Let’s see how we can create complex MapReduce workflows by programming in a high-level language.

The Pig System

- Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins: Pig Latin: a not-so-foreign language for data processing. SIGMOD Conference 2008: 1099-1110
- Several slides courtesy Chris Olston and Utkarsh Srivastava
- Open source project under the Apache Hadoop umbrella

Overview

- Design goal: find sweet spot between declarative style of SQL and low-level procedural style of MapReduce
- Programmer creates Pig Latin program, using high-level operators
- Pig Latin program is compiled to MapReduce program to run on Hadoop

Why Not SQL or Plain MapReduce?

- SQL difficult to use and debug for many programmers
- Programmer might not trust automatic optimizer and prefers to hard-code best query plan
- Plain MapReduce lacks convenience of readily available, reusable data manipulation operators like selection, projection, join, sort
- Program semantics hidden in “opaque” Java code
  – More difficult to optimize and maintain

Example Data Analysis Task

Find the top 10 most visited pages in each category

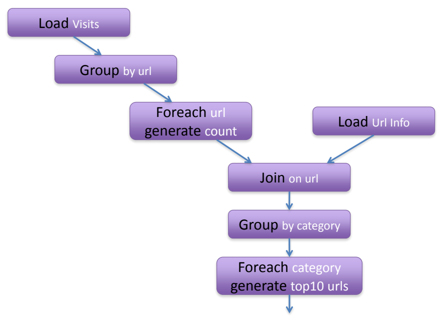
Visits

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



In Pig Latin

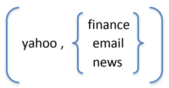
```
visits       = load '/data/visits' as (user, url, time);
gVisits      = group visits by url;
visitCounts  = foreach gVisits generate url, count(visits);

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 top(visitCounts,10);

store topUrls into '/data/topUrls';
```

Pig Latin Notes

- No need to import data into database
  - Pig Latin works directly with files
- Schemas are optional and can be assigned dynamically
  - Load '/data/visits' as (user, url, time);
- Can call user-defined functions in every construct like Load, Store, Group, Filter, Foreach
  - Foreach gCategories generate top(visitCounts,10);

Pig Latin Data Model

- Fully-nestable data model with:
  - Atomic values, tuples, bags (lists), and maps



- More natural to programmers than flat tuples
  - Can flatten nested structures using FLATTEN
- Avoids expensive joins, but more complex to process

Pig Latin Operators: LOAD

- Reads data from file and optionally assigns schema to each record
- Can use custom deserializer

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

Pig Latin Operators: FOREACH

- Applies processing to each record of a data set
- No dependence between the processing of different records
  - Allows efficient parallel implementation
- GENERATE creates output records for a given input record

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

Pig Latin Operators: FILTER

- Remove records that do not pass filter condition
- Can use user-defined function in filter condition

```
real_queries =
      FILTER queries BY userId neq `bot‘;
```

Pig Latin Operators: COGROUP

- Group together records from one or more data sets

results

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

revenue

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

COGROUP results BY queryString, revenue BY queryString

```
( 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)  }
```

Pig Latin Operators: GROUP

- Special case of COGROUP, to group single data set by selected fields
- Similar to GROUP BY in SQL, but does not need to apply aggregate function to records in each group

grouped_revenue = GROUP revenue BY queryString;

Pig Latin Operators: JOIN

- Computes equi-join

join_result = JOIN results BY queryString, revenue BY queryString;

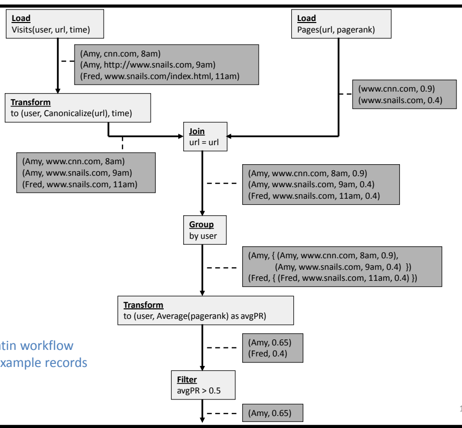
- Just a syntactic shorthand for COGROUP followed by flattening

temp_var = COGROUP results BY queryString, revenue BY queryString;

join_result = FOREACH temp_var GENERATE FLATTEN(results), FLATTEN(revenue);

Other Pig Latin Operators

- UNION: union of two or more bags
- CROSS: cross product of two or more bags
- ORDER: orders a bag by the specified field(s)
- DISTINCT: eliminates duplicate records in bag
- STORE: saves results to a file
- Nested bags within records can be processed by nesting operators within a FOREACH operator


Load
Visits(user, url, time)
(Amy, cnn.com, 8am)
(Amy, http://www.snails.com, 9am)
(Fred, www.snails.com/index.html, 11am)
Load
Pages(url, pagerank)
(www.cnn.com, 0.9)
(www.snails.com, 0.4)
Transform
to (user, Canonicalize(url), time)
(Amy, www.cnn.com, 8am)
(Amy, www.snails.com, 9am)
(Fred, www.snails.com, 11am)
Join
url = url
(Amy, www.cnn.com, 8am, 0.9)
(Amy, www.snails.com, 9am, 0.4)
(Fred, www.snails.com, 11am, 0.4)
Group
by user
(Amy, { (Amy, www.cnn.com, 8am, 0.9),
(Amy, www.snails.com, 9am, 0.4) })
(Fred, { (Fred, www.snails.com, 11am, 0.4) })
Transform
to (user, Average(pagerank) as avgPR)
(Amy, 0.65)
(Fred, 0.4)
Filter
avgPR > 0.5
(Amy, 0.65)

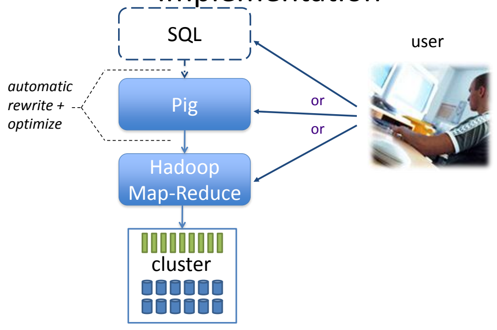
Pig Latin workflow and example records

MapReduce in Pig Latin

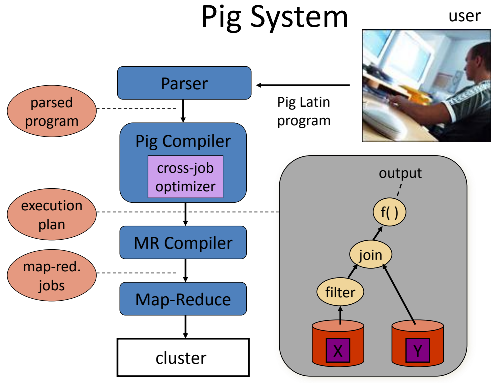
map_result = FOREACH input GENERATE
    FLATTEN(map(*));
key_groups = GROUP map_result BY $0;
output = FOREACH key_groups GENERATE reduce(*);

- Map() is a UDF, where * indicates that the entire input record is passed to map()
- $0 refers to first field, i.e., the intermediate key here
- Reduce() is another UDF

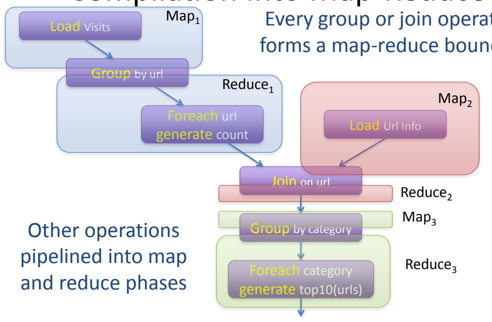
Implementation



Pig System



Compilation into Map-Reduce



Every group or join operation forms a map-reduce boundary

Other operations pipelined into map and reduce phases

Is Pig a DBMS?

	DBMS	Pig
workload	Bulk and random reads & writes; indexes, transactions	Bulk reads & writes only
data representation	System controls data format Must pre-declare schema	Pigs eat anything
programming style	System of constraints	Sequence of steps
customizable processing	Custom functions second-class to logic expressions	Easy to incorporate custom functions