Pig, a high level data processing system on Hadoop

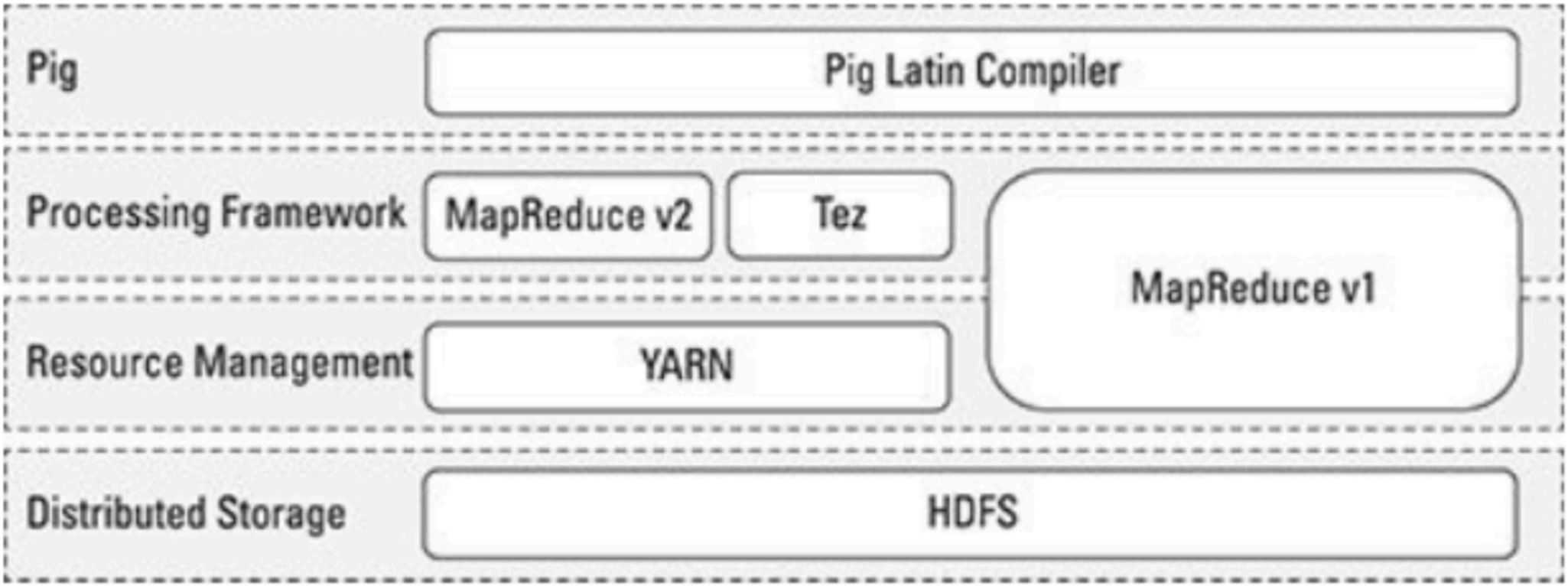
Apache Pig

- **Apache Pig** is a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs.
- Pig's infrastructure layer consists of
  - a compiler that produces sequences of Map-Reduce programs,
  - Pig's language layer currently consists of a textual language called Pig Latin, which has the following key properties:
    - **Ease of programming.** It is trivial to achieve parallel execution of simple, "embarrassingly parallel" data analysis tasks. Complex tasks comprised of multiple interrelated data transformations are explicitly encoded as data flow sequences, making them easy to write, understand, and maintain.
    - **Optimization opportunities.** The way in which tasks are encoded permits the system to optimize their execution automatically, allowing the user to focus on semantics rather than efficiency.
    - **Extensibility.** Users can create their own functions to do special-purpose processing.

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.

Pig



At its core, Pig Latin is a *dataflow* language, where you define a data stream and a series of transformations that are applied to the data as it flows through your application. This is in contrast to a *control flow* language (like C or Java), where you write a series of instructions. In control flow languages, we use constructs like loops and conditional logic (like an if statement). You won't find loops and if statements in Pig Latin.

Pig Latin

- Data flow language
  - User specify a sequence of operations to process data
  - More control on the process, compared with declarative language
- Various data types supports
- Schema supports
- User defined functions supports

MapReduce

- Large scale data processing framework
- Map phase + Reduce phase
- Proposed at 2004 by Google
- Variance and extension in open source community (Hadoop, Pig, Hive, etc.)

MapReduce Programming Model

- Programmers think in a data-centric fashion
  - Apply transformations on data sets
- The MR framework handles the Hard Stuff
  - Fault tolerance
  - Distributed execution, scheduling, concurrency
  - Coordination
  - Network communication

MapReduce not Good Enough?

- Restrict programming model
  - Only two phases
  - Job chain for long data flow
- Put the logic at the right phase
  - Programmers are responsible for this
- Too many lines of code even for simple logic
  - How many lines do you have for word count?

MapReduce vs. Pig cont.

- Join in MapReduce
  - Various algorithms. None of them are easy to implement in MapReduce
  - Multi-way join more complicated
  - Hard to integrate into SPJA workflow

MapReduce vs. Pig cont.

- Join in Pig
  - Various algorithms already available.
  - Some of them are generic to support multi-way join
  - No need to consider integration into SPJA workflow. Pig does that for you!

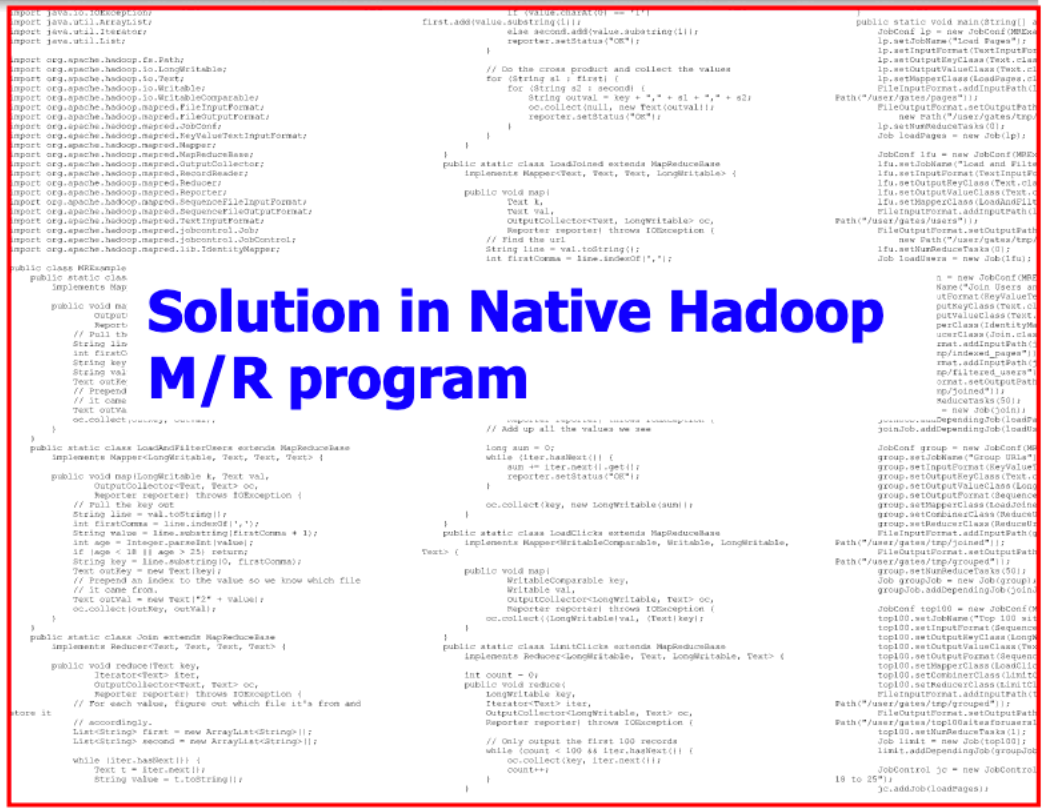
```
A = LOAD 'input/join/A';
B = LOAD 'input/join/B';
C = JOIN A BY $0, B BY $1;
DUMP C;
```

Pig to Rescure

- High level dataflow language (Pig Latin)
  - Much simpler than Java
  - Simplify the data processing
- Put the operations at the apropriate phases
- Chains multiple MR jobs

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



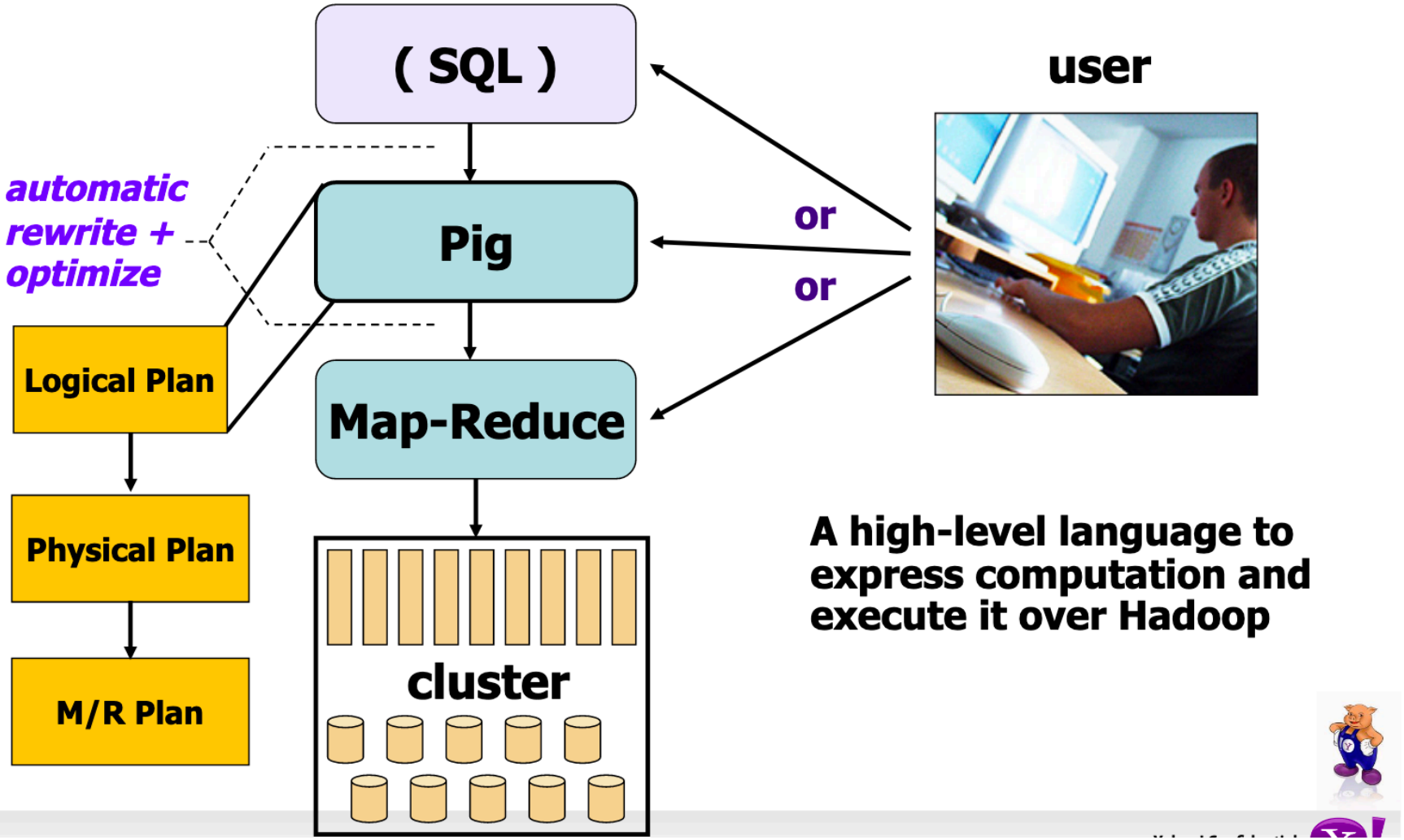

```
Users = LOAD 'users' AS (name, age);
Fltrd = FILTER Users by age >= 18 and age <= 25;
Pages = LOAD 'pages' AS (user, url
Jnd = JOIN Fltrd BY name, Pages BY
Grpd = GROUP Jnd by url;
Smmd = FOREACH Grpd GENERATE group
       clicks;
Srtd = ORDER Smmd BY clicks;
Top100 = LIMIT Srtd 100;
STORE Top100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

Solution using Pig Script

How Pig is used in Industry

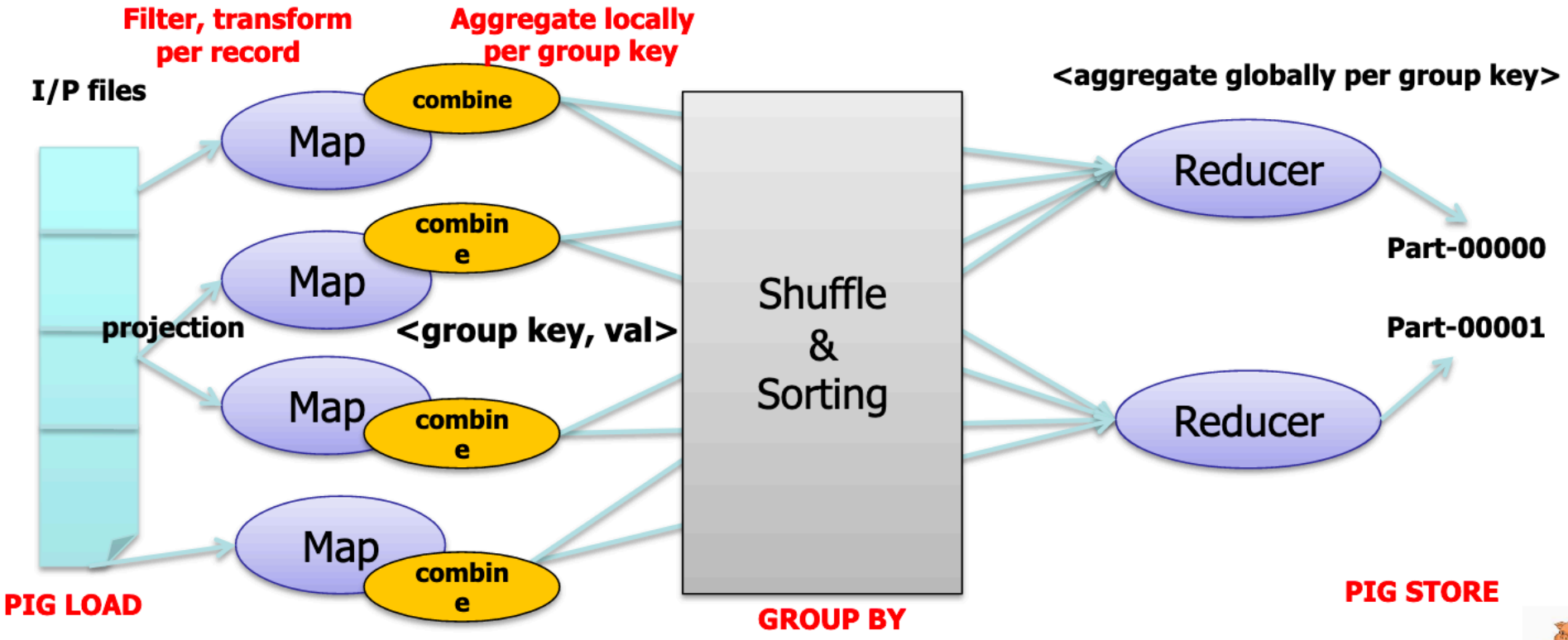
- At Yahoo, 70% MapReduce jobs are written in Pig
- Used to
  - Process web log
  - Build user behavior models
  - Process images
  - Data mining
- Also used by Twitter, Linkin, Ebay, AOL, etc.

Pig Architecture: Map-Reduce as Backend



SQL Operators to M/R

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

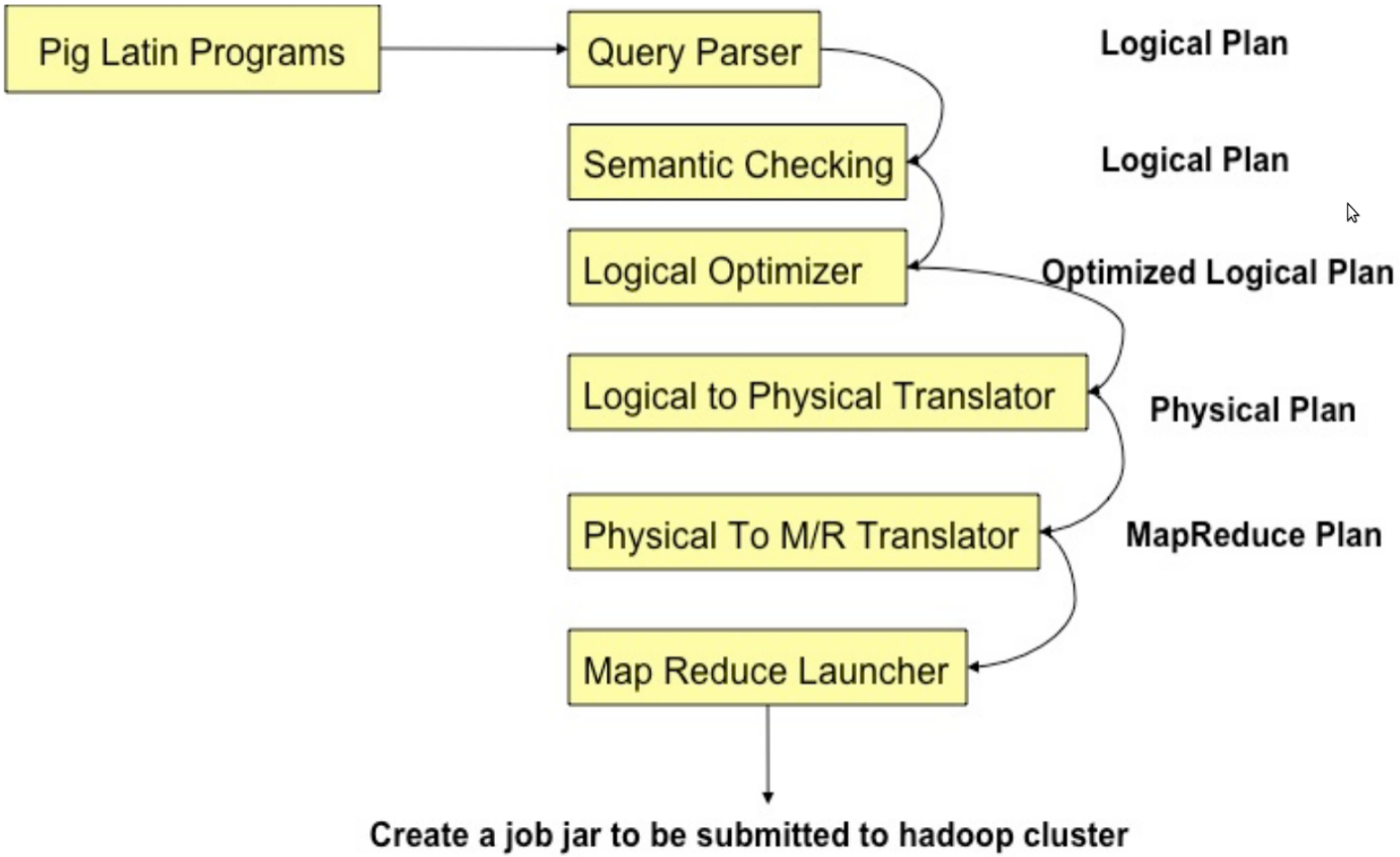


Hadoop M/R provides Distributed Group by Aggregate

Pig Execution Modes

- Local mode
  - Launch single JVM
  - Access local file system
  - No MR job running
- Hadoop mode
  - Execute a sequence of MR jobs
  - Pig interacts with Hadoop master node

Compilation



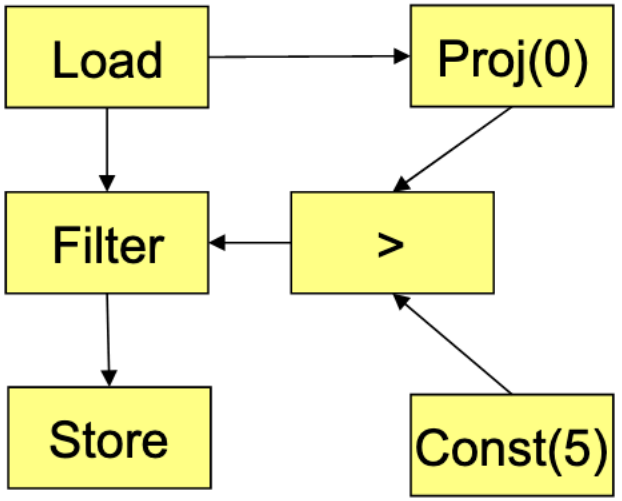
Parsing

- Type checking with schema
- References verification
- Logic plan generating
  - One-to-one fashion
  - Independent of execution platform
  - Limited optimization
  - No execution until DUMP or STORE

Logical Plan

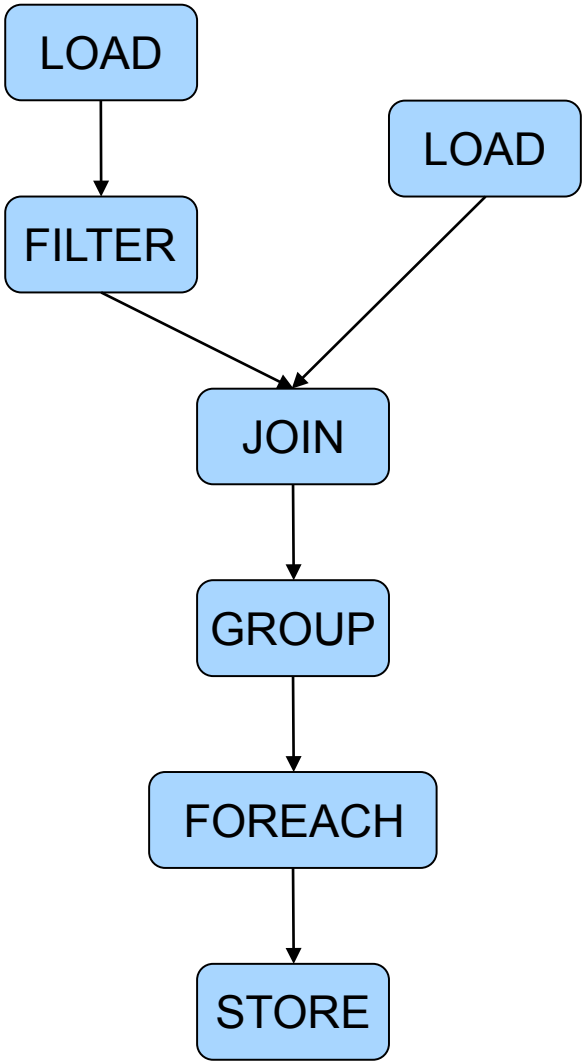
- Consists of DAG of Logical Operators as nodes and Data Flow represented as edges
  - Logical Operators contain list of i/p's o/p's and schema
- Logical operators
  - Aid in post parse stage checking (type checking)
  - Optimization
  - Translation to Physical Plan

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



Logical 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';
```

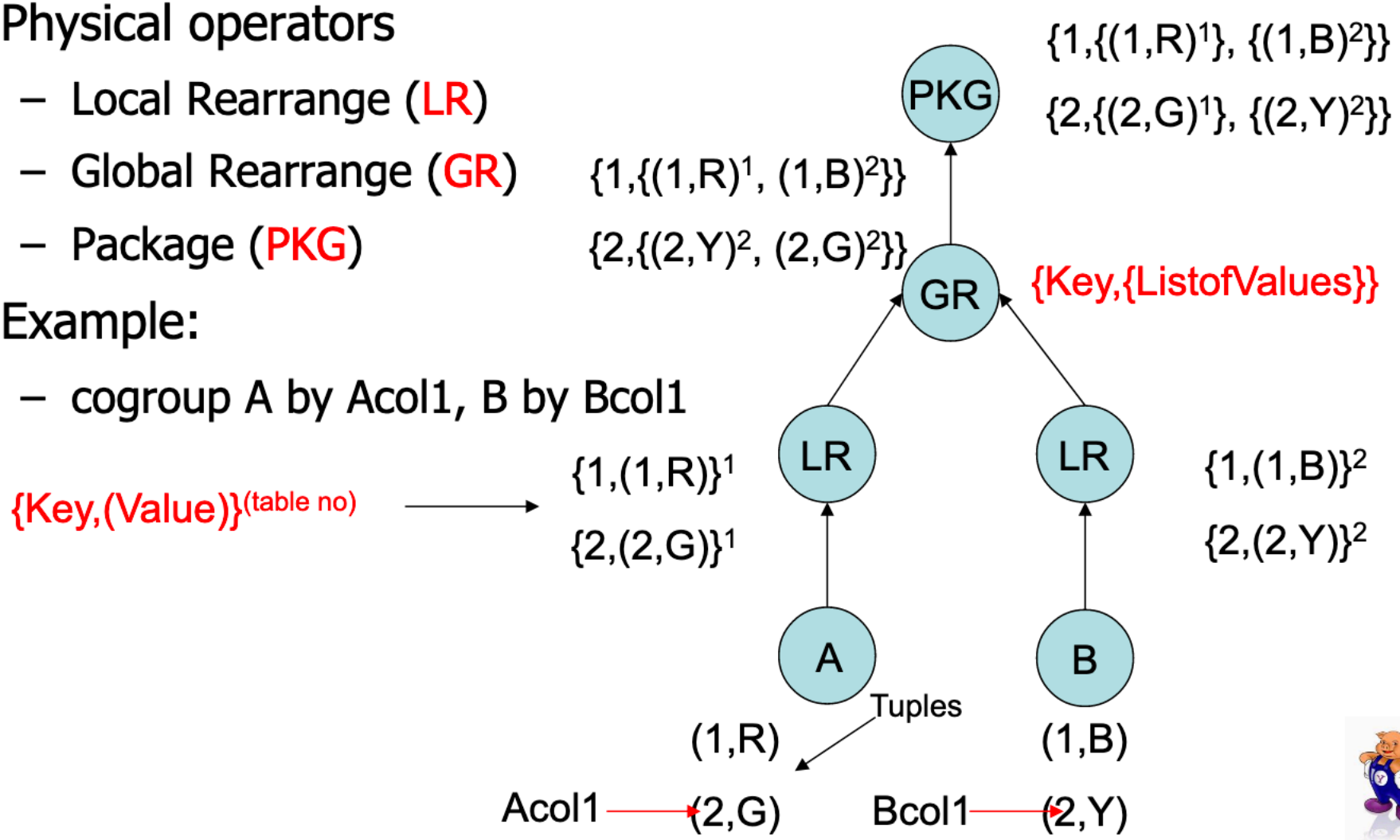


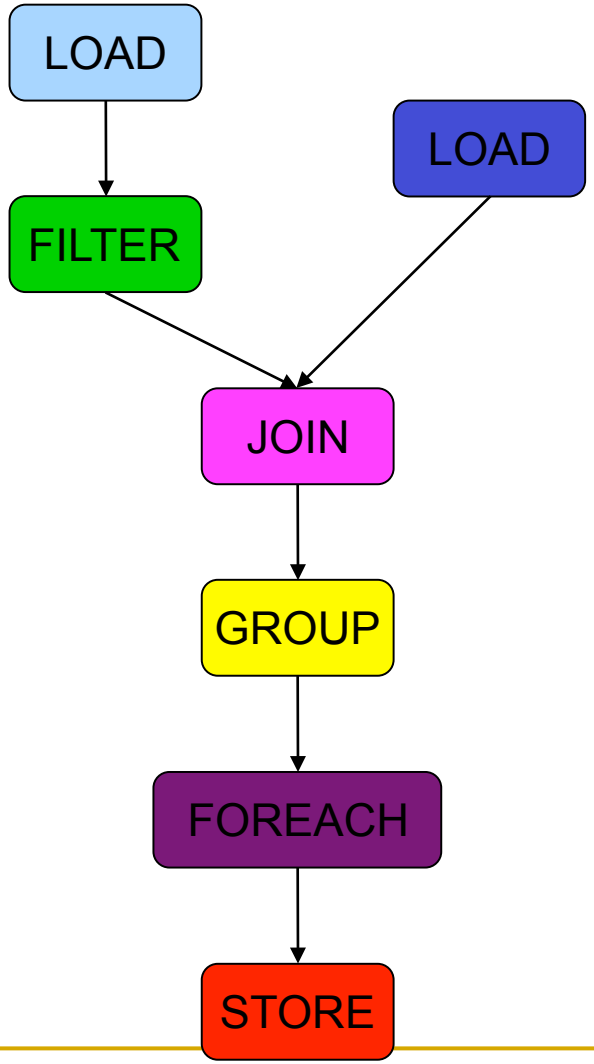
Physical Plan

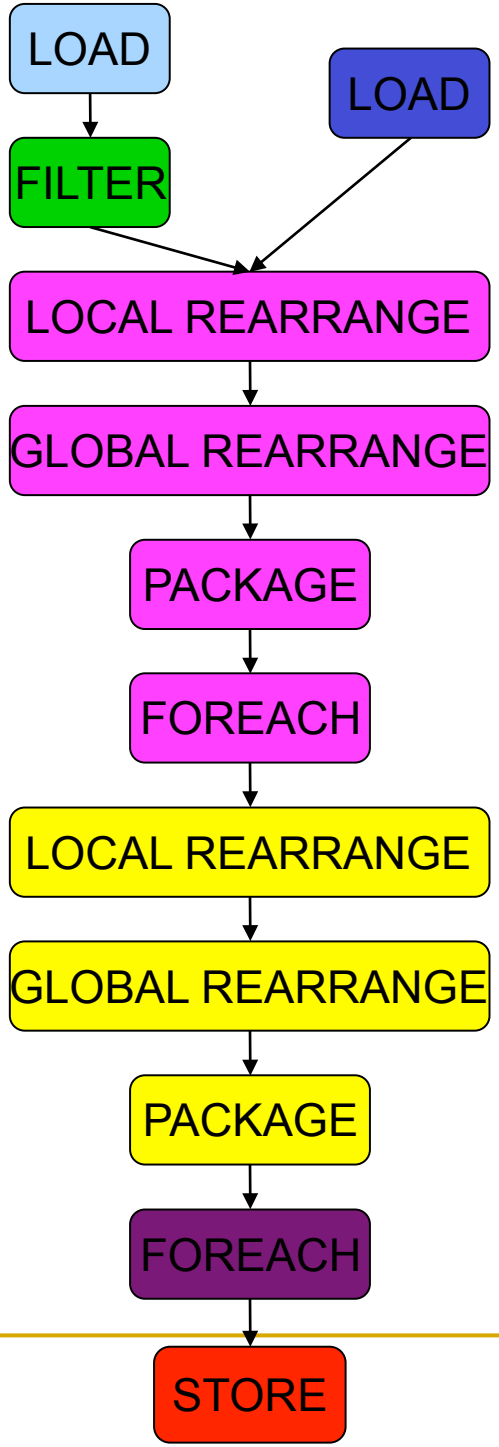
- Layer to map Logical Plan to multiple back-ends, one such being M/R (Map Reduce)
  - Chance for code re-use if multiple back-ends share same operator
- Consists of operators which Pig will run on the backend
- Currently most of the physical plan is placed as operators in the map reduce plan
- Logical 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

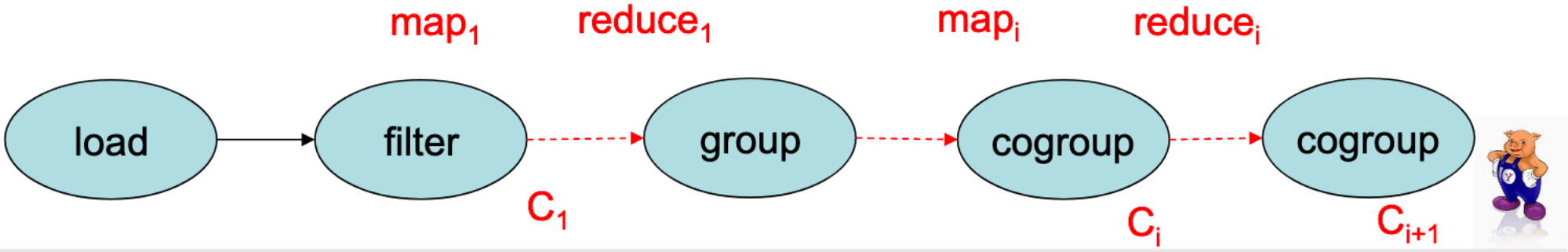



LOAD
LOAD
FILTER
JOIN
GROUP
FOREACH
STORE


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

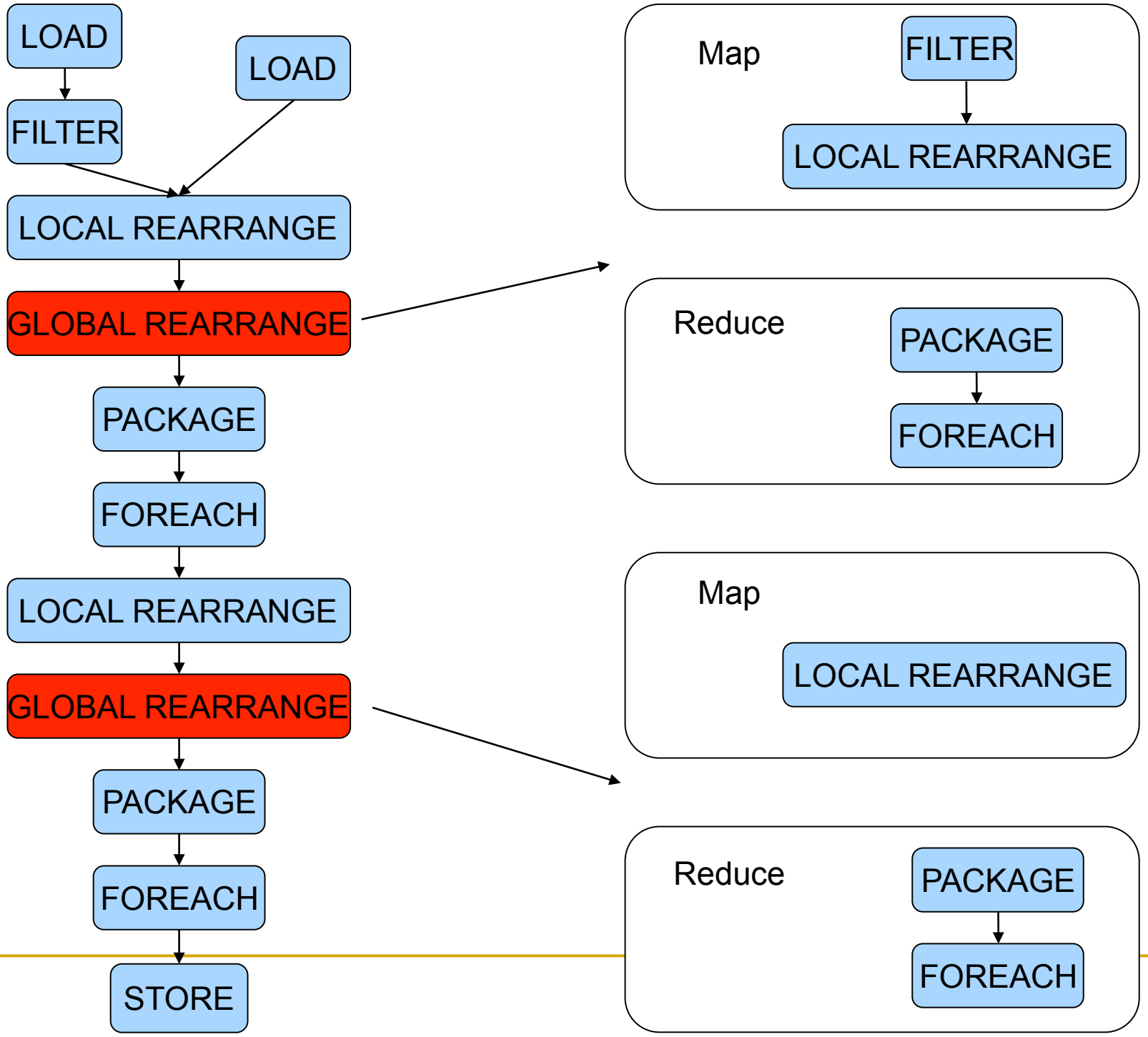
Map Reduce Plan

- 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



MapReduce Plan

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

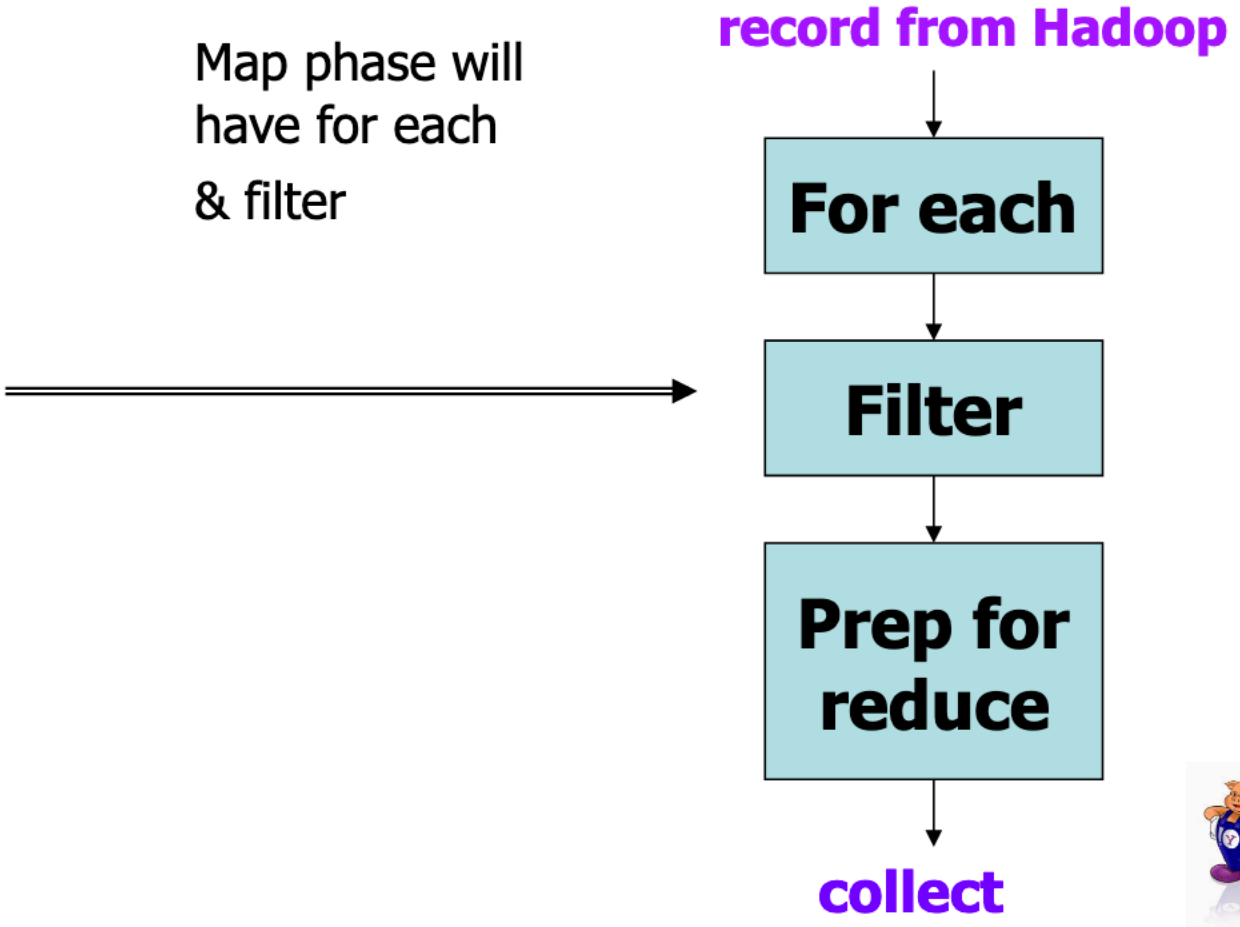

LOAD
LOAD
FILTER
LOCAL REARRANGE
GLOBAL REARRANGE
PACKAGE
FOREACH
LOCAL REARRANGE
GLOBAL REARRANGE
PACKAGE
FOREACH
STORE
Map
FILTER
LOCAL REARRANGE
Reduce
PACKAGE
FOREACH
Map
LOCAL REARRANGE
Reduce
PACKAGE
FOREACH

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

Schema

- User can optionally define the schema of the input data
- Once the schema of the source data is given, all the schema of the intermediate relation will be induced by Pig

Schema cont.

- Why schema?
  - Scripts are more readable (by alias)
  - Help system validate the input
- Similar to Database?
  - Yes. But schema here is optional
  - Schema is not fixed for a particular dataset, but changable

Schema cont.

- Schema 1

```
A = LOAD 'input/A' as (name:chararray, age:int);
B = FILTER A BY age != 20;
```

- Schema 2

```
A = LOAD 'input/A' as (name:chararray, age:chararray);
B = FILTER A BY age != '20';
```

- No Schema

```
A = LOAD 'input/A' ;
B = FILTER A BY A.$1 != '20';
```

Date Types

- Every attribute can always interpreted as bytearray, without further type definition
- Simple data types
  - For each attribute
  - Defined by user in the schema
  - Int, double, chararray ...
- Complex data types
  - Usually contructed by relational operations
  - Tuple, bag, map

Date Types cont.

- Type casting
  - Pig will try to cast data types when meets type inconsistency.
  - Warning will be thrown if casting fails. Process still goes on
- Validation
  - Null will replace the inconvertable data type in type casting
  - User can tell a corrupted record by detecting whether a particular attribute is null

Data types

- **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]`

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;

Functions

- Eval Functions
  - Record transformation
- Filter Functions
  - Test whether a record satisfy particular predicate
- Comparison Functions
  - Impose ordering between two records. Used by ORDER operation
- Load Functions
  - Specify how to load data into relations
- Store Functions
  - Specify how to store relations to external storage

Functions

- Built in Functions
  - Hard-coded routines offered by Pig.
- User Defined Function (UDF)
  - Supports customized functionalities
  - Piggy Bank, a warehouse for UDFs.
  - Re-think about Word Count in Pig

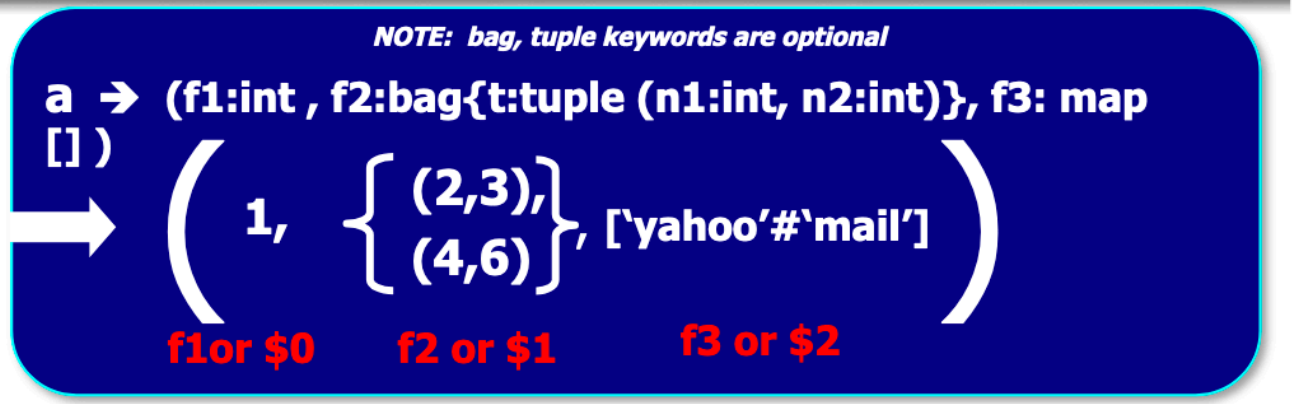
Pig Built-in Functions

- 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

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**

a → (f1:int , f2:bag{t:tuple (n1:int, n2:int)}, f3: map[])

(1, { (2,3), (4,6) } ,['yahoo'#'mail'])

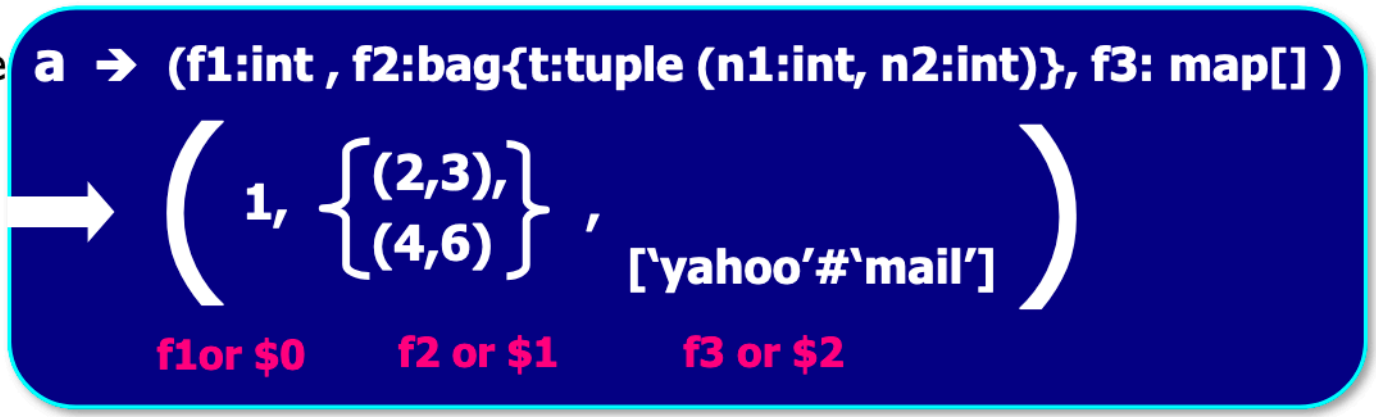
f1or $0 f2 or $1 f3 or $2

- **==, !=, >, >=, <, <= 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'

Statement

- A statement represents an operation, or a stage in the data flow.
- Usually a variable is used to represent the result of the statement
- Not limited to data processing operations, but also contains filesystem operations

Operators

- Relational Operators
  - Represent an operation that will be added to the logical plan
  - LOAD, STORE, FILTER, JOIN, FOREACH...GENERATE

```
grunt> DUMP A;
(2,Tie)
(4,Coat)
(3,Hat)
(1,Scarf)
grunt> DUMP B;
(Joe,2)
(Hank,4)
(Ali,0)
(Eve,3)
(Hank,2)
```

```
grunt> C = JOIN A BY $0, B BY $1;
grunt> DUMP C;
(2,Tie,Joe,2)
(2,Tie,Hank,2)
(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

Operators

- Diagnostic Operators
  - Show the status/metadata of the relations
  - Used for debugging
  - Will not be integrated into execution plan
  - DESCRIBE, EXPLAIN, ILLUSTRATE.

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>>   AS (year, temperature:int, quality:int);
grunt> DESCRIBE records;
records: {year: bytearray,temperature: int,quality: int}
```

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)})
```

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

(Co)Group Data

Data 1:

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

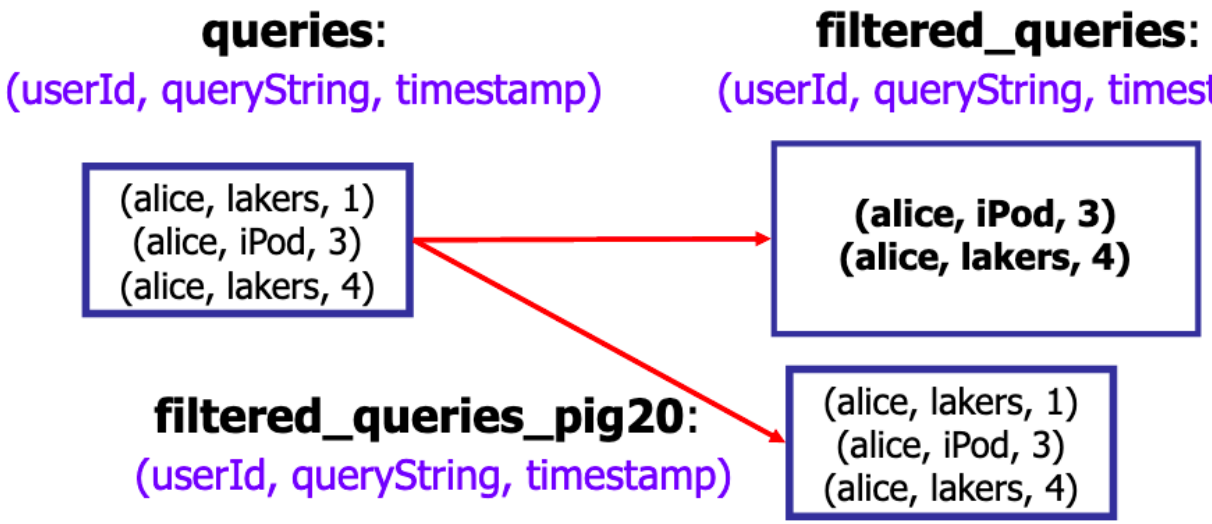
Data 2:

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);
```

Filter Data

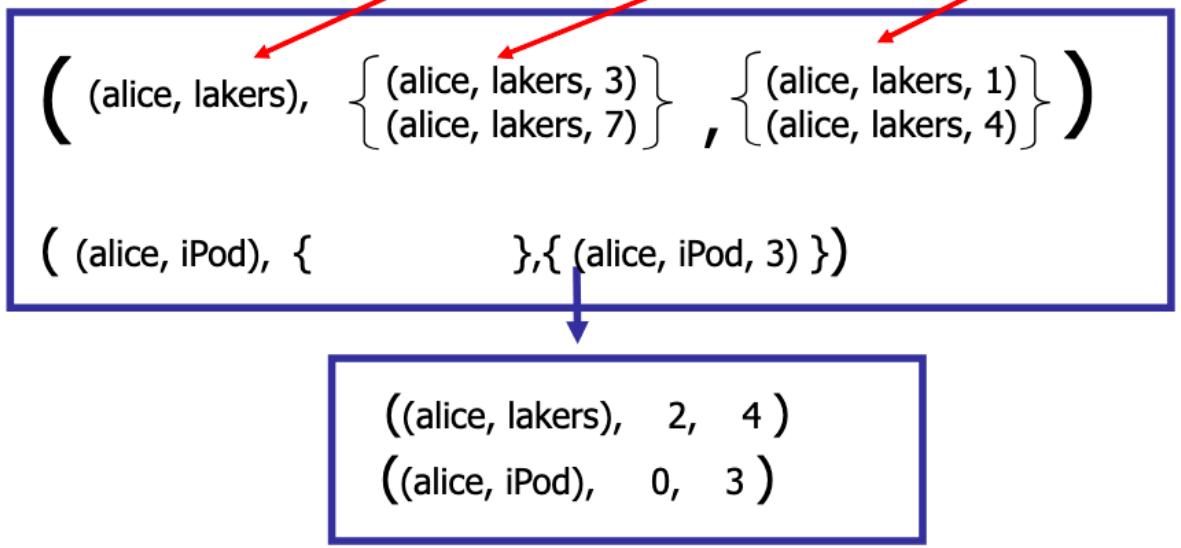
```
queries = load 'query_log.txt' using PigStorage('\u0001') as (userId:
chararray, queryString: chararray, timestamp: int);

filtered_queries =  filter queries by timestamp > 1;

filtered_queries_pig20 = filter queries by timestamp is not null;

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



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, -, -)

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`;
```

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;
```

```
(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.com,10:05,www.flickr.com,0.9F)
(Amy,www.myblog.com,10:00,www.myblog.com,0.7F)
(Fred,cnn.com/index.htm,12:00,cnn.com/index.htm,0.1F)
(Fred,cnn.com/index.htm,13:00,cnn.com/index.htm,0.1F)
```

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

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
----------------
```