

Pig Latin

Zoran B. Djordjević
csci e185 Big Data Analytics

@Zoran B. Djordjević

1

Pig

- Map Reduce is relatively difficult to program.
- Data processing is usually accomplished in terms of data flow operations, such as loops and filters.
- In Map Reduce, you think at the level of mapper and reducer functions and job chaining.
- Certain functions that are treated as first-class operations in higher-level languages are nontrivial to implement in Map Reduce.
- **Pig** is a Hadoop extension that simplifies Hadoop programming by giving you a high-level data processing and data flow language while preserving Hadoop's scalability and reliability.
- Yahoo! Research developed Pig to address the need for a higher level language that replaces Map Reduce programming.
- Yahoo runs 40 percent of all its Hadoop jobs with Pig.

@Zoran B. Djordjević

2

Pig Latin

- Pig has two major components:
 1. A **high-level data processing language** called **Pig Latin**.
 2. A **compiler** that compiles and runs your Pig Latin script on Hadoop.
- Pig works on Hadoop clusters, but also supports a local mode for development purposes.
- Pig simplifies programming because of the ease of expressing your code in Pig Latin.
- The compiler helps to automatically exploit optimization opportunities in your script.
- As the Pig compiler improves, your Pig Latin program will also get an automatic speed-up.
- Crucial to efficient use of Pig are the design choices of its programming language (Pig Latin), the data types it supports, and its treatment of user-defined functions (UDFs) as first-class citizens.

@Zoran B. Djordjević

3

Pig & Pig Latin

- Motivation
 - Map Reduce is very powerful, but:
 - It requires a Java programmer.
 - User has to re-invent common functionality (join, filter, etc.)
- Pig Latin is a higher level language, that:
 - Increases productivity.
 - In one test 10 lines of Pig Latin \approx 200 lines of Java.
 - What took 4 hours to write in Java took 15 minutes in Pig Latin.
 - Opens the system to non-Java programmers.
 - Provides common relational-like operations like:
 - join,
 - group,
 - filter,
 - sort.

@Zoran B. Djordjević

4

Data Flow Language

- Pig Latin programs are written in a sequence of steps where each step is a single high-level data transformation.
- The transformations support relational-style operations, such as filter, union, group, and join.
- A Pig Latin program processing a search query log may look like


```
log = LOAD 'excite-small.log' AS (user, time, query);
grpd = GROUP log BY user;
cntd = FOREACH grpd GENERATE group, COUNT(log);
DUMP cntd;
```
- Operations are relational in style, however, Pig Latin is a data flow language. A data flow language is friendlier to programmers who think in terms of algorithms, which are more naturally expressed by the data and control flows.
- A declarative language such as SQL is easier for analysts who prefer to just state the results one expects from a program.
- Hive is a different Hadoop project which is closer to the SQL model.

@Zoran B. Djordjević

5

Installing Pig

- We can download the latest release of Pig from <http://pig.apache.org/releases.html>.
- Pig requires Java 1.6 or later. We need to point JAVA_HOME to the root of our Java installation. Windows users should install Cygwin.
- Your Hadoop cluster should already be set up before installing Pig. Both a real cluster in fully distributed mode, and a pseudo-distributed setup is fine for practice.
- You install Pig on your local machine by unpacking the downloaded distribution. There's nothing to modify on your Hadoop installation.
- Think of the Pig distribution as a compiler and some development and deployment tools.
- Pig enhances MapReduce programming but is otherwise only loosely coupled with the production Hadoop cluster.
- If in hurry we could run Pig in the Cloud using services of AWS Elastic Map Reduce service

@Zoran B. Djordjević

6

Download and Setup on CDH4.2 VM

- For any Pig release you can go to
 - <http://pig.hadoop.org/releases.html>
- Download and untar archives and copy them to a customary place. Like:


```
$ tar xzf pig-0.5.0.tar.gz
$ sudo mv pig-0.5.0.tar.gz /usr/local
```
- To match pig with the installation of CDH4.2 we have on our VM-s we better download the Cloudera's Pig tarball:


```
pig-0.10.0-cdh4.2.0.tar.gz
```
- Just like the above, we will untar the tarball and move the resulting directory to /usr/local. Subsequently we set JAVA_HOME, PIG_INSTALL and PIG_CLASSPATH environmental variables.
- PIG_CLASSPATH points to the directory where Hadoop's file `hadoop-site.xml` resides. On our installation of CDH4.2 that directory appears to be `/etc/hadoop/conf.empty`
- We add all of those variables to our `.bash_profile` file.

@Zoran B. Djordjević

7

.bash_profile file

- The following is the relevant content of the `.bash_profile` file:

```
JAVA_HOME=/usr/local/java/jdk1.6.0_31
export JAVA_HOME
PIG_INSTALL=/usr/local/pig-0.10.0-cdh4.2.0
export PIG_INSTALL
PIG_CLASSPATH=/etc/hadoop/conf.empty
export PIG_CLASSPATH
PATH=$JAVA_HOME/bin:$PIG_INSTALL/bin:$PATH
export PATH
```

- To make new variables visible to your user, type:

```
$ source .bash_profile
$ echo $PIG_INSTALL
/usr/local/pig-0.10.0-cdh4.2.0
$ which pig
/usr/local/pig-0.10.0-cdh4.2.0/bin/pig
```

NOTE: In this CDH4.2 distribution, an important jar file, `piggybank.jar`, resides in the directory:

```
/usr/local/pig-0.10.0-cdh4.2.0/contrib/piggybank/java
```

@Zoran B. Djordjević

8

Pig and Elastic MapReduce

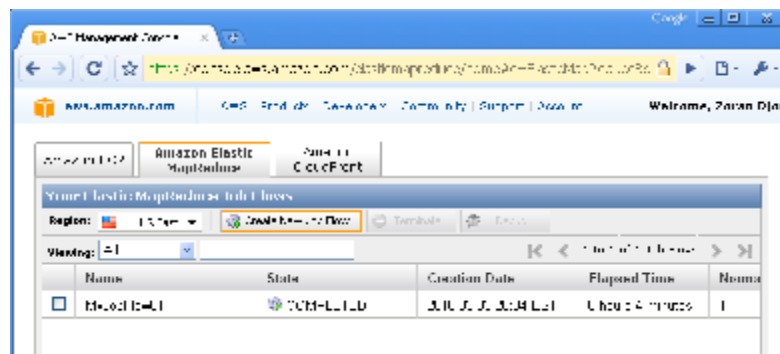
- Amazon Elastic MapReduce is a web service which provides you with the infrastructure on which you could run Pig programs.
- Pig compiler generates a series of map and reduce routines that run on a Hadoop cluster for efficient processing of large data sets.
- EMR utilizes a hosted Hadoop framework running on the web-scale infrastructure of Amazon Elastic Compute Cloud (Amazon EC2) and Amazon Simple Storage Service (Amazon S3).
- Using Amazon Elastic Map Reduce, you can instantly provision Hadoop clusters of arbitrary size in order to perform data-intensive tasks for applications such as web indexing, data mining, log file analysis, machine learning, financial analysis, scientific simulation, and bioinformatics research.
- Amazon Elastic Map Reduce automatically sub-divides the data in a job flow into smaller chunks so that they can be processed by map functions in parallel, and eventually recombined into the final solution (the “reduce” function).
- Amazon S3 serves as the source for the data being analyzed, and as the output destination for the end results.

@Zoran B. Djordjević

9

Create a new Job Flow

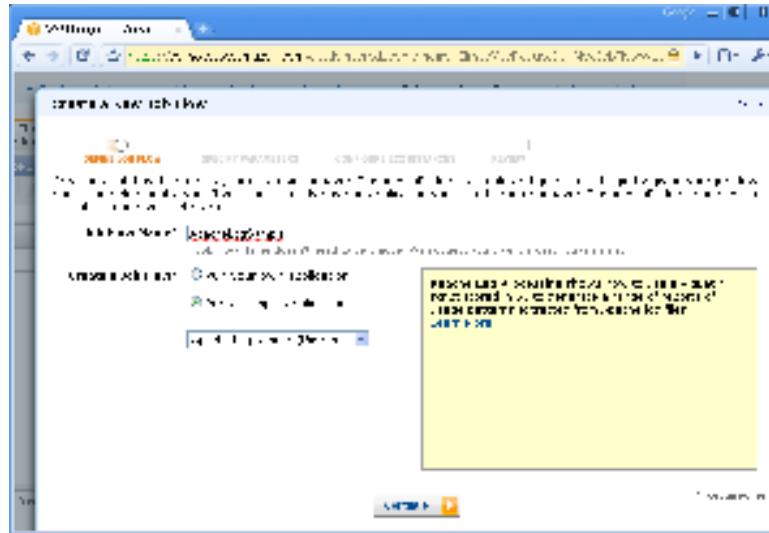
- Login into AWS Management Console
- Select Elastic MapReduce
- Click on Create New Job Flow



@Zoran B. Djordjević

10

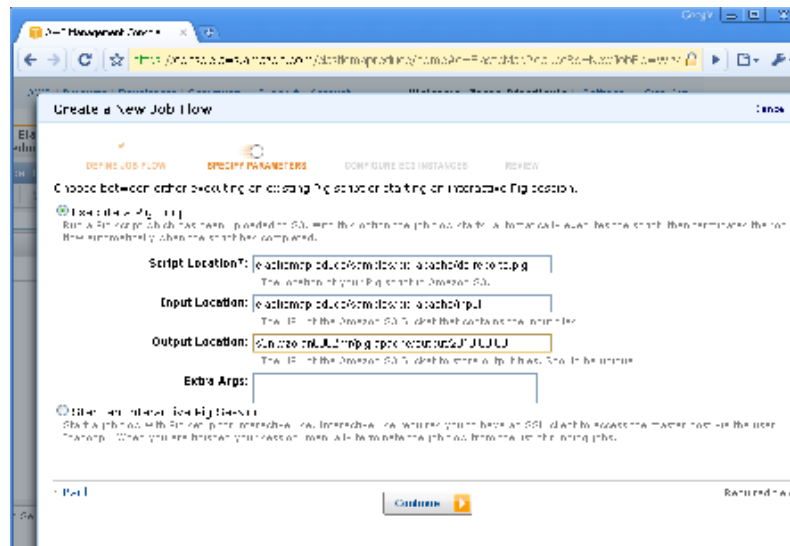
Select Sample Application



@Zoran B. Djordjević

11

Select your S3 bucket for output



@Zoran B. Djordjević

12

Select your Key and Number of Instances

Create a New Job Flow

Specify the number and type of EC2 instances you'd like to run your job flow on.

Number of Instances: 4

Type of Instance: emr-ml-m3.x1

Amazon EC2 Key Pair: ml-j-ml-1

Configure additional settings (learn more)

Enable Debugging: Yes

Amazon S3 Log Path: s3://aws-logs-9-us-east-1-123456789012-us-east-1

Enable Hadoop Debugging: Yes

Continue

@Zoran B. Djordjević

13

Job will run for a few minutes

Your Elastic MapReduce Job Flows

Name	State	Creation Date	Elapsed Time	Normalized Instance Hrs
elasticmapreduce	STARTING	2013-03-28 20:14:11	0:00:00.000000	0
elasticmapreduce	STARTING	2013-03-28 20:14:11	0:00:00.000000	0

Job Flow selected

ID:	j-EP3CCHP7J007	Creation Date:	2013-03-28 20:14:11
Name:	elasticmapreduce	Start Date:	-
State:	STARTING	End Date:	-
Last State Change Reason:	Job Flow is starting.		
Availability Zone:	us-east-1b	Instance Count:	4

@Zoran B. Djordjević

14

Job is Completed

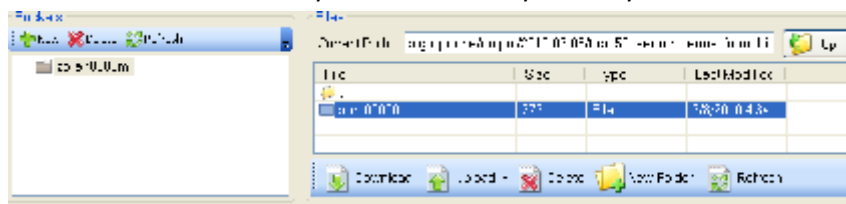
[illegible]

@Zoran B. Djordjević

15

Go to your Bucket, Download Result

- Click a few levels into your bucket until you find your results



value	625
views	426
login	224
search	195
items	112
bigtable	68
google+bigtable	59
%23%21%2Fusr%2Fbin%2Fperl+-w	46
philmont+pictures	45
%23%21%2Fusr%2Fbin%2Fperl	44
philmont	37
google+quick+links	33
pvc+instrument	33

@Zoran B. Djordjević

16

Example of a Pig Script: do-report.pig

```
-- setup piggyback functions
register file:/home/hadoop/lib/pig/piggybank.jar
DEFINE EXTRACT org.apache.pig.piggybank.evaluation.string.EXTRACT();
DEFINE FORMAT org.apache.pig.piggybank.evaluation.string.FORMAT(); . . . .
-- import logs and break into tuples
raw_logs =
  -- load the weblogs into a sequence of one element tuples
  LOAD '$INPUT' USING TextLoader AS (line:chararray);
logs_base =
  -- for each weblog string convert the weblog string into a
  -- structure with named fields
  FOREACH
    raw_logs
  GENERATE FLATTEN ( EXTRACT(
    line,
    '^(\S+) (\S+) (\S+) \[([\\w:/]+\s[+-]\d{4})\]' "(.*)" "(\S+) (\S+)"
    "([\"']*)" "([\"']*)"
  )
  AS (
    remoteAddr: chararray, remoteLogname: chararray, user: chararray, time:
    chararray,
    request: chararray, status: int, bytes_string: chararray, referrer: chararray,
    browser: chararray
  ) ;
referrer matches '.*google.*' . . . .
```

@Zoran B. Djordjević

17

How could we Access Pig

- Submit a script directly.
- Interactively, through Grunt, the pig shell.
- Through PigServer, a Java class, with a JDBC like interface, that allows Java programs to execute Pig queries.
- Pig could only be run on your Client machine.
- No need to install anything extra on your Hadoop cluster.
- Jobs Pig initiates will run on the Hadoop clusters.
- Pig needs to be installed separately.
- In order to run local tests you need both local Hadoop and Pig installations.
- In the development environment, Pig could be run in a local mode which does not use Hadoop at all.

@Zoran B. Djordjević

18

How Pig Works

- You write a script:

```
-- max_temp.pig: Finds the maximum temperature by year
records = LOAD 'input/ncdc/micro-tab/sample.txt'
        AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
        (quality == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality ==
        9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
        MAX(filtered_records.temperature);
DUMP max_temp;
```

- Pig parses, checks, optimizes, creates execution plan, and submits a `pig.jar` to Hadoop
- Execution plan contains Map and Reduce routines. Sometimes many of them.
- Pig monitors job progress and reports on errors and results.
- Is Pig slowing you down? Pig adds 20-40% overhead.

@Zoran B. Djordjević

19

Data Types

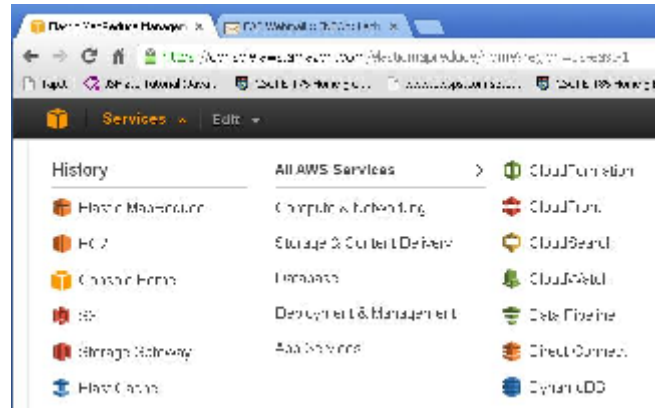
- Basic data types of Pig Latin are:
 - Java like scalar type:
 - int,
 - long,
 - double,
 - chararray,
 - bytearray.
 - Complex types:
 - map: associative array (Hash table).
 - tuple: ordered list of data, elements maybe of any scalar or complex type.
 - bag: unordered collection of tuples
- Pig's philosophy toward data types is summarized in its slogan of "Pigs eat anything." Input data can come in any format. Popular formats, such as tab-delimited text files, are natively supported.
- Pig is architected from the ground up with support for user-defined functions.

@Zoran B. Djordjević

20

Getting Pig Development Environment

- One way of establishing Pig Development environment is to start an Elastic MapReduce Job Flow and select an Interactive Pig session.
- From the list of AWS Services, select Elastic MapReduce.



- On the following screen select "Create New Job Flow"

@Zoran B. Djordjević

21

Name the Job, select Hadoop Version

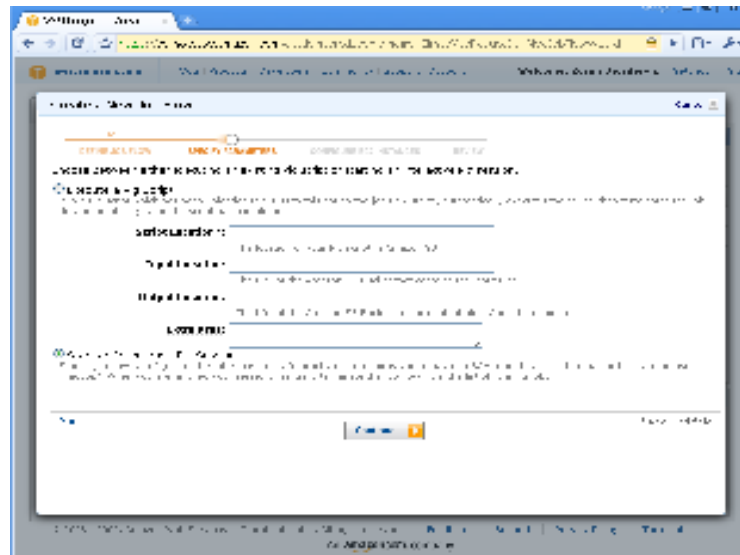
- On the "Create New Job Flow" screen, please name your job, and more importantly select
- Hadoop Version 0.20.205 (MapR M3 Edition v1.2.8).
- Pig will run on other versions of Hadoop. There are just some issues with setup on those other releases.
- Also select "Run your own application and" and "Pig Program"



@Zoran B. Djordjević

22

Select "Start an Interactive Pig Session"

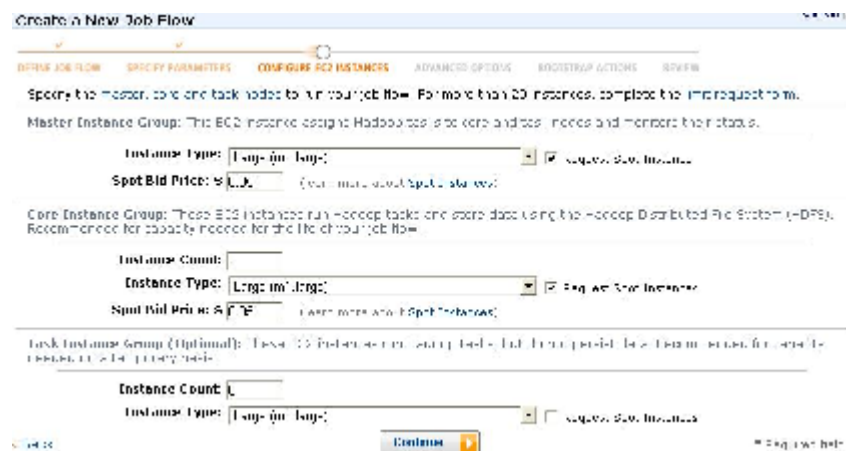


@Zoran B. Djordjević

23

Configure EC2 Instances

- For development and testing select the smallest available instances and the minimal count.



@Zoran B. Djordjević

24

Provide EC2 Key Pair & Log Path

- For development always Enable Debugging.
- After Continue, Proceed with no Bootstrap Action and Create Job Flow

Create a New Job Flow

DEFINE JOB FLOW | SPECIFY PARAMETERS | CONFIGURE EC2 INSTANCES | **ADVANCED OPTIONS** | BOOTSTRAP ACTIONS | REVIEW

Here you can choose which cloud provider you want to use to create the job flow. By default, Amazon EC2 is selected.

Amazon EC2 Key Pair:

Amazon VPC Subnet ID:

Configure your logging options. Learn more.

Amazon S3 Log Path:

Enable Debugging: ☒ Yes ☐ No

Set advanced job flow options.

Keep Alive: ☒ Yes ☐ No

Termination Protection: ☐ Yes ☒ No

Visible To All IAM Users: ☐ Yes ☒ No

[View Job Flow Details](#) [Continue](#) [Request Help](#)

@Zoran B. Djordjević

25

Create Job Flow

Create a New Job Flow

DEFINE JOB FLOW | SPECIFY PARAMETERS | CONFIGURE EC2 INSTANCES | **ADVANCED OPTIONS** | BOOTSTRAP ACTIONS | REVIEW

Job Flow Name: [Edit Job Flow Definition](#)

Parameters: [Edit Parameters](#)

Master Instance Type: Instance Count: Spot Bid Price: Instance Count: Spot Bid Price:

Amazon Subnet ID:

Amazon S3 Log Path:

Enable Debugging: ☒ Yes ☐ No

Termination Protected: ☐ Yes ☒ No

Keep Alive: ☒ Yes ☐ No

Visible To All Users: ☐ Yes ☒ No

Bootstrap Actions: [Edit Advanced Options](#)

[Back](#) [Create Job Flow](#) [View Job Flow Details](#)

Note: Once you click Create Job Flow, the job flow will be created and you will be taken to the Job Flow Details page.

@Zoran B. Djordjević

26

This job must be manually terminated



- In difference from the automatic jobs, these must be be terminated by you.
- To kill the job, go to the job console, select the job and hit Terminate button.
- You can also terminate EC2 instances making the job.

Your Elastic MapReduce Job Flows

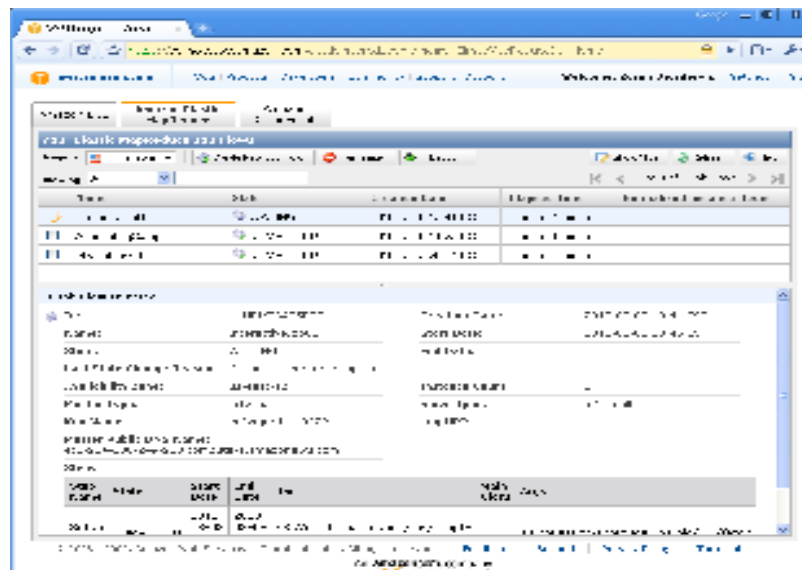
Viewing: **Waiting**

Name	State	Creation Date	Elapsed Time	Normalized Instance Hours
My interactive job flow	WAITING	2013-03-29 12:00	1 hours 1 minutes	1

@Zoran B. Djordjević

27

Wait until job status changes to Waiting

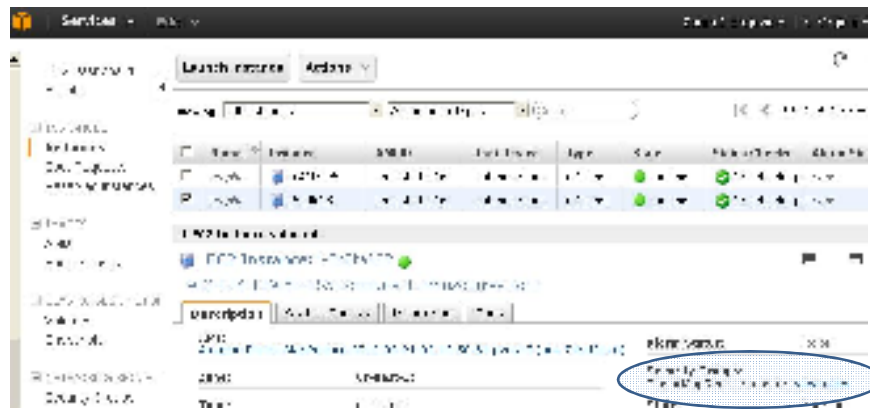


@Zoran B. Djordjević

28

Go to EC2 Console and Identify Master Node

- We have only 2 instances, one of them is the Master, the other a slave. The Security Group on the master contains noun master in its name. That security group betrays the master.
- Once the master is identified, right click on it and select Connect. On the Connect wizard select the Public DNS of the master.



@Zoran B. Djordjević

29

Connection Wizard

- We use the Connection wizard to select the public DNS of the Master.

Connect to an Instance Cancel

Instance: ec2-1

☐ Connect with a standard SSH Client

☒ Connect from your browser using the Java SSH Client (Java Required)

Enter the required information, which will be used to connect to your instance. The wizard will automatically select the public IP address of your instance. You need to enter a username and a password or a private key file to connect to the instance.

Public DNS: **ec2-17-49-51-88.us-east-1.amazonaws.com**

Username: **root**

Key name: **ec2_key**

Private key path: **/home/jeffrey/.ssh/private-key/ec2_key.pem**

Save key location: ☐ Store in browser wallet

Launch SSH Client

@Zoran B. Djordjević

30

Fetch Public DNS (Master), `ssh` to the Master

- Use user name “hadoop”, public DNS of the Master, and our Private Key to login into the Master of our cluster (even if we have just one machine in that cluster ☺).

- Go to your Cygwin prompt, `cd` to your `ec2` directory and type:

```
$ ssh -i ec2_hu.pem hadoop@ec2-174-129-51-185.compute-1.amazonaws.com
Welcome to Amazon Elastic MapReduce running Hadoop and Debian/Squeeze.
Hadoop is installed in /home/hadoop. Log files are in
    /mnt/var/log/hadoop. Check
    /mnt/var/log/hadoop/steps for diagnosing step failures.
The Hadoop UI can be accessed via the following commands:
```

```
JobTracker    lynx http://localhost:9100/
NameNode      lynx http://localhost:9101/
hadoop@ip-10-125-14-184:~$
```

- You are on the Hadoop Master.
- Our objective is to enter Pig command shell `grunt` and develop Pig scripts.
- Most of that work could be done in the “local” mode. Once we have ready scripts and a lot of data we could switch to the `mapreduce` mode.

@Zoran B. Djordjević

31

Open `grunt` session in the local mode

- Hadoop is very good in dealing with distributed processes and big data.
- For now we want to “prototype” something and the local mode is perfectly appropriate. Type

```
$ pig -x local      << this brings us into grunt session
grunt> help         << lists all grunt commands
Commands:
<pig latin statement>; - See the PigLatin manual for details:
    http://hadoop.apache.org/pig
File system commands:
    fs <fs arguments> - Equivalent to Hadoop dfs command:
    http://hadoop.apache.org/common/docs/current/hdfs_sell.html
. . . . .
quit - Quit the grunt shell.
grunt>
```

- The number of commands is not enormous. Still, you need to print the output of the `grunt> help` command and study it.

@Zoran B. Djordjević

32

Pig and Hadoop support 3 File Systems

- Pig and Hadoop support three file systems: you Linux file system, denoted by `file:/`, MapReduce (hdfs) file system, denoted by `maprfs:/` and S3 bucket file system denoted by `s3n:/`.
- You navigate between those file systems using `cd` command and you inquire in which file system you currently reside by typing `pwd` command.

```
grunt> pwd          // pwd will tell you in which FS you are in now
maprfs:/            // we are at the root of maprfs (HDFS) file system
grunt> ls
maprfs:/cluster-info <dir> // These are directories in HDFS file system
maprfs:/hbase <dir>
maprfs:/var <dir>
grunt>
grunt> cd file:/// // We are switching to the Linux file system
grunt> pwd
file:/              // We are at the root of the Linux file system
grunt> ls
file:/bin <dir> // These are some directories at the root of Linux Fs
file:/home <dir>
file:/mnt <dir>
. . . .
grunt> cd s3n://elasticmapreduce // Will take us to an s3 bucket
Grunt> pwd
Grunt> s3n://elasticmapreduce
```

@Zoran B. Djordjević

33

Move data from s3n://elasticmapreduce/samples to local FS

- We have sample data (Apache access logs) in the bucket:

```
grunt> ls s3n://elasticmapreduce/samples/pig-apache/input
s3n://elasticmapreduce/samples/pig-apache/input/access_log_1<r 1> 8754118
s3n://elasticmapreduce/samples/pig-apache/input/access_log_2<r 1> 8902171
s3n://elasticmapreduce/samples/pig-apache/input/access_log_3<r 1> 8896201
grunt>
```

- However, we do not want to go “over there” for every bit of data.
- Pig can copy data from one file system to another.

```
grunt> cp s3n://elasticmapreduce/samples/pig-apache/input/access_log_1 file:///home/hadoop
2013-03-30 04:12:54,424 [main] INFO org.apache.hadoop.fs.s3native.NativeS3FileSystem -
Opening 's3n://elasticmapreduce/samples/pig-apache/input/access_log_1' for reading
2013-03-30 04:12:54,790 [main] INFO org.apache.hadoop.util.NativeCodeLoader - Loaded the
native-hadoop library
grunt> cd file:///home/hadoop
Grunt> ls
file:/home/hadoop/bin <dir>
file:/home/hadoop/.pig_history<r 1> 476
file:/home/hadoop/PATCHES.txt<r 1> 0
file:/home/hadoop/lib <dir>
file:/home/hadoop/contrib <dir>
file:/home/hadoop/access_log_1<r 1> 8754118
file:/home/hadoop/hadoop-0.18-test.jar<r 1> 1014156
file:/home/hadoop/.ssh <dir>
```

@Zoran B. Djordjević

34

hadoop fs commands on grunt> prompt

- When in `maprfs:/` (HDFS) file system, you can use standard `hadoop fs` commands and they will act on the HDFS as if you are typing them on the Linux prompt. You do not type `hadoop`, though.
- For example:

```
grunt> pwd
maprfs:/
grunt> fs -ls /
Found 3 items
drwxr-xr-x - hadoop hadoop      1 2013-03-30 03:29 /cluster-info
drwxrwxrwx - root root          0 2013-03-30 03:29 /hbase
drwxr-xr-x - root root          1 2013-03-30 03:29 /var
grunt> fs -mkdir users
grunt> fs -ls
Found 4 items
drwxr-xr-x - hadoop hadoop      1 2013-03-30 03:29 /cluster-info
drwxrwxrwx - root root          0 2013-03-30 03:29 /hbase
drwxr-xr-x - hadoop hadoop      0 2013-03-30 04:54 /users
drwxr-xr-x - root root          1 2013-03-30 03:29 /var
```

@Zoran B. Djordjević

35

Linux shell commands on grunt> prompt

- In order to run standard Linux shell commands from the `grunt>` shell prompt, just prefix them with `"sh"`.
- The commands will produce results as if you were on the Linux command prompt. For example:

```
grunt> grunt> sh ls
PATCHES.txt
access_log_1
bin
grunt> sh mkdir trash
grunt> sh ls -la trash
total 8
drwxr-xr-x 2 hadoop hadoop 4096 Mar 30 05:02 .
drwxr-xr-x 7 hadoop hadoop 4096 Mar 30 05:02 ..
grunt> sh touch trash/somefile.txt
grunt> sh ls trash
somefile.txt
```

@Zoran B. Djordjević

36

Processing Data

- The first step is loading data into Pig.
- **LOAD** command loads data into a “bag”, a collection of tuples

```
grunt> RAW_LOGS = LOAD 'file:///home/hadoop/access_log_1'
      using TextLoader as (line:chararray);
```
- Commands are normally terminated by a semicolon (;).
- If a command is to continue on the next line, the prompt changes to >>.
- Clause `(line:chararray)` creates a schema. It states that data are inserted into a tuple with a single column of type `chararray`.
- To see what we are doing we use **ILLUSTRATE** command.
- **Illustrate** takes a small sample of data and presents them to us.
- It actually presents the pipeline of processes that took place.
- If we know that we have a very small amount of data, we could also **DUMP** command, which dumps `raw_logs`, or any other variable.

@Zoran B. Djordjević

37

Illustrate RAW_LOGS

- ```
grunt> illustrate RAW_LOGS
```

```

| RAW_LOGS | line: bytearray
|
|-----
| | 74.125.75.17 - - [21/Jul/2009:12:28:16 -0700] "GET /gadgets/adp
owers/AlexaRank/ALL_ALL.xml HTTP/1.1" 200 1160 "-" "Mozilla/5.0 (compatible) Fee
dfetcher-Google; (+http://www.google.com/feedfetcher.html)"
RAW_LOGS

owers/AlexaRank/ALL_ALL.xml HTTP/1.1" 200 1160 "-" "Mozilla/5.0 (compatible) Fee
dfetcher-Google; (+http://www.google.com/feedfetcher.html)"

```
- Pig treated input as a `bytearray` and transformed it into a `chararray`.
- Please note that names of variables and many functions are case sensitive.
- Most frequently used commands are not.

@Zoran B. Djordjević

38

## Function to split the line

- To split lines into tokens or sections we need special functions.
- All UDF (User Defined Functions) are created as extensions of a set of Java classes which are stored in jar-s and then registered with the Pig.
- PiggyBank is one such library of functions. Information on functions contained in the PiggyBank could be found at:

<https://cwiki.apache.org/confluence/display/PIG/PiggyBank>

- On Elastic MapReduce EC2 instance with MapR M3 0.2. version of Hadoop, file `piggybank.jar` resides in the directory `/home/hadoop/lib/pig`.
- To use a particular function within `piggybank.jar` we need to create an alias for that function using `DEFINE` command.
- We know that function `EXTRACT` breaks the line using a regular expression and places matched regions into tuples. We use `grunt` command `DEFINE` to create short alias `EXTRACT`, otherwise we will have to call the function by its full Java path.

```
grunt> DEFINE EXTRACT
 org.apache.pig.piggybank.evaluation.string.EXTRACT();
```

@Zoran B. Djordjević

39

## Regular Expression

- The regular expression is a little tricky because the Apache log defines a couple of fields with quotes. What you need is:

```
'^(\S+) (\S+) (\S+) \[([\\w:/]+\s[+-]\d{4})\] "(.+)"
 (\S+) (\S+) "([^\"]*)" "([^\"]*)"'
```

- Java, and by extension, Pig need to escape all of those back slashes, so your expression reads:

```
'^((\\S+) (\\S+) (\\S+) \\[([\\w:/]+\\s[+\\-]\\d{4})\\]
 "(.*)" (\\S+) (\\S+) "([^\"]*)" "([^\"]*)"')
```

- Function `EXTRACT` takes `chararray line` as it first argument and the above regular expression as the second and `FOR EACH line` returns a tuple with matched strings, i.e. strings selected by parenthesis `((\\S+))` as elements.

@Zoran B. Djordjević

40

## Pig Parsing Command

```
LOGS_BASE = FOREACH RAW_LOGS GENERATE
 FLATTEN(
 EXTRACT(line, '^((\\S+) (\\S+) (\\S+) \\[([\\w:/]+\\S+
[+\\-]\\d{4})\\] "(.*)" (\\S+) (\\S+) "[^"]*)" "[^"]*)"')
)
 as (
 remoteAddr: chararray,
 remoteLogname: chararray,
 user: chararray,
 time: chararray,
 request: chararray,
 status: int,
 bytes_string: chararray,
 referrer: chararray,
 browser: chararray
);
```

- To see what came out, we could use: `ILLUSTRATE LOGS_BASE`

@Zoran B. Djordjević

41

## ILLUSTRATE LOGS\_BASE

```
grunt> ILLUSTRATE LOGS_BASE;

| RAW_LOGS | line: bytearray

| | 85.137.49.58 - - [21/Jul/2009:13:39:28 -0700] "GET /gwidgets/go
ogle-glossary.html HTTP/1.1" 200 870 "-" "Java/1.6.0_13"
RAW_LOGS

| | 85.137.49.58 - - [21/Jul/2009:13:39:28 -0700] "GET /gwidgets/go
ogle-glossary.html HTTP/1.1" 200 870 "-" "Java/1.6.0_13"
LOGS_BASE
rray
status: int
ay
200
grunt>
```

- We did split every line into a tuple

@Zoran B. Djordjević

42

## FOR ... EACH

- `FOREACH ... GENERATE` command creates a tuple for every (each) row of data.
- The “`FLATTEN`” command flattens nested structures.
- `FLATTEN` generates a new row for every element of a nested data bag.
- For example:

```
FLATTEN { ('foo.txt', ('bar', 'baz', 'bam')) }
```

Creates:

```
{ ('foo.txt', 'bar'),
 ('foo.txt', 'baz'),
 ('foo.txt', 'bam') }.
```

@Zoran B. Djordjević

43

## Schema

- Clause:

```
as (
 remoteAddr: chararray,
 remoteLogname: chararray,
 user: chararray,
 time: chararray,
 request: chararray,
 status: int,
 bytes_string: chararray,
 referrer: chararray,
 browser: chararray
);
```

- defined the schema for generated tuples. Most elements of `LOGS_BASE` bag of tuples are `chararrays`. `status` is apparently `int`.
- If schema is not defined, Pig tries to infer it based on element usage.

@Zoran B. Djordjević

44

## Narrow Query

- We want to determine the top 50 search terms used to refer to the website. This site apparently has the UTL : `http://example.com`
- We need to look at the referrer element in the tuples.
- The first thing to do is create a bag containing tuples with just this element:

```
grunt> REFERRER_ONLY = FOREACH LOGS_BASE GENERATE referrer;
```

- We want to see more tuples of data than ILLUSTRATE would provide.
- The DUMP command outputs the complete contents of a bag to the screen. There is usually too much data to display so we add a LIMIT instruction:

@Zoran B. Djordjević

45

## DUMP TEMP

```
grunt> DUMP TEMP;
2010-03-08 20:30:36,540 [main] INFO org.apache.pig.backend.local.executionengin
e.LocalPigLauncher
2010-03-08 20:30:36,540 [main] INFO org.apache.pig.backend.local.executionengin
e.LocalPigLauncher
2010-03-08 20:30:36,540 [main] INFO org.apache.pig.backend.local.executionengin
e.LocalPigLauncher
2010-03-08 20:30:36,540 [main] INFO org.apache.pig.backend.local.executionengin
e.LocalPigLauncher
2010-03-08 20:30:36,540 [main] INFO org.apache.pig.backend.local.executionengin
e.LocalPigLauncher
2010-03-08 20:30:36,540 [main] INFO org.apache.pig.backend.local.executionengin
e.LocalPigLauncher
(-)
(-)
(http://example.org/)
(http://example.org/)
(-)
grunt>
```

- Messages with dash (-) values don't have referrers.
- `http://example.org/` is just the site referring to itself.

@Zoran B. Djordjević

46

## Add FILTER to select google and bing

```
grunt> FILTERED = FILTER REFERRER_ONLY BY referrer matches '.*bing.*'
 OR referrer matches '.*google.*';
grunt> TEMP = LIMIT FILTERED 10;
grunt> DUMP TEMP;

(http://www.bing.com/search?q=value)
(http://www.bing.com/search?q=philmont)
(http://www.bing.com/search?q=value)
(http://www.bing.com/search?q=philmont)
(http://images.google.co.th/imgres?imgurl=http://example.org/images/toothin
imgrefurl=http://example.org/%3Fnews%3Dall&usg=__KOkVEA0KxJVHqDP2vTY1XLJKZN
340&w=640&sz=48&hl=th&start=7&tbnid=CFyUh41SsH2g9M:&tbnh=73&tbnw=137&prev=/
s%3Fq%3Dapple%2Badvertisement%26gbv%3D2%26hl%3Dth%26sa%3DX)
(http://images.google.co.th/imgres?imgurl=http://
imgrefurl=http://example.org/%3Fnews%3Dall&usg=__
340&w=640&sz=48&hl=th&start=7&tbnid=CFyUh41SsH2g9M:&tbnh=73&tbnw=137&prev=/
s%3Fq%3Dapple%2Badvertisement%26gbv%3D2%26hl%3Dth%26sa%3DX)
```

@Zoran B. Djordjević

47

## Extract Search Queries

- Both search engines indicate the beginning of the query string using a key of "q=" and then separating query terms with "+".
- To extract these, the first step is to use our EXTRACT function to grab everything from the "q=" up to the end of a string or an ampersand (&).
- We then FILTER out any string that does not match our regular expression.

```
grunt> SEARCH_TERMS = FOREACH FILTERED GENERATE
>> FLATTEN(EXTRACT(referrer, '.*[&\\?]q=([^&]+).*'))
>> as terms:chararray;
grunt> SEARCH_TERMS_FILTERED = FILTER SEARCH_TERMS BY NOT $0 IS
 NULL;
grunt> DUMP SEARCH_TERMS_FILTERED
(search)
(value)
(about+me+website)
(linux+%2Fusr%2Fbin%2Fperl)
(%21%2Fusr%2Fbin%2Fperl+-w)
(value)
```

@Zoran B. Djordjević

48



## Count the Search Terms

- To count most frequently used terms we use Pig operators GROUP and COUNT:

```
Grunt> SEARCH_TERMS_COUNT = FOREACH (GROUP SEARCH_TERMS_FILTERED BY $0)
 GENERATE $0, COUNT($1) as num;
Grunt> SEARCH_TERMS_COUNT_SORTED = LIMIT (ORDER SEARCH_TERMS_COUNT BY num
 DESC) 50;
Grunt > DUMP SEARCH_TERMS_COUNT_SORTED;
```

@Zoran B. Djordjević

49

## Most Frequently Queried Terms

|                                   |                                    |
|-----------------------------------|------------------------------------|
| (value,100L)                      | (escalator,3L)                     |
| (views,70L)                       | (fishing,3L)                       |
| (login,39L)                       | (hadoop+0.20,3L)                   |
| (search,37L)                      | (homemade,3L)                      |
| (items,19L)                       | (m0n0wall+ipv6,3L)                 |
| (bigtable,12L)                    | (pebble,3L)                        |
| (google+bigtable,9L)              | (phone,3L)                         |
| (%23%21%2Fusr%2Fbin%2Fperl,8L)    | (pig+sample,3L)                    |
| (philmont+pictures,7L)            | (seahawks,3L)                      |
| (%23%21%2Fusr%2Fbin%2Fperl+-w,6L) | (travis,3L)                        |
| (philmont,6L)                     | (%21%2Fusr%2Fbin%2Fperl+-w,2L)     |
| (google+quick+links,5L)           | (%22rm+-rf%22,2L)                  |
| (pig,5L)                          | (%23!%2Fusr%2Fbin%2Fperl+-w,2L)    |
| (pvc+instrument,5L)               | (%23%21%2Fusr%2Fbin%2Fperl+-wT,2L) |
| (vegas,5L)                        | (Andrew+sample,2L)                 |
| (about+me+website,4L)             | (BigTable,2L)                      |
| (google+big+table,4L)             | (Google+BigTable,2L)               |
| (pig+the+pc+nerd,4L)              | (Website+about+me,2L)              |
| (seattle,4L)                      | (apple,2L)                         |
| (walla,4L)                        | (balam,2L)                         |
| (bikes,3L)                        | (big+table,2L)                     |
| (biking,3L)                       | (bigtable+example,2L)              |
| (comments,3L)                     | (bigtable+google,2L)               |

@Zoran B. Djordjević

50

## Store Your Results

- Store your data with STORE:

```
STORE SEARCH_TERMS_COUNT_SORTED into
 'file:///home/hadoop/output/run0';
```

- Examine the file with CAT

```
CAT file:///home/hadoop/output/run0
```

To get out we type:

```
grunt> quit;
```

@Zoran B. Djordjević

51

## Store Your Script

- We do not want to keep repeating this typing over and over again and keep working in the interactive mode.
- We can save our commands in a file (script), save that script in an S3 bucket and in the future invoke that script for another job on, perhaps, a different machine.
- We will do that on the command prompt of the remote machine, where we could open `vi` or `vim` and save the script.
- Note that all command lines end with semicolon “;”.
- Also, while working in the interactive mode our input file was accessed as:
- `RAW_LOGS = LOAD 'file:///home/hadoop/access_log_1'`
- That is not convenient. We might have that file in another location.
- We “parameterize” log location by introducing place holder `'$INPUT'`.
- Similarly we parameterize the output file with `'$OUTPUT'`

@Zoran B. Djordjević

52

## Saved Script

```
register file:/home/hadoop/lib/pig/piggybank.jar
DEFINE EXTRACT org.apache.pig.piggybank.evaluation.string.EXTRACT();
RAW_LOGS = LOAD '$INPUT' USING TextLoader as (line:chararray);
LOGS_BASE = foreach RAW_LOGS generate FLATTEN (EXTRACT (line, '^(\S+)
(\S+) (\S+) \[([\\w:/]+\s[+-]\d{4})\]' "(.*)" (\S+) (\S+)
"([\"]*)" "([\"]*)"') as (remoteAddr:chararray,
remoteLogname:chararray, user:chararray, time:chararray,
request:chararray, status:int, bytes_string:chararray,
referrer:chararray, browser:chararray) ;
REFERRER_ONLY = FOREACH LOGS_BASE GENERATE referrer;
FILTERED = FILTER REFERRER_ONLY BY referrer matches '.*bing.*' OR referrer
matches '.*google.*';
SEARCH_TERMS = FOREACH FILTERED GENERATE FLATTEN(EXTRACT(referrer,
'.*[&\\?]q=([&]+).*)') as terms:chararray;
SEARCH_TERMS_FILTERED = FILTER SEARCH_TERMS BY NOT $0 IS NULL;
SEARCH_TERMS_COUNT = FOREACH (GROUP SEARCH_TERMS_FILTERED BY $0) GENERATE
$0, COUNT($1) as num;
SEARCH_TERMS_COUNT_SORTED = LIMIT (ORDER SEARCH_TERMS_COUNT BY num DESC) 50;
STORE SEARCH_TERMS_COUNT_SORTED into '$OUTPUT';
```

- Save in /home/hadoop/script.pig

@Zoran B. Djordjević

53

## Run Script from the Command Line

- To run your Pig job from the command line do the following:
- On the command prompt of your Hadoop system type:

```
$ pig -p INPUT=file:///home/hadoop/access_log_1
-p OUTPUT=file:///home/hadoop/output/run2
file:///home/hadoop/script.pig
```

- The output should end up in /home/hadoop/output/run2 directory.

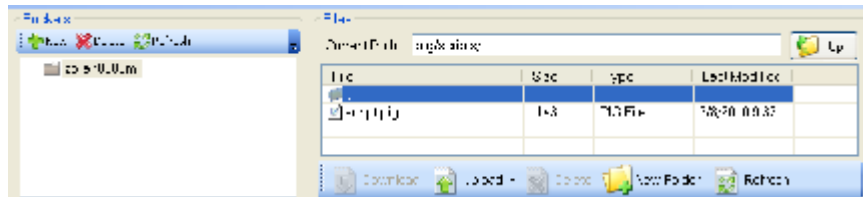
@Zoran B. Djordjević

54

## Upload Script to S3

- Hadoop's `dfs` command, executed on the command prompt of the remote (Hadoop) system will copy the script to a properly named S3 bucket:

```
$ hadoop dfs -copyFromLocal /home/hadoop/script.pig
s3://zoran0302mr/pig/scripts/script.pig
```



Since we are done with the interactive job flow we could terminate it.

@Zoran B. Djordjević

55

## Pig Latin Syntax

Zoran B. Djordjević  
csci e185 Big Data Analytics

56

## Data Types

- Pig Latin statements work with relations.
- A relation can be defined as follows:
  - A relation is a bag (more specifically, an outer bag).
  - A bag is a collection of tuples.
  - A tuple is an ordered set of fields.
  - A field is a piece of data.
- A Pig relation is similar to a table in a relational database, where the tuples in the bag correspond to the rows in a table.
- Unlike a relational table, Pig relations don't require that every tuple contain the same number of fields or that the fields in the same position (column) have the same type.
- Relations are unordered which means there is no guarantee that tuples are processed in any particular order.
- Processing may be parallelized in which case tuples are not processed according to any ordering.

57

## Referencing Relations, Fields

- Names are assigned by you using schemas (or, in the case of the GROUP operator and some functions, by the system).
- You can use any name that is not a Pig keyword;
- When you names are assigned to fields we can still refer to the fields using positional notation. For debugging and comprehension, better use names.

```
A = LOAD 'student' USING PigStorage() AS (name:chararray,
age:int, gpa:float); << A is a relation
X = FOREACH A GENERATE name,$2; << Could mix notations
DUMP X;
(John,4.0F)
A = LOAD 'data' AS (f1:int,f2:int,f3:int);
B = FOREACH A GENERATE $3; << Will generate an error.
DUMP B;
2009-01-21 23:03:46,715 [main] ERROR
org.apache.pig.tools.grunt.GruntParser - java.io.IOException:
Out of bound access. Trying to access non-existent : 3. Schema
{f1: bytearray,f2: bytearray,f3: bytearray} has 3 column(s).
```

58

## Referencing Fields of Complex Types

- Fields in a `tuple` could be atomic or complex type: `bag`, `tuple`, and `maps`.
- Schemas could name fields in complex data types.
- Dereference operator (dot, ".") is used for referencing fields in complex types.
- A schema for complex data types (in this case, `tuple`) is used to load the data. Then, dereference operators (the dot in `t1.t1a` and `t2.$0`) accesses fields in the tuples.

```
cat da;
(3,8,9) (4,5,6)
(1,4,7) (3,7,5)
(2,5,8) (9,5,8)
A = load 'da' 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;
DUMP X;
(3,4)
(1,3)
(2,9)
```

Note that `tuple` is key word  
and the data type

59

## Data Types

| Simple Data Types  | Description                                      | Example                                                                 |
|--------------------|--------------------------------------------------|-------------------------------------------------------------------------|
| Scalars            |                                                  |                                                                         |
| int                | Signed 32-bit integer                            | 10                                                                      |
| long               | Signed 64-bit integer                            | Data: 10L or 10l<br>Display: 10L                                        |
| float              | 32-bit floating point                            | Data: 10.5F or 10.5f or 10.5e2f or 10.5E2F<br>Display: 10.5F or 1050.0F |
| double             | 64-bit floating point                            | Data: 10.5 or 10.5e2 or 10.5E2<br>Display: 10.5 or 1050.0               |
| Arrays             |                                                  |                                                                         |
| chararray          | Character array (string) in Unicode UTF-8 format | hello world                                                             |
| bytearray          | Byte array (blob)                                |                                                                         |
| Complex Data Types |                                                  |                                                                         |
| tuple              | An ordered set of fields.                        | (19,2)                                                                  |
| bag                | An collection of tuples.                         | {(19,2),(18,1)}                                                         |
| map                | A set of key value pairs.                        | [open#apache]                                                           |

60

## Implicit Conversion, Casting

- Use schemas to assign types to fields. Untyped fields default to bytearray.
- Implicit conversion applied based on context in which that data is used.

```
A = LOAD 'data' AS (f1,f2,f3);
B = FOREACH A GENERATE f1 + 5; << f1 converted to int
C = FOREACH A generate f1 + f2; << f1 and f2 converted to double
```

- If the data does not conform to the schema, the loader will generate a null value or an error.

```
A = LOAD 'data' AS (name:chararray, age:int, gpa:float);
```

- If an explicit cast is not supported, an error will occur. Cannot cast a chararray to int.

```
A = LOAD 'data' AS (name:chararray, age:int, gpa:float);
B = FOREACH A GENERATE (int)name; << Will cause an error
```

- Incompatible types that could not be cast implicitly cause an error.
- For example, you cannot add chararray and float.

```
A = LOAD 'data' AS (name:chararray, age:int, gpa:float);
B = FOREACH A GENERATE name + gpa;
```

61

## Tuple

- A tuple is an ordered set of fields.
- Syntax
 

```
(field [, field ...])
```

  - A tuple is enclosed in parentheses ( ).
  - A field in a tuple is a piece of data.
  - A field can be any data type (including tuple and bag).
- A tuple is a row with one or more fields, where each field can be any data type and any field may or may not have data.
- If a field has no data, then the following happens:
  - In a load statement, the loader will inject null into the tuple.
  - The actual value that is substituted for null is loader specific;
  - PigStorage substitutes an empty field for null.
  - In a non-load statement, if a requested field is missing from a tuple, Pig will inject null.

62

## Bag

- A bag is a collection of tuples.
- Syntax: Inner bag  

```
{ tuple [, tuple ...] }
```
- An inner bag is enclosed in curly brackets { }.
- A bag can have duplicate tuples.
- A bag can have tuples with differing numbers of fields.
  - If Pig tries to access a field that does not exist, a null value is substituted.
- A bag can have tuples with fields that have different data types.
  - For Pig to effectively process bags, the schemas of the tuples within those bags should be the same.
  - If half of the tuples include chararray fields and while the other half include float fields, only half of the tuples will participate in any kind of arithmetics because the chararray fields will be converted to null.
- Bags have two forms: outer bag (or relation) and inner bag.

63

## Outer Bag

```
A = LOAD 'data' as (f1:int, f2:int, f3:int);
DUMP A;
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
```

- A is a relation or bag of tuples. This is an outer bag.

64



### Inner Bag

- We could group relation A by the first field to form relation X.
- X is a relation or bag of tuples.
  - The tuples in relation X have two fields.
  - The first field is type int.
  - The second field is type bag; That bag is an inner bag.

```
X = GROUP A BY f1;
DUMP X;
(1, { (1, 2, 3) })
(4, { (4, 2, 1) , (4, 3, 3) })
(8, { (8, 3, 4) })
```

65

### Map

- A map is a set of key value pairs.
- Syntax
 

```
[key#value <, key#value ...>]
```
- Maps are enclosed in straight brackets [ ].
- Key and value in a pair are separated by the pound sign #.
- Key : Must be chararray data type. Must be a unique value in a relation.
- Value: Any data type.
- Key values within a relation must be unique.
- In the following example the map includes two key value pairs.
 

```
[name#John, phone#555-1212]
```
- keys are: name and phone, corresponding values are John and 555-1212

66

## Nulls

- Nulls are implemented using the SQL meaning of unknown or non-existent.
- Nulls can occur naturally in data or can be the result of an operation.
- If a FILTER expression results in null value, the filter does not pass.

| Operator                                             | Result of Interaction with Null                                          |
|------------------------------------------------------|--------------------------------------------------------------------------|
| Comparison operators: $=, <, >, <=, >=, <>$          | If either sub-expression is null, the result is null.                    |
| Comparison operator: matches                         | If the string being matched or pattern str. null, the result is null.    |
| Arithmetic operators: $+, *, /, \% modulo, / bitand$ | If either sub-expression is null, the resulting expression is null.      |
| Null test: is null                                   | If the tested value is null, returns true; otherwise, returns false.     |
| Null test: is not null                               | If the tested value is not null, returns true; otherwise, returns false. |
| Derivative operators                                 | If the derived tuple or msg is null, returns null. Tuple ( ) or msg ( )  |
| Cast operator                                        | Casting a null from one type to another type results in a null.          |
| AVG, MIN, MAX, SUM                                   | These functions ignore nulls.                                            |
| COUNT                                                | This function counts all values, including nulls.                        |
| CONCAT                                               | If either sub-expression is null, the resulting expression is null.      |
| SIZE                                                 | If the tested object is null, returns null.                              |

67

## Operations producing Nulls

- Nulls can be the result of:
  - Division by zero
  - User defined functions (UDFs)
  - Dereferencing a field that does not exist.
  - Dereferencing a key that does not exist in a map
  - Accessing a field that does not exist in a tuple.
- Example: Accessing a field that does not exist in a tuple

```
cat data;
 2 3
4
A = LOAD 'data' AS (f1:int,f2:int,f3:int)
DUMP A;
(,2,3)
(4,,)
B = FOREACH A GENERATE f1,f2;
DUMP B;
(,2)
(4,)
```

68

## Nulls and Load Function

- Nulls can occur naturally in the data.
- If nulls are part of the data, it is the responsibility of the load function to handle them correctly. What is considered a null value is loader-specific.
- The load function should always communicate null values to Pig by producing Java nulls.
- The Pig Latin load functions (for example, PigStorage and TextLoader) produce null values wherever data is missing. Empty strings (chararrays) are not loaded; instead, they are replaced by nulls.
- PigStorage is the default load function for the LOAD operator.
- In the following "is not null" operator is used to filter names with null values.

```
A = LOAD 'student' AS (name, age, gpa);
B = FILTER A BY name is not null
```

69

## Constants

- Pig provides constant representations for all data types except bytearrays

| Data Type     | Example                           |                                          |
|---------------|-----------------------------------|------------------------------------------|
| Scalars       |                                   |                                          |
| int           | 19                                |                                          |
| long          | 19L                               |                                          |
| float         | 19.2F or 1.92e2F                  |                                          |
| double        | 19.2 or 1.92e2                    |                                          |
| Arrays        |                                   |                                          |
| chararray     | 'hello world'                     |                                          |
| bytearray     | <del>bytearray</del>              | Not applicable.                          |
| Complex Types |                                   |                                          |
| tuple         | (19, 2, 1)                        | A constant in this form creates a tuple. |
| bag           | { (19, 2), (1, 2) }               | A constant in this form creates a bag.   |
| map           | [ 'name' : 'A John', 'age' : 25 ] | A constant in this form creates a map.   |

- Complex constants (either with or without values) can be used in the same places scalar constants can be used; that is, in FILTER and GENERATE statements.
- A = LOAD 'data' USING MyStorage() AS (T: tuple(name:chararray, age: int));
- B = FILTER A BY T == ('john', 25);

70

```

D = FOREACH B GENERATE T.name, (25 * 5 - C1) * ((1 - 5 - 10))

```

## Expressions

- Expressions are language constructs used with the FILTER, FOREACH, GROUP, and SPLIT operators as well as the eval functions.
- Expressions are written in conventional mathematical infix notation and are adapted to the UTF-8 character set. Depending on the context, expressions can include:
  - Any Pig data type (simple data types, complex data types)
  - Any Pig operator (arithmetic, comparison, null, boolean, dereference, sign, and cast)
  - Any Pig built-in function.
  - Any user-defined function (UDF) written in Java.

71

## Examples of Expressions

- An arithmetic expression could look like this:

```
X = GROUP A BY f2*f3;
```

- A string expression could look like this, where a and b are both chararrays:

```
X = FOREACH A GENERATE CONCAT(a,b);
```

- A boolean expression could look like this:

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

72

### Star Expression

- The star symbol, \*, can be used to represent all the fields of a tuple. It is equivalent to writing out the fields explicitly.
- In the following example the definition of B and C are the same
- `MyUDF` will be invoked with exactly the same arguments in both cases.

```
A = LOAD 'data' USING MyStorage() AS (name:chararray, age: int);
B = FOREACH A GENERATE *, MyUDF(name, age);
C = FOREACH A GENERATE name, age, MyUDF(*);
```

- A common error when using the star expression is the following:

```
G = GROUP A BY $0;
C = FOREACH G GENERATE COUNT(*)
```

- In this example, the programmer really wants to count the number of elements in the bag in the second field: `COUNT($1)`.

73

### Schemas

- Schemas enable you to assign names to and declare types for fields.
- Schemas are optional but encouraged.
- Type declarations result in better parse-time error checking and efficient code.
- Defined with `AS` keyword with `LOAD`, `STREAM`, and `FOREACH` operators.
- You can define a schema that includes both the field name and field type.
- You can define a schema that includes the field name only; in this case, the field type defaults to `bytearray`.
- You can choose not to define a schema; in this case, the field is un-named and the field type defaults to `bytearray`.
- If you assign a name to a field, you can refer to that field by name or by position.
- If you don't assign a name to a field (the field is un-named) you can only refer to the field using positional notation.
- If you assign a type to a field, you can subsequently change the type using the cast operators.
- If you don't assign a type to a field, the field defaults to `bytearray`; you can change the default type using the cast operators.

74

## Schemas with LOAD, Stream, FOREACH

- With LOAD and STREAM statements, the schema following the AS keyword must be enclosed in parentheses.

- This LOAD statement includes a schema definition for simple data types.

```
A = LOAD 'data' AS (f1:int, f2:int);
```

- With FOREACH statements, the schema following the AS keyword must be enclosed in parentheses when the FLATTEN operator is used. Otherwise, the schema should not be enclosed in parentheses.

- This FOREACH statement includes FLATTEN and a schema for simple types.

```
X = FOREACH C GENERATE FLATTEN(B) AS (f1:int, f2:int, f3:int);
```

- The following FOREACH statement includes a schema for simple types and no parenthesis

```
X = FOREACH A GENERATE f1+f2 AS x1:int;
```

### Syntax

```
(alias[:type]) [, (alias[:type]) ...]
```

alias: The name assigned to the field.

type: (Optional) The data type assigned to the field.

The alias and type are separated by a colon (:).

75

## Schema Examples with Simple Types

```
cat student;
John 18 4.0
Mary 19 3.8
Joe 18 3.8
A = LOAD 'student' AS (name:chararray, age:int, gpa:float);
DESCRIBE A;
A: {name: chararray,age: int,gpa: float}
DUMP A;
(John,18,4.0F)
(Mary,19,3.8F)
(Joe,18,3.8F)
cat student;
John 18 4.0
Mary 19 3.8
Joe 18 3.8
A = LOAD 'data' AS (name:chararray, age:int, gpa);
DESCRIBE A;
A: {name: chararray,age: int,gpa: bytearray}
DUMP A;
(John,18,4.0)
(Mary,19,3.8)
(Joe,18,3.8)
```

76

## A Schema with one tuple

### ■ Syntax

- `alias[:tuple] (alias[:type])[, (alias[:type]) ...]` )
- `alias`: The name assigned to the tuple.
- `:tuple` (Optional) The data type, tuple (case insensitive).
- `()` The designation for a tuple, a set of parentheses.
- `alias[:type]` : The constituents of the tuple.

### ■ Next schema defines one tuple. The load statements are equivalent.

```
cat data;
(3,8,9)
(1,4,7)
(2,5,8)
A = LOAD 'data' AS (T: tuple (f1:int, f2:int, f3:int));
A = LOAD 'data' AS (T: (f1:int, f2:int, f3:int));
DESCRIBE A;
A: {T: (f1: int,f2: int,f3: int)}
DUMP A;
((3,8,9))
((1,4,7))
((2,5,8))
```

77

## A Schema with two tuples

```
cat data;
(3,8,9) (mary,19)
(1,4,7) (john,18)
(2,5,8) (joe,18)
A = LOAD data
 AS (F:tuple(f1:int,f2:int,f3:int),T:tuple(t1:chararray,t2:int));
DESCRIBE A;
A: {F: (f1: int,f2: int,f3: int),T: (t1: chararray,t2: int)}

DUMP A;
((3,8,9),(mary,19))
((1,4,7),(john,18))
((2,5,8),(joe,18))
```

78

## Bag Schema

- A bag is a collection of tuples.
- Syntax
  - `alias[:bag] {tuple}`
- { } The designation for a bag, a set of curly brackets.
- This schema defines a bag. The two load statements are equivalent.
- `cat data;`
- `{{(3,8,9)}}`
- `{{(1,4,7)}}`
- `{{(2,5,8)}}`
- `A = LOAD 'data' AS (B: bag {T: tuple(t1:int, t2:int, t3:int)});`
- `A = LOAD 'data' AS (B: {T: (t1:int, t2:int, t3:int)});`
- `DESCRIBE A;`
- `A: {B: {T: (t1: int,t2: int,t3: int)}}`
- `DUMP A;`
- `(( (3,8,9) ))`
- `(( (1,4,7) ))`
- `(( (2,5,8) ))`

79

## Map Schema

- A map is a set of key value pairs.
- Syntax
  - `alias<:map> [ ]`
- [ ] The designation for a map, a set of straight brackets.
- This schema defines a map. The load statements are equivalent.
- `cat data;`
- `[open#apache]`
- `[apache#hadoop]`
- `A = LOAD 'data' AS (M:map [ ]);`
- `A = LOAD 'data' AS (M: [ ]);`
- `DESCRIBE A;`
- `a: {M: map[ ]}`
- `DUMP A;`
- `([open#apache])`
- `([apache#hadoop])`

80



### Schemas with Multiple Types

- You can define schemas for data that includes multiple types.
- These schemas include a tuple, bag, and map.

```
A = LOAD 'mydata' AS (T1:tuple(f1:int, f2:int),
 B:bag{T2:tuple(t1:float,t2:float)}, M:map[]);
```

```
A = LOAD 'mydata' AS (T1:(f1:int, f2:int),
 B:{T2:(t1:float,t2:float)}, M:[]);
```

81

### Parameter Substitution

- Pig allows you to substitute values of parameters at run time.
- We could pass parameters to `pig`, `exec`, `run`, and `explain` commands.
- Parameters are passed at the command line or using preprocessor statements:

- Specifying parameters using the Pig command line

```
pig {-param param_name = param_value | -param_file
 file_name} [-debug | -dryrun] script
```

- Specifying parameters using preprocessor statements in a Pig script

```
{%declare | %default} param_name param_value
```

82

## Preprocessor Statements

### %declare

- Preprocessor statement included in a Pig script.
- Use to describe one parameter in terms of other parameters.
- The declare statement is processed prior to running the Pig script.
- The scope of a parameter value defined using declare is all the lines following the declare statement until the next declare statement that defines the same parameter is encountered.
- In this example the command is executed and its `stdout` is used as the parameter value.

```
%declare CMD `generate_date`;
A = LOAD '/data/mydata/$CMD';
B = FILTER A BY $0>'5';
```

- In this example the characters are enclosed in single or double quotes, and the quote within the sequence of characters is escaped.

```
%declare DES 'Joe\'s URL';
A = LOAD 'data' AS (name, description, url);
B = FILTER A BY description == '$DES';
```

83

## Preprocessor Statements

### %default

- Preprocessor statement included in a Pig script.
- Provides a default value for a parameter. The default value has the lowest priority. Used if a parameter value has not been defined by other means.
- The default statement is processed prior to running the Pig script.
- The scope is the same as for `%declare`.
- In this example the parameter `(DATE)` and value `('20090101')` are specified in the Pig script using the default statement.
- If a value for `DATE` is not specified elsewhere, the default value `20090101` is used.

```
%default DATE '20090101';
A = load '/data/mydata/$DATE';
```

84

### Parameters on the command line

- Suppose we have a data file called 'mydata' and a pig script called 'myscript.pig'.

mydata

```
1 2 3
4 2 1
8 3 4
```

myscript.pig

```
A = LOAD '$data' USING PigStorage() AS (f1:int, f2:int, f3:int);
DUMP A;
```

- The parameter (data) and the parameter value (mydata) are specified in the command line.
- If the parameter name in the command line (data) and the parameter name in the script (\$data) do not match, the script will not run.

```
$ pig -param data=mydata myscript.pig
(1,2,3)
(4,2,1)
(8,3,4)
```

85

### Parameters in a parameter file

- We could have a parameter file called 'myparams.' with the following content:

```
my parameters
data1 = mydata1
cmd = `generate_name`
```

- The parameters and values are passed to the script using that file.

```
$ pig -param_file myparams script2.pig
```

86

## Arithmetic Operators

| Operator       | Symbol | Description                                                                                                                                                                                                                                |
|----------------|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| addition       | +      |                                                                                                                                                                                                                                            |
| subtraction    | -      |                                                                                                                                                                                                                                            |
| multiplication | *      |                                                                                                                                                                                                                                            |
| division       | /      |                                                                                                                                                                                                                                            |
| modulo         | %      | Returns the remainder of a divided by b (a%b).<br>Works with integral numbers (int, long).                                                                                                                                                 |
| bincond        | ?:     | (condition ? value_if_true : value_if_false)<br>The bincond should be enclosed in parenthesis.<br>The schemas for the two conditional outputs of the bincond should match.<br>Use expressions only (relational operators are not allowed). |

- Pig has a sign operator, as well.
- + does nothing, - changes the sign.

```
A = LOAD 'data' as (x, y, z);
B = FOREACH A GENERATE -x, y;
```

87

## Example of an arithmetic calculation

- Suppose we have relation A.

```
A = LOAD 'data' AS (f1:int, f2:int,
 B:bag{T:tuple(t1:int,t2:int)});
DUMP A;
(10,1,{(2,3),(4,6)})
(10,3,{(2,3),(4,6)})
(10,6,{(2,3),(4,6),(5,7)})
```

- The modulo operator is used with fields f1 and f2.

```
X = FOREACH A GENERATE f1, f2, f1%f2;
DUMP X;
(10,1,0)
(10,3,1)
(10,6,4)
```

88

### Example of an arithmetic calculation

- In this example the bincond operator is used with fields f2 and B.
- The condition is "f2 equals 1";
- if the condition is true, return 1; if the condition is false, return the count of the number of tuples in B.

```
X = FOREACH A GENERATE f2, (f2==1?1:COUNT(B));
```

```
DUMP X;
```

```
(1, 1L)
```

```
(3, 2L)
```

```
(6, 3L)
```

- Fortunately, there are no fractions.

89

### Comparison Operators

| Operator                 | Symbol  | Notes                                                                                        |
|--------------------------|---------|----------------------------------------------------------------------------------------------|
| equal                    | ==      |                                                                                              |
| not equal                | !=      |                                                                                              |
| less than                | <       |                                                                                              |
| greater than             | >       |                                                                                              |
| less than or equal to    | <=      |                                                                                              |
| greater than or equal to | >=      |                                                                                              |
| pattern matching         | matches | Regular expression matching.<br>Use the Java <a href="#">format</a> for regular expressions. |

90

### Examples of use of Comparison Operators

- Use comparison operators with numeric and string data

- Example: numeric

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

- Example: string

```
X = FILTER A BY (f2 == 'apache');
```

- Example: matches

```
X = FILTER A BY (f1 matches '.*apache.*');
```

- Types Table: equal (==) and not equal (!=) operators

91

### Boolean Operators

- Pig supports Boolean operators: AND, OR and NOT.
- Pig does not support a Boolean data type.
- However, the result of a Boolean expression (an expression that includes Boolean and comparison operators) is always of type Boolean (true or false).

- Example

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

92

## Dereference Operators

- Tuple dereferencing can be done by name
  - `(tuple.field_name)` or position
  - `(mytuple.$0)`.
- If a set of fields are dereferenced
  - `(tuple.(name1, name2))` or
  - `(tuple.($0, $1))`,
- the expression represents a tuple composed of the specified fields.
- If the dot operator is applied to a bytearray, the bytearray will be assumed to be a tuple.

93

## Example, Dereferencing Tuples

- Suppose we have relation A.
 

```
LOAD 'data' as (f1:int, f2:tuple(t1:int,t2:int,t3:int));
DUMP A;
(1, (1,2,3))
(2, (4,5,6))
(3, (7,8,9))
(4, (1,4,7))
(5, (2,5,8))
```
- In this example dereferencing is used to retrieve two fields from tuple f2.
 

```
X = FOREACH A GENERATE f2.t1,f2.t3;
DUMP X;
(1,3)
(4,6)
(7,9)
(1,7)
(2,8)
```

94

## Dereference Operators

- Bag dereferencing can be done by name
  - `bag.field_name` or position
  - `bag.$0.`
- If a set of fields are dereferenced
  - `bag.(name1, name2 or`
  - `bag.($0, $1)),`
- the expression represents a bag composed of the specified fields.
  
- Map dereferencing must be done by key
  - `field_name#key` or position
  - `$0#key).`
- If the pound operator is applied to a bytearray, the bytearray is assumed to be a map.
- If the key does not exist, the empty string is returned.

95

## Examples of Dereference Operator with Bags

- Suppose we have relation B, formed by grouping relation A.
- ```
A = LOAD 'data' AS (f1:int, f2:int,f3:int);
DUMP A;
(4,2,1)
(8,3,4)
(4,3,3)
(8,4,3)
B = GROUP A BY f1;
DUMP B;
(4, {(4,2,1), (4,3,3)})
(8, {(8,3,4), (8,4,3)})
ILLUSTRATE B;
```
- ```

| b | group: int | a: bag({f1: int,f2: int,f3: int}) |

```
- We want to project the first field (f1) of each tuple in the bag (a).
- ```
X = FOREACH B GENERATE a.f1;
DUMP X;
({(4), (4)})
({(8), (8)})
```

96

Dereferencing a Map

- Suppose we have relation A.

```
A = LOAD 'data' AS (f1:int, f2:map[]);
DUMP A;
(1,[open#apache])
(2,[apache#hadoop])
(3,[hadoop#pig])
(4,[pig#grunt])
```

- In this example dereferencing is used to look up the value of key 'open'.

```
X = FOREACH A GENERATE f2#'open';
DUMP X;
(apache)
()
()
()
```

97

Dereferencing Tuple and Bag

- Suppose we have relation B, formed by grouping relation A.

```
A = LOAD 'data' AS (f1:int, f2:int, f3:int);
DUMP A;
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
B = GROUP A BY (f1,f2);
DUMP B;
((1,2),{(1,2,3)})
((4,2),{(4,2,1)})
((4,3),{(4,3,3)})
((8,3),{(8,3,4)})
ILLUSTRATE B;
-----
| b | group: tuple({f1: int,f2: int}) | a: bag({f1: int,f2: int,f3: int}) |
-----
|   | (8, 3)                        | {(8, 3, 4), (8, 3, 4)} |
-----
```

- Project a field (f1) from a tuple (group) and a field (f1) from a bag (a) .

```
X = FOREACH B GENERATE group.f1, a.f1;
DUMP X;
(1,{(1)})
(4,{(4)})
(4,{(4)})
(8,{(8)})
```

98

FLATTEN Operator

- The FLATTEN operator changes the structure of tuples and bags.
- Flatten un-nests tuples as well as bags. The idea is the same, but the operation and result is different for each type of structure.
- For tuples, flatten substitutes the fields of a tuple in place of the tuple.
- For example, if a relation has a tuple of the form
 $(a, (b, c))$
- Expression
`GENERATE $0, flatten($1),`
- will cause that tuple to become
 $(a, b, c).$

99

FLATTEN Operator

- For bags, the situation becomes more complicated.
- When we un-nest a bag, we create new tuples.
- If we have a relation that is made up of tuples of the form
 $((b, c), (d, e))$
- and we apply
`GENERATE flatten($0),`
- we end up with two tuples
 (b, c) and $(d, e).$
- When we remove a level of nesting in a bag, sometimes we cause a cross product to happen.
- Consider a relation that has a tuple of the form $(a, \{(b, c), (d, e)\})$, commonly produced by the GROUP operator.
- If we apply the expression `GENERATE $0, flatten($1)` to this tuple, we will create new tuples: (a, b, c) and $(a, d, e).$

100

Cast

- Cast operators enable you to cast or convert data from one type to another, as long as conversion is supported.
- Suppose you have an integer field, myint, which you want to convert to a string. You can cast this field from int to chararray using (chararray)myint.
- A field can be explicitly cast. Once cast, the field remains that type (it is not automatically cast back). In this example \$0 is explicitly cast to int.

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

- Where possible, Pig performs implicit casts. In this example \$0 is cast to int (regardless of underlying data) and \$1 is cast to double.
- B = FOREACH A GENERATE \$0 + 1, \$1 + 1.0
- When two bytearrays are used in arithmetic expressions or with built-in aggregate functions (such as SUM) they are implicitly cast to double.
- If the underlying data is really int or long, you'll get better performance by declaring the type or explicitly casting the data. 101

Supported Casts

	To								
From	bag	tuple	map	int	long	float	double	chararray	bytearray
bag		no	no	no	no	no	no	no	no
tuple	yes		yes	no	yes	yes	yes	yes	yes
		no		no	no	no	no	no	no
int	no	no	no		yes	yes	yes	no	no
long	no	no	no	yes		yes	yes	no	no
float	yes	no	yes	yes	yes		yes	yes	yes
double	no	no	no	yes	yes	yes		no	no
chararray	yes	no	yes	no	yes	no	no		no
bytearray	yes	yes	yes	yes	yes	yes	yes	yes	

Syntax

```
{(data_type) | (tuple(data_type)) | (bag{tuple(data_type)}) | (map[])} field
```

102

Examples of Cast

- In this example a `bytearray` (fld in relation A) is cast to type `map`.

```
cat data;
[open#apache]
[apache#hadoop]
A = LOAD 'data' AS fld:bytearray;
DESCRIBE A;
A: {fld: bytearray}
DUMP A;
([open#apache])
([apache#hadoop])

B = FOREACH A GENERATE ((map[])fld;
DESCRIBE B;
B: {map[ ]}
DUMP B;
([open#apache])
([apache#hadoop])
```

103

bytearray Cast to type bag

- In this example a `bytearray` (fld in relation A) is cast to type `bag`.

```
cat data;
{(4829090493980522200L)}
{(4893298569862837493L)}
A = LOAD 'data' AS fld:bytearray;
DESCRIBE A;
A: {fld: bytearray}
DUMP A;
({(4829090493980522200L)})
({(4893298569862837493L)})
B = FOREACH A GENERATE (bag{tuple(long)}fld;
DESCRIBE B;
B: {{(long)}}
DUMP B;
({(4829090493980522200L)})
({(4893298569862837493L)})
```

104

bytearray Cast to type tuple

- In this example a bytearray (fld in relation A) is cast to type tuple.

```
cat data;
(1,2,3)
(4,2,1)
A = LOAD 'data' AS fld:bytearray;
DESCRIBE A;
a: {fld: bytearray}
DUMP A;
((1,2,3))
((4,2,1))
B = FOREACH A GENERATE (tuple(int,int,float)) fld;
DESCRIBE B;
b: {(int,int,float)}
DUMP B;
((1,2,3))
((4,2,1))
```

105

Relational Operators

Pig offers a number of operators which provide some of the standard functionality found in relational database system.

- GROUP, Groups the data in one or multiple relations
- COGROUP, the same as GROUP but used with multiple relations
- CROSS, Computes the cross product of two or more relations.
- DISTINCT, Removes duplicate tuples in a relation.
- FILTER, Selects tuples from a relation based on some condition (where).
- FOREACH ... GENERATE, Generates data by picking some columns of data
- JOIN (inner), Performs inner, equijoin of two or more relations based on common field values
- JOIN (outer), Performs an outer join of two or more relations based on common field values.
- LIMIT, Limits the number of output tuples.
- LOAD, Loads data from the file system.

106

Relational Operators

Pig offers a number of operators which provide some of the standard functionality found in relational database system.

- GROUP, Groups the data in one or multiple relations
- COGROUP, the same as GROUP but used with multiple relations
- CROSS, Computes the cross product of two or more relations.
- DISTINCT, Removes duplicate tuples in a relation.
- FILTER, Selects tuples from a relation based on some condition (where).
- FOREACH ... GENERATE, Generates data by picking some columns of data
- JOIN (inner), Performs inner, equijoin of two or more relations based on common field values
- JOIN (outer), Performs an outer join of two or more relations based on common field values.
- LIMIT, Limits the number of output tuples.
- LOAD, Loads data from the file system.

107

Relational Operators

- ORDER, Sorts a relation based on one or more fields.
- SAMPLE, Partitions a relation into two or more relations.
- SPLIT, Partitions a relation into two or more relations.
- STORE, Stores or saves results to the file system.
- STREAM, Sends data to an external script or program.
- UNION, Computes the union of two or more relations.

- DESCRIBE, diagnostic operator, returns the schema of an alias
- DUMP, diagnostic operator, dumps or displays results to screen.
- EXPLAIN, diagnostic operator, displays execution plans.
- ILLUSTRATE, displays a step-by-step execution of a sequence of statements.

108

Comparison with SQL

- SQL is high level language that specifies a query execution plan.
- Example: For each sufficiently large category, retrieve the average pagerank of high-pagerank urls in that category.
- Assume there is a table `URLS (url , category, pagerank)`

```
SELECT      category, AVG(pagerank)
FROM  urls
WHERE pagerank > 0.2
GROUP BY   category
HAVING     count(*) > 1000000;
```

109

Pig Latin

- The same problem
 - Example: For each sufficiently large category, retrieve the average pagerank of high-pagerank urls in that category.
 - Assume existence of a relation: `urls` with necessary data: `(url , category, pagerank)`
 - Availability of schema is optional!
 - Columns are referenced using `$0, $1, $2,`
- ```
Good_urls = FILTER urls BY pagerank > 0.2;
Groups = GROUP Good_urls BY category;
Big_groups = FILTER Groups by COUNT(Good_urls) > 1000000;
Output = FOREACH Big_groups GENERATE category,
 AVG(Good_urls, AVG(Good_urls.pagerank));
```

110

## CROSS

- Use the CROSS operator to compute the cross product (Cartesian product) of two or more relations.
- CROSS is an expensive operation and should be used sparingly.
- Suppose we have relations A and B.

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

```
DUMP A;
```

```
(1,2,3)
```

```
(4,2,1)
```

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

```
DUMP B;
```

```
(2,4)
```

```
(8,9)
```

- The cross product of relation A and B is:

```
X = CROSS A, B;
```

```
DUMP X;
```

```
(1,2,3,2,4)
```

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

```
(4,2,1,2,4)
```

```
(4,2,1,8,9)
```

111

## DISTINCT

- Use the DISTINCT operator to remove duplicate tuples in a relation. DISTINCT does not preserve the original order of the contents (to eliminate duplicates, Pig must first sort the data).
- You cannot use DISTINCT on a subset of fields. To do this, use FOREACH ... GENERATE to select the fields, and then use DISTINCT.

```
DUMP A;
```

```
(8,3,4)
```

```
(1,2,3)
```

```
(4,3,3)
```

```
(4,3,3)
```

```
(1,2,3)
```

- All duplicate tuples are removed.

```
X = DISTINCT A;
```

```
DUMP X;
```

```
(1,2,3)
```

```
(4,3,3)
```

```
(8,3,4)
```

112



## FILTER

- Use the FILTER operator to select particular rows of data.

```
A = LOAD 'data' AS (a1:int,a2:int,a3:int);
DUMP A;
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
(8,4,3)
```

- The condition states that if the third field equals 3, then include the tuple with relation X.

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

113

## FOREACH ... GENERATE

### Syntax

```
alias = FOREACH (gen_blk nested_gen_blk) [AS schema];
```

### Terms

|                       |                                                                                                                                                                                                                                                                                                                                                                                                       |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>gen_blk</b>        | FOREACH ... GENERATE used with a relation (outer bag). Use this syntax:<br><code>alias = FOREACH (alias AS (schema) [restriction] [restriction] ... )</code>                                                                                                                                                                                                                                          |
| <b>nested_gen_blk</b> | FOREACH ... GENERATE used with a inner bag. Use this syntax:<br><code>alias = FOREACH (alias AS (schema) alias)</code><br><code>alias = nested_gen_blk (alias = schema.gen_blk ... )</code><br><code>GENERATE expression [, expression ... ]</code><br>The nested block is enclosed in opening and closing brackets ({}).<br>The GENERATE keyword must be the last statement within the nested block. |
| <b>schema.gen_blk</b> | The format of the inner bag                                                                                                                                                                                                                                                                                                                                                                           |
| <b>schema.gen</b>     | Allowed operators are: DISTINCT, FIRST, FIRST, COUNT and SUM.                                                                                                                                                                                                                                                                                                                                         |
|                       | The FOREACH ... GENERATE operation itself is not allowed since would create infinite levels.                                                                                                                                                                                                                                                                                                          |
| <b>AS</b>             | Keyword                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>schema</b>         | A schema using the AS keyword (see Schemast).<br>If the <b>FLATTEN</b> operator is used, enclose the schema in parentheses.<br>If the <b>FLATTEN</b> operator is not used, don't enclose the schema in parentheses.                                                                                                                                                                                   |

114

### FOREACH ... GENERATE, Inner vs. Outer Bag

- FOREACH ...GENERATE works with relations (outer bags) as well as inner bags:
- If A is a relation (outer bag), a FOREACH statement could look like this.

```
X = FOREACH A GENERATE f1;
```

- If A is an inner bag, a FOREACH statement could look like this.

```
X = FOREACH B {
 S = FILTER A BY 'xyz';
 GENERATE COUNT (S.$0);
}
```

115

### Example with Nested Block

- Suppose we have relations A and B. Relation B contains an inner bag.

```
A = LOAD 'data' AS (url:chararray,outlink:chararray);
DUMP A;
(www.ccc.com,www.hjk.com)
(www.ddd.com,www.xyz.org)
(www.aaa.com,www.cvn.org)
(www.www.com,www.kpt.net)
(www.www.com,www.xyz.org)
(www.ddd.com,www.xyz.org)
B = GROUP A BY url;
DUMP B;
(www.aaa.com,{ (www.aaa.com,www.cvn.org) })
(www.ccc.com,{ (www.ccc.com,www.hjk.com) })
(www.ddd.com,{ (www.ddd.com,www.xyz.org), (www.ddd.com,www.xyz.org) })
(www.www.com,{ (www.www.com,www.kpt.net), (www.www.com,www.xyz.org) })
```

116

## FOREACH ... GENERATE in the Inner block

- We perform two of the operations allowed in a nested block, FILTER and DISTINCT.
- The last statement in the nested block must be GENERATE.

```
X = foreach B {
 FA= FILTER A BY outlink == 'www.xyz.org';
 PA = FA.outlink;
 DA = DISTINCT PA;
 GENERATE GROUP, COUNT(DA);
}

DUMP X;
(www.ddd.com, 1L)
(www.www.com, 1L)
```

117

## GROUP or COGROUP

### Syntax

```
dist = GROUP dist [ALL | BY expression], dist ALL | BY expression ... [[USING collected] [PARALLEL n]]
```

### Terms

|                   |                                                                                                                                                                                                                                                                                                                                                                                                     |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>dist</b>       | The name of the relation.                                                                                                                                                                                                                                                                                                                                                                           |
| <b>ALL</b>        | Keyword. Use ALL if you want all fields to go to a single group, for example, when doing <u>aggregates</u> across entire relations.<br>R = GROUP A ALL;                                                                                                                                                                                                                                             |
| <b>BY</b>         | Keyword. Use this clause to group the relation by field, <u>tuple</u> , or expression.<br>R = GROUP A BY M;                                                                                                                                                                                                                                                                                         |
| <b>expression</b> | A <u>valid</u> expression. This is the group key or key field. If the result of the tuple expression is a single field, the key will be the value of the first field rather than a tuple with one field. To group using multiple keys, enclose the keys in parentheses:<br>R = GROUP A BY (key1, key2);                                                                                             |
| <b>USING</b>      | Keyword                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>PARALLEL n</b> | Increase the parallelism of a join by specifying the number of reduce tasks, n.<br>The default value for n is 1 (one reduce task). Note the following:<br>Parallel only affects the number of reduce tasks. Map parallelism is determined by the input file, <u>agg.map</u> for each HDFS block.<br>If you don't specify parallel, you still get the same map parallelism but only one reduce task. |

118

### Example: GROUP

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

- Group relation A on field "age". Relation B has two fields. The first field is named "group" and is type int, the same as field "age" in relation A. The second field is name "A" after relation A and is type bag.

```
B = GROUP A BY age;
DESCRIBE B;
B: {group: int, A: {name: chararray,age: int,gpa: float}}
ILLUSTRATE B;

| B | group: int | A: bag({name: chararray,age: int,gpa: float}) |

| | 18 | {(John, 18, 4.0), (Joe, 18, 3.8)} |
| | 20 | {(Bill, 20, 3.9)} |
```

119

### Example, GROUP continued

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

- As shown in following FOREACH statements, we can refer to the fields in relation B by names "group" and "A" or by positional notation.

```
C = FOREACH B GENERATE group, COUNT(A);
DUMP C;
(18,2L)
(19,1L)
(20,1L)
C = FOREACH B GENERATE $0, $1.name;
DUMP C;
(18,{(John),(Joe)})
(19,{(Mary)})
(20,{(Bill)})
```

120

## COGROUP Example

```
A = LOAD 'data1' AS (owner:chararray,pet:chararray);
DUMP A;
(Alice,turtle) (Alice,goldfish) (Alice,cat) (Bob,dog)
B = LOAD 'data2' AS (friend1:chararray,friend2:chararray);
DUMP B;
(Cindy,Alice) (Mark,Alice) (Paul,Bob)
```

- Tuples are co-grouped using field “owner” from relation A and field “friend2” from relation B as the key fields. R
- Relation X, has three fields, "group", "A" and "B".

```
X = COGROUP A BY owner, B BY friend2;
DESCRIBE X;
X: {group: chararray,A: {owner: chararray,pet: chararray},b:
 {friend1: chararray,friend2: chararray}}
DUMP X;
(Alice,{ (Alice,turtle), (Alice,goldfish), (Alice,cat) },{(Cindy,Alice),
 (Mark,Alice)})
(Bob,{ (Bob,dog) },{(Paul,Bob)})
```

121

## COGROUP Example, continued

- Next, tuples are co-grouped and the INNER keyword is used to ensure that only bags with at least one tuple are returned.

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

- In this following, tuples are co-grouped and the INNER keyword is used asymmetrically on only one of the relations.

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

122

## JOIN (inner)

### Syntax

```
alias = JOIN alias1 BY (expression1[, expression2[, ...]]) [, alias2 BY (expression1
[, expression2[, ...]]) [(USING "implicit") | "skewed" | "merge") [PARALLEL n];
```

### Terms

|            |                                                                                                                                                                                                                                                                                                                                                                                                   |
|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| alias      | The name of a relation.                                                                                                                                                                                                                                                                                                                                                                           |
| BY         | Keyword                                                                                                                                                                                                                                                                                                                                                                                           |
| expression | A field expression.<br>Example: X = JOIN A BY fieldA, B BY fieldB, C BY fieldC;                                                                                                                                                                                                                                                                                                                   |
| USING      | Keyword                                                                                                                                                                                                                                                                                                                                                                                           |
| "implicit" | Use to perform replicated joins (see <a href="#">Replicated Joins</a> ).                                                                                                                                                                                                                                                                                                                          |
| "skewed"   | Use to perform skewed joins (see <a href="#">Skewed Joins</a> ).                                                                                                                                                                                                                                                                                                                                  |
| "merge"    | Use to perform merge joins (see <a href="#">Merge Joins</a> ).                                                                                                                                                                                                                                                                                                                                    |
| PARALLEL n | Increase the parallelism of a job by specifying the number of reduce tasks, n.<br>The default value for this is 1 (one reduce task). Note the following:<br>Parallel only affects the number of reduce tasks. Map parallelism is determined by<br>the input file, one map for each HDFS block.<br>If you don't specify parallel, you still get the same map parallelism but only one reduce task. |

123

## JOIN, A and B joined by their first field

```
A = LOAD 'data1' AS (a1:int,a2:int,a3:int);
DUMP A;
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
B = LOAD 'data2' AS (b1:int,b2:int);
DUMP B;
(2,4)
(8,9)
(1,3)
(2,7)
(2,9)
(4,6)
X = JOIN A BY a1, B BY b1;
DUMP X;
(1,2,3,1,3)
(4,2,1,4,6)
(4,3,3,4,6)
(8,3,4,8,9)
```

124

## JOIN (outer)

### Syntax

```
alias = JOIN left-alias BY left-alias-column [(LEFT | RIGHT | FULL)] [(OUTER)],
right-alias BY right-alias-column [USING "replicated" | "skewed"] PARALLEL n;
```

### Terms

|              |                                                                                                                                                                                                                                                                                                                                                                                           |
|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| alias        | The name of a relation. Applies to alias, left-alias and right-alias.                                                                                                                                                                                                                                                                                                                     |
| alias-column | The name of the join column for the corresponding relation. Applies to left-alias-column and right-alias-column.                                                                                                                                                                                                                                                                          |
| BY           | Keyword                                                                                                                                                                                                                                                                                                                                                                                   |
| LEFT         | Left outer join.                                                                                                                                                                                                                                                                                                                                                                          |
| RIGHT        | Right outer join.                                                                                                                                                                                                                                                                                                                                                                         |
| FULL         | Full outer join.                                                                                                                                                                                                                                                                                                                                                                          |
| OUTER        | (Optional) Keyword                                                                                                                                                                                                                                                                                                                                                                        |
| USING        | Keyword                                                                                                                                                                                                                                                                                                                                                                                   |
| "replicated" | Use to perform replicated joins.<br>Only left outer join is supported for replicated joins.                                                                                                                                                                                                                                                                                               |
| "skewed"     | Use to perform skewed skewed joins.                                                                                                                                                                                                                                                                                                                                                       |
| PARALLEL n   | Increase the parallelism of a job by specifying the number of reduce tasks, n (the default value for n is 1 (one reduce task)). Note the following:<br>Parallel only affects the number of reduce tasks. Map parallelism is determined by the input file, one map for each HDFS block.<br>If you don't specify parallel, you still get the same map parallelism but only one reduce task. |

125

## Examples, Outer JOIN

- This example shows a 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;
```

- This example shows a 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;
```

- This example shows a replicated left outer join.

```
A = LOAD 'large';
B = LOAD 'tiny';
C = JOIN A BY $0 LEFT, B BY $0 USING "replicated";
```

- This example shows a skewed full outer join.

```
A = LOAD 'studenttab' as (name, age, gpa);
B = LOAD 'votertab' as (name, age, registration, contribution);
C = JOIN A BY name FULL, B BY name USING "skewed";
```

126

## ORDER

- **Syntax**

```
alias = ORDER alias BY { * [ASC|DESC] | field_alias [ASC|DESC] [,
 field_alias [ASC|DESC] ...] } [PARALLEL n];
```

- In Pig, relations are unordered (see Relations, Bags, Tuples, and Fields):
- If you order relation A to produce relation X  
`X = ORDER A BY * DESC;`
- relations A and X still contain the same thing.
- If you retrieve the contents of relation X (`DUMP X;`) they are guaranteed to be in the order you specified (descending).
- However, if you further process relation X :  
`Y = FILTER X BY $0 > 1;`
- there is no guarantee that the contents will be processed in the order you originally specified (descending).

127

## Example ORDER

- Suppose we have relation A.

```
A = LOAD 'data' AS (a1:int,a2:int,a3:int);
DUMP A;
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
```

- In this example relation A is sorted by the third field, f3 in descending order. Note that the order of the three tuples ending in 3 can vary.

```
X = ORDER A BY a3 DESC;
DUMP X;
(7,2,5)
(8,3,4)
(1,2,3)
(4,3,3)
(4,2,1)
```

128



## SAMPLE

- Use the SAMPLE operator to select a random data sample with the stated sample size. SAMPLE is a probabilistic operator;
- There is no guarantee that the exact same number of tuples will be returned for a particular sample size each time the operator is used.
  
- In this example relation X will contain 1% of the data in relation A.

```
A = LOAD 'data' AS (f1:int,f2:int,f3:int);
X = SAMPLE A 0.01;
```

129

## SPLIT

**Syntax:** SPLIT alias INTO alias IF expression, alias IF expression [, alias IF expression ...];

- Use the SPLIT operator to partition the contents of a relation into two or more relations based on some expression.
- Depending on the conditions stated in the expression: A tuple may be assigned to more than one or none relation

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

130

## STREAM

- Use the STREAM operator to send data through an external script or program. Multiple stream operators can appear in the same Pig script. The stream operators can be adjacent to each other or have other operations in between.

- When used with a command, a stream statement could look like this:

```
A = LOAD 'data';
B = STREAM A THROUGH `stream.pl -n 5`;
```

- When used with a cmd\_alias, a stream statement could look like this, where cmd is the defined alias.

```
A = LOAD 'data';
DEFINE cmd `stream.pl -n 5`;
B = STREAM A THROUGH cmd;
```

131

## UNION

- Syntax:** alias = UNION alias, alias [, alias ...];
- Use the UNION operator to merge the contents of two or more relations. The UNION operator:
- Does not preserve the order of tuples.
- Does not ensure (as databases do) that all tuples adhere to the same schema or that they have the same number of fields.

```
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)
```

132

## Sum All Columns

- Imagine data

```
(1950, 0.0,1)
(1950, 22.0,1)
(1950,-11.0,1)
(1949,111.0,1)
```

```
A = LOAD 'data' as (year:int, temp:double, qual:int);
```

- We want something that, in SQL, would be done as

```
SELECT sum(qual) from A;
grunt> allData = GROUP A by 1;
grunt> describe allData;
allData: {group: int,A: {year: int,temp: double,qual: int}}
grunt> sumAll = foreach allData generate group, SUM(A.qual);
grunt> describe sumAll;
sumAll: {group: int,long}
grunt> suma = FOREACH sumAll GENERATE $1;
grunt> dump suma;
(4L)
```

133

## Summing with GROUP ALL

- One should be able to do previous sum with command like:
- sumAll = foreach A generate GROUP ALL SUM(qual);

134