NoSQL Systems

Apache Pig & Pig Latin

Making it easier to process, clean and analyze "Big Data" in Hadoop

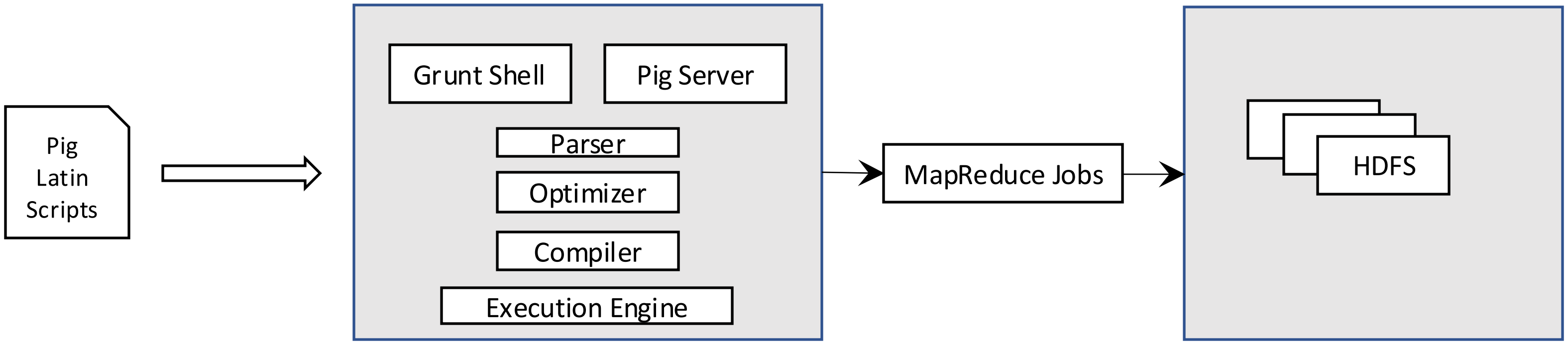
Vinu Venugopal
ScaDS.ai Lab, IIIT Bangalore

Motivation

- MapReduce is too low level and rigid
- Writing low level MapReduce code is slow, need a lot of expertise to optimize Map Reduce code, prototyping is slow
- A lot of custom code required even for a simple task, and it is hard to manage more complex map reduce job chains
  - Creating an **inverted document index** with TF/IDF scores takes at least **two MapReduce iterations**: one for the DF values and one for the combined TF/IDF values
  - Translating an **SQL query** into MapReduce typically takes **one MapReduce iteration for each relational operator** (*select*, *project*, *join*, *group-by*, etc. → compared to HBase/HIVE, next Chapters.)

What is Apache Pig?

- An Apache open-source project
- An **abstraction layer** for MapReduce (specifically: Apache Hadoop)
- Provides an engine for executing **data flows** in parallel on Hadoop
- Includes a language, ***Pig Latin***, for expressing these data flows
  - Includes operators for many of the **traditional data operations** (join, sort, filter, etc.)
  - Includes options for **user defined functions**



The **Pig compiler** translates a so-called "dataflow program" (formulated in the **Pig Latin scripting language**) into a series of MapReduce jobs.

Pig History

Pig and the Pig Latin scripting language were originally developed at Yahoo! research in 2006 (see: http://research.yahoo.com/node/90).

Shubham Chopra, Alan Gates, Shravan Narayanamurthy, Olga Natkovich, Arun Murthy, Pi Song, Santhosh Srinivasan, Amir Youssefi

Just like Apache Hadoop, Pig is currently/still developed by and available from the Apache software foundation (see: http://pig.apache.org/).

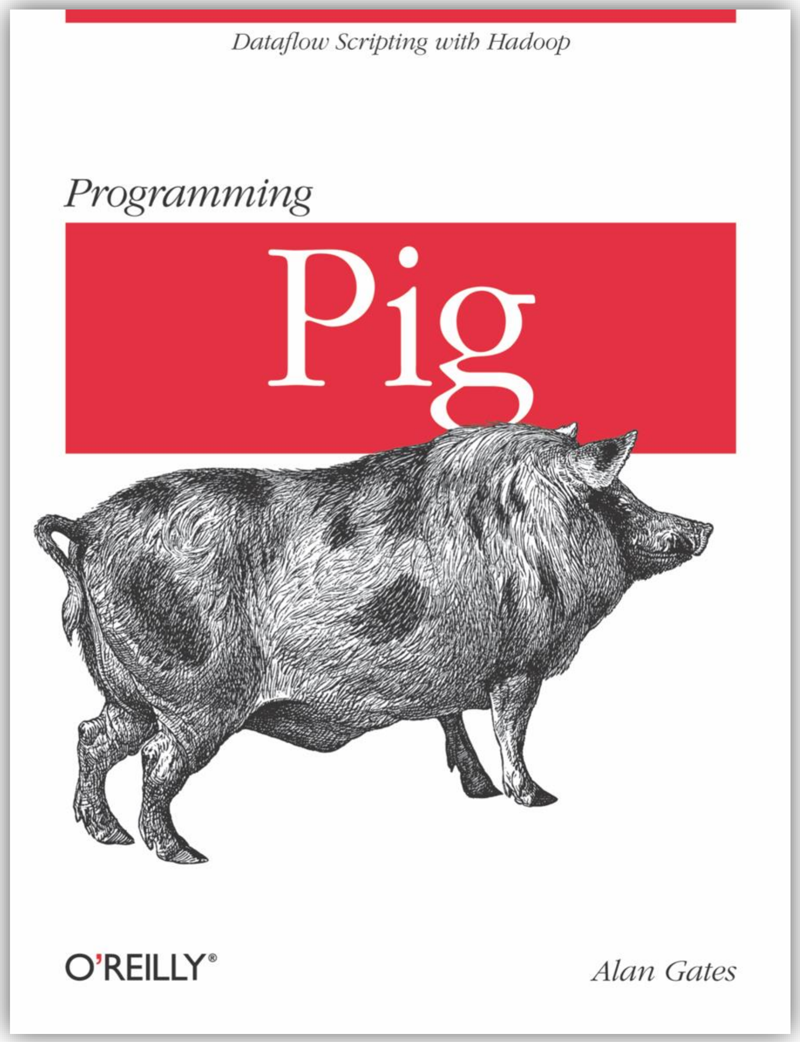
Pig History

- Paper: Christopher Olston et al, “Pig Latin: A Not-So-Foreign Language for Data Processing,” available at https://dl.acm.org/doi/10.1145/1376616.1376726 (SIGMOD’08).
- Pig started out as a research project in Yahoo! Research
- In 2007, Pig was open sourced via the Apache Incubator.
- The first Pig release came a year later in September 2008.
- By the end of 2009 about half of Hadoop jobs at Yahoo! were Pig jobs.

Why Is It Called Pig?

One question that is frequently asked is, “Why is it named Pig?” People also want to know whether Pig is an acronym. It is not. The story goes that the researchers working on the project initially referred to it simply as “the language.” Eventually they needed to call it something. Off the top of his head, one researcher suggested Pig, and the name stuck. It is quirky yet memorable and easy to spell. While some have hinted that the name sounds coy or silly, it has provided us with an entertaining nomenclature, such as Pig Latin for a language, Grunt for a shell, and Piggybank for a CPAN-like shared repository.

Pig Reference



Pro's & Con's for using Pig instead of using MapReduce directly

- Pig is an **abstraction layer** for MapReduce, which in turn is an abstraction layer for distributed programming (and, more generally, for distributed computing).
- Pig enables for a much **faster development** of complex data processing tasks in MapReduce via predefined data processing operations.
- Pig Latin **programs may however be slightly slower** than comparable, custom MapReduce functions (at least if implemented efficiently), but the translation and implementation of the predefined MapReduce functions is constantly being updated and improved.

(→ latest Pig 0.17.0 release is from June 19, 2017)

Pig Philosophy

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. And it can easily be extended to operate on data beyond files, including key/value stores, databases, etc.

Pigs live anywhere

- Pig is intended to be a language for parallel data processing. It is not tied to one particular parallel framework. It has been implemented first on Hadoop, but we do not intend that to be only on Hadoop.

Pig Philosophy

Pigs are domestic animals

- Easily controlled and modified by its users
- Allows integration of user code wherever possible, so it currently supports user defined field transformation functions, user defined aggregates, and user defined conditionals

Pigs fly

- Processes data quickly

Pig Execution Modes

Pig has two basic execution modes:

- **Local mode**
- **MapReduce mode**

In local mode, Pig **accesses the local file system and runs in a single Java Virtual Machine (JVM)**, which then simulates a MapReduce/Hadoop environment.

```
% pig –x local
```

In MapReduce mode, Pig **operates over the Hadoop distributed file system** (HDFS) and translates a series of statements provided via command-line inputs or a script file into a series of MapReduce jobs.

Running Pig Programs

Command line mode:

- *grunt* is the built-in command-line interface of Pig.

```
% pig
grunt> MyRecords = LOAD '/myfiles/sample.txt';
```

Scripting mode:

- either run a .pig script from the system command-line, or run the script via exec in grunt.

```
% pig
grunt> exec '/myfiles/sample.pig';
```

Embedded mode:

- Pig also has a JDBC-like interface for running Pig Latin scripts from Java (PigServer class).

Pig’s Data Model (I)

The basic data model of Pig are **nested relations**.

(if no nesting, similar to a relation in a DBMS, also *allowing duplicates*)

```
{ (1, Luxembourg), (2, Esch), (3, Remisch), (2, Esch) }
```

Elements of a relation may be of a ***simple type*** or of a ***complex type***.

- Pig knows 6 simple (atomic) data types:
  - `int` (32-bit signed integer), `long` (64-bit signed integer),
  - `float` (32-bit floating point number), `double` (64-bit floating point number),
  - `chararray` (array of UTF-16 chars), `bytearray` (byte array, 8-bit each, default type)
- And 3 complex types: `bag`, `tuple`, `map` (key must be `chararray`!)

Caution: complex types may be nested!

```
{ (1, (Maison-du-Nombre, Belval)) }
```

Pig Data Model (II)

Map

- **Key-Value pair;** Key should be chararray type and Value can be any Pig type, including a complex type.
- By default, there is no requirement that all values in a map must be of the same type.
- Uses brackets to delimit the map, a **hash** between keys and values, and a **comma** between key-value pairs. E.g., ['name'#'bob', 'age'#55]

Tuple

- Fixed-length, **ordered** collection of Pig data elements.
- **Divided into *fields***, with each field containing one data element.
- Analogous to a **row** in SQL, with the fields being SQL columns.
  - but is not required to, have a schema and a name for each field.
- Uses **parentheses** to indicate the tuple and **commas** to delimit fields in the tuple. E.g., ('bob', 55) describes a tuple constant with two fields.

Pig Data Model (III)

Bag

- **Unordered collection of tuples**
- **Not possible** to reference tuples in a bag **by position**
- Constructed using **braces**, with tuples in the bag separated by **commas**.
- E.g, {('bob', 55), ('sally', 52), ('john', 25)} constructs a bag with three tuples, each with two fields.

Pig Data Model (IV)

- Relations are usually loaded from structured text files (TSV, CSV, etc.).
- Schemas (i.e., both attribute names and their types) are optional.
- Input files do not come with their own schema, but the LOAD operator allows for assigning a schema to the loaded fields at runtime.

```
MyRecords = LOAD '/myfiles/sample.txt’ AS (name:chararray,
rollno:long, emailid:chararray, groupid:int);
```

- Fields that **violate the schema** are automatically turned into null values.
- If two schemas are merged (e.g., using the UNION operator), the input schemas need to be compatible, otherwise the types will be converted to the default type bytearray.

Statements & Logical Query Plans

- **Statements** (each terminated by a ;) are parsed directly when entered in the command-line or when loaded from a script.
- The Pig interpreter builds a **logical query plan** for each **relational operation** (these form the core of a Pig Latin program).
- **Syntax errors** yield a **compile-time error**, while **semantic problems** (e.g., undefined aliases) yield either a **compile-time or a run-time error** (e.g., function calls with incompatible types).
- **Lazy Evaluation:** Data loading and processing takes place only once the program is finished by a DUMP or STORE statement.

DataFlow Processing

- A Pig Latin script defines the dataflow
- Consists of **Data processing operators** and predefined templates of MapReduce functions for common **relational operations** such as *joins*, *grouping*, *sorting*, etc.
- As a rule of thumb, each such operator in a Pig Latin script is **translated into *one* MapReduce job**.

The **common Hadoop strategies** regarding file splits and job allocation **apply for Pig** as well:

By default, one MapReduce task is allocated for each HDFS block (usually 64-128 MB = 1 filesplit);

the task is attempted to be run on the same compute node where the HDFS block is located.

Example 1: Simple Pig Latin Script

```
MyRecords = LOAD '/myfiles/sample.txt' AS (name:chararray,
rollno:long, emailid:chararray, groupid:int);
FilteredRecords = FILTER Records BY groupid <= 10;
GroupedRecords = GROUP FilteredRecords BY groupid;
Counts = FOREACH GroupedRecords GENERATE group AS ID,
    COUNT(FilteredRecords) AS StudentCount;
DUMP Counts;
```

Note: `GROUP` creates a new (nested) relation in which the first field is the grouping field with the actual alias "`group`". The second field is a bag containing the grouped fields with the name of the previous relation as the alias (in this case `FilteredRecords`).

Generally, attributes may be referred to by explicit aliases (e.g., `MyRecords.City`) or by their position within a tuple (e.g., `MyRecords.$1`).

Example 2: WordCount in Pig Latin

```
-- Extract words from each line and put them into a bag datatype
lines = LOAD 'yago_test.tsv' AS (line:chararray);
-- then flatten the bag to get one word for each tuple.
words = FOREACH lines GENERATE FLATTEN(TOKENIZE(line)) AS word;
-- Filter (=keep) words that consist of word char. optionally surrounded by '<' and '>'.
filtered_words = FILTER words BY word MATCHES '[<]*\\w+[>]*';
-- Create a group for each filtered word.
word_groups = GROUP filtered_words BY word;
-- Count the entries in each group.
word_counts = FOREACH word_groups GENERATE group AS word,
COUNT(filtered_words) AS count;
-- Finally order the records by count.
ordered_word_counts = ORDER word_counts BY count DESC;
--Store the list
STORE ordered_word_counts INTO './yagowordcounts';
```

Pig Latin: the language

- Core of Pig Latin are built-in (relational) **data processing operators**. It is however extensible via **user-defined functions** (UDFs).

- Pig Latin is a ***scripting*** *language*, designed to work much like an ***imperative*** programming language. (how to do, but not what to do!)
  - Via the option to define macros and UDF's, Pig Latin is ***procedural***, but it is not object-oriented.

- In addition to being imperative and procedural, **Pig Latin is also an *expression language***, meaning that it supports expressions that evaluate to values, much like mathematical or programming expressions.

```
data = LOAD 'students.txt' AS (name:chararray, age:int, marks:int);
result = FOREACH data GENERATE name, age + 1;
DUMP result;
```

Pig Latin Expressions

Expressions can be used as part of a statement that contains a relational operator. Expressions may yield a value of simple or complex type. Expressions may be nested.

Examples:

Literal (constant value)	`1.0`, `' 017066772B '`
Container alias (variable name)	`MyRecords`
Field (by name)	`EmailId`
Field (by position)	`$1`
Field (in input container)	`MyRecords.City`
Field (disambiguation after op's)	`MyRecords::City`
Map lookup*	`MyMap#'City'`
Type casting	`(int) Year`
Null checks	`City is null / City is not null`
Conditional	`Year==2013?'YES':'NO'`

***MyMap#'City'** extracts the value associated with the 'City' key from the map

Comparison to SQL

Many language constructs in Pig Latin are inspired by SQL. However, the **data model of Pig is "more relaxed"** than the relational data model and allows for a more lazy evaluation of the operators and verification of types.

Schemas are not required by any of the operators. UDFs may expect certain types as input, but a violation thereof only results in a runtime exception.

Pig is strongly **file-oriented**, whereas DBMS's do not have a mechanism for running queries over files (esp. not in parallel).

Pig allows for splitting data streams into **multiple pipelines** (called **multi-query execution**), whereas SQL is oriented towards a single query result.

Pig has **no notion of indexes** to accelerate data access if queries are selective.

Multi-Query Execution in Pig

In Pig, the **same input dataset can be processed in multiple ways simultaneously** without reloading it multiple times.

This is useful when:

- You need **different transformations** on the same dataset.
- You want to **save multiple outputs** from one processing pipeline.
- It helps **reduce redundant computation**, making execution **faster and more efficient**.

Example -1 : Multi-Query Execution

The following dataflow program is possible in scripting mode only (output are two TSV files `b.txt` and `c.txt`):

```
A  = LOAD '/myfiles/a.txt';
B = FILTER A BY $1 == 'apple';
C = FILTER A BY $1 != 'apple';
STORE B INTO '/myfiles/b.txt';
STORE C INTO '/myfiles/c.txt';
```

Rather than reading the data in `a.txt` twice, this script creates the output for `B` and `C` in a single scan.

This (non-trivial!) feature is called **multi-query execution**.

Example -2 : Multi-Query Execution

```
data = LOAD 'students.txt' AS (name:chararray, age:int, marks:int);

-- First pipeline: Get students older than 18
adults = FILTER data BY age > 18;
STORE adults INTO 'adults_data';

-- Second pipeline: Find students who scored above 80
top_students = FILTER data BY marks > 80;
STORE top_students INTO 'top_students_data’;
```

Pig optimizes execution by sharing intermediate results instead of running multiple independent queries.

Pig Latin Operators & Commands Overview

Data processing (and "relational") operators

`LOAD`, `STORE`, `DUMP`, `FILTER`, `JOIN`, `DISTINCT`,…

Macros & UDF registering commands

`REGISTER`, `DEFINE`, `IMPORT`

Diagnostic commands

`DESCRIBE`, `EXPLAIN`, `ILLUSTRATE`

File system commands

`cat`, `cd`, `ls`, `copyFromLocal`,…

Utility commands

`kill`, `exec`, `help`, `quit`,…

Not part of a script; thus need not be terminated by a semicolon.

Data Processing Operators (I)

`LOAD`	Loads data from file.
`STORE`	Saves data to file.
`DUMP`	Prints a relation to the console.
`FILTER`	Removes individual tuples.
`DISTINCT`	Removes duplicate tuples.
`FOREACH`…	Manipulates fields within a tuple.
`GENERATE`	(similar to the SELECT clause in SQL)

Data Processing Operators (I)

`LOAD`	Loads data from file.
`STORE`	Saves data to file.
`DUMP`	Prints a relation to the console.
`FILTER`	Removes individual tuples.
`DISTINCT`	Removes duplicate tuples.
`FOREACH…`	Manipulates fields within a tuple.
`GENERATE`	(similar to the SELECT clause in SQL)

```
data = LOAD 'students.txt' AS (name:chararray, age:int, marks:int);
result = FOREACH data GENERATE name, age;
DUMP result;
```

Data Processing (Relational) Operators (II)

`JOIN`	Joins two or more bags.
`GROUP`	Groups elements in a bag.
`COGROUP`	Groups elements in two or more bags.
`CROSS`	Creates a cross-product between two bags.
`UNION`	Unions two bags (caution: must have compatible schemas!)
`SPLIT`	Splits tuples in a bag into multiple bags.
`ORDER`	Sorts a bag by one or more fields
`LIMIT`	Cuts-off elements from a bag.

Grouping & Joining (I)

JOIN

```
grunt> DUMP A;
(2,Tie)
(4,Coat)
(3,Hat)
(1,Scarf)
grunt> DUMP B;
(Joe,2)
(Hank,4)
(Ali,0)
(Eve,3)
(Hank,2)
```

The semantics for inner and outer joins in Pig Latin is taken over from its relational counterparts

```
grunt> C = JOIN A BY $0, B BY $1;
grunt> DUMP C;
(2,Tie,Joe,2)
(2,Tie,Hank,2)
(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

```
grunt> C = JOIN A BY $0 LEFT OUTER,
              B BY $1;
grunt> DUMP C;
(1,Scarf,,)
(2,Tie,Joe,2)
(2,Tie,Hank,2)
(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

Grouping & Joining (II)

GROUP-BY

```
grunt> DUMP A;
(Joe,cherry)
(Ali,apple)
(Joe,banana)
(Eve,apple)
```

```
grunt> B = GROUP A BY SIZE($1);
grunt> DUMP B;
(5,{(Ali,apple),(Eve,apple)})
(6,{(Joe,cherry),(Joe,banana)})
```

```
grunt> C = GROUP A ALL;
grunt> DUMP C;
(all,{(Joe,cherry),(Ali,apple),
      (Joe,banana),(Eve,apple)})
```

Unlike in SQL, grouping is not combined with a projection.

The results are tuples with nested bags of tuples.

ALL groups all tuples in a relation into one tuple.

Grouping & Joining (III)

```
grunt> DUMP A;
(2,Tie)
(4,Coat)
(3,Hat)
(1,Scarf)
grunt> DUMP B;
(Joe,2)
(Hank,4)
(Ali,0)
(Eve,3)
(Hank,2)
```

CO-GROUP

```
grunt> D = COGROUP A BY $0, B BY $1;
grunt> DUMP D;
(0,{},{(Ali,0)})
(1,{(1,Scarf)},{})
(2,{(2,Tie)},{(Joe,2),(Hank,2)})
(3,{(3,Hat)},{(Eve,3)})
(4,{(4,Coat)},{(Hank,4)})
```

`COGROUP` is similar to `JOIN` but creates a nested bag of tuples for each matching combination, thus keeping the input structure intact.

May be useful for subsequent operations.

Sorting

ORDER-BY and LIMIT

```
grunt> DUMP A;
(2,3)
(1,2)
(2,4)
```

```
grunt> B = ORDER A BY $0, $1 DESC;
grunt> DUMP B;
(1,2)
(2,4)
(2,3)
```

```
grunt> D = LIMIT B 2;
grunt> DUMP D;
(1,2)
(2,4)
```

`ORDER-BY` is again similar to its SQL counterpart.

Caution: Only `DUMP`, `STORE` and `LIMIT` are guaranteed to retain the order of an ORDER-BY operation.

Combining & Splitting

UNION and **SPLIT**

```
grunt> DUMP A;
(2,3)
(1,2)
(2,4)
grunt> DESCRIBE A;
A: {f0:int,f1:int}
grunt> DUMP B;
(z,x,8)
(w,y,1)
grunt> DESCRIBE B;
B: {f0:chararray,f1:chararray,f2:int}
```

```
grunt> C = UNION A, B;
grunt> DUMP C;
(z,x,8)
(w,y,1)
(2,3)
(1,2)
(2,4)
grunt> DESCRIBE C;
Schema for C unknown.
```

`SPLIT` is "in some sense" the counterpart of `UNION`.

```
grunt> SPLIT A INTO GoodRecords IF $0 > 1,
  BadRecords IF $0 <= 1;
```

Macros & UDF Commands

`REGISTER`	Registers a Jar file with the Pig runtime (e.g., containing the UDFs).
`DEFINE`	Defines and creates an alias for a macro. (see later slides)
`IMPORT`	Imports macros defined in a separate file.

Macros

- **Macros** allow for the definition of complex Pig Latin scripts as reusable components of code.

```
DEFINE MaxByGroup(X, GroupKey, MaxField)
    RETURNS Y {
  A = GROUP $X BY $GroupKey;
  $Y = FOREACH A GENERATE group, MAX($X.$MaxField);
};
```

We can run the macro as follows:

```
grunt> MyRecords = LOAD '/myfiles/sample.txt’ AS
     (ID:integer, City:chararray, temperature:integer);

grunt> GroupedRecords = MaxByGroup(MyRecords, groupID, temperature);
-- Max. temperature reported for each city
grunt> DUMP GroupedRecords;
```

Note: Aliases with a $ prefix relate to *local variables*, otherwise they relate to *global variables*.

Macros

- **Macros** allow for the definition of complex Pig Latin scripts as reusable components of code.

```
DEFINE MaxByGroup(X, GroupKey, MaxField)
    RETURNS Y {
  A = GROUP $X BY $GroupKey;
  $Y = FOREACH A GENERATE group, MAX($X.$MaxField);
};
```

After the Pig preprocessor expands the macro, the actual Pig script looks like this:

```
MyRecords = LOAD '/myfiles/sample.txt' AS (ID:integer, City:chararray, temperature:integer);

A = GROUP MyRecords BY City;

GroupedRecords = FOREACH A GENERATE group, MAX(MyRecords.temperature);

DUMP GroupedRecords;
```

Diagnostic Commands

`DESCRIBE`	Prints schema information.
`EXPLAIN`	Prints a logical MapReduce plan.
`ILLUSTRATE`	Shows a sample execution, using a sampled subset of the input data.

File System & Utility Commands

Includes the usual Unix commands:

`cat`, `cd`, `cp`, `fs`, `ls`, `mkdir`, `mv`, `pwd`, `rm` (and `rmf`)

And additional the HDFS commands:

`copyFromLocal`, `copyToLocal`

Hadoop utility commands:

`kill`	Kills a MapReduce job.
`exec`	Runs a script in a new grunt shell.
`run`	Runs a script in the existing grunt shell.
`sh`	Runs a shell command.
`set`	Sets various parameters.
`help`	Shows available commands.
`quit`	Exits the Pig interpreter.

Built-In Functions

- As part of expressions, Pig Latin supports the common **built-in functions** for aggregations, set difference, string concatenation, etc.

`AVG`, `SUM`, `COUNT`, `MAX`, `MIN`	similar semantics as SQL
`SIZE`	yields the size of a type (number of bytes or elements)
`DIFF`	computes the set difference among bags
`CONCAT`, `TOKENIZE`	for corresponding string operations.

- Additionally, Pig has 3 functions to **create complex types**:

`TOTUPLE`	Converts one or more expressions to a tuple.
`TOBAG`	Converts one or more expressions to a tuple, then puts them into a bag.
`TOMAP`	Converts an even number of arguments into a map.

Built-In Functions

data.txt

(1,Alice,25)
(2,Bob,30)
(3,Charlie,28)

TOTUPLE (Creates a **Tuple**)

```
A = LOAD 'data.txt' USING PigStorage(',') AS (id:int, name:chararray, age:int);
B = FOREACH A GENERATE id, TOTUPLE(name, age) AS person_tuple;
DUMP B;
```

TOBAG (Creates a **Bag**)

```
A = LOAD 'data.txt' USING PigStorage(',') AS (id:int, name:chararray, age:int);
B = FOREACH A GENERATE id, TOBAG(name, age) AS person_bag;
DUMP B;
```

TOMAP (Creates a **Map**)

```
A = LOAD 'data.txt' USING PigStorage(',') AS (id:int, name:chararray, age:int);
B = FOREACH A GENERATE id, TOMAP(name, age) AS person_bag;
DUMP B;
```

Built-In Functions

data.txt
(1,Alice,25)
(2,Bob,30)
(3,Charlie,28)

TOTUPLE (Creates a **Tuple**)

```
A = LOAD 'data.txt' USING PigStorage(',') AS (id:int, name:chararray, age:int);
B = FOREACH A GENERATE id, TOTUPLE(name, age) AS person_tuple;
DUMP B;
```

(1,(Alice,25))
(2,(Bob,30))
(3,(Charlie,28))

TOBAG (Creates a **Bag**)

```
A = LOAD 'data.txt' USING PigStorage(',') AS (id:int, name:chararray, age:int);
B = FOREACH A GENERATE id, TOBAG(name, age) AS person_bag;
DUMP B;
```

(1,{(Alice,25)})
(2,{(Bob,30)})
(3,{(Charlie,28)})

TOMAP (Creates a **Map**)

```
A = LOAD 'data.txt' USING PigStorage(',') AS (id:int, name:chararray, age:int);
B = FOREACH A GENERATE id, TOMAP(name, age) AS person_bag;
DUMP B;
```

(1,[name#Alice, age#25])
(2,[name#Bob, age#30])
(3,[name#Charlie, age#28])

User-defined Functions (UDFs)

- Pig distinguishes **4 types of functions**:
  - **Eval functions** (e.g., `MAX`) take **one or more expressions** as input and return a **single** value.
  - **Filter functions** are **special Eval functions** that take a **single expression** as input and return a **Boolean value**.
  - **Load functions** specify **how to load data** from files. Load functions may return more than one value as return type (e.g., a bag or a map).
  - **Store functions** specify **how to save the contents** of a relation or type into a file.

- **User-defined functions** in Pig are implemented as **Java classes** that inherit predefined methods for above 4 basic function types.

Example: Filter Function as UDF

```
import org.apache.pig.FilterFunc;
…
public class IsEven extends FilterFunc {
  @Override
  public Boolean exec(Tuple tuple) throws IOException {
    if (tuple == null || tuple.size() == 0) return false;
    try {
       Object object = tuple.get(0);
       if (object == null) return false;
       int i = (Integer) object;
       return i == 2 || i == 4 || i == 6 || i == 8 || i == 10;
    } catch (ExecException e) {
       throw new IOException(e);
    }
  }
}
```

The Java class needs to be compiled, loaded into a Jar file, and the new function has to be registered in the Pig runtime:

```
grunt> REGISTER IsEven.jar;
```

Running a UDF

After registering the Jar file containing the new UDF in the Pig runtime, the function can be used just like any other function (or macro) by using the full Java class name (including the package structure) with the desired arguments.

```
MyRecords = LOAD ‘sample.txt' AS
(name:chararray,rollno:chararray,emailid:chararray,groupid:int);
REGISTER IsEven.jar;
FilteredRecords = FILTER MyRecords BY IsEven(groupid);
```

Caution: The type of the arguments have to match the types assumed in the function implementation. Otherwise, the function call fails.

`getArgToFuncMapping()` in `EvalFunc` can be employed to tell Pig how the arguments should be interpreted.

(see **IsEven.java** example on Moodle)

Example: Load Function as UDF

See CutLoadFunc.java on Moodle!

Cut plain-text input files into fields by some column ranges:

```
grunt> Records = load 'temperature-2014.txt' Using
CutLoadFunc('16-19,88-92,93-93') as (year:int, temperature:int,
quality:int);

grunt> DUMP Records;

    (1950, 0,   1)
    (1950, 22,  1)
    (1950, -11, 1)
    (1950, 111, 1)
    (1950, 78,  1)
```

0089010010999992014010101004+70933-008667FM-12+000999999V0203601N00
4019999999N999999999-00041-00401101211ADDMA1999990101091MD1810071
+9990REMSYN04801001 46/// /3604 11004 21040 30109 40121 58007=

Note on Controlling Parallelism in Pig

Parallelism is controlled in Pig **dynamically** for the job size.

- One Reducer per 1 GB of input data.
- One Mapper per HDFS block (usually 64-128 MB).

The number of Reducers can be controlled in Pig Latin by the `PARALLEL` clause for operators that run in the reduce phase.

```
GroupedRecords = GROUP MyRecords BY City
  PARALLEL = 30;
```

The Pig API provides its own classes for `PigSplit`, `PigInputFormat` (incl. `PigTextInputFormat`), `PigOutputFormat` etc.

Again, custom implementations of these classes can be registered to the runtime via various parameters.

More Reading..

Aggregate Functions: a type of eval function that takes a bag and returns a scalar value.

These functions can be **algebraic** which means computed incrementally in a distributed fashion – just like, in Hadoop we do partial computations by map and combiner, and final result can be computed by the reducer.

Reference: https://pig.apache.org/docs/latest/udf.html