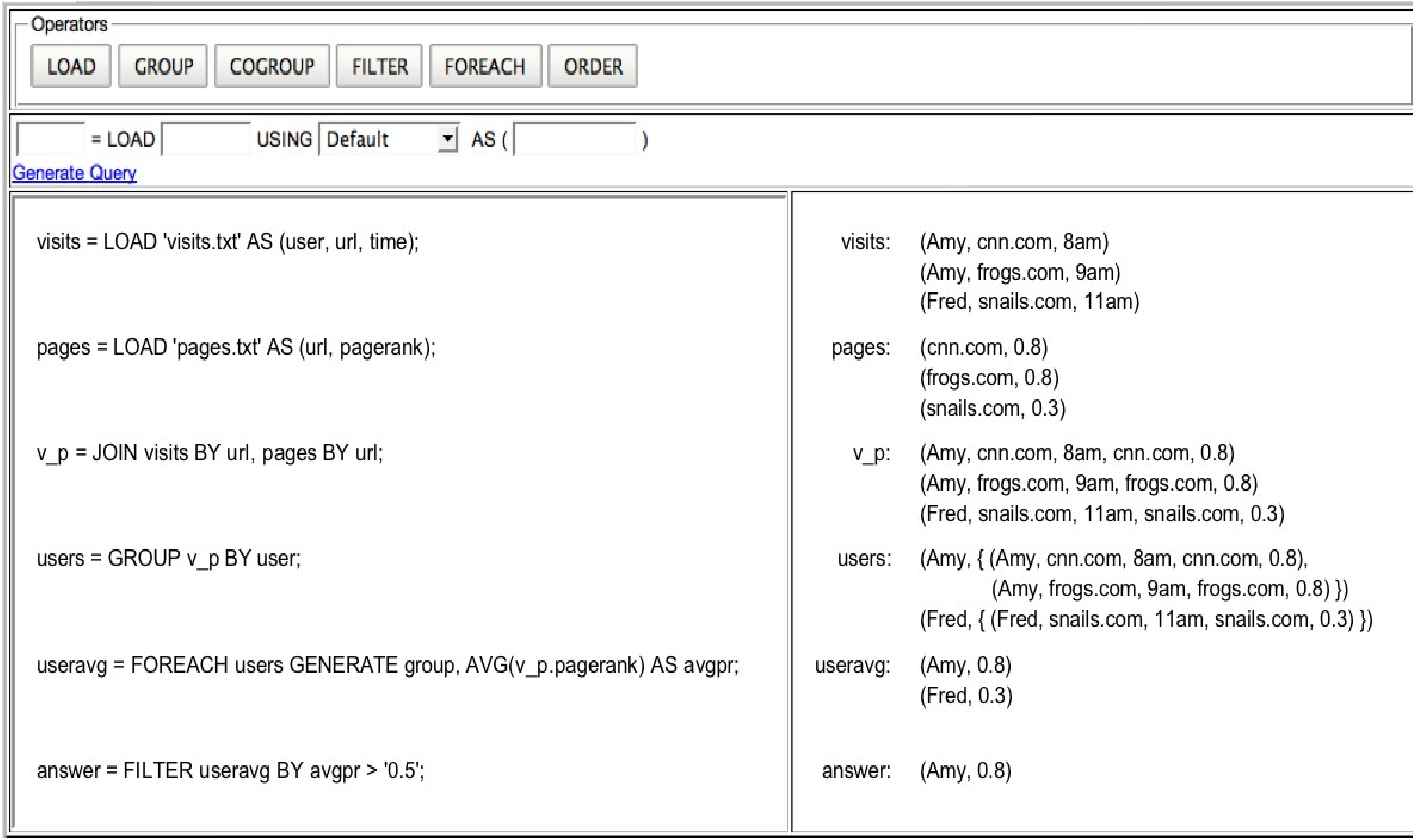
Pig Latin: Other Commands

- UNION : Returns the union of two or more bags
- CROSS: Returns the cross product
- ORDER: Orders a bag by the specified field(s)
- DISTINCT: Eliminates duplicate tuple in a bag

Pig Latin: Nested Operations

```
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);
};
```

Pig Pen: Screen Shot



Pig Latin: Example 1

Suppose we have a table

urls: (url, category, pagerank)

Simple SQL query that finds,

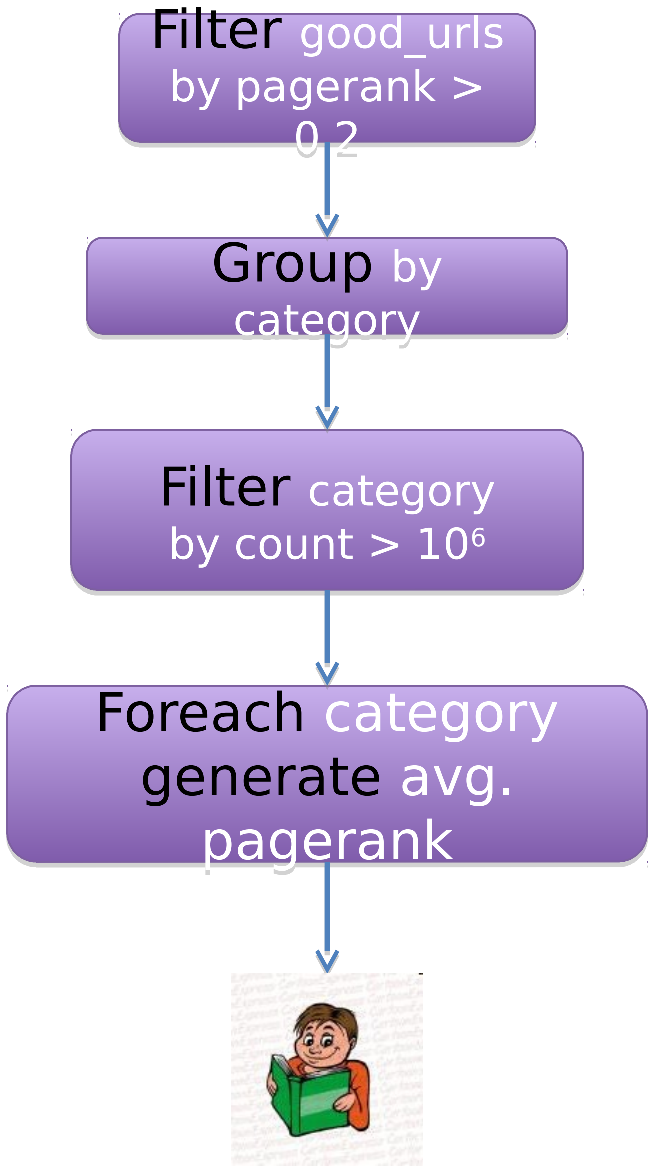
For each sufficiently large category, the average pagerank of high-pagerank urls in that category

SELECT category, Avg(pagetank)

FROM urls WHERE pagerank > 0.2

GROUP BY category HAVING COUNT(*) > 10^6

Data Flow



Filter good_urls by pagerank > 0.2
Group by category
Filter category by count > 10^6
Foreach category generate avg. pagerank

Equivalent 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);

Example 2: Data Analysis Task

d the top 10 most visited pages in each categ

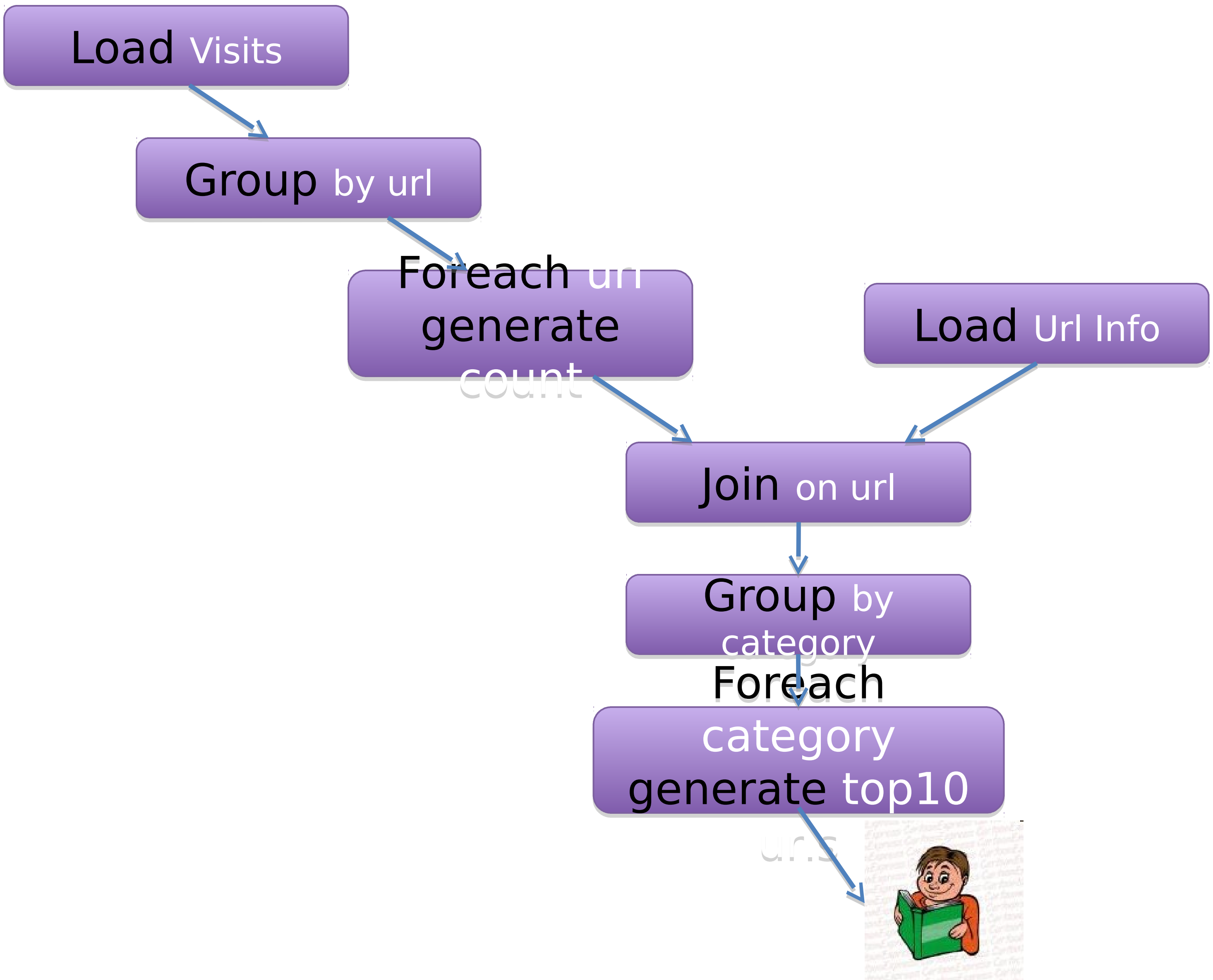
Visits

User	Url	Time
Amy	cnn.com	8:00
Amy	bbc.com	10:00
Amy	flickr.com	10:05
Fred	cnn.com	12:00

Url Info

Url	Category	PageRank
cnn.com	News	0.9
bbc.com	News	0.8
flickr.com	Photos	0.7
espn.com	Sports	0.9

Data Flow


Load Visits
Group by url
Foreach url generate count
Load Url Info
Join on url
Group by category
Foreach category generate top10 urls

Equivalent Pig Latin

```
visits           = load '/data/visits' as (user, url,
   time);
gVisits          = group visits by url;
visitCounts  = foreach gVisits generate url,
   count(visits);

urlInfo          = load '/data/urlInfo' as (url,
   category, pRank);
visitCounts  = join visitCounts by url, urlInfo by
   url;

gCategories = group visitCounts by category;
topUrls = foreach gCategories generate
```

Quick Start and Interoperability

```
visits              = load ‘/data/visits’ as (user, url,
   time);
gVisits            = group visits by url;
visitCounts  = foreach gVisits generate url,
   count(urlVisits);

urlInfo            = load ‘/data/urlInfo’ as (url,
   catego

visitCoun                                   l, urlInfo by
   url;
gCategories = group visitCounts by category;
topUrls = foreach gCategories generate
   top(visitCounts,10);
```

Operates directly over files

Quick Start and Interoperability

visits = load '/data/visits' as (user, url, time);

gVisits = group visits by url;

visitCounts = foreach gVisits generate url, count(urlVisits);

urlInfo = load '/data/urlInfo' as (url,
catego

Schemas optional;
Can be assigned dynamically

visitCoun urlInfo by url;

gCategories = group visitCounts by category;

topUrls = foreach gCategories generate

User-Code as a First-Class Citizen

visits ... s (user, url,
time...
gVisits...
visitC... erate url,
cou...

User-defined functions (UDFs) can be used in every construct

- Load, Store
- Group, Filter, Foreach

```
urlInfo        = load '/data/urlInfo' as (url,
   category, pRank);

visitCounts  = join visitCounts by url, urlInfo by
   url;
gCategories = group visitCounts by category;
topUrls = foreach gCategories generate
```

Nested Data Model

- Pig Latin has a fully nested data model with:
  - Atomic values, tuples, bags (lists), and maps

finance
yahoo , email
news

- Avoids expensive joins

Nested Data Model

Decouples grouping as an independent operati

User	Url	Time
Amy	cnn.com	8:00
Amy	bbc.com	10:00
Amy	bbc.com	10:05
Fred	cnn.com	12:00

group by url

group	Visits		
cnn.com	Amy	cnn.com	8:00
	Fred	cnn.com	12:00
	Amy	bbc.com	10:00
			10:05

- Com
  sets
- Pow
  sequ
- Efficient Implementation (see paper)

I frankly like pig much better than SQL in some respects (group + optional flatten), I love nested data structures).”

Ted Dunning
Chief Scientist, Veoh

CoGroup

results

query	url	rank
Lakers	nba.com	1
Lakers	espn.com	2
Kings	nhl.com	1
Kings	nba.com	2

revenue

query	adSlot	amount
Lakers	top	50
Lakers	side	20
Kings	top	30
Kings	side	10

group	results			revenue		
Lakers	Lakers	nba.com	1	Lakers	top	50
	Lakers	espn.com	2	Lakers	side	20
Kings	Kings	nhl.com	1	Kings	top	30
	Kings	nba.com	2	Kings	side	10

Cross-product of the 2 bags would give natural join

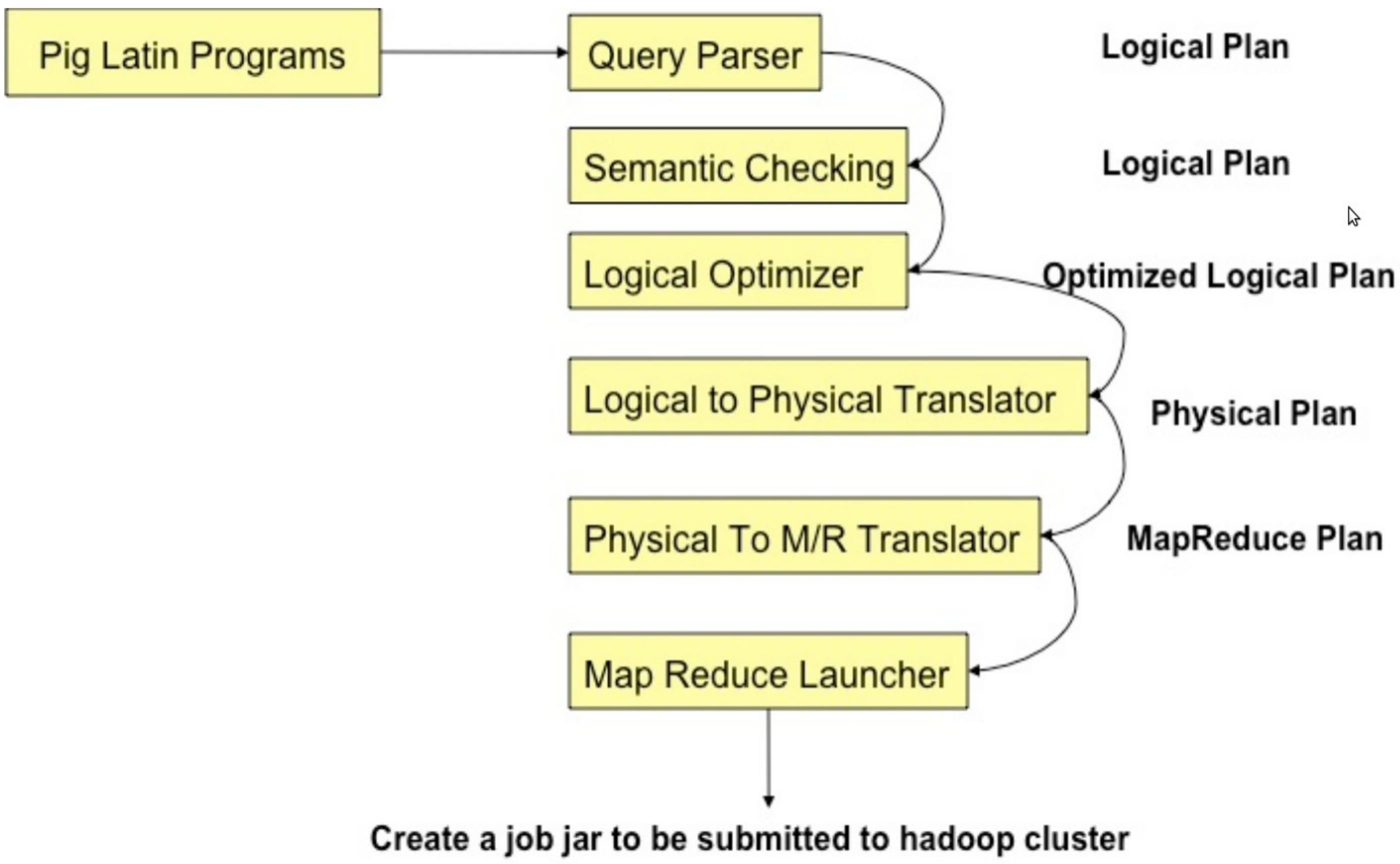
Pig Features

- Explicit Data Flow Language unlike SQL
- Low Level Procedural Language unlike Map-Reduce
- Quick Start & Interoperability
- Mode (Interactive Mode, Batch, Embedded)
- User Defined Functions
- Nested Data Model

Outline

- Map-Reduce and the need for Pig Latin
- Pig Latin
- Compilation into Map-Reduce
- Optimization
- Future Work

Compilation



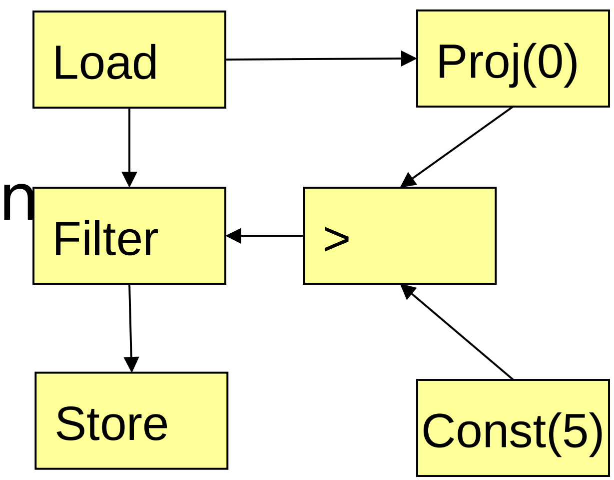
Parsing

- Type checking with schema
- References verifying
- Logic plan generating
  - One-to-one fashion
  - Independent of execution platform
  - Limited optimization

Logical Plan

- Consists of DAG of Logical Operators as nodes and Data Flow represented as edges
  - Logical Operators contain list of i/p’s o/p’s and schema
- Logical operators
  - Aid in post parse stage checking (type checking)
  - Optimization
  - Translation to Physical Plan

```
a = load ‘myfile’;
b = filter a by $0 > 5;
store b into ‘myfilteredfile’;
```

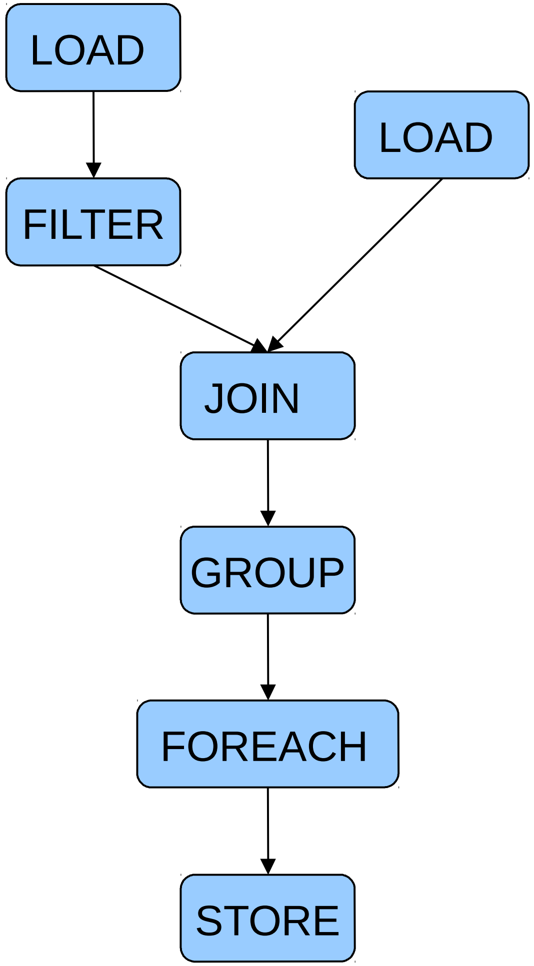


Physical Plan

- Layer to map Logical Plan to multiple back-ends, one such being M/R (Map Reduce)
  - Chance for code re-use if multiple back-ends share same operator
- Consists of operators which pig will run on the backend
- Currently most of the physical plan is placed as operators in the map reduce plan
- Logical to Physical Translation
  - 1:1 correspondence for most Logical operators
  - Except Cross, Distinct, Group, Co-group and Order

Logic Plan

```
A=LOAD 'file1' AS (x, y, z);

B=LOAD 'file2' AS (t, u, v);

C=FILTER A by y > 0;

D=JOIN C BY x, B BY u;

E=GROUP D BY z;

F=FOREACH E GENERATE
        group, COUNT(D);

STORE F INTO 'output';
```

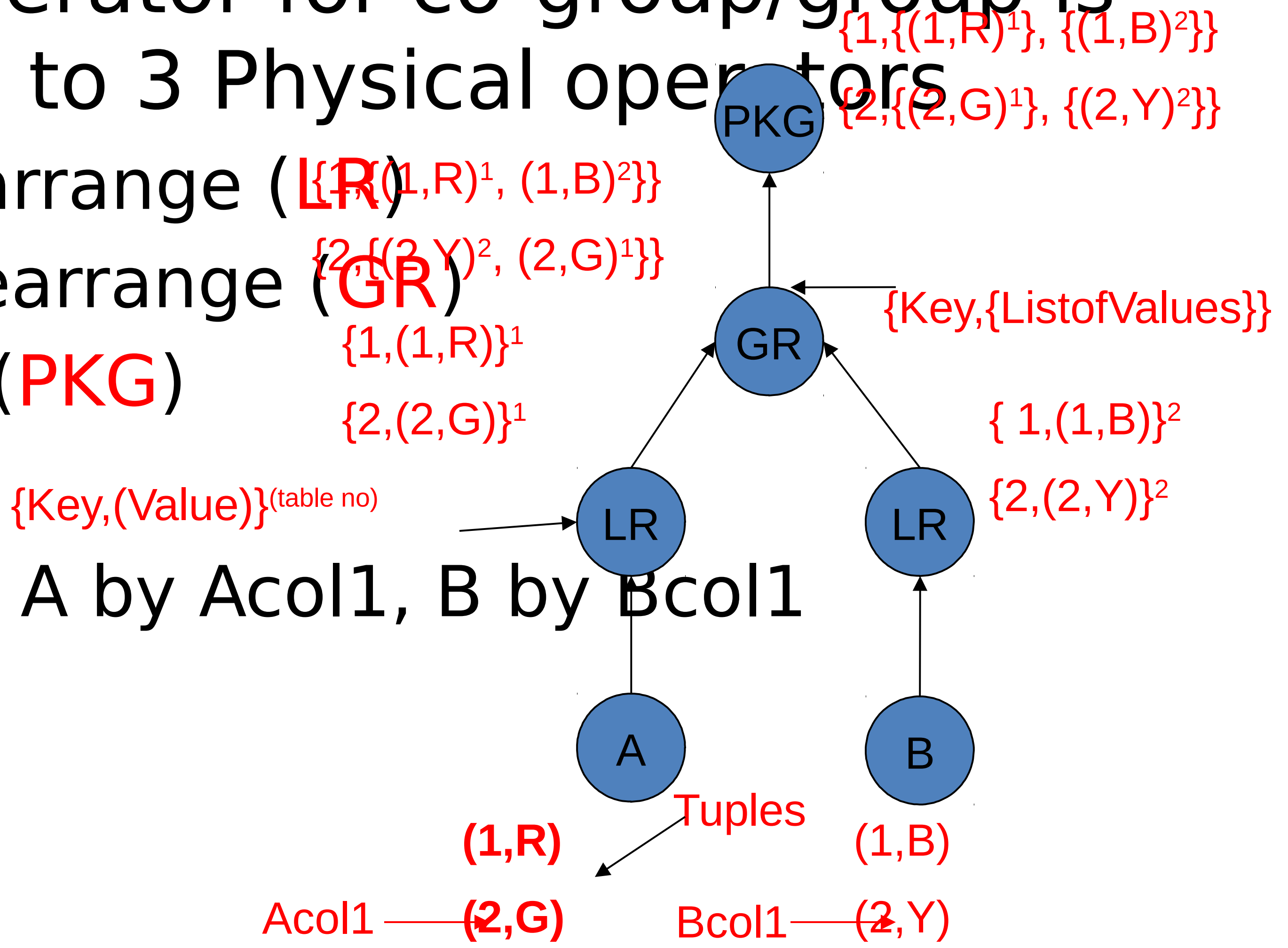


Physical Plan

- 1:1 correspondence with most logical operators
- Except for:
  - DISTINCT
  - (CO)GROUP
  - JOIN
  - ORDER

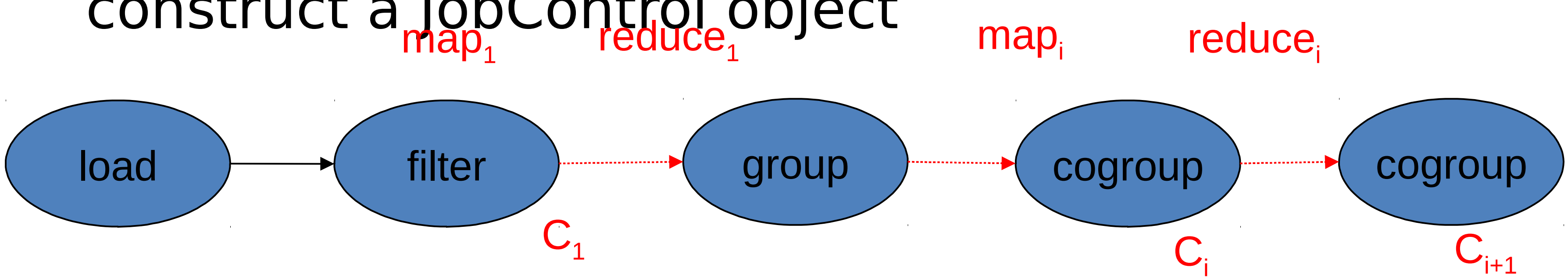
Logical to Physical Plan for Group operator

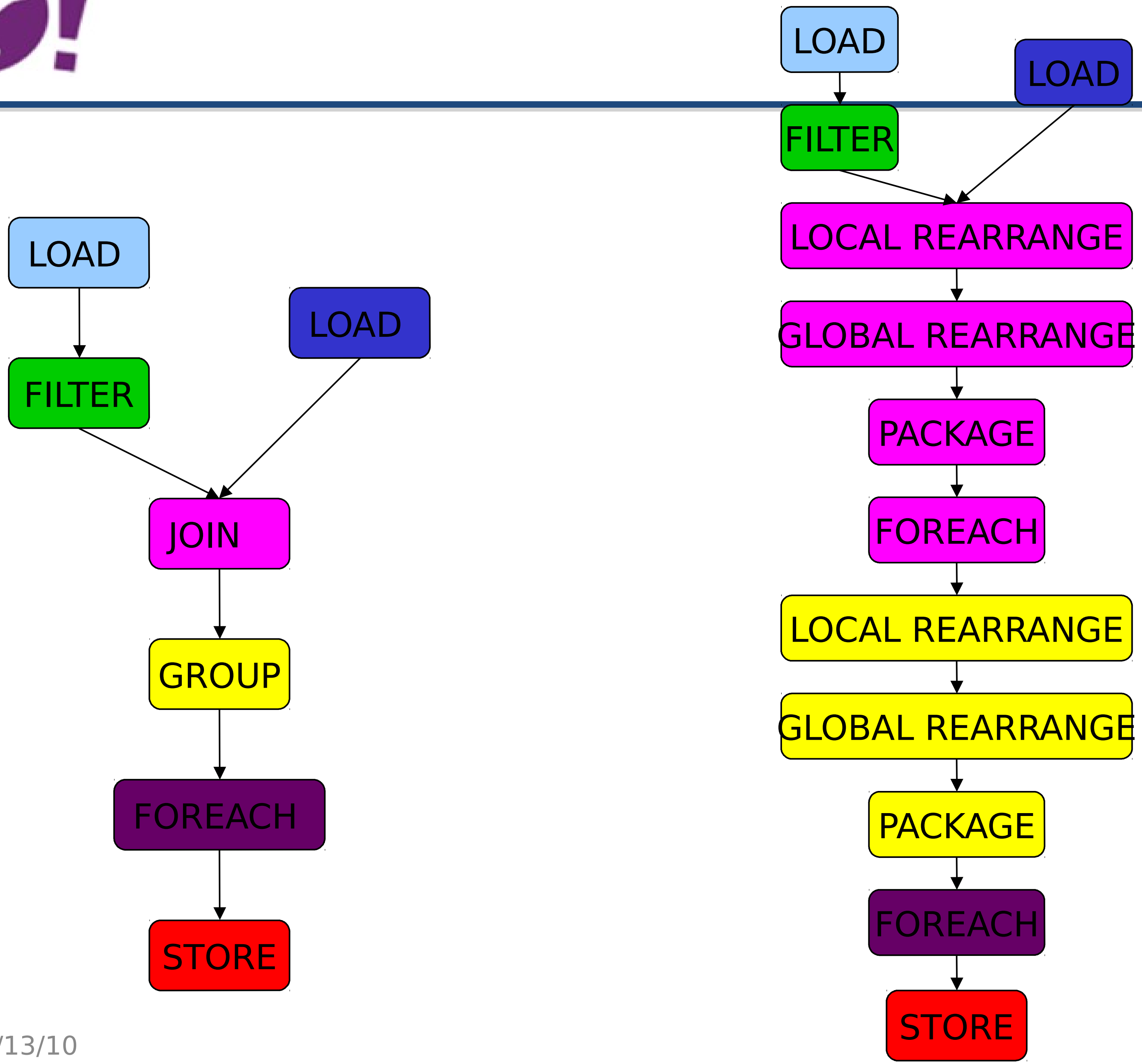
- Logical operator for co-group/group is converted to 3 Physical operators
  - Local Rearrange (LR)
  - Global Rearrange (GR)
  - Package (PKG)
- Example:
  - Co-group A by Acol1, B by Bcol1



Map Reduce Plan

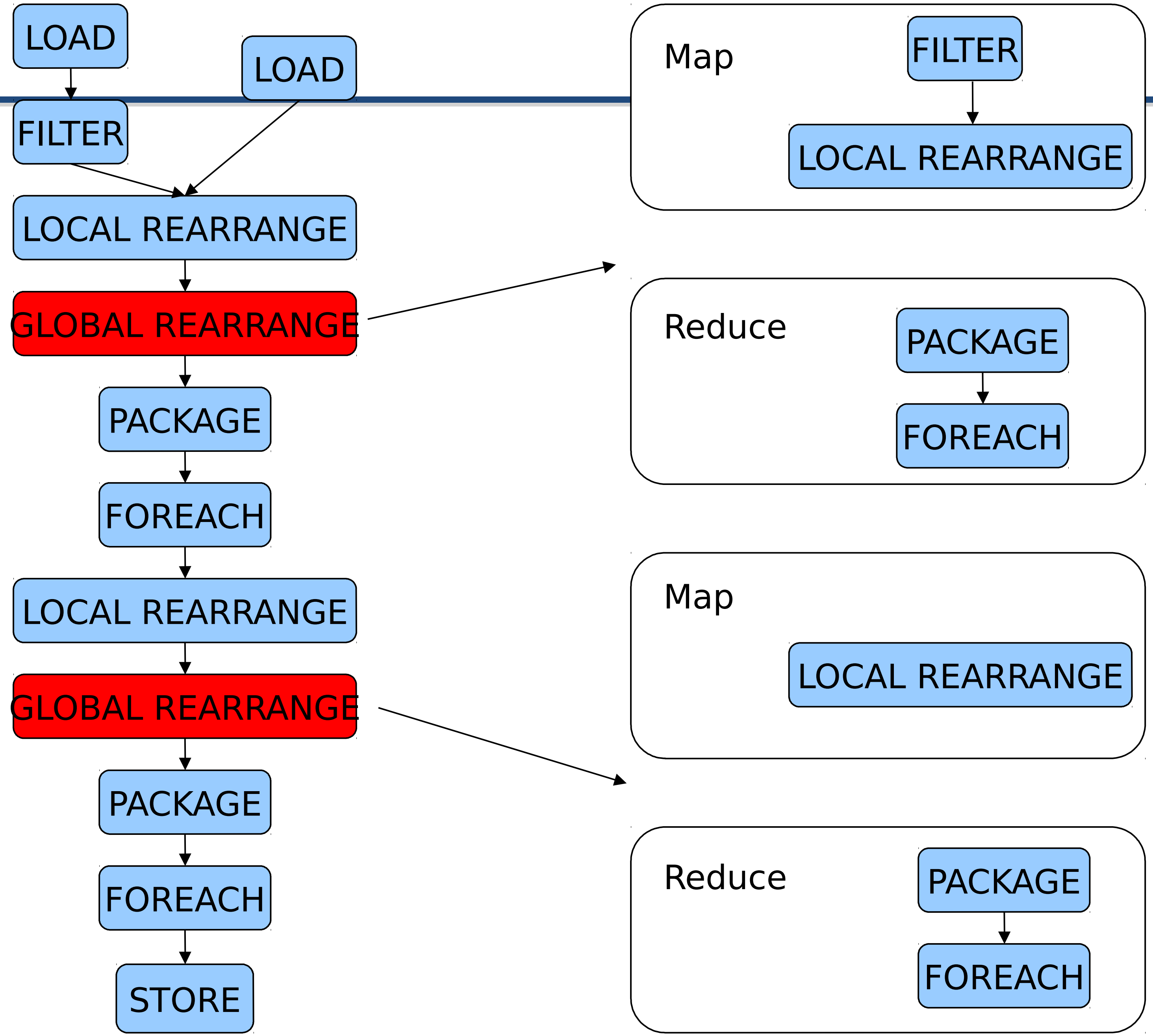
- Physical to Map Reduce (M/R) Plan conversion happens through the MRCompiler
  - Converts a physical plan into a DAG of M/R operators
- Boundaries for M/R include cogroup/group, distinct, cross, order by, limit (in some cases)
  - Push all subsequent operators between cogroup to next cogroup into reduce
  - order by is implemented as 2 M/R jobs
- JobControlCompiler then uses the M/R plan to construct a JobControl object




LOAD
LOAD
FILTER
JOIN
GROUP
FOREACH
STORE
LOAD
LOAD
FILTER
LOCAL REARRANGE
GLOBAL REARRANGE
PACKAGE
FOREACH
LOCAL REARRANGE
GLOBAL REARRANGE
PACKAGE
FOREACH
STORE

MapReduce Plan

- Determine MapReduce boundaries
  - GLOBAL REARRANGE
- Some operations are done by MapReduce framework
- Coalesce other operators into Map & Reduce stages
- Generate job jar file


LOAD
LOAD
FILTER
LOCAL REARRANGE
GLOBAL REARRANGE
PACKAGE
FOREACH
LOCAL REARRANGE
GLOBAL REARRANGE
PACKAGE
FOREACH
STORE
Map
FILTER
LOCAL REARRANGE
Reduce
PACKAGE
FOREACH
Map
LOCAL REARRANGE
Reduce
PACKAGE
FOREACH

Pig Latin to Physical Plan

A = LOAD ‘file1’ AS (x,y,z);

B = LOAD ‘file2’ AS (t,u,v);

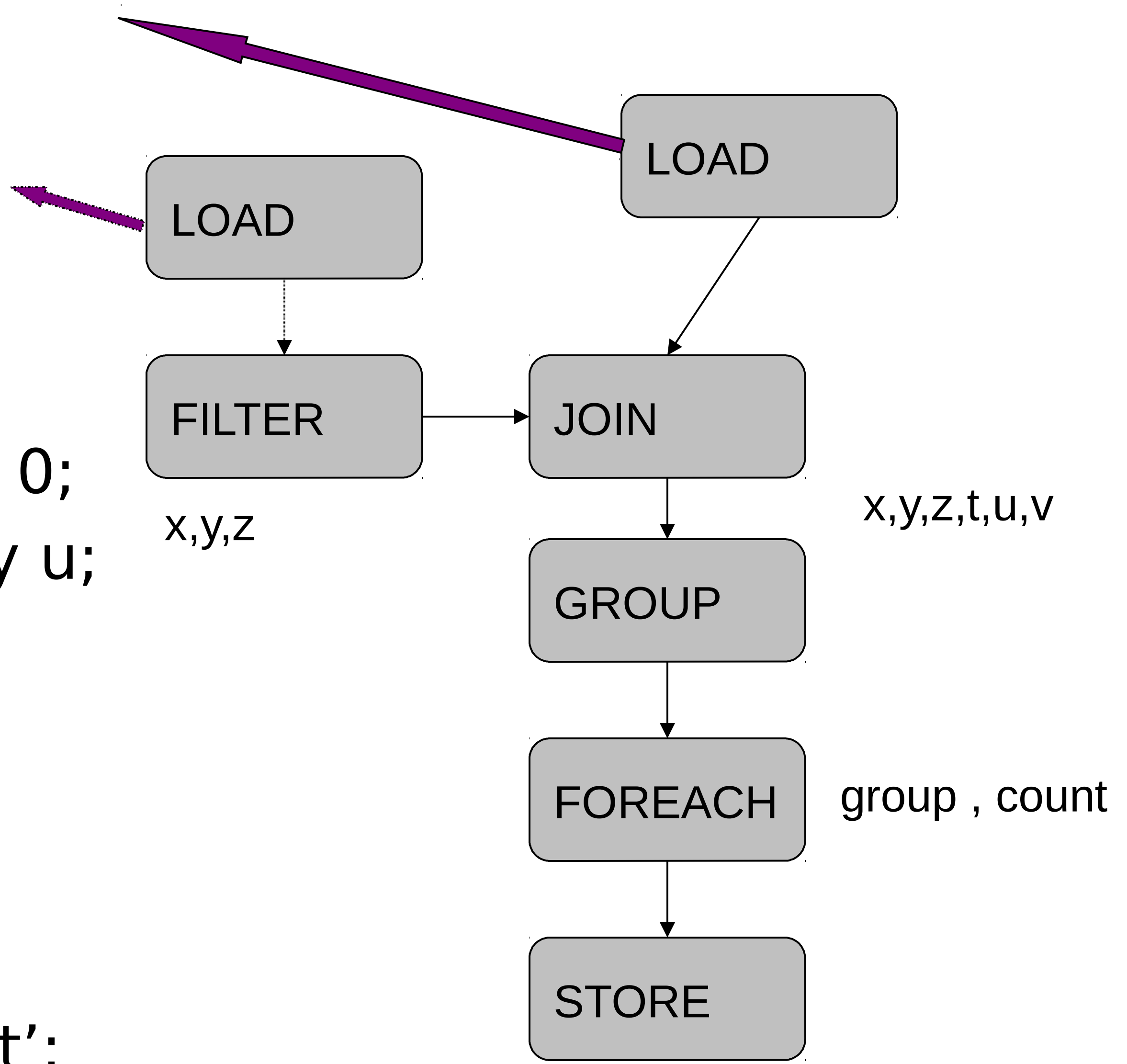
C = FILTER A by y > 0;

D = JOIN C by x,B by u;

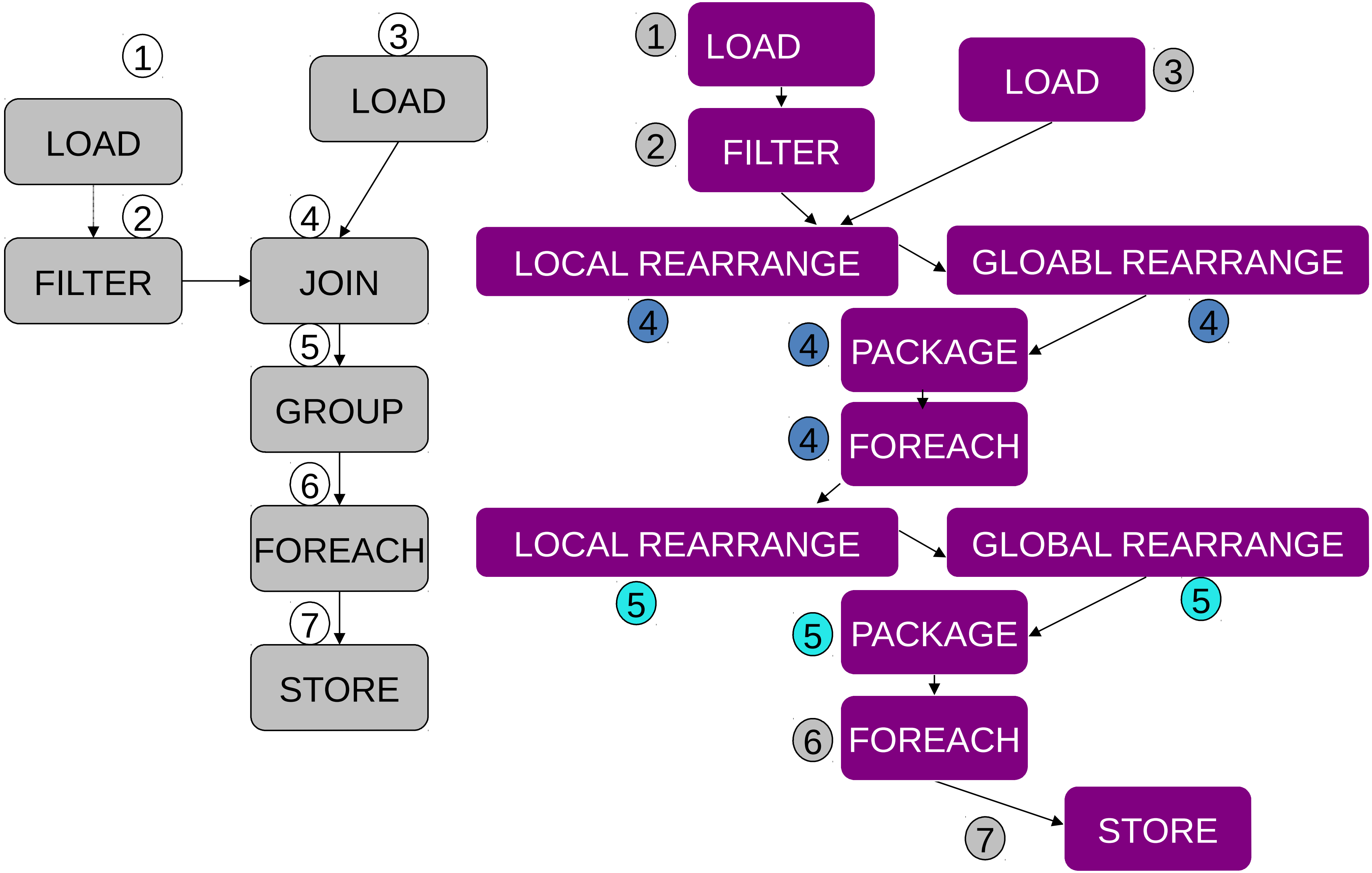
E = GROUP D by z;

F = FOREACH E generate group, COUNT(D);

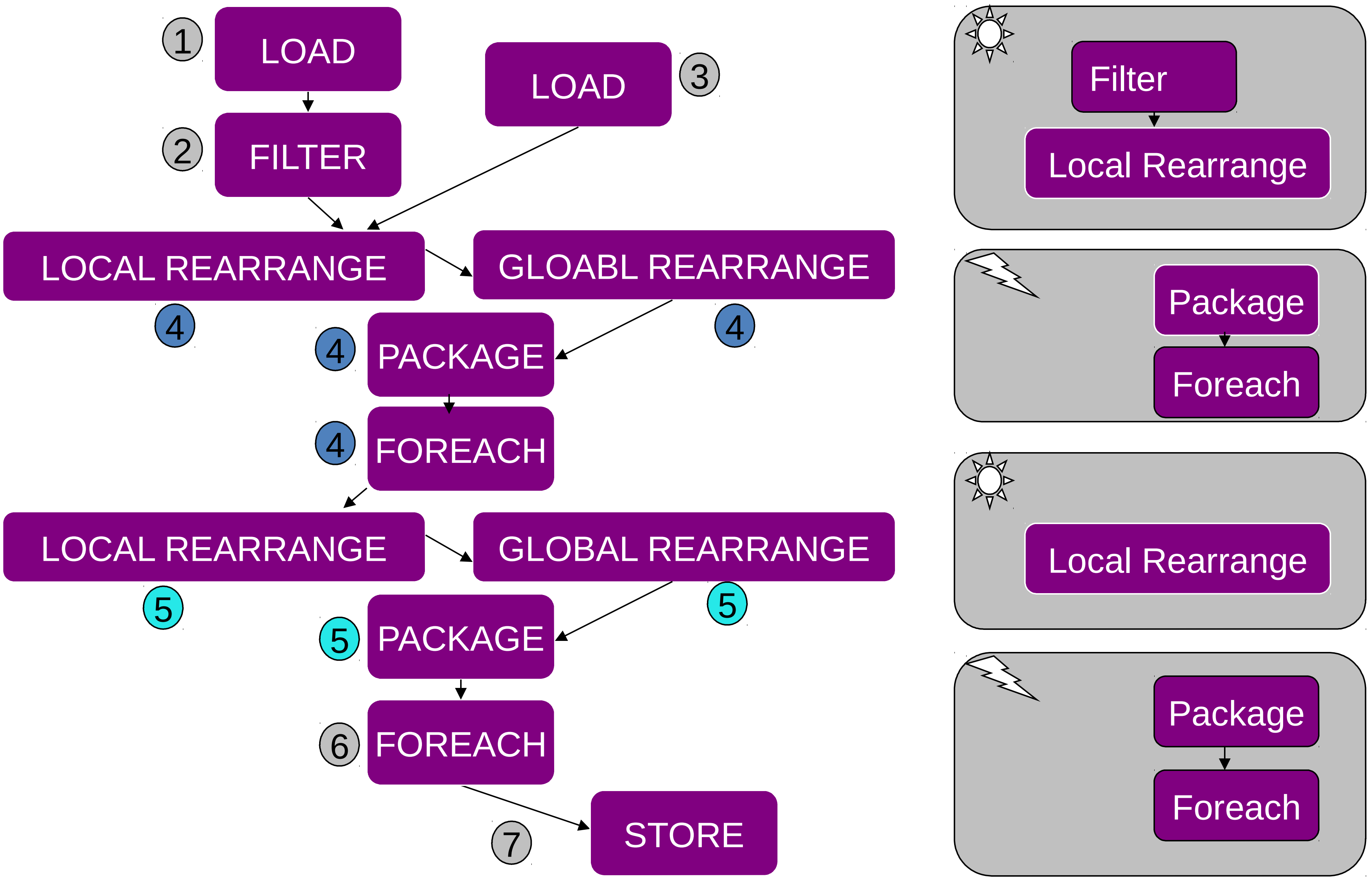
STORE F into ‘output’;



Logical Plan to Physical Plan



Physical Plan to Map-Reduce Plan



Implementation

SQL

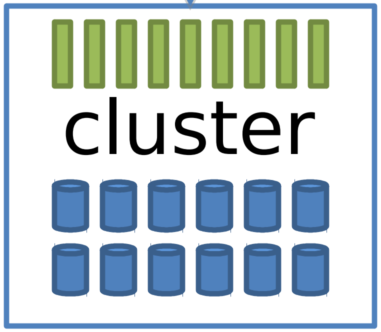
user

automatic rewrite + optimize

Pig is open-source.

http://hadoop.apache.org/pig

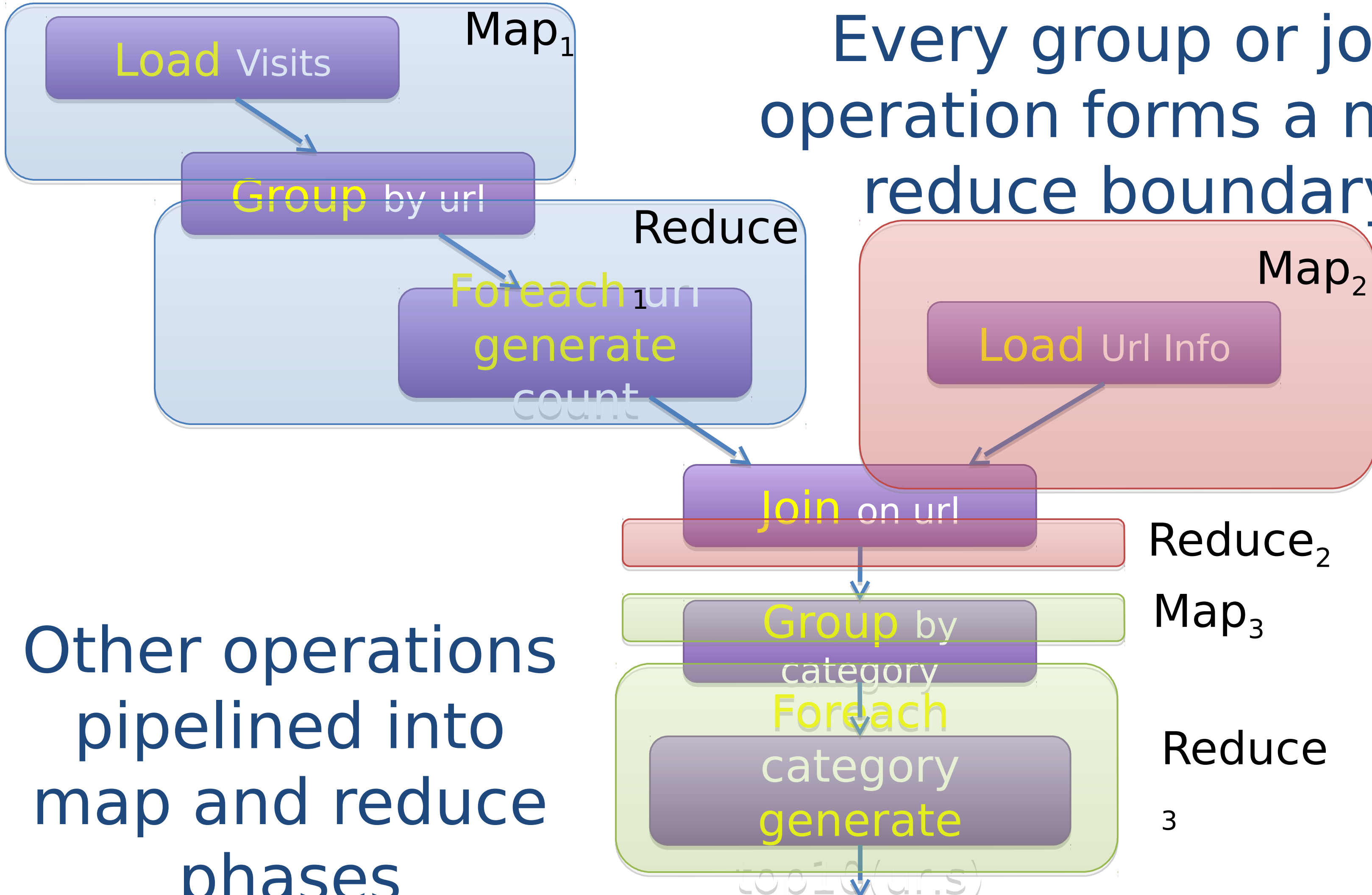
Reduce



- ~50% of Hadoop jobs at Yahoo! are Pig
- 1000s of jobs per day

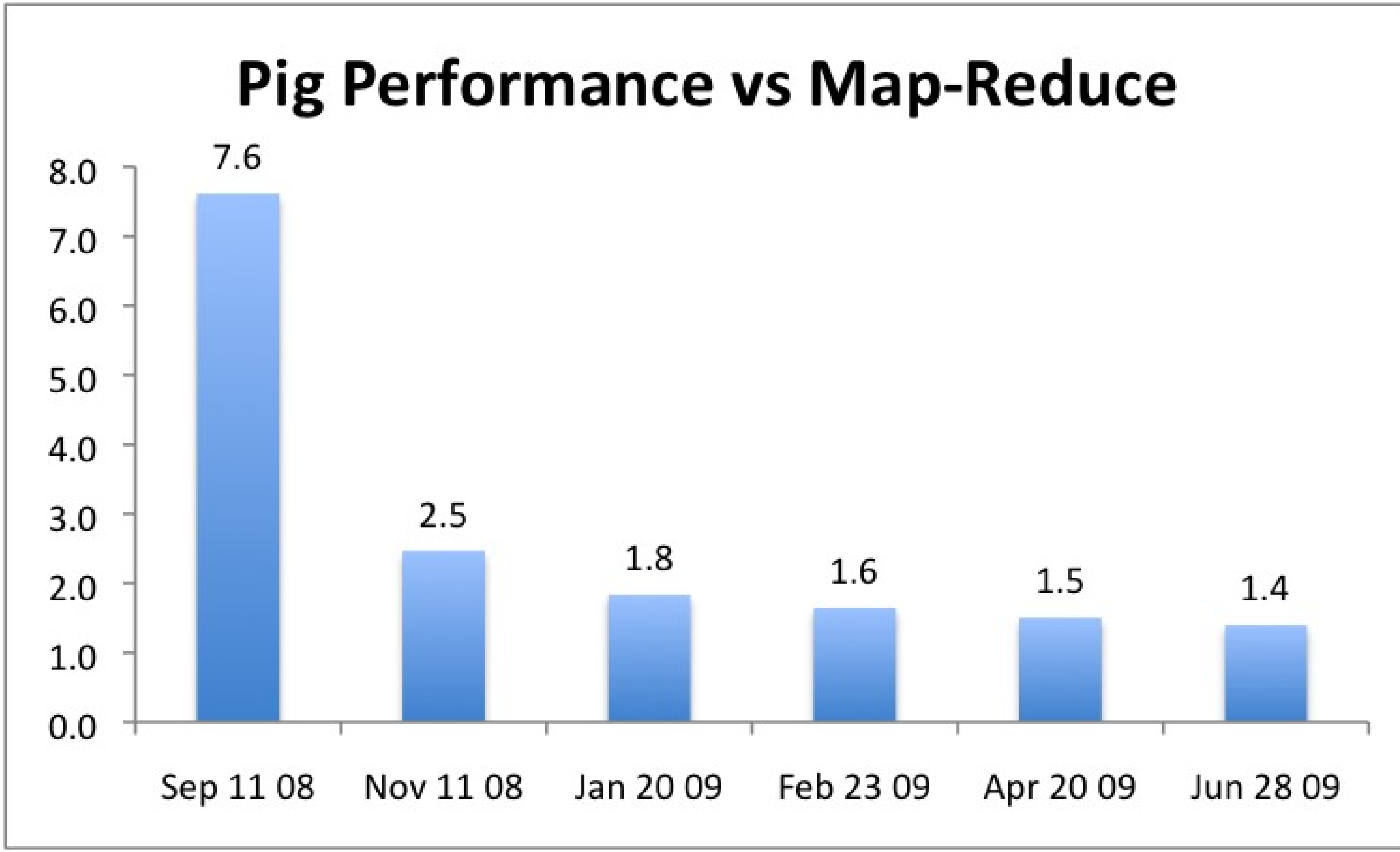
Compilation into Map-Reduce

Every group or join operation forms a map-reduce boundary

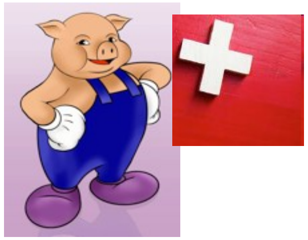


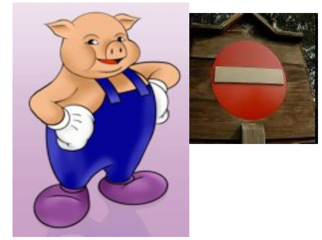
Other operations pipelined into map and reduce phases

Performance



Strong & Weak Points





The step-by-step method of creating a program in Pig is much cleaner and simpler to use than the single block method of SQL. It is easier to keep track of what your variables are, and where you are in the process of analyzing your data.

Jasmine Novak
Engineer, Yahoo.

- Scalability
- Processing
- Open Source

- Memory
- nt
- on for Non Java Users
- Limited Optimization

With the various interleaved clauses in SQL
It is difficult to know what is actually happening sequentially.

David Ciemiewicz
Search Excellence, Yaho

Summary

- Big demand for parallel data processing
  - Emerging tools that do not look like SQL DBMS
  - Programmers like dataflow pipes over static files
- Hence the excitement about Map-

Pig Latin

Sweet spot between map-reduce and SQL

- But, Map-Reduce is too low-level and rigid