BeeP – Bigdata Education & Enablement Program

Pig

EBS BI, MSLW & ESG
FY 2013-2014

Topics to be covered

- Introduction to Pig
- Pig Architecture
- Reading and Writing Data
- PigLatin
- Advanced PigLatin
- Debugging Pig
- Pig Best Practices

Introduction to Pig

What is Pig?

- PIG is a top level Apache Software Foundation open source project which runs on top of Hadoop
- Pig was originally created at Yahoo, Pig has data processing capability similar to Hive
- Pig is a platform for analyzing large data sets that consists of a dataflow language. Pig's language is called PigLatin
- PigLatin scripts are turned into MapReduce jobs and executed on the Hadoop cluster
- By Default, Pig reads input files from HDFS, uses HDFS to store intermediate data between MapReduce jobs, and writes its output to HDFS

Pig Installation

- Installation of Pig requires no modification to the cluster

- The Pig interpreter runs on the client machine
  - Turns PigLatin into standard Java MapReduce jobs, which are then submitted to the JobTracker

- There is (currently) no shared metadata, so no need for a shared metastore of any kind unlike Hive.

- Pig can use HCatalog
  - HCatalog is a Table and storage management service for data created using Apache Hadoop
  - Provides a shared schema and data type mechanism

Pig Concepts

- In Pig, a single element of data is an atom
  - Analogous to a field
- A collection of atoms is a tuple
  - Analogous to a row, or partial row, in database terminology
- A collection of tuples is a bag
  - Sometimes known as a relation or result set
- Typically, a PigLatin script starts by loading one or more datasets into bags, and then creates new bags by modifying those it already has

Pig Features

- Pig supports many features which allow developers to perform sophisticated data analysis without having to write Java MapReduce code
  - Joining datasets
  - Grouping data
  - Referring to elements by position rather than name
  - Useful for datasets with many elements
  - Loading non-delimited data using a custom SerDe
  - Creation of user defined functions, written in Java
  - And more…
- Pig can be extended using custom user-defined functions (UDFs)

How is Pig being used?

- Rapid prototyping of algorithms for processing large data sets
- Data processing for web search platforms
- Ad hoc queries across large data sets
- Web log processing

Use Case 1: Time Sensitive Data Loads

- **Challenge:** Loading large volumes of data can become a problem as the Volume of data increases
  - The more data there is, the longer it takes to load. Parallelizing ETL processes can be hard even on one machine

- **Solution:** Pig is built on top of Hadoop, so it's able to scale across multiple servers
  - Easy to process massive data sets
  - ETL processes can be decomposed into manageable chunks
  - Doubling throughput is as easy as doubling number of servers

Use Case 2: Processing Many Data Sources

- **Challenge:** Combining information from multiple sources to gain a deeper understanding of customer behavior
  - i.e. web server traffic, IP geo/location, click through metrics
- **Solution:** Pig can be used with complex data flows and extend them with custom code
  - Creating this rich view of data is possible and easy
  - Pig supplies complex features like joins, sorting, grouping and aggregation
  - Pig's focus on data flow makes it easy to write complex jobs
  - Rather than creating complex logic in SQL, Pig jobs walk through data step by step
  - Easy to rapidly prototype and performance-tune Pig jobs

Use Case 3: Analytic Insight Through Sampling

- **Challenge:** Sampling on large volumes of data can be time consuming

- **Solution:** One of Pig's strengths is its ability to perform sampling of large datasets
  - Easy to reduce the set of data operating on using sampling

  - Sampling with a random distribution of data reduces the amount of data to be analyzed and still delivers meaningful results

Hive v/s Pig

	Hive	Pig
Language	HiveQL(SQL-like)	PigLatin, a data flow language
Schema	Table definitions are stored in a metadata.Metastore available	A schema is optionally defined at runtime. Metastore coming soon
Programmatic access	JDBC	PigServer (Java API)

Many Similarities

- Standard features such as filtering data, joining data sets, grouping and ordering
- Extensibility: Java UDFs and custom scripts
- Custom input and output formats
- Client side shell access

Choosing Between Pig and Hive

- Typically, organizations wanting an abstraction on top of standard MapReduce will choose to use either Hive or Pig
- Which one is chosen depends on the skill-set of the target users
  - Those with an SQL background will naturally gravitate towards Hive
  - Those who do not know SQL will often choose Pig
- Each has strengths and weaknesses; it is worth spending some time investigating each so as to make an informed decision
- Some organizations are now choosing to use both
  - Pig deals better with less structured data, so Pig is used to manipulate the data into a more structured form, then Hive is used to query that structured data

Use cases of PIG

- Pig Latin use cases tend to fall into three separate categories:
  - Traditional extract transform load (ETL) data pipelines
  - Research on raw data
  - Iterative processing
- The largest use case is data pipelines
  - A common example is web companies bringing in logs from their web servers, cleansing the data, and pre-computing common aggregates before loading it into their data warehouse
  - In this case, the data is loaded onto the grid, and then Pig is used to clean out records with corrupt data

Use case of PIG (cont'd.)

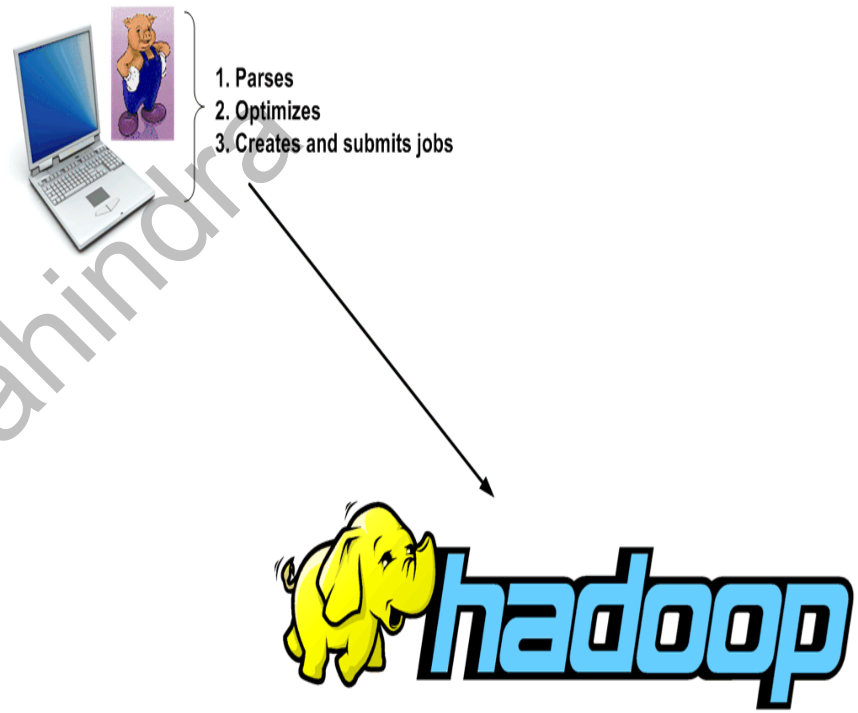
- It is also used to join web event data against user databases so that user cookies can be connected with known user information
- Some users prefer PigLatin because Pig can operate in situations where the schema is unknown, incomplete or inconsistent and because it can easily manage nested data
- Researchers who want to work on data before it has been cleaned and loaded into the warehouse often prefer Pig
- Users who want to build **iterative processing models** are also starting to use Pig

Pig Architecture

Pig Architecture

Pig is a client side application

- Pig resides on the user machine
- Pig translates PigLatin scripts into MapReduce jobs
- Jobs are submitted to the cluster and executed on the cluster
- No special software is installed on the Hadoop cluster
- Pig only needs the location of the JobTracker and NameNode



Pig Architecture (cont'd)

Pig is a dataflow language

- A, B, C, D are executed line by line
- Pig interpreter parses, checks, optimizes and plans execution
- Hadoop runs the mappers and reducers

- Example – PigLatin – Count Job

```
A = LOAD 'myfile' AS
(x,y,z);
B = FILTER A BY x>0;
C = GROUP B BY x;
D = FOREACH C
GENERATE x, COUNT(B);
STORE D INTO
'output';
```

→

- Parses
- Checks
- Optimizes
- Plans Execution
- Submits Jar to Hadoop
- Monitors Job Progress

→

Execution Plan
Map: Filter
Reduce: Counter

Installing & Configuring Pig

- Use RPMs/packages from Cloudera's Distribution including Apache Hadoop (CDH)
  - Or download the tarball and install manually. Latest version of Pig pig-**0.11.1-1** can be downloaded from

- Required software
  - Hadoop
  - Java 6

- **Tell Pig where your cluster** is: in **conf/pig.properties** set
  - fs.default.name=hdfs://namenode_server:8020/
  - mapred.job.tracker=jobtracker_server:8021

Pig's Modes

- Local Mode
  - Uses Hadoop's LocalJobRunner and the local filesystem
  - $ pig -x local
  - Good for debugging on small data sets
- MapReduce Mode
  - Runs the jobs in the Hadoop cluster and reads/writes to HDFS
  - This is the default

```
/* local mode */
$ pig -x local ...
/* mapreduce mode */
$ pig ...
or
$ pig -x mapreduce ...
```

Accessing Pig

- InteractiveMode
  - Grunt, the Pig shell
- Batch Mode
  - Submit a Pig script directly
- PigServer Java class, a JDBC like interface
- PigPen, an Eclipse plugin
  - Allows textual and graphical scripting
  - Sample data and shows example data flow

Accessing Pig

- Interactive mode (typing commands into the grunt shell)
  - Execution is delayed until output is required (i.e. DUMP or STORE)

- Batch mode (Pig script)
  - Entire script is parsed and multiple jobs are combined together if possible
  - Multiple outputs are allowed per Pig script

```
/* id.pig */

A = load 'passwd' using PigStorage(':');  -- load the passwd file
B = foreach A generate $0 as id;          --extract the userIDs
store B into ‘id.out’;                 -- write the results to id.out

$ pig id.pig
```

Using grunt shell

- Starting Grunt

```
$ pig
grunt>
```

- Useful Commands

```
$ pig -help (or -h)
$ pig -version (-i)
$ pig -execute (-e)
$ pig script.pig
```

- Some HDFS commands can be used from within Grunt

```
grunt> ls
```

- For e.g. Commands like **cat, mkdir, rm, cd, pwd, mv, copyFromLocal, copyToLocal** can be used from within Grunt

Using grunt shell (cont'd)

- exec and run
  - **exec** runs the script in a new grunt shell
  - **run** executes within the existing grunt shell

```
grunt> exec script.pig
```

Pig Scripts

- Pig scripts are PigLatin statements/commands in a single file
  - Good practice to identify the file using the *.pig extension

- Pig scripts can be run from the command line and from the Grunt shell
  - run and exec

- Can pass values to parameters using parameter substitution

- Comments are allowed in Pig Scripts
  - For multiline comments use /* …. */
  - For single line comments use //

- Can run scripts that are stored in HDFS
  - The script's full location URI is required

```
$ pig hdfs://nn.mydomain.com:8020/myscripts/script.pig
```

What is Pig Server?

- org.apache.pig.PigServer class
  - Allows Java programs to invoke Pig commands
  - Use local or mapreduce to indicate run mode
  - Use registerQuery() method to add commands

- PigServer is not a daemon server!
  - It is a single-threaded stub to run Pig in a Java application

```
PigServer pigServer = new PigServer( local );
pigServer.registerQuery( data = LOAD 'file' );
pigServer.registerQuery( results = FILTER data BY $0
=='foo' );
pigServer.store( results ,  outfile );
```

Hands-On Exercise

Reading and Writing Data

Reading Various data formats

- PigStorage
  - Loads/stores relations in HDFS using field-delimited text format
- BinStorage
  - Loads/stores relations in HDFS from or to binary files
- BinaryStorage
  - Loads/stores relations in HDFS containing only single-field tuples with a value of type bytearray
- TextLoader
  - Loads relations in HDFS from a plain-text format
  - Loads the whole line as a single column
- PigDump
  - Stores relations to HDFS by writing the toString() representation of tuples, one per line

Example

Here we are loading the data file **u.user** into PIG

```
$ cat u.user
1|24|M|technician|85711
2|53|F|other|94043
3|23|M|writer|32067
4|24|M|technician|43537
5|33|F|other|15213
6|42|M|executive|98101
7|57|M|administrator|91344
8|36|M|administrator|05201
9|29|M|student|01002
10|53|M|lawyer|90703 . . .
```

Example (cont'd)

```
-- Default
-- Assumes data file is tab delimited!
-- Not the right choice for our data file
user = LOAD 'u.user' AS (id, age, gender, occup, zip);

-- PigStorage
user = LOAD 'u.user' USING PigStorage('|')
AS (id, age, gender, occup, zip);
DUMP user;
(1,24,M,technician,85711) . . .

-- TextLoader
-- Loads the entire line as a single column
user = LOAD 'u.user' USING TextLoader();
DUMP user;
(1|24|M|technician|85711) . . .
```

Bags and Aliases

- Each statement defines a new bag
  - Possibly in terms of existing bags
- Each bag is immutable
  - It cannot be changed
- Bags can be given aliases to use later
- In this example
  - user is the alias that references the bag returned by LOAD

```
user = LOAD 'u.user' AS (id, age, gender, occup, zip);
```

Pig Tuples and Bags

- LOAD statements return a bag
  - Each bag has multiple elements
  - Elements (or atoms) can be referenced by position or by name
- In this example
  - Each line of 'u.user' file is a tuple
  - Each id, age, gender, occupation and zipcode field in that line is an atom

```
(1,24,M,technician,85711) . . .
```

- DUMP returns multiple tuples
  - Multiple tuples are referred to as a bag

Bad/Missing data

- Pig substitutes NULL for bad data, e.g. a character in an int field
  - When storing data, NULL is output as empty value
- Find or eliminate all bad records:
  - ...= FILTER records BY field IS NOT NULL;
- Split good from bad:
  - SPLIT records INTO
    good records IF field IS NOT NULL,
    bad records   IF field IS NULL;

Referencing atoms

- Best practice is to define the column names in the LOAD
- Debugging and reading Pig is very difficult otherwise
- By default, LOAD will read the tab delimited fields into non typed aliases $0, $1, $2, etc
- Atoms can then be referenced by those aliases

```
-- Load the u.user data file. Note: column names are not def.
user = LOAD 'u.user' USING PigStorage('|');

females = FILTER user BY $2 =='F';
-- Filter where the third column = 'F`

DUMP females;

(2,53,F,other,94043)
(5,33,F,other,15213)…
```

Defining fields

- Define Column names

```
log = LOAD 'u.user' AS (id, age, gender, occup, zip);
```

- Adding data types

```
log = LOAD 'u.user'
AS (id:int, age:int, gender:chararray,
occup:chararray, zip:int);
```

Pig Fields

Type	Example
Int	42
long	42L
Float	42.0F
Double	42.0
chararray	hello
bytearray	
tuple	(123,dcutting)
bag	{(123,dcutting),(124)}
map	[key#value]

Using Complex/Nested data types

- Complex data types can be loaded from a file
- In this example, ‘file’ contains key/value pairs which are loaded into a map
  - Maps can be enclosed in brackets and use # between the key and value

```
grunt> cat file;
[name#doug,phone#555-555-5555]
[name#tom,phone#555-444-5555]
grunt> data = LOAD 'file' AS (m:map[]);
grunt> DUMP data;
([name#doug,phone#555-555-5555])
([name#tom,phone#555-444-5555])
```

Viewing the Schema

- Use DESCRIBE and/or ILLUSTRATE to view the schema

```
user = LOAD 'u.user' USING PigStorage('|')
AS (id:int, age:int, gender:chararray, occup:chararray, zip:int);
DESCRIBE user;
user: {id:int,age: int,gender: chararray,occup: chararray,zip: int}
ILLUSTRATE user;
------------------------------------------------------------------
| user | id: bytearray | age: bytearray | gender: bytearray | . . .
------------------------------------------------------------------
| | 250 | 29 | M |
------------------------------------------------------------------
------------------------------------------------------------------
| user | id: int | age: int | gender: chararray | occup: . . .
------------------------------------------------------------------
| | 250 | 29 | M | executive|
--------------------------------------------------------
```

The Dump Command

- Writes the contents of a bag to the screen

- In this example, PigStorage loads data from a pipe-delimited file
  - Because the fields are not named they default to type byte array
  - Tab is the default delimiter for a load file
  - The delimiter is specified with the USING qualifier

```
grunt> user = LOAD 'u.user' USING PigStorage('|');
DUMP user;
(1,24,M,technician,85711)
(2,53,F,other,94043)
(3,23,M,writer,32067)
(4,24,M,technician,43537)
(5,33,F,other,15213)
(6,42,M,executive,98101)
(7,57,M,administrator,91344)
(8,36,M,administrator,05201)
(9,29,M,student,01002). . .
```

Storing Results

- STORE
  - Executes PigLatin and saves results to the file system
  - Can specify a field delimiter using PigStorage

- Creates an output directory
  - Each process will write to a file in that directory
  - One file per reducer (or mapper if map-only job)

Store Example

```
grunt> user = LOAD 'u.user'AS (id, age, gender, occup, zip);

grunt> STORE user INTO 'myoutput' USING PigStorage ('#');

$ hadoop fs -ls myoutput
Found 2 items
drwxr-xr-x training /user/training/myoutput/_logs
-rw-r--r-- training /user/training/myoutput/part-m-00000

$ hadoop fs -cat /user/training/myoutput/part-m-00000
1#24#M#technician#85711
2#53#F#other#94043
3#23#M#writer#32067
4#24#M#technician#43537
5#33#F#other#15213
6#42#M#executive#98101
7#57#M#administrator#91344 . . .
```

Hands-On Exercise

Pig Latin

What is PigLatin?

- A data flow language composed of a series of statements
  - Expresses data flows as a series of statements (also called operations or transformations)
  - Can accomplish sophisticated data processing with only a few lines of code
- Each statement operates on a bag, does some transformation and returns a bag
  - It is common to give each resulting bag a new alias

Example: Loading data files

Here we are loading the data file u.data into PIG

```
$ cat u.data
196     242     3       881250949
186     302     3       891717742
22      377     1       878887116
244      51     2       880606923
166     346     1       886397596
298     474     4       884182806
115     265     2       881171488
253     465     5       891628467
305     451     3       886324817
6        86     3       883603013
62      257     2       879372434 . . .
```

Example: Find Highly Rated films

- The u.data file contains 100,000 ratings by 943 users on 1,682 movies
  - The file contains user id, item id, rating and timestamp

- The following script uses the u.data file to find the movies that received an average rating over4.5

```
ratings = LOAD 'u.data' AS (userid:int,
itemid:int, rating:int, timestamp:int);

grpd = GROUP ratings BY itemid;

avg_rating = FOREACH grpd GENERATE
group, AVG(ratings.rating) AS score;

good_movies = FILTER avg_rating BY score>4.5;
```

GROUP BY…

- Used to group records in a bag into groups
- Syntax:

```
bag2 = GROUP bag1 BY expression;
```

- Example

```
grpd = GROUP ratings BY itemid;
dump grpd;

(1656,{(713,1656,2,888882085),(883,1656,5,891692168)})
(1657,{(727,1657,3,883711991)})
(1658,{(782,1658,2,891500230),(894,1658,4,882404137),
(733,1658,3,879535780)})
(1659,{(747,1659,1,888733313)})
(1660,{(747,1660,2,888640731)})
```

GROUP BY Example

```
grunt> cat data;
Doug cat
Tom dog
Mike cat
Sarah fish

grunt> a = LOAD 'data' AS (name:chararray, pet:chararray);

grunt> b = GROUP a BY pet;

grunt> dump b;

(cat,{(Doug,cat),(Mike,cat)})
(dog,{(Tom,dog)})
(fish,{(Sarah,fish)})
```

FOREACH…GENERATE

- FOREACH can do an operation on each record in a bag
  - Iterates through the records in a bag
- Syntax:

```
bag2 = FOREACH bag1 GENERATE expression [,expression ...]
```

- Example

```
avg_rating = FOREACH grpd GENERATE
group, AVG(ratings.rating) AS score;

dump avg_rating;

(1,3.8783185840707963)
(2,3.2061068702290076)
(3,3.0333333333333333)
(4,3.550239234449761) . . .
```

FILTER…BY

- Removes data from a bag if it does not match specified criteria
  - This could be a basic expression or a compound expression that looks at multiple fields
- Syntax:

```
bag2 = FILTER bag1 BY expression;
```

- Example

```
good_movies = FILTER avg_rating BY score > 4.5;
dump good_movies;

(814,5.0)
(1122,5.0)
(1189,5.0)
(1201,5.0) . . .
```

Eliminating Duplicates

- DISTINCT eliminates duplicate records in a bag
  - All fields must be equal to be considered a duplicate
- Syntax:

```
bag2 = DISTINCT bag1;
```

- Example

```
ratings = LOAD 'u.data' AS (userid:int,
itemid:int, rating:int, timestamp:int);
items = FOREACH ratings GENERATE itemid;
uniques = DISTINCT items;
(1)
(128)
(129)
(130) . . ..
```

Using the Grouped results

- FOREACH works for grouped data too
- The grouped bag has a special field named group

```
a = LOAD 'data' AS (name:chararray,pet:chararray);
b = GROUP a BY pet;
c = FOREACH b GENERATE group, COUNT(a);
DUMP c;
(cat,2L)
(dog,1L)
(fish,1L)
```

Implicit field name given to the group key

To refer to a specific field, use a.field

GROUP...ALL

- Use GROUP...ALL to put all records in a single group
- Syntax:

```
bag2 = GROUP bag1 ALL;
```

- Example

```
a = LOAD 'data' AS (name:chararray, pet:chararray);
b = GROUP a ALL;
DUMP b;
(all,{(Doug,cat),(Tom,dog),(Mike,cat),(Sarah,fish)})
c = FOREACH b GENERATE COUNT(a);
DUMP c;
(4).
```

ORDER...BY

- Use ORDER...BY to sort the records in a bag
  - Default order is ascending. Adding DESC sorts descending
- Syntax:

```
bag2 = ORDER bag1 BY field [DESC];
```

- Example

```
b = ORDER a BY pet;
(cat)
(cat)
(dog)
(fish)
```

Watch-out for non-typed data

```
grunt> cat data
42
100
3
grunt> a = LOAD 'data';
grunt> b = ORDER a BY $0;
grunt> DUMP b;
(100)
(3)
(42)
```

bytearrays order by byte order, not numerically

LIMIT

- Use LIMIT to reduce the number of output records
  - Unless an ORDER BY is also specified, the records returned from a LIMIT are random and may change from one execution to another

- Syntax:

```
bag2 = LIMIT bag1 n;
```

- Example

```
b = LIMIT a 10;
```

Top-N Queries

- Use ORDER BY and LIMIT together for top n results
- Example

```
ordered = ORDER items BY cost DESC;
expensive = LIMIT ordered 10;
```

Nested Ordering

- ORDER BY can be applied within each group
- This example sorts by descending order of name within each group:

```
a = LOAD 'data.txt' AS (name:chararray,pet:chararray);
b = GROUP a BY pet;
c = FOREACH b {
ordered = ORDER a BY name DESC;
GENERATE group, ordered;
}
```

Nested Ordering

- ORDER BY can be applied within each group
- Example

```
a = LOAD 'data.txt' AS (name:chararray,pet:chararray);
b = GROUP a BY pet;
c = FOREACH b {
ordered = ORDER a BY name DESC;
GENERATE group, ordered;
}
```

```
DUMP b;
(dog,{(Tom,dog)})
(cat,{(Doug,cat),(Mike,cat)})
(fish,{(Sarah,fish)})
```

Nested Ordering

- ORDER BY can be applied within each group
- Example

```
a = LOAD 'data.txt' AS (name:chararray,pet:chararray);
b = GROUP a BY pet;
c = FOREACH b {
ordered = ORDER a BY name DESC;
GENERATE group, ordered;
}
```

```
DUMP c;
(cat,{(Mike,cat),(Doug,cat)})
(dog,{(Tom,dog)})
(fish,{(Sarah,fish)})
```

Adding comments to a script

- Single line comments:
    - -- This is a comment

- Multi line comments:
    - /*
    - This is a longer
    - comment.
    - */

Case Sensitivity in Pig

Case -sensitive	Case-insensitive
Aliases (names of relations and fields)	Keywords (LOAD, USING, FILTER, ls, quit, etc.)
Functions (COUNT, AVG, PigStorage, etc.)	
String literals	

Hands-On Exercise

Advanced PigLatin

Advanced Relational Operations

- FLATTEN
- FOREACH
- JOINS
    - OUTER JOIN
- COGROUP
- UNION
- SAMPLE
- SPLIT
- CROSS

FLATTEN

- Used to remove a level of nesting from a bag or tuples
  - Often used to un-nest the results of union
- Example
  - Given a list of names of people with multiple pets, produce a list of names, group by pet

```
grunt> a = load 'pets.txt' as (name:chararray,pet:chararray);

grunt> b = GROUP a by pet;

grunt> dump b;
(cat,{(Doug,cat),(Tom,cat),(Mike,cat)})
(dog,{(Doug,dog),(Tom,dog),(Sarah,dog)})
(bird,{(Doug,bird)})
(fish,{(Mike,fish),(Sarah,fish)})
(hamster,{(Mike,hamster)})
```

FLATTEN (Cont’d)

- Flatten pulls the pet entries out of the ‘a’ bag

```
grunt> c = foreach b generate flatten(a.name), group;

grunt> dump c;

(Doug,cat)
(Doug,dog)
(Doug,bird)
(Tom,dog)
(Tom,cat)
(Mike,cat)
(Mike,fish)
(Mike,hamster)
(Sarah,fish)
(Sara,dog)
```

FLATTEN (Cont'd)

- When we remove a level of nesting in a bag, sometimes we cause a cross product to happen
- Example:

```
Ex: group g = (a, {(b,c), (d,e)})

grunt> c = foreach g generate $0 flatten($1), group;
grunt> dump c;

(a,b,c)
(a,d,e)
```

Nested FOREACH

- FOREACH can apply a set of relational operations to each record in the data stream
  - It processes every row to generate a derived set of rows, using GENERATE clause to define the fields in each derived row
- Example:
  - Given the 'u.user' dataset, return the average age by gender and occupation

```
grunt> data = load 'u.user' USING PigStorage('|') as
(id:int,age:int,gender:chararray,occup:chararray,zip:int);

(1,24,M,technician,85711)
(2,53,F,other,94043)
(3,23,M,writer,32067)
(4,24,M,technician,43537)
(5,33,F,other,15213)
(6,42,M,executive,98101)
(7,57,M,administrator,91344) . . .
```

Nested FOREACH

- FOREACH calculates the AVG age for each ‘grp’

```
grunt> grp = group data by (gender, occup);
((F,engineer),{(827,23,F,engineer,80228),(786,36,F, . . .
((F,salesman),{(925,18,F,salesman,49036),(531,30,F, . . .
((F,executive),{(186,39,F,executive,0),(835,44,F,ex . . .

grunt> resultset = foreach grp {
GENERATE group.gender, group.occup, AVG(data.age); };
(F,engineer,29.5)
(F,salesman,27.0)
(F,executive,44.0)
(F,retired,70.0)
(M,engineer,34.5)
(M,salesman,32.3)
(M,executive,36.6)
(M,retired,62.5) . . .
```

JOINS

- Pig Latin supports inner and outer joins of two or more relations
  - Supports joining on multiple fields
- Syntax for inner join:

```
bag2 = JOIN bag1 BY field [,bagn BY field...]
```

- Examples

```
joined = JOIN pets BY names, hobbies BY names;
joined = JOIN a BY $0, b BY $2, c BY $1;
joined = join a by (id, name), b by (id, name)
```

OUTER JOINS

- Pig can perform left, right and full outer joins(similar to SQL)

- Outer Join Support
  - Pig must know the schema for the side it may need to fill in nulls for
  - For e.g. Left outer joins must have right side bag schema defined, Right outer joins must have left side bag schema defined

- Syntax:

```
bagx = JOIN bag1 BY field[LEFT|RIGHT|FULL], bag2 BY field;
```

- Different JOINS like Replicated, Merge and Skewed Joins are covered in the Best Practices section

COGROUP

- COGROUP is a generalization of GROUP
  - GROUP collects records of one input based on a key
  - COGROUP collects records of n inputs based on a key

- Relations are implicitly grouped on join field

- Syntax:

```
bag3 = COGROUP bag1 BY field, bag2 BY field;
```

- Output is a set of tuples for each group key:

(group, {bag of records}, {bag of records})

records from first relation

records from second relation

COGROUP Example

Pets.txt:

Doug	Cat
Tom	Dog
Mike	Cat
Sarah	fish

Hobbies.txt:

Doug	reading
Tom	Swimming
Mike	biking
Philip	reading

```
grunt> grpd = COGROUP pets BY name, hobbies BY name;
grunt> DUMP grpd;

(Tom,{(Tom,dog)},{(Tom,swimming)})
(Doug,{(Doug,cat)},{(Doug,reading)})
(Mike,{(Mike,cat)},{(Mike,biking)})
(Sarah,{(Sarah,fish)},{})
(Philip,{},{(Philip,reading)})
```

COGROUP (Cont'd)

- Another way to think of COGROUP is as the first half of a join
  - The keys are collected together, but the cross product is not done
- In fact, COGROUP plus FOREACH, where each bag is flattened, is equivalent to a join, as long as there are no null values in the keys
- COGROUP handles null values in the keys similarly to group and unlike join i.e. all records with a null value in the key will be collected together

SAMPLE

- Use SAMPLE to chose a random set of tuples from dataset
- Syntax:

```
bag2 = SAMPLE bag1 N
```

N should be a number between 0-1,
For example **.05** or **.5**

SPLIT

- Partitions a relation into 2 or more relations
  - SPLIT has no 'else' clause
  - One record could match more than one split

- Syntax:

```
SPLIT bag1 INTO bag2 IF expression,
bag3 IF expression [,...]
```

- Example:

```
SPLIT users INTO males IF gender == 'M',
females IF gender == 'F',
voters IF (age >= 18 AND resident = 1);
```

CROSS

- CROSS matches the mathematical set operation of the same name
- In the following PigLatin script, CROSS takes every record in NYSE_daily and combines it with every record in NYSE_dividends

```
--cross.pig
-- you may want to run this in a cluster, it produces about
3G of data

daily = load 'NYSE_daily' as (exchange:chararray,
symbol:chararray,
date:chararray, open:float, high:float, low:float,
close:float, volume:int, adj_close:float);

divs = load 'NYSE_dividends' as (exchange:chararray,
symbol:chararray,
date:chararray, dividends:float);

tonsodata = cross daily, divs parallel 10;
```

CROSS (Cont'd)

- CROSS tends to produce a lot of data
- Given inputs with n and m records respectively, CROSS will produce output with n x m records

CROSS Example

- Pig's join operator supports only equi-joins, i.e. joins on an equality condition
- Non equi-joins (also called *theta joins*) are difficult to do
- They can be done in Pig using CROSS followed by filter

```
--thetajoin.pig
--running this one on a cluster too

daily = load 'NYSE_daily' as (exchange:chararray,
symbol:chararray,
date:chararray, open:float, high:float, low:float,
close:float, volume:int, adj_close:float);

divs = load 'NYSE_dividends' as (exchange:chararray,
symbol:chararray,date:chararray, dividends:float);

crossed = cross daily, divs;

tjnd = filter crossed by daily::date < divs::date;
```

PIG Built-in Functions

Function	Description
AVG	average of the values in a column
CONCAT	concatenates 2 strings
COUNT	count the no of elements, ignore NULLs
COUNT_STAR	count the no of elements, including NULLs
DIFF	find the differing elements
IsEMPTY	Tests if a bag is empty
MAX/MIN	max or min value in a column
SUM	adds the values in a column
SIZE	The number of elements in a dataset
TOKENIZE	split a string into words

Extending Pig's functionality

- Scripts
  - External scripts can be accessed with the STREAM operator
- UDFs
  - Three supported languages: Java, Python and JavaScript
  - The most extensive support is provided for Java
  - Limited support is provided for Python and JavaScript
- Macros
  - Pig supports definition, expansion and import of macros
- Piggy Bank
  - Access Java UDFs written by other users
  - Contribute java UDFs

STREAM

- The STREAM operator sends a relation through an external script
  - The script reads the incoming records as tab-delimited
  - Scripts can also be defined and reused

- Example

```
b = STREAM a THROUGH 'script.py';
b = STREAM a THROUGH 'cut -f 2';
DEFINE mycmd 'script.py';
b = STREAM a THROUGH mycmd;
```

Java UDFs

- Pig allows user-defined functions written in Java
  - Write a java class that extends ‘EvalFunc’ and implements ‘exec’ method
  - Compile and package into jar
  - Tell Pig about the jar using the REGISTER keyword
  - Optionally DEFINE a function name
  - Invoke the function name in the pig script

```
public class MyFunc extends EvalFunc {
public Double exec(Tuple input) {
...
```

```
grunt> REGISTER my-code.jar;
grunt> DEFINE myFunc com.examples.MyFunc();
grunt> b = FOREACH a GENERATE myFunc($0);
```

Java UDFs Example – UpperCASE UDF

- The UDF provided in this example takes an ASCII string and produces an uppercase of version
  - The UDF must be registered before it can be used
  - Multiple register commands can be used in the same script

```
-- myscript.pig
REGISTER myudfs.jar;
A = LOAD 'student_data'
AS (name: chararray, age: int, gpa: float);
B = FOREACH A GENERATE myudfs.UPPER(name);
DUMP B;
```

Java UDFs Example – UpperCASE UDF

- Implementation of UPPER UDF

```
package myudfs;
public class UPPER extends EvalFunc<String>
{
	public String exec(Tuple input) throws IOException {
		if (input == null || input.size() == 0)
			return null;
		try{
			String str = (String)input.get(0);
			return str.toUpperCase();
		}catch(Exception e){
		throw new IOException("Caught exception
	processing input row ", e);
		}
	}
}
```

Java UDFs Example – UpperCASE UDF

- Implementation of UPPER UDF

```
package myudfs;
public class UPPER extends EvalFunc<String>
{
                                                                                {
```

The exec function is invoked on every input tuple. The input into the function is a tuple with input parameters in the order they are passed to the function in the Pig script. In this example, it will contain a single string field corresponding to the student name.

```
              }
        }
}
```

Java UDFs Example – UpperCASE UDF

- Implementation of UPPER UDF

```
package myudfs;
public class UPPER extends EvalFunc<String>
{
        public String exec(Tuple input) throws IOException {
                if (input == null || input.size() == 0)
                        return null;
                try{
                        String str = (String)input.get(0);
                        return str.toUpperCase();
```

If the format of the data does not match the expected type, a NULL value should be returned else return the string converted to upper case.

```
}
```

Example – SubtractOne UDF

- The UDF provided in this example takes in a file of random ints and subtracts 1

```
import org.apache.pig.*;
public class SubtractOne extends
                            org.apache.pig.EvalFunc<Long> {
       public Long exec(org.apache.pig.data.Tuple input)
              throws java.io.IOException {
       try {
              int param = (Integer)input.get(0);
              return (long)param - 1L;
       } catch(Exception e) {
              throw new java.io.IOException("Something bad
                                           happened!", e);
              }
       }
}
```

Example – SubtractOne UDF

- Start the grunt shell in local mode

```
$ pig -x local
```

- Register the ‘subone.jar’ file

```
grunt> register subone.jar;
```

- Use the jar file in a Pig script

```
grunt> numbers = LOAD 'input' AS (i:int);
grunt> smaller = FOREACH numbers GENERATE
SubtractOne(i);
grunt> dump smaller;
```

Example – SubtractOne UDF

- Start the grunt shell in local mode

```
$ cat input
6625
473
745
20348
10182
23625
5471
18112
25029
8317
24777
```

```
grunt> dump smaller;
(6624)
(472)
(744)
(20347)
(10181)
(23624)
(5470)
(18111)
(25028)
(8316)
(24776)
```

Accessing *PiggyBank*

- **PiggyBank** is Pig's repository of user contributed functions
  - Functions for math, parsing date/strings, custom loaders, etc.
  - http://wiki.apache.org/pig/piggybank

- **PIggyBank** functions are the part of Pig distribution
  - The piggybank jar must be registered to use them

```
REGISTER '/usr/lib/pig/contrib/piggybank/java/
piggybank.jar';
backwards = FOREACH pets GENERATE
org.apache.pig.piggybank.evaluation.string.Reverse($1);
DUMP backwards;
(tac)
(god)
(tac)
(hsif)
```

Pig Macros

- DEFINE (macros) available in Pig
  - Can appear anywhere in Pig script
  - Can include references to other macros
  - Recursive references are not allowed

- Syntax

```
DEFINE macro_name (alias|integer|float|string literal)
RETURNS {void | alias [, alias ...]} { pig code };
```

- Example

```
DEFINE filter_macro(A, sortkey) RETURNS C {
B = FILTER $A BY my_filter(*);
$C = ORDER B BY $sortkey; };
```


Pig Debugging

Debugging Pig – diagnostic Operators

- DESCRIBE
- DUMP
- EXPLAIN
- ILLUSTRATE

DESCRIBE

- DESCRIBE returns the schema of a relations
  - Can view outer relations as well as relations in nested FOREACH statement

- Example

```
A = LOAD 'u.user' AS
(id:int,age:int,gender:chararray,occup:chararray,zip:int);
B = GROUP A BY occup;
DESCRIBE A;
data: {id: int,age: int,gender: chararray,occup:
chararray,zip: int}
DESCRIBE B;
b: {group: chararray,data: {id: int,age: int,gender:
chararray,occup:chararray,zip: int}}
```

DUMP

- DUMP displays the results to screen
  - Can be used as debugging device to make sure that the results you were expecting are actually generated

- Example

```
A = LOAD 'u.user' AS
(id:int,age:int,gender:chararray,occup:chararray,zip:int);
DUMP A;
(925,18,F,salesman,49036)
(926,49,M,entertainment,1701)
(927,23,M,programmer,55428) . . .
B = FILTER A BY gender matches 'F';
DUMP B;
(908,44,F,librarian,68504)
(909,50,F,educator,53171)
(911,37,F,writer,53210)
(914,44,F,other,8105)
(917,22,F,student,20006) . . .
```

EXPLAIN

- EXPLAIN displays the execution plans
  - Outputs the logical, physical and MapReduce execution plans

- Example

```
A = LOAD 'u.user' AS (id:int,age:int,gender:chararray, . . .
B = GROUP A BY gender;
C = FOREACH B GENERATE COUNT(A.age);
EXPLAIN C;
-----------------------------------------------
Logical Plan:
-----------------------------------------------
Store xxx-Fri Dec 05 19:42:29 UTC 2008-23 Schema: {long}
Type: Unknown
|---ForEach xxx-Fri Dec 05 19:42:29 UTC 2008-15 Schema:
{long} Type: bag
etc ...
```

ILLUSTRATE

- Chooses a subset of records for testing all data transformations
- Syntax:

```
ILLUSTRATE alias;
```

- Notes
  - The load must provide a schema
  - Not all operations work (ex: LIMIT)

ILLUSTRATE – Example

```
pets = LOAD 'data' AS (name:chararray,pet:chararray);
grpd = GROUP pets BY pet;
ILLUSTRATE grpd;
------------------------------------------------
| pets | name: chararray | pet: chararray |
------------------------------------------------
| | Doug | cat |
| | Mike | cat |
| | Sarah | fish |
------------------------------------------------
-------------------------------------------------------------
| grpd | group: chararray | pets: bag({name:
chararray,pet: chararray}) |
-------------------------------------------------------------
| | cat | {(Doug, cat), (Mike, cat)} |
| | fish | {(Sarah, fish)} |
```

Collecting Statistics

- PIG Statistics
  - Framework for collecting and storing script-level statistics
  - Statistics are collected during script execution
  - Statistics can be retrieved after job is done

- Accessing Statistics
  - At the end of each script run
  - Using Java API
  - From a PIG script using Hadoop Job history Loader

Accessing Statistics

- Job statistics are visible at the end of each run

```
Job Stats (time in seconds):
JobId Maps Reduces MaxMapTime MinMapTIme AvgMapTime
MaxReduceTime . . .
job_20 1 0 2 2 2 0
Input(s):
Successfully read 943 records (22983 bytes) from:
"hdfs://localhost/user/training/u.user"
Output(s):
Successfully stored 943 records (28286 bytes) in:
"hdfs://localhost/tmp/temp1996599494/tmp-535576523"
Counters:
Total records written : 943
Total bytes written : 28286
Spillable Memory Manager spill count : 0
Total bags proactively spilled: 0
Total records proactively spilled: 0
```

Accessing Statistics (Cont'd)

- Can also be collected after a run with the Java API
  - Located in the package org.apache.pig.tools.pigstats
  - The PigRunner class gives users a statistics object back
- Example

```
public abstract class PigRunner {
public static PigStats run(String[] args,
PigProgressNotificationListener listener)
}
public interface PigProgressNotificationListener extends
java.util.EventListener {
// just before the launch of MR jobs for the script
public void LaunchStartedNotification(int numJobsToLaunch);
// number of jobs submitted in a batch
public void jobsSubmittedNotification(int numJobsSubmitted);
```

Pig Unit - Testing Pig Scripts

- Pig Unit
  - Unit framework that enables you to easily test your pig scripts
  - Unit testing, Regression testing and rapid prototyping
  - No cluster setup is required if run in a local mode

- Running PigUnit in Local mode
  - Runs in Pig’s local mode by default
  - Fast, and can use the local file system as the data store

- Running PigUnit in MapReduce mode
  - Requires a Hadoop cluster
  - Enabled using the java system property ‘pigunit.exectype.cluster’
    Ex: -D pigunit.exectype.cluster=true
  - The cluster must be specified in the CLASSPATH

Accessing Statistics (Cont'd)

- Can also be accessed in a Pig script with Hadoop Job History Loader
  - Loads Hadoop job history files and job xml files
  - For each mapreduce job, the loader provides a tuple with schema (j:map[], m:map[], r:map[])
  - The first map in the schema contains job-related entries

- Example
  - Find scripts that generate more than three MR jobs

```
a = load '/mapred/history/done' using
HadoopJobHistoryLoader() as (j:map[], m:map[], r:map[]);
b = group a by (j#'PIG_SCRIPT_ID', j#'USER', j#'JOBNAME');
c = foreach b generate group.$1, group.$2, COUNT(a);
d = filter c by $2 > 3;
dump d;
```

Pig Unit Example

- To compute the top N of the most common queries

```
@Test
public void testTop2Queries() {
String[] args = { "n=2" };
PigTest test = new PigTest("top_queries.pig", args);
String[] input = { "yahoo", "yahoo", "yahoo", "twitter",
"facebook", "facebook", "linkedin" };
String[] output = {"(yahoo,3)", "(facebook,2)"};
test.assertOutput("data", input, "queries_limit",
output);
}
```

The Hadoop Web UI

- All Hadoop daemons contain a web server
  - Exposes information on a well-known port
- Most important for the developers is the JobTracker Web UI
  - http://<job_tracker_address:50030>/
  - http://localhost:50030/ (for pseudo-distributed mode)
- Also the Name Node Web UI
  - http://<name_node_address:50070>/

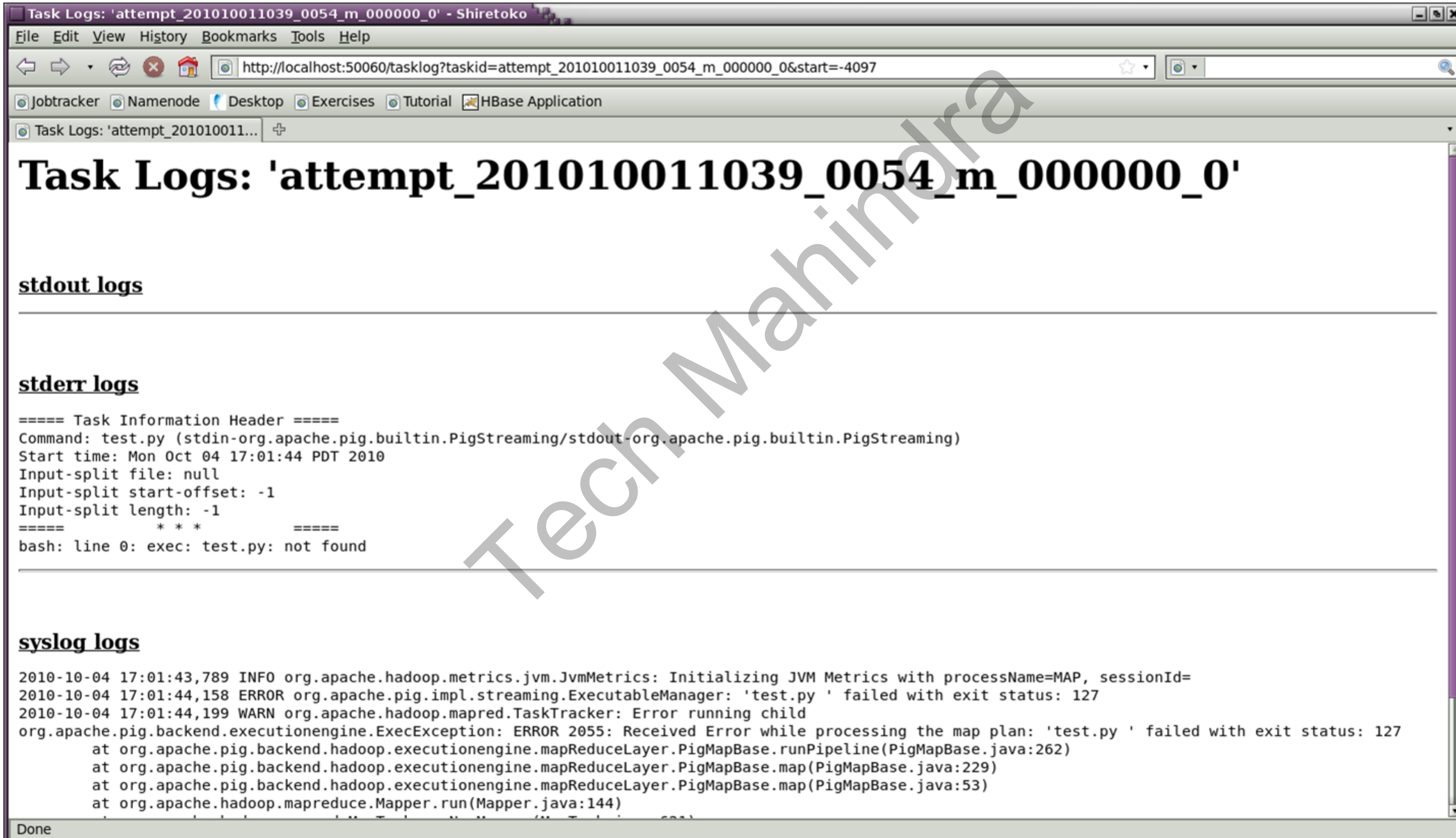
The JobTracker UI

- Job Tracker UI displays statistics on your Map Reduce cluster
  - Cluster summary stats
  - Scheduling information stats
  - Running jobs
  - Completed jobs
  - Failed jobs

- Drill down access to individual jobs and error logs

JobTracker UI

- http://localhost:50030/



PIG BEST PRACTICES

Pig Best Practices

- Pig's best practices can be classified into the following sections
  - Combiner
  - Multi-query execution
  - Performance enhancers
  - Join optimizations

Combiner – Optimizing FOREACH

- What is a COMBINER?
  - An optimizer that is automatically invoked by Pig
  - Significantly improves performance
  - Used to optimize specific FOREACH use cases
- When is it used?
  - Determined by the GROUP and FOREACH statements
  - Non-nested FOREACH
    - Used when all projections are either expressions on the group column or expressions on algebraic UDFs
  - Nested FOREACH
    - Used as long as the only nested operation used in DISTINCT
- Sometimes its possible to rearrange the script such that the combiner will guaranteed to run

When is the Combiner not used?

- A combiner is not used when any of the operator comes between the GROUP and FOREACH
- Even if the statements are next to each other, the optimizer might rearrange them
- Example
  - In this example the optimizer will push FILTER above FOREACH thus preventing the use of COMBINER

```
A = LOAD 'u.user' AS (id,age,gender,occup,zip);
B = group A by age;
C = foreach B generate group, COUNT (A);
D = filter C by group.age <30;
```

When is the Combiner not used?

- Non-nested FOREACH Example
  - The GROUP statement can be referred to as a whole or by accessing individual fields
  - The GROUP statement and its elements can appear anywhere in the projection
  - A variety of expressions can be applied to algebraic functions

```
A = LOAD 'u.user' AS (id,age,gender,occup,zip);
B = group A by age;
C = foreach B generate
COUNT(org.apache.pig.builtin.Distinct(A.occup)), group.age;
```

Combiner – Rearranging Pig

- How can we change this code to guarantee that the COMBINER will run?

```
A = LOAD 'u.user' using PigStorage('|') AS (id:int,
age:int, gender:chararray, occup:chararray,
zip:chararray);
B = group A by age;
C = foreach B generate group, COUNT(A);
D = filter C by A.age <30;
```

- Move the FILTER before the GROUP

```
A = LOAD 'u.user' using PigStorage('|') AS (id:int,
age:int, gender:chararray, occup:chararray,
zip:chararray);
B = filter A by age < 30;
C = group B by age;
D = foreach C generate group, COUNT(B);
```

Multi-Query Execution

- What is a Multi-Query execution?
  - Allows pig to process a script or a batch of statements simultaneously
  - Turned on by default

- How does it work?
  - The script is first parsed to determine if intermediate tasks can be combined
  - Execution starts only after the parsing is completed

- Two (step) run scenarios are optimized
  - Explicit and Implicit splits
  - String intermediate results

Explicit SPLITS

- Makes the split non-blocking and allows processing to continue
  - Reduces the amount of data that has to be stored at the split
- Allows multiple outputs from a job
  - Some results can be stored as a side-effect of the main Job
- Allows multiple split branches to carried on to the combiner
  - Reduces the amount of I/O in the case where multiple branches in the split can benefit from a combiner run

```
A = LOAD ...
SPLIT A INTO B IF ..., C IF ...
STORE B ...
STORE C ...
```

Implicit SPLITS

- Without multi-query execution
  - Executes all the dependencies of B and stores it and then executes all the dependencies of C and stores it
- With multi-query execution
  - Adds an implicit split, eliminates the processing of A multiple times
  - Then treats (the bag) as an Explicit split
- Allows multiple split branches to carried on to the combiner
  - Reduces the amount of I/O in the case where multiple branches in the split can benefit from a combiner run

```
A = LOAD ...
B = FILTER A ...
C = FILTER A ...
STORE B ...
STORE C ...
```

Storing Intermediate Results

- STORE v/s DUMP
  - Use STORE to save your results
  - Do not use DUMP as it will disable multi-query execution

- DUMP example
  - Multi-query execution will be disabled
  - Two separate jobs will be created to execute this script
    - A > B > DUMP and then
    - A > B > C > STORE

- Allows multiple split branches to carried on to the Combiner
  - Reduces the amount of I/O in the case where multiple branches in the split can benefit from a Combiner run

```
A = LOAD 'input' AS (x, y, z);
B = FILTER A BY x > 5;
DUMP B;
C = FOREACH B GENERATE y, z;
STORE C INTO 'output';
```

Performance Enhancers – Use Types

- By default, Pig treats numeric data as type double
  - Actual data might be much smaller and can be "typed" as integer or long
  - Specifying the correct types speeds arithmetic computation
  - Also allows for early error detection

```
-- Slow Query
A = load 'u.user' as (id,age,gender,occup,zip);
B = foreach A generate id + 100;
-- Fast Query
A = load 'u.user' as (id:int,age:int,gender . .
B = foreach A generate id + 100;
```

Project early & often

- In Pig, fields which are unused are also loaded into memory
  - Improve performance by removing unused fields from a row
  - Reduces amount of data carried through the Map and Reduce phases

- Example
  - The age ,occup, movieid and zip fields are never used

```
A = load 'u.user' as (uid,age,gender,occup,zip);
B = load 'u.data' as (uid,movieid,rating);
C = join A by uid, B by uid;
D = group C by gender;
E = foreach D generate group, COUNT(rating);
```

Filter data as early as possible

- Remove unnecessary columns

```
A = load 'u.user' as (uid,age,gender,occup,zip);
B = load 'u.data' as (uid,movieid,rating);
userfields = FOREACH A GENERATE uid, gender;
C = join userfields by uid, B by uid;
D = group C by gender;
E = foreach D generate group, COUNT(rating);
```

- Its also beneficial to remove unnecessary records as early as possible

```
users = load 'u.user' as (uid,age,gender,occup,zip);
adults = FILTER users BY age >= 18;
```

Read Once, Use Many times

- Pig allows a relation to be manipulated & used in multiple ways
  - Data can be streamed once but processed multiple times in different ways
  - Especially useful in a script where it is needed to issue multiple STORE operations

```
a = LOAD 'data' ...
b = FILTER a ...
c = GROUP b BY f1;
d = GROUP b BY f2;
```

Choosing the right parallelism

- Choose the number of reducers using the PARALLEL keyword
- May be used with GROUP, COGROUP, JOIN, DISTINCT, LIMIT or ORDER BY
- Example

```
b = GROUP a BY f1 PARALLEL 20;
```

- You can set the default value for PARALLEL with
  set default_parallel n

By default, Pig sets the number of reducers using a heuristic based on the size of the input data

Parameter Substitution

- Parameterize the scripts that will be run repeatedly
- Parameters are prefixed with $ (ex: $filename)
- Supply parameter values with: *pig –param filename = /path/file*

Join Optimizations – Default Joins

- Regular joins are reduce-side joins
  - The last relation is guaranteed to be streamed and not loaded into memory

- Can be optimized by listing the biggest relation last

```
smaller = LOAD 'smalldata' AS (a, b, c);
bigger = LOAD 'bigger' AS (x, y, z);
joined = JOIN smaller BY a, bigger BY x;
```

Drop NULLS before Joins

- Rows with the null key will always be dropped
  - But only at the last possible moment
  - All null keys go to a single reducer
  - If the key is null, even for a few records, dropping the nulls can give significant performance benefit

- Default joins can be further optimized dropping NULLs

```
A = load 'myfile' as (t, u, v);
B = load 'myotherfile' as (x, y, z);
A1 = filter A by t is not null;
B1 = filter B by x is not null;
C = join A1 by t, B1 by x;
```

Specialized Joins

Pig has optimizations for three types of specialized joins

- Replicated Join
  - If outer join is on a bag small enough to fit into memory
- Merge Join
  - If both inputs are sorted on the join key
- Skewed Join
  - If Join key is skewed

Replicated Joins

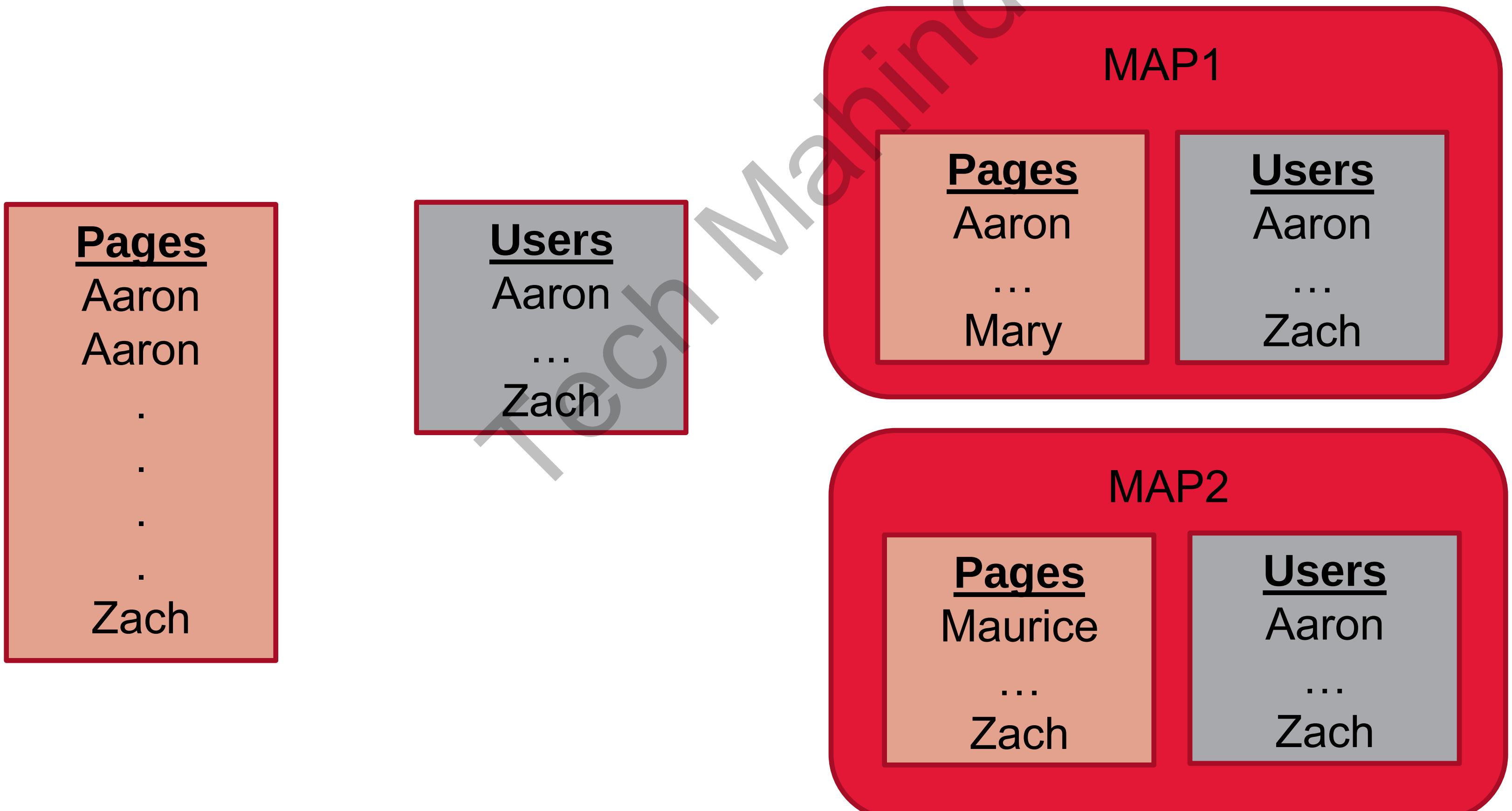
- Map side joins (Replicated joins) can be used if the smaller relations fit in memory

- List the large table first and specify USING the *"replicated"*

```
smaller = LOAD 'smalldata' AS (a, b, c);
bigger = LOAD 'bigger' AS (x, y, z);
joined = JOIN bigger BY x, smaller BY a USING
"replicated";
```

Replicated Joins - Illustrated

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using
'replicated';
```



Merge Joins

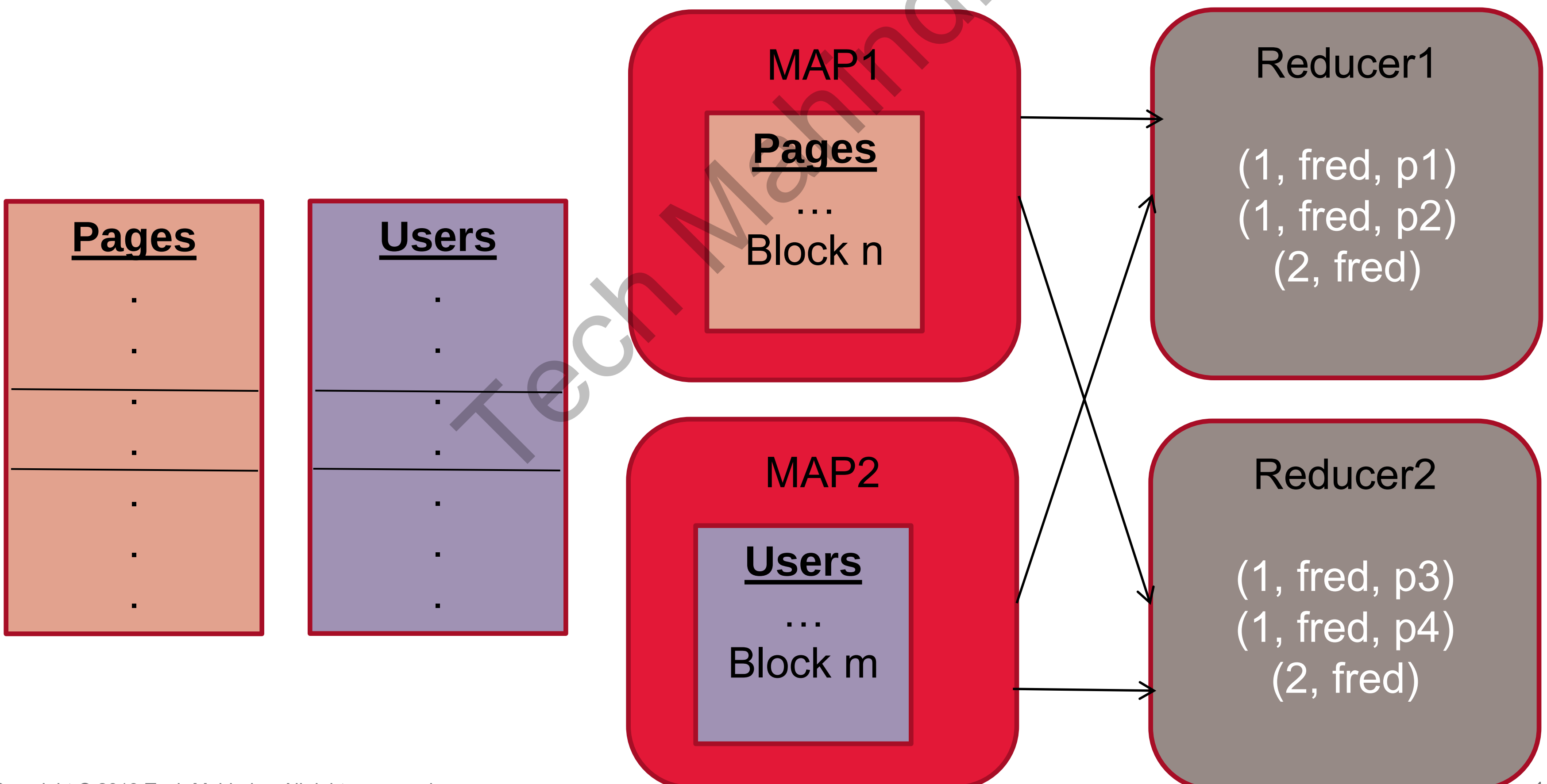
- Inputs are already sorted on the join key
  - Data is joined in the map phase
  - Builds an index for each sampled record - the keys and the file offsets

- Each map uses the index to seek the appropriate record in the right input and begin doing the join

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

Skewed Joins – Illustrated

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Users by name, Pages by user using 'skewed';
```



Skewed Joins

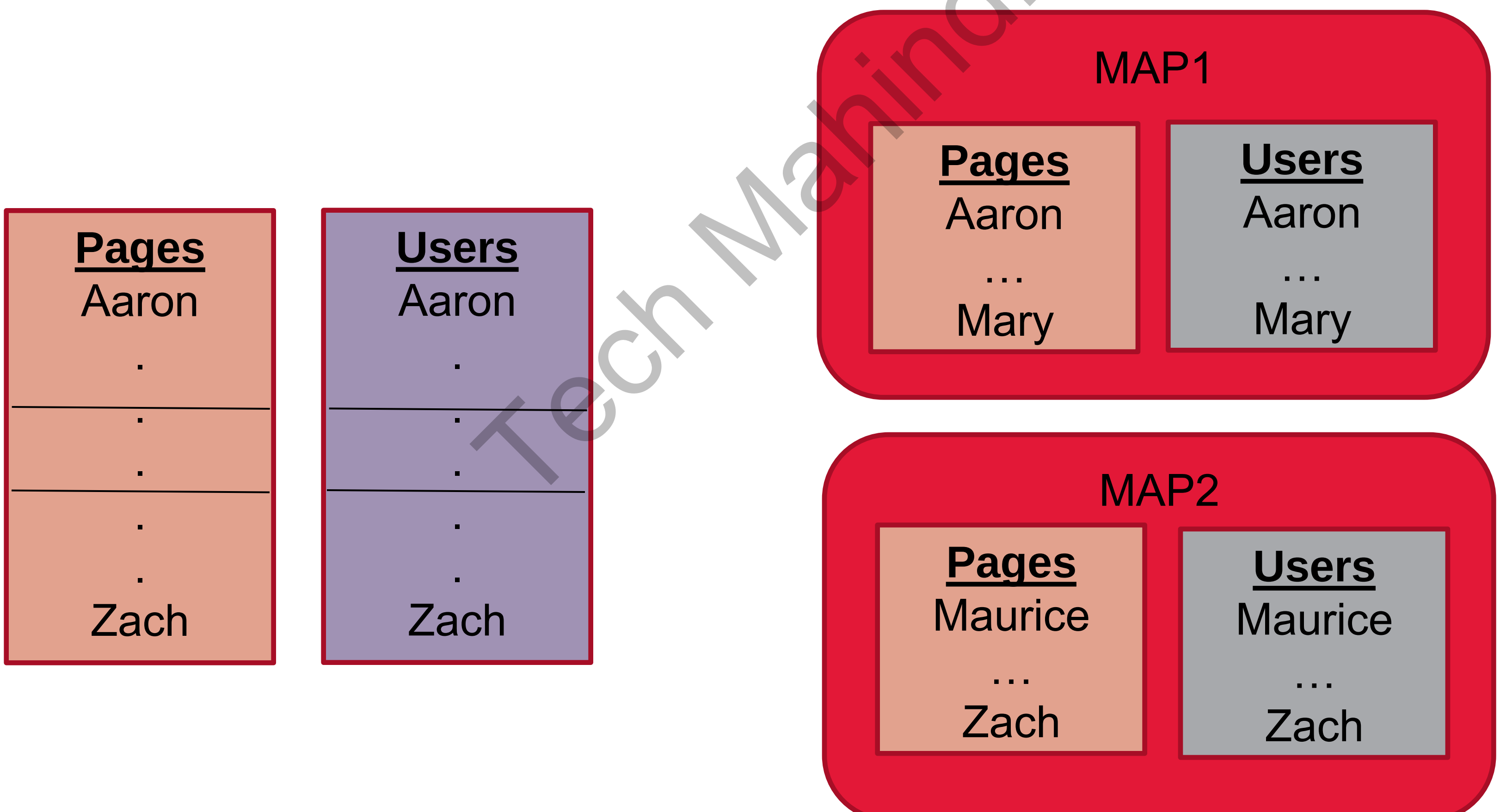
- Parallel joins are vulnerable to the presence of data skew
  - If the underlying data is sufficiently skewed, load imbalances will swamp any of the parallelism gains

- Skewed join computes a histogram of the key space
  - Uses this data to allocate reducers for the given key
  - Splits the left input on the join and streams the right input
  - The left input is sampled to create the histogram

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

Merge Joins – Illustrated

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Users by name, Pages by user using 'merge';
```



Hands-On Exercise

Thank You

Disclaimer

Tech Mahindra Limited, herein referred to as TechM provide a wide array of presentations and reports, with the contributions of various professionals. These presentations and reports are for informational purposes and private circulation only and do not constitute an offer to buy or sell any securities mentioned therein. They do not purport to be a complete description of the markets conditions or developments referred to in the material. While utmost care has been taken in preparing the above, we claim no responsibility for their accuracy. We shall not be liable for any direct or indirect losses arising from the use thereof and the viewers are requested to use the information contained herein at their own risk. These presentations and reports should not be reproduced, re-circulated, published in any media, website or otherwise, in any form or manner, in part or as a whole, without the express consent in writing of TechM or its subsidiaries. Any unauthorized use, disclosure or public dissemination of information contained herein is prohibited. Unless specifically noted, TechM is not responsible for the content of these presentations and/or the opinions of the presenters. Individual situations and local practices and standards may vary, so viewers and others utilizing information contained within a presentation are free to adopt differing standards and approaches as they see fit. You may not repackage or sell the presentation. Products and names mentioned in materials or presentations are the property of their respective owners and the mention of them does not constitute an endorsement by TechM. Information contained in a presentation hosted or promoted by TechM is provided “as is” without warranty of any kind, either expressed or implied, including any warranty of merchantability or fitness for a particular purpose. TechM assumes no liability or responsibility for the contents of a presentation or the opinions expressed by the presenters. All expressions of opinion are subject to change without notice.