APACHE PIG

Presenter by: Le Ngoc Thanh

Outlines

- Introduction to Apache Pig
- Apache Pig Environment
- Pig Latin Language
- User Defined Function
- Apache Pig Engine

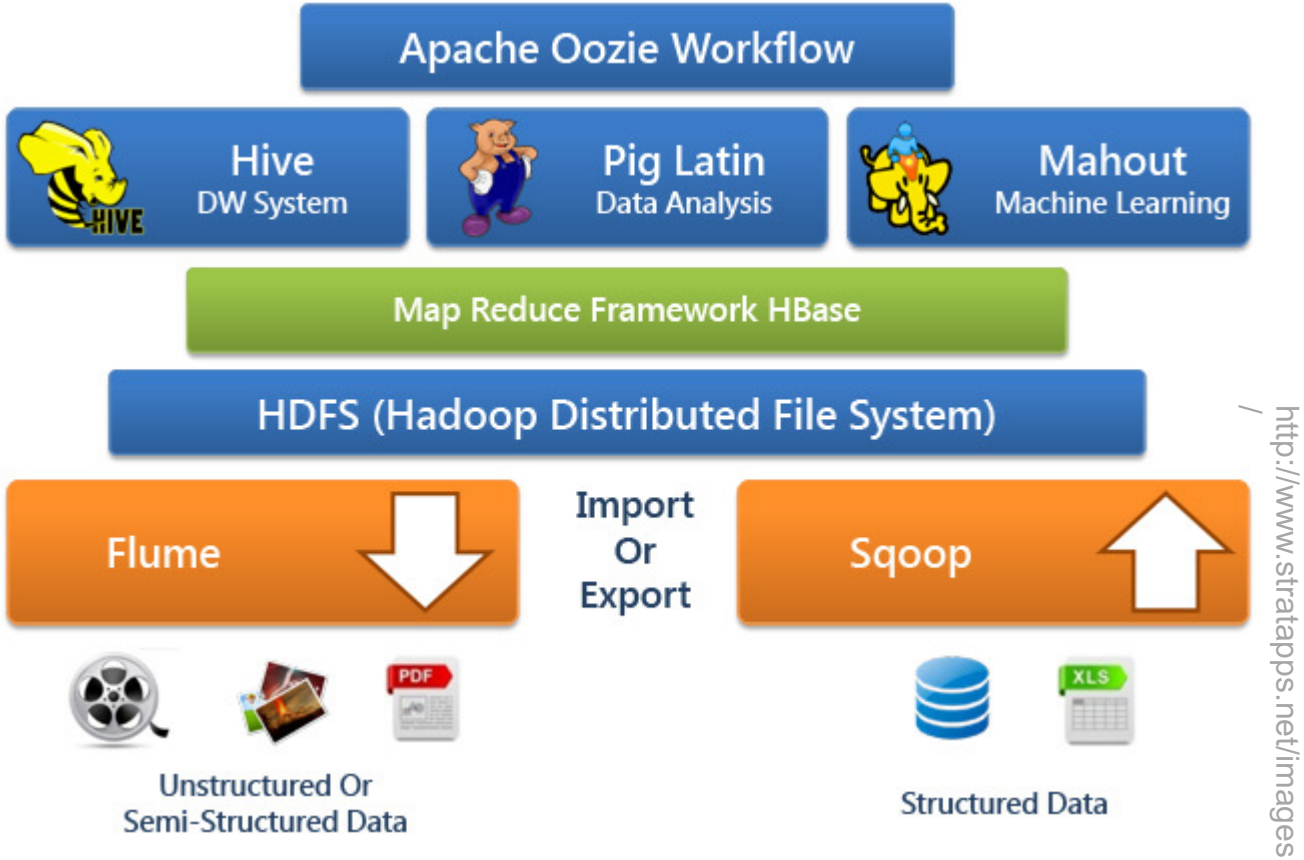
Problems with MapReduce

- MapReduce is a low-level API.
  - Every job in MapReduce consists of three main phases: map, shuffle, and reduce.
- Complex Java codes to perform MapReduce tasks.
- The data processing inside the map and reduce phases is opaque to the system.
  - No opportunity to optimize or check the user's code.

Apache Pig

What is Apache Pig?

- Pig is an open-source high-level dataflow system.
- It provides a simple language for queries and data manipulation Pig



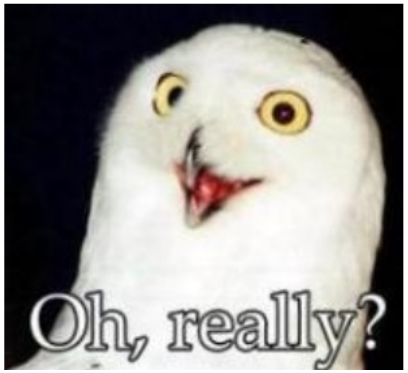
http://www.stratapps.net/images/

What is Apache Pig?

- Latin, that is compiled into map-reduce jobs that are run on Hadoop

✓ Do you know Java?

✓ 10 lines of PIG = 200 lines of Java



+ Built in operations like:

- ✓ Join
- ✓ Group
- ✓ Filter
- ✓ Sort
- ✓ and more…

edureka!

History of Pig

- 2006 – Apache Pig was developed as a research project at Yahoo, especially to create and execute MapReduce jobs on every dataset.
- 2007 – Apache Pig was open sourced via Apache incubator, is member of Apache Software Foundation (ASF)
- 2008 – The first release of Apache Pig came out
- 2010 – Apache Pig graduated as an Apache top-level project
- 90% of Yahoo's MapReduce is done by Pig, 80% of Twitter's MapReduce is also done by Pig and various other companies such as Sales force, LinkedIn, AOL and Nokia also employ Pig

What is Apache Pig?

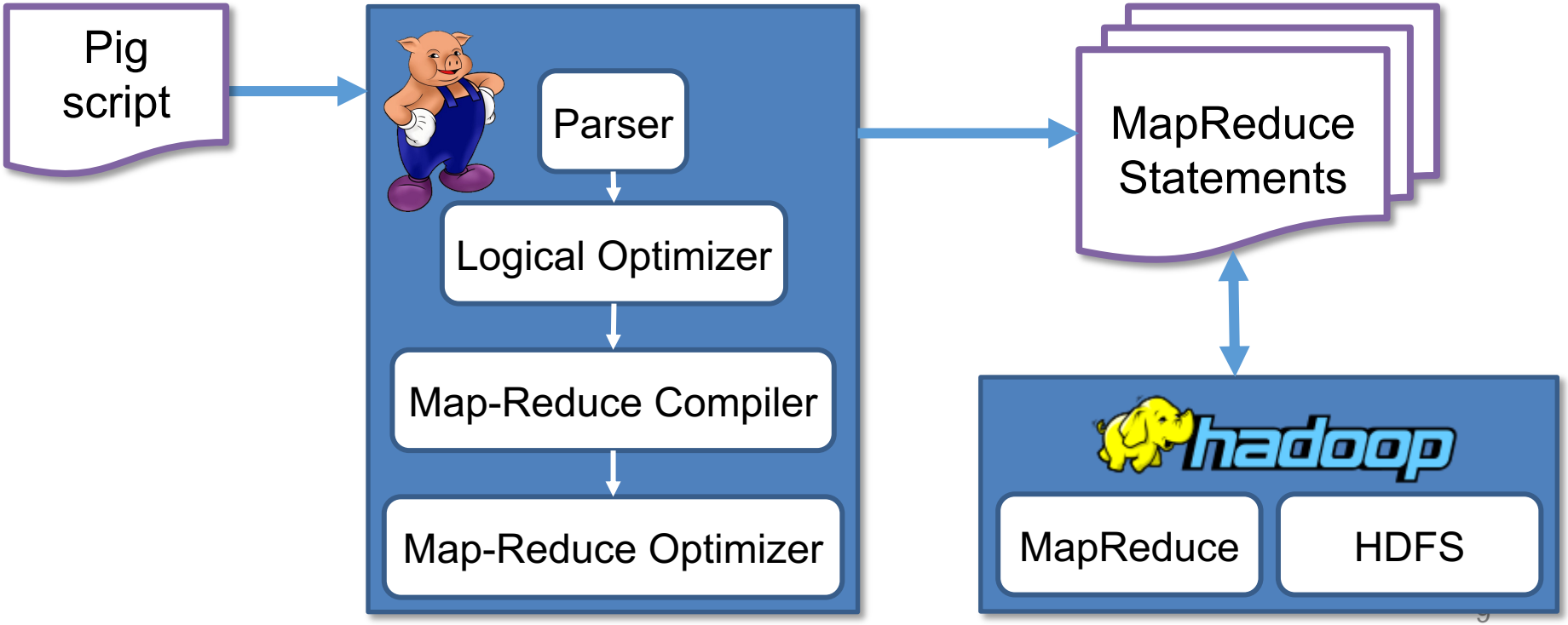
- It includes a language, Pig Latin, for expressing these data flows.

```
input_lines = LOAD '/tmp/my-copy-of-all-pages-on-internet' AS (line:chararray);
words = FOREACH input_lines GENERATE FLATTEN(TOKENIZE(line)) AS word;
filtered_words = FILTER words BY word MATCHES '\\w+';
word_groups = GROUP filtered_words BY word;
word_count = FOREACH word_groups GENERATE COUNT(filtered_words) AS count, group AS word;
ordered_word_count = ORDER word_count BY count DESC;
STORE ordered_word_count INTO '/tmp/number-of-words-on-internet';
```

Word Count in Pig Latin

Behind Apache Pig

- To analyze data using Apache Pig, programmers need to write scripts using Pig Latin language.
- With default settings, all these scripts are internally converted to Map and Reduce tasks.



Why need Apache Pig?

- An ad-hoc way of creating and executing map-reduce jobs on very large data sets.
- Rapid Development
- No Java is required

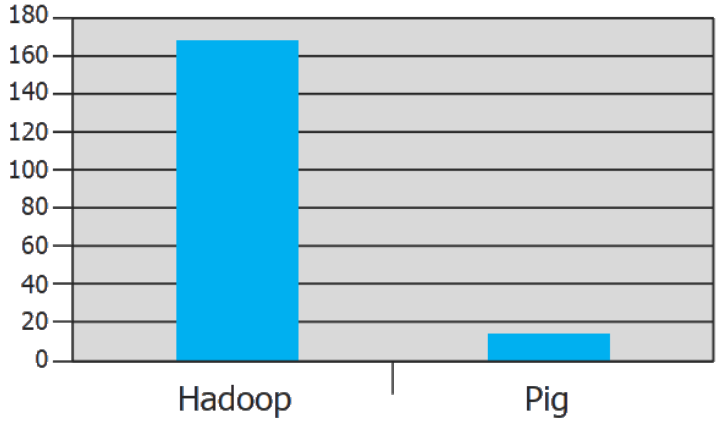
Pig vs MapReduce

Pig	MapReduce
Apache Pig is a data flow language.	MapReduce is a data processing paradigm.
It is a high level language.	MapReduce is low level and rigid.
Performing a Join operation in Apache Pig is pretty simple.	It is quite difficult in MapReduce to perform a Join operation between datasets.
Any novice programmer with a basic knowledge of SQL can work conveniently with Apache Pig.	Exposure to Java is must to work with MapReduce.
Reducing the length of the codes to a great extent.	Require almost 20 times more the number of lines to perform the same task.
No need for compilation. On execution, every Apache Pig operator is converted internally into a MapReduce job.	MapReduce jobs have a long compilation process.

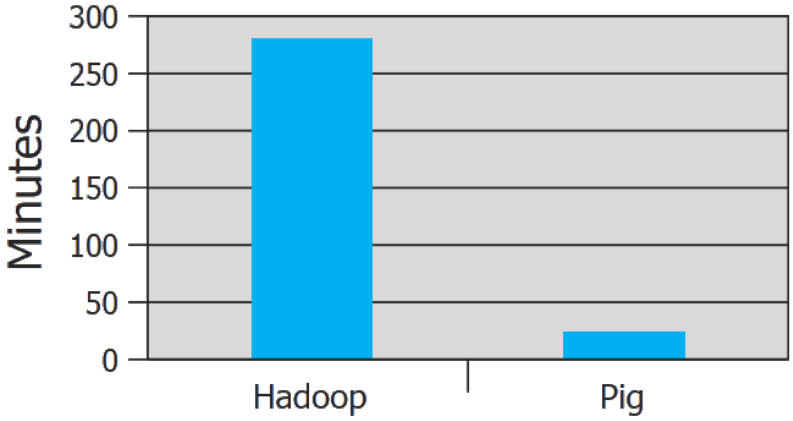
Pig vs MapReduce

- With Pig script, 1/20th the lines of code and 1/16th the development time when compared to writing raw MapReduce.

1/20 the lines of Code



1/16 the development Time



Performance On Par With Raw Hadoop

Pig vs MapReduce

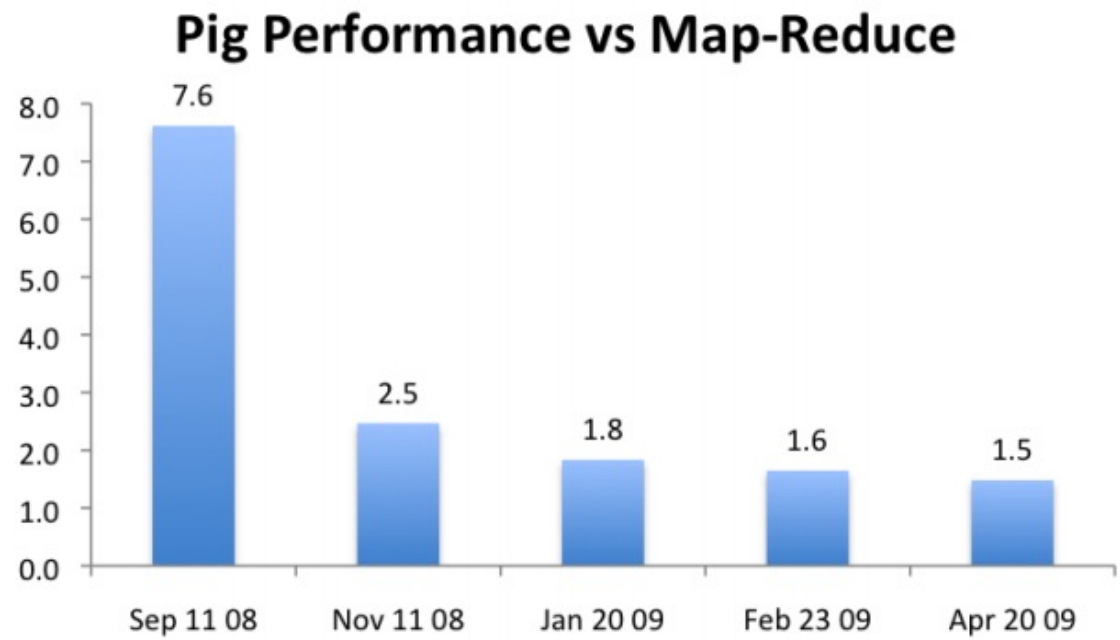
- Java MapReduce: 170 lines of code which takes 4 hours to develop

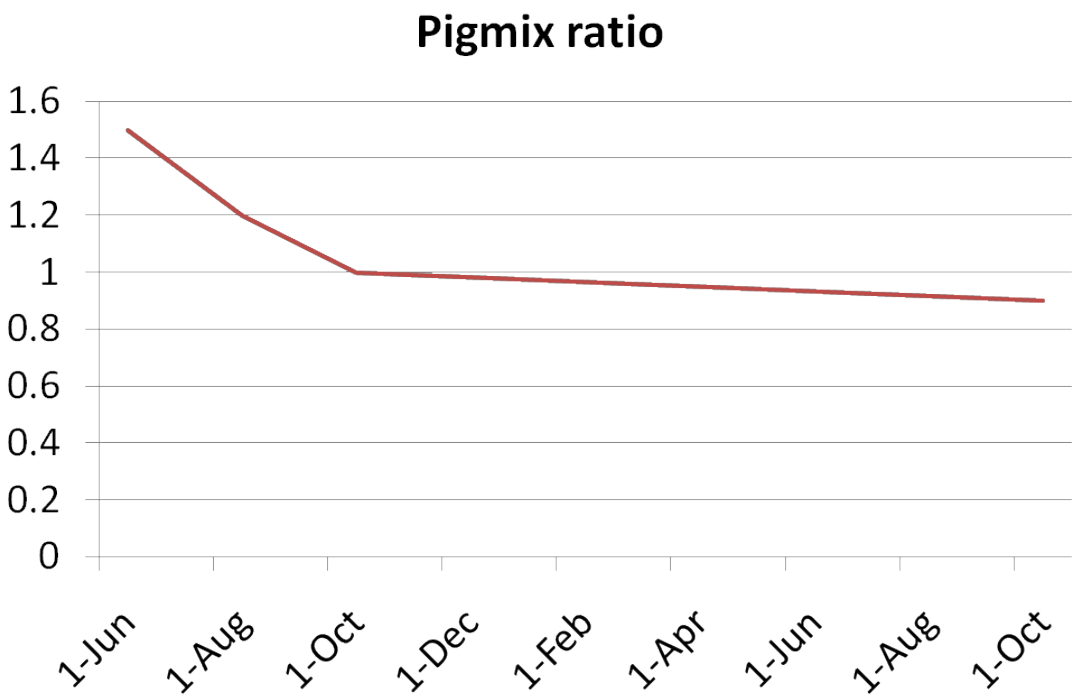
```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.SequenceFileInputFormat;
import org.apache.hadoop.mapred.SequenceFileOutputFormat;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.jobcontrol.Job;
import org.apache.hadoop.mapred.jobcontrol.JobControl;
import org.apache.hadoop.mapred.lib.IdentityMapper;

public class MRExample {
    public static class LoadPages extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String key = line.substring(0, firstComma);
            String value = line.substring(firstComma + 1);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("1" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class LoadAndFilterUsers extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String value = line.substring(firstComma + 1);
            int age = Integer.parseInt(value);
            if (age < 18 || age > 25) return;
            String key = line.substring(0, firstComma);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("2" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class Join extends MapReduceBase
        implements Reducer<Text, Text, Text, Text> {

        public void reduce(Text key,
                Iterator<Text> iter,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // For each value, figure out which file it's from and
store it
            // accordingly.
            List<String> first = new ArrayList<String>();
            List<String> second = new ArrayList<String>();

            while (iter.hasNext()) {
                Text t = iter.next();
                String value = t.toString();
                if (value.charAt(0) == '1')
first.add(value.substring(1));
                else second.add(value.substring(1));
                reporter.setStatus("OK");
            }

            // Do the cross product and collect the values
            for (String s1 : first) {
                for (String s2 : second) {
                    String outval = key + "," + s1 + "," + s2;
                    oc.collect(null, new Text(outval));
                    reporter.setStatus("OK");
                }
            }
        }
    }
    public static class LoadJoined extends MapReduceBase
        implements Mapper<Text, Text, Text, LongWritable> {

        public void map(
                Text k,
                Text val,
                OutputCollector<Text, LongWritable> oc,
                Reporter reporter) throws IOException {
            // Find the url
            String line = val.toString();
            int firstComma = line.indexOf(',');
            int secondComma = line.indexOf(',', firstComma);
            String key = line.substring(firstComma, secondComma);
            // drop the rest of the record, I don't need it anymore,
            // just pass a 1 for the combiner/reducer to sum instead.
            Text outKey = new Text(key);
            oc.collect(outKey, new LongWritable(1L));
        }
    }
    public static class ReduceUrls extends MapReduceBase
        implements Reducer<Text, LongWritable, WritableComparable,
Writable> {

        public void reduce(
                Text key,
                Iterator<LongWritable> iter,
                OutputCollector<WritableComparable, Writable> oc,
                Reporter reporter) throws IOException {
            // Add up all the values we see

            long sum = 0;
            while (iter.hasNext()) {
                sum += iter.next().get();
                reporter.setStatus("OK");
            }

            oc.collect(key, new LongWritable(sum));
        }
    }
    public static class LoadClicks extends MapReduceBase
        implements Mapper<WritableComparable, Writable, LongWritable,
Text> {

        public void map(
                WritableComparable key,
                Writable val,
                OutputCollector<LongWritable, Text> oc,
                Reporter reporter) throws IOException {
            oc.collect((LongWritable)val, (Text)key);
        }
    }
    public static class LimitClicks extends MapReduceBase
        implements Reducer<LongWritable, Text, LongWritable, Text> {

        int count = 0;
        public void reduce(
            LongWritable key,
            Iterator<Text> iter,
            OutputCollector<LongWritable, Text> oc,
            Reporter reporter) throws IOException {

            // Only output the first 100 records
            while (count < 100 && iter.hasNext()) {
                oc.collect(key, iter.next());
                count++;
            }
        }
    }
    public static void main(String[] args) throws IOException {
        JobConf lp = new JobConf(MRExample.class);
        lp.setJobName("Load Pages");
        lp.setInputFormat(TextInputFormat.class);
        lp.setOutputKeyClass(Text.class);
        lp.setOutputValueClass(Text.class);
        lp.setMapperClass(LoadPages.class);
        FileInputFormat.addInputPath(lp, new
Path("/user/gates/pages"));
        FileOutputFormat.setOutputPath(lp,
            new Path("/user/gates/tmp/indexed_pages"));
        lp.setNumReduceTasks(0);
        Job loadPages = new Job(lp);

        JobConf lfu = new JobConf(MRExample.class);
        lfu.setJobName("Load and Filter Users");
        lfu.setInputFormat(TextInputFormat.class);
        lfu.setOutputKeyClass(Text.class);
        lfu.setOutputValueClass(Text.class);
        lfu.setMapperClass(LoadAndFilterUsers.class);
        FileInputFormat.addInputPath(lfu, new
Path("/user/gates/users"));
        FileOutputFormat.setOutputPath(lfu,
            new Path("/user/gates/tmp/filtered_users"));
        lfu.setNumReduceTasks(0);
        Job loadUsers = new Job(lfu);

        JobConf join = new JobConf(MRExample.class);
        join.setJobName("Join Users and Pages");
        join.setInputFormat(KeyValueTextInputFormat.class);
        join.setOutputKeyClass(Text.class);
        join.setOutputValueClass(Text.class);
        join.setMapperClass(IdentityMapper.class);
        join.setReducerClass(Join.class);
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/indexed_pages"));
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/filtered_users"));
        FileOutputFormat.setOutputPath(join, new
Path("/user/gates/tmp/joined"));
        join.setNumReduceTasks(50);
        Job joinJob = new Job(join);
        joinJob.addDependingJob(loadPages);
        joinJob.addDependingJob(loadUsers);

        JobConf group = new JobConf(MRExample.class);
        group.setJobName("Group URLs");
        group.setInputFormat(KeyValueTextInputFormat.class);
        group.setOutputKeyClass(Text.class);
        group.setOutputValueClass(LongWritable.class);
        group.setOutputFormat(SequenceFileOutputFormat.class);
        group.setMapperClass(LoadJoined.class);
        group.setCombinerClass(ReduceUrls.class);
        group.setReducerClass(ReduceUrls.class);
        FileInputFormat.addInputPath(group, new
Path("/user/gates/tmp/joined"));
        FileOutputFormat.setOutputPath(group, new
Path("/user/gates/tmp/grouped"));
        group.setNumReduceTasks(50);
        Job groupJob = new Job(group);
        groupJob.addDependingJob(joinJob);

        JobConf top100 = new JobConf(MRExample.class);
        top100.setJobName("Top 100 sites");
        top100.setInputFormat(SequenceFileInputFormat.class);
        top100.setOutputKeyClass(LongWritable.class);
        top100.setOutputValueClass(Text.class);
        top100.setOutputFormat(SequenceFileOutputFormat.class);
        top100.setMapperClass(LoadClicks.class);
        top100.setCombinerClass(LimitClicks.class);
        top100.setReducerClass(LimitClicks.class);
        FileInputFormat.addInputPath(top100, new
Path("/user/gates/tmp/grouped"));
        FileOutputFormat.setOutputPath(top100, new
Path("/user/gates/top100sitesforusers18to25"));
        top100.setNumReduceTasks(1);
        Job limit = new Job(top100);
        limit.addDependingJob(groupJob);

        JobControl jc = new JobControl("Find top 100 sites for users
18 to 25");
        jc.addJob(loadPages);
        jc.addJob(loadUsers);
        jc.addJob(joinJob);
        jc.addJob(groupJob);
        jc.addJob(limit);
        jc.run();
    }
}
```

Pig vs MapReduce

- Pig Latin: 9 lines of code, written in 15 minutes

```
Users = load 'users' as (name, age);
Fltrd = filter Users by
        age >= 18 and age <= 25;
Pages = load 'pages' as (user, url);
Jnd = join Fltrd by name, Pages by user;
Grpd = group Jnd by url;
Smmd = foreach Grpd generate group,
       COUNT(Jnd) as clicks;
Srtd = order Smmd by clicks desc;
Top5 = limit Srtd 5;
store Top5 into 'top5sites';
```





Pig performance evaluation: https://cwiki.apache.org/confluence/display/PIG/PigMix

Pig vs SQL

Pig	SQL
Pig Latin is a procedural language.	SQL is a declarative language.
In Apache Pig, schema is optional. We can store data without designing a schema (values are stored as 01,02 etc.)	Schema is mandatory in SQL.
The data model in Apache Pig is nested relational.	The data model used in SQL is flat relational.
Apache Pig provides limited opportunity for Query optimization.	There is more opportunity for query optimization in SQL.

Pig vs SQL

- SQL

```
SELECT category, AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > 10^6
```

- Pig Latin

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls) > 10^6;
output = FOREACH big_groups GENERATE
category, AVG(good_urls.pagerank);
```

Pig vs Hive

Pig	Hive
Apache Pig uses a language called Pig Latin.	Hive uses a language called HiveQL.
It was originally created at Yahoo.	It was originally created at Facebook.
Pig Latin is a data flow language.	HiveQL is a query processing language.
Apache Pig can handle structured, unstructured, and semi-structured data.	Hive is mostly for structured data.

Researchers and Programmers

Data Analyst

Features of Pig

- Rich Set of Operators
  - Pig consists of a collection of rich set of operators in order to perform operations such as join, filer, sort and many more.
- Easy programming
  - If you have knowledge of SQL language, then it is very easy to learn Pig Latin language as it is similar to SQL language.
- Optimization opportunities
  - The execution of the task in Apache Pig gets automatically optimized by the task itself, hence the programmers need to only focus on the semantics of the language.

Features of Pig

- Extensibility
  - By using the existing operators, users can easily develop their own functions to read, process, and write data.
- User Define Functions (UDF)
  - With the help of facility provided by Pig of creating UDF's, we can easily create User Defined Functions on a number of programming languages such as Java, Python and **invoke** or **embed** them in Pig Scripts.
- All types of data handling
  - Analysis of all types of Data (i.e. both structured as well as unstructured) is provided by Apache Pig and the results are stored inside HDFS.

When not use Pig?

- Really nasty data formats or completely unstructured data (video, audio, raw human-readable text).
- Pig is definitely slow compared to Map Reduce jobs.
- When you would like more power to optimize your code.

Summary advantages and disadvantages

Advantages	Disadvantages
1. Less development time 2. Easy to learn 3. Procedural language 4. Dataflow 5. Easy to control execution 6. UDFs 7. Lazy evaluation 8. Usage of Hadoop features 9. Effective for unstructured 10. Base Pipeline	1. Errors of Pig 2. Not mature 3. Support 4. Minor one 5. Implicit data schema 6. Delay in execution

Examples Of Data Analysis Task

- Find users who tend to visit “good” pages

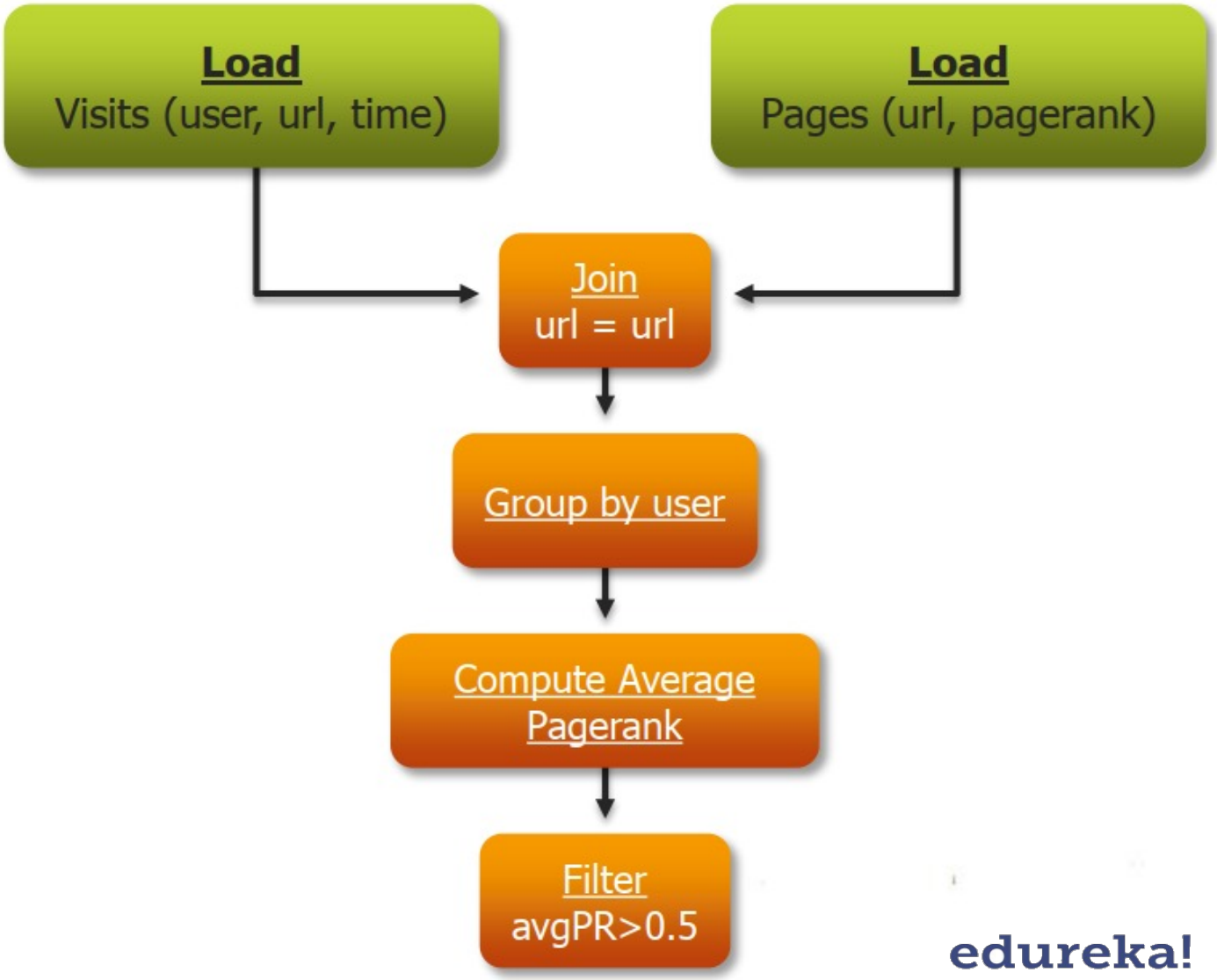
VISITS

User	URL	Time
Amy	www.cnn.com	8:00
Amy	www.crap.com	8:05
Amy	www.myblog.com	10:00
Amy	www.flickr.com	10:05
Fred	cnn.com/index.htm	12:00

PAGES

Page Rank	Page Rank
www.cnn.com	0.9
www.flickr.com	0.9
www.myblog.com	0.7
www.crap.com	0.2

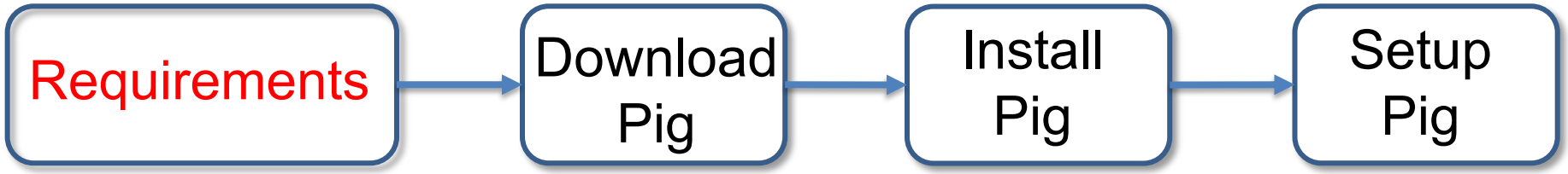
Data Flow



edureka!

Apache Pig Environment

Install Apache Pig

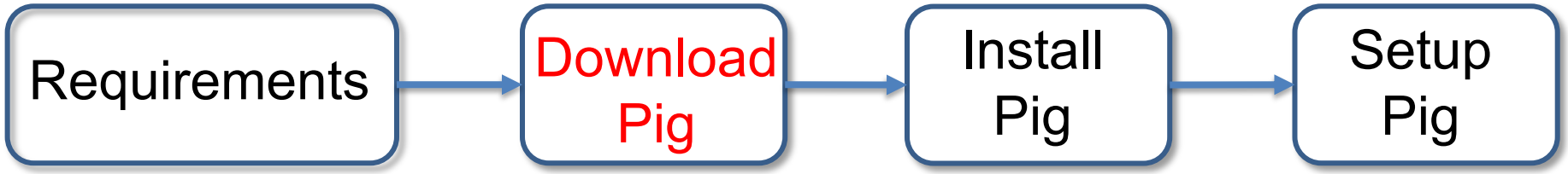


- Install the right version for Hadoop and Java (JDK or openjdk)

```
[centos@localhost ~]$ hadoop version
Hadoop 2.6.4
Subversion https://git-wip-us.apache.org/repos/asf/hadoop.git -r 5082c73637530b0b7e115f9625ed7fac69f937e6
Compiled by jenkins on 2016-02-12T09:45Z
Compiled with protoc 2.5.0
From source with checksum 8dee2286ecdbbbc930a6c87b65cbc010
This command was run using /home/centos/hadoop/share/hadoop/common/hadoop-common-2.6.4.jar
```

```
[centos@localhost ~]$ java -version
java version "1.8.0_73"
Java(TM) SE Runtime Environment (build 1.8.0_73-b02)
Java HotSpot(TM) 64-Bit Server VM (build 25.73-b02, mixed mode)
```

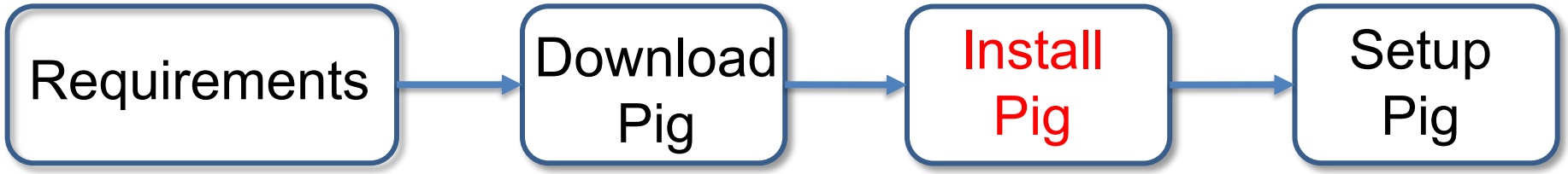
Install Apache Pig



- Download Pig version from https://pig.apache.org/releases.html
- Example: download file pig-0.15.0.tar.gz to install Pig version 0.15.0

- *Note check the compatibility between Pig and Hadoop versions*

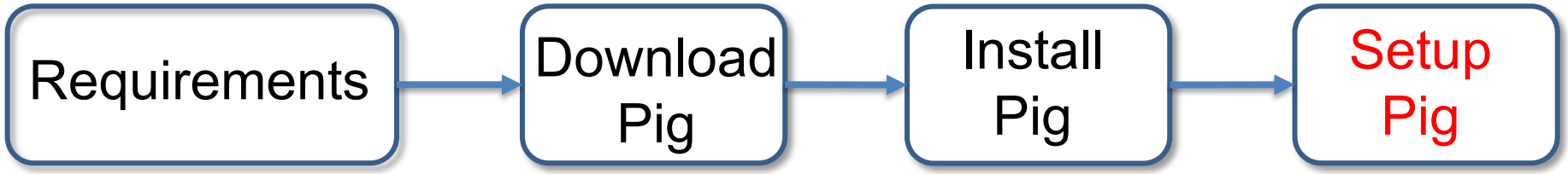
Install Apache Pig



- Choose an appropriate location in the system and create a directory containing the contents of the tar.gz extraction
- For example, let's say Pig is located at /home/centos/hadoop/pig_0.15.0

```
[centos@localhost ~]$ ls hadoop/pig-0.15.0
bin        CHANGES.txt  contrib  ivy      legacy  lib-src  LICENSE.txt  pig-0.15.0-core-h1.jar  README.
build.xml  conf         docs     ivy.xml  lib     license  NOTICE.txt   pig-0.15.0-core-h2.jar  RELEASE
```

Install Apache Pig



- Configure the .bashrc file to set Pig running system parameters

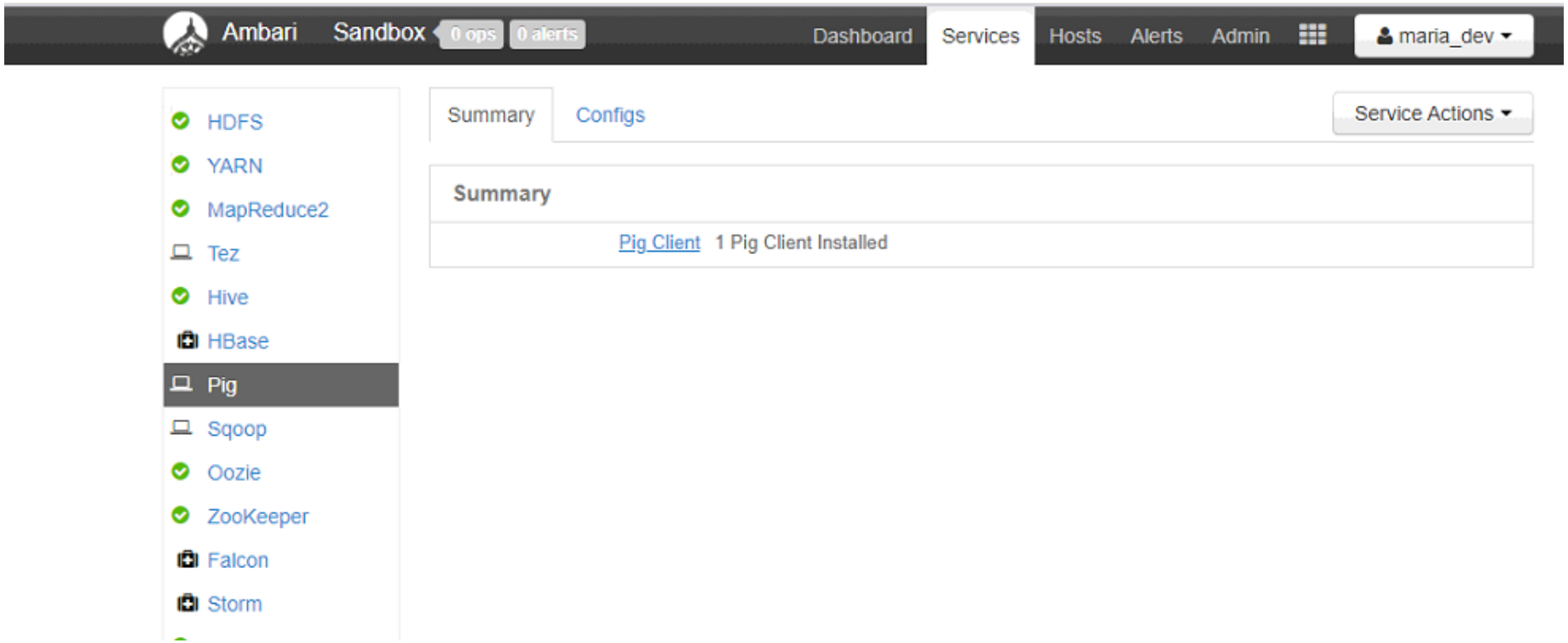
```
# User specific aliases and functions
export HADOOP_HOME=/home/centos/hadoop
export JAVA_HOME=/usr/lib/jvm/jdk
export PIG_HOME=/home/centos/hadoop/pig-0.15.0
export PATH=$PATH:$PIG_HOME/bin:$HADOOP_HOME/bin
export PIG_CLASSPATH=$PIG_HOME/conf
```

- Configure the pig.properties file to set Pig option parameters
- Pig check has been successfully set up?

```
[centos@localhost ~]$ pig -version
Apache Pig version 0.15.0 (r1682971)
compiled Jun 01 2015, 11:44:35
```

Apache Pig with Hortonworks Platform

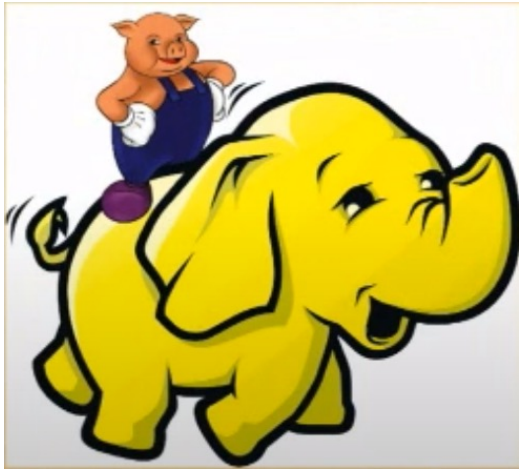
- Check installed Pig in Hortonworks' Ambari



Apache Pig with Hortonworks Platform

- Install Pig in Hortonworks
  - On all the hosts where you execute Pig programs, install the package.
  - For RHEL or CentOS:

    yum install pig
  - The package installs Pig libraries to /usr/hdp/current/pig-client.
    - Pig configuration files are placed in /usr/hdp/current/pig-client/conf.



Run Pig Script

- Apache Pig scripts can be executed in three ways, namely, interactive mode (grunt shell), batch mode, and embedded mode.
  - Script (batch mode):
    - Pig can run a script file that contains Pig commands.
    - Example: *pig script.pig* runs the commands in the local file script.pig.
  - Grunt shell:
    - Grunt is an interactive shell for running Pig commands.
    - It is also possible to run Pig scripts from within Grunt using run and *exec* (execute).
  - Embedded:
    - Embedded can run Pig programs from Java, much like you can use JDBC to run SQL programs from Java.

Run Pig Script

- Script Mode or Batch Mode: In script mode, pig runs the commands specified in a script file.
- The following example shows how to run a pig programs from a script file:

```
[centos@localhost hadoop]$ cat input/sample.pig
/* These are the multi-line comments
   In the pig script */
--we can write single line comments like this.
student = LOAD 'hdfs://localhost:9000/pig_data/student_data.txt' USING PigStorage(',');
STORE student INTO 'hdfs://localhost:9000/pig_output/' USING PigStorage (',');
```

pig -x local sample.pig

Title: ld_with_schema

Pig script: PIG helper

```
e = LOAD '/user/sample_data/employee/employee.csv' USING PigStorage (',')
as (eid:chararray,fname:chararray,lname:chararray,department:chararray);

e_sample = limit e 10;
dump e_sample;
```

Pig View in Hortonworks

Run Pig Script

- Grunt Mode or Interactive Mode: The grunt mode can also be called as interactive mode.
  - Grunt is pig's interactive shell.
- It is started when no file is specified for pig to run

```
hduser@rashmi-dell:~$ pig -x local
16/02/28 22:38:59 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
16/02/28 22:38:59 INFO pig.ExecTypeProvider: Picked LOCAL as the ExecType
2016-02-28 22:38:59,635 [main] INFO  org.apache.pig.Main - Apache Pig versi
01 2015, 11:44:35
2016-02-28 22:38:59,635 [main] INFO  org.apache.pig.Main - Logging error me
9339634.log
2016-02-28 22:38:59,657 [main] INFO  org.apache.pig.impl.util.Utils - Defau
otup not found
2016-02-28 22:39:00,076 [main] INFO  org.apache.hadoop.conf.Configuration.d
precated. Instead, use fs.defaultFS
2016-02-28 22:39:00,077 [main] INFO  org.apache.hadoop.conf.Configuration.d
 deprecated. Instead, use mapreduce.jobtracker.address
2016-02-28 22:39:00,079 [main] INFO  org.apache.pig.backend.hadoop.executic
ing to hadoo
2016-02-28 2
 is deprecat
grunt>
```

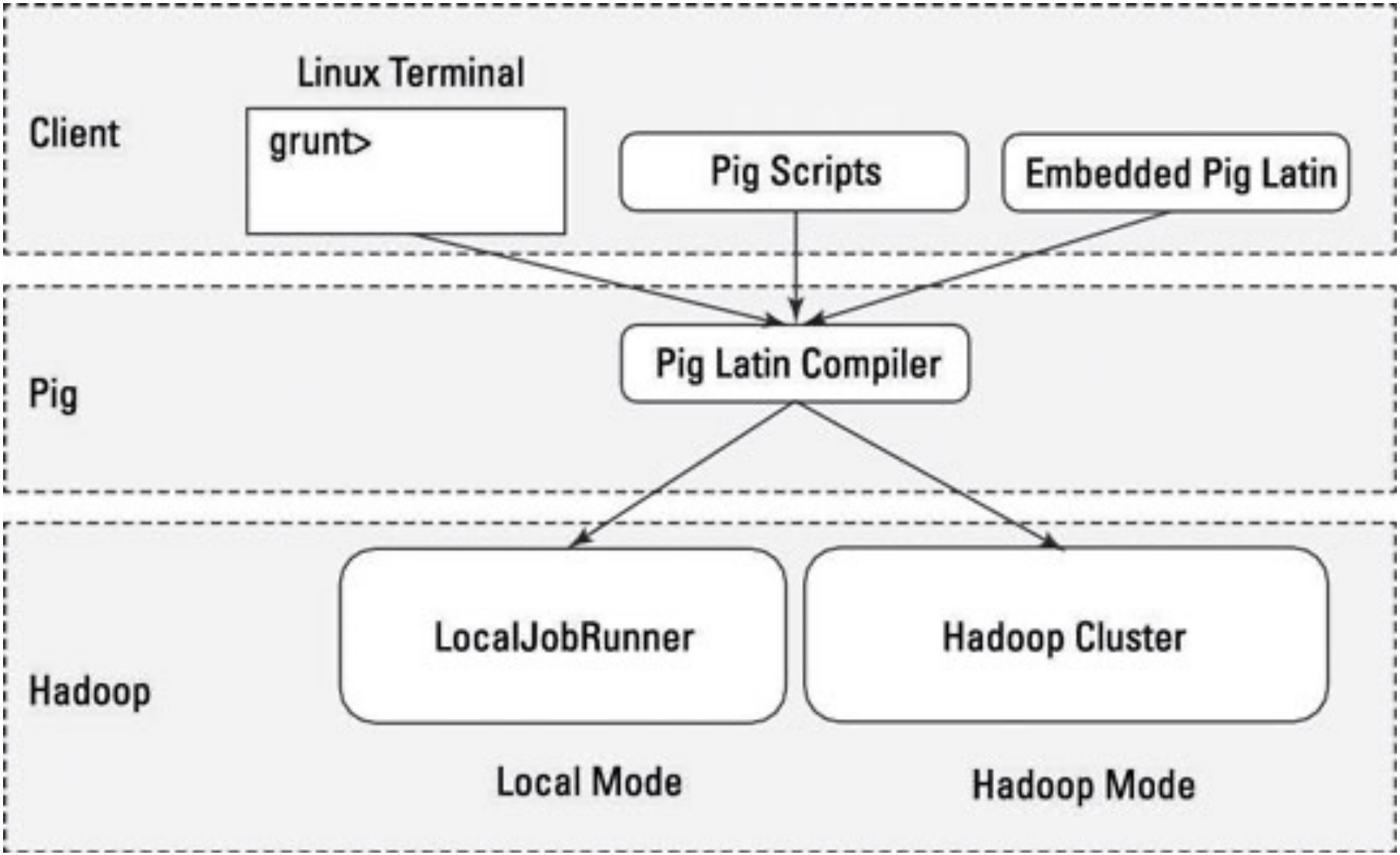
```
$ pig -x local
grunt> A = LOAD 'grunt_file';
grunt> DUMP A;
```

Run Pig Script

- Embeded mode: embed pig programs in Java, python and ruby and can run from the same

```
#!/usr/bin/jython
from org.apache.pig.scripting import *

@outputSchema('greeting:chararray')
def hello(name):
   return 'Hello, %s' % (name)

if __name__ == '__main__':
    P = Pig.compile("""a = LOAD '/names' as (name:chararray);
                       b = FOREACH a GENERATE hello(name);
                       STORE b INTO '/output';""")

result = P.bind().runSingle();
```

Run Pig Script



Run Pig Script

- Local mode: all scripts are run on a local machine without requiring Hadoop MapReduce and HDFS.
  - This can be useful for developing and testing Pig logic
  - Build a simulated mapreduce job running off of a local file on disk. In theory equivalent to MapReduce, but it's not a "real" mr job.

```
pig -x local
```

```
>> [centos@localhost hadoop]$ pig -x local
16/06/05 22:20:03 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
16/06/05 22:20:03 INFO pig.ExecTypeProvider: Picked LOCAL as the ExecType
2016-06-05 22:20:03,792 [main] INFO  org.apache.pig.Main - Apache Pig version 0.15.0 (r1682971) compiled Jun 01 2015, 11:44:35
2016-06-05 22:20:03,801 [main] INFO  org.apache.pig.Main - Logging error messages to: /home/centos/hadoop/pig_1465140003787.log
2016-06-05 22:20:04,859 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop file system at: file:///
grunt>
```

Run Pig Script

- MapReduce mode (also known as Hadoop mode): Pig is executed on the Hadoop cluster.
  - In this case, the Pig Script gets converted into a series of MapReduce jobs that are then run on the Hadoop cluster

```
pig -x mapreduce
```

```
[centos@localhost hadoop]$ pig -x mapreduce
16/06/06 00:39:03 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
16/06/06 00:39:03 INFO pig.ExecTypeProvider: Trying ExecType : MAPREDUCE
16/06/06 00:39:03 INFO pig.ExecTypeProvider: Picked MAPREDUCE as the ExecType
2016-06-06 00:39:03,326 [main] INFO  org.apache.pig.Main - Apache Pig version 0.15.0 (r1682971) compiled Jun 01 2015, 11:44:35
2016-06-06 00:39:03,330 [main] INFO  org.apache.pig.Main - Logging error messages to: /home/centos/hadoop/pig_1465148343317.log
2016-06-06 00:39:05,632 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop file system at: hdfs://localhost:9000
grunt>
```

```
Users = load 'users' as (name, age);
Fltrd = filter Users by
        age >= 18 and age <= 25;
Pages = load 'pages' as (user, url);
Jnd = join Fltrd by name, Pages by user;
Grpd = group Jnd by url;
Smmd = foreach Grpd generate group,
        COUNT(Jnd) as clicks;
Srtd = order Smmd by clicks desc;
Top5 = limit Srtd 5;
store Top5 into 'top5sites';
```

Pig Latin Language

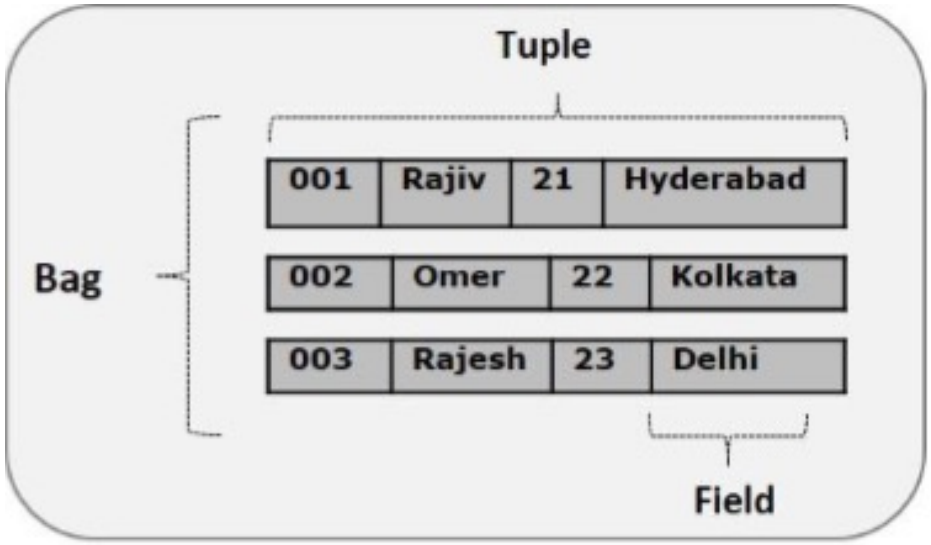
Pig Latin Program

Pig Latin Program

It is made up of a series of operations or transformations that are applied to the input data to produce output.

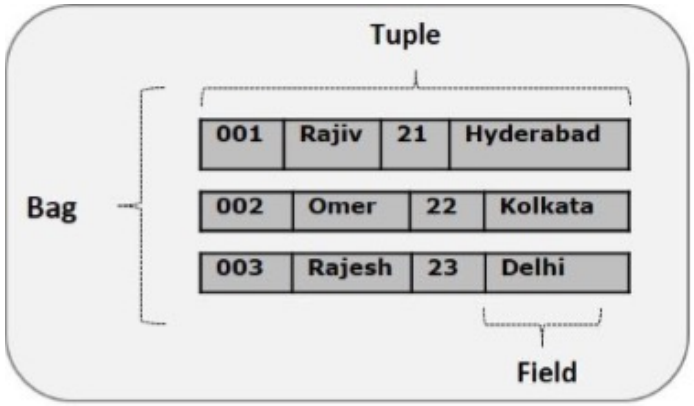
Data Model

- The data model of Pig is fully nested.
- A **Relation** is the outermost structure of the Pig Latin data model.
  - The relations in Pig Latin are unordered (there is no guarantee that tuples are processed in any particular order)

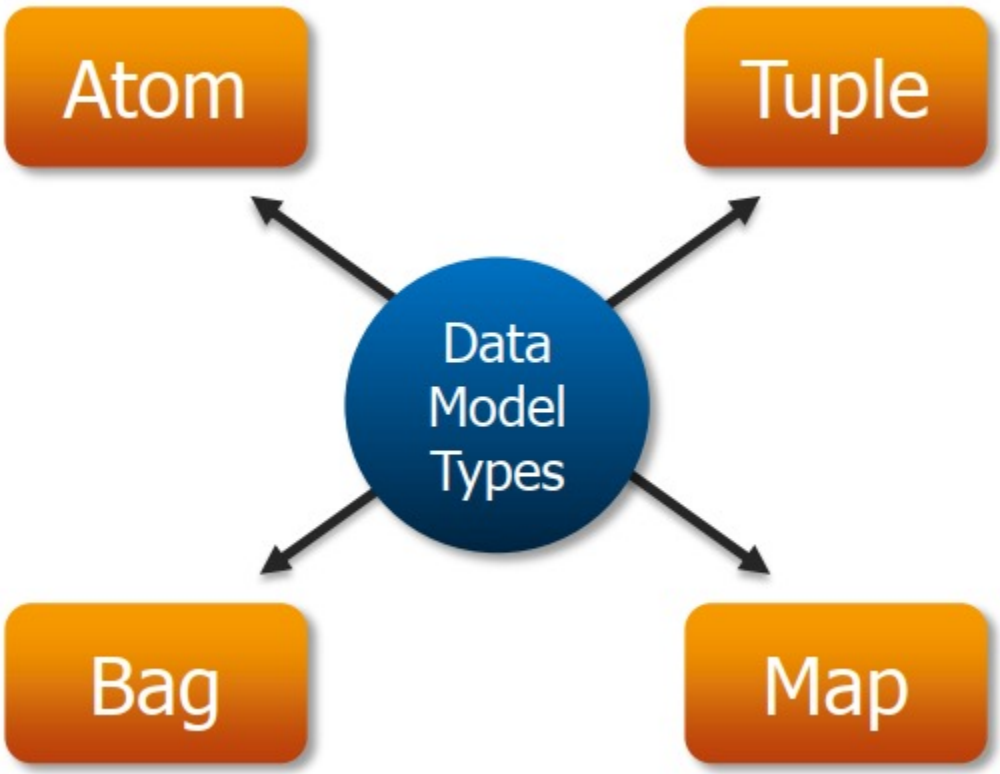


Data Model

- 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, however, 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.



Data Model Types



Data Model Types

- Atom (Field)
  - Any single value in Pig Latin, irrespective of their data, type is known as an Atom.
  - It is stored as string and can be used as string and number.
  - Example: ‘raja’ or ‘30’
- Tuple
  - A record that is formed by an ordered set of fields is known as a tuple, the fields can be of any type.
  - A tuple is similar to a row in a table of RDBMS.
  - Example: (Raja, 30)

Data Model Types

- Bag
  - A bag is an unordered set of tuples.
    - Example: {(Raja, 30), (Mohammad, 45)}
  - A bag can be a **field** in a relation; in that context, it is known as inner bag.
    - Example: {Raja, 30, {9848022338, raja@gmail.com}}
- Map
  - A map (or data map) is a set of key-value pairs.
  - The key needs to be of type chararray and should be unique.
  - The value might be of any type.
  - Example: [name#Raja, age#30]

Simple Types

Simple Types	Description	Example
int	Signed 32-bit integer	10
long	Signed 64-bit integer	Data: 10L or 10l Display: 10L
float	32-bit floating point	Data: 10.5F or 10.5f or 10.5e2f or 10.5E2F Display: 10.5F or 1050.0F
double	64-bit floating point	Data: 10.5 or 10.5e2 or 10.5E2 Display: 10.5 or 1050.0
chararray	Character array (string) in Unicode UTF-8 format	hello world
bytearray	Byte array (blob)	
boolean	boolean	true/false (case insensitive)
datetime	datetime	1970-01-01T00:00:00.000+00:00
biginteger	Java BigInteger	200000000000
bigdecimal	Java BigDecimal	33.456783321323441233442

Special Value - null

- Values for all the above data types can be NULL.
  - Apache Pig treats null values in a similar way as SQL does.
- A null can be an unknown value or a non-existent value. It is used as a placeholder for optional values.
- These nulls can occur naturally or can be the result of an operation.

File – Student

Name	Age	GPA
Joe	18	2.5
Sam		3.0
Angel	21	7.9
John	17	9.0
Joe	19	2.9

In Pig, when a data element is NULL, it means the value is unknown.

Pig → Includes the concept of a data element being… → NULL

Data of any type can be NULL.

Case Sensitivity

- The names (aliases) of relations and fields are case sensitive.
- The names of Pig Latin functions are case sensitive.
- The names of parameters and all other Pig Latin keywords are case insensitive.

```
grunt> A = LOAD 'data' USING PigStorage() AS (f1:int, f2:int, f3:int);
grunt> B = GROUP A BY f1;
grunt> C = FOREACH B GENERATE COUNT ($0);
grunt> DUMP C;
```

- The names (aliases) of relations A, B, and C are case sensitive.
- The names (aliases) of fields f1, f2, and f3 are case sensitive.
- Function names PigStorage and COUNT are case sensitive.
- Keywords LOAD, USING, AS, GROUP, BY, FOREACH, GENERATE, and DUMP are case insensitive. They can also be written as load, using, as, group, by, etc.

Pig Latin: Statement

- Statement is command to work with relation
  - It has structure including expression and schema
  - Every statement ends with a semicolon (;).
  - Example:

```
student = LOAD 'input/student_data.txt' USING PigStorage(',');
```

- Except LOAD and STORE, statement takes input as **a relation** and produces output as **another relation**.
- Lazy Evaluation:
  - When executing LOAD in Grunt Shell, Pig checks the syntax of the statement, but …
  - Only runs MapReduce job to actually load data into the system when **DUMP**, **STORE** is activated.

Arithmetic Operators

Operator	Description	Example
+	**Addition** – Adds values on either side of the operator	a + b will give 30
–	**Subtraction** – Subtracts right hand operand from left hand operand	a – b will give –10
*	**Multiplication** – Multiplies values on either side of the operator	a * b will give 200
/	**Division** – Divides left hand operand by right hand operand	b / a will give 2
%	**Modulus** – Divides left hand operand by right hand operand and returns remainder	b % a will give 0
? :	**Bincond** – Evaluates the Boolean operators. It has three operands as shown below. variable **x** = (expression) ? **value1** *if true* : **value2** *if false*.	b = (a == 1)? 20: 30; if a = 1 the value of b is 20. if a!=1 the value of b is 30.
CASE WHEN THEN ELSE END	**Case** – The case operator is equivalent to nested bincond operator.	CASE f2 % 2 WHEN 0 THEN 'even' WHEN 1 THEN 'odd' END

Boolean Operators

Operator	Symbol	Notes
AND	and	
OR	or	
IN	in	IN operator is equivalent to nested OR operators.
NOT	not	

```
X = FILTER A BY (f1==8) OR (NOT (f2+f3 > f1)) OR (f1 IN (9, 10, 11));
```

Cast Operators

- Cast operators enable you to cast or convert data from one type to another, as long as conversion is supported.
  - Syntax:

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

  - Example:

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

```
B = FOREACH A GENERATE (tuple(int,int,float))fld;
```

Comparison Operators

Operator	Description	Example
==	**Equal** – Checks if the values of two operands are equal or not; if yes, then the condition becomes true.	(a = b) is not true
!=	**Not Equal** – Checks if the values of two operands are equal or not. If the values are not equal, then condition becomes true.	(a != b) is true.
>	**Greater than** – Checks if the value of the left operand is greater than the value of the right operand. If yes, then the condition becomes true.	(a > b) is not true.
<	**Less than** – Checks if the value of the left operand is less than the value of the right operand. If yes, then the condition becomes true.	(a < b) is true.
>=	**Greater than or equal to** – Checks if the value of the left operand is greater than or equal to the value of the right operand. If yes, then the condition becomes true.	(a >= b) is not true.
<=	**Less than or equal to** – Checks if the value of the left operand is less than or equal to the value of the right operand. If yes, then the condition becomes true.	(a <= b) is true.
matches	**Pattern matching** – Checks whether the string in the left-hand side matches with the constant in the right-hand side.	f1 matches '.*tutorial.*'

Type Construction Operators

Operator	Description	Example
()	**Tuple constructor operator** – This operator is used to construct a tuple.	(Raju, 30)
{}	**Bag constructor operator** – This operator is used to construct a bag.	{(Raju, 30), (Mohammad, 45)}
[]	**Map constructor operator** – This operator is used to construct a tuple.	[name#Raja, age#30]

Dereference Operators

Operator	Symbol	Notes
tuple dereference	tuple.id or tuple.(id,...)	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. Note that if the dot operator is applied to a bytearray, the bytearray will be assumed to be a tuple.
bag dereference	bag.id or bag.(id,...)	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 dereference	map#'key'	Map dereferencing must be done by key (field_name#key or $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.

```
A = LOAD 'data' as (f1:int, f2:tuple(t1:int,t2:int,t3:int));
X = FOREACH A GENERATE f2.t1,f2.t3;
```

Disambiguate Operator

- The disambiguate operator is used to identify field names in case there is a ambiguity.

```
A = load 'data1' as (x, y);
B = load 'data2' as (x, y, z);
C = join A by x, B by x;
D = foreach C generate A::y, z;
```

Flatten Operator

- Flatten is used to un-nest a tuple, a bag or a map.
  - For tuples, flatten substitutes the fields of a tuple in place of the tuple.
    - Consider a relation that has a tuple of the form (a, (b, c)).
    - The expression GENERATE $0, flatten($1), will cause that tuple to become (a, b, c).
  - For bags, we create new tuples.
    - ({(b,c),(d,e)}) → GENERATE flatten($0) → (b,c) and (d,e)
    - Sometimes we cause a cross product to happen
      - (a, {(b,c), (d,e)}) → GENERATE $0, flatten($1) → (a, b, c) and (a, d, e)
  - For maps, flatten creates a tuple with two fields containing the key and value.
    - (m[k1#v1, k2#v2]) → GENERATE flatten(kvpair)→(k1,v1) and (k2,v2)
    - When there are additional projections in the expression, a cross product will happen similar to bags.

Relational Operations

- Loading and Storing: LOAD, STORE
- Filtering: STREAM, FOREACH – GENERATE, FILTER, DISTINCT
- Grouping và Joining: JOIN, COGROUP, GROUP, CROSS
- Sorting: ORDER,LIMIT
- Combining và Splitting: UNION, SPLIT
- Diagnostic operators: DUMP, DESCRIBE, EXPLAIN, ILLUSTRATE

Data Processing with PIG

Sample Data (movies_data.txt)

```
1,Dhadakebaz,1986,3.2,7560
2,Dhumdhadaka,1985,3.8,6300
3,Ashi hi banva banvi,1988,4.1,7802
4,Zapatlela,1993,3.7,6022
5,Ayatya Gharat Gharoba,1991,3.4,5420
6,Navra Maza Navsacha,2004,3.9,4904
7,De danadan,1987,3.4,5623
8,Gammat Jammat,1987,3.4,7563
9,Eka peksha ek,1990,3.2,6244
10,Pachhadlela,2004,3.1,6956
```

Input and Output

Load Data

- **$ pig -x local**
- **grunt>** movies = LOAD 'movies_data.txt' USING PigStorage(',') as (id,name,year,rating,duration)
- **grunt>** dump movies;

```
1,Dhadakebaz,1986,3.2,7560
2,Dhumdhadaka,1985,3.8,6300
3,Ashi hi banva banvi,1988,4.1,7802
4,Zapatlela,1993,3.7,6022
5,Ayatya Gharat Gharoba,1991,3.4,5420
6,Navra Maza Navsacha,2004,3.9,4904
7,De danadan,1987,3.4,5623
8,Gammat Jammat,1987,3.4,7563
9,Eka peksha ek,1990,3.2,6244
10,Pachhadlela,2004,3.1,6956
```

Load Data

- Syntax

```
Relation_name = LOAD 'Input path'
        [USING function] [as schema];
```

  - Relation_name: relation name
  - Input path – The name of the file or directory, in single quotes (in local file system or HDFS)
  - function – Select a function from among supported functions: BinStorage, JsonLoader, PigStorage (default), TextLoader.
  - schema – A schema using the AS keyword, enclosed in parentheses

```
(column1 : data type, column2 : data type, column3 : data type);
```

Loader function

- BinStorage: "binary" storage
- PigStorage: loads and stores data that is delimited by something.
  - Support gzip and bzip compression for both read (load) and write (store)
- TextLoader: loads data line by line (delimited by the newline character)
  - support gzip and bzip compression for both read (load) and write (store)
- CSVLoader: Loads CSV files
- XMLLoader: Loads XML files
- JsonLoader, JsonStorage, PigDump, HBaseStorage, AvroStorage, TrevniStorage, AccumuloStorage, OrcStorage
- Can write your own load function (UDF) if data is in a format that cannot be processed by the built in functions.

Data storing

- **STORE** operator will write data to file after Pig parse entire script prior.
- **grunt>** `store movies into 'my_movies';`

Data storing

- Syntax

```
STORE Relation_name
INTO 'Output path' [USING function];
```

  – Output path: specify a directory to save output files (in local file system or HDFS)

- Example:

```
STORE movies INTO 'hdfs://localhost:9000/pig_output/'
USING PigStorage (',');
```

```
Input(s):
Successfully read 6 records (613 bytes) from: "hdfs://localhost:9000/pig_data/student_data.txt"

Output(s):
Successfully stored 6 records (223 bytes) in: "hdfs://localhost:9000/pig_output"

Counters:
Total records written : 6
Total bytes written : 223
Spillable Memory Manager spill count : 0
Total bags proactively spilled: 0
Total records proactively spilled: 0
```

Diagnostic

Diagnostic

- The load statement will simply load the data into the specified relation in Apache Pig.
- To verify the execution of the Load statement, you have to use the Diagnostic Operators
- Pig Latin provides four different types of diagnostic operators
  - Dump operator
  - Describe operator
  - Explanation operator
  - Illustration operator

View data with DUMP

- DUMP operator display output results, will always trigger execution.
- Syntax: DUMP Relation_Name
  - Relation_name: specifies the relation to view the data
- Example: grunt> dump movies;

```
2016-06-06 10:28:43,574 [main] INFO  org.apache.hadoop.yarn.client.RMProxy - Connecting to ResourceManager at /0.0.0.0:8032
2016-06-06 10:28:43,654 [main] INFO  org.apache.hadoop.mapred.ClientServiceDelegate - Application state is completed. FinalApplicationStatus=SUCCEEDED. Redirect
ing to job history server
2016-06-06 10:28:43,879 [main] INFO  org.apache.hadoop.yarn.client.RMProxy - Connecting to ResourceManager at /0.0.0.0:8032
2016-06-06 10:28:43,906 [main] INFO  org.apache.hadoop.mapred.ClientServiceDelegate - Application state is completed. FinalApplicationStatus=SUCCEEDED. Redirect
ing to job history server
2016-06-06 10:28:44,031 [main] INFO  org.apache.hadoop.yarn.client.RMProxy - Connecting to ResourceManager at /0.0.0.0:8032
2016-06-06 10:28:44,048 [main] INFO  org.apache.hadoop.mapred.ClientServiceDelegate - Application state is completed. FinalApplicationStatus=SUCCEEDED. Redirect
ing to job history server
2016-06-06 10:28:44,259 [main] INFO  org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MapReduceLauncher - Success!
2016-06-06 10:28:44,276 [main] INFO  org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2016-06-06 10:28:44,282 [main] INFO  org.apache.pig.data.SchemaTupleBackend - Key [pig.schematuple] was not set... will not generate code.
2016-06-06 10:28:44,386 [main] INFO  org.apache.hadoop.mapreduce.lib.input.FileInputFormat - Total input paths to process : 1
2016-06-06 10:28:44,392 [main] INFO  org.apache.pig.backend.hadoop.executionengine.util.MapRedUtil - Total input paths to process : 1
(1,Rajiv,Reddy,9848022337,Hyderabad)
(2,siddarth,Battacharya,9848022338,Kolkata)
(3,Rajesh,Khanna,9848022339,Delhi)
(4,Preethi,Agarwal,9848022330,Pune)
(5,Trupthi,Mohanthy,9848022336,Bhuwaneshwar)
(6,Archana,Mishra,9848022335,Chennai)
grunt> _
```

View the schema of a relation

- DESCRIBE: prints the schema of the relation at standard output.

```
grunt>
grunt>
grunt> A = load '/data1' as (a1:int, a2:int);
grunt> B = load '/data2' as (b1:int, b2:int);
grunt> X = UNION A, B;
grunt> DESCRIBE X;
X: {a1: int,a2: int}
grunt>
grunt>
```

View execution plans

- Use the EXPLAIN operator to review the logical, physical, and map reduce execution plans that are used to compute the specified relationship.
  - The logical plan shows a pipeline of operators to be executed to build the relation. Type checking and backend-independent optimizations (such as applying filters early on) also apply.

```
grunt> EXPLAIN X;
2013-08-02 03:58:25,292 [main] INFO  org.apache.pig.backend.hadoop.executionengi
ne.HExecutionEngine - pig.usenewlogicalplan is set to true. New logical plan wil
l be used.
#-----------------------------------------------
# Logical Plan:
#-----------------------------------------------
fake: Store 1-20 Schema: {a1: int,a2: int} Type: Unknown
|
|---X: Union 1-17 Schema: {a1: int,a2: int} Type: bag
    |
    |---A: Load 1-15 Schema: {a1: int,a2: int} Type: bag
    |
    |---B: Load 1-16 Schema: {b1: int,b2: int} Type: bag
```

View execution plans

- Use the EXPLAIN operator to review the logical, physical, and map reduce execution plans that are used to compute the specified relationship.
  - The physical plan shows how the logical operators are translated to backend-specific physical operators. Some backend optimizations also apply.

```
#-----------------------------------------------
# Physical Plan:
#-----------------------------------------------
X: Store(fakefile:org.apache.pig.builtin.PigStorage) - scope-17
|
|---X: Union[bag] - scope-16
    |
    |---A: New For Each(false,false)[bag] - scope-7
    |   |   |
    |   |   Cast[int] - scope-2
    |   |   |
    |   |   |---Project[bytearray][0] - scope-1
    |   |   |
    |   |   Cast[int] - scope-5
    |   |   |
    |   |   |---Project[bytearray][1] - scope-4
    |   |
    |   |---A: Load(/data1:org.apache.pig.builtin.PigStorage) - scope-0
    |
    |---B: New For Each(false,false)[bag] - scope-15
        |   |
        |   Cast[int] - scope-10
        |   |
        |   |---Project[bytearray][0] - scope-9
        |   |
        |   Cast[int] - scope-13
        |   |
        |   |---Project[bytearray][1] - scope-12
        |
        |---B: Load(/data2:org.apache.pig.builtin.PigStorage) - scope-8
```

View execution plans

- Use the EXPLAIN operator to review the logical, physical, and map reduce execution plans that are used to compute the specified relationship.
  - The mapreduce plan shows how the physical operators are grouped into map reduce jobs.

```
#-----------------------------------------------
# Map Reduce Plan
#-----------------------------------------------
MapReduce node scope-20
Map Plan
X: Store(fakefile:org.apache.pig.builtin.PigStorage) - scope-17
|
|---X: Union[bag] - scope-16
    |
    |---A: New For Each(false,false)[bag] - scope-7
    |   |   |
    |   |   Cast[int] - scope-2
    |   |   |
    |   |   |---Project[bytearray][0] - scope-1
    |   |   |
    |   |   Cast[int] - scope-5
    |   |   |
    |   |   |---Project[bytearray][1] - scope-4
    |   |
    |   |---A: Load(/data1:org.apache.pig.builtin.PigStorage) - scope-0
    |
    |---B: New For Each(false,false)[bag] - scope-15
        |   |
        |   Cast[int] - scope-10
        |   |
        |   |---Project[bytearray][0] - scope-9
        |   |
        |   Cast[int] - scope-13
        |   |
        |   |---Project[bytearray][1] - scope-12
        |
        |---B: Load(/data2:org.apache.pig.builtin.PigStorage) - scope-8--------
Global sort: false
#--------------------------------------------------
```

Illustrate

- The illustrate operator gives you the step-by-step execution of a sequence of statements.
  - ILLUSTRATE operator is used to review how data is transformed through a sequence of Pig Latin statements.
  - ILLUSTRATE command is your best friend when it comes to debugging a script.

```
ILLUSTRATE Students;

-------------------------------------------------------------------------------
---------------------------------------
| Students    | student_id:long    | name:chararray     | surname:chararray |
gender:chararray     | age:int |
-------------------------------------------------------------------------------
---------------------------------------
|               | 9                  | Princess           | Peach             |
F                    | 21      |
-------------------------------------------------------------------------------
---------------------------------------

grunt>
```

Relation Name

Filtering

FILTER

- Filter: selects tuples from a relation based on some condition.
  - Conversely, to filter out (remove) the data you don't want.
- Syntax:

alias = FILTER alias BY expression;

```
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)

X = FILTER A BY f3 == 3;

DUMP X;
(1,2,3)
(4,3,3)
(8,4,3)

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

DUMP X;
(4,2,1)
(8,3,4)
(7,2,5)
(8,4,3)
```

FILTER

```
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)

X = FILTER A BY f3 == 3;

DUMP X;
(1,2,3)
(4,3,3)
(8,4,3)

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

DUMP X;
(4,2,1)
(8,3,4)
(7,2,5)
(8,4,3)
```

Distinct

- Distinct: removes duplicate tuples in a relation
  - DISTINCT does not preserve the original order of the contents
  - You cannot use DISTINCT on a subset of fields.
- Syntax:

  alias = DISTINCT alias [PARTITION BY partitioner] [PARALLEL n];

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

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

DUMP X;
(1,2,3)
(4,3,3)
(8,3,4)
```

FOREACH

- FOREACH: generates data transformations based on columns of data.
- There are two syntax:

```
alias  = FOREACH block;
alias  = FOREACH nested_block;
```

FOREACH with block

- FOREACH for Projection:

```
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)
```

```
X = FOREACH A GENERATE a1, a2;

DUMP X;
(1,2)
(4,2)
(8,3)
(4,3)
(7,2)
(8,4)
```

FOREACH with block

- FOREACH for Nested Projection: if one of the fields in the input relation is a tuple, bag or map, we can perform a projection on that field (using a deference operator)

```
X = FOREACH C GENERATE group, B.b2;

DUMP X;
(1,{(3)})
(4,{(6),(9)})
(8,{(9)})
```

```
X = FOREACH C GENERATE group, A.(a1, a2);

DUMP X;
(1,{(1,2)})
(4,{(4,2),(4,3)})
(8,{(8,3),(8,4)})
```

FOREACH with block

- FOREACH with defined schema:

```
X = FOREACH A GENERATE a1+a2 AS f1:int;

DESCRIBE X;
x: {f1: int}

DUMP X;
(3)
(6)
(11)
(7)
(9)
(12)
```

FOREACH with block

- FOREACH applying functions:

```
X = FOREACH C GENERATE group, SUM (A.a1);

DUMP X;
(1,1)
(4,8)
(8,16)
```

Grouping & Joining

GROUP

- GROUP: groups the data in one or more relations.
  - Groups together tuples that have the same group key (key field)
  - The first field is named "group" and is the same type as the group key.
  - The second field takes the name of the original relation and is type bag.
- Syntax:

```
alias = GROUP alias { ALL | BY expression} [, alias ALL | BY expression …] [USING 'collected' | 'merge'] [PARTITION BY partitioner] [PARALLEL n];
```

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)
```

```
B = GROUP A BY age;

DESCRIBE B;
B: {group: int, A: {name: chararray,age: int,gpa: float}}

ILLUSTRATE B;
etc ...
-------------------------------------------------------------------------
| 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)}                             |
-------------------------------------------------------------------------

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

GROUP with expression

```
A = LOAD 'data' as (f1:chararray, f2:int, f3:int);

DUMP A;
(r1,1,2)
(r2,2,1)
(r3,2,8)
(r4,4,4)

X = GROUP A BY f2*f3;

DUMP X;
(2,{(r1,1,2),(r2,2,1)})
(16,{(r3,2,8),(r4,4,4)})
```

COGROUP

- The GROUP and COGROUP operators are identical.
  - Both operators work with one or more relations.
  - For readability GROUP is used in statements involving one relation and COGROUP is used in statements involving two or more relations.

COGROUP

```
A = LOAD 'data1' AS (owner:chararray,pet:chararray);

DUMP A;
(Alice,turtle)
(Alice,goldfish)
(Alice,cat)
(Bob,dog)
(Bob,cat)

B = LOAD 'data2' AS (friend1:chararray,friend2:chararray);

DUMP B;
(Cindy,Alice)
(Mark,Alice)
(Paul,Bob)
(Paul,Jane)
```

COGROUP

```
X = COGROUP A BY owner, B BY friend2;

DESCRIBE X;
X: {group: chararray,A: {owner: chararray,pet: chararray},B: {friend1: chararray,friend2:
```

- Tuples are co-grouped using field “owner” from relation A and field “friend2” from relation B as the key fields.
- A tuple is created for each unique key field. The tuple includes the key field and two bags.
  - The first bag is the tuples from the first relation with the matching key field.
  - The second bag is the tuples from the second relation with the matching key field.
  - If no tuples match the key field, the bag is empty.

```
(Alice,{(Alice,turtle),(Alice,goldfish),(Alice,cat)},{(Cindy,Alice),(Mark,Alice)})
(Bob,{(Bob,dog),(Bob,cat)},{(Paul,Bob)})
(Jane,{},{(Paul,Jane)})
```

JOIN (inner join)

- JOIN (inner join) operator to perform an inner, equijoin join of two or more relations based on common field values.
  - The GROUP and JOIN operators perform similar functions. GROUP creates a nested set of output tuples while JOIN creates a flat set of output tuples.
- Syntax:

alias = JOIN alias BY {expression|'('expression [, expression …]')'} (, alias BY {expression|'('expression [, expression …]')'} …) [USING 'replicated' | 'bloom' | 'skewed' | 'merge' | 'merge-sparse'] [PARTITION BY partitioner] [PARALLEL n];

JOIN (inner join)

```
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)
(8,4,3)

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

DUMP B;
(2,4)
(8,9)
(1,3)
(2,7)
(2,9)
(4,6)
(4,9)
```

```
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)
(4,2,1,4,9)
(4,3,3,4,9)
(8,3,4,8,9)
(8,4,3,8,9)
```

JOIN (outer join)

- JOIN (outer join): performs an outer join of two relations based on common field values.
  - Perform left, right, or full outer joins.
- Syntax:

alias = JOIN left-alias BY left-alias-column [LEFT|RIGHT|FULL] [OUTER], right-alias BY right-alias-column [USING 'replicated' | 'bloom' | 'skewed' | 'merge'] [PARTITION BY partitioner] [PARALLEL n];

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;
```

CROSS

- CROSS computes the cross product (Cartesian product) of two or more relations.
  - CROSS is an expensive operation and should be used sparingly.
- Syntax:

alias = CROSS alias, alias [, alias …] [PARTITION BY partitioner] [PARALLEL n];

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

```
X = CROSS A, B;

DUMP X;
(1,2,3,2,4)
(1,2,3,8,9)
(1,2,3,1,3)
(4,2,1,2,4)
(4,2,1,8,9)
(4,2,1,1,3)
```

Combining & Spliting

UNION

- UNION computes the union of two or more relations.
  - Merge the contents of two or more relations.
- Syntax:

alias = UNION [ONSCHEMA] alias, alias [, alias …] [PARALLEL n];

UNION (without SCHEMA)

```
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 A;
(2,4)
(8,9)
(1,3)

X = UNION A, B;

DUMP X;
(1,2,3)
(4,2,1)
(2,4)
(8,9)
(1,3)
```

UNION (with SCHEMA)

- Use the ONSCHEMA clause to base the union on named fields (rather than positional notation).

```
L1 = LOAD 'f1' USING (a : int, b : float);
DUMP L1;
(11,12.0)
(21,22.0)

L2 = LOAD  'f1' USING (a : long, c : chararray);
DUMP L2;
(11,a)
(12,b)
(13,c)

U = UNION ONSCHEMA L1, L2;
DESCRIBE U ;
U : {a : long, b : float, c : chararray}

DUMP U;
(11,12.0,)
(21,22.0,)
(11,,a)
(12,,b)
(13,,c)
```

SPLIT

- SPLIT: partitions a relation into two or more relations
- Syntax:

SPLIT alias INTO alias IF expression, alias IF expression [, alias IF expression …] [, alias OTHERWISE];

```
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)
```

Sorting

ORDER BY

- ORDER BY: sorts a relation based on one or more fields.
- Syntax:

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

```
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)
```

```
X = ORDER A BY a3 DESC;

DUMP X;
(7,2,5)
(8,3,4)
(1,2,3)
(4,3,3)
(8,4,3)
(4,2,1)
```

LIMIT

- LIMIT: limits the number of output tuples.
- Syntax:

alias = LIMIT alias n;

```
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)
```

```
X = LIMIT A 3;

DUMP X;
(1,2,3)
(4,3,3)
(7,2,5)
```

RANK

- RANK returns each tuple with the rank within a relation.
  - When specifying no field to sort on, the RANK operator simply prepends a sequential value to each tuple.
  - Otherwise, the RANK operator uses each field (or set of fields) to sort the relation.
- Syntax:

alias = RANK alias [BY { * [ASC|DESC] | field_alias [ASC|DESC] [, field_alias [ASC|DESC] …] } [DENSE]];

RANK

```
A = load 'data' AS (f1:chararray,f2:int,f3:chararray);

DUMP A;
(David,1,N)
(Tete,2,N)
(Ranjit,3,M)
(Ranjit,3,P)
(David,4,Q)
(David,4,Q)
(Jillian,8,Q)
(JaePak,7,Q)
(Michael,8,T)
(Jillian,8,Q)
(Jose,10,V)
```

```
B = rank A;

dump B;
(1,David,1,N)
(2,Tete,2,N)
(3,Ranjit,3,M)
(4,Ranjit,3,P)
(5,David,4,Q)
(6,David,4,Q)
(7,Jillian,8,Q)
(8,JaePak,7,Q)
(9,Michael,8,T)
(10,Jillian,8,Q)
(11,Jose,10,V)
```

```
C = rank A by f1 DESC, f2 ASC;

dump C;
(1,Tete,2,N)
(2,Ranjit,3,M)
(2,Ranjit,3,P)
(4,Michael,8,T)
(5,Jose,10,V)
(6,Jillian,8,Q)
(6,Jillian,8,Q)
(8,JaePak,7,Q)
(9,David,1,N)
(10,David,4,Q)
(10,David,4,Q)
```

RANK

```
A = load 'data' AS (f1:chararray,f2:int,f3:chararray);

DUMP A;
(David,1,N)
(Tete,2,N)
(Ranjit,3,M)
(Ranjit,3,P)
(David,4,Q)
(David,4,Q)
(Jillian,8,Q)
(JaePak,7,Q)
(Michael,8,T)
(Jillian,8,Q)
(Jose,10,V)
```

```
C = rank A by f1 DESC, f2 ASC DENSE;

dump C;
(1,Tete,2,N)
(2,Ranjit,3,M)
(2,Ranjit,3,P)
(3,Michael,8,T)
(4,Jose,10,V)
(5,Jillian,8,Q)
(5,Jillian,8,Q)
(6,JaePak,7,Q)
(7,David,1,N)
(8,David,4,Q)
(8,David,4,Q)
```

DENSE: no gaps in ranking values.

Built-in Functions

Evaluation Functions

- AVG
- BagToString
- BagToTuple
- Bloom
- CONCAT
- COUNT
- COUNT_STAR
- DIFF
- IsEmpty
- MAX
- MIN
- PluckTuple
- SIZE
- SUBTRACT
- SUM
- IN
- TOKENIZE

Math Functions

- ABS
- ACOS
- ASIN
- ATAN
- CBRT
- CEIL
- COS
- COSH
- EXP
- FLOOR
- LOG
- LOG10
- RANDOM
- ROUND
- ROUND_TO
- SIN
- SINH
- SQRT
- TAN
- TANH

String Functions

- ENDSWITH
- EqualsIgnoreCase
- INDEXOF
- LAST_INDEX_OF
- LCFIRST
- LOWER
- LTRIM
- REGEX_EXTRACT
- REGEX_EXTRACT_ALL
- REGEX_SEARCH
- REPLACE
- RTRIM
- SPRINTF
- STARTSWITH
- STRSPLIT
- STRSPLITTOBAG
- SUBSTRING
- TRIM
- UCFIRST
- UPPER
- UniqueID

Datetime Functions

- AddDuration
- CurrentTime
- DaysBetween
- GetDay
- GetHour
- GetMilliSecond
- GetMinute
- GetMonth
- GetSecond
- GetWeek
- GetWeekYear
- GetYear
- HoursBetween
- MilliSecondsBetween
- MinutesBetween
- MonthsBetween
- SecondsBetween
- SubtractDuration
- ToDate
- ToMilliSeconds
- ToString
- ToUnixTime
- WeeksBetween
- YearsBetween

Tuple, Bag, Map Functions

- TOTUPLE
- TOBAG
- TOMAP
- TOP

User Defined Functions (UDF)

User Defined Functions

- Pig provides extensive support for user defined functions (UDFs) as a way to specify custom processing.
- Pig UDFs can currently be implemented in six languages:
  - Java
  - Jython
  - Python
  - JavaScript
  - Ruby
  - Groovy.

A Program to create UDF in Java

```
public class IsOfAge extends FilterFunc {
    @Override
    public Boolean exec(Tuple tuple) throws IOException {
        if (tuple == null || tuple.size() == 0) {
        return false;
        }

        try {
            Object object = tuple.get(0);
            if (object == null) {
                return false;
            }
            int i = (Integer) object;
            if (i == 18 || i == 19 || i == 21 || i == 23 || i == 27) {
                return true;
            } else {
                return false;
            }
        } catch (ExecException e) {
            throw new IOException(e);
        }
    }
}
```

How to call a UDF?

```
register myudf.jar;
X = filter A by IsOfAge(age);
```

A Program to create UDF in Python

A typical test.py looks like this:

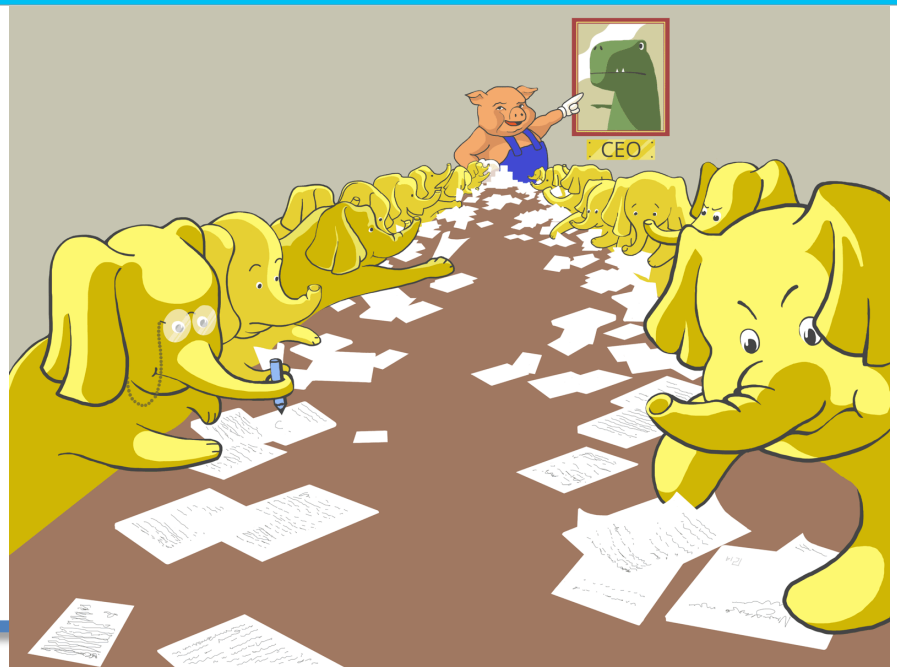
```
from pig_util import outputSchema

@outputSchema("as:int")
def square(num):
    if num == None:
        return None
    return ((num) * (num))

@outputSchema("word:chararray")
def concat(word):
    return word + word
```

You can register a Python script as shown here.

```
Register 'test.py' using streaming_python as myfuncs;
```

The following syntax is also supported, where myfuncs is the namespace created for all the functions inside test.py.

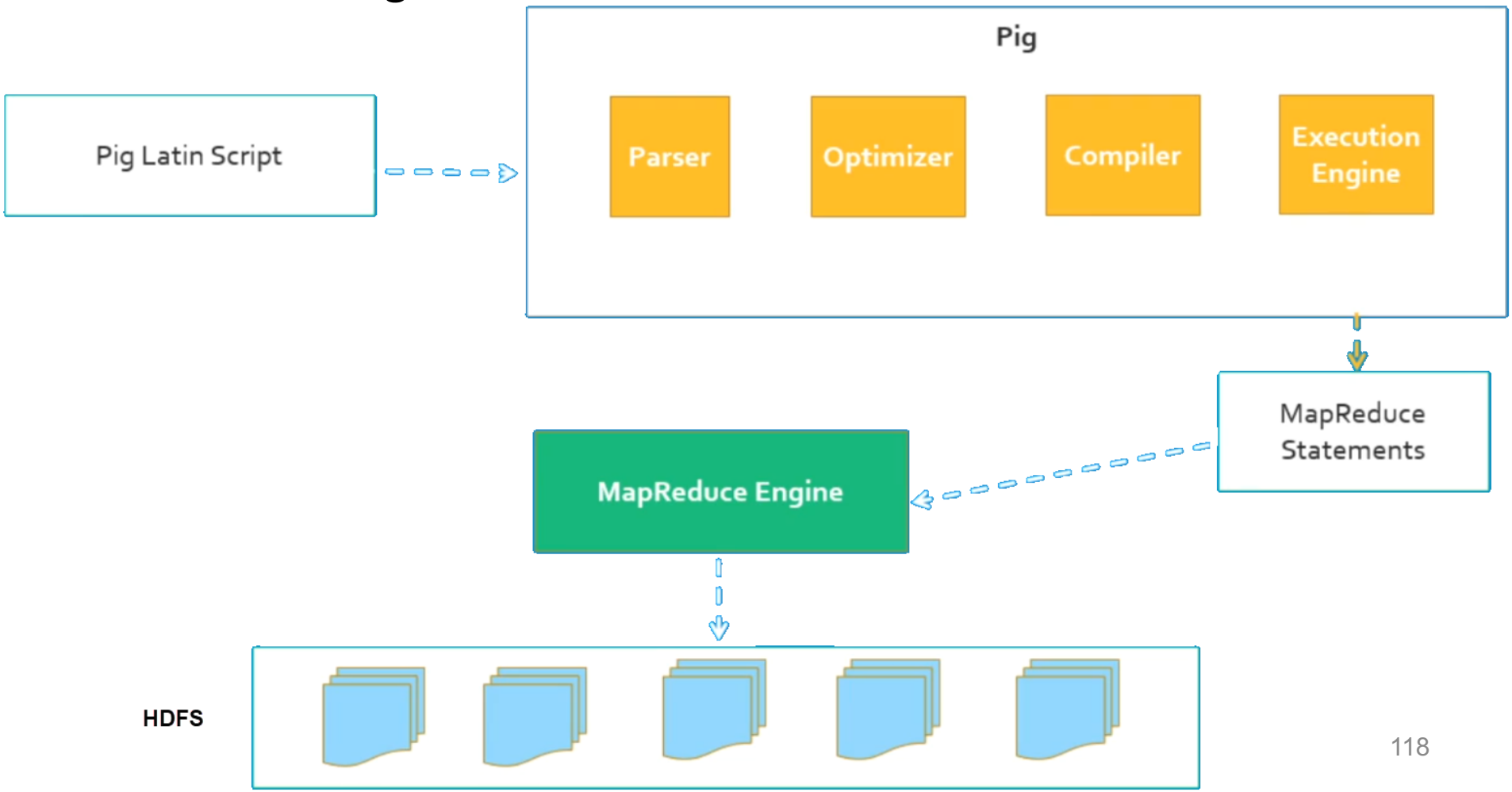
```
register 'test.py' using org.apache.pig.scripting.streaming.python.PythonScriptEngine as myfuncs;
b = foreach a generate myfuncs.concat('hello', 'world'), myfuncs.square(3);
```



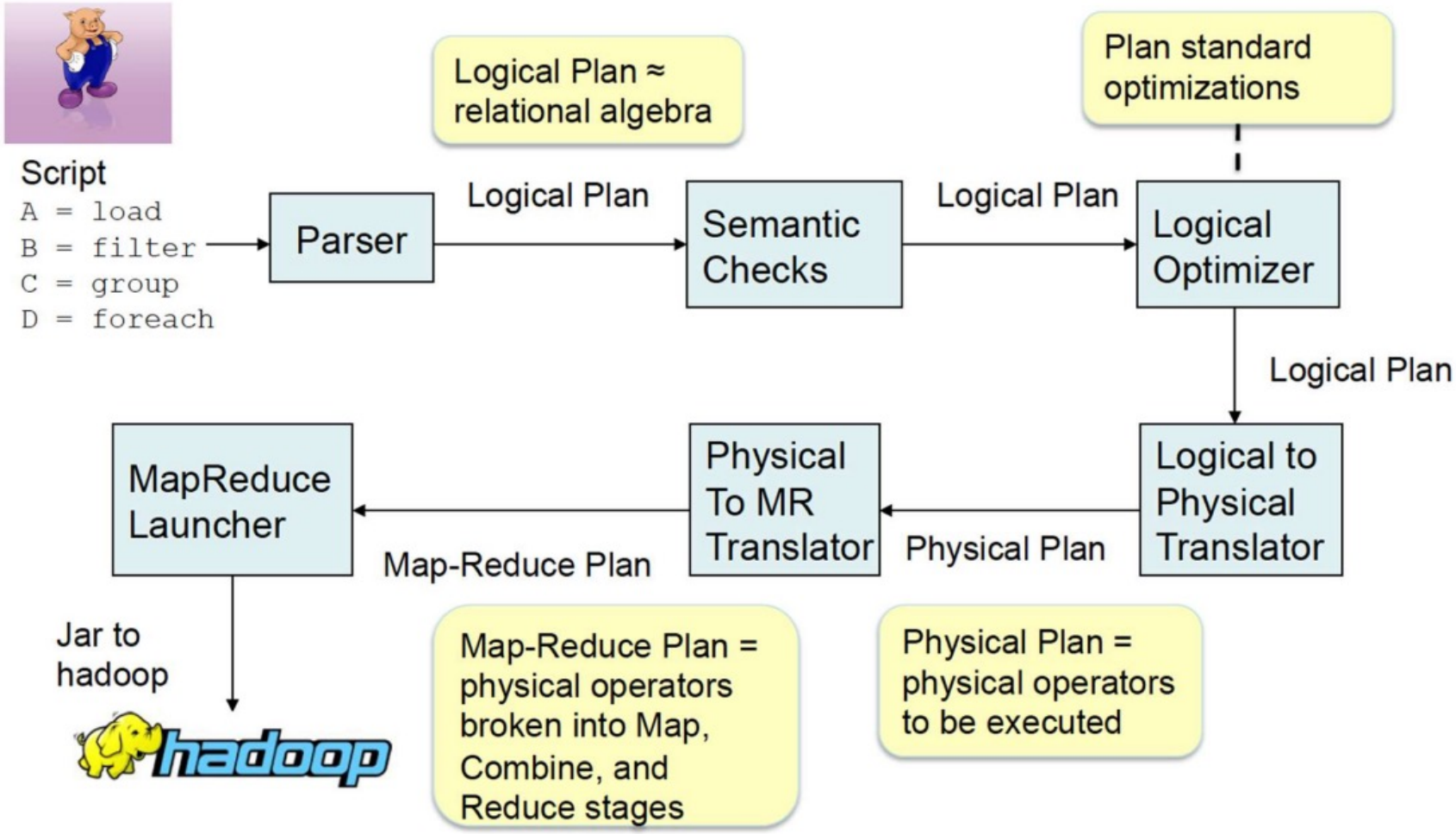
Apache Pig Engine

How Pig Works

- Pig Engine: executing data flows in parallel on Hadoop
- A Pig program needs to go through a series of transformations before being executed



How Pig Works



Parser

- Parser: checks the syntax of the script, does type checking, and other miscellaneous checks.
  - Input: Pig Latin script
  - Output: logical plan
    - Map 1-1 from Pig statement to logical operator
    - Represented as a a DAG (directed acyclic graph).
- Parser check
  - Script correct syntax? Have a reference variable defined?
  - Type checking and schema inference
  - The ability to layers corresponds to UDF
  - Existence of dataflow processing programs referenced by the user's program

Logical Optimizer

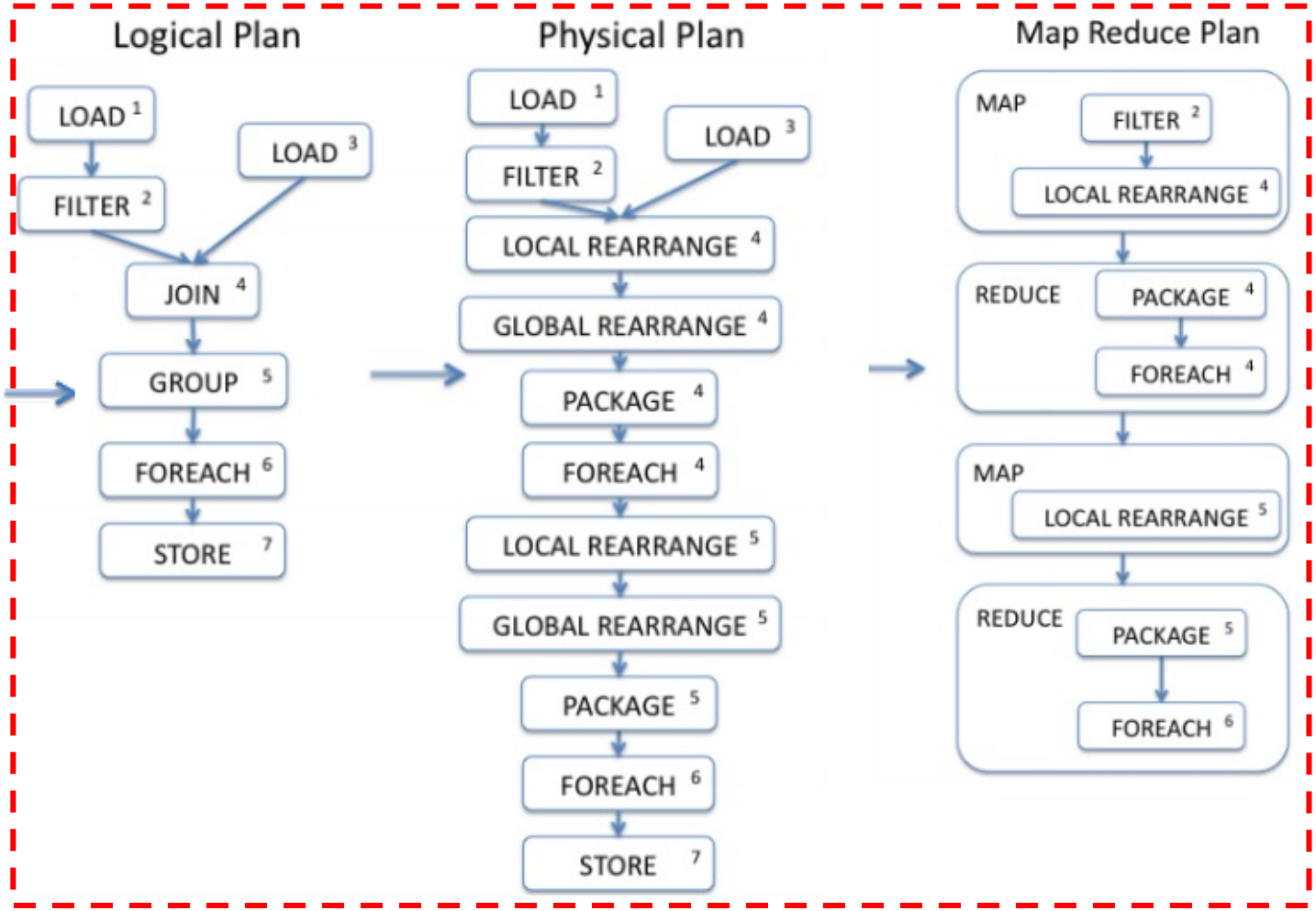
- Optimizer: the logical plan (DAG) is passed to the logical optimizer, which carries out the logical optimizations such as projection pushdown
  - Input: logical plan from Parser
  - Output: optimized logical plan
  - Logical Optimizer
    - E.g: projection pushdown

Compiler

- Compiler: compiles the optimized logical plan into a series of MapReduce jobs

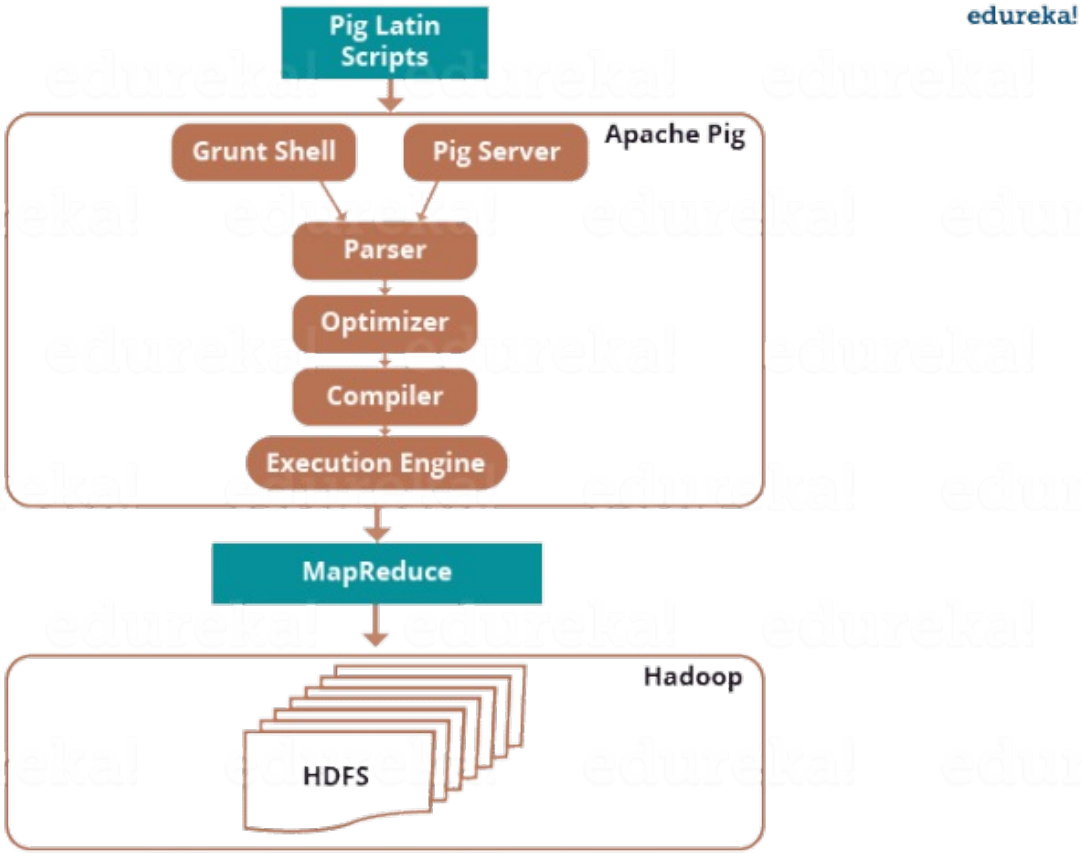
Pig Latin

```
A = LOAD 'file1' AS (x, y, z);
B = LOAD 'file2' AS (t, u, v);
C = FILTER A by y > 0;
D = JOIN C BY x, B BY u;
E = GROUP D BY z;
F = FOREACH E GENERATE
    group, COUNT(D);
STORE F INTO 'output';
```



Execution engine

- Execution engine: finally the MapReduce jobs are submitted to Hadoop in a sorted order.
- These MapReduce jobs are executed on Hadoop producing the desired results.

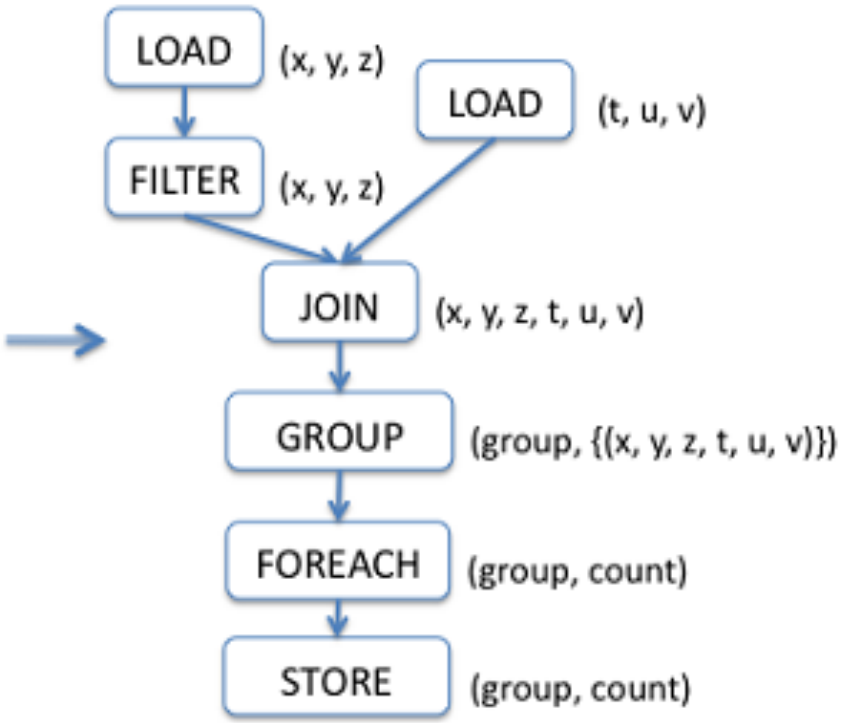


From Pig Latin to MapReduce

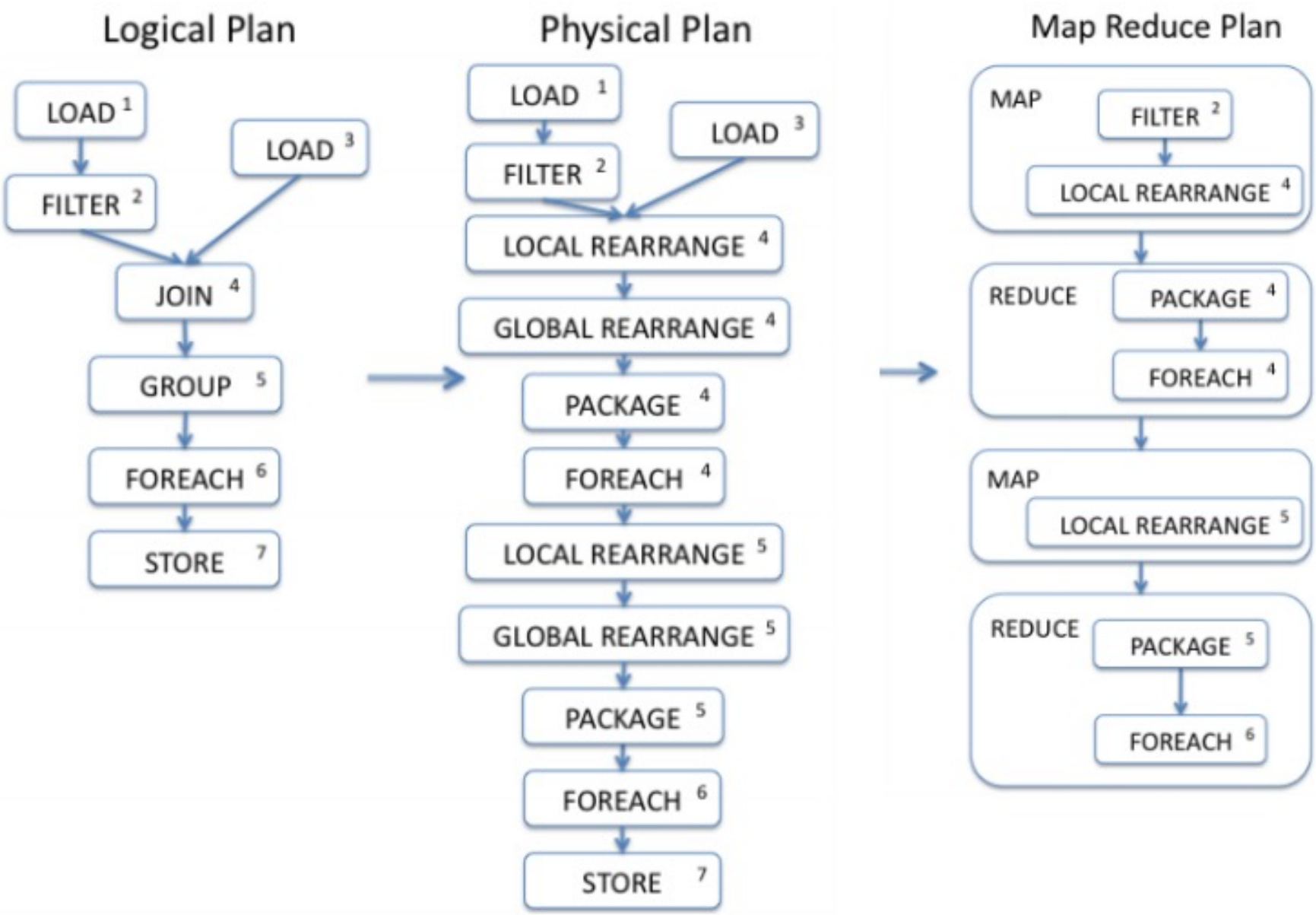
Pig Latin

```
A = LOAD 'file1' AS (x, y, z);
B = LOAD 'file2' AS (t, u, v);
C = FILTER A by y > 0;
D = JOIN C BY x, B BY u;
E = GROUP D BY z;
F = FOREACH E GENERATE
    group, COUNT(D);
STORE F INTO 'output';
```

Logical Plan



From Pig Latin to MapReduce



Compiler: Logical Plan to Physical Plan

- (CO)GROUP becomes a sequence of three physical operators: local rearrange, global rearrange and package
- Rearrange = hashing or sorting by key
- All tuples with the same group-by-key will enter the same processing unit and are adjacent to each other in the data stream
  - When there are multiple input lines, a local rearrange marks each tuple so that it identifies its origin, and then the package places consecutive tuples of the same key into a single-tuple package (including keys and a bag of tuples for each origin stream)

Compiler: Logical Plan to Physical Plan

- JOINs are handled in two ways
  - Rewrite to COGROUP, followed by FOREACH to flattening, creating a parallel hash-join or sort-merge join.
  - Fragment-replicate join: executes entirely at map or reduce phase
- The selection of a JOIN strategy is controlled by the syntax

Compiler: Logical Plan to Physical Plan

- Pig assigns physical operators to Hadoop phases to minimize the number of reduce phases to use.
- In Map-Reduce plan, local rearrange notes the key and stream id for tuples and Hadoop's local sort step does the rest.
- The global rearrange was removed because their logic was implemented in the Hadoop shuffle and merge step.
- The load and store are also removed because the Hadoop framework will take care of the reading and writing of the data.

References

- Tom White. 2009. *Hadoop: The Definitive Guide* (1st ed.). O'Reilly Media, Inc.. Chapter 16: Pig
- Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. 2009. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. Proc. VLDB Endow. 2, 2 (August 2009), 1414-1425. DOI=http://dx.doi.org/10.14778/1687553.1687568
- Pig Tutorial của TutorialsPoint: http://www.tutorialspoint.com/apache_pig/