BIG DATA ANALYTICS

Pig Latin

https://www.youtube.com/c/RASINENIMADANAMOHANA

Pig Latin: *Outline*

- Structure
- Statements
- Expressions
- Types
- Schemas
- Functions
- Macros

Pig Latin: *Introduction*

- The language used to express data flows, called Pig Latin.
- The execution environment to run Pig Latin programs
  - ✓ local execution in a single JVM and
  - ✓ distributed execution on a Hadoop cluster
- A Pig Latin program is made up of a series of operations, or transformations, that are applied to the input data to produce output.
- Pig has two execution types or modes: local mode and MapReduce mode.

Pig Latin: *Introduction*

Local mode:

- In local mode, Pig runs in a single JVM and accesses the local filesystem.
- This mode is suitable only for small datasets and when trying out Pig.
- The execution type is set using the -x or -exectype option.
- To run in local mode, set the option to local:

```
% pig -x local
grunt>
```

This starts Grunt, the Pig interactive shell.

Pig Latin: *Introduction*

MapReduce mode:

- In MapReduce mode, Pig translates queries into MapReduce jobs and runs them on a Hadoop cluster.
- The cluster may be a pseudo or fully distributed cluster.
- MapReduce mode (with a fully distributed cluster) is what we use when we want to run Pig on large datasets.

Pig Latin: *Outline*

- Structure 
- Statements
- Expressions
- Types
- Schemas
- Functions
- Macros

Pig Latin: *Structure*

- A Pig Latin program consists of a collection of statements
- A statement can be thought of as an operation or a command

Example:

- A GROUP operation is a type of statement:

```
grouped_records = GROUP records BY year;
```

  Statements are usually terminated with a semicolon

- The command to list the files in a Hadoop filesystem is another example of a statement:

```
ls /
```

  ls command, on the other hand, does not have to be terminated with a semicolon.

Pig Latin: *Structure*

- As a general guideline, statements or commands for interactive use in Grunt do not need the terminating semicolon.
- This group includes the interactive Hadoop commands, as well as the diagnostic operators such as DESCRIBE.
- It's never an error to add a terminating semicolon, so if in doubt, it's simplest to add one.
- Statements that have to be terminated with a semicolon can be split across multiple lines for readability:

```
records = LOAD 'input/ncdc/micro-tab/sample.txt'
  AS (year:chararray, temperature:int, quality:int);
```

Pig Latin: *Structure*

- Pig Latin has two forms of comments.
- Double hyphens are used for single-line comments.
- Everything from the first hyphen to the end of the line is ignored by the Pig Latin interpreter:

```
-- My program
DUMP A; -- What's in A?
```

- C-style comments are more flexible since they delimit the beginning and end of the comment block with /* and */ markers. They can span lines or be embedded in a single line:

```
/*
* Description of my program spanning
* multiple lines.
*/
A = LOAD 'input/pig/join/A';
B = LOAD 'input/pig/join/B';
C = JOIN A BY $0, /* ignored */ B BY $1;
DUMP C;
```

Pig Latin: *Structure*

- Pig Latin has a list of keywords that have a special meaning in the language and cannot be used as identifiers.
- These include the operators (LOAD, ILLUSTRATE), commands (cat, ls), expressions (matches, FLATTEN), and functions (DIFF, MAX)
- Pig Latin has mixed rules on case sensitivity.
- Operators and commands are not case sensitive (to make interactive use more forgiving); however, aliases and function names are case sensitive.

Pig Latin: *Outline*

- Structure
- Statements 
- Expressions
- Types
- Schemas
- Functions
- Macros

Pig Latin: *Statements*

- As a Pig Latin program is executed, each statement is parsed in turn.
- If there are syntax errors or other (semantic) problems, such as undefined aliases, the interpreter will halt and display an error message.
- The interpreter builds a logical plan for every relational operation, which forms the core of a Pig Latin program.
- The logical plan for the statement is added to the logical plan for the program so far, and then the interpreter moves on to the next statement.

Pig Latin: *Statements*

- It's important to note that no data processing takes place while the logical plan of the program is being constructed.
- For example, consider the following Pig Latin program:

```
-- max_temp.pig: Finds the maximum temperature by year
records = LOAD 'input/ncdc/micro-tab/sample.txt'
AS (year:chararray, temperature:int, quality:int);

filtered_records = FILTER records BY temperature != 9999 AND
quality IN (0, 1, 4, 5, 9);

grouped_records = GROUP filtered_records BY year;

max_temp = FOREACH grouped_records GENERATE group,

MAX(filtered_records.temperature);

DUMP max_temp;
```

Pig Latin: *Statements*

Example: cont'd.

- When the Pig Latin interpreter encounters the first line with the LOAD statement, it checks its syntax and semantics, adds it to the logical plan, but does not load the data or verify the file's existence.
- Loading data prematurely is impractical because not all input data might be needed due to later filtering. Processing only begins when the entire flow is defined.
- Similarly, GROUP and FOREACH...GENERATE statements are validated and added to the logical plan without execution.
- Execution is triggered by the DUMP statement, which compiles the logical plan into a physical plan, executing it as a series of MapReduce jobs. In local mode, these jobs run in the local JVM; in MapReduce mode, they run on a Hadoop cluster.

Pig Latin: *Statements*

- The relational operators that can be a part of a logical plan in Pig are summarized in below Table-1 (*Pig Latin relational operators*):

Category	Operator	Description
Loading and storing	LOAD	Loads data from the filesystem or other storage into a relation
	STORE	Saves a relation to the filesystem or other storage
	DUMP (\d)	Prints a relation to the console
Filtering	FILTER	Removes unwanted rows from a relation
	DISTINCT	Removes duplicate rows from a relation
	FOREACH...GENERATE	Adds or removes fields to or from a relation
	MAPREDUCE	Runs a MapReduce job using a relation as input
	STREAM	Transforms a relation using an external program
	SAMPLE	Selects a random sample of a relation
	ASSERT	Ensures a condition is true for all rows in a relation; otherwise, fails

Pig Latin: *Statements*

Table-1: *Pig Latin relational operators*: cont'd.

Category	Operator	Description
Grouping and joining	JOIN	Joins two or more relations
	COGROUP	Groups the data in two or more relations
	GROUP	Groups the data in a single relation
	CROSS	Creates the cross product of two or more relations
	CUBE	Creates aggregations for all combinations of specified columns in a relation
Sorting	ORDER	Sorts a relation by one or more fields
	RANK	Assign a rank to each tuple in a relation, optionally sorting by fields first
	LIMIT	Limits the size of a relation to a maximum number of tuples
Combining and splitting	UNION	Combines two or more relations into one
	SPLIT	Splits a relation into two or more relations

Pig Latin: *Statements*

Table-2: *Pig Latin diagnostic operators*:

Operator (Shortcut)	Description
DESCRIBE (\de)	Prints a relation's schema
EXPLAIN (\e)	Prints the logical and physical plans
ILLUSTRATE (\i)	Shows a sample execution of the logical plan, using a generated subset of the input

Table-3: *Pig Latin macro and UDF statements*:

Statement	Description
REGISTER	Registers a JAR file with the Pig runtime
DEFINE	Creates an alias for a macro, UDF, streaming script, or command specification
IMPORT	Imports macros defined in a separate file into a script

Pig Latin: *Statements*

Table-4: *Pig Latin commands*:

Category	Command	Description
Hadoop filesystem	cat	Prints the contents of one or more files
	cd	Changes the current directory
	copyFromLocal	Copies a local file or directory to a Hadoop filesystem
	copyToLocal	Copies a file or directory on a Hadoop filesystem to the local filesystem
	cp	Copies a file or directory to another directory
	fs	Accesses Hadoop's filesystem shell
	ls	Lists files
	mkdir	Creates a new directory
	mv	Moves a file or directory to another directory
	pwd	Prints the path of the current working directory
	rm	Deletes a file or directory
	rmf	Forcibly deletes a file or directory (does not fail if the file or directory does not exist)

Pig Latin: *Statements*

Table-4: *Pig Latin commands*: cont'd.

Category	Command	Description
Hadoop MapReduce	kill	Kills a MapReduce job
Utility	clear	Clears the screen in Grunt
	exec	Runs a script in a new Grunt shell in batch mode
	help	Shows the available commands and options
	history	Prints the query statements run in the current Grunt session
	quit (\q)	Exits the interpreter
	run	Runs a script within the existing Grunt shell
	set	Sets Pig options and MapReduce job properties
	sh	Runs a shell command from within Grunt

Pig Latin: *Statements*

- The filesystem commands can operate on files or directories in any Hadoop filesystem, and they are very similar to the hadoop fs commands (which is not surprising, as both are simple wrappers around the Hadoop FileSystem interface).
- We can access all of the Hadoop filesystem shell commands using Pig's fs command.

For example,

```
fs -ls
```

will show a file listing, and

```
fs -help
```

will show help on all the available commands.

Pig Latin: *Outline*

- Structure
- Statements
- Expressions 
- Types
- Schemas
- Functions
- Macros

Pig Latin: *Expressions*

- An expression is something that is evaluated to yield a value.
- Expressions can be used in Pig as a part of a statement containing a relational operator.
- Pig has a rich variety of expressions, many of which will be familiar from other programming languages.
- Pig Latin expressions are listed in *below Table*, with brief descriptions and examples.

Pig Latin: *Expressions*

Table: *Pig Latin expressions*:

Category	Expressions	Description	Examples
Constant	`Literal`	Constant value	`1.0, 'a'`
Field (by position)	`$n`	Field in position n (zero-based)	`$0`
Field (by name)	`f`	Field named f	`year`
Field (disambiguate)	`r::f`	Field named f from relation r after grouping or joining	`A::year`
Projection	`c.$n, c.f`	Field in container c (relation, bag, or tuple) by position, by name	`records.$0, records.year`
Map lookup	`m#k`	Value associated with key k in map m	`items#'Coat'`
Cast	`(t) f`	Cast of field f to type t	`(int) year`
Arithmetic	`x + y, x - y`	Addition, subtraction	`$1 + $2, $1 - $2`
	`x * y, x / y`	Multiplication, division	`$1 * $2, $1 / $2`
	`x % y`	Modulo, the remainder of x divided by y	`$1 % $2`
	`+x, -x`	Unary positive, negation	`+1, -1`

Pig Latin: *Expressions*

Table: *Pig Latin expressions*: cont'.

Category	Expressions	Description	Examples
Conditional	`x ? y : z`	Bincond/ternary; y if x evaluates to true, z otherwise	`quality == 0 ? 0 : 1`
	`CASE`	Multi-case conditional	`CASE q WHEN 0 THEN 'good' ELSE 'bad' END`
Comparison	`x == y, x != y`	Equals, does not equal	`quality == 0, temperature != 9999`
	`x > y, x < y`	Greater than, less than	`quality > 0, quality < 10`
	`x >= y, x <= y`	Greater than or equal to, less than or equal to	`quality >= 1, quality <= 9`
	`x matches y`	Pattern matching with regular expression	`quality matches '[01459]'`
	`x is null`	Is null	`temperature is null`
	`x is not null`	Is not null	`temperature is not null`

Pig Latin: *Expressions*

Table: *Pig Latin expressions*: cont'.

Category	Expressions	Description	Examples
Boolean	`x OR y`	Logical OR	`q == 0 OR q == 1`
	`x AND y`	Logical AND	`q == 0 AND r == 0`
	`NOT x`	Logical negation	`NOT q matches '[01459]'`
	`IN x`	Set membership	`q IN (0, 1, 4, 5, 9)`
Functional	`fn(f1,f2,...)`	Invocation of function *fn* on fields *f1*, *f2*, etc.	`isGood(quality)`
Flatten	`FLATTEN(f)`	Removal of a level of nesting from bags and tuples	`FLATTEN(group)`

Pig Latin: *Outline*

- Structure
- Statements
- Expressions
- Types 
- Schemas
- Functions
- Macros

Pig Latin: *Types*

- Pig has a boolean type and six numeric types: `int`, `long`, `float`, `double`, `biginteger`, and `bigdecimal`, which are identical to their Java counterparts.
- There is also a `bytearray` type, like Java's byte array type for representing a blob of binary data, and chararray, which, like `java.lang.String`, represents textual data in `UTF-16` format (although it can be loaded or stored in `UTF-8` format).
- The `datetime` type is for storing a date and time with millisecond precision and including a time zone.
- Pig does not have types corresponding to Java's byte, short, or char primitive types. These are all easily represented using Pig's `int` type, or chararray for `char`.
- The Boolean, numeric, textual, binary, and temporal types are simple atomic types.
- Pig Latin also has three complex types for representing nested structures: `tuple`, `bag`, and `map`.

Pig Latin: *Types*

- All of Pig Latin's types are listed in *below Table (Pig Latin types)*:

Category	Type	Description	Literal Example
Boolean	`boolean`	True/false value	`true`
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`
	`biginteger`	Arbitrary-precision integer	`'10000000000'`
	`bigdecimal`	Arbitrary-precision signed decimal number	`'0.11000100000000000000000001'`
Text	`chararray`	Character array in UTF-16 format	`'a'`
Binary	`bytearray`	Byte array	`Not supported`
Temporal	`datetime`	Date and time with time zone	`Not supported, use ToDate built-in function`

Pig Latin: *Types*

Table: Pig Latin types cont'd.

Category	Type	Description	Literal Example
Complex	`tuple`	Sequence of fields of any type	`(1,'pomegranate')`
	`bag`	Unordered collection of tuples, possibly with duplicates	`{(1,'pomegranate'),(2)}`
	`map`	Set of key-value pairs; keys must be character arrays, but values may be any type	`['a'#'pomegranate']`

- The complex types are usually loaded from files or constructed using relational operators.
- The raw form in a file is usually different when using the standard `PigStorage` loader.
- For example, the representation in a file of the `bag` complex type as shown in the above Table would be `{(1,pomegranate),(2)}` (note the lack of quotation marks), and with a suitable schema, this would be loaded as a relation with a single field and row, whose value was the bag.

Pig Latin: *Types*

- Pig provides the built-in functions `TOTUPLE`, `TOBAG`, and `TOMAP`, which are used for turning expressions into tuples, bags, and maps.
- Although relations and bags are conceptually the same (unordered collections of tuples), in practice Pig treats them slightly differently. A relation is a top-level construct, whereas a bag has to be contained in a relation. But there are a few restrictions that can trip up the uninitiated.
- For example, it's not possible to create a relation from a bag literal. So, the following statement fails:

```
A = {(1,2),(3,4)}; -- Error
```

The simplest workaround in this case is to load the data from a file using the `LOAD` statement.

Pig Latin: *Types*

- Another example, we can't treat a relation like a bag and project a field into a new relation (`$0` refers to the first field of `A`, using the positional notation):

```
B = A.$0;
```

Instead, we have to use a relational operator to turn the relation `A` into relation `B`:

```
B = FOREACH A GENERATE $0;
```

It's possible that a future version of Pig Latin will remove these inconsistencies and treat relations and bags in the same way.

Pig Latin: *Outline*

- Structure
- Statements
- Expressions
- Types
- Schemas 
- Functions
- Macros

Pig Latin: *Schemas*

- A relation in Pig may have an associated schema, which gives the fields in the relation names and types.
- See the following example, how an `AS` clause in a `LOAD` statement is used to attach a schema to a relation:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>> AS (year:int, temperature:int, quality:int);
grunt> DESCRIBE records;
Output:
records: {year: int,temperature: int,quality: int}
```

Pig Latin: *Schemas*

- Pig's flexibility in the degree to which schemas are declared contrasts with schemas in traditional SQL databases, which are declared before the data is loaded into the system.
- Pig is designed for analyzing plain input files with no associated type information, so it is quite natural to choose types for fields later than we would with an RDBMS.
- It's possible to omit type declarations completely, too:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>> AS (year, temperature, quality);
grunt> DESCRIBE records;
Output:
records: {year: bytearray,temperature: bytearray,quality:
bytearray}
```

Pig Latin: *Schemas*

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>> AS (year, temperature, quality);
grunt> DESCRIBE records;
Output:
records: {year: bytearray,temperature: bytearray,quality:
bytearray}
```

In this case, we have specified only the names of the fields in the schema: year, temperature, and quality. The types default to `bytearray`, the most general type, representing a binary string.

Pig Latin: *Schemas*

- We don't need to specify types for every field; we can leave some to default to bytearray, as we have done for year in this declaration:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>> AS (year, temperature:int, quality:int);
grunt> DESCRIBE records;
Output:
records: {year: bytearray,temperature: int,quality: int}
```

  - However, if we specify a schema in this way, we do need to specify every field. Also, there's no way to specify the type of a field without specifying the name.

Pig Latin: *Schemas*

- On the other hand, the schema is entirely optional and can be omitted by not specifying an `AS` clause:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
grunt> DESCRIBE records;
Output:
Schema for records unknown
```

Pig Latin: *Schemas*

- Fields in a relation with no schema can be referenced using only positional notation: `$0` refers to the first field in a relation, `$1` to the second, and so on. Their types default to `bytearray`:

```
grunt> projected_records = FOREACH records GENERATE $0, $1, $2;
grunt> DUMP projected_records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
grunt> DESCRIBE projected_records;
projected_records: {bytearray,bytearray,bytearray}
```

Pig Latin: *Outline*

- Structure
- Statements
- Expressions
- Types
- Schemas
- Functions 
- Macros

Pig Latin: *Functions*

Functions in Pig come in four types:

1. *Eval* function
2. *Filter* function
3. *Load* function
4. *Store* function

Pig Latin: *Functions*

1. *Eval* function

- A function that takes one or more expressions and returns another expression.
- An example of a built-in eval function is `MAX`, which returns the maximum value of the entries in a bag.
- Some eval functions are `aggregate functions`, which means they operate on a bag of data to produce a scalar value; `MAX` is an example of an aggregate function.
- Furthermore, many aggregate functions are `algebraic`, which means that the result of the function may be calculated incrementally.
- In MapReduce terms, algebraic functions make use of the combiner and are much more efficient to calculate.
- `MAX` is an algebraic function, whereas a function to calculate the median of a collection of values is an example of a function that is not algebraic.

Pig Latin: *Functions*

2. Filter function

- A special type of eval function that returns a logical Boolean result.
- As the name suggests, filter functions are used in the `FILTER` operator to remove unwanted rows.
- They can also be used in other relational operators that take Boolean conditions, and in general, in expressions using Boolean or conditional expressions.
- An example of a built-in filter function is `IsEmpty`, which tests whether a bag or a map contains any items.

Pig Latin: *Functions*

3. `Load` function

- A function that specifies how to load data into a relation from external storage.

4. `Store` function

- A function that specifies how to save the contents of a relation to external storage.
- Often, load and store functions are implemented by the same type.
- For example, `PigStorage`, which loads data from delimited text files, can store data in the same format.

Pig Latin: *Functions*

Pig comes with a collection of built-in functions, a selection of which are listed in below Table: *Table. A selection of Pig's built-in functions*

Category	Function	Description
Eval	AVG	Calculates the average (mean) value of entries in a bag.
	CONCAT	Concatenates byte arrays or character arrays together.
	COUNT	Calculates the number of non-null entries in a bag.
	COUNT_STAR	Calculates the number of entries in a bag, including those that are null.
	DIFF	Calculates the set difference of two bags. If the two arguments are not bags, returns a bag containing both if they are equal; otherwise, returns an empty bag.
	MAX	Calculates the maximum value of entries in a bag.
	MIN	Calculates the minimum value of entries in a bag.

Pig Latin: *Functions*

Table. A selection of Pig's built-in functions cont'd.

Category	Function	Description
Eval (cont'd.)	SIZE	Calculates the size of a type. The size of numeric types is always 1; for character arrays, it is the number of characters; for byte arrays, the number of bytes; and for containers (tuple, bag, map), it is the number of entries.
	SUM	Calculates the sum of the values of entries in a bag.
	TOBAG	Converts one or more expressions to individual tuples, which are then put in a bag. A synonym for ().
	TOKENIZE	Tokenizes a character array into a bag of its constituent words.
	TOMAP	Converts an even number of expressions to a map of key-value pairs. A synonym for [].
	TOP	Calculates the top n tuples in a bag.
	TOTUPLE	Converts one or more expressions to a tuple. A synonym for {}.

Pig Latin: *Functions*

Table. A selection of Pig's built-in functions cont'd.

Category	Function	Description
Filter	`IsEmpty`	Tests whether a bag or map is empty.
Load/Store	`PigStorage`	Loads or stores relations using a field-delimited text format. Each line is broken into fields using a configurable field delimiter (defaults to a tab character) to be stored in the tuple's fields. It is the default storage when none is specified.
	`TextLoader`	Loads relations from a plain-text format. Each line corresponds to a tuple whose single field is the line of text.
	`JsonLoader`, `JsonStorage`	Loads or stores relations from or to a (Pig-defined) JSON format. Each tuple is stored on one line.
	`AvroStorage`	Loads or stores relations from or to Avro datafiles.
	`ParquetLoader`, `ParquetStorer`	Loads or stores relations from or to Parquet files.
	`OrcStorage`	Loads or stores relations from or to Hive ORCFiles.
	`HBaseStorage`	Loads or stores relations from or to HBase tables.

Pig Latin: *Outline*

- Structure
- Statements
- Expressions
- Types
- Schemas
- Functions
- Macros 

Pig Latin: *Macros*

- Macros provide a way to package reusable pieces of Pig Latin code from within Pig Latin itself.
- For example, we can extract the part of our Pig Latin program that performs grouping on a relation and then finds the maximum value in each group by defining a macro as follows:

```
DEFINE max_by_group(X, group_key, max_field) RETURNS Y {
A = GROUP $X by $group_key;
$Y = FOREACH A GENERATE group, MAX($X.$max_field);
};
```

The macro, called `max_by_group`, takes three parameters: a relation, `X`, and two field names, `group_key` and `max_field`. It returns a single relation, `Y`. Within the macro body, parameters and return aliases are referenced with a `$` prefix, such as `$X`.

Pig Latin: *Macros*

- The macro is used as follows:

```
records = LOAD 'input/ncdc/micro-tab/sample.txt'
    AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
    quality IN (0, 1, 4, 5, 9);
max_temp = max_by_group(filtered_records, year, temperature);
DUMP max_temp
```

Pig Latin: *Macros*

At runtime, Pig will expand the macro using the macro definition.

After expansion, the program looks like the following, with the expanded section in bold:

```
records = LOAD 'input/ncdc/micro-tab/sample.txt'
      AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
      quality IN (0, 1, 4, 5, 9);
macro_max_by_group_A_0 = GROUP filtered_records by (year);
max_temp = FOREACH macro_max_by_group_A_0 GENERATE group,
  MAX(filtered_records.(temperature));
DUMP max_temp
```