PIG – Latin Language

Presented By,
Siva Kumar Bhuchipalli

"Action Is The Foundational Key
To All Success "

PIG

- Pig is an open source data flow language
- Platform for analysing large sets of data
- Pig Latin is used to express the queries and data manipulation operations in simple scripts
- Pig converts the scripts into a sequence of underlying M/R jobs.
- Pig is used to process different file formats like csv file , JSON file , Log records from web server and records from database

Why PIG?

- It is a data flow language
- Provides standard data processing operations
- Insulates Hadoop complexity
- Abstracts Map Reduce
- Increases programmer productivity
- Similar SQL syntax
- … but there are cases where Pig is not suitable

If We Use PIG.

```
Users          = load ‘users.csv’ as (username: chararray, age: int);
Users_1825 = filter Users by age >= 18 and age <= 25;

Pages          = load ‘pages.csv’ as (username: chararray, url: chararray);

Joined         = join Users_1825 by username, pages by username;
Grouped      = group Joined by url;
Summed      = foreach Grouped generate group as url, COUNT(Joined) As  Views;
Sorted         = order Summed by Views desc;
Top_5          = limit Sorted 5;

Store Top_5 into ‘’top_5_sites.csv’ ;
```

If We Use MapReduce.

PIG Vs MR

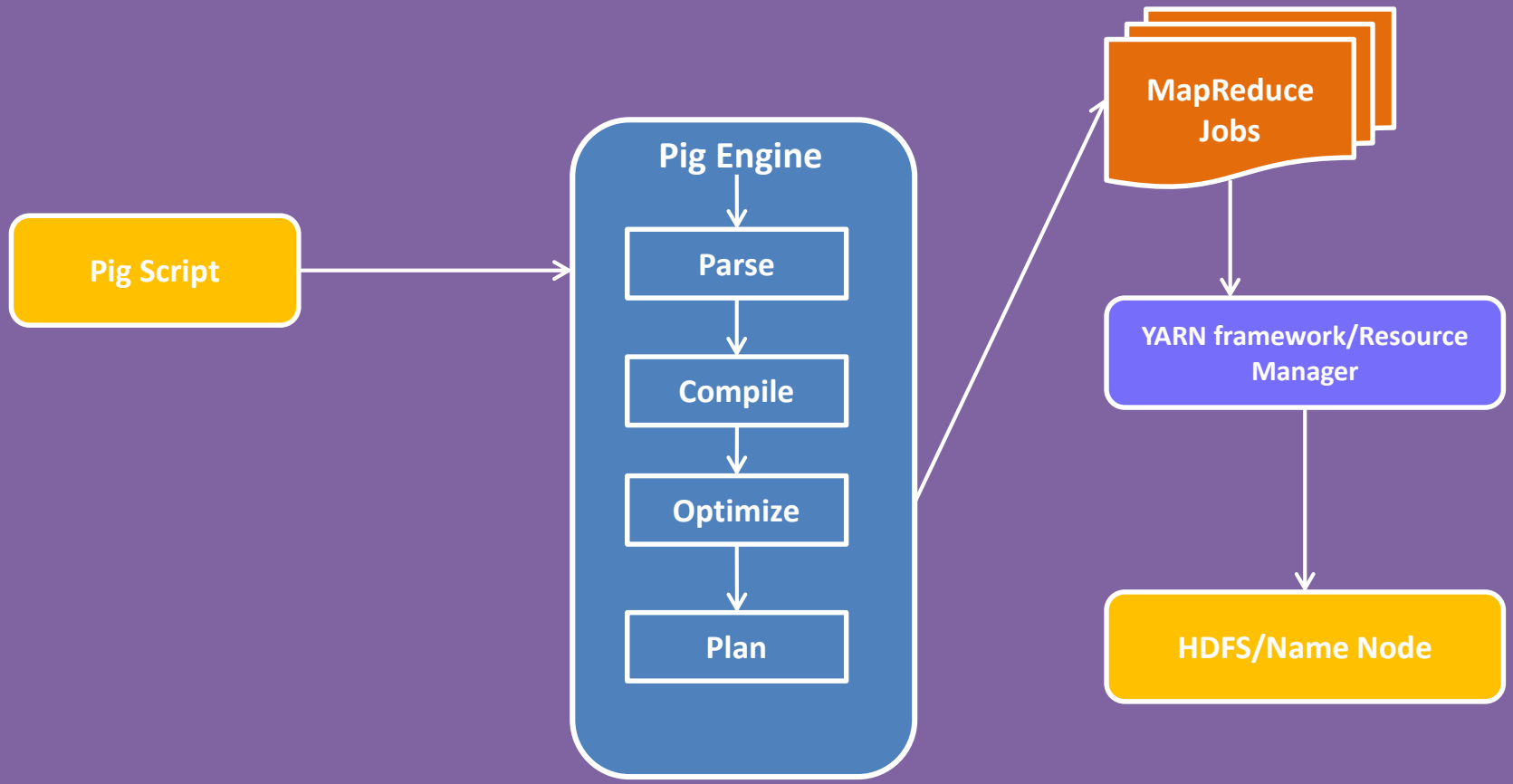
- 1/20th lines of code
- 1/16th development time
- Map reduce provides powerful mechanism for parallel computation
- Map reduce gives more control on algorithm execution
- very rigid in structure (if syntax is missed doesn't accept)
- Pig act as higher level language over Map reduce , insulates programmers from Underlying complexity Hadoop concepts (It doesn't care what happens on Framework)
- Provides seamless integration with a range of underlying Hadoop components

Why Not To Use PIG ?

Pig is not to be used when,

- **Input data format is really nasty (video , audio etc)**
- **We need more fine grained control on processing**
- **Pig lacks control structures, so loops are not possible and if/else or case statements are difficult**

PIG Architecture ?



Modes Of PIG

- **Local mode :** pig -x local

In this mode, entire pig job runs in a single jvm process
Picks and stores data from local Linux path

- **Map reduce :** pig -x mapreduce

In this mode, pig job runs as a series of map reduce job
Input and output paths are assumed as HDFS paths

- **TEZ :** pig -x tez

In this mode, tez job runs as a series of tez jobs
Input and output paths are assumed as HDFS paths

2Pieces Of PIG

- **Pig Latin** - Language used to express data flows
- **Pig Engine** - Execution environment(to run Pig Latin programs): 2 types:

- A Pig Latin program is made up of a series of operations, or transformations that are applied to the input data to produce output

- Under the covers, Pig runs the transformations into a series of Map Reduce jobs

- Virtually, all pig latin operations(loading/storing/filtering/grouping/joining) can be altered by UDFs(user defined functions)

Help

Prints a list of Pig commands or properties.

Example
Use "-help" to get a list of commands.

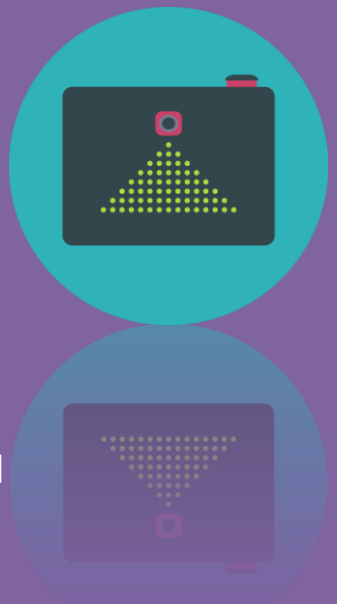
```
$ pig -help
Apache Pig version 0.8.0-dev (r987348)
compiled Aug 19 2010, 16:38:44
USAGE: Pig [options] [-] : Run interactively in grunt shell.
       Pig [options] -e[xecute] cmd [cmd ...] : Run cmd(s).
       Pig [options] [-f[ile]] file : Run cmds found in file.
  options include:
    -4, -log4jconf - Log4j configuration file, overrides log conf
    -b, -brief - Brief logging (no timestamps)
    -c, -check - Syntax check
etc
```

Loading

Storing

Pig Eats Anything

Loading



- PigStorage – for most cases
- TextLoader – to load text files
- JSONLoader – to load JSON files
- Custom loaders – You can write your own custom loaders as well

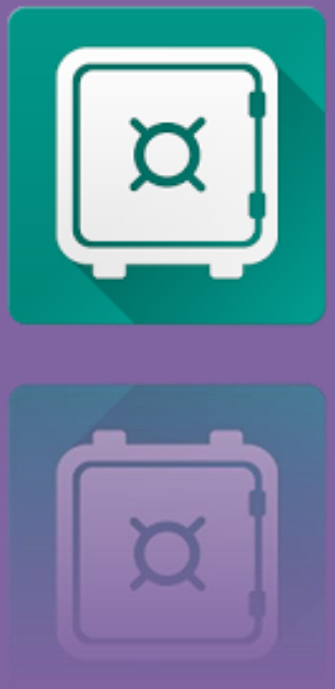
```
file = LOAD 'data/dropbox-policy.txt' AS (line);

data = LOAD 'data/tweets.csv' USING PigStorage(',');

data = LOAD 'data/tweets.csv' USING PigStorage(',') AS (list, of, fields);
```

Viewing Data

DUMP input;

Very useful for debugging, but don't use it on huge datasets.
Always use it after limiting your dataset.

Storing



- Similar to `LOAD`, lot of options are available
- Can store locally or in HDFS
- You can write your own custom Storage as well

```
STORE data INTO 'output_location';

STORE data INTO 'output_location' USING PigStorage();

STORE data INTO 'output_location' USING PigStorage(',');

STORE data INTO 'output_location' USING BinStorage();
```

Example:

```
data = LOAD 'data/data-bag.txt' USING PigStorage(',');

STORE data INTO 'data/output/load-store' USING PigStorage('|');
```

PIG Latin Language

Pig Basic Data Type

- int, long – (32, 64 bit) integer
- float, double – (32, 64 bit) floating point
- boolean (true/false)
- chararray (String in UTF-8)
- bytearray (blob) (DataByteArray in Java)
- If you don't specify anything bytearray is used by default

Complex Data Types

- tuple – ordered set of fields
- (data) bag – collection of tuples
- map – set of key value pairs

Tuple

- Tuple is an ordered set of atoms/fields
- Fields (or atoms) can be of any data type
- Ordering is important
- Enclosed inside parentheses ()

e.g. ('a', 'big', 'data', 'problem'), (Brad, 176, 80.2F)

Bag

- Set of tuples
- SQL equivalent is Table
- Each tuple can have different set of fields
- Can have duplicates
- Inner bag uses curly braces {}
- Outer bag doesn't use anything

Example Outer bag

(1,2,3)
(1,2,4)
(2,3,4)
(3,4,5)
(4,5,6)

Inner bag

(1,{(1,2,3),(1,2,4)})
(2,{(2,3,4)})
(3,{(3,4,5)})
(4,{(4,5,6)})

Map

- Set of key value pairs
- Similar to HashMap in Java
- Key must be unique
- Key must be of chararray data type
- Values can be any type
- Key/value is separated by #
- Map is enclosed by []

[name#James, height#176, weight#80.5F]

[name#(James, Anderson), height#176, weight#80.5F]

[name#(James, Anderson), languages#(Java, Pig, Python)]

MAP

```
A = LOAD 'complex_map.txt' AS (M:map []);

DUMP A;
        [open#apache]
        [apache#hadoop]
DESCRIBE A;
```

BAG

```
A = LOAD 'complex1_bag.txt' AS (B: bag {T: tuple(t1:int,t2:int,
t3:int)});

DUMP A;
        {(3,8,9)}
        {(1,4,7)}
        {(2,5,8)}
```

Tuple

```
A =  LOAD 'complex_tuple.txt' AS (t1:tuple(t1a:int,t1b:int,t1c:int),t2:tuple(t2a:int,t2b:int,t2c:int));

DUMP A;
        (3,8,9) (4,5,6)
        (1,4,7) (3,7,5)
        (2,5,8) (9,5,8)
X = FOREACH A GENERATE t1.t1a,t2.$0;
```

Accessing Fields Example

```
data = LOAD 'data/nested-schema.txt' AS (f1:int, f2:bag{t:tuple(n1:int, n2:int)}, f3:map[]);

by_pos = FOREACH data GENERATE $0;
DUMP by_pos;

by_field = FOREACH data GENERATE f2.n1;
DUMP by_field;

by_map = FOREACH data GENERATE f3#'name';
DUMP by_map;
```

Arithmetic Operators

X = FOREACH A GENERATE f1, f2, f1+f2;

X = FOREACH A GENERATE f1, f2, f1-f2;

X = FOREACH A GENERATE f1, f2, f1/f2;

X = FOREACH A GENERATE f1, f2, f1%f2;

X = FOREACH A GENERATE f1, f2, f1*f2;

Logical Operators

AND

X = FILTER A BY (f1==8) AND (f2==8);

OR

X = FILTER A BY (f1==8) OR (f2==8);

Not

X = FILTER A BY (f1==8) OR (NOT(f2+f3>f1));

Comparison Operators

Equals

X = FILTER A BY (f1 == 8);

Not Equal

X = FILTER B BY (f2 != 9);

Less than

X = FILTER C BY (f3 < 8);

Greater than

X = FILTER D BY (f2 < 9);

Less than Or Equal to

X = FILTER A BY (f2 <= 3);

Relational Operators

- FOREACH
- FLATTEN
- GROUP
- FILTER
- COUNT
- ORDER BY
- DISTINCT
- LIMIT
- JOIN
- Etc…

FOREACH

- Generates data transformations based on columns of data.
- Similar to SELECT col1, col2 FROM table;

```
X = FOREACH data GENERATE f1, f2;
```

Examples:

```
x = FOREACH data GENERATE *;

x = FOREACH data GENERATE $0, $1;

x = FOREACH data GENERATE $0 AS first, $1 AS second;
```

GROUP

- Groups the data in one or more relations.
- Groups tuples that have the same group key
- Similar to SQL group by operator

```
A = load 'student' AS (name:chararray, age:int, gpa:float);
DUMP A;
        (John,18,4.0F)
        (Mary,19,3.8F)
        (Bill,20,3.9F)
        (Joe,18,3.8F)
```

```
B = GROUP A BY age;
DUMP B;
        (18,{(John,18,4.0F),(Joe,18,3.8F)})
        (19,{(Mary,19,3.8F)})
        (20,{(Bill,20,3.9F)})
```

Examples:

```
outerbag = LOAD 'data/data-bag.txt' USING
PigStorage(',') AS (f1:int, f2:int, f3:int);
DUMP outerbag;

innerbag = GROUP outerbag BY f1;
DUMP innerbag;
```

FLATTEN

Flatten un-nests tuples as well as bags

```
X = LOAD '/home/user/data' AS (value : chararray);
DUMP X;
        {(a,b),(c,d)}


Y = FOREACH X GENERATE FLATTEN X;
DUMP Y;
        (a,b) and (c,d)
```

Works:

(a, (b, c)) => (a,b,c)

({(a,b),(d,e)}) => (a,b) and (d,e)

(a, {(b,c), (d,e)}) => (a, b, c) and (a, d, e)

COGROUP

GROUP is used in statements involving one relation and COGROUP is used in statements involving two or more relations.

```
A = LOAD 'data1' AS (owner:chararray, pet:chararray);
DUMP A;
        (Alice,turtle)
        (Alice,goldfish)
        (Alice,cat)
        (Bob,dog)
        (Bob,cat)

B = LOAD 'data2' AS (friend1:chararray,friend2:chararray);
DUMP B;
        (Cindy,Alice)
        (Mark,Alice)
        (Paul,Bob)
        (Paul,Jane)

 X = COGROUP A BY owner, B BY friend2;
DUMP X;
        (Alice,{(Alice,turtle),(Alice,goldfish),(Alice,cat)},{(Cindy,Alice),(Mark,Alice)})
        (Bob,{(Bob,dog),(Bob,cat)},{(Paul,Bob)})
        (Jane,{},{(Paul,Jane)})
```

COUNT

```
data = LOAD 'data/data-bag.txt' USING PigStorage(',') AS (f1:int, f2:int, f3:int);
grouped = GROUP data BY f2;
counted = FOREACH grouped GENERATE group, COUNT (data);
DUMP counted;
```

ORDER BY

```
A = LOAD 'data' AS (a1:int,a2:int,a3:int);
DUMP A;
        (1,2,3)
        (4,2,1)
        (8,3,4)

 X = ORDER A BY a3 DESC;
        (8,3,4)
        (1,2,3)
        4,2,1)
```

FILTER

```
A = LOAD 'data' AS (a1:int,a2:int,a3:int);
DUMP A;
        (1,2,3)
        (4,2,1)
        (4,3,3)
        (8,4,3)

X = FILTER A BY f3 == 3;
DUMP X;
        (1,2,3)
        (4,3,3)
        (8,4,3)
```

RANK

```
A = load 'data' AS (f1:chararray,f2:int,);
DUMP A;
        (John,1)
        (merge,2)
        (Bob,3)

B = RANK A;
        (1,John,1)
        (2,merge,2)
        (3,Bob,3)
```

DISTINCT

Removes duplicates from a relation

```
A = LOAD 'data' AS (a1:int,a2:int,a3:int);
        (8,3,4)
        (1,2,3)
        (4,3,3)
        (4,3,3)

X = DISTINCT A;
        (1,2,3)
        (4,3,3)
        (8,3,4)
```

LIMIT

```
A = LOAD 'data' AS (a1:int,a2:int,a3:int);
        (1,2,3)
        (4,2,1)
        (8,3,4)

X = LIMIT A 2;
        (1,2,3)
        (8,3,4)
```

CROSS

```
A = LOAD 'data1' AS (a1:int,a2:int,a3:int);
        (1,2,3)
        (4,2,1)

B = LOAD 'data2' AS (b1:int,b2:int);
DUMP B;

X = CROSS A, B;
        (1,2,3,2,4)
        (1,2,3,8,9)
        (1,2,3,1,3)
        (4,2,1,2,4)
        (4,2,1,8,9)
        (4,2,1,1,3)
```

LOAD

```
A = LOAD ‘myfile.txt' USING PigStorage(‘\t’);
DUMP A;
        (1,2,3)
        (4,2,1)
        (8,3,4)
```

STORE

```
A= LOAD 'data' AS (a1:int,a2:int,a3:int);
DUMP A;
        (1,2,3)
        (4,2,1)
        (8,3,4)
        (4,3,3)

STORE A INTO 'myoutput' USING PigStorage ('*');
CAT myoutput;
        1*2*3
        4*2*1
        8*3*4
        4*3*3
```

SPLIT

```
A = LOAD 'data' AS (f1:int,f2:int,f3:int);
        (1,2,3)
        (4,5,6)
        (7,8,9)

SPLIT A INTO X IF f1<7, Y IF f2==5, Z IF (f3<6 OR f3>6);
        (1,2,3)
        (4,5,6)

        (4,5,6)

        (1,2,3)
        (7,8,9)
```

SAMPLE

```
A = LOAD 'data' AS (f1:int,f2:int,f3:int);
        (2,3,4)
        (5,6,7)

X = SAMPLE A 0.50;
        (2,3,4)
```

UNION

```
A = LOAD 'data' AS (a1:int,a2:int,a3:int);
DUMP A;
        (1,2,3)
        (4,2,1)

B = LOAD 'data' AS (b1:int,b2:int);
DUMP B;
        (2,4)
        (8,9)

X = UNION A, B;
DUMP X;
        (1,2,3)
        (4,2,1)
        (2,4)
        (8,9)
```

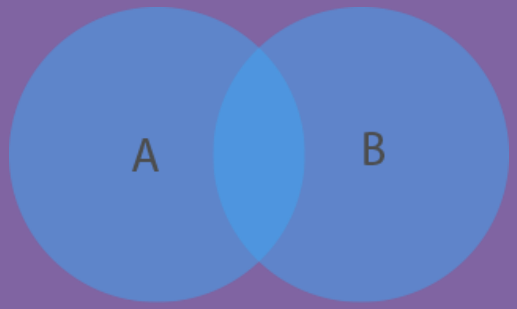
INNER JOIN

Performs an inner join of two or more relations based on common field values.

```
A = LOAD 'data1' AS (a1:int,a2:int,a3:int);
DUMP A;
        (1,2,3)
        (4,2,1)
        (4,3,3)
        (7,2,5)
```

```
B = LOAD 'data2' AS (b1:int,b2:int);
DUMP B;
        (2,4)
        (2,7)
        (2,9)
        (4,6)
        (4,9)
```

```
X = JOIN A BY a1, B BY b1;
DUMP X;
        (4,2,1,4,6)
        (4,3,3,4,6)
        (4,2,1,4,9)
        (4,3,3,4,9)
```



OUTER JOIN

Left Outer Join

```
A = LOAD 'a.txt' AS (n:chararray, a:int);
B = LOAD 'b.txt' AS (n:chararray, m:chararray);
C = JOIN A by $0 LEFT OUTER, B BY $0;
```

Full Outer Join

```
A = LOAD 'a.txt' AS (n:chararray, a:int);
B = LOAD 'b.txt' AS (n:chararray, m:chararray);
C = JOIN A BY $0 FULL, B BY $0;
```

Right Outer Join

```
A = LOAD 'a.txt' AS (n:chararray, a:int);
B = LOAD 'b.txt' AS (n:chararray, m:chararray);
C = JOIN A by $0 RIGHT OUTER, B BY $0;
```

AVG

Calculates the average of the numeric values

```
A = LOAD '/home/user/Desktop/DATA.txt' USING PigStorage (',') as
(name:chararray, sal:float);
DUMP A;
        (John,300)
        (John,300)
        (John,400)
        (John,300)
        (Mary,300)
        (Mary,300)
        (Mary,400)
        (Mary,400)

B = GROUP A BY name;
DUMP B;
        (John,{(John,300),(John,400),(John,300),(John,300)})
        (Mary,{(Mary,400),(Mary,400),(Mary,300),(Mary,300)})

C = FOREACH B GENERATE A.name, AVG(A.sal);
DUMP C;
        ({(John),(John),(John),(John)},325.0)
        ({(Mary),(Mary),(Mary),(Mary)},350.0)
```

CONCAT

Concatenates two expressions of identical type.

```
A = LOAD '/home/user/Desktop/concatinate_txt' using PigStorage (',') as (name:chararray,
nic:chararray);
DUMP A;
        (sandy,candy)
        (ramu,bheemu)

B = FOREACH A GENERATE CONCAT(name,nic);
DUMP B;
        (sandycandy)
        (ramubheemu)
```

TOKENIZE

Splits a string and outputs a bag of words.

```
A = LOAD '/home/user/Desktop/concatinate_txt' AS (f1:chararray);
DUMP A;
        (my name is radha)
        (my name is rekha)
        (my name is rasi)

X = FOREACH A GENERATE TOKENIZE(f1);
DUMP X;
        ({(my),(name),(is),(radha)})
        ({(my),(name),(is),(rekha)})
        ({(my),(name),(is),(rasi)})
```

SUM

Computes the sum of the numeric values in a single-column bag. SUM requires a preceding GROUP ALL statement for global sums and a GROUP BY statement for group sums.

```
A = LOAD '/home/user/Desktop/DATA.txt' USING PigStorage (',') as (name:chararray, sal:int);
DUMP A;
        (John,300)
        (John,300)
        (John,400)
        (John,300)
        (Mary,300)
        (Mary,300)
        (Mary,400)
        (Mary,400)

B = GROUP A BY name;
        (John,{(John,300),(John,400),(John,300),(John,300)})
        (Mary,{(Mary,400),(Mary,400),(Mary,300),(Mary,300)})

C = FOREACH B GENERATE A.NAME, SUM(A.sal);
DUMP C;
        ({(John),(John),(John),(John)},1300)
        ({(Mary),(Mary),(Mary),(Mary)},1400)
```

SIZE

Computes the number of elements based on any Pig data type.

```
A = LOAD '/home/user/Desktop/DATA.txt' USING PigStorage (',') as (name:chararray, sal:int);
DUMP A;
        (John,300)
        (Jockey,300)
        (James,400)

B = FOREACH A GENERATE SIZE(name);
DUMP B;
        (4)
        (6)
        (5)
```

MIN

Computes the minimum of the numeric values or chararrays in a single-column bag. MIN requires a preceding GROUP ALL statement for global minimums and a GROUP BY statement for group minimums.

```
A = LOAD '/home/user/Desktop/DATA.txt' USING PigStorage (',') as (name:chararray, sal:int);
DUMP A;
        (John,300)
        (John,400)
        (Mary,400)
        (Mary,300)

B = GROUP A BY name;
        (John,{(John,300),(John,400)})
        (Mary,{(Mary,400),(Mary,300)})

C = FOREACH B GENERATE A.name, MIN(A.sal);
DUMP C;
        ({(John),(John)},300)
        ({(Mary),(Mary)},300)
```

MAX

Computes the maximum of the numeric values or chararrays in a single-column bag. MAX requires a preceding GROUP ALL statement for global maximums and a GROUP BY statement for group maximums

```
A = LOAD '/home/user/Desktop/DATA.txt' USING PigStorage (',') as (name:chararray, sal:int);
DUMP A;
        (John,300)
        (John,400)
        (Mary,400)
        (Mary,300)

B = GROUP A BY name;
        (John,{(John,300),(John,400)})
        (Mary,{(Mary,400),(Mary,300)})

C = FOREACH B GENERATE A.name, MAX(A.sal);
DUMP C;
        ({(John),(John)},400)
        ({(Mary),(Mary)},400)
```

IsEMPTY

Checks if a bag or map is empty.

```
A = LOAD '/home/user/Desktop/DATA.txt' USING PigStorage (',') AS (name:chararray, sal:int);
DUMP A;
        (John,300)
        (John,400)
        (Mary,400)

B = LOAD '/home/user/Desktop/concatinate_txt' AS (sal:int);
DUMP B;
        (500)
        (600)
        (400)

C = JOIN B BY sal LEFT OUTER, A BY sal;
DUMP C;
        (400,Mary,400)
        (400,John,400)
        (500,,)
        (600,,)

Y = filter C by IsEmpty(name);
DUMP Y;
/* only keep those SAL for which there is no name */
        (100,,)
        (500,,)
        (600,,)
```

DIFF

Compares two fields in a tuple

In this example DIFF compares the tuples in two bags.

```
A = LOAD 'bag_data' AS (B1:bag{T1:tuple(t1:int,t2:int)},B2:bag{T2:tuple(f1:int,f2:int)});
DUMP A;
        ({(8,9),(0,1)},{(8,9),(1,1)})
        ({(2,3),(4,5)},{(2,3),(4,5)})
        ({(6,7),(3,7)},{(2,2),(3,7)})

DESCRIBE A;
        a: {B1: {T1: (t1: int,t2: int)},B2: {T2: (f1: int,f2: int)}}

X = FOREACH A GENERATE DIFF(B1,B2);
DUMP X;
        ({(0,1),(1,1)})
        ({})
        ({(6,7),(2,2)})
```

COUNT

Computes the number of elements in a bag.

```
A = LOAD '/home/user/Desktop/data_txt' USING PigStorage (',') AS (f1:int,f2:int,f3:int);
DUMP A;
        (1,2,3)
        (4,2,1)
        (8,3,4)
        (4,3,3)
        (7,2,5)
        (8,4,3)

B = GROUP A BY f1;
DUMP B;
        (1,{(1,2,3)})
        (4,{(4,2,1),(4,3,3)})
        (7,{(7,2,5)})
        (8,{(8,3,4),(8,4,3)})

X = FOREACH B GENERATE COUNT(A);
DUMP X;
        (1L)
        (2L)
        (1L)
```

Now Lets Write WordCount With Functions And Operators Learned So Far

```
file = LOAD '/data/dropbox-policy.txt' AS (line);

words = FOREACH file GENERATE FLATTEN(TOKENIZE(line)) AS word;

grouped  = GROUP words BY word;

counted  = FOREACH grouped GENERATE group, COUNT(words);

sorted_counted = ORDER counted BY $1;

DUMP sorted_counted;
```

Shell Commands

fs

Invokes any FsShell command from within a Pig script or the Grunt shell.

Examples
In these examples a directory is created, a file is copied, a file is listed.

```
fs -mkdir /tmp
fs -copyFromLocal file-x file-y
fs -ls file-y
```

Sh

Invokes any sh shell command from within a Pig script or the Grunt shell.

Example
In this example the ls command is invoked.

```
grunt> sh ls
bigdata.conf
nightly.conf
```

Utility Commands

Clear

Clear the screen of Pig grunt shell and position the cursor at top of the screen.

Example
In this example the clear command clean up Pig grunt shell.

```
grunt> clear
```

Exec

Run a Pig script.

Examples
In this example the script is displayed and run.

```
grunt> cat myscript.pig
a = LOAD 'student' AS (name, age, gpa);
b = LIMIT a 3;
DUMP B;

grunt> exec myscript.pig
        (alice,20,2.47)
        (luke,18,4.00)
        (holly,24,3.27)
```

History

Display the list of statements used so far.

History command shows all the statements with line numbers and without them.

```
grunt> a = LOAD 'student' AS (name, age, gpa);
grunt> b = order a by name;
grunt> history
        1 a = LOAD 'student' AS (name, age, gpa);
        2 b = order a by name;

grunt> c = order a by name;
grunt> history -n
        a = LOAD 'student' AS (name, age, gpa);
        b = order a by name;
        c = order a by name;
```

Kill

Kills a job. In this example the job with id job_0001 is killed.

```
grunt> kill job_0001
```

Quit

Quits from the Pig grunt shell. In this example the quit command exits the Pig grunt shell.

```
grunt> quit
```

Set

Assigns values to keys used in Pig.
In this example key value pairs are set at the command line.

```
grunt> SET debug 'on'
grunt> SET default_parallel 100
grunt> SET mapred.child.java.opts –Xmx2G
```

RUN

Run a Pig script.

In this example the script interacts with the results of commands issued via the Grunt shell.

```
grunt> cat myscript.pig
b = ORDER a BY name;
c = LIMIT b 10;
grunt> a = LOAD 'student' AS (name, age, gpa);
grunt> run myscript.pig
grunt> d = LIMIT c 3;
grunt> DUMP d;
        (alice,20,2.47)
        (alice,27,1.95)
        (alice,36,2.27)
```

Pig Script

Local Mode -

cat PigScript

```
A = LOAD '/home/user/Desktop/days.txt' USING PigStorage (',') as (name:chararray, age:int, sal:int);
B = FOREACH A GENERATE age;
dump B;
```

Run from Out of local mode

```
pig -x local -f /home/user/Desktop/pig_script
```

```
(22)
(23)
(24)
```

Mapred Mode -

move days .txt (file) to hdfs

cat PigScript

```
A = LOAD '/line' USING PigStorage (',') as (name:chararray, age:int, sal:int);
B = FOREACH A GENERATE age;
DUMP B;
```

run in mapred mode

```
Exec /home/user/Desktop/pig_script
```

```
(22)
(23)
(24)
```

Pig Debugging Commands

For Some Extent we can debug from below Commands

DUMP - Use the DUMP operator to run (execute) Pig Latin statements and display the results to your screen.

ILLUSTRATE - Use the ILLUSTRATE operator to review how data is transformed through a sequence of Pig Latin statements. ILLUSTRATE allows you to test your programs on small datasets and get faster turnaround times.

EXPLAIN - Use the EXPLAIN operator to review the logical, physical, and map reduce execution plans that are used to compute the specified relationship.

DESCRIBE - Use the DESCRIBE operator to view the schema of a relation. You can view outer relations as well as relations defined in a nested FOREACH statement.

SQL Vs PIG

- SELECT * FROM users;
- users = LOAD '/hdfs/users' USING PigStorage (‘\t’) AS (name:chararray, age:int, weight:int);
- SELECT * FROM users where weight < 150 ;
- skinnyUsers = FILTER users BY weight < 150;
- SELECT name, age FROM users where weight< 150;
- skinnyUserNames = FOREACH skinnyUsers GENERATE name, age;
- SELECT name, SUM(orderAmount) FROM orders GROUP BY name…
- A = GROUP orders BY name; B = FOREACH A GENERATE $0 AS name, SUM($1.orderAmount) AS orderTotal
- HAVING SUM(orderAmount) > 500...
- D = ORDER C BY name ASC;
- ORDER BY name ASC;
- names = FOREACH users GENERATE name;uniqueNames = DISTINCT names;
- SELECT DISTINCT name FROM users;
- usersByName = GROUP users BY name; numAgesByName = FOREACH usersByName { ages = DISTINCT users.age; GENERATE FLATTEN(group), COUNT(ages);}
- SELECT name, COUNT(DISTINCT age) FROM users GROUP BY name;

Load/Store HBase Tables In Pig

LOAD

```
test_staging = LOAD 'hbase://test_staging' USING
org.apache.pig.backend.hadoop.hbase.HBaseStorage('dim:dim_map
omg_counters:counter_map md:tt', '-loadKey true') AS (id: bytearray,dim:
chararray,omg: chararray,md: chararray);

omg_staging_typed = FOREACH test_staging GENERATE (chararray)id, dim, omg, md;
```

STORE

```
store test_staging_filter_remove_quote INTO '${PIGOUTPUT}' using PigStorage('|');
```

JSON LOADER

Load or store JSON data.

```
A = LOAD '/home/user/Desktop/days.txt' USING JsonLoader('food:chararray');
        Tacos
        Tomato Soup
        Grilled Cheese

STORE A INTO '/home/user/Desktop/ap.txt' USING JsonStorage();
```

ORC STORAGE

Loads from or stores data to Orc file

```
A = LOAD '/home/user/Desktop/days.txt' USING PigStorage(',') as (name:chararray, age:int);
STORE A INTO '/home/uset/Desktop/ap.txt' Using OrcStorage();
B = LOAD '/home/user/Desktop/ap.txt' USING OrcStorage();
        (roja,22)
        (rani,23)

describe B;
        B: {name: chararray,age: int}
```

String Function

INDEX OF

Returns the index of the first occurrence of a character in a string, searching forward from a start index.

```
A = LOAD '/home/user/Desktop/days.txt' as (username:chararray);
DUMP A;
        (test hahaha hohoho)

B = Foreach A Generate INDEXOF(username, 'hahaha', 0);
        (5,test hahaha hohoho)
```

LAST_INDEX OF

Returns the index of the last occurrence of a character in a string, searching backward from a start index.

```
A = LOAD '/home/user/Desktop/days.txt' as (username:chararray);
DUMP A;
        (test hahaha hohoho)

B = Foreach A Generate LASTINDEXOF(username, 'hahaha');
        (5,test hahaha hohoho)
```

LCFIRST

Converts the first character in a string to lower case.

```
A = LOAD '/home/user/Desktop/days.txt' as (username:chararray);
DUMP A;
        (Test hahaha hohoho)

B = Foreach A Generate LCFIRST(username);
        (test hahaha hohoho)
```

LOWER

Converts all characters in a string to lower case.

```
A = LOAD '/home/user/Desktop/days.txt' as (username:chararray);
DUMP A;
        (TEST HAHAHA HOHOHO)

B = Foreach A Generate LOWER(username);
        (test hahaha hohoho)
```

REGEX_EXTRACT

Performs regular expression matching and extracts the matched group defined by an index parameter.

```
A = LOAD '/home/user/Desktop/days.txt' as (username:chararray);
DUMP A;
        (FASTEST CASHTEST LATEST)

B = Foreach A Generate REGEX_EXTRACT(username, (*TEST), 0);
        (FASTEST CASHTEST LATEST)
```

REPLACE

Replaces existing characters in a string with new characters.

```
A = LOAD '/home/user/Desktop/days.txt' as (username:chararray);
DUMP A;
        (FASTEST THEBEST LATEST)

B = Foreach A Generate REPLACE(username, 'FASTEST', 'LATEST');
        (LATEST THEBEST LATEST)
```

STRSPLIT

Splits a string around matches of a given regular expression.

```
A = LOAD '/home/user/Desktop/days.txt' USING PigStorage(',')
AS(a1:chararray,a2:chararray,a3:chararray);
DUMP A;
        aa.kyl,data,data
        bbb.kkk,data,data
        cccccc.hj,data,data
        qa.dff,data,data

B = FOREACH A GENERATE FLATTEN(STRSPLIT(a1,'\\u002E')) as (a1:chararray,
a1of1:chararray),a2,a3;
        (aa,kyl,data,data)
        (bbb,kkk,data,data)
        (cccccc,hj,data,data)
        (qa,dff,data,data)

C = FOREACH B GENERATE a1of1,a2,a3;
        (kyl,data,data)
        (kkk,data,data)
        (hj,data,data)
        (dff,data,data)
```

SUBSTRING

Returns a substring from a given string.

```
A = LOAD '/home/user/Desktop/days.txt' as (username:chararray);
dump A;
        substring

B = GROUP A By SUBSTRING(username, 0, b);
DUMP B;
        (sub,{(substring)})
```

TRIM

Returns a copy of a string with leading and trailing white space removed.

```
A = LOAD '/home/user/Desktop/days.txt' as (username:chararray);
dump A;
                (s u b s t ring)

B = foreach A GENERATE TRIM(username);
DUMP B;
                substring
```

UCFIRST

Returns a string with the first character converted to upper case.

```
A = LOAD '/home/user/Desktop/days.txt' as (username:chararray);
dump A;
        (s u b s t ring)

B = foreach A GENERATE UCFIRST(username);
        (S u b s t ring)
```

UPPER

Returns a string converted to upper case.

```
A = LOAD '/home/user/Desktop/days.txt' as (username:chararray);
dump A;
        (s u b s t ring)

B = foreach A GENERATE UPPER(username);
        (S U B S T RING)
```

UDF – User Defined Function

User Defined Functions

- Do operations on more than one field
- Do more than grouping and filtering
- Programmer is comfortable

- Traditionally UDF can be written only in Java. Now other languages like Python are also supported

- Types of Udfs:
  - ✓ Eval Functions
  - ✓ Filter functions

Eval Function

- Can be used in FOREACH statement
- Most common type of UDF
- Can return simple types or Tuples

  ✓b = FOREACH a generate udf($0);

  ✓b = FOREACH a generate udf($0, $1);

- Extend EvalFunc<T> interface
- Input comes as a Tuple
- Should check for empty and nulls in input
- Extend exec() function and it should return the value
- Extend getArgToFuncMapping() to let UDF know about Argument mapping
- Extend outputSchema() to let UDF know about output schema

User Defined Functions

Allow you to perform more complex operations upon fields Written in java, compiled into a jar, loaded into your Pig script at runtime

```
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;
public class UPPER extends EvalFunc<String> {
        public String exec(Tuple input) throws IOException {
                if (input == null || input.size() == 0)
                return null;
                try{
                String str = (String)input.get(0);
                return str.toUpperCase();
                }
                catch(Exception e){
                                e.printStackTrace();
                                return null;
                }
        }
}
```

Using JAVA UDF In Pig Script

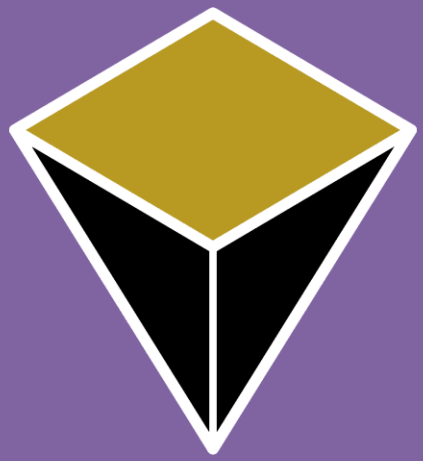
- Create a jar file which contains your UDF classes
- Register the jar at the top of Pig script
- Register dependent third party jars as well if needed
- Define the UDF function
- Use your UDF function

```
REGISTER udf/dist/udf.jar;
REGISTER udf/libs/commons-lang3-3.1.jar;

DEFINE StripQuote com.trianz.pig.udf.StripQuote();

data = LOAD 'data/tweets.csv' USING PigStorage(',');

tweets = FOREACH data GENERATE StripQuote($7);

DUMP tweets;
```

Filter Function

- Extends FilterFunc, which is a EvalFunc<Boolean>
- Should return a boolean
- Input is same as EvalFunc<T>
- Should check for empty and nulls in input
- Extend getArgToFuncMapping() to let UDF know about Argument mapping
- Can be used in the Filter statements
- Returns a boolean value

Eg: vim_tweets = FILTER data By FromVim(StripQuote($6));



PIG OPTIMIZATION

Reduce Your Operator Pipeline

For clarity of your script, you might choose to split your projects into several steps for instance:

```
A = load '/home/user/Desktop/map.txt' as (in: map[]);
B = foreach A generate in#'k1' as k1, in#'k2' as k2;
C = foreach B generate CONCAT(k1, k2)
```

Query 2

Combining the two foreach statements will improve query performance:

```
A = load '/home/user/Desktop/map.txt' as (in: map[]);
B = foreach A generate CONCAT(in#'k1', in#'k2');
```

USE TYPES

If types are not specified in the load statement, Pig assumes the type of =double= for numeric computations. A lot of the time, your data would be much smaller, maybe, integer or long. Specifying the real type will help with speed of arithmetic computation. It has an additional advantage of early error detection.

Query 1

```
A = LOAD '/home/user/Desktop/data.txt' USING PigStorage (',') as (t, u, v);
B = foreach A generate t + u;
DUMP B;
```

```
(5.0)
(7.0)
(4.0)
(3.0)
```

Query 2

```
A = LOAD '/home/user/Desktop/data.txt' USING PigStorage (',') as (t:int, u:int, v:int);
B = foreach A generate t + u;
DUMP B;
```

```
(5)
(7)
(4)
(3)
```

The second query will run more efficiently than the first.

PROJECT EARLY AND OFTEN

Query 1

```
A = LOAD '/home/user/Desktop/data.txt' USING PigStorage (',') as (t:int, u:int, v:int);
B = LOAD '/home/user/Desktop/content.txt' USING PigStorage (',') as (x, y, z);
C = join A by t, B by x;
D = group C by u;
E = foreach D generate group, COUNT($1);
```

(1,1)
(3,2)

Changing the query above to the query below will greatly reduce the amount of data being carried through the map and reduce phases by pig.

Query 2

```
A = LOAD '/home/user/Desktop/data.txt' USING PigStorage (',') as (t:int, u:int, v:int);
A1 = foreach A generate t, u;
B = LOAD '/home/user/Desktop/content.txt' USING PigStorage (',') as (x, y, z);
B1 = foreach B generate x;
C = join A1 by t, B1 by x;
C1 = foreach C generate t, u;
D = group C1 by u;
E = foreach D generate group, COUNT($1);
```

(1,1)
(3,2)

FILTER EARLY AND OFTEN

As with early projection, in most cases it is beneficial to apply filters as early as possible to reduce the amount of data flowing through the pipeline.

Query 1

```
A = LOAD '/home/user/Desktop/data.txt' USING PigStorage (',') as (t:int, u:int, v:int);
B = LOAD '/home/user/Desktop/content.txt' USING PigStorage (',') as (x, y, z);
C = join A by t, B by x;
D = group C by u;
E = foreach D generate group, COUNT($1);
F = filter E by C.t == 1;
```

(3,1)

Query 2

```
A = LOAD '/home/user/Desktop/data.txt' USING PigStorage (',') as (t:int, u:int, v:int);
B = LOAD '/home/user/Desktop/content.txt' USING PigStorage (',') as (x, y, z);
C = filter A by t == 1;
D = join C by t, B by x;
E = group D by u;
F = foreach E generate group, COUNT($1);
```

The Second query is clearly more efficient than the First one because it reduces the amount of data going into the join.

DROPPING NULLS BEFORE JOIN

Dropping nulls before Join tends to less transfer of data , transfering data will be reduced

Query 1

```
A = LOAD '/home/user/Desktop/data.txt' USING PigStorage (',') as (t:int, u:int, v:int);
B = LOAD '/home/user/Desktop/content.txt' USING PigStorage (',') as (x, y, z);
C = join A by t, B by x;
```

Query 2

```
A = LOAD '/home/user/Desktop/data.txt' USING PigStorage (',') as (t:int, u:int, v:int);
B = LOAD '/home/user/Desktop/content.txt' USING PigStorage (',') as (x, y, z);
A1 = filter A by t is not null;
B1 = filter B by x is not null;
C = join A1 by t, B1 by x;
```

TAKE ADVANTAGE OF JOIN OPTIMIZATION

Regular Join Optimizations

Optimization for regular joins ensures that the last table in the join is not brought into memory but streamed through instead. Optimization reduces the amount of memory used which means you can avoid spilling the data and also should be able to scale your query to larger data volumes.

To take advantage of this optimization, make sure that the larger table is the last table in your query. In some of our tests we saw 10x performance improvement as the result of this optimization.

```
A = LOAD '/home/user/Desktop/data.txt' USING PigStorage (',') as (t:int, u:int, v:int);
B = LOAD '/home/user/Desktop/content.txt' USING PigStorage (',') as (x, y, z);
C = join A by t, B by x;
```

USE THE LIMIT OPERATOR

Often you are not interested in the entire output but rather a sample or top results. In such cases, using LIMIT can yield a much better performance as we push the limit as high as possible to minimize the amount of data travelling through the pipeline.

Sample:

```
A = LOAD '/home/user/Desktop/data.txt' USING PigStorage (',') as (t:int, u:int, v:int);
B = limit A 500;
```

Top results:

```
A = LOAD '/home/user/Desktop/data.txt' USING PigStorage (',') as (t:int, u:int, v:int);
B = order A by t;
C = limit B 500;
```

PREFER DISTINCT OVER GROUP BY/GENERATE

To extract unique values from a column in a relation you can use DISTINCT or GROUP BY/GENERATE. DISTINCT is the preferred method; it is faster and more efficient.

Example using GROUP BY - GENERATE:

```
A = LOAD '/home/user/Desktop/data.txt' USING PigStorage (',') as (t:int, u:int, v:int);
B = foreach A generate u;
C = group B by u;
D = foreach C generate group as uniquekey;
dump D;
```

Example using DISTINCT:

```
A = LOAD '/home/user/Desktop/data.txt' USING PigStorage (',') as (t:int, u:int, v:int);
B = foreach A generate u;
C = distinct B;
dump C;
```

USING SPECIFIED JOINS

Replicated Joins

Perform a replicated join with the USING clause (see JOIN (inner) and JOIN (outer)). In this example, a large relation is joined with two smaller relations. Note that the large relation comes first followed by the smaller relations; and, all small relations together must fit into main memory, otherwise an error is generated.

```
big = LOAD 'big_data' AS (b1,b2,b3);

tiny = LOAD 'tiny_data' AS (t1,t2,t3);

mini = LOAD 'mini_data' AS (m1,m2,m3);

C = JOIN big BY b1, tiny BY t1, mini BY m1 USING 'replicated' ;
```

SKEWED JOINS

Parallel joins are vulnerable to the presence of skew in the underlying data. If the underlying data is sufficiently skewed, load imbalances will swamp any of the parallelism gains. In order to counteract this problem, skewed join computes a histogram of the key space and uses this data to allocate reducers for a given key. Skewed join does not place a restriction on the size of the input keys. It accomplishes this by splitting the left input on the join predicate and streaming the right input. The left input is sampled to create the histogram.

Skewed join can be used when the underlying data is sufficiently skewed and you need a finer control over the allocation of reducers to counteract the skew. It should also be used when the data associated with a given key is too large to fit in memory.
Usage

Perform a skewed join with the USING clause (see JOIN (inner) and JOIN (outer)).

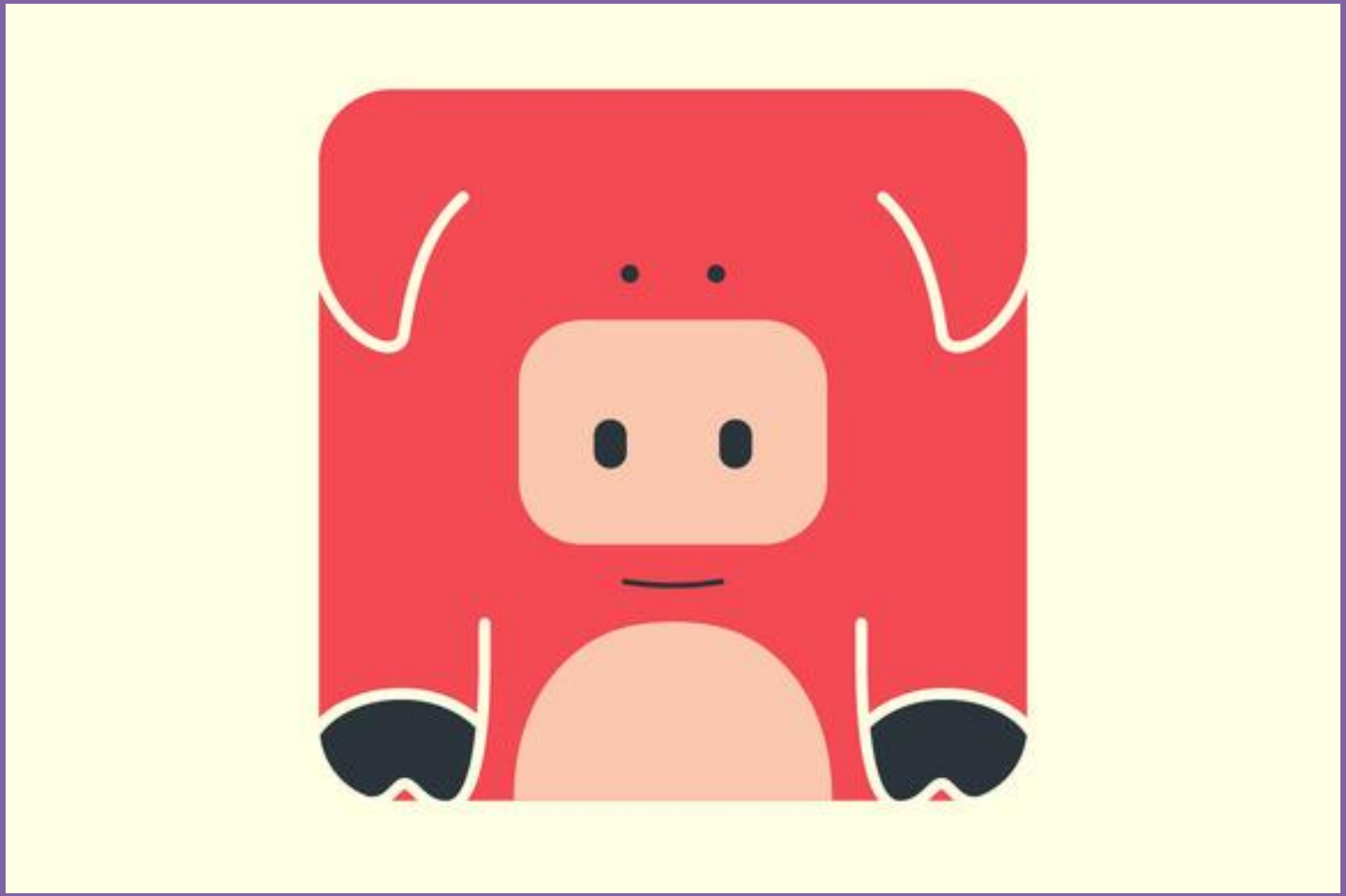
```
big = LOAD 'big_data' AS (b1,b2,b3);
massive = LOAD 'massive_data' AS (m1,m2,m3);
C = JOIN big BY b1, massive BY m1 USING 'skewed'
```

MERGE JOINS

Often user data is stored such that both inputs are already sorted on the join key. In this case, it is possible to join the data in the map phase of a MapReduce job. This provides a significant performance improvement compared to passing all of the data through unneeded sort and shuffle phases. Pig has implemented a merge join algorithm, or sort-merge join. It works on pre-sorted data, and does not sort data for you

```
C = JOIN A BY a1, B BY b1, C BY c1 USING 'merge';
```

ANY QUESTIONS?



Thank You
&
Visit This Presentation Again