RELATIONAL OPERATIONS IN APACHE PIG

Relational Operations in Apache Pig

- Order By
- Limit
- Rank
- Distinct
- Sample
- Group By
- Split
- Union
- Cross
- Cogroup
- Join

ORDER BY

- Sorts a relation based on (one or more) fields, ascending or descending
- The default sort behavior is ascending

```
grunt> describe A;

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

grunt> B = ORDER A BY age ASC, gpa DESC;
```

ORDER BY

```
B = ORDER A BY age ASC, gpa DESC;
```

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

analyticscenter

LIMIT

- Limits the number of output rows of a relation
- When performed after ORDER .. BY, LIMIT can be used for top-n behavior
- Otherwise, there is no guarantee in which n rows would be returned, and no M/R jobs are launched for performing limit (which makes LIMIT operator very useful for debugging)

```
grunt> describe B;

B:{name: chararray, age: int, gpa: chararray}

grunt> C = LIMIT B 10;
```

LIMIT

```
C = LIMIT A 3;
```

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	gpa
Sade	19	2.78
Russel	18	2.3
Alice	18	2.87

RANK

- Returns each row with its rank based on an ordering criterion
- The resulting relation is sorted by the rank column
- The name of the rank column is rank_<relation-name>
- If more than one rows tie on the sort field, they get the same rank

```
grunt> describe A;

A:{name: chararray, age: int}

grunt> B = RANK A BY age ASC;
grunt> describe B;

B:{rank_A: long, name: chararray, age: int}
```

RANK

```
B = RANK A BY age;
```

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

rank_A	name	age
1	Alice	18
1	Russel	18
3	Sade	19
3	Ralph	19
5	Addison	21
5	Bob	21
7	Karen	22
8	Ivana	24
9	Linda	25

DISTINCT

- Removes the duplicate rows in a relation:

```
grunt> B = DISTINCT A;
```

DISTINCT

- Passing a subset of fields to DISTINCT is not allowed
- To have the same effect, a projection (FOREACH .. GENERATE) followed by DISTINCT (possibly in a nested block) can be performed

```
# equivalent of SELECT DISTINCT(age) from A;
grunt> B = FOREACH A GENERATE age;

grunt> C = DISTINCT B;
```

SAMPLE

- Returns a random sample of data

```
# SAMPLE <relation> <size_expr>;

grunt> B = SAMPLE A 0.1;
```

GROUP BY

```
B = GROUP A BY 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

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

GROUP BY

- Groups data into one or more bags
- The GROUP operator groups together the rows that have the same group key

```
grunt> describe A;

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

grunt> B = GROUP A BY name;

grunt> describe B;

B:{group: chararray, A: {(name: chararray, age:int, gpa: chararray)}
```

GROUP BY

- Notice the schema of the resulting relation
- It is a relation with one tuple (row) per group
- It contains two fields named as : i) 'group', ii) the original relation name

```
grunt> describe A;

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

grunt> B = GROUP A BY name;

grunt> describe B;

B:{group: chararray, A: {(name: chararray, age:int, gpa: chararray)}
```

GROUP BY

- ‘group’ field holds the value of the expression by which we perform grouping

```
grunt> describe A;

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

grunt> B = GROUP A BY name;

grunt> describe B;

B:{group: chararray, A: {(name: chararray, age:int, gpa: chararray)}
```

GROUP BY

- Per group, a bag field holding the set of rows associated with that group is created

```
grunt> describe A;

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

grunt> B = GROUP A BY name;

grunt> describe B;

B:{group: chararray, A: {(name: chararray, age:int, gpa: chararray)}
```

GROUP BY

- Then, aggregations can be performed on the entire group

```
grunt> describe A;

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

grunt> B = GROUP A BY name;

grunt> describe B;

B:{group: chararray, A: {(name: chararray, age:int, gpa: chararray)}

grunt> C = FOREACH B GENERATE group as name, COUNT(A) as cnt;
grunt> describe C;

C:{name: chararray, cnt: long}
```

GROUP BY

```
B = GROUP A BY 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

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

analyticscenter

GROUP BY

```
C = FOREACH B GENERATE group as age, COUNT(A) as cnt;
```

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

age	cnt
18	2
19	2
21	2
22	1
24	1
25	1

GROUP BY

- Then, aggregations can be performed on a field via the dereference operator

```
grunt> describe A;

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

grunt> B = GROUP A BY name;

grunt> describe B;

B:{group: chararray, A: {(name: chararray, age:int, gpa: chararray)}

grunt> C = FOREACH B GENERATE group as name, AVG(A.age) as avg_age;
grunt> describe C;

C:{name: chararray, avg_age: double}
```

GROUP ALL

- GROUP ALL treats the entire table as one group
- This is the required first step to perform table-level aggregations in Pig

```
# Same as SELECT COUNT(*) FROM A;
grunt> describe A;

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

grunt> B = GROUP A ALL;

grunt> describe B;

B:{group: chararray, A: {(name: chararray, age:int, gpa: chararray)}

grunt> C = FOREACH B GENERATE COUNT(A) as cnt;
```

GROUP ALL

- GROUP ALL treats the entire table as one group
- This is the required first step to perform table-level aggregations in Pig

```
# Same as SELECT AVG(age) FROM A;
grunt> describe A;

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

grunt> B = GROUP A ALL;

grunt> describe B;

B:{group: chararray, A: {(name: chararray, age:int, gpa: chararray)}

grunt> C = FOREACH B GENERATE AVG(A.age) as avg_age;
```

GROUP ALL

```
B = GROUP A all;
```

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

group	A
all	{(Alice, 18, 2.87), (Russel, 18, 2.3), (Ralph, 19, 2.2), (Ivana, 24, 2.9), (Sade, 19, 2.78), (Addison, 21, 2.09), (Bob, 21, 3.5), (Karen, 22, 1.4) , (Linda, 25, 3.3)}

GROUP ALL

```
C = FOREACH B GENERATE COUNT(A) as cnt;
```

group	A
all	{(Alice, 18, 2.87), (Russel, 18, 2.3), (Ralph, 19, 2.2), (Ivana, 24, 2.9), (Sade, 19, 2.78), (Addison, 21, 2.09), (Bob, 21, 3.5), (Karen, 22, 1.4) , (Linda, 25, 3.3)}

cnt
9

Demo

Demonstration of the GROUP BY Operator

Demo

Demonstration of the GROUP ALL Operator

Demo

Single-dataset Relational Operators

SPLIT

- Creates multiple relations based on criteria
- The resulting relations may not be disjoint (this solely depends on the splitting criteria)
- A tuple (row) may not be assigned into a resulting relation

```
grunt> SPLIT A
        INTO B IF age>25
        INTO C IF age>40
        INTO D IF (age>25 AND age<=40);
```

UNION

- Merges multiple relations into one
- By default, merging is position based
- Duplicates are not removed
- Order is not preserved

```
-- A: (a1:long, a2:long)
-- B: (b1:long, b2:long, b3:long)
-- C = A union B: null

C = UNION A, B;

/* C will have 3 fields accessible by position ($i),
and it will have a null schema when two relations
have different number of fields */
```

UNION

- Merges multiple relations into one
- By default, merging is position based
- Duplicates are not removed
- Order is not preserved

```
-- A: (a1:long, a2:long)
-- B: (b1:float, b2:long)
-- C = A union B: (a1:float, a2:long)

C = UNION A, B;

/* C will have 2 fields accessible by field names of
the first relation, and it will have a schema of
escalate types when two relations have compatible
schemas*/
```

UNION

- Merges multiple relations into one
- By default, merging is position based
- Duplicates are not removed
- Order is not preserved

```
-- A: (a1:long, a2:long)
-- B: (b1: (b11:long, b12:int), b2:long)
-- C = A union B: (a1:bytearray, a2:long)

C = UNION A, B;

/* C will have 2 fields accessible by field names of
the first relation, and the fields representing the
union of incompatible types would have a bytearray
type*/
```

UNION

```
C = UNION A, B;
```

a1	a2	a3
1	2	3
4	2	1

b1	b2
2	4
8	9
1	2
3	1
4	1

1	2	3
4	2	1
2	4	
1	2	
3	1	
4	1	
8	9	

UNION ONSCHEMA

- When UNION is wanted to applied to multiple relations on field aliases (based on field name rather than the position) UNION ONSCHEMA operator can be used

```
-- A: (a1:long, a2:long)
-- B: (a1:long, b1:long, b2:long)
/*C = A union onschema B: (a1:long, a2:long, b1:long,
b2:long)*/

C = UNION ONSCHEMA A, B;
```

UNION ONSCHEMA

```
C = UNION ONSCHEMA A, B;
```

a1	a2
2	4
8	9
1	2
3	1
4	1

a1	b1	b2
1	2	3
4	2	1

a1	a2	b1	b2
2	4		
8	9		
1	2		
3	1		
4	1		
1		2	3
4		2	1

CROSS

- Computes the cross product of multiple relations:

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

grunt> describe B;
B: {name: chararray, gpa: float}

grunt> C = CROSS A, B;
grunt> describe C;

C: {A::name: chararray, A::email: chararray, B::name:
chararray, B::gpa: float}
```

CROSS

```
C = CROSS A, B;
```

a1	a2
2	4
8	9
1	2

A::a1	A::a2	B::a1	B::b1	B::b2
2	4	1	2	3
2	4	4	2	1
8	9	1	2	3
8	9	4	2	1
1	2	1	2	3
1	2	4	2	1

a1	b1	b2
1	2	3
4	2	1

COGROUP

- Groups data in more than one relations

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

grunt> describe B;
B: {name: chararray, course: int, gpa: float}

grunt> C = COGROUP A BY name, B BY name;
grunt> describe C;

C: {group: chararray, A: {name: chararray, email:
chararray}, B:{name: chararray, course: int, gpa:
float}}
```

COGROUP

- The behavior of COGROUP is exactly same as the ordinary GROUP, i.e. it creates a row per unique key in the resulting relation
- However, a row has multiple bag fields (as many as the number of input relations) this time, each of which containing the collection of tuples from the corresponding relation with the matching key field

COGROUP

- COGROUP can be interpreted as un-flattened (nested) JOIN
- If no tuple in a relation matches the key, the corresponding bag is empty

COGROUP

```
C = COGROUP A BY name, B BY name;
```

name	age
Addison	21
Sade	19
Ivana	24

name	course	gpa
Sade	1	2.78
Ivana	2	2.9
Addison	2	3.3
Ivana	3	2.3
Sade	2	3.2

group	A	B
Addison	{(Addison, 21)}	{(Addison, 2, 3.3)}
Sade	{(Sade, 19)}	{(Sade, 1, 2.78), (Sade, 2, 3.2)}
Ivana	{(Ivana, 24)}	{(Ivana, 2, 2.9), (Ivana, 3, 2.3)}

analyticscenter

INNER JOIN

- Performs the inner join of multiple fields based on field expressions:

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

grunt> describe B;
B: {name: chararray, course: int, gpa: float}

grunt> C = JOIN A BY name, B BY name;
grunt> describe C;

C: {A::name: chararray, A::email: chararray, B::name:
chararray, B::course: int, B::gpa: float}
```

INNER JOIN

```
C = JOIN A BY name, B BY name;
```

name	age
Addison	21
Sade	19
Ivana	24
Tom	22

name	course	gpa
Sade	1	2.78
Ivana	2	2.9
Addison	2	3.3
Ivana	3	2.3
Sade	2	3.2
Linda	1	2.7

A::name	A::age	B::name	B::course	B::gpa
Addison	21	Addison	2	3.3
Sade	19	Sade	1	2.78
Sade	19	Sade	2	3.2
Ivana	24	Ivana	3	2.3
Ivana	24	Ivana	2	2.9

OUTER JOINS

- Pig also supports RIGHT, LEFT, OR FULL OUTER JOINS
- Only two-way outer join is supported

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

grunt> describe B;
B: {name: chararray, course: int, gpa: float}

grunt> C = JOIN A BY name LEFT OUTER, B BY name;
grunt> describe C;

C: {A::name: chararray, A::email: chararray, B::name:
chararray, B::course: int, B::gpa: float}
```

LEFT OUTER JOIN

```
C = JOIN A BY name LEFT OUTER, B BY name;
```

name	age
Addison	21
Sade	19
Ivana	24
Tom	22

name	course	gpa
Sade	1	2.78
Ivana	2	2.9
Addison	2	3.3
Ivana	3	2.3
Sade	2	3.2
Linda	1	2.7

A::name	A::age	B::name	B::course	B::gpa
Addison	21	Addison	2	3.3
Sade	19	Sade	1	2.78
Sade	19	Sade	2	3.2
Ivana	24	Ivana	3	2.3
Ivana	24	Ivana	2	2.9
Tom	22			

RIGHT OUTER JOIN

```
C = JOIN A BY name RIGHT OUTER, B BY name;
```

name	age
Addison	21
Sade	19
Ivana	24
Tom	22

name	course	gpa
Sade	1	2.78
Ivana	2	2.9
Addison	2	3.3
Ivana	3	2.3
Sade	2	3.2
Linda	1	2.7

A::name	A::age	B::name	B::course	B::gpa
Addison	21	Addison	2	3.3
Sade	19	Sade	1	2.78
Sade	19	Sade	2	3.2
Ivana	24	Ivana	3	2.3
Ivana	24	Ivana	2	2.9
		Linda	1	2.7

FULL OUTER JOIN

```
C = JOIN A BY name RIGHT OUTER, B BY name;
```

name	age
Addison	21
Sade	19
Ivana	24
Tom	22

name	course	gpa
Sade	1	2.78
Ivana	2	2.9
Addison	2	3.3
Ivana	3	2.3
Sade	2	3.2
Linda	1	2.7

A::name	A::age	B::name	B::course	B::gpa
Addison	21	Addison	2	3.3
Sade	19	Sade	1	2.78
Sade	19	Sade	2	3.2
Ivana	24	Ivana	3	2.3
Ivana	24	Ivana	2	2.9
		Linda	1	2.7
Tom	22			

Lab

Relational Operators

Relational Operations in Apache Pig

End of Chapter