INTRODUCTION TO APACHE PIG

Introduction to Apache Pig

- **Pig Identifiers**
- Pig Data Types
- Loading/Storing Data
- Filter Operation
- Foreach Operation
- Constructing an Expression
- Built-in Functions

Pig Identifiers

- Name of relations (aliases) and fields are identifiers
- Identifiers start with a letter and can be followed by any number of letters, digits, and underscores
- Identifiers are case-sensitive

Introduction to Apache Pig

- Pig Identifiers
- **Pig Data Types**
- Loading/Storing Data
- Filter Operation
- Foreach Operation
- Constructing an Expression
- Built-in Functions

Pig Data Types

- A Pig relation is essentially of a Pig Data Type, namely, **outer bag**
- A **bag** is a **complex data** type, which is an **unordered set of tuples**, which is **an ordered collection of fields**
- Considering **a bag as a table**, its records with fields are represented as **tuples**
- A field can itself be of a complex data type, allowing nested schemas, or of a **scalar data type** such as **int**, **float**, **chararray**, …

Pig Scalar Types

- Scalar types are the common primitive types
  - They are internally represented as their **`java.lang.<type>`** counterparts
- Scalar types in Pig are:
  - int (**`java.lang.Integer`**)
  - long (**`java.lang.Long`**)
  - float (**`java.lang.Float`**)
  - double (**`java.lang.Double`**)
  - chararray (**`java.lang.String`**)
  - bytearray (**`DataByteArray`** wrapping **`byte[]`**)

Pig Complex Types

- The complex types in Pig are:
  - Map
    - A mapping of chararray keys to complex types
    - In default PigStorage load/store function, a constant map instance is represented as:

```
['name'#'alice', 'age'#33]
```

    - A map can be associated with a schema (both per field and for all fields)

Pig Complex Types

- The complex types in Pig are (cont.):
  - Tuple
    - A Tuple is an ordered collection of fields with a fixed length (like a row in a relation)
    - In PigStorage, a constant tuple instance is represented as:

      ```
      ('alice', 33)
      ```

    - A tuple can be associated with a schema (name and type for its fields)
    - Fields in a tuple can be referred by their positions, or names

Pig Complex Types

- The complex types in Pig are (cont.):
  - Bag
    - A Bag is an unordered collection of Tuples
    - In PigStorage, a constant Bag instance is represented as:

```
{('alice', 33), ('bob', 44), ('john', 22)}
```

    - A bag can be associated with a schema (defining the schema for all tuples contained by this bag)
    - A bag does not have to fit in the memory, in fact an entire Pig relation is a bag

Pig Schemas

- We start with noting that even though a schema is not associated with a relation, Pig tries its best to process it
  - Pig implicitly casts data to the expected type of a method call:
    - Casting int to long, float, double, and chararray is possible
    - Casting long to int, float, double, and chararray is possible
    - Casting float to long, int, double, and chararray is possible
    - Casting double to long, int, float, and chararray is possible
    - Casting chararray to long, int, float, and double is possible
      - Chararrays containing non-numeric characters are converted to null values

Pig Schemas

- A schema is defined with the LOAD and FOREACH operations
- A schema can be defined with:
  - Both field name and value
  - The field name only (field type is default to bytearray)

Pig Schemas

- A schema is assigned with **as** clause, such as:
  - `as (a: int), as (a)`
  - `as (a: long), as (a)`
  - `as (a: map[]), as (a: map[int])`
  - `as (a: tuple()), as (a: tuple(x: int, y: int))`
  - `as (a: bag{}), as (a: bag {t: (x: int, y: int)})`
  - `as (a: bag {t: (x:int, y:map[int], z: bag {t: (q:chararray)})})`

Pig Schemas

- Pig enforces a schema during the actual execution
  - Schema enforcing is executed by casting the input data to the expected type
- Pig determines the data types and performs the conversions on the fly

Introduction to Apache Pig

- Pig Identifiers
- Pig Data Types
- **Loading/Storing Data**
- Filter Operation
- Foreach Operation
- Constructing an Expression
- Built-in Functions

Loading a Relation

- To load a relation, we require a load function to which we provide:
  - The input location

```
A = LOAD '/data/students/all' USING PigStorage(',')
```

  - If no load function is specified, PigStorage with tab delimiters is used by default

```
-- This is is legit
A = LOAD '/data/students/all'
```

Loading a Relation

- PigStorage is used for loading/storing relations as they are stored in text files, of records of lines, fields delimitied by a character (\t, by default)
- A load function is defined by an **`org.apache.pig.LoadFunc`**, which is tightly coupled with Hadoop's **`InputFormat`**

Loading a Relation

- We can attach a schema to the input relation

```
A = LOAD '/data/students/all'
  USING PigStorage(',')
  AS (name: chararray, age: int, gpa: float);
```

Loading a Relation

- Of course, we can define schemas of complex types

```
/* This is how the data is actually stored
(3,8,9) (4,5,6)
(1,4,7) (3,7,5)
(2,5,8) (9,5,8)
*/

A = LOAD 'data'
    AS (t1:tuple(t1a:int,t1b:int,t1c:int),
       t2:tuple(t2a:int,t2b:int,t2c:int));
```

Loading a Relation

- The schema information passed with the as clause is used to cast the fields of tuples from bytearrays into the appropriate data types, as defined in the associated LoadCaster of the load function
  - For example, one cannot define integer fields while loading with a TextLoader

Storing a Relation

- To store a relation, we require a store function to which we provide:
  - The output location

```
STORE A into '/data/students/all'
USING PigStorage(',');
```

  - If no store function is specified, PigStorage with tab delimiters is used by default

```
-- This is is legit
STORE A into '/data/students/all';
```

Storing a Relation

- PigStorage is used for loading/storing relations as they are stored in text files, of records of lines, fields delimitied by a character (\t, by default)
- A store function is defined by an **`org.apache.pig.StoreFunc`**, which is tightly coupled with Hadoop's **`OutputFormat`**

Introduction to Apache Pig

- Pig Identifiers
- Pig Data Types
- Loading/Storing Data
- **Filter Operation**
- Foreach Operation
- Constructing an Expression
- Built-in Functions

FILTER

- Operation for selecting tuples that satisfy on a condition:

```
grunt> describe A;
A: {name: chararray, age: int, gpa: float}

grunt> B = FILTER A BY age>20
```

FILTER

```
B = FILTER A BY age>20;
```

A

name	age	gpa
Addison	21	2.09
Sade	19	2.78
Ivana	24	2.9
Karen	22	1.4
Alice	18	2.87
Linda	25	3.3
Russel	18	2.3
Ralph	19	2.2
Bob	21	3.5

B

name	age	gpa
Addison	21	2.09
Ivana	24	2.9
Karen	22	1.4
Linda	25	3.3
Bob	21	3.5

FILTER

- To apply a filter, we need a comparison operator(like **`age>20`** in the example)
- Comparison operators in Apache Pig are:
  - The usual ones: **`=, !=, <, >, <=, >=`**
  - And **`matches`** for regular expression matching:

```
B = FILTER A BY expr1 matches '.*apache.*';
```

Introduction to Apache Pig

- Pig Identifiers
- Pig Data Types
- Loading/Storing Data
- Filter Operation
- **Foreach Operation**
- Constructing an Expression
- Built-in Functions

FOREACH .. GENERATE

- Operation for generating transformations based on columns of data (projection)

```
grunt> describe A;

A: {f1: int, f2: chararray}

grunt> B = FOREACH A GENERATE f1;

grunt> describe B;

B: {f1: int}
```

FOREACH .. GENERATE

```
B = FOREACH A GENERATE name, age;
```

name	age	gpa
Addison	21	2.09
Sade	19	2.78
Ivana	24	2.9
Karen	22	1.4
Alice	18	2.87
Linda	25	3.3
Russel	18	2.3
Ralph	19	2.2
Bob	21	3.5

name	age
Addison	21
Sade	19
Ivana	24
Karen	22
Alice	18
Linda	25
Russel	18
Ralph	19
Bob	21

FOREACH .. GENERATE

- Nested projections are possible

```
-- D: {f1: int, f2: {(email:chararray, age: int)}}
-- retain only the age column of the inner bag
-- . is the dereference operator

grunt> E = FOREACH D GENERATE f1, f2.age as ages
grunt> describe E;

E: {f1: int, ages: {(age:int)}}
```

FOREACH .. GENERATE

- Schema assignment is allowed

```
-- A: {f1: chararray, f2: int, f3: int}

grunt> B = FOREACH A GENERATE f1, f2*f3 as (f4:long)
grunt> describe B;

E: {f1: int, f4: long}
```

FOREACH .. GENERATE

- Using FOREACH .. GENERATE, aggregations can be performed on a field of type inner bag

```
grunt> describe D;

D: {f1: int, f2: {(email:chararray, age: int)}}

-- an aggregation; COUNT is a built-in function
grunt> E = FOREACH D GENERATE f1, COUNT(f2) as cnt;

grunt> describe E;

E: {f1: int, cnt: long}
```

FOREACH .. GENERATE

- GENERATE statement can be put into a block as the last statement

```
-- D: {f1: int, f2: {(email:chararray, age: int)}}
-- another aggregation, followed by a filter, nested
-- this time

grunt> E = FOREACH D {
         S = FILTER f2 BY age>20;
         GENERATE f1, COUNT(S) as cnt;
        }

grunt> describe E;

E: {f1: int, cnt: int}
```

Demo

Load/Store

Demo

Projections and Aggregations with FOREACH .. GENERATE

Demo

FOREACH .. GENERATE in Blocks

Introduction to Apache Pig

- Pig Identifiers
- Pig Data Types
- Loading/Storing Data
- Filter Operation
- Foreach Operation
- **Constructing an Expression**
- Built-in Functions

Expressions

- An expression in Pig can be constructed by:
  - Any field expression (the field, or the dereference operator applied to a field)
  - Any Pig data type
  - Any Pig operator: arithmetic, comparison, boolean, dereference, sign, cast, and null
  - Any Pig built-in function
  - Any Pig UDF (user defined function)

Dereference Operator

- Often, we need to assign to a new field an element of a containing field with a complex type
- This is achieved by dereference **`(. or #)`**:
  - Tuple dereference: **`tuple.id`** or **`tuple.(id1, id2, …)`** where id can be the name or position: **`mytuple.age`**, **`mytuple.$0`**, **`mytuple.(age,email)`**, …
    - This expression represents a tuple
  - Bag dereference: **`bag.id`** or **`bag.(id1, id2, …)`** where id can be the name or the position, applied to each contained tuple within the bag: **`mybag.age`**, **`mybag.$0`**, **`mybag.(age, email)`**, …
    - This expression represents a bag

Dereference Operator

- Often, we need to assign to a new field an element of a containing field with a complex type
- This is achieved by dereference **`(. or #)`**:
  - Tuple dereference: **`tuple.id`** or **`tuple.(id1, id2, …)`** where id can be the name or position: **`mytuple.age`**, **`mytuple.$0`**, **`mytuple.(age,email)`**, …
    - This expression represents a tuple
  - Bag dereference: **`bag.id`** or **`bag.(id1, id2, …)`** where id can be the name or the position, applied to each contained tuple within the bag: **`mybag.age`**, **`mybag.$0`**, **`mybag.(age, email)`**, …
    - This expression represents a bag

Dereference Operator

- Often, we need to assign to a new field an element of a containing field with a complex type
- This is achieved by dereference **`(. or #)`**:
  - Map dereferencing is done by key on a field name or position: **`map_field#key`**, **`$0#key`**
    - If the key exists in the map, this expression returns the value
    - If the key doesn't exist in the map, this expression returns the empty string

Dereference Example

```
-- D: {f1: int, f2: {(email:chararray, age: int)}}
-- retain only the age column of the inner bag
-- . is the dereference operator

grunt> E = FOREACH D GENERATE f1, f2.age as ages
grunt> describe E;

E: {f1: int, ages: {(age:int)}}
```

Arithmetic Operators

- An expression can be constructed by applying an arithmetic operation on field(s)
  - `+, -, /, *, %` are allowed

```
-- A: {f1: chararray, f2: int, f3: int}

grunt> B = FOREACH A GENERATE f1, f2*f3 as (f4:long)
grunt> describe B;

E: {f1: int, f4: long}
```

Comparison Operators

- Comparison operators in Apache Pig are:
  - The usual ones: `=, !=, <, >, <=, >=`
  - And **`matches`** for regular expression matching:

```
B = FILTER A BY expr1 matches '.*apache.*';
```

Branching

- Branching with **?:**
  - We can construct

(condition?value_if_true:value_else) kind of expressions

```
/*For all tuples in A, a new field called f is
projected, and the assignment is done based on
the value of the f1 field, i.e. f1 itself if f1
matches the the regex '.*apache.*', 'not_apache'
otherwise
*/

B = FOREACH A  GENERATE
(f1 matches '.*apache.*'? f1:'not_apache') as f;
```

Branching

- Branching with **CASE *expr* WHEN *val* THEN *res1* ELSE *res2* *END***

```
/*For all tuples in A, a new field called f is
projected, and the assignment is done based on
the value of the f1 field, i.e. ‘even’ if f1 is a
multiple of 2, ‘odd’ otherwise.
*/

B = FOREACH A  GENERATE
(CASE f1%2
    WHEN 0 THEN ‘even’
    ELSE ‘odd’
 END) as f;
```

Boolean Operators

- For constructing boolean expressions:
  - **`AND, OR, NOT, IN`** operators can be used

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

Range and Star Expressions

- All fields in a tuple can be represented by star: `*`
- A range of fields of a tuple can be expressed by:
  - `$pos1 .. $pos2` (both inclusive), e.g.: `$0 .. $4`
  - `$pos1 ..` (inclusive), e.g.: `$3 ..`
  - `.. $pos2` (inclusive), e.g.: `.. $11`
  - Any $pos statement can be replaced by a column name, if a schema is assigned before,

    e.g.: `col1 .. $3`, `col1 .. col5`, `$1 .. col5`, `col1 ..`

Cast Operator

- An expression can be cast into a desired scalar type

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

- Pig performs implicit casting when required

Casting a Bag to a Scalar

- A relation (a bag), can be cast to a scalar if it contains a single tuple
- This makes sense when we want to use the result of an aggregation in later computations

Casting a Bag to a Scalar

```
grunt> describe A;
A: {name:chararray, frequency:int}

/*Suppose we have a relation B, containing a single
tuple of a dummy field f and a bag*/
grunt> describe B;
B: {f: chararray, data:{(name: chararray,
frequency:int)}}

grunt> C = FOREACH B GENERATE SUM(data.frequency) as
           total;

/*C also has only one row with the schema:
C: {total: int}
Suppose now we want to normalize the frequencies in
A. The C.total dereference actually returns a bag,
but Pig implicitly casts it to an int*/
grunt> D = FOREACH A GENERATE name, frequency/C.total
           as (relative_freq:float);

grunt> describe D;
D: {name: chararray, relative_freq: float};
```

Null Operator

- Null operator is a special comparison for checking whether a value is null or not

```
X = FILTER A BY f1 is null
```

```
Y = FILTER A BY f1 is not null
```

Null Values in Pig

- Nulls in Pig are interpreted as ‘unknown’ or ‘non-existent’
- Comparison with null results in null
- Arithmetic operations involving null result in null
- COUNT_STAR function also counts nulls
- Most aggregations (COUNT, SUM, AVG, …) ignore nulls

Null Values in Pig

- Nulls can be produced by many operations
  - At load time: It is the responsibility of the load function to interpret nulls. For example, PigStorage replaces empty strings with nulls
  - A function can return null
  - Division by zero returns null
  - Accessing a field that doesn't exist produces null

Introduction to Apache Pig

- Pig Identifiers
- Pig Data Types
- Loading/Storing Data
- Filter Operation
- Foreach Operation
- Constructing an Expression
- **Built-in Functions**

Built-in Functions

- There are many built-in functions that come with Apache Pig, classified as follows:
  - Eval Functions
  - Load/Store Functions
  - Math Functions
  - String Functions
  - Datetime Functions
  - Tuple/Bag/Map Functions

Built-in Functions

- Pig users are not limited to using merely the built-in functions, one can always:
  - Write his own function implementing the appropriate interfaces
  - Register the function
  - Use the function
- Such functions are called **User Defined Functions** (**UDF**s)
- Additionally, Pig developers can use static functions from Java libraries, provided that they take some combination of String, int, long, double, and float arguments (and arrays with these same types).
  - Such functions are dynamically invoked, and there is a Java reflection cost at each call

Built-in Functions

- Calling a built-in function example

```
-- within a pig script
-- SUM is a built-in function

B = FOREACH A GENERATE f1, SUM(f2)
```

Built-in Functions

- Calling a UDF example

```
-- within a pig script
-- MY_SUM is a UDF, which can be found in myudfs.jar,
-- which should be registered for MY_SUM to be used

REGISTER myudfs.jar
B = FOREACH A GENERATE f1, com.ac.udfs.MY_SUM(f2)
```

Built-in Functions

- There are many built-in functions that come with Apache Pig, classified as follows:
  - Eval Functions
  - Load/Store Functions
  - Math Functions
  - String Functions
  - Datetime Functions
  - Tuple/Bag/Map Functions

Built-in Functions

- Dynamic Invoker Example

```
-- within a pig script
-- java.net.URLDecoder.decode(String, String) is a
-- Java function that decodes an encoded URL string,
-- which we can use in a Pig script

DEFINE DECODE InvokeForString('java.net.URLDecoder.decode', 'String String');

B = FOREACH A GENERATE f1, DECODE(f2, 'UTF-8');
```

Example Eval Functions

- Aggregate functions: **`AVG, COUNT, SUM, MAX, COUNT_STAR, MIN`**
  - These functions take an expression as input: **`AVG(expr)`**
  - The input expression should be a bag-returning one
- **`BagToString(vals, delimiter)`**: Concatenates the elements of a bag into a String
- **`SIZE(expr)`**: Computes the number of elements in a Pig type. When the input is a bag, **`SIZE`** is semantically identical to **`COUNT_STAR`**

Example Eval Functions

- **`DIFF(bag1, bag2)`** and **`SUBTRACT(bag1, bag2):`** All tuples that are not included in both bags; all tuples that are included in bag1 but not in bag2, respectively. Both functions require the input bags should fit in memory
- **`TOKENIZE(str_expr, delimiter):`** Creates a bag of words from a string. Useful for text processing

A Note on Aggregate Functions

- In Apache Pig, aggregate functions are applied to bags, and:
  - Built-in functions
  - Dynamic invoke
  - UDFs are available to serve this purpose
- But the aggregations running on large bags are only efficient if they are optimized for Hadoop
- To achieve that optimization, such functions better implement the **`Algebraic`** and **`Accumulator`** interfaces, which allows aggregate computations are performed in an incremental and distributed (i.e. Map-Combine-Reduce) fashion
- Otherwise, a large bag would need to be passed as its entirety to an aggregate functions, which is a bad idea

FLATTEN

- Flatten is an operator that un-nests a collection field (a bag or a tuple)
- When it is applied to a tuple, the flatten operator **substitutes the fields in place** of the original tuple

```
-- FLATTEN on a tuple
-- D: {f1: int, f2: (email:chararray, age: int)}
-- un-nest the tuple

grunt> E = FOREACH D GENERATE f1, FLATTEN(f2)
grunt> describe E;

E: {f1: int, email: chararray, age:int}
```

FLATTEN

```
B = FOREACH A GENERATE name, FLATTEN(f2) as (age, gpa);
```

name	f2
Addison	(21, 2.09)
Sade	(19, 2.78)
Ivana	(24, 2.9)
Karen	(22, 1.4)
Alice	(18, 2.87)
Linda	(25, 3.3)
Russel	(18, 2.3)
Ralph	(19, 2.2)
Bob	(21, 3.5)

name	age	gpa
Addison	21	2.09
Sade	19	2.78
Ivana	24	2.9
Karen	22	1.4
Alice	18	2.87
Linda	25	3.3
Russel	18	2.3
Ralph	19	2.2
Bob	21	3.5

FLATTEN

- Flatten is an operator that un-nests a collection field (a bag or a tuple)
- When it is applied to a bag, the flatten operator **creates a row for each tuple** in the bag
- If the generation yields extra columns, a cross product occurs between the set of rows created and the one-row of remaining columns

```
-- FLATTEN on a bag
-- D: {zip:chararray , b1: {(email:chararray, age: int)}}
-- un-nest the bag and perform cross-product

grunt> E = FOREACH D GENERATE zip, FLATTEN(b1)
grunt> describe E;

E: {zip: chararray, b1::email: chararray, b1::age: int}
```

FLATTEN

```
B = FOREACH A GENERATE city, FLATTEN(b1);
```

city	b1
Istanbul	{(SmartCon, 2015), (DDD, 2015), (SmartCon, 2016)}
NY	{(Strata, 2015), (IEEE, 2015), (DS, 2016)}
Brussels	{(Hadoop Summit, 2015)}

city	b1::conf	b1::year
Istanbul	SmartCon	2015
Istanbul	DDD	2015
Istanbul	SmartCon	2016
NY	Strata	2015
NY	IEEE	2015
NY	DS	2015
Brussels	Hadoop Sumit	2015

FLATTEN

- Flatten looks like a function, but it is not, it is an operator
- Flatten changes the schema of the original relation

Demo

Demonstration of the FLATTEN Operator

Introduction to Apache Pig

End of Chapter