Pig Pending topics

Complex data type

Pig has three complex types: **maps, tuples and bags**. These complex types can contain scalar types and other complex types. So, it is possible to have a map where value field is a bag which has a tuple where one of the fields is a map.

**Map**: A map is a chararray to data element mapping which is expressed in key-value pairs. The key should always be of type chararray and can be used as index to access the associated value. It is not necessary that all the values in a map be of the same type.

Map constants are defined by square brackets with '#' separating keys from values and ',' separating key-value pairs.

['Name'#'John', 'Age'#22]

The above defines a map constant with two key-value pairs. Notice that the keys are always of type chararray while values take type chararray and int respectively.

In order to load data from files as maps, the data should be structured as below:

[this#1.9, is#2.5]

[my#3.3, vocabulary#4.1]

Sample PigLatin statements to load the above data sample as map

grunt> mapdata = load 'MapData' as (a:map[float]);

grunt> values = foreach mapdata generate a#'this' as value;

grunt> value = FILTER values BY value is not null;

grunt> dump value

The output of above statements is:

(1.9)

The load statement will construct two maps having two key-value pairs each. Notice that we specify the data type of values as 'float' in load statement. We can choose not to specify the type of values as below:

grunt> mapdata = load 'MapData' as (a:map[]);

In this case Pig assumes the type of values to be bytearray and performs implicit casts to appropriate type depending on how your PigLatin statements handle the data.

In the second statement we are trying to retrieve the value associated with 'this'. Notice the syntax

a#'this'

which will return 1.9.

**Tuple**: Tuples are fixed length, ordered collection of Pig data elements. Tuples contain fields which may be of different Pig types. A tuple is analogous to a row in Sql with fields as columns.

Since tuples are ordered it is possible to reference a field by it's position in the tuple. A tuple can, but is not required to declare a schema which describes each field's data type and provides a name for the field.

Tuple constants use parentheses to define tuple and commas to separate different fields.

('John', 25)

The above declares a tuple constant with two fields of data types, chararray and int respectively.

grunt> data = load 'StudentData';

grunt> finaldata = foreach data generate $0;

grunt> dump finaldata

In above statements, data is an outer bag (explanation of bags is coming next) which contains tuples loaded from StudentData file. Notice that we did not declare a schema for the tuples (type/name of the fields contained in the tuple). In this case, schema for the tuple is unknown.

However, we can reference individual fields in the tuple by their position ($0 references to the first field in the tuple).

grunt> data = load 'StudentData' as (name:chararray, age:int);

grunt> finaldata = foreach data generate name;

grunt> dump finaldata

In this case, we have defined a schema for the tuples.

**Bag**: Bags are unordered collection of tuples. Since bags are unordered, we cannot reference a tuple in a bag by its position. Bags are also not required to declare a schema. In case of bags, schema describes all the tuples in the bag.

Bag constants are constructed using braces with commas separating the tuples inside bag.

{('John', 25), ('Nathan', 30)}

The above constructs a bag with two tuples.

Conversion

grunt> todate\_data = foreach date\_data generate ToDate(date,'yyyy/MM/dd HH:mm:ss')

as (date\_time:DateTime >);

c2 = FOREACH c1 GENERATE c1.age,c1.gender,(chararray)c1.zipcode;

DESCRIBE C2;

Data\_ID = FOREACH File GENERATE(int) REPLACE(Date,'-','');

Split

SPLIT student\_details into student\_details1 if age<23, student\_details2 if (22<age and age>25);

grunt> strsplit\_data = FOREACH emp\_data GENERATE (id,name), STRSPLIT (name,'\_',2);

**Debugging Pig Latin**

1. Use the DESCRIBE operator to review the schema of a relation.  
2. Use the EXPLAIN operator to view the logical, physical, or map reduce execution plans to compute a relation.  
3. Use the ILLUSTRATE operator to view the step-by-step execution of a series of statements.

**Using Comments in Scripts**

1. For multi-line comments use /\* …. \*/  
2. For single line comments use – –

###### Multi-Query Execution

With multi-query execution Pig processes an entire script or a batch of statements at once.

Multi-query execution is turned on by default. To turn it off and revert to Pig’s “execute-on-dump/store” behavior, use the “-M” or “-no\_multiquery” options.

1. $ pig -M myscript.pig
2. or
3. $ pig -no\_multiquery myscript.pig

Error

DEFINE udf onError("myudf")

A = load 'bla';

temp\_B = foreach A generate udf(\*);

SPLIT temp\_B INTO B IF code = 0, B\_ERROR IF code = 1;

B = FOREACH B GENERATE result;

B\_ERROR = FOREACH B\_ERROR GENERATE result;

C = load 'morebla';

D = join C by x, B\_error by x;

## example

DEFAULT ONERROR Ignore(); ...

DESCRIBE A;

A: {name: chararray, age: int, gpa: float}

-- fail it more than 1% errors

B1 = FOREACH A GENERATE Foo(age, gpa), Bar(name) ONERROR FailOnThreshold(0.01) ;

-- custom handler that counts errors and logs on the client side

C1 = FOREACH A GENERATE Foo(age, gpa), Bar(name) ONERROR CountMyErrors() ;

-- uses default handler and SPLIT

B2 = FOREACH A GENERATE Foo(age, gpa), Bar(name) ONERROR SPLIT INTO B2\_ERRORS;

-- B2\_ERRORS can not really contain the input to the UDF as it would have a different schema depending on what UDF failed

DESCRIBE B2\_ERRORS;

B2\_ERRORS: {input: (name: chararray, age: int, gpa: float), udf: chararray, error:(class: chararray, message: chararray, stacktrace: chararray) }

-- example of filtering on the udf

C2 = FOREACH A GENERATE Foo(age, gpa), Bar(name) ONERROR SPLIT INTO C2\_FOO\_ERRORS IF udf='Foo', C2\_BAR\_ERRORS IF udf='Bar';

-- uses handler and SPLIT

A3 = FOREACH A GENERATE Foo(age, gpa), Bar(name) ONERROR HandleItMyWay() SPLIT INTO A3\_ERRORS;

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | -- Problem Stmt : find the number of items bought by each customer  -- which item he/she bought highest time.  -- load the input data :: Schema ( customerId , itemId , order Date, delivery Date );  orders = load '/testData100k' using PigStorage(',') as (cstrId:int, itmId:int, orderDate: long, deliveryDate: long );  -- group by  custorer-id and item-id  grpd\_cstr\_itm = group orders by (cstrId,itmId);  grpd\_cstr\_itm\_cnt = foreach grpd\_cstr\_itm generate group.cstrId as cstrId, group.itmId as itmId, COUNT(orders) as itmCnt;  -- group by cstrId  grpd\_cstr = group grpd\_cstr\_itm\_cnt by cstrId ;  describe grpd\_cstr;  -- grpd\_cstr: {group: int,grpd\_cstr\_itm\_cnt: {cstrId: int,itmId: int,itmCnt: long}}  -- iterate over grpd\_cstr\_itm and find total number of items bought by customer and which item he/or she bought higest time.  result = foreach grpd\_cstr{      total\_orders = SUM(grpd\_cstr\_itm\_cnt.itmCnt);      srtd\_orders = order grpd\_cstr\_itm\_cnt by itmCnt desc;      higest\_bought = limit srtd\_orders 1;      generate FLATTEN(higest\_bought),total\_orders as totalCnt;  };  -- result will contains ( customer\_id , itm\_id\_bought\_higest\_times, number\_of\_times\_it\_bought, total\_items);  describe result;  -- result: {higest\_bought::cstrId: int,higest\_bought::itmId: int,higest\_bought::itmCnt: long,totalCnt: long} |