Introduction to Pig

Do The Math

Chicago, IL
Bangalore, India
www.mu-sigma.com

December, 2013

Agenda

- Pig – An Overview
- Pig Data Model
- Introduction to Pig Latin
- Executing Pig
- Optimizing Pig
- Control Stuctures
- Pig Examples
- Exercises

Pig Philosophy

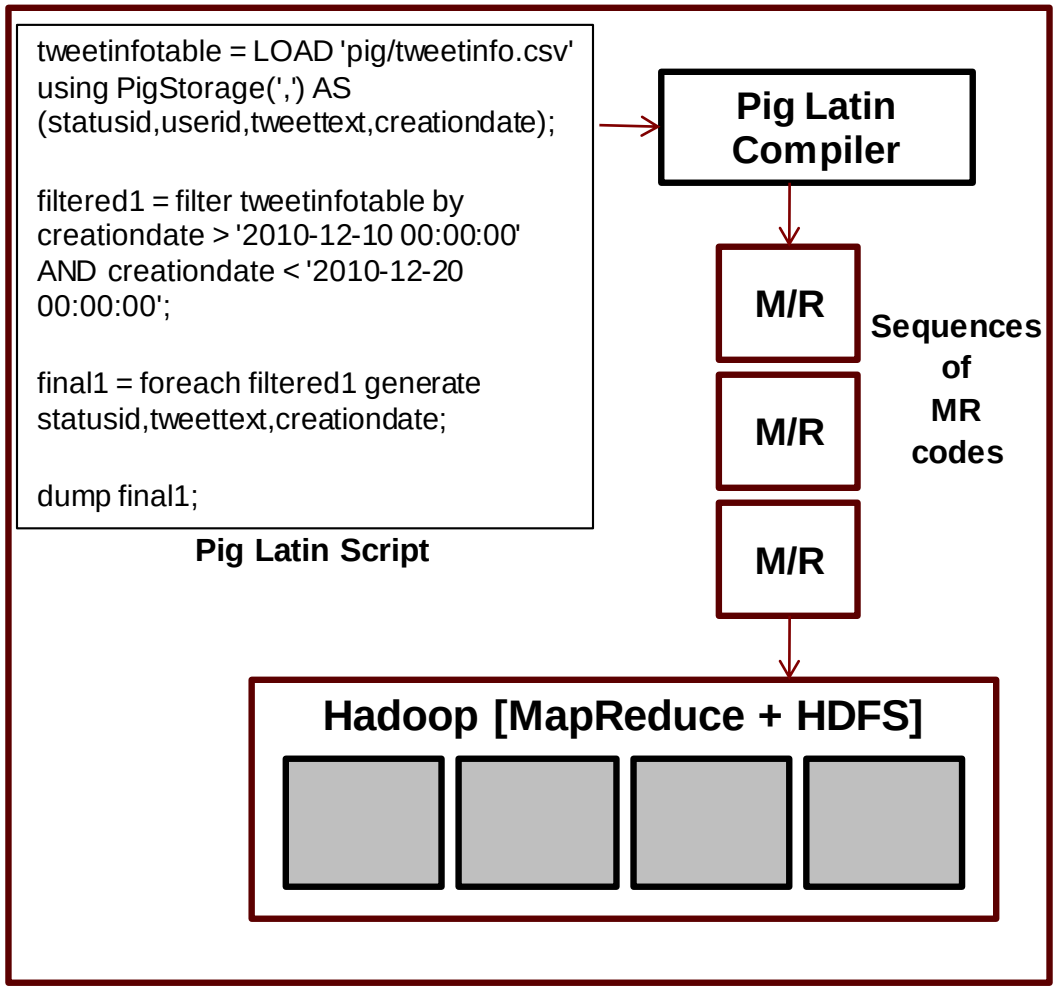
- Apache Pig, developed by Yahoo, is a platform for analyzing large data sets that uses Hadoop map-reduce framework and HDFS.
- It provides an engine for executing data flows in parallel on Hadoop
- **Pigs Eat Anything**
  - Pig can operate on data whether it has metadata or not. It can operate on data that is relational, nested, or unstructured.
- **Pigs Live Anywhere**
  - Pig is intended to be a language for parallel data processing. It is not tied to one particular parallel framework.
- **Pigs Are Domestic Animals**
  - Pig is designed to be easily controlled and modified by its users. Pig allows integration of user code where ever possible, so it currently supports user defined field transformation functions, user defined aggregates, and user defined conditionals.

Pig Use cases

- **Traditional Extract Transform Load (ETL)**
- **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.
- **Research on raw data**
  - Pig can operate in situations where the schema is unknown or incomplete or inconsistent and since it can easily manage nested data, researchers who want to work on data before it has been cleaned often prefer Pig.
- **Iterative processing**
  - Pig is strong contender for designing iterative based models. Suppose new data comes in every five minutes in your model, which needs a join to be done against the whole model. This step consists of inserts, updates, and deletes on the entire model. It is possible and reasonably convenient to express this combination in Pig Latin.

Pig Architecture

- Pig's infrastructure layer consists of
  - a compiler that produces sequences of Map-Reduce programs
  - Pig's language layer currently consists of a textual language called **Pig Latin**
  - Execution environment that allows user to submit Pig jobs. There are currently 2 environments – one local execution using local system's JVM, second distributed execution using Hadoop cluster
  - includes operators for many of the traditional data operations (load, store, join, sort, filter, etc.) as well as the ability for users to develop their own functions for reading, processing, and writing data



Agenda

- Pig – An Overview
- **Pig Data Model**
- Introduction to Pig Latin
- Executing Pig
- Optimizing Pig
- Control Stuctures
- Pig Examples
- Exercises

Pig comes with set of Simple to Complex Data Types

Category	Type	Description	Literal example
Numeric	int	32-bit signed integer	1
	long	64-bit signed integer	1L
	float	32-bit floating-point number	1.0F
	double	64-bit floating-point number	1.0
Text format	chararray	Character array in UTF-16	'a'
Binary	bytearray	Byte array	
Complex	tuple	Sequence of fields of any type	(1,‘Java')
	bag	An unordered collection of tuples, possibly with duplicates	{(1,‘Java'),(2,’R’)}
	map	A set of key-value pairs. Keys must be character arrays; values may be any type	['a‘ # ‘Java']

Pig Support for different Arithmetic operators (+. - , *, /)

	bag	tuple	map	int	long	float	double	chararray	bytearray
bag	X	X	X	X	X	X	X	X	X
tuple		X	X	X	X	X	X	X	X
map			X	X	X	X	X	X	X
int				int	long	float	double	X	cast as int
long					long	float	double	X	cast as long
float						float	double	X	cast as float
double							double	X	cast as double
chararray								X	X
bytearray									cast as double

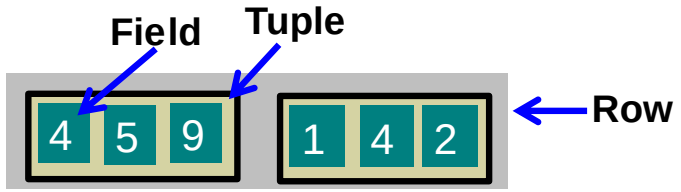
Pig Support for Modulus Operator (%)

	int	long	bytearray
int	int	long	cast as int
long		long	cast as long
bytearray			error

Pig Complex Data types - tuple

- **tuple**
  - A tuple can be considered as an ordered set of fields.
  - **Syntax -** (field [, field …])
- **Terms**
  - **( )** - A tuple is enclosed in parentheses ( ).
  - **Field** - A piece of data. A field can be any data type (including tuple and bag).
- **Usage**
  - You can think of a tuple as a row with one or more fields, where each field can be any data type and any field may or may not have data. If a field has no data, then the following happens:
    - » In a load statement, the loader will inject null into the tuple.
    - » In a non-load statement, if a requested field is missing from a tuple, Pig will inject null.
- **Example**
  - In this example the tuple contains three fields:
  - (Mary,20,3.5) : (Student:tuple (Student_name:chararray, Student_age:int, Student_GPA:double))
  - A row containing 2 tuples - (4,5,9) (1,4,2)

(r1:tuple(r1a:int, r1b:int,r1c:int),r2:tuple(r2a:int,r2b:int,r2c:int));

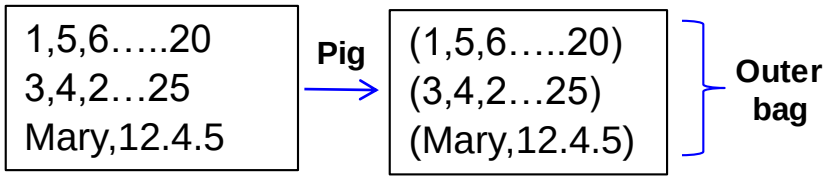


Pig Complex Data types - bag

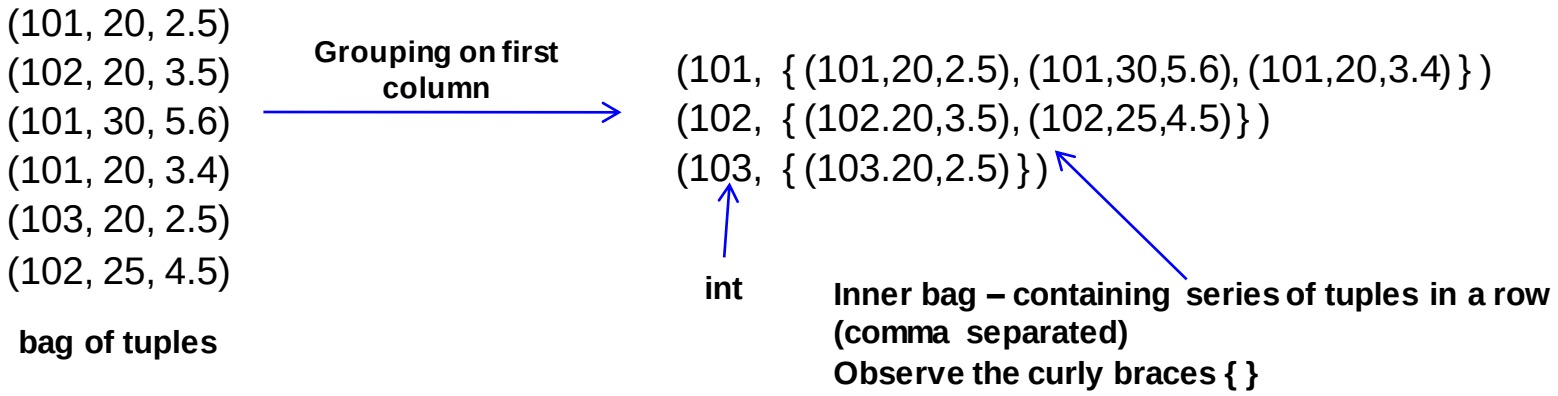
- **Bag**
  - A bag is a collection of tuples.
  - **Syntax: Inner bag -** { tuple [, tuple …] }
- **Terms**
  - { } - An inner bag is enclosed in curly brackets { }.
  - tuple - A tuple.
- **Usage**
  - A bag can have duplicate tuples.
  - A bag can have tuples with differing numbers of fields. However,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

Pig Complex Data types – bag (contd.)

- **Example: Outer Bag**
  - A data once loaded is bag of tuples. You can think of this bag as an outer bag.
  - Data



- **Example: Inner Bag**
  - Now, suppose we group, data by the first field to form another dataset.
  - In this example X is a relation or bag of tuples. The tuples in relation X have two fields. The first field is type int. The second field is type bag; you can think of this bag as an inner bag.



Pig Complex Data types - Map

- **Map**
  - A map is a set of key value pairs.
  - **Syntax (<> denotes optional)** [key#value <, key#value …>]
- **Terms**
  - [ ] - Maps are enclosed in straight brackets [ ].
  - # - Key value pairs are separated by the pound sign #.
  - Key - Must be chararray data type. Must be a unique value.
  - Value - Any data type.
- **Usage**
  - Key values within a relation must be unique.
- **Example**
  - In this example the map includes two key value pairs.
  - [name#John, age#30]

Relations, Bags, Tuples, Fields

- Pig Latin statements work with relations. A relation can be defined as follows:
  - A relation is a bag (more specifically, an outer bag).
  - A bag is a collection of tuples.
  - A tuple is an ordered set of fields.
  - A field is a piece of data.
- A Pig relation is a bag of tuples. 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.
- Relations are unordered which means there is no guarantee that tuples are processed in any particular order. Furthermore, processing may be parallelized in which case tuples are not processed according to any total ordering.

Agenda

- Pig – An Overview
- Pig Data Model
- **Introduction to Pig Latin**
- Executing Pig
- Optimizing Pig
- Control Stuctures
- Pig Examples
- Exercises

Pig Latin Statements

- A Pig Latin statement is an operator that takes a relation as input and produces another relation as output.
  - A LOAD statement reads data from the file system.
  - A series of "transformation" statements process the data.
  - A STORE statement writes output to the file system; or, a DUMP statement displays output to the screen.

- **Referencing Relations**
  - Relations are referred to by name (or alias). Names are assigned by you as part of the Pig Latin statement. In this example the name (alias) of the relation is Student_Data.

```
Student_Data = LOAD '/user/hadoop/student.txt' USING PigStorage() AS (name:chararray, age:int, gpa:double);
DUMP Student_Data;
```

  - **Sample Output:**

    (Peter,18,4.5) ====> Each tuple/row is referred by (name, age and gpa)

    (Samuel,19,3.8)

    (Bill,20,3.9)

    (Joe,18,3.8)

Pig Referencing

- **Referencing Fields**
  - Fields are referred to by positional notation or by name (alias).
  - Positional notation is generated by the system. Positional notation is indicated with the dollar sign ($) and begins with zero (0); for example, $0, $1, $2.
  - Names are assigned by you using schemas (or, in the case of the GROUP operator and some functions, by the system). You can use any name that is not a Pig keyword; for example, f1, f2, f3 or a, b, c or name, age, gpa.

```
Student_Data = LOAD '/user/hadoop/tweet.txt' USING PigStorage() AS (username:chararray, userid:int, friendcount:int);
```

	1st Field	2nd Field	3rd Field
Data type	chararray	int	int
Positional notation (system-generated)	$0	$1	$2
Possible name (user-generated)	username	userid	friendcount
Field value (Peter,188291,35)	Peter	188291	35

Pig Referencing

- **Referencing Fields that are Complex Data Types**
  - As noted, the fields in a tuple can be any data type, including the complex data types: bags, tuples, and maps.
  - Use the schemas for complex data types to name fields that are complex data types.
  - Use the dereference operator (dot operator) to reference and work with fields that are complex data types.

Operator	Symbol
Tuple dereference	tuple_name.field or tuple.(field,…)
bag dereference	bag.id or bag.(id,…)
map dereference	map#key

- **Example**
  - ((2,3,4),(5,3,9)) `(r1:tuple (a:int,b:int,c:int),r2:tuple(a:int,b:int,c:int));`
  - `r1.a = 2, r1.c = 4, r2.b = 3`
  - `r1.$0 = 2, r2.$1 = 3`

LOAD operator

- **LOAD**
  - Loads data from the file system.
  - **Syntax -** LOAD 'data' [USING function] [AS schema];

- **Terms**
  - **'data'** - The name of the file or directory, in single quotes. If you specify a directory name, all the files in the directory are loaded.
  - **USING** (Keyword) - If the USING clause is omitted, the default load function PigStorage is used.
  - **Function** - The load function.
    - » You can use a built-in function provided by pig. PigStorage is the default load function and does not need to be specified (simply omit the USING clause).
    - » You can write your own load function if your data is in a format that cannot be processed by the built-in functions.
  - **AS** (Keyword)
  - **Schema -** A schema using the AS keyword, enclosed in parentheses
    - » The loader produces the data of the type specified by the schema. If the data does not conform to the schema, depending on the loader, either a null value or an error is generated.
  - Note : [USING function] [As schema] – both are optional

Load Operator

- **Example**

- Student.txt (tab-separated) – Suppose its present on hadoop – "/user/hadoop/TrainingDatasets/Pig/IntroductionToPig/Student"

Student.txt

Roy	20	4.5
Joy	30	2.5
Nick	25	4.0

```
Student_Data = Load 'Student';

Student_Data = Load 'Student' USING PigStorage('\t');

Student_Data = Load 'Student' USING PigStorage('\t') AS (name:chararray, age:int, gpa:double);
```

DUMP Operator

- **DUMP**
  - Dumps or displays results to screen.
  - **Syntax -** DUMP alias;
- **Terms**
  - alias - The name of a relation.
- **Usage**
  - DUMP operator is used to display results on the console. It mainly provides interactivity by displaying the results without any persistence.
  - Mainly used for debugging.
  - DUMP is not preferred for running production level scripts as it doesn't make use of optimizations provided by pig. We should use STORE instead of DUMP for production scripts.
- **Example:**

```
A = LOAD 'TwitterData' AS (name:chararray, userid:int, friendcount:int);
DUMP A;
(Jessy,18234,40)
(Joy,19234,  23)
(Bill, 23324.93)
```

FILTER OPERATOR

- **FILTER**
  - Allows you to get rid of unwanted data. Basically selects tuples/rows from a relation based on some given condition.
  - **Syntax -** alias = FILTER alias BY expression;
- **Terms**
  - Alias - The name of the relation.
  - BY - Required keyword.
  - Expression - A boolean expression.
- **Usage**
  - Use the FILTER operator to work with tuples or rows of data, similar to SELECT-WHERE clause in SQL
  - FILTER is commonly used to select the data that you want; or, conversely, to filter out (remove) the data you don't want.
- **Example:**

```
A = LOAD 'TwitterData' AS (name:chararray, userid:int, friendcount:int);
B = FILTER A BY friendcount >25;
DUMP B;
```

(Jessy,18234,40)
(Joy,19234, 23)
(Bill, 23324.93)

After Filtering →

(Jessy,18234,40)
(Bill, 23324.93)

Filter Operator

- **Specifying Conditions**.
  - The logical connectives AND, OR and NOT can be used to build a condition from various atomic conditions.
  - Comparison Operators

Operator	Symbol
equal	==
not equal	!=
less than	<
greater than	>
less than or equal to	<=
greater than or equal to	>=
pattern matching	matches

- Thus, a somewhat more complicated condition can be

```
Y = FILTER A BY (f1 == '8') OR (NOT (f2*f3 > f1 OR f2>10));
Z = FILTER A BY (f1 matches ' *hadoop* ');
```

FOREACH operator

- **FOREACH – Pig's Projection Operator**
  - Mainly allows data transformations based on columns of data.
  - *foreach* takes a set of expressions and applies them to every record in the data pipeline;
  - **Syntax** alias = FOREACH { gen_blk | nested_gen_blk } [AS schema];

- **Terms**
  - alias - The name of relation (outer bag).
  - **gen_blk** –
    - » FOREACH … GENERATE used with a relation (outer bag).
    - » SYNTAX : alias = FOREACH alias GENERATE expression [expression ….]
  - **nested_gen_blk**
    - » FOREACH … GENERATE used with a inner bag.
    - » SYNTAX :

```
alias = FOREACH nested_alias {
        alias = nested_op; [alias = nested_op; …]
        GENERATE expression [, expression …]
        };
```

FOREACH operator

- Example:

```
Data = LOAD 'AddressData' AS (name:chararray, address:tuple(street:chararray,
       city:chararray, country:chararray) );

name_city = FOREACH Data GENERATE name, address.city ;

Dump name_city;
```

Austin	(Mu-Sigma street, Bangalore, India)
Chitra	(CMR street, Bangalore, India)
Kevin	(Little Street, California, US)

After ForEach →

(Austin, Bangalore)	
(Chitra, Bangalore)	
(Kevin, California)	

Relation – containing 2 columns:
1. name – chararray
2. address – tuple:-
   a. street:chararray
   b. city:chararray
   c. country:chararray

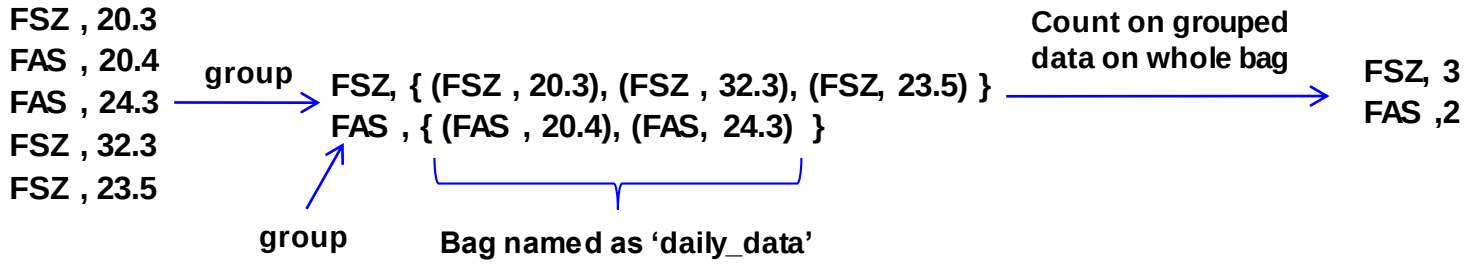
Relation – containing 2 columns:
1. name – chararray
2. city - chararray

GROUP OPERATOR

- **Group:**
  - collects records with the same key together
  - In SQL the group by clause works with aggregate functions. In Pig Latin there is no direct connection between group and aggregate functions. Instead, group does exactly what it says. It collects all records with the same value for the provided key together into a bag.
  - After Grouping, the first field of the tuple will be given the name **'group'** and has the value on which the grouping has been performed, and the second field will have the same name as the **'relation'** on which group is performed. It will be transformed into a bag containing the tuples belonging to that group.

- **Example:**

```
daily_data = load 'StockData' as (stock_name, stock_price);
groupd_stock = group daily_data by stock_name;
count = foreach groupd_stock generate group, COUNT(daily_data);
```



ORDER BY OPERATOR

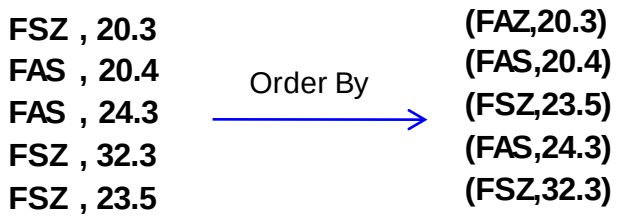
- **Order by**
  - The *order* statement does sorting on your data. Its use is similar to Group by.
  - Data is sorted based on the types of the indicated fields
    - chararray fields are sorted in lexical order
    - Int / float / double are sorted numerically
    - Byte array fields are sorted lexically
    - For all data types, nulls are taken to be smaller than all possible values
    - Sorting by maps, tuples, or bags produces errors

- Example:

daily_data = load 'StockData' as (stock_name, stock_price);

byprice = order daily_data by stock_price;

Dump byprice;



"order daily_data by stock_price desc" – will sort it in descending order

JOIN OPERATOR

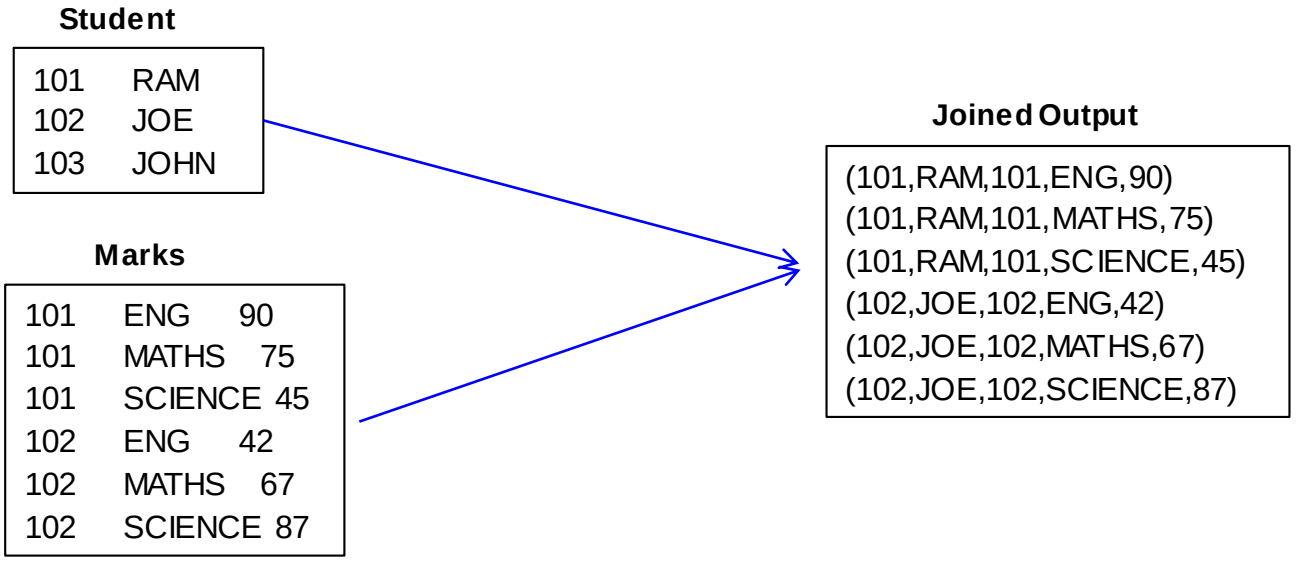
- **Join**
  - Join selects records from one input to put together with records from another input.
  - This is done by indicating keys for each input. When those keys are equal the two rows are joined. Records for which no match is found are dropped.
  - You can do joining on mutiple keys also if you have common keys on different relations.

- **Example**

A =load 'StudentForJoin';

B =load 'MarksForJoin';

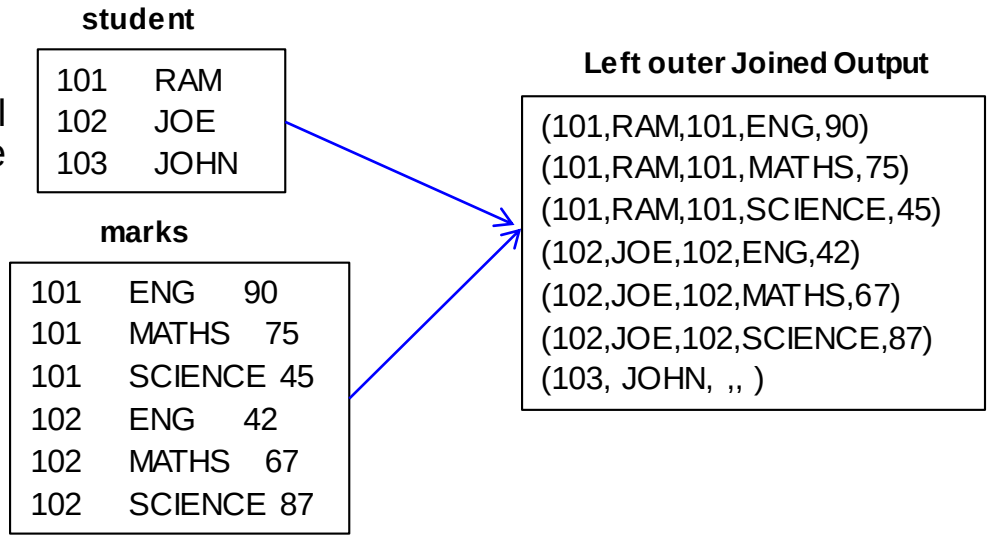
C =Join A by $0, B by $0;



JOIN OPERATOR

- **Left Outer joins:**
  - In outer joins records that do not have a match on the other side are included, with null values being filled in for the missing fields
  - A left outer join means records from the left side will be included even if they do not have a match on the right side.

```
A =Load 'StudentForJoin' AS (id:int, name:chararray);
B =Load 'MarksForJoin' AS (id:int,
   subject:chararray,marks:int);
C =Join A by id left outer, B by id;
Dump C;
```



- **Right Outer joins**:
  - Right outer joins means records from the right side will be included even if they do not have a match on the left side.

- **Full Outer joins:**
  - A full outer join means records from both sides are taken even when they do not have matches.

Other Operators

- COGROUP
- CROSS
- DISTINCT
- LIMIT
- SAMPLE
- SPLIT
- STORE
- STREAM
- UNION
- DESCRIBE
- EXPLAIN
- ILLUSTRATE
- Eval Functions (AVG , CONCAT COUNT COUNT_STAR DIFF IsEmpty MAX MIN SIZE SUM TOKENIZE)

Agenda

- Pig – An Overview
- Pig Data Model
- Introduction to Pig Latin
- **Executing Pig**
- Optimizing Pig
- Control Stuctures
- Pig Examples
- Exercises

Executing Pig

- **Local Mode**
  - Running Pig locally on your machine is referred to in Pig parlance as *local mode*. Local mode is useful for prototyping and debugging your Pig Latin scripts.
  - Pig scripts are generally stored with a .pig extension.
  - (TestScript.pig)

```
daily_data = load 'stock_data.csv' as (stock_name, stock_price);
byprice = order daily_data by stock_price;
Dump byprice;
```

  - $PIG_HOME/bin/pig –x local (pig_file_to_be_executed)
  - $PIG_HOME/bin/pig –x local TestScript.pig

- **Distributed Mode**
  - Running Pig distributed implies running on your hadoop cluster.
  - To run the pig on hadoop, data input files need to be present on hdfs
  - $PIG_HOME/bin/pig TestScript.pig (if you don't provide –x local, pig automatically run on hadoop)

Executing Pig

- **Grunt Shell**
  - Pig also comes with a interactive shell, called **'Grunt'.** It enables users to enter Pig Latin interactively, as well as provides a shell for users to interact with HDFS or local filesystem.
  - To execute Grunt Shell to use hadoop cluster:

$PIG_HOME/bin/pig

[main] INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop file system at: hdfs://lxe9700:54310

[main] INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to map-reduce job tracker at: lxe9700:54311

Grunt >

Grunt > A = Load 'Table.txt' as (name:chararray, marks:int);

Grunt > B = Group A by name;

Grunt > Dump B;

  - Grunt will look for each command and prompts, if it find any syntactical errors.
  - When Grunt finds DUMP or STORE command, then only your script get executed.
  - To execute Grunt shell on local mode:

$PIG_HOME/bin/pig –x local

Setting properties in pig

- Pig supports a number of Java properties that you can use to customize Pig behavior. You can retrieve a list of the properties using the help properties command. All of these properties are optional
  - $ pig -help properties

  The following properties are supported:

- Logging:
  - verbose=true|false; default is false.
    brief=true|false; default is false.
    debug=OFF|ERROR|WARN|INFO|DEBUG;
    aggregate.warning=true|false

- Performance tuning:
  - pig.cachedbag.memusage=<mem fraction>;
  - pig.skewedjoin.reduce.memusagea=<mem fraction>;
  - pig.exec.nocombiner=true|false;
  - opt.multiquery=true|false;
  - pig.tmpfilecompression=true|false;
  - pig.tmpfilecompression.codec=lzo|gzip…etc

- Miscellaneous:
  - exectype=mapreduce|local..etc

Setting properties in pig

- The set command
  - Assigns values to keys used in Pig.
  - All Pig and Hadoop properties can be set, either in the Pig script or via the Grunt command line.

Syntax	Example

set key 'value'	`grunt> SET debug 'on'`
	`grunt> SET job.name 'my job'`
	`grunt> SET default_parallel 100`

Shell Commands

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

Syntax	Example

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

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

Syntax	Example

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

Shell Commands - Utility

- **clear**
  - Clear the screen of Pig grunt shell and position the cursor at top of the screen.
- **exec**
  - Run a Pig script.
- **help**
  - Prints a list of Pig commands or properties.
- **quit**
  - Quits from the Pig grunt shell.
- **kill**
  - Kills a job.
- **history**
  - Display the list of statements used so far.

Agenda

- Pig – An Overview
- Pig Data Model
- Introduction to Pig Latin
- Executing Pig
- **Optimizing Pig**
- Control Stuctures
- Pig Examples
- Exercises

Pig Optimizations

- Before diving into the details of how to optimize your Pig Latin, it is worth understanding what items tend to create bottlenecks in Pig jobs:

- **Input size**
  - Hadoop's parallelism reduces I/O bound but does not entirely remove it. You can always add more map tasks. Additional maps take more time to start up, and MapReduce has to find more slots in which to run them. If you have twice as many maps as you have slots to run them, it will take twice your average map time to run all of your maps.

- **Shuffle size**
  - The data that is moved from your map tasks to your reduce tasks. All of this data has to be serialized, sorted, moved over the network, merged, and deserialized. Also, the number of maps and reduces matters. So if there are m maps and r reduces, the shuffle will have m x r networkconnections.

- **Output size**
  - Every record written out by a MapReduce job has to be serialized, possibly com-pressed, and written to the store. When the store is HDFS, it must be written to three separate machines before it is considered written.

- **Intermediate results size**
  - Pig moves data between MapReduce jobs by storing it in HDFS. Thus the size of these intermediate results is affected by the input size and output size factors mentioned previously.

- **Memory**
  - Some calculations require your job to hold a lot of information in memory, for example, joins. If Pig cannot hold all of the values in memory simultaneously, it will need to spill some to disk. This causes a significant slowdown, as records must be written to and read from disk, possibly multiple times.

Pig Optimizations

Use Optimization

- Pig supports various optimization rules which are turned on by default.
  - » The pig.optimizer.rules.disabled pig property, which accepts a comma-separated list of optimization rules to disable; the all keyword disables all non-mandatory optimizations. (e.g.: set pig.optimizer.rules.disabled 'ColumnMapKeyPrune';)
  - » The -t, -optimizer_off command-line options. (e.g.: pig -optimizer_off [opt_rule | all])

Use Types

- If types are not specified in the load statement, Pig assumes the type of =double= for numeric computations. A lot of the time, your data would be much smaller, maybe, integer or long.
- Query 1
- ```
  A = load 'myfile' as (t, u, v);
  B = foreach A generate t + u;
  ```
- Query 2
- ```
  A = load 'myfile' as (t: int, u: int, v);
  B = foreach A generate t + u;
  ```
- The second query will run more efficiently than the first. In some of our queries with see 2x speedup.

Pig Optimizations

- **Project Early and Often**
  - Pig does not (yet) determine when a field is no longer needed and drop the field from the row. For example, say you have a query like:
  -
```
A = load 'myfile' as (t, u, v);
B = load 'myotherfile' as (x, y, z);
C = join A by t, B by x;
D = group C by u;
E = foreach D generate group, COUNT($1);
```

- There is no need for v, y, or z to participate in this query. And there is no need to carry both t and x past the join, just one will suffice. Changing the query above to the query below will greatly reduce the amount of data being carried through the map and reduce phases by pig.
  -

```
A = load 'myfile' as (t, u, v);
A1 = foreach A generate t, u;
B = load 'myotherfile' as (x, y, z);
B1 = foreach B generate x;
C = join A1 by t, B1 by x;
C1 = foreach C generate t, u;
D = group C1 by u;
E = foreach D generate group, COUNT($1);
```

Pig Optimizations

- **Filter Early and Often**
  - As with early projection, in most cases it is beneficial to apply filters as early as possible to reduce the amount of data flowing through the pipeline.
  - Query 1
  - ```
    A = load 'myfile' as (t, u, v);
    B = load 'myotherfile' as (x, y, z);
    C = filter A by t == 1;
    D = join C by t, B by x;
    E = group D by u;
    F = foreach E generate group, COUNT($1);
    ```
  - Query 2
  - ```
    A = load 'myfile' as (t, u, v);
    B = load 'myotherfile' as (x, y, z);
    C = join A by t, B by x;
    D = group C by u;
    E = foreach D generate group, COUNT($1);
    F = filter E by C.t == 1;
    ```
  - The first query is clearly more efficient than the second one because it reduces the amount of data going into the join.

Pig Optimizations

- **Drop Nulls Before a Join**

- With the introduction of nulls, join and cogroup semantics were altered to work with nulls. The semantic for cogrouping with nulls is that nulls from a given input are grouped together, but nulls across inputs are not grouped together.

- Since flattening an empty bag results in an empty row, in a standard join the rows with a null key will always be dropped.
  - ```
    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;
    ```
  - Here the nulls will be dropped before the join. Since all null keys go to a single reducer, if your key is null even a small percentage of the time the gain can be significant.
  - In one test where the key was null 7% of the time and the data was spread across 200 reducers, we saw a about a 10x speed up in the query by adding the early filters.
    ```

Pig Optimizations

- **Prefer DISTINCT over GROUP BY - GENERATE**
- Using DISTINCT
  - ```
    A = load 'myfile' as (t, u, v);
    B = foreach A generate u;
    C = distinct B; dump C;
    ```
- In pig initially, DISTINCT is just GROUP BY/PROJECT under the hood.
- From pig 0.2.0 it is not, and it is much faster and more efficient (depending on your key cardinality, up to 20x faster in pig team's tests). Therefore, the use of DISTINCT is recommended over GROUP BY - GENERATE.
    ```

Pig Optimizations

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

- **Use the PARALLEL Clause**
  - Use the PARALLEL clause to increase the parallelism of a job:
  - PARALLEL sets the number of reduce tasks for the MapReduce jobs generated by Pig. The default value is 1 (one reduce task).
  - PARALLEL only affects the number of reduce tasks. Map parallelism is determined by the input file, one map for each HDFS block.
  - If you don't specify PARALLEL, you still get the same map parallelism but only one reduce task.

- **Take Advantage of Join Optimizations**
  - **Regular Join Optimizations**
    - » Optimization for regular joins ensures that the last table in the join is not brought into memory but streamed through instead. Optimization reduces the amount of memory used which means you can avoid spilling the data and also should be able to scale your query to larger data volumes.
    - »
```
small = load 'small_file' as (t, u, v);
large = load 'large_file' as (x, y, z);
C = join small by t, large by x;
```

Pig Optimizations – Specialized Joins

- **Replicated Joins**
  - Fragment replicate join is a special type of join that works well if one or more relations are small enough to fit into main memory. In such cases, Pig can perform a very efficient join because all of the hadoop work is done on the map side. In this type of join the large relation is followed by one or more small relations
  - In this example, a large relation is joined with two smaller relations. Note that the large relation comes first followed by the smaller relations; and, all small relations together must fit into main memory, otherwise an error is generated.
  - ```
    big = LOAD 'big_data' AS (b1,b2,b3);

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

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

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

Pig Optimizations – Specialized Joins

- **Skewed Joins**
  - If the underlying data is sufficiently skewed, load imbalances will swamp any of the parallelism gains. In order to counteract this problem, skewed join computes a histogram of the key space and uses this data to allocate reducers for a given key.
  - Skewed join does not place a restriction on the size of the input keys. It accomplishes this by splitting the left input on the join predicate and streaming the right input. The left input is sampled to create the histogram.
  - ```
    big = LOAD 'big_data' AS (b1,b2,b3);
    massive = LOAD 'massive_data' AS (m1,m2,m3);
    C = JOIN big BY b1, massive BY m1 USING 'skewed';
    ```
- **Merge Joins**
  - Often user data is stored such that both inputs are already sorted on the join key. In this case, it is possible to join the data in the map phase of a MapReduce job. This provides a significant performance improvement compared to passing all of the data through unneeded sort and shuffle phases.
  - Pig has implemented a merge join algorithm, or sort-merge join. It works on pre-sorted data, and does not sort data for you
  - ```
    C = JOIN A BY a1, B BY b1, C BY c1 USING 'merge';
    ```

Pig Optimizations – Specialized Joins

- **Merge-Sparse Joins**
  - Merge-Sparse join is a specialization of merge join. Merge-sparse join is intended for use when one of the tables is very sparse, meaning you expect only a small number of records to be matched during the join.
  - In tests this join performed well for cases where less than 1% of the data was matched in the join.
  - ```
    a = load 'sorted_input1' using
    org.apache.pig.piggybank.storage.IndexedStorage('\t', '0');
    b = load 'sorted_input2' using
    org.apache.pig.piggybank.storage.IndexedStorage('\t', '0');
    c = join a by $0, b by $0 using 'merge-sparse';
    store c into 'results';
    ```
    ```

Agenda

- Pig – An Overview
- Pig Data Model
- Introduction to Pig Latin
- Executing Pig
- Optimizing Pig
- **Control Stuctures**
- Pig Examples
- Exercises

Control Structures

- To enable control flow, you can embed Pig Latin statements and Pig commands in the Python and JavaScript scripting languages using a JDBC-like compile, bind, run model.
  - For Python, make sure the Jython jar is included in your class path.
  - For JavaScript, make sure the Rhino jar is included in your classpath.

- **Invocation Process**
  - You invoke Pig in the host scripting language through an embedded Pig object.

- **Compile:**
  - Compile is a static function on the Pig object and in its simplest form takes a fragment of Pig Latin that defines the pipeline as its input:
  - ```
    # COMPILE: compile method returns a Pig object that represents the pipeline
    P = Pig.compile("""A = load '$in'; store A into '$out';""")
    ```

- **Bind**: Resolve the parameters during the bind call.
  - ```
    input = "original"
    output = "output"

    # BIND: bind method binds the variables with the parameters in the pipeline
    and returns a BoundScript object
    Q = P.bind({'in':input, 'out':output})
    ```

Control Structures contd…

- Please note that all parameters must be resolved during bind. Having unbound parameters while running your script is an error. Also note that even if your script is fully defined during compile, bind without parameters still must be called.

- **Run:** Bind call returns an instance of BoundScript object that can be used to execute the pipeline. The simplest way to execute the pipeline is to call runSingle function. (However, as mentioned later, this works only if a single set of variables is bound to the parameters. Otherwise, if multiple set of variables are bound, an exception will be thrown if runSingle is called.)
  - `result = Q.runSingle()`
  - The function returns a PigStats object that tells you whether the run succeeded or failed. In case of success, additional run statistics are provided.

Agenda

- Pig – An Overview
- Pig Data Model
- Introduction to Pig Latin
- Executing Pig
- Pig Examples
- Exercises

Word Count in Pig

```
lines = LOAD 'WordCount' as(line:chararray);
tokens = FOREACH lines GENERATE TOKENIZE(line) as token;
words = FOREACH tokens generate FLATTEN(token) as word;
word_group = group words by word;
word_count = foreach word_group generate group, COUNT(words) as num;
dump word_count;
```

TOKENIZE splits the line into a field for each word.

FLATTEN will take the collection of records returned by TOKENIZE and produce a separate record for each one, calling the single field in the record word.

Now group them together by each word.

WordCount.pig - Illustrated

- **lines = load 'data' as(line:chararray);**

- **Dump**:
  - (Apache Hadoop Training)
  - (Training on Pig)
  - (Pig is part of Hadoop)
  - (Hadoop is part of Apache)

lines	line:chararray
	Pig is part of Hadoop Hadoop is part of Apache

- **tokens = foreach lines generate TOKENIZE(line) as token;**

- **Dump:**
  - ({(Apache),(Hadoop),(Training)})
  - ({(Training),(on),(Pig)})
  - ({(Pig),(is),(part),(of),(Hadoop)})
  - ({(Hadoop),(is),(part),(of),(Apache)})

tokens	token:bag{tuple_of_tokens:tuple(token:chararray)}
	{(Pig), ..., (Hadoop)} {(Hadoop), ..., (Apache)}

WordCount.pig - Illustrated

- **words = foreach tokens generate flatten(token) as word;**

- **Dump:**
  - (Apache)
  - (Hadoop)
  - (Training)
  - (Training)
  - (on)
  - (Pig)
  - (Pig)
  - (is)
  - (part)
  - (of)
  - (Hadoop)
  - (Hadoop)
  - (is)
  - (part)
  - (of)
  - (Apache)

words	word:chararray
	Pig is part hadoop

WordCount.pig - Illustrated

- **word_group = group words by word;**

- **Dump:**
  - (is,{(is),(is)})
  - (of,{(of),(of)})
  - (on,{(on)})
  - (Pig,{(Pig),(Pig)})
  - (part,{(part),(part)})
  - (Hadoop,{(Hadoop),(Hadoop),(Hadoop)})
  - (Apache,{(Apache),(Apache)})
  - (Training,{(Training),(Training)})

word_group	group:chararray	words:bag{:tuple(word:chararray)}
	Hadoop	{(Hadoop), (Hadoop), (Hadoop)}
	Apache	{(Apache),(Apache)}

- **word_count = foreach word_group generate group, COUNT(words) as num;**

- **Dump:**
  - (is,2)
  - (of,2)
  - (on,1)

word_count	group:chararray	num:long
	is	2
	on	1

Writing UDFs in Pig

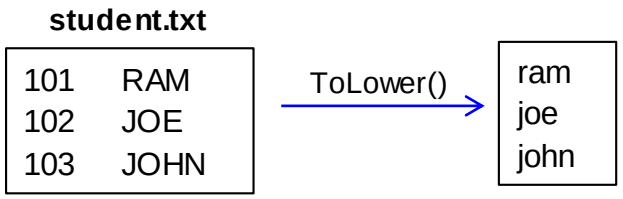
- Sample UDF for changing the fields into lower (ToLower)

```
public class ToLower extends EvalFunc<String> {
    public String exec(Tuple input) throws IOException {                    // Providing tuple as input
        if(input == null || input.size() == 0)
            return null;
        try{
            String query = (String)input.get(0);
            return query.toLowerCase().trim();
        }catch(Exception e){
            System.err.println("ToLower:  failed to process input;error - " + e.getMessage());
            return null;
        }
    }
//Generating the output schema
public Schema outputSchema(Schema input) {
        return new Schema(new Schema.FieldSchema(getSchemaName(this.getClass().getName().toLowerCase(),  input),
DataType.CHARARRAY));
    }

// Function overloading . Making sure function should run for bytearray as well as chararray
public List<FuncSpec> getArgToFuncMapping() throws FrontendException {
        List<FuncSpec> funcList = new ArrayList<FuncSpec>();
        funcList.add(new FuncSpec(this.getClass().getName(), new Schema(new Schema.FieldSchema(null, DataType.CHARARRAY))));
        return funcList;
    }
```

Running UDFs

- Compiling code :
  - Your UDF requires your pig.jar as well as hadoop-core jar to compile.
  - Assuming your code is in /tmp/myUDFs/ToLower.java
    - *javac -classpath : $PIG_HOME/pig-without-hadoop.jar: $HADOOP_HOME/hadoop-core-0.20.205.jar ToLower.java –d classes/*
  - If your code is fine, it should compile smoothly.
  - To make jar file
    - *jar –cvf ToLower.jar –C classes/ .*
  - To use the jar in the pig scripts (running a sample script in local grunt shell)
    - *$ pig –x local*
    - *Grunt > Register '/tmp/myUDFs/ToLower.jar';*
    - *Grunt > Load A = 'student.txt';*
    - *Grunt > B = FOREACH A generate ToLower($1);*
    - *Grunt > DUMP B;*



What are types ?

- Type values are the static values assigned by Pig Data types internally for different data types
  - public static final byte UNKNOWN = 0;
  - public static final byte NULL = 1;
  - public static final byte BOOLEAN = 5;
  - public static final byte BYTE = 6;
  - public static final byte INTEGER = 10;
  - public static final byte LONG = 15;
  - public static final byte FLOAT = 20;
  - public static final byte DOUBLE = 25;
  - public static final byte BYTEARRAY = 50;
  - public static final byte CHARARRAY = 55;
  - public static final byte MAP = 100;
  - public static final byte TUPLE = 110;
  - public static final byte BAG = 120;
  - public static final byte ERROR = -1;

Example – Passing schema inside pig scripts

- Folder – "/home/hadoop/Documents/Pig/IntroductionToPig/Datasets/PolicyInfo"
  - » Files - "PolicyInfo.txt" , ".pig-schema" (Notice the dot at .pig_schema)

```
(Austin,22121, Jan , Dec)
(Ram,341231, May, June)
(Joe,20232, Sept, Dec)
```

PolicyInfo.txt

```
{
"fields":
[
        {"name":"name",         "type":55} ,
        {"name":"policyid",     "type":15},
        {"name":"startmonth", "type":55},
        {"name":"endmonth", "type":55},

]
}
```

.pig_schema

- » $ pig – x local
- » Grunt > A = Load '/tmp/Data/PolicyInfo/PolicyInfo.txt' ; // Pig will automatically look for .pig_schema file and if // it find it, it uses the schema defined in it.
- » Grunt > B = Foreach A generate name , policyid;
- » Grunt > Dump B;

Running Pig Tutorials/Examples

- Pig Setup comes with a set of tutorials and UDFs for practice.Copy the pigtutorial.tar.gz from $PIG_HOME/tutorials/ to your local directory.
- Unzip the pigtutorial.tar.gz file.
  - *$ tar –xzf pigtutorial.tar.gz*
- A new directory named pigtmp is created. This directory contains the Pig tutorial scripts.
- The pig scripts can be run in both local and map-reduce mode
- To run the Pig Scripts in Local Mode:
  - From the pigtmp directory, execute the following command(using either script1-local.pig or script2-local.pig)
  - You can probably see inside those scripts and check out what they are doing inside.
  - These scripts are processing a search query log file from the Excite search engine and finds search phrases that occur with particular high frequency during certain times of the day.
  - *$ pig -x local script1-local.pig*
  - Results will be available in the script1-local-results.txt directory.
    - » *$ cd script1-local-results.txt*
    - » *$ nano part-r-00000*

Exercises

- **Level 1**
  - Consider Twitter_user_data (userid,username,timezone,friendcount)
  - Consider Tweet_data (tweet_id, tweet, userid, tweet_category)
    - » Count the total number of twitter users
    - » Dump userid+username+tweet+tweet_category
    - » Dump all positive tweets/negative tweets/neutral tweets.
    - » Dumping all users with friendcount > *_some_constant_*

- **Level 2**
  - Data files: cust_info , premium_data

    Schema for cust_info : (ID:int,name:chararray,region:chararray)
    Schema for premium_data:(ID:int, premium:float, start_year:int,end_year:int)
    - » 1. Make a .pig_schema file for the data.
    - » 2. For every customer ID find the gross premium ,name and region.
    - » 3. Find the top two customers who paid the highest gross premium. (hint: use the results of 1)
    - » 4. (OPTIONAL) Try 1 and 2 with -

      a).pig_schema

      b) without .pig_schema and

      c) with no schema defined in load statement

- **Level 3:**
  - Generate Pig UDFs such as Taking logs, Count, Row-wise Average

Useful Links

- Ebook
  - http://ofps.oreilly.com/titles/9781449302641/index.html
- http://pig.apache.org/docs/
- http://www.scribd.com/doc/31652181/Twitter-Pig-and-HBase-For-Bay-Area-Hadoop-User-Group-May-2010
- http://pig.apache.org/docs/r0.7.0/cookbook.html
- http://pig.apache.org/docs/r0.7.0/piglatin_ref2.html#DISTINCT
- http://pig.apache.org/docs/r0.9.1/cont.html

Thank You

Chicago, IL
Bangalore, India
December, 2013
www.mu-sigma.com