Pig Programming

12-2pm

Agenda

- **Introduction to PIG**
  - Overview
  - Architecture
- **Pig Latin**
- **Pig Examples**
  - Writing your own UDF's
- **Pig Streaming and Parameter Substitution**
- **Translating SQL Queries into Pig Latin**

What is PIG?

- **SQL-like language (PIG Latin) for processing large semi-structured data sets using Hadoop M/R**
  - PIG is an Hadoop (Apache) sub-project
  - ~30 % Grid users in Yahoo! use PIG
- **PIG Latin**
  - PIG is a procedural (data flow) language:
    - Lets users to specify explicit sequence of steps for data processing
    - Think of it as: a scripting harness to chain together multiple map-reduce jobs
  - Supports relational model:
    - But NOT strongly typed
    - Supports primitive relational operators like FILTER, PROJECTIONS, JOIN, GROUPING, etc.

Why use Pig over Hadoop

Hadoop is great for distributed computation but..

1. Several simple, frequently-used operations

Filtering, projecting, grouping, aggregation, joining, & sorting are commonly functions

2. Users' end-goal typically requires several Map-Reduce jobs

This makes it hard to maintain, extend, debug, optimize

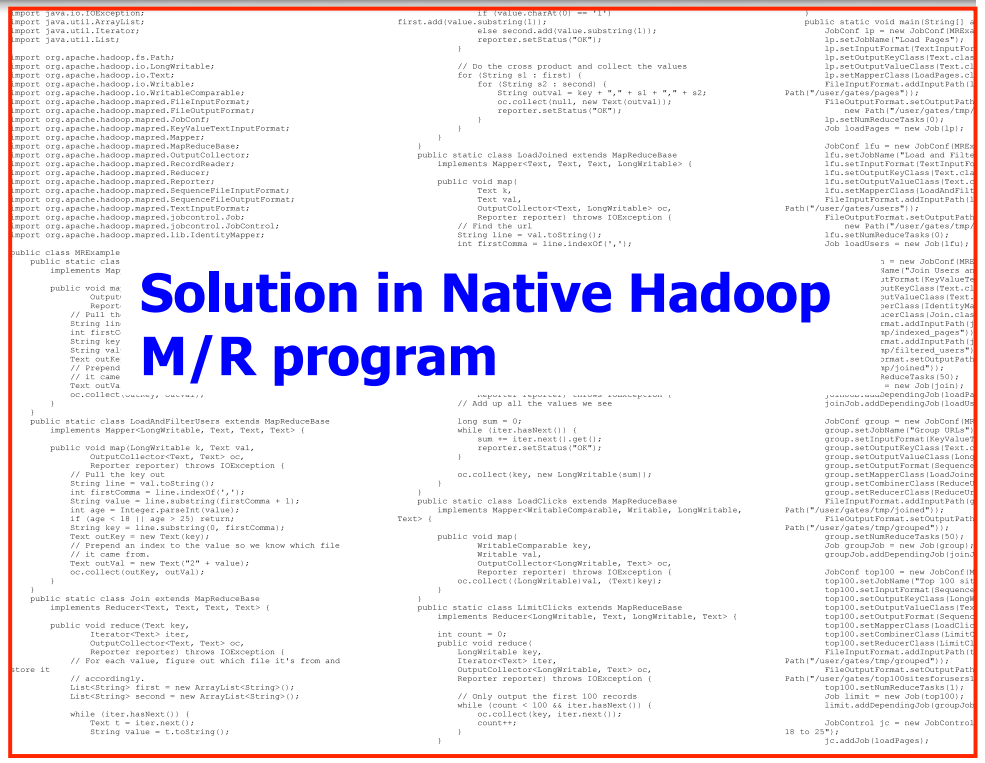
Advantages of Using PIG

- **PIG can be treated as a higher-level language**
  - Increases programmer productivity
    - smaller learning curve
  - Decreases duplication of effort
  - Opens the M/R (Hadoop) programming system to more users

- **PIG insulates against Hadoop complexity**
  - Hadoop version upgrades
  - JobConf configuration tuning

Pig: Making Hadoop Easy

- How to find the top 100 most visited sites by users aged 18 to 25??
- Using Native Hadoop Java code requires 20x more lines of code than a Pig script
- Native Hadoop Java code requires 16x more development time than Pig
- Native Hadoop Java code takes 11 min to run, while Pig requires 20 min

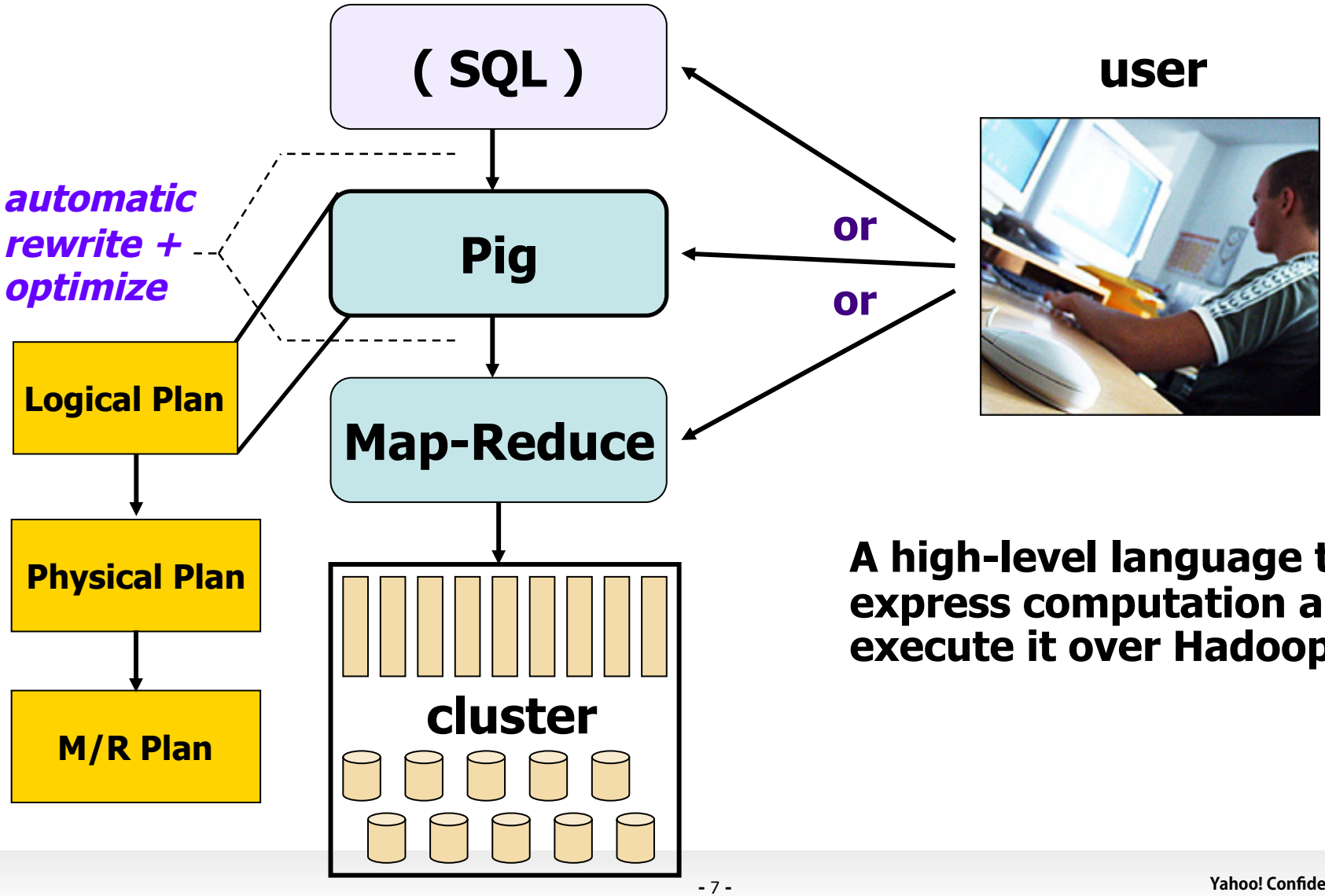


Solution in Native Hadoop M/R program

```
Users = LOAD 'users' AS (name, age);
Fltrd = FILTER Users by age >= 18 and age <= 25;
Pages = LOAD 'pages' AS (user, url
Jnd = JOIN Fltrd BY name, Pages BY
Grpd = GROUP Jnd by url;
Smmd = FOREACH Grpd GENERATE group
       clicks;
Srtd = ORDER Smmd BY clicks;
Top100 = LIMIT Srtd 100;
STORE Top100 INTO 'top100sites';
```

Solution using Pig Script

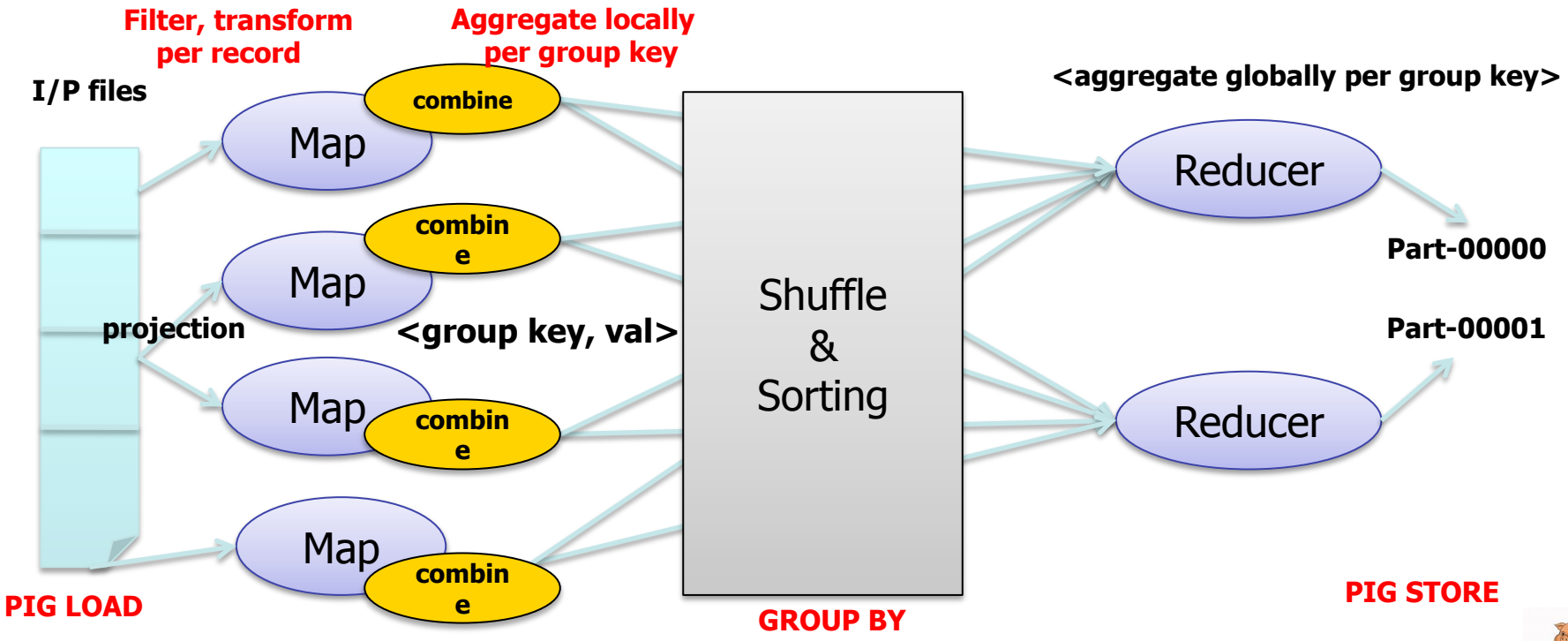
Pig Architecture: Map-Reduce as Backend



A high-level language to express computation and execute it over Hadoop

SQL Operators to M/R

- Select a, UPPER(b), COUNT(c) -- projection, scalar, aggregate
- Where a > 10 -- filter expression
- Group by a; -- group by



Hadoop M/R provides Distributed Group by Aggregate

Pig Architecture

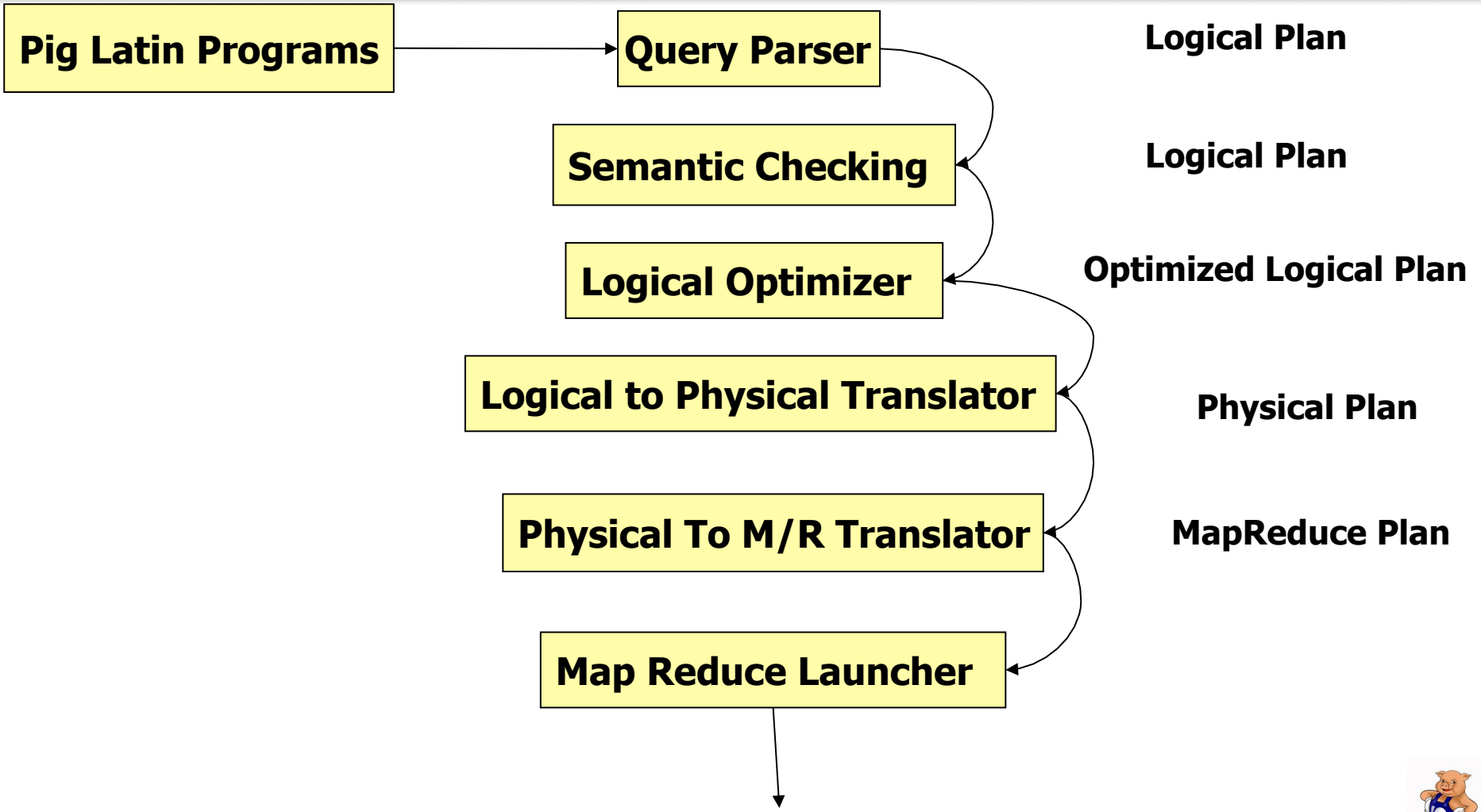
Front End

- Parsing
- Semantic checking
- Planning
- Runs on user machine or gateway

Back End

- Script execution
- Runs on grid

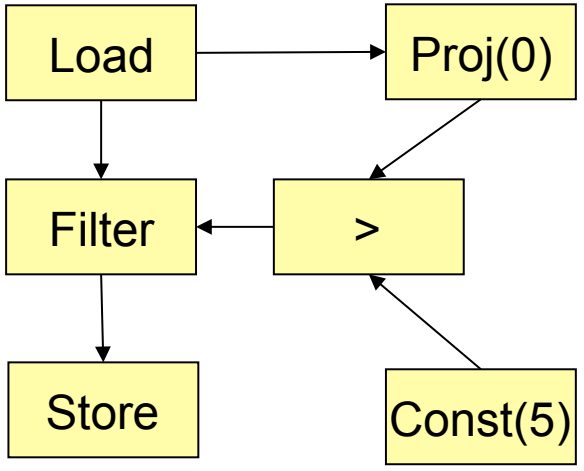
Front End Architecture



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';
```

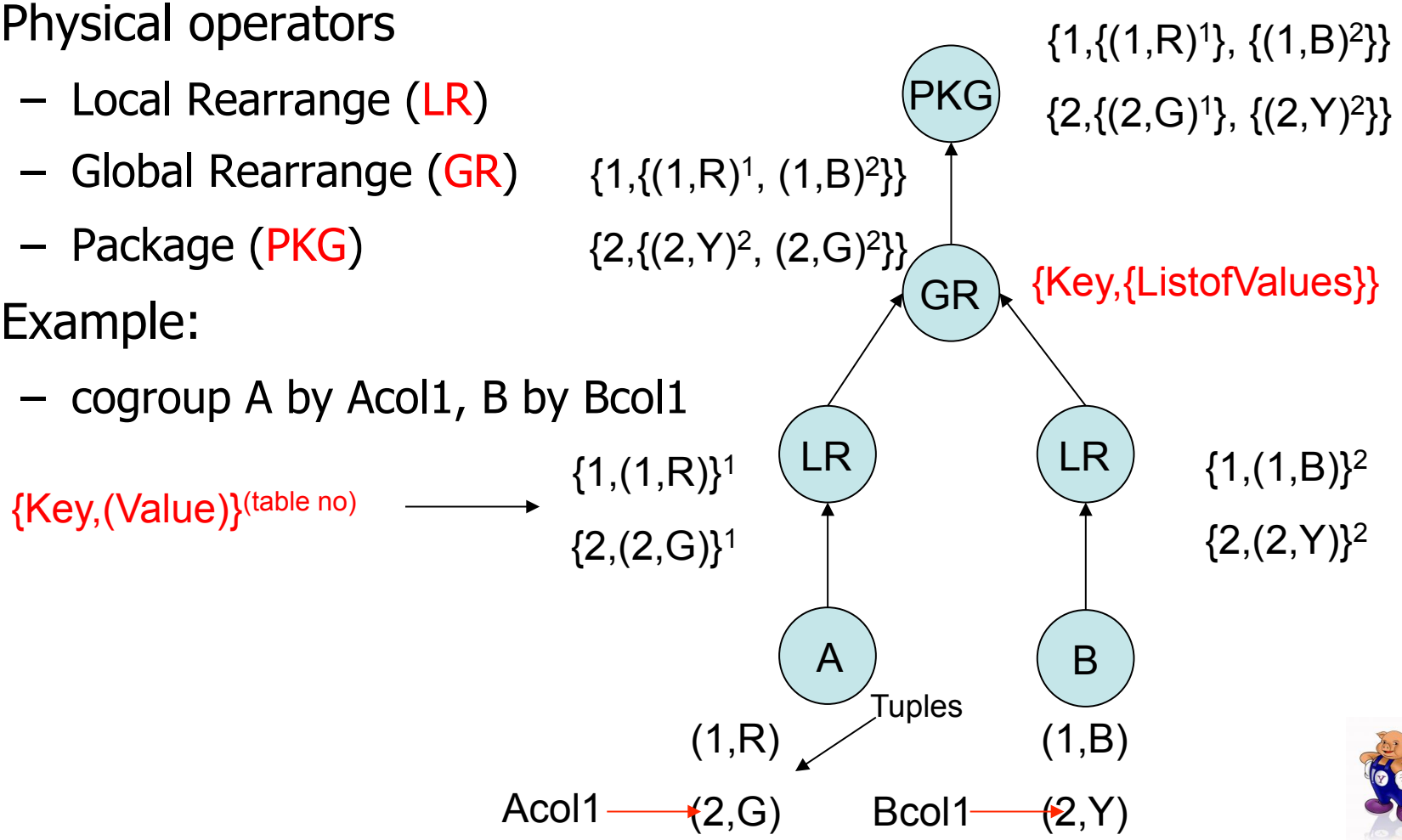


Physical Plan

- 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

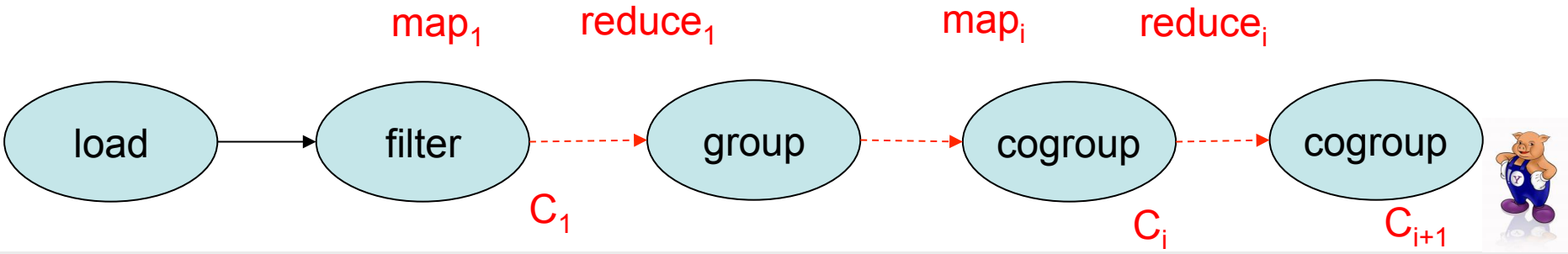
Logical to Physical Plan for Co-Group operator

- Logical operator for co-group/group is converted to 3 Physical operators
  - Local Rearrange (LR)
  - Global Rearrange (GR)
  - Package (PKG)
- Example:
  - cogroup 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

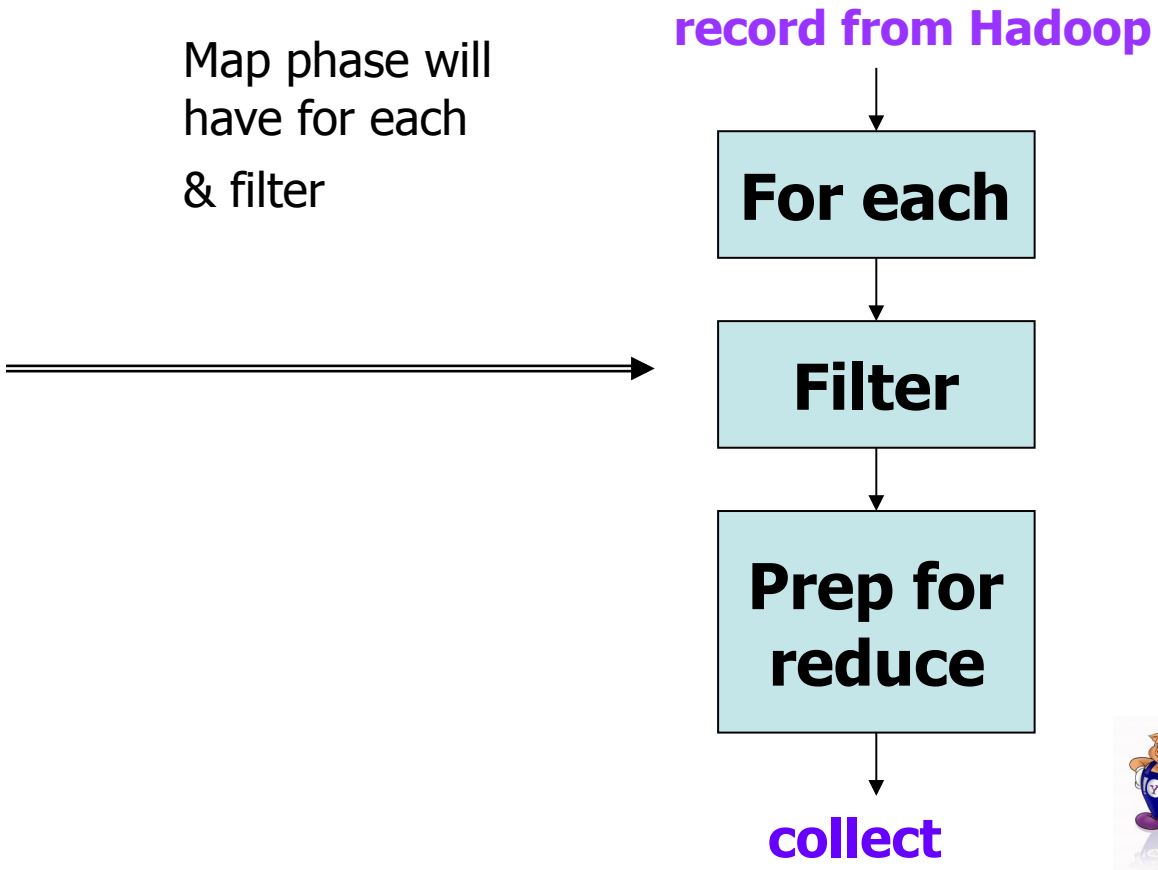


Back End Architecture

Pig operators are placed in a pipeline and each record is then pulled through.

```
A = load …
B = foreach …
C = filter …
D = group …
…
```

Map phase will have for each & filter



Lazy Execution

- **Nothing executes until you request output**
- **M/R execution takes place on your cluster only when the `store` or `dump` command is encountered**
- **Advantages:**
  - **In-memory pipelining**
  - **Filter re-ordering across multiple commands**

Pig Latin

Introduction to Pig Latin

- **Pig Latin** is a dataflow language through which users can write programs in Pig
- Pig Latin consists of the following datatypes:
  - **Data Atom**

    Simple atomic data value (e.g. `alice` or `1.0`)
  - **Tuple**

    Tuple is a row w/ one or more fields of any data type, consists of a sequence of "fields" e.g. `(alice,lakers)` or `(alice,kings)`
  - **Data Bag**

    Set of Tuples equivalent to a "table" in SQL terminology

    Can have tuples w/ different number of fields & even fields can have different data types

    Can have duplicate tuples: e.g.

    ```
    {(alice,lakers), (alice,kings)}
    ```
  - **Data Map**

    A set of key value pairs `[likes#(lakers,kings),age#22]`

Pig Latin: Data Types

Type	Example
1. Data Atom an atomic value	'alice'
2. Tuple a record	(alice,lakers)
3. Data Bag collection supporting scan	{(alice,lakers) (alice,kings)}
4. Data Map collection with key lookup	['likes'# {(lakers), (kings)} 'age'#22]

Pig Types

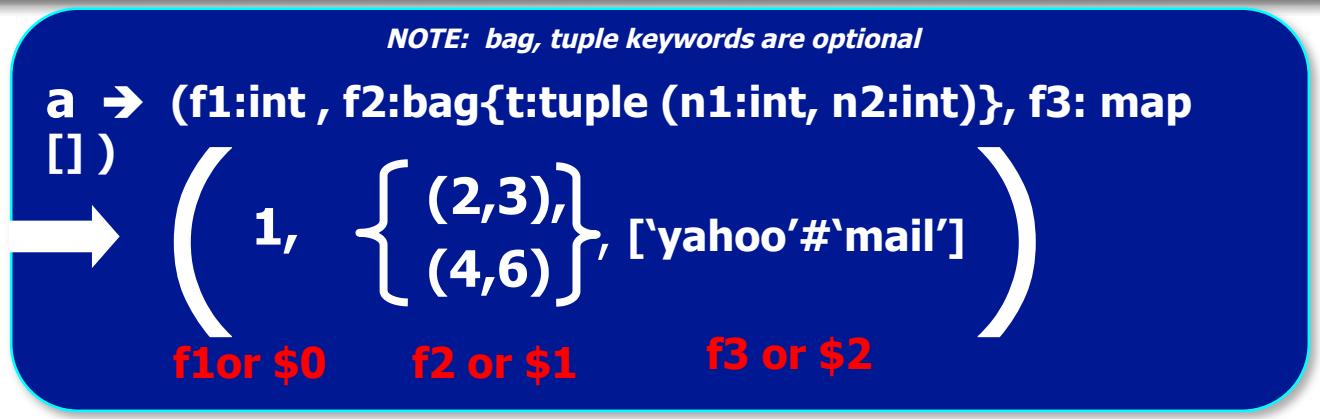
- Pig supports the following types in addition to **map, bag,** and **tuples**
  - **int:** 32-bit integer, signed type
  - **long:** 64-bit for unsigned type
    - (alice,25,6L)
  - **float:** 32-bit floating point
    - (alice,25, 5.6f),(alice,25,1.92e2f)
  - **double:** 64-bit floating point
    - (alice,22,1.92),(alice,22,1.92e2)
  - **chararray:** set of characters stored in Unicode (Java String)
    - ('alice', 'bob')
  - **bytearray:** array of bytes (Java DataByteArray)
    - default type if you do not specify the type of the field

Pig Latin: Basic Operations & Expressions

Expressions

a = name of alias which contains types int, tuple and map

Sample tuple of **a**



Field referred to by position

$1 = {(2,3), (4,6)}

$0 = 1

Field referred to by name

f2 = {(2,3), (4,6)}

Projection of a data item

f2.$0 = {(2), (4)}

Map lookup

f3# 'yahoo' = 'mail'

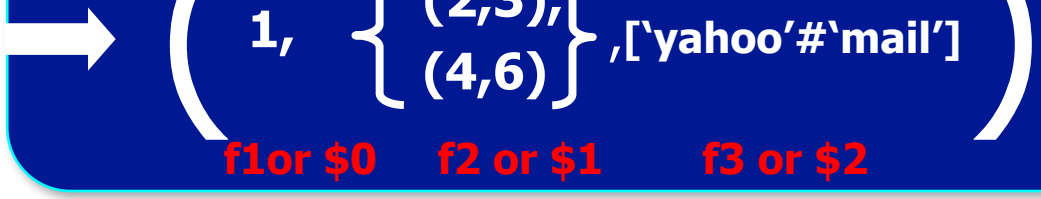
Function application

SUM(f2.$0) = 6 (2+4)
COUNT(f2) = 2L

Conditions (Boolean Expressions)

a = name of alias which contains types int, tuple and map

Sample tuple of **a**

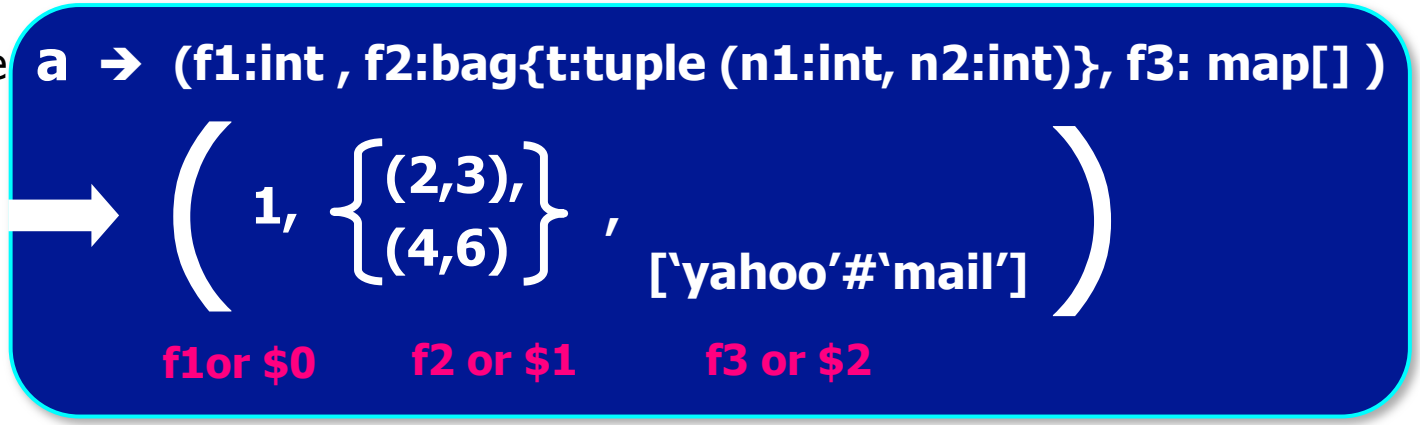


- ==, !=, >, >=, <, <= for numerical comparison
  - only f1==1 works as f1 is type integer
  - eq, neq, gt, gte, lt, lte for string comparisons
  - f3#'yahoo' gt 'ma'
  - matches for regular expression matching
  - f3#'yahoo' matches '(?i)MAIL'

Pig Latin Operators & Conditional Expressions

a = name of alias that contains types int, tuple and map

Sample tuple of **a**



- **Logical Operators**
  - AND, OR, NOT
  - **f1==1 AND f3#'yahoo' eq 'mail'**
- **Conditional Expression (aka BinCond)**
  - (Condition?exp1:exp2)
  - **f3#'yahoo'matches '(?i)MAIL' ? 'matched': 'notmatched'**

Pig Latin Language by Example

Examples

1. Word Count
2. Compute average number of page visits by user
3. Identify users who visit highly-ranked or good pages
4. Namenode Audit Log processing via UDF's in Chukwa (User Defined Functions)

Word Count Using Pig

wc.txt

```
program program
pig pig
program pig
hadoop pig
latin latin
latin latin
```

- Count frequency of each word in a text file
- Basic idea:
  - **Load** this file using a loader
  - **Foreach** record **generate** word token
  - **Group** by each word
  - **Count** number of words in a group
  - **Store** to file

Word Count Using Pig

```
myinput = load '/user/viraj/wc.txt' USING TextLoader() as (myword:chararray);
```

```
(program program)
(pig pig)
(program pig)
(hadoop pig)
(latin latin)
(latin latin)
```

```
words = FOREACH myinput GENERATE FLATTEN(TOKENIZE(myword));
```

```
(program)
(program)
(pig)
(pig)
(program)
(pig)
(hadoop)
(pig)
(latin)
(latin)
(latin)
(latin)
```

```
grouped: {group: chararray,words: {token: chararray}}
```

```
grouped = GROUP words BY $0;
```

```
(pig,{(pig),(pig),(pig),(pig)})
(latin,{(latin),(latin),(latin),(latin)})
(hadoop,{(hadoop)})
```

```
counts = FOREACH grouped GENERATE group, COUNT(words);
```

```
(pig,4L)
(latin,4L)
(hadoop,1L)
(program,3L)
```

```
store counts into '/user/viraj/pigoutput' using PigStorage();
```

Write to HDFS /user/viraj/pigoutput/part-* file

Compute Average Number of Page Visits by User

Visits log

user	url	time
Amy	www.cnn.com	8:00
Amy	www.crap.com	8:05
Amy	www.myblog.com	10:00
Amy	www.flickr.com	10:05
Fred	cnn.com/index.htm	12:00
Fred	cnn.com/index.htm	1:00
	⋮	

- Logs of user visiting a webpage consists of (user,url,time)
- Fields of the log are tab-separated and in text format
- Basic idea:
  - **Load** the log file
  - **Group** based on the **user** field
  - **Count** the group
  - Calculate **average** for **all** users

How to Program This in Pig Latin

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

```
(Amy,www.cnn.com,8:00)
(Amy,www.crap.com,8:05)
(Amy,www.myblog.com,10:00)
(Amy,www.flickr.com,10:05)
(Fred,cnn.com/index.htm,12:00)
(Fred,cnn.com/index.htm,13:00)
```

→ dump visits;

USER_VISITS = group VISITS by user;

```
(Amy,{(Amy,www.cnn.com,8:00),(Amy,www.crap.com,8:05),(Amy,www.myblog.com,10:00),(Amy,www.flickr.com,10:05)})
(Fred,{(Fred,cnn.com/index.htm,12:00),(Fred,cnn.com/index.htm,13:00)})
```

USER_CNTS = foreach USER_VISITS generate group as user, COUNT(VISITS) as numvisits;

```
(Amy,4L)
(Fred,2L)
```

ALL_CNTS = group USER_CNTS all;

```
(all,{(Amy,4L),(Fred,2L)})
```

AVG_CNT = foreach ALL_CNTS generate AVG(USER_CNTS.numvisits);

```
(3.0)
```

Identify Users Who Visit "Good Pages"

Visits log

user	url	time
Amy	www.cnn.com	8:00
Amy	www.crap.com	8:05
Amy	www.myblog.com	10:00
Amy	www.flickr.com	10:05
Fred	cnn.com/index.htm	12:00
Fred	cnn.com/index.htm	1:00
⋮		

Good page

Pages log

url	pagerank
www.cnn.com	0.9
www.flickr.com	0.9
www.myblog.com	0.7
www.crap.com	0.2
⋮	

- **Good pages:** those pages visited by users whose page rank is greater than 0.5
- **Basic idea:**
  - Join tables based on **URL**
  - Group based on **user**
  - Calculate average pagerank of **user**-visited pages
  - Filter user who has average pagerank greater than 0.5
  - Store the result

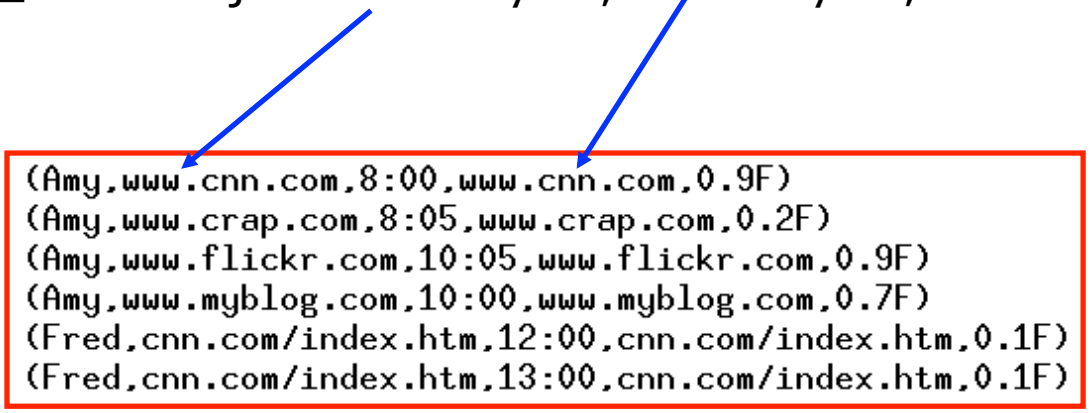
How to Program This in Pig

Load files for processing with appropriate types

```
VISITS = load `/user/viraj/visits.txt' as (user:chararray, url:chararray, time:chararray);
PAGES = load '/user/viraj/pagerank.txt' as (url:chararray, pagerank:float);
```

Join them on URL fields of table

```
VISITS_PAGES = join VISITS by url, PAGES by url;
```



How to Program This in Pig

Group by user; in our case: Amy, Fred

USER_VISITS = group VISITS_PAGES by user;

```
(Amy,{(Amy,www.cnn.com,8:00,www.cnn.com,0.9F),(Amy,www.crap.com,8:05,www.crap.com,0.2F),(Amy,www.flickr.co
og.com,10:00,www.myblog.com,0.7F)})
(Fred,{(Fred,cnn.com/index.htm,12:00,cnn.com/index.htm,0.1F),(Fred,cnn.com/index.htm,13:00,cnn.com/index.h
```

Generate an average pagerank per user

USER_AVGPR = foreach USER_VISITS generate group, AVG(VISITS_PAGES.pagerank) as avgpr;

```
(Amy,0.6749999858438969)
(Fred,0.10000000149011612)
```

Filter records based on pagerank

GOOD_USERS = filter USER_AVGPR by avgpr > 0.5f;

```
(Amy,0.6749999858438969)
```

store **GOOD_USERS** into '/user/viraj/goodusers'; **Store results in hdfs**

Chukwa Namenode Audit log processing

(1232063871538L, [capp#/hadoop/log/audit.log,ctags#cluster="cluster2", body#2009-01-15 23:57:51,538 INFO org.apache.hadoop.fs.FSNamesystem.audit: ugi=users,hdfs<tab> ip=/ 11.11.11.11<tab>cmd=mkdirs src=/user/viraj/output/part-0000<tab>dst=null<tab> perm=users:rwx------ ,csource#machine1@grid.com])

audit.log generated by Chukwa

(1232063879123L, [capp#/hadoop/log/audit.log,ctags#cluster="cluster2",body#2009-01-15 23:57:59,123 INFO org.apache.hadoop.fs.FSNamesystem.audit: ugi=users,hdfs ip=/11.12.11.12 <tab>cmd=rename<tab>src=/user/viraj/output/part-0001<tab> dst=/user/viraj/output/ part-0002<tab>perm=users:rw-------, csource#machine2@grid.com])

- Count number of HDFS operations that took place
- Basic idea:
  - Load this file using a custom loader known as ChukwaStorage into a long column and a list of map fields
  - Extract the value of the body key as it contains the cmd the namenode executed
  - Cut the body tag on tabs to extract only the cmd value using a UDF
  - Group by cmd value and Count it

How to Program this in Pig

Register the jars which contain your udf's

```
register chukwa-core.jar
register chukwaexample.jar
```

Jar internally used by ChukwaStorage

Contains both ChukwaStorage and Cut classes

```
A = load '/user/viraj/Audit.log' using
    chukwaexample.ChukwaStorage() as (ts: long,fields)
```

Project the right value for key body

```
B = FOREACH A GENERATE fields#'body' as log;
```

Cut the body based on tabs to extract "cmd"

```
C = FOREACH B GENERATE FLATTEN(chukwaexample.Cut
    (log)) as (timestamp, ip, cmd, src, dst, perm);
```

How to Program this in Pig

Project the right column for use in group

```
D = FOREACH C GENERATE cmd as cmd:chararray;
```

Group by command type

```
E = group D by cmd;
```

Count the commands and dump

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

dump F;
```

```
(cmd=open,65L)
(cmd=create,39L)
(cmd=delete,38L)
(cmd=mkdirs,49L)
(cmd=rename,27L)
(cmd=listStatus,92L)
(cmd=setPermission,8L)
(cmd=setReplication,2L)
```

Commands for Manipulating Data Sets

Load Data Using PigStorage

```
query = load '/user/viraj/query.txt' using PigStorage()
                as (userId, queryString, timestamp)

queries = load '/user/viraj/querydir/part-*' using PigStorage()
                as (userId, queryString, timestamp)
```

All files under querydir containing part-* are loaded

- Can be a file
- Can be a path with globbing
- userID, queryString, timestamp fields should be tab-separated

Store Data Using PigStorage & View Data

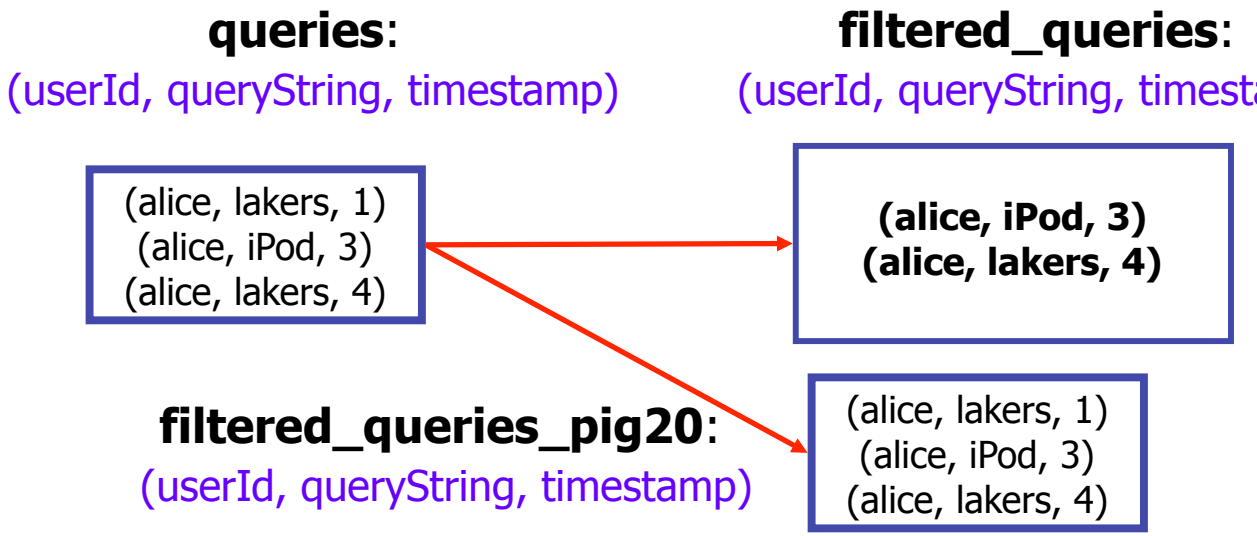
```
store joined_result into '/user/viraj/output' ;
store joined_result into '/user/viraj/myoutput/' using PigStorage('\u0001');
store joined_result into '/user/viraj/myoutputbin' using BinStorage();
store sports_views into '/user/viraj/myoutput' using PigDump();

dump sports_views;
```

```
(alice,lakers,3L)
(alice,lakers, 0.7f)
```

- **Default PigStorage if you do not specify anything**
  Result stored as ASCII text in files part-* under output dir
- **Similar to load can use PigStorage ('{delimiter}')**
  Stores records with fields delimited by Unicode {delimiter}
  - Default is tab i.e. you don't specify anything
  - Store as Ctrl+A separated by specifying PigStorage('\u0001')
- **BinStorage store arbitrarily nested data**
  Used for loading intermediate results that were previously stored using it
- **Dump displays data on terminal**
  Use PigDump storage class internally

Filter Data

```
queries = load 'query_log.txt' using PigStorage('\u0001') as (userId:
chararray, queryString: chararray, timestamp: int);

filtered_queries =  filter queries by timestamp > 1;

filtered_queries_pig20 = filter queries by timestamp is not null;

store filtered_queries into  'myoutput' using PigStorage();
```



Join in Map Reduce

page_view

pageid	**userid**	time
1	**111**	9:08:01
2	**111**	9:08:13
1	**222**	9:08:14

X

user

userid	age	gender
111	25	female
222	32	male

=

pv_users

pageid	age
1	25
2	25
1	32

Join in Map Reduce

page_view

pageid	**userid**	time
1	**111**	9:08:01
2	**111**	9:08:13
1	**222**	9:08:14

user

userid	age	gender
111	25	female
222	32	male

Map

key	value
111	<1,1>
111	<1,2>
222	<1,1>

key	value
111	<2,25>
222	<2,32>

Shuffle Sort

key	value
111	<1,1>
111	<1,2>
111	<2,25>

key	value
222	<1,1>
222	<2,32>

Reduce

pv_

pag
1
2

re

page
1

r

(Co)Group Data

Data 1:

queries: (userId: charrarray, queryString : chararray, timestamp: int)

Data 2:

sports_views: (userId: chararray, team: chararray, timestamp: int)

```
queries = load query.txt as ()..;
sport_views = load sports_views.txt as ....;

team matches = cogroup  sports_views by (userId, team),
                        queries by (userId, queryString);
```

Example of Cogrouping

sports_views:
(userId, team, timestamp)

(alice, lakers, 3) (alice, lakers, 7)

queries:
(userId, queryString, timestamp)

(alice, lakers, 1) (alice, iPod, 3) (alice, lakers, 4)

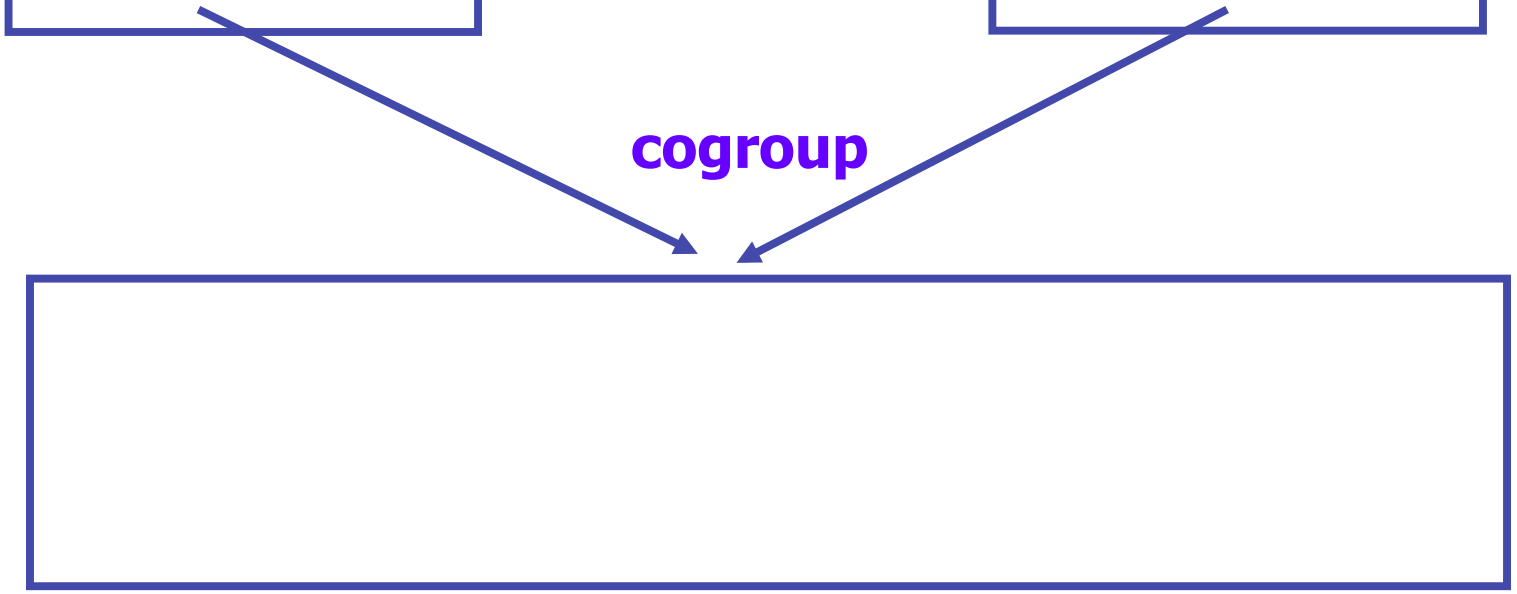
Example of Cogrouping

sports_views:
(userId, team, timestamp)

(alice, lakers, 3)
(alice, lakers, 7)

queries:
(userId, queryString, timestamp)

(alice, lakers, 1)
(alice, iPod, 3)
(alice, lakers, 4)



team_matches: (group, sports_views, queries)

```
team matches = cogroup  sports_views by (userId, team),
                        queries by (userId, queryString);
```

Example of Cogrouping

sports_views:
(userId, team, timestamp)

(alice, lakers, 3)
(alice, lakers, 7)

queries:
(userId, queryString, timestamp)

(alice, lakers, 1)
(alice, iPod, 3)
(alice, lakers, 4)

cogroup

((alice, lakers), { (alice, lakers, 3) (alice, lakers, 7) } , { (alice, lakers, 1) (alice, lakers, 4) })

team_matches: (group, sports_views, queries)

Example of Cogrouping

sports_views:
(userId, team, timestamp)

(alice, lakers, 3)
(alice, lakers, 7)

queries:
(userId, queryString, timestamp)

(alice, lakers, 1)
(alice, iPod, 3)
(alice, lakers, 4)

cogroup

((alice, lakers), { (alice, lakers, 3) (alice, lakers, 7) } , { (alice, lakers, 1) (alice, lakers, 4) })

((alice, iPod), { },{ (alice, iPod, 3) })

team_matches: (group, sports_views, queries)

Cogrouping

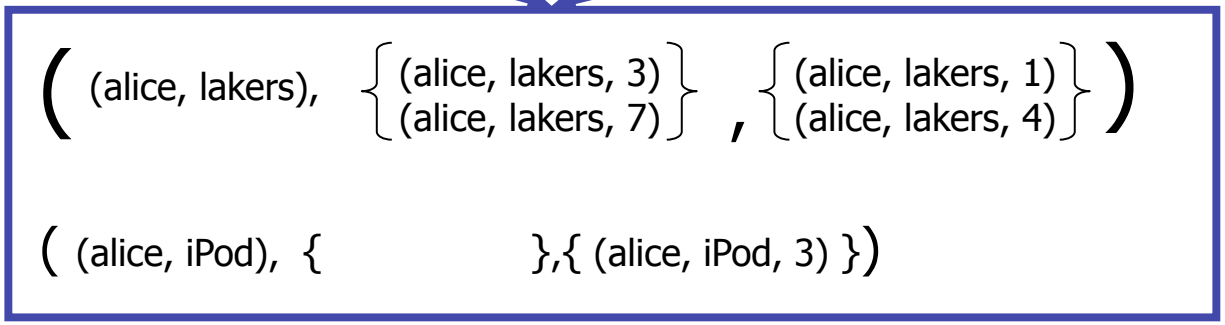
sports_views:
(userId, team, timestamp)

(alice, lakers, 3)
(alice, lakers, 7)

queries:
(userId, queryString, timestamp)

(alice, lakers, 1)
(alice, iPod, 3)
(alice, lakers, 4)

cogroup



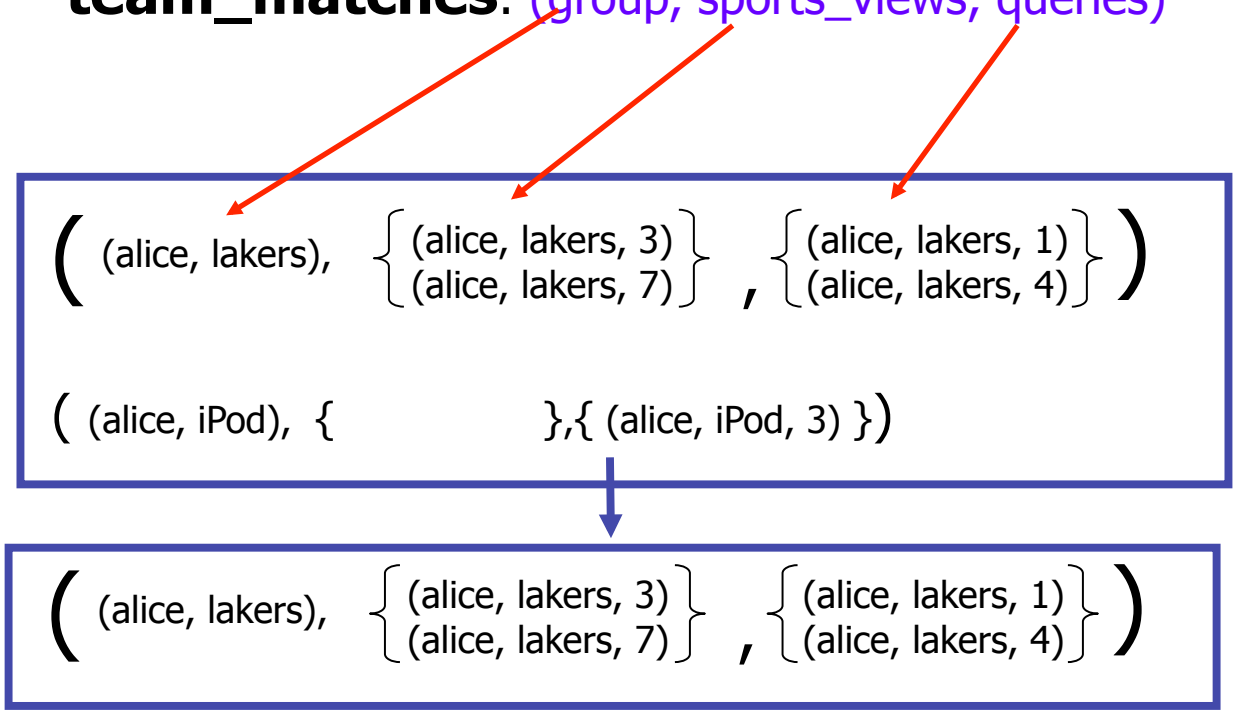
team_matches: (group, sports_views, queries)

Can operate on grouped data just as on base data using foreach

Filter Records of Grouped Data

```
filtered_matches = filter team_matches
                   by COUNT(sports_views) > '0';
```

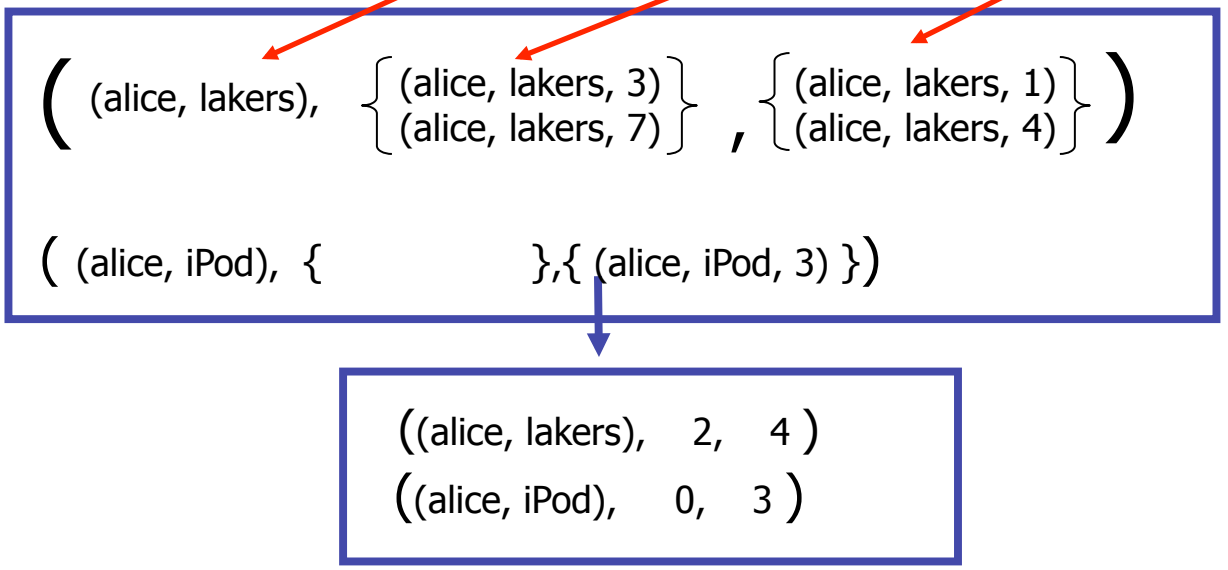
team_matches: (group, sports_views, queries)



Per-Record Transformations Using FOREACH

```
times_and_counts = foreach team_matches
                   generate group,
                            COUNT(sports_views),
                            MAX(queries.timestamp);
```

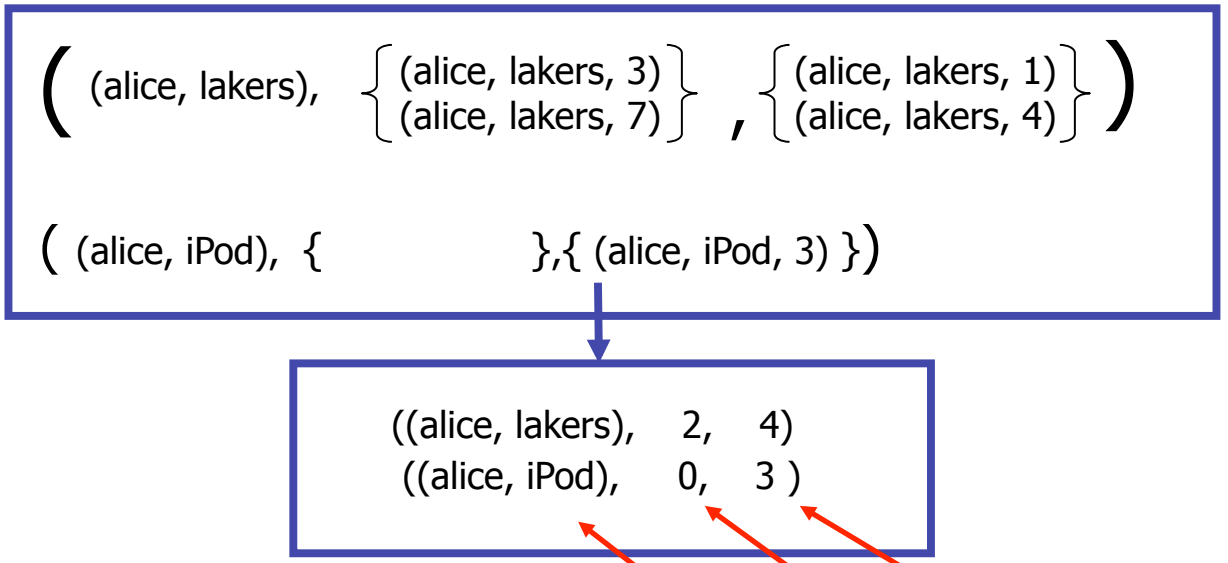
team_matches: (group, sports_views, queries)



times_and_counts: (group, -, -)

Give Names to Output Fields in FOREACH

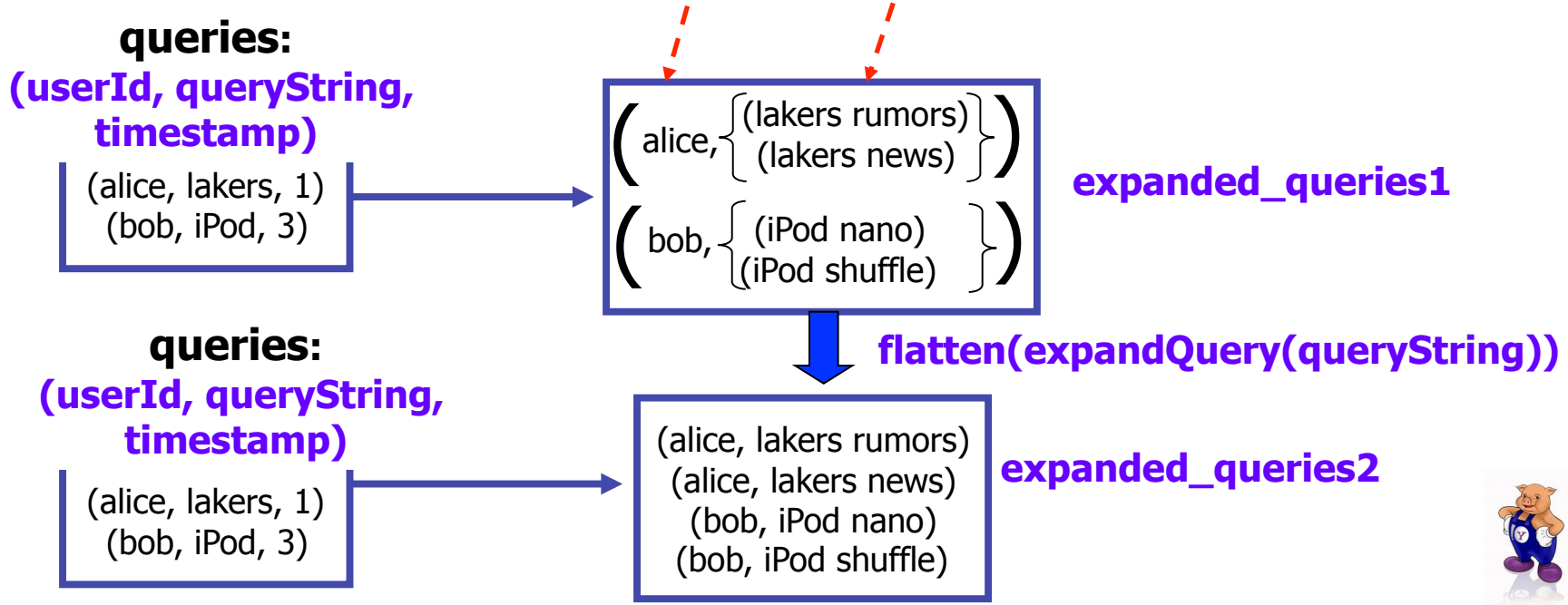
```
times_and_counts = foreach team_matches
                   generate group,
                            COUNT(sports_views) as count,
                            MAX(queries.timestamp) as max;
```



times_and_counts: (group, count, max)

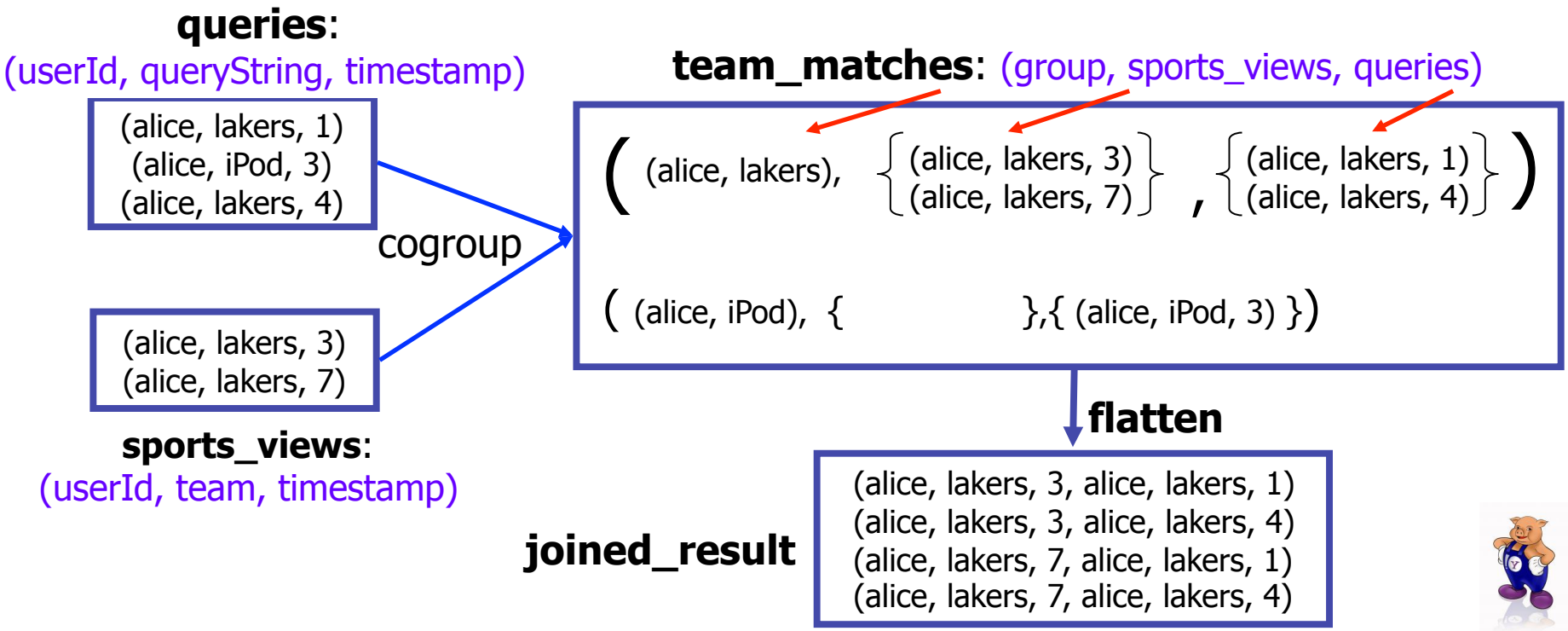
Eliminate Nesting: FLATTEN

```
expanded_queries1 = foreach queries
                    generate userId, expandQuery(queryString);

expanded_queries2 =  foreach queries
                    generate userId, flatten(expandQuery
(queryString));
```



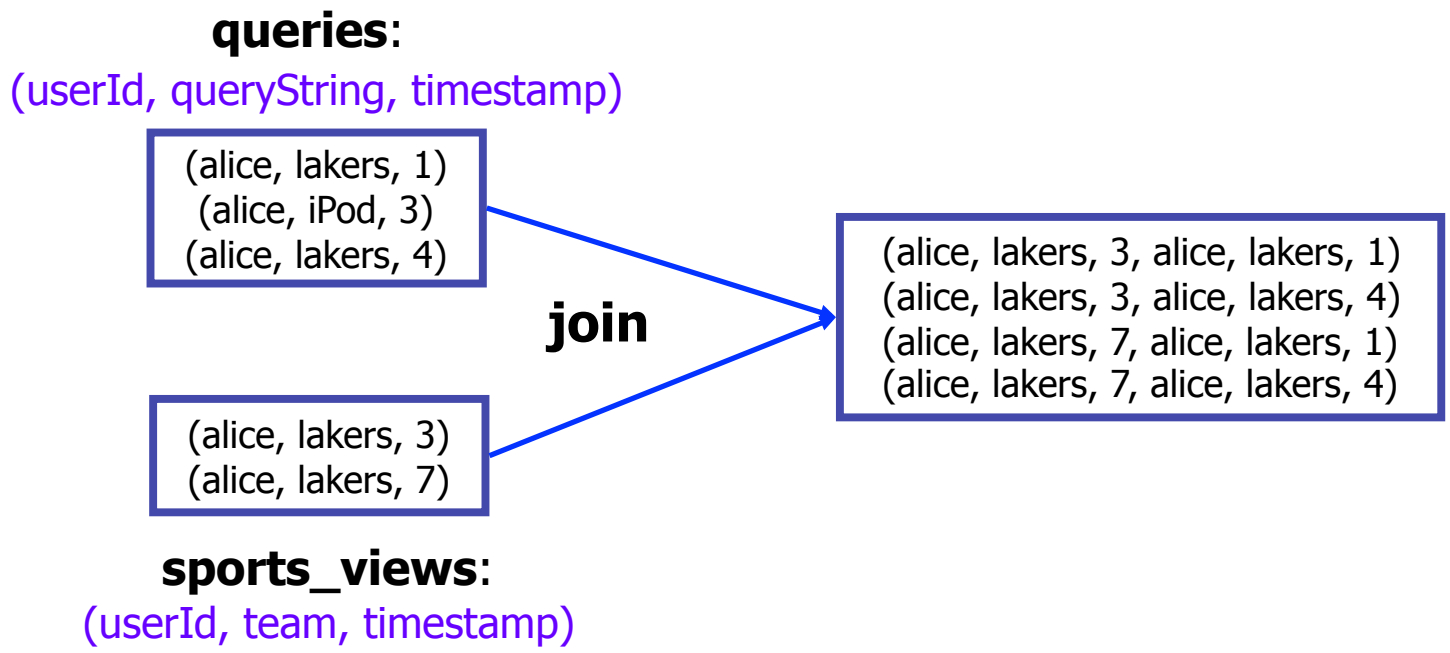
Flatten Multiple Items Resulting from COGROUP

```
team matches = cogroup  sports_views by (userId, team),
                        queries by (userId, queryString);

joined_result = foreach team_matches generate flatten(sports_views),
                flatten(queries);
```



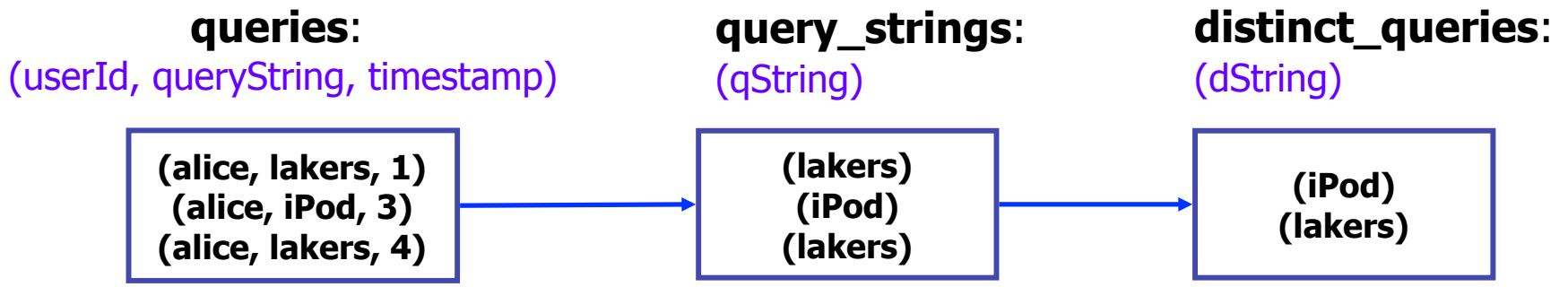
Syntax Shortcut: JOIN = COGROUP + FLATTEN

```
joined_result = join sports_views by (userId, team),
                     queries by (userId, queryString);
```



Eliminate Duplicates with DISTINCT

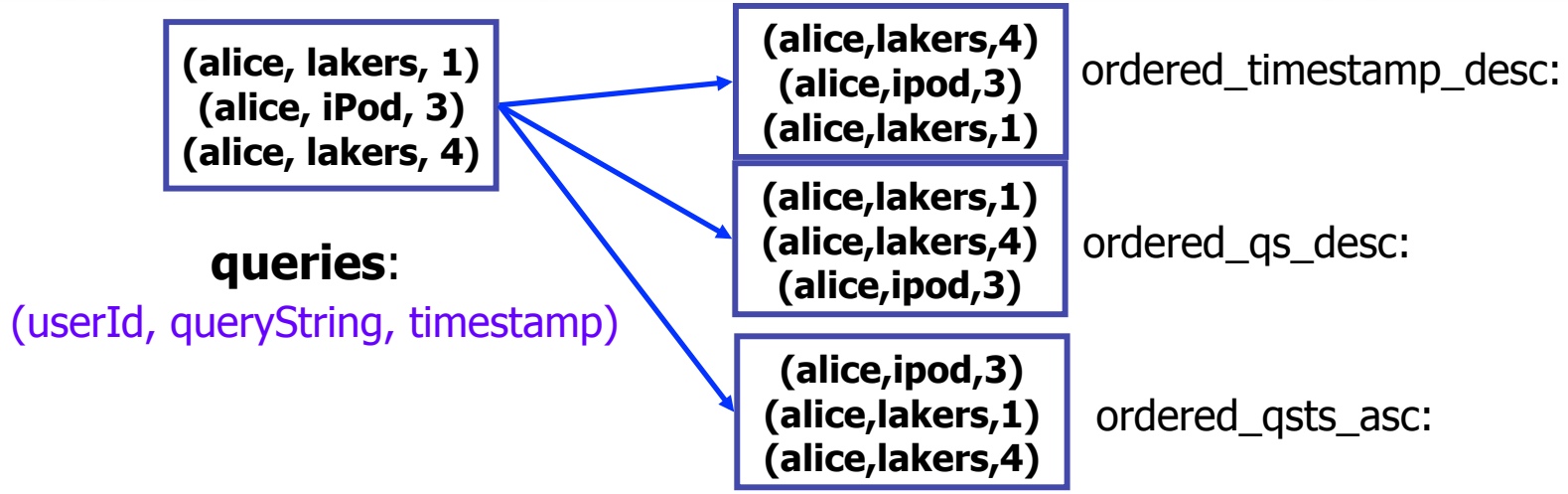
```
queries = load 'queries.txt' as (userId, queryString: chararray, timestamp);

query_strings = foreach queries generate queryString as qString;

distinct_queries = distinct query_strings as dString;
```



Duplicate detection done as bytearrays if type is not specified

Order Data with ORDER BY

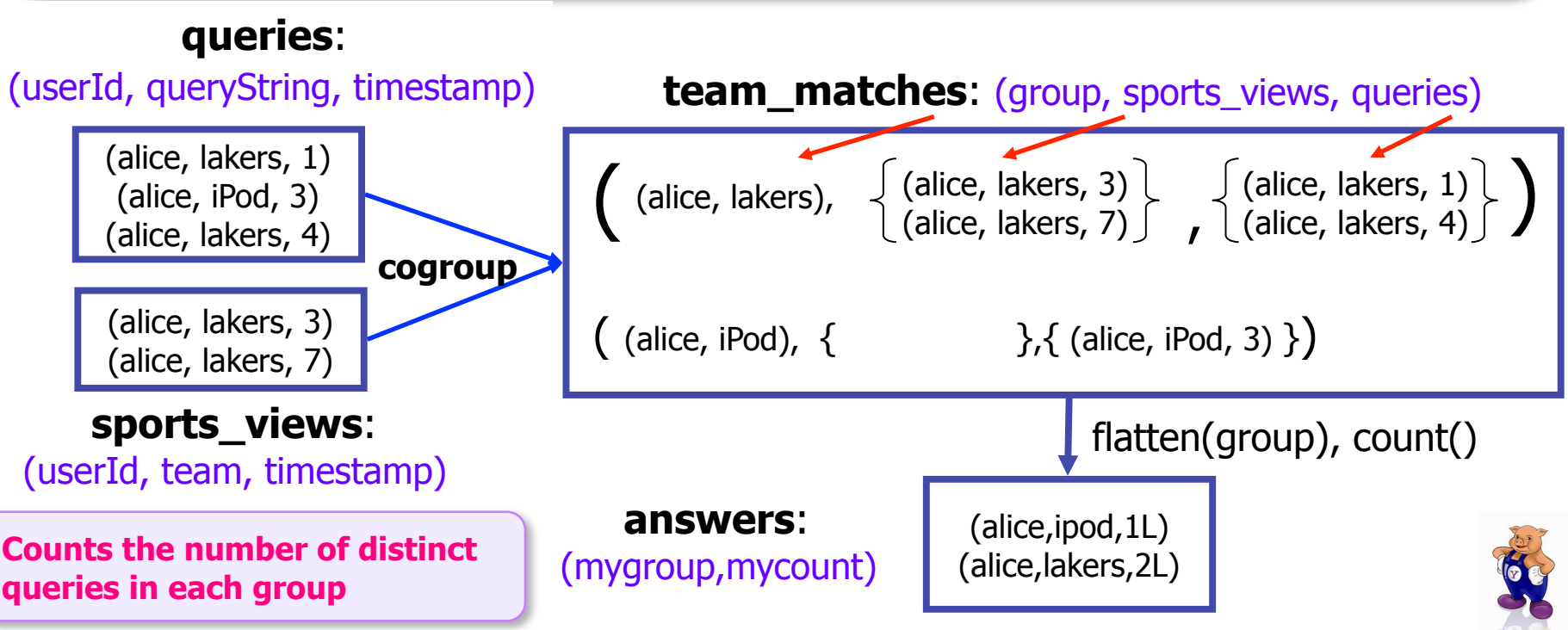
```
queries = load 'queries.txt' as (userId: chararray, queryString: chararray,
timestamp: int);

ordered_timestamp_desc = order queries by timestamp desc;
ordered_qs_desc = order queries by queryString desc;

ordered_qsts_asc =  order queries by queryString, timestamp parallel 7;
```



- Parallel use 7 reducers for this operation
  Generates 7-part files
- Pig supports both desc ordering, default is ascending

Nested Operations

```
team matches = cogroup  sports_views by (userId, team), queries by (userId, queryString);

answers = foreach team_matches { distinct_qs = DISTINCT queries;
generate flatten(group) as mygroup, COUNT(distinct_qs) as mycount; }
```



Counts the number of distinct queries in each group

Pig Streaming & Param Substitution

Pig Streaming

- Entire portion of the dataset that corresponds to an alias is sent to the external task and output streams out
  - Use the `stream` keyword
- Example using Python and Unix commands in Pig to process records:

```
shell> cat data/test10.txt;
Apple is the best fruit.     80
Plum is the best fruit. 20
```

```
a = load 'data/test10.txt';
b = stream a through `python -c "import sys; print sys.stdin.read().replace
    ('Apple','Orange');"`;
dump b;
```

```
(Orange is the best fruit., 80)
(Plum is the best fruit., 20)
```

```
c = stream a through `cut -f2,2`;
dump c;
```

```
(80)
(20)
```

Pig Streaming Using Perl/Python Programs

- Invoke a Perl, Python script from Pig and then execute it

py_test.py:

```
#!/usr/local/bin/python
import sys; print sys.stdin.read().replace('Apple','Orange');
shell> chmod 700 py_test.py
```

```
d = stream a through `py_test.py`;
```

Parameter Substitution in Pig

- Enables you to have parameters within a Pig script & provides values for these parameters at runtime
- Provides parameter values in a file or at command line
- Generates parameter values at runtime by running a binary or a script

```
$ pig -param date=20080202
$ pig -param_file my_param_file
```

```
%default date '20080101'
A = load '/data/mydata/$date';
```

Using declare:

```
%declare date '20080101'
```

```
%declare CMD `generate_date_script`
```

This runs a script called generate_date_script

Pig 2.0 Support for NULLs

- **Similar to SQL semantic of NULL as unknown, and not the C/Java semantic of null as unset**
- **When/how are NULLS generated??**
  - Nulls can be generated by operations, such as dividing by 0
  - Nulls can be generated by User-Defined Functions
  - By de-referencing a map key that doesn't exist in a map

    For example:

    If map info = `[name#fred, phone#555121]`

    then `info#address` will return `NULL`

Interaction of NULLs with Various Operators in Pig

Operator	Interaction
Comparison operators	If either sub-expression is null, then the result of the equality comparison will be null
Matches	If either the string being matched against or the string defining the match is null, then the result will be null
Is null	Will return true if the tested value is null
Arithmetic operators, concat	If either sub-expression is null, then the resulting expression will be null
Size	If the tested object is null, then size will return null
Dereference of a map or tuple	If the dereferenced map or tuple is null, then the result will be null
Cast	Casting a null from one type to another will result in a null
Aggregates	Built-in aggregate functions will ignore nulls, just as in SQL. *However,* user-defined aggregates are free to handle nulls the way they see fit.

Casts in Pig 2.0

- Previously, all data fields were represented as **chararray** – i.e., no notion of types
- Fields can be explicitly cast

```
B = FOREACH A GENERATE (int)$0 + 1;
```

- Once cast, a field remains that type – it's not automatically cast back
- Wherever possible, implicit casts are done

```
B = FOREACH A GENERATE $0 + 1, $1 + 1.0
```

$0 will be cast to int (regardless of underlying data) and $1 to double

- Implicit casts used, then declared data types do not match operation

```
A = LOAD 'data' as (a: int, b: float);
B = FOREACH A GENERATE a + b; -- a is converted to float
```

- If the underlying data is really int or long, you'll get better performance by declaring the type or casting the data

Translating SQL Queries into Pig Latin

Constructs

SQL	Pig	Example
From table	Load file(s)	**SQL**: from X; **Pig**: A = load 'mydata' using PigStorage('\t') as (col1, col2, col3);
Select	Foreach … generate	**SQL**: select col1 + col2, col3 … **Pig**: B = foreach A generate col1 + col2, col3;
Where	Filter	**SQL**: select col1 + col2, col3 from X where col2>2; **Pig**: C = filter B by col2 > '2';

Constructs

SQL	Pig	Example
Group by	Group + foreach … generate	**SQL:** select col1, col2, sum(col3) from X group by col1, col2; **Pig**: D = group A by (col1, col2); E = foreach D generate flatten(group), SUM(A.col3);
Having	Filter	**SQL**: select col1, sum(col2) from X group by col1 having sum(col2) > 5; **Pig**: F = filter E by $1 > '5';
Order By	Order … By	**SQL**: select col1, sum(col2) from X group by col1 order by col1; **Pig**: H = ORDER E by $0;

Constructs

SQL	Pig	Example
Distinct	Distinct	**SQL:** select distinct col1 from X; **Pig:** I = foreach A generate col1; J = distinct I;
Distinct Agg	Distinct in foreach	**SQL:** select col1, count (distinct col2) from X group by col1; **Pig:** K = foreach D { L = distinct A.col2; generate flatten(group), SUM(L); }

Constructs

SQL	Pig	Example
Join	Cogroup + flatten (also shortcut: JOIN)	**SQL:** select A.col1, B.col3 from A join B using (col1); **Pig:** A = load 'data1' using PigStorage('\t') as (col1, col2); B = load 'data2' using PigStorage('\t') as (col1, col3); C = cogroup A by col1 **inner**, B by col1 **inner**; D = foreach C generate flatten(A), flatten(B); E = foreach D generate A.col1, B.col3;

Pig Built-in Functions

- Pig has a variety of built-in functions:
  - **Storage**
    - **TextLoader**: for loading unstructured text files. Each line is loaded as a tuple with a single field which is the entire line.
  - **Filter**
    - **isEmpty**: tests if bags are empty
  - **Eval Functions**
    - **COUNT**: computes number of elements in a bag
    - **SUM**: computes the sum of the numeric values in a single-column bag
    - **AVG**: computes the average of the numeric values in a single-column bag
    - **MIN/MAX**: computes the min/max of the numeric values in a single-column bag.
    - **SIZE**: returns size of any datum example map
    - **CONCAT**: concatenate two chararrays or two bytearrays
    - **TOKENIZE**: splits a string and outputs a bag of words
    - **DIFF**: compares the fields of a tuple with arity 2

Pig Interactive Mode: Grunt Shell

- **Useful for learning Pig & for debugging**
  - Invoke grunt on hdfs
    - `pig`
  - Invoke grunt to use local file system only
    - `pig -x local`
- **Supports TAB completion and history**
- **Supports basic dfs commands**
  - `cd, ls, cp, mv, pwd, copyToLocal, copyFromLocal`
- **`set` command enables you to set key-value pairs**
  - Example: `set debug on`, `set job.name 'my job'`
- **Supports all Pig commands – easy to test & debug**
  - describe the alias
  - dump what the alias onto the terminal

```
grunt>sports_views = load 'sports_views.txt' as (userId:
   chararray,team:chararray,timestamp: int);
grunt>group_sports_views = group sports_views by userId;
grunt> describe group_sports_views;
group_sports_views: {group: chararray,sports_views: {userId: chararray,team:
   chararray,timestamp: integer}}
grunt> dump sports_views;
```

```
(alice,{(alice,lakers,3),(alice,lakers,7)})
```

Use EXPLAIN to Understand Logical, Physical & M/R Plan

```
grunt>sportsviews = load 'sportsviews.txt' as (userId: chararray,team:chararray,timestamp: int);
grunt>groupsportsviews = group sportsviews by userId;
grunt> describe group_sportsviews;
groupsportsviews: {group: chararray,sports_views: {userId: chararray,team: chararray,timestamp:
    integer}}
grunt> dump sportsviews;
(alice,{(alice,lakers,3),(alice,lakers,7)})
grunt> explain groupsportsviews
```

```
-----------------------------------------------
| Map Reduce Plan                             |
-----------------------------------------------
MapReduce node viraj-Sat Oct 25 20:38:23 UTC 2008-2261
Map Plan
Local Rearrange[tuple]{chararray}(false) - viraj-Sat Oct 25 20:38:23 UTC 2008-2258
|   |
|   Project[chararray][0] - viraj-Sat Oct 25 20:38:23 UTC 2008-2259
|
|---New For Each(false,false,false)[bag] - viraj-Sat Oct 25 20:38:23 UTC 2008-2255
    |   |
    |   Cast[chararray] - viraj-Sat Oct 25 20:38:23 UTC 2008-2250
    |   |
    |   |---Project[bytearray][0] - viraj-Sat Oct 25 20:38:23 UTC 2008-2249
    |   |
    |   Cast[chararray] - viraj-Sat Oct 25 20:38:23 UTC 2008-2252
    |   |
    |   |---Project[bytearray][1] - viraj-Sat Oct 25 20:38:23 UTC 2008-2251
    |   |
    |   Cast[integer] - viraj-Sat Oct 25 20:38:23 UTC 2008-2254
    |   |
    |   |---Project[bytearray][2] - viraj-Sat Oct 25 20:38:23 UTC 2008-2253
    |
    |---Load(sports_views.txt:org.apache.pig.builtin.PigStorage) - viraj-Sat Oct 25 20:38:23 UTC 2008-2248--------
Reduce Plan
Store(fakefile:org.apache.pig.builtin.PigStorage) - viraj-Sat Oct 25 20:38:23 UTC 2008-2260
|
|---Package[tuple]{chararray} - viraj-Sat Oct 25 20:38:23 UTC 2008-2257--------
Global sort: false
----------------
```