5-2011

Templates for Supporting Sequenced Temporal Semantics in Pig Latin

Dhaval Deshpande
Utah State University

Templates for Supporting Sequenced Temporal Semantics in Pig Latin

by

Dhaval Deshpande

A report submitted in partial fulfillment
of the requirements for the degree
of

MASTER OF SCIENCE

in

Computer Science

Approved:

Dr. Curtis Dyreson
Major Professor

Dr. Stephen Clyde
Committee Member

Dr. Stephen Allan
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah

2011

ABSTRACT

Templates for Supporting Sequenced Temporal Semantics in Pig Latin

by

Dhaval Deshpande, Master of Science

Utah State University, 2010

Major Professor: Dr Curtis Dyreson
Department: Computer Science

This report describes proposed templates for supporting sequenced temporal semantics in Pig Latin, a dataflow language used primarily for the analysis of very large data sets. Sequence semantics says that if we take a relation and divide it into smaller relations based on timestamps, while still carrying out the regular Pig Latin program over it, the result should be the same as when carrying out the temporal Pig Latin program over the original relation. In real time, the relations can be enormous, and dividing such relations into smaller ones based on every possible timestamp creates an extremely large number of smaller relations. Hence, we create temporal programs, which eliminates the need to divide a relation into smaller relations and carry out additional operations over those smaller relations. We look at each of the templates and discuss their functionality. One example of such a template is temporal grouping, which provides an ability to group a set of tuples or a whole relation based on

timestamps. Using temporal grouping, a user can find the number of tuples that exist at a given point of time. Another example is temporal coalescing, which allows a user to project multiple tuples and the timestamps of their existence in the database. We compare the complexity of the templates with the existing operations.

ACKNOWLEDGMENTS

I would like to thank Dr. Curtis Dyreson for giving me the opportunity to be a part of this project. His guidance and support have been invaluable. I owe my knowledge of databases to him. I would like to thank my parents for their continuous support and encouragement ,without which this project would not have been possible. Lastly, I would like to thank my friend Swathi Rajashekar for her invaluable support and interest in my project.

Dhaval Deshpande

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGMENTS v

LIST OF FIGURES ....

CHAPTER

1 INTRODUCTION 1

1.1 Sequenced Temporal Semantics in Pig Latin 2

2 TEMPLATES FOR TEMPORAL OPERATIONS IN PIG LATIN 4

2.1 Coalesce 6
2.2 Cogroup 8

2.2.1. Non Temporal Cogroup Result 9
2.2.2. Temporal Cogroup Result 9

2.3 Cross Join 12

2.3.1. Non Temporal Cross Join Result on S_a and S_b 12
2.3.2. Temporal Cross Join Result on S_a and S_b 13

2.4 Distinct 15
2.4.1. Non Temporal Distinct Result on S 15
2.4.2. Temporal Distinct Result on S 15

2.5 Filter 16
2.6 Foreach 16
2.7 Full Outer Join 17

2.7.1. Non Temporal Full Outer Join Result on S_a and S_b 17
2.7.2. Partial Temporal Full Outer Join Result on S_a and S_b 18
2.7.3. Expected Temporal Full Outer Join Result 18

2.8 Inner Join 21
2.8.1. Non Temporal Inner Join Result on S_a and S_b 21

2.8.2. Temporal Inner Join Result on S_a and S_b 21

2.9 Left Outer Join 23

2.9.1. Non Temporal Left Outer Join Result on S_a and S_b 23
2.9.2. Temporal Left Outer Join Result on S_a and S_b 23
2.9.3. Expected Temporal Left Outer Join Result 23

2.10 Right Outer Join 26

2.10.1. Non Temporal Right Outer Join Result on S_a and S_b 26
2.10.2. Temporal Right Outer Join Result on S_a and S_b 26
2.10.3. Expected Temporal Left Outer Join Result 27

2.11 Sample 29
2.12 Split 29
2.13 Temporal Group 30
2.14 Union 35

3 COMPLEXITY OF TEMPORAL PIG LATIN OPERATIONS 36

3.1 Hardware Specifications 36
3.2 Operating System Specification 36
3.3 Software Specification 36
3.4 Analysis 39

4 SUMMARY AND FUTURE WORK 40

4.1 Future Work 40

REFERENCES 41
APPENDICES ....

Appendix A 43
Appendix B 48

LIST OF FIGURES

FIGURE 1. COALESCE PATTERN 6

FIGURE 2. COGROUP PATTERN 10

FIGURE 3. CROSS JOIN PATTERN 14

FIGURE 4. FULL OUTER JOIN PATTERN 20

FIGURE 5. INNER JOIN PATTERN 22

FIGURE 6. LEFT OUTER JOIN PATTERN 25

FIGURE 7. RIGHT OUTER JOIN PATTERN 28

FIGURE 8. TEMPORAL GROUP PATTERN 31

FIGURE 9. GRAPH OF FILE SIZE VS EXECUTION TIME FOR TEMPORAL OPERATIONS 37

FIGURE 10. GRAPH OF FILE SIZE VS EXECUTION TIME FOR TEMPORAL CROSS OPERATION 37

FIGURE 11. COMPLEXITY GRAPH 38

CHAPTER 1

INTRODUCTION

Pig Latin is a dataflow language[3] used primarily for analyzing very large data sets [1] . Because it allows storage and projection of the intermediate results, Pig Latin lends itself well to data transformations like filter and group. Some of the key properties of Pig Latin are:

1. **Ease of programming:** Pig Latin makes trivial achieving parallel execution of simple, "embarrassingly parallel" data analysis tasks. Complex tasks comprised of multiple interrelated data transformations are explicitly encoded as data flow sequences, making them easy to write, understand, and maintain.
2. **Optimization opportunities:** The way in which tasks are encoded in Pig Latin permits the system to optimize their execution automatically, allowing the user to focus on semantics rather than efficiency.
3. **Extensibility:** Users can create their own functions to do special-purpose processing.

Pig Latin programs run in a distributed fashion on a cluster. For quick prototyping, Pig Latin programs can also run in "local mode" without a cluster [2]. Below is a sample Pig program.

```
A = load 'input' using PigStorage('\t') as (name: chararray,begin: int, end: int);

R = FOREACH A generate name;

store R into 'output.txt' using PigStorage('\t');
```

Here, *A* is the relation that contains the contents of the file “input”. “*PigStorage('\t')*” extracts the data from the file delimited by tab. We next project the names from relation *A* and store them in an intermediate relation *R*. Finally, we store *R* in a separate file.

In this project we focus primarily on creating templates for supporting sequenced temporal semantics in Pig Latin. Sequence semantics says that if we take a relation and divide it into smaller relations based on timestamps, while still carrying out the regular Pig Latin program over it, the result should be the same as when carrying out the temporal Pig Latin program over the original relation. In real time, the relations can be enormous, and dividing such relation into smaller relation based on every possible time stamp can creates an extremely large number of smaller relations. Hence, we create temporal programs, which eliminates the need to divide a relation into smaller relations and carry out additional operations over those smaller relations. The time stamps considered here are discrete integers.

1.1 Sequenced Temporal Semantics in Pig Latin

The templates for sequenced temporal operations created for this project enhance the functionality of existing Pig Latin operations.

Now, let's examine the join operation in Pig Latin. The sample code below shows how a join operation is carried out in Pig Latin.

```
A = LOAD '../file_1mb.txt' using PigStorage('\t') as (name: chararray, b: int, e: int);

B = LOAD '../file_1mb.txt' using PigStorage('\t') as (name1: chararray, b1: int, e1: int);

R = JOIN B by name1, A by name;
```

The above Pig Latin program joins relation *A* with relation *B* by name. The temporal version of a join operation does more than just the inner join by name. A temporal join first triggers the inner join on two relations. It then projects only those timestamps of relation *A* whose start times are greater than the start time of relation *B* and whose end times of relation *A* are less than the end time of relation *B* .

```
A = load '../file_1mb.txt' using PigStorage('\t') as (name: chararray, b: int, e: int);

B = load '../file_1mb.txt' using PigStorage('\t') as (name1: chararray, b1: int, e1: int);

R = JOIN B by name1, A by name;

R1 = FOREACH R generate name,name1,(b>b1?b:b1), (e<e1?e:e1);

R2 = FILTER R1 by b <= e;

R3 = DISTINCT R2;
```

The above Pig Latin program gives the temporal version of a join operation in Pig Latin. The program carries out a regular inner join in Pig Latin and stores it in an intermediate relation *R*. We then project only the maximum of starting timestamps and minimum of ending timestamps from relation *R*, which are basically the intersection of timestamps, and store this information in relation *R1*. We next filter out the tuples in *R1* whose begin time is less than or equal to end timestamps and store this information in relation *R2*. Finally, we filter out duplicates from relation *R2*. 1.2 Outline of Report

Chapter 2 looks at other temporal operations in Pig Latin and discusses them in detail. Chapter 3 compares the complexity of regular Pig Latin operations with that of the temporal Pig Latin operations. We compare the complexity by plotting graphs. Chapter 4 provides a summary and the future work for the project. Appendices A and B contain the code developed for implementation of the proposed templates.

CHAPTER 2

TEMPLATES FOR TEMPORAL OPERATIONS IN PIG LATIN

In the previous chapter, we look at how Pig Latin programs work, as well as some of its salient features. We also highlighted the template for a temporal join operation. This chapter covers the other temporal operations in Pig Latin. Definitions of the

temporal operations discussed herein follow:

- Coalesce.

  Temporal coalescing helps a user to find at a tuple that has previously existed in the database.

- Cogroup.

  The cogroup operation does an inner join between two relations and groups the tuples based on a join key.

- Cross.

  The cross operation carries out a cross join between two or more relations.

- Distinct.

  The distinct operation removes any duplicate tuples from the relation.

- Filter.

  The filter operation filters a relation based on the given criteria.

- Foreach.

  The foreach operation is used to project the columns from a relation.

- Full outer join.

This operation carries out a regular full outer join between two relations.

- Inner join.

  This operation carries out an inner join between two relations.

- Left outer join.

  This operation carries out a left outer join between two relations.

- Right outer join.

  This operation carries out a right outer join between two relations.

- Sample.

  This operation projects random tuples from a relation based on certain probability.

- Split.

  This operation splits a relation into two or more relations based on a certain criteria.

- Temporal group.

  This operation provides an ability to group a set of tuples or a whole relation based on timestamps.

- Union.

  A union is used to merge two or more relations.

2.1 Coalesce

Temporal coalescing helps a user to find a tuple that has previously existed in the database. Take a look at Figure 1.

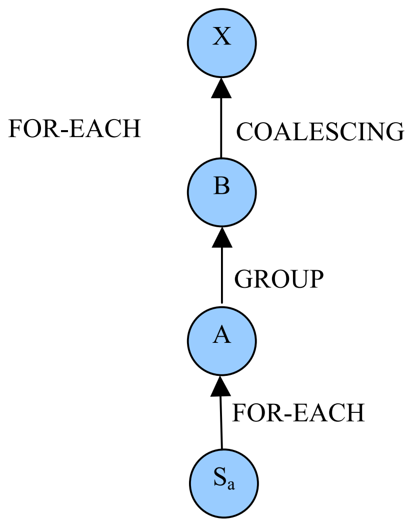


Figure 1. COALESCE PATTERN.

Each node in the above figure is a relation with the start state of the relation S and the end state of the relation X. At each step, the relation undergoes a transformation until it reaches state X, which is the desired output. Let us look at the code below and compare it with the figure given above.

```
Sa = LOAD '../file_1mb.txt' using PigStorage('\t') as (name, st, et);
```

The above code snippet tells the Pig Latin compiler to load the contents of a file into relation *S_a*. Our relation has only three columns, name, start time, and end time. Relation *S_a* here is node *S_a* in Figure 1. As shown in Figure 1, *S_a* undergoes a transformation to produce node *A*, which can be verified by the code snippet below.

```
A = FOREACH Sa generate name, st,et;
```

Here, the foreach operation is used to project the out name, start time, and end time out of relation *S_a*. We can see that step 2 of this operation is redundant since it generates the same data as in relation S_a. We need do this operation, however, because in real time tables can have lots of attributes, and we need to project the out name, start time, and end time to carry out the coalesce operation.

We then carry out a group operation on relation *A*. This operation groups the data in *A* by name and generates new node *B*.

```
B = GROUP A by name;
```

Now, we go through each group of *B* and carry out a coalesce operation on them. The coalesce function takes the group of timestamps for a single group in *B*, returns the intended output ,and stores it in relation X. Relation X now has all the timestamps at which a single tuple existed in database.

```
X = FOREACH B generate FLATTEN(myudfs.Coalesce(group,A)) as name;
```

The user-defined function *myudfs.Coalesce* shown in the snippet above is the user-defined function written in Java. We pass two arguments to this function. The first is the attribute by which the columns were grouped, and the second argument is the collection of timestamps (bag) related to the first argument. This is explained further by the example below.

a	1	6
a	2	8
b	0	3
a	9	11

Relation S_a

In this example, we carry out a coalesce operation on relation $\boldsymbol{S_a}$. As seen above, we first project the out name, start time, and end time, and then we store this result in relation A. Next, we carry out the group operation on relation A and store the result in relation B. The result produced by the group operation contains the name by which it is grouped and the bag containing the tuples that fall into this group. The sample output is shown below.

(a, {(a,1,6), (a,2,8), (a,9,11)})

(b, {(0,3)})

We now carry out our user-defined function (UDF) myudfs.Coalesce on relation B. The arguments passed to this function are the name and the bag of relation B. We store the output of the coalesce function in relation X which contains the desired output as shown below.

(a, {(1,8), (9,11)})

(b, {0,3})

2.2 Cogroup

a cogroup operation in Pig Latin carries out an inner join between two tables and groups the resultant data by the join key. A temporal Cogroup also performs the same operation, but it takes the timestamps into account as well. A temporal operation preserves the overlapping timestamps. Thus, a temporal operation first performs the regular cogroup operation and then makes a series of operations over this result to maintain the overlapping timestamps. The example below shows two sample relations S_a and S_b. Both the relations have name, begin time, and end time as their tuples. The result of a regular cogroup operation and the temporal cogroup operation are as shown below.

a	1	15
b	5	9
c	0	4
a	4	8

Relation S_b

2.2.1 Nontemporal Cogroup Result

(a, { (a) , (a) } , { (a) , (a) })

(b, { (b) } , { (b) })

(c, { } , { (c) })

(d, { (d) } , { })

The first attribute here is the join key. The second attribute is a bag that contains all the tuples in relation S_a, that come under a particular group. Similarly, the third attribute is also a bag containing the satisfying tuples of Relation S_b.

2.2.2 Temporal Cogroup Result

(a,a,1,6)

(a,a,2,8)

(a,a,4,6)

(a,a,4,8)

(d,,8,10)

(,c,0,4)

The above result shows that the temporal operation on the regular cogroup preserves the overlapping timestamps. Figure 2 shows the pattern to create a temporal cogroup operation.

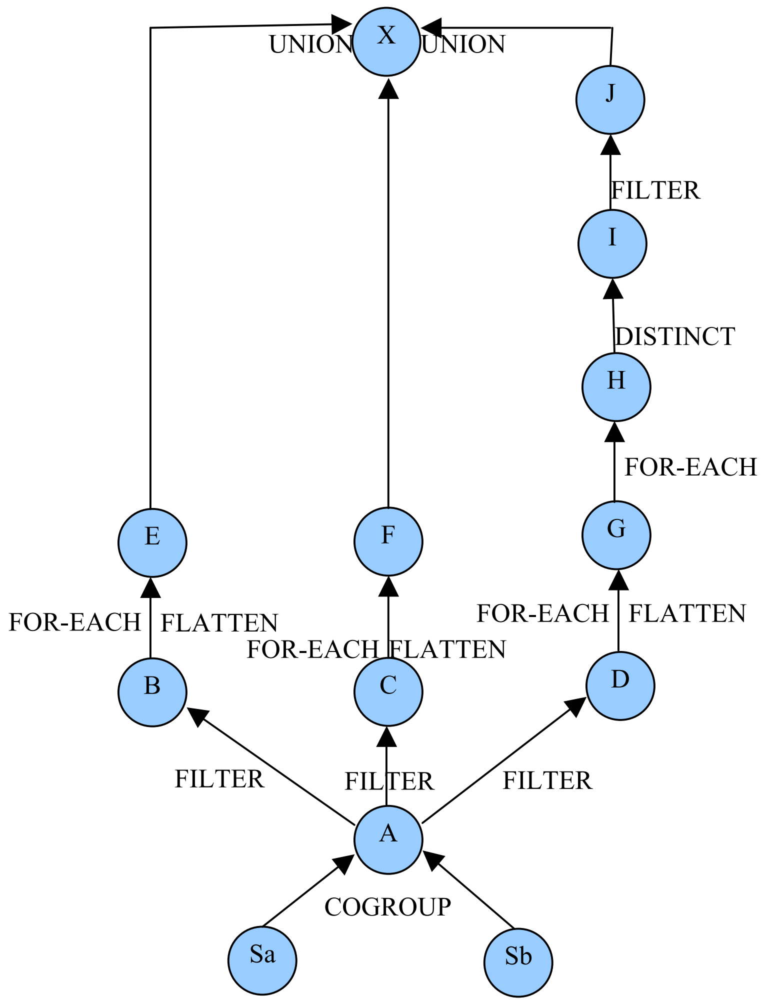


Figure 2. COGROUP PATTERN

In Figure 2, there are two start states S_a and S_b. We perform a regular cogroup operation on these, and the output is stored in relation *A*, as shown in the code below.

```
A = COGROUP S_a by name, S_b by name1;
```

We then carry out three separate filter operations on relation *A*, based on any empty bags in S_a or S_b and store the result in *B*, *C,* and *D* respectively, as shown.

```
B = FILTER A by IsEmpty(S_a);
C = FILTER A by IsEmpty(S_b);
D = FILTER A by not IsEmpty(S_a) and not IsEmpty(S_b);
```

We again go through each tuple in relations *B*, *C,* and *D* and carry out a flatten operation over those tuples. We then store the resultant data in relations *E*, *F,* and *G,* respectively. A flatten operation works exactly the opposite of a group operation. Flatten is used to ungroup the grouped data.

```
E = FOREACH B generate FLATTEN(S_b);
F = FOREACH C generate FLATTEN(S_a);
G = FOREACH D generate FLATTEN(S_a), FLATTEN(S_b);
```

We then go through each tuple of *G,* project out tuples, and store the resultant data in relation *H*. This operation gives us the overlapping timestamps.

```
H = FOREACH G generate name,(b>b1?b:b1) as b, (e<e1?e:e1) as e;
```

At this point, we want to remove any duplicate tuples that exist in the table. Hence, we do a distinct operation on relation *H* to produce relation *I*.

```
I = DISTINCT H;
```

We again carry out a filter operation on *I* to remove any of the tuples that have a begin time greater than the end time, and we store the output in relation *J*.

```
J = FILTER I by b <= e;
```

Finally, we do a union of relation *E*, *F,* and *J* to produce the desired relation *X*.

```
X = UNION E, F, J;
```

2.3 Cross Join

A temporal cross join does the regular cross join on two relations and performs a series of operations to preserve only the overlapping timestamps. The result of a nontemporal cross join and a temporal cross join on the above two relations, S_a and S_b are shown in Sections 2.31 and 2.3.2 below. Figure 3 shows the pattern to create a temporal cross join operation.

2.3.1 Nontemporal Cross Join Result on S_a and S_b

(b, c)

(b, a)

(b, b)

(a, c)

(a, a)

(a, b)

(d, c)

(d, a)

(d, b)

2.3.2 Temporal Cross Join Result on S_a and S_b

(a,a,1,6)

(a,a,2,8)

(a,a,4,6)

(a,a,4,8)

(a,b,5,6)

(a,b,5,8)

(a,c,1,4)

(a,c,2,4)

(b,a,1,3)

(b,c,0,3)

(d,a,8,8)

(d,a,8,10)

(d,b,8,9)

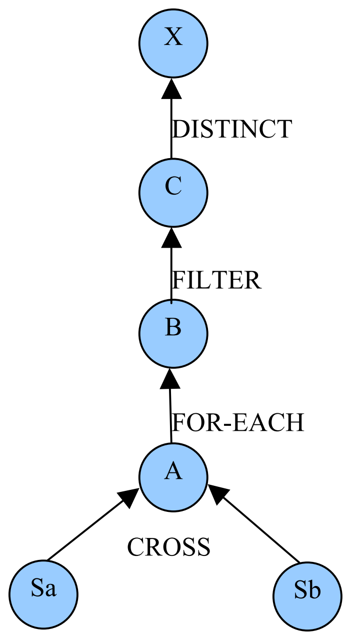


Figure 3. CROSS JOIN PATTERN

As shown in Figure 3, we perform a regular cross join operation on two start states S_a and S_b, and the output is stored in relation *A*. The code for the same is shown below.

```
A = CROSS Sa , Sb;
```

We now iterate through all the tuples of relation *A*, and project out the maximum of start timestamps and minimum of end timestamps. The output of this operation is stored in relation *B*. The code to perform this process is shown below.

```
B = FOREACH A generate name,name1,(b>b1?b:b1), (e<e1?e:e1);
```

We then filter the tuples of relation *B* to eliminate any whose begin time is greater than the end time and store the output in relation *C*.

```
C = FILTER B by b <= e;
```

Finally, we remove any repeated tuples and store the final output in relation *X*.

```
X = DISTINCT C;
```

2.4 Distinct

The distinct operation removes any repeated tuples in the relation. The temporal distinct operation will be same as Coalesce operation because in coalesce operation we can find various timestamps for the same tuple. For example consider relation *S*:

a
a
b

```
A = DISTINCT S;
```

2.4.1 Non Temporal Distinct Result on S

When we carry out a regular distinct operation as shown above on relation *S* as shown above, we get the following output.

(a)

(b)

2.4.2 Temporal Distinct Result on S

But when we perform a temporal distinct or coalesce operation on relation S_a, as explained previously, we get the following output.

(a, {(1,8), (9,11)})

(b, {0,3})

2.5 Filter

A filter operation filters the tuples of a relation, based on any given criteria. Similar to the above distinct operation, a temporal filter does not make any changes to the timestamps. Thus, like the temporal filter operation, a regular filter operation remains the same.

```
A = FILTER Sa by name is not null;
```

As an example, consider relation *S* shown below.

a	1
null	2

When we perform a filter operation on *S* based on the condition that the name should not be null, we see that the timestamps are not modified. Hence, the temporal operation is the same as the regular filter operation. The output of filter operation is

(a,1)

2.6 Foreach

This operation iterates through all the tuples of a given relation. The foreach operation also does not make any changes to the tuples and more precisely to the timestamps. Therefore, the temporal foreach operation also remains the same as a regular operation. The foreach operation on a start state S_a is shown below.

```
A = FOREACH Sa generate name,b,e;
```

As an example, consider relation *S*.

a	1
b	2

When we perform a foreach operation on *S,* we are basically projecting the columns from relation *S.* As we can see, the timestamps are not modified here; hence, the temporal operation is the same as the regular foreach operation. The output of the foreach operation is

(a,1)

(b,2)

2.7 Full Outer Join

A temporal full outer join does the regular full outer join on two relations and performs a series of operations to preserve only the overlapping timestamps, that is the maximum of the start timestamps and the minimum of the end timestamps. We perform a partial temporal full outer join. The results of a nontemporal full outer join, partial temporal full outer join, and the expected temporal full outer join on the above two relations, namely, S_a and S_b, are shown in Sections 2.7.1, 2.7.2, and 2.7.3 below.

2.7.1 Nontemporal Full Outer Join Result on S_a and S_b

(a,a)

(b, b)

(,c)

(d,)

2.7.2 Partial Temporal Full Outer Join Result on S_a and S_b

(a,a,1,6)

(a,a,2,8)

(a,a,4,6)

(a,a,4,8)

(,c,0,4)

(d,,8,10)

2.7.3 Expected Temporal Full Outer Join Result

(a,1,6,a)

(,6,15,a)

(a,4,6,a)

(a,1,4,)

(,6,8,a)

(a,2,8,a)

(a,1,2,)

(,8,15,a)

(a,4,8,a)

(a,2,4,)

(b,0,3,)

(,5,9,b)

(,0,4,c)

(d,8,10,)

As seen from the above results, we have implemented a partial temporal full outer join. That is, the result only contains those tuples in which both the tuples of the left table and the right table existed together. The actual temporal full outer join should even include those timestamps in which only the tuples of the left table existed but not the right table tuples and vice versa. As an example, one of the tuples of a full outer join is

(a,1,6,a,4,8)

Our partial temporal full outer join produces only one tuple with overlapping timestamp, that is

(a,4,6)

The actual temporal full outer join should even include those tuples wherein only tuples of the left table existed and those tuples wherein only tuples of the right table existed.

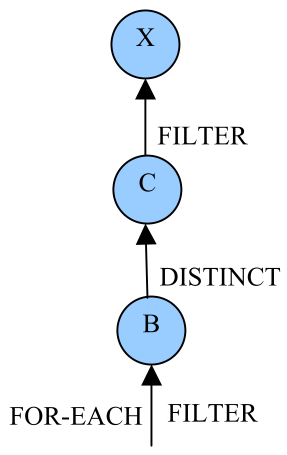
Thus, the expected full outer join result for the above tuple is:

- (a,4,6,a) : Wherein tuples of both tables existed;
- (a,1,4,) : Wherein only tuples of the left table existed.
- (,6,8,a) : Wherein only tuples of the right table existed.

Figure 4 shows the pattern to create a partial temporal full outer join operation. The actual and complete full outer join is left for future work.

Similar to a temporal cross join, we first perform a regular full outer join on the two start states S_a and S_b and store the result in Relation *A*.

```
A = JOIN Sa by name FULL OUTER, Sb by name1;
```



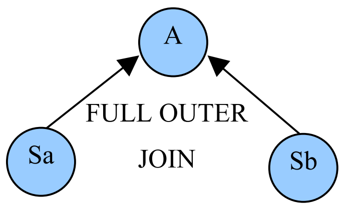


FIGURE 4 . FULL OUTER JOIN PATTERN

A full outer join may result in some empty S_a bags or some empty S_b bags. If a tuple has empty an S_a bag, we directly use S_b. Similarly if S_b bag is empty in a tuple, we use S_a. If both are not empty, we perform further checks and operations to obtain the overlapping begin and end timestamps. The code for this is as below.

```
B = FOREACH A generate ((b is null) ? name1 : name) as name,((b is null) ? b1 : ((b1
is not null) ? ((b > b1) ? b : b1) : b)) as b , ((b is null) ? e1 : ((b1 is not null) ? ((e < e1)
? e : e1) : e)) as e;
```

We finally remove any repeated tuples, storing the result in relation *C* and perform a filter to remove any invalid timestamps. The final output is then stored in relation *X*.

```
C = DISTINCT B;

X = FILTER C by b <= e;
```

2.8 Inner Join

Again, a temporal inner join does the work of a regular inner join on two relations and performs a series of operations to preserve only the overlapping timestamps. The result of a nontemporal inner join and a temporal inner join on the above two example relations, S_a and S_b, are shown below in Sections 2.81 and 2.8.2. Figure 5 shows the series of operations performed on a regular inner join to obtain a temporal inner join.

2.8.1 Nontemporal Inner Join Result on S_a and S_b

(a, a)

(b, b)

2.8.2 Temporal Inner Join Result on S_a and S_b

(a,a,1,6)

(a,a,2,8)

(a,a,4,6)

(a,a,4,8)

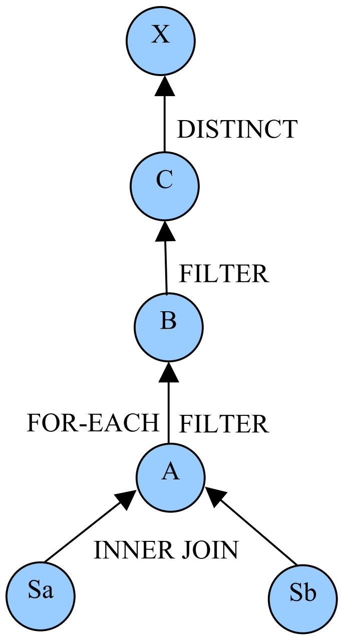


FIGURE 5. INNER JOIN PATTERN

As done earlier, the first step is the regular operation. We perform a regular inner join on the two start states S_a and S_b and store the result in Relation *A*.

```
A = JOIN Sa by name , Sb by name1;
```

Now, we iterate through all the tuples in relation *A,* check for tuples with overlapping timestamps, and store this result in relation *B*.

```
B = FOREACH A generate name,name1,(b>b1?b:b1), (e<e1?e:e1);
```

We finally perform a filter on relation *B* to remove any invalid timestamps, storing the result in relation *C* and removing any repeated tuples. The final output is then stored in relation *X*.

```
C = filter B by b <= e;

X = DISTINCT C;
```

2.9 Left Outer Join

A temporal left outer join does a regular left outer join on two relations and performs a series of operations to preserve only the overlapping timestamps. We perform a partial temporal left outer join. The result of a nontemporal left outer join, partial temporal left outer join, and temporal left outer join on the above two example relations, S_a and S_b, are shown in Sections 2.91, 2.9.2, and 2.9.3 below.

2.9.1 Nontemporal Left Outer Join Result on S_a and S_b

(a, a)

(b, b)

(d,)

2.9.2 Partial Temporal Left Outer Join Result on S_a and S_b

(a,a,1,6)

(a,a,2,8)

(a,a,4,6)

(a,a,4,8)

(d,,8,10)

2.9.3 Expected Temporal Left Outer Join Result

(a,1,6,a)

(a,4,6,a)

(a,1,4,)

(a,2,8,a)

(a,1,2,)

(a,4,8,a)

(a,2,4,)

(b,0,3,)

(d,8,10,)

We have implemented a partial temporal left outer join. That is, the result only contains those tuples in which both the tuples of the left table and right table existed together. But the actual temporal left outer join should also include those timestamps in which only the tuples of the left table existed. As an example, one of the results of a left outer join is (a,1,6,a,4,8)

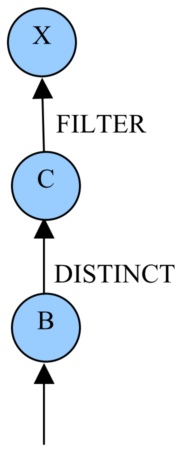
Our partial temporal left outer join produces only one tuple with overlapping timestamps, that is (a,4,6).

The actual temporal left outer join should even include tuples in which only tuples of the left table existed.

Thus, the expected left outer join result for the above tuple is:

- (a,4,6,a) : Wherein tuples of both tables existed;
- (a,1,4,) : Wherein only tuples of the left table existed.

Figure 6 shows the operations carried out on a regular left outer join to obtain the partial temporal left outer join results. The expected temporal left outer join remains for future work.



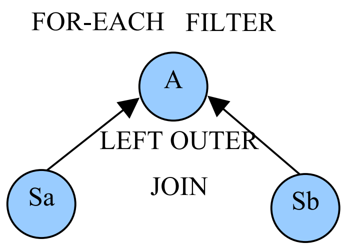


FIGURE 6. LEFT OUTER JOIN PATTERN

As described above, the first step is the regular operation. We perform a regular left outer join on the two start states S_a and S_b and store the result in Relation *A*.

```
A = JOIN Sa by name LEFT OUTER, Sb by name1;
```

A left outer join may result in some empty S_b bags, as can be seen from the above set of results for a nontemporal left outer join. If a tuple has empty S_b bag, we directly use just S_a, but if S_b is not empty, we perform further checks and operations to obtain the overlapping begin and end timestamps. The code for this is below.

```
B = FOREACH A generate name,((b1 is null) ? b : ((b > b1) ? b : b1)) as b,((b1 is null)
? e : ((e < e1) ? e : e1)) as e;
```

We finally remove any repeated tuples, storing the result in relation *C* and perform a filter on the result to remove any invalid timestamps. The final output is then stored in relation X.

```
C = DISTINCT B;

X = FILTER C by b <= e;
```

2.10 RIGHT OUTER JOIN

The right outer join is similar to the left outer join. As in all other operations, the temporal right join also performs a regular right outer join first. It then performs further operations to obtain the temporal results for the same. We have implemented the partial temporal right outer join. The result of a nontemporal right outer join, partial temporal right outer join, and temporal right outer join on the above two example relations, S_a and S_b are shown in Sections 2.10.1, 2.10.2, and 2.10.3 below.

2.10.1 Nontemporal Right Outer Join Result on S_a and S_b

(a, a)

(b, b)

(,c)

2.10.2 Temporal Right Outer Join Result on S_a and S_b

(a,a,1,6)

(a,a,2,8)

(a,a,4,6)

(a,a,4,8)

(,c,0,4)

2.10.3 Expected Temporal Full Outer Join Result

(a,1,6,a)

(,6,15,a)

(a,4,6,a)

(,6,8,a)

(a,2,8,a)

(,8,15,a)

(a,4,8,a)

(,5,9,b)

(,0,4,c)

We have implemented a partial temporal right outer join, whose result only contains those tuples in which the tuples of both the left table and the right table existed together. But the actual temporal right outer join should even include those timestamps in which only the tuples of the right table existed. As an example, one of the results of the right outer join is (a,1,6,a,4,8).

Our partial temporal full outer join produces a single tuple with overlapping timestamps, that is (a,4,6).

The actual temporal full outer join should even include tuples in which only tuples of the left table existed and also tuples in which only tuples of the right table existed.

Thus, the expected right outer join result for the above tuple is:

- (a,4,6,a) : Wherein tuples of both tables existed;
- (,6,8,a) : Wherein only tuples of the right table existed.

Figure 7 shows the series of operations carried out on the regular right out join to obtain the partial temporal right outer join results. The expected temporal right outer join remains for future work.

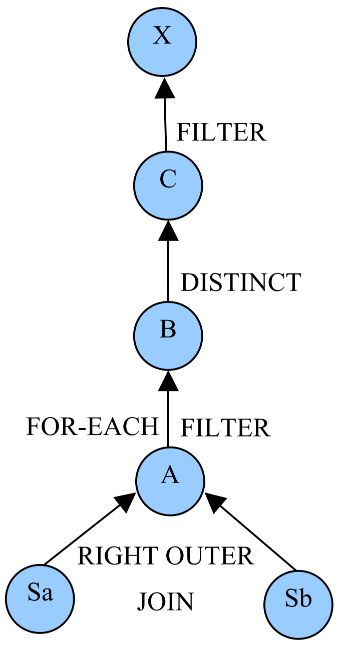


FIGURE 7. RIGHT OUTER JOIN PATTERN

The first step being the regular operation, we perform a regular right outer join on the two start states S_a and S_b and store the result in Relation *A*.

```
A = JOIN Sa by name RIGHT OUTER, Sb by name1;
```

A right outer join may result in some empty S_a bags, as can be seen from the above set of results for a nontemporal right outer join. If a tuple has empty S_a bag, we directly use S_b, but if S_a is not empty, we perform further checks and operations to obtain the overlapping begin and end timestamps. The code for this is below.

```
B = FOREACH A generate ((b is null) ? name1 : name) as name,((b is null) ? b1 : ((b >
b1) ? b : b1)) as b,((b is null) ? e1 : ((e < e1) ? e : e1)) as e;
```

We finally remove any repeated tuples, storing the result in relation *C* and perform a filter on this to remove any invalid timestamps. The final output is then stored in relation *X*.

```
C = DISTINCT B;

X = FILTER C by b <= e;
```

2.11 SAMPLE

Sample operation projects out a random tuple from the relation, based on some given probability. Since sample operation only projects a tuple and does not make any changes to the tuple nor the timestamps, a temporal sample is the same as the regular sample operation. The code for the sample is as follows.

```
A = SAMPLE Sa 0.5;
```

As an example, consider the relation S_a. When we perform a sample operation on S_a , we get the following output.

(a,1,6)

2.12 SPLIT

A split operation splits the given relation into as many relations as specified by a predetermined criteria. This operation, as well, does not modify tuples or timestamps. Thus, the temporal split operation is the same as the regular split operation. The instructions for a split on start state S_a is below.

```
SPLIT S_a INTO X IF b<7, Y IF e==5, Z IF (b<6 OR b>6);
```

As an example when we carry the above operation on S_a,, we get the following output.

- Relation *X*

```
(a,1,6)
(a,2,8)
(b,0,3)
```

- Relation *Y*

```
( )
```

- Relation *Z*

```
(a,1,6)
(a,2,8)
(b,0,3)
(a,9,11)
```

2.13 Temporal Group

A temporal grouping allows a user to find all the tuples that existed at a given point of time. This is a special operation and does not have a corresponding Pig Latin operation.

Figure 8 shows the series of steps for a temporal group operation.

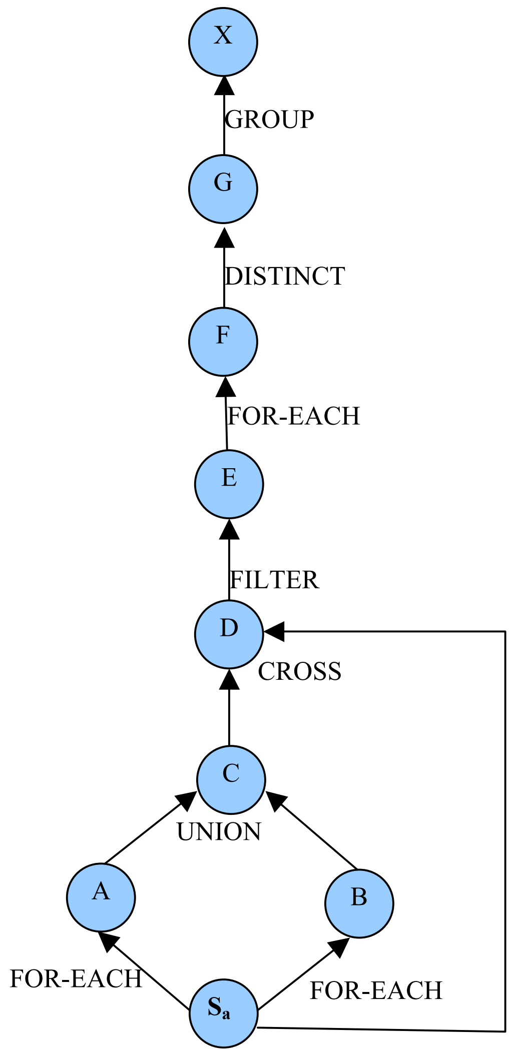


FIGURE 8. TEMPORAL GROUP PATTERN

Below are the steps to create a temporal group operation.

A = FOREACH S_a generate b as b1;

B = FOREACH A generate e as e1;

C = UNION A,B;

We follow a similar approach to what we have been doing in previous sections. In order to find various tuples at a single point t, we need to do a cross join of the relation S_a with the relation that contains only begin and end timestamps. By doing this, we have access to all the tuples that are related to time t.

```
D = CROSS Sa , C;

E = FILTER D by b1 <= e and b1 >= b;
```

We then apply a strategy similar to a join to get the overlapping timestamps and filter out the rest of the tuples. At this point in time, we need to project out only those columns in which we are interested. Hence, we project the name, begin time, and the end time. We now have all the timestamps and corresponding names that existed during that time.

```
F = FOREACH E generate name, b1;

G = DISTINCT F;

X = GROUP G by b1;
```

Before we go ahead and group all those names by their corresponding timestamps, to make sure there is not any data that is redundant or repeated, we carry out a distinct operation followed by a group operation to get the intended output. Let us look at an example before we look at other temporal operations.

We carry out the temporal grouping on relation S_a. We first project out the start time and end time for relation S_a and store them in relations *A* and *B,* respectively. We then carry out the cross operation of S_a with a union of *A* and *B* and store the result in relation *C*. The output of relation *C* is shown below.

```
(a,2,8,0)
```

(a,2,8,1)

(a,2,8,6)

(a,2,8,11)

(a,2,8,8)

(a,2,8,9)

(a,2,8,2)

(a,2,8,3)

(a,1,6,0)

(a,1,6,6)

(a,1,6,1)

(a,1,6,11)

(a,1,6,8)

(a,1,6,9)

(a,1,6,2)

(a,1,6,3)

(a,9,11,0)

(a,9,11,1)

(a,9,11,6)

(a,9,11,11)

(a,9,11,8)

(a,9,11,9)

(a,9,11,2)

(a,9,11,3)

(b,0,3,0)

(b,0,3,6)

(b,0,3,1)

(b,0,3,11)

(b,0,3,8)

(b,0,3,9)

(b,0,3,2)

(b,0,3,3)

We then follow the same strategy to project the overlapping timestamps. We carry out the filter operation over relation *C* and store the result in *D*. We next project the start time and the name from relation *D* and store it in relation *E*. At this point, we want to remove any duplicates that exist in relation *E*. Hence, we carry out the distinct operation and store the distinct tuples of relation *E* in relation *F*. Finally, we group relation *F* by attribute b1 and store it in relation *X*. The output of relation *X* is shown below.

(0,{(b,0)})

(1,{(a,1),(b,1)})

(2,{(a,2),(b,2)})

(3,{(a,3),(b,3)})

(6,{(a,6)})

(8,{(a,8)})

(9,{(a,9)})

(11,{(a,11)})

2.14 UNION

A union operation appends the tuples of one relation to another relation. Suppose there are two start states S_a and S_b. The union on S_a and S_b appends all the tuples of S_b to S_a. Aunion operation also does not make any kind of changes to the timestamps; therefore, a temporal union operation is the same as a regular union operation.

```
X = UNION Sa, Sb;
```

As an example, when we carry out the above operation on S_a and S_b, we get the following output.

```
(a,1,6)
(a,2,8)
(b,0,3)
(a,9,11)
(a,1,15)
(b,5,9)
(c,0,4)
(a,4,8)
```

CHAPTER 3

COST OF TEMPORAL PIG LATIN OPERATIONS

In the previous chapter, we look at templates for various temporal operations. In this chapter, we compare the cost of temporal Pig Latin operations with the cost of regular Pig Latin operations.

3.1 Hardware Specifications

- Processor : Intel(R) Core(TM) i5 CPU M 450 @ 2.40GHz
- RAM : 6 GB

3.2 Operating System Specification

- Ubuntu 10.10

3.3 Software Specification

- JDK 1.6
- Pig 0.7.0
- Hadoop 0.20.2

We take three files of sizes 10 KB, 100 KB and 1000 KB for our experiment and test all the temporal operations with these files as input. The execution time taken for these operations are noted, and a graph is plotted to show the temporal operation execution time and the slowdown of the temporal operations as compared to the regular operations.

Figure 9 shows this data for all of the temporal operations except the temporal cross operation. Figure 10 shows this data for the temporal cross operation. The temporal cross operation was not tested on a 1000KB file, as the time taken would have been prohibitively large.

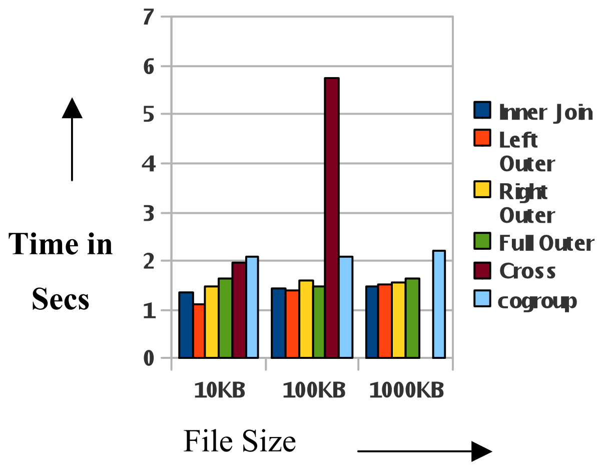


FIGURE 9. GRAPH OF FILE SIZE VERSUS EXECUTION TIME FOR TEMPORAL OPERATIONS

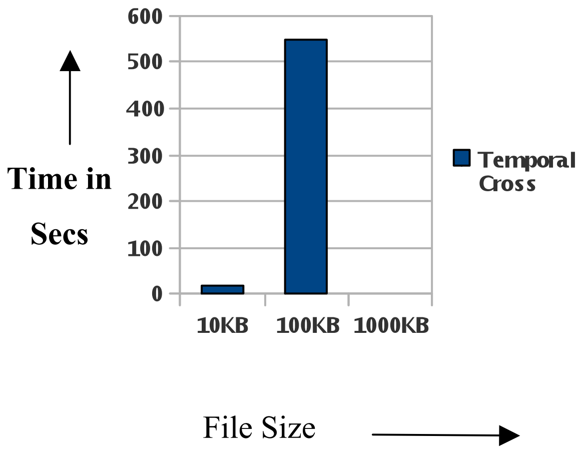


FIGURE 10. GRAPH OF FILE SIZE VS EXECUTION TIME FOR TEMPORAL CROSS OPERATION

Figure 11 shows a comparison of regular operations and temporal operations. Note the slowdown of the temporal operations compared to the regular operations.

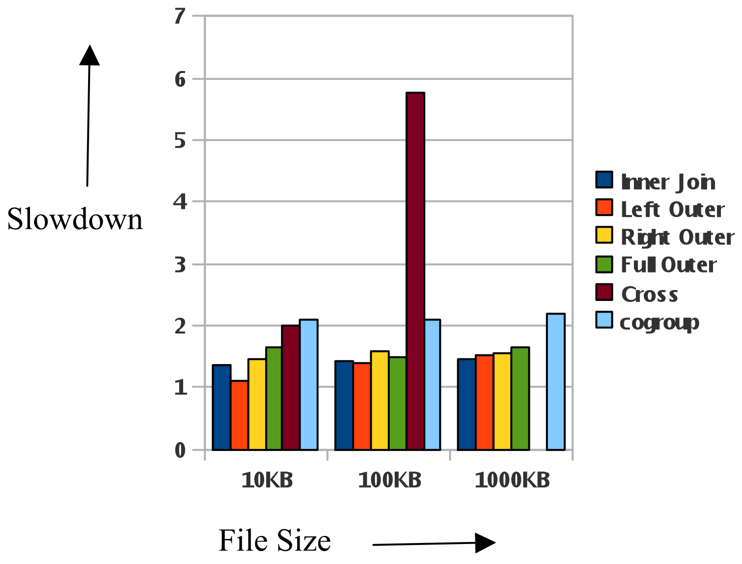


FIGURE 11. COMPLEXITY GRAPH

The operations are performed on three different file sizes, 10KB, 100KB, and 1000KB. Firstly, we perform regular operations, that is inner join, left outer join, right outer join, full outer join, cross join, and cogroup on the three different file sizes, with one exception. The execution time for the cross Join on 1000KB files are not performed, as the execution time would be prohibitively large. All execution times are noted.

We then perform temporal operations, that is temporal inner join, temporal left outer join, temporal right outer join, temporal full outer join, temporal cross join and temporal cogroup on the same files and note the execution time for the same.

Now that we have the execution time for both regular and temporal operations, for each operation we perform:

(Execution Time for Temporal Operation) / (Execution Time for Regular Operation). This gives us a scalar for determining which is greater the execution time of a temporal operation or the execution time of a regular operation.

3.4 Analysis

Note in Figure 11 that temporal operations take more time than regular operations, since we perform a series of operations on the regular version of the corresponding operation.

As the file size increases, we can note that complexity for cross join significantly increases, complexity for inner join, right outer join, full outer join, and cogroup almost all remain the same, and the complexity for left outer join increases a small amount.

Compared to the regular cogroup operation, the temporal cogroup operation execution takes almost twice the time as a regular operation on all file sizes.

CHAPTER 4

SUMMARY AND FUTURE WORK

Pig Latin is a dataflow language used primarily for analyzing very large data sets. It lends itself well to data transformations like filter and group. It allows storage and projection of the intermediate results, which can be very useful in achieving desired results.

Presently, Pig Latin does not support temporal operations. For this project, we created templates in order to support temporal operations in Pig Latin. These templates can be later added to Pig Latin source code and could be used as built in operations.

4.1 Future Work

There are lot of enhancements that could be done in this area. Firstly, the templates for outer joins are not complete as the results of outer join are only partial. Much research could be done in this area to achieve more complete results. Secondly, the temporal grouping does not scale well for large data sets. Hence, some improvements can be made in this direction, too.

Finally, sequenced grouping is another area for potential improvements.

REFERENCES

[1] Hadoop. Apache Pig. Available at http://pig.apache.org/.

[2] Pig Wiki. Apache Pig Wiki. Available at http://wiki.apache.org/pig/.

[3] Turing Complete Pig. Available at http://wiki.apache.org/pig/TuringCompletePig

APPENDICES

APPENDIX A

COALESCE

```
S = LOAD '../file_1mb.txt' using PigStorage('\t') as (name, st, et);

A = FOREACH S generate name, st,et;

B = GROUP A by name;

X = FOREACH B generate FLATTEN(myudfs.Coalesce(group,A)) as name;
```

COGROUP

```
A = COGROUP Sa by name, Sb by name1;

B = FILTER A by IsEmpty(Sa);

C = FILTER A by IsEmpty(Sb);

D = FILTER A by not IsEmpty(Sa) and not IsEmpty(Sb);

E = FOREACH B generate FLATTEN(Sb);

F = FOREACH C generate FLATTEN(Sa);

G = FOREACH D generate FLATTEN(Sa), FLATTEN(Sb);

H = FOREACH G generate name,(b>b1?b:b1) as b, (e<e1?e:e1) as e;

I = DISTINCT H;

J = FILTER I by b <= e;

X = UNION E, F, J;
```

CROSS

```
A = CROSS $S_a$,$S_b$;

B = FOREACH A generate name,name1,(b>b1?b:b1), (e<e1?e:e1);

C = FILTER B by b <= e;

X = DISTINCT C;
```

DISTINCT

```
A = DISTINCT $S_a$;
```

FILTER

```
A = FILTER $S_a$ by name is not null;
```

FOREACH

```
A = FOREACH $S_a$ generate name,b,e;
```

FULL OUTER JOIN

```
A = JOIN S_a by name FULL OUTER, S_b by name1;

B = FOREACH A generate ((b is null) ? name1 : name) as name,((b is null) ? b1 : ((b1 is not null) ? ((b > b1) ? b : b1) : b)) as b , ((b is null) ? e1 : ((b1 is not null) ? ((e < e1) ? e : e1) : e)) as e;

C = DISTINCT B;

X = FILTER C by b <= e;
```

INNER JOIN

```
A = JOIN S_a by name , S_b by name1;

B = FOREACH A generate name,name1,(b>b1?b:b1), (e<e1?e:e1);

C = filter B by b <= e;

X = DISTINCT C;
```

LEFT OUTER JOIN

```
A = JOIN S_a by name LEFT OUTER, S_b by name1;

B = FOREACH A generate name,((b1 is null) ? b : ((b > b1) ? b : b1)) as b,((b1 is null) ? e :
((e < e1) ? e : e1)) as e;

C = DISTINCT B;

X = FILTER C by b <= e;
```

RIGHT OUTER JOIN

```
A = JOIN Sa by name RIGHT OUTER, Sb by name1;

B = FOREACH A generate ((b is null) ? name1 : name) as name,((b is null) ? b1 : ((b >
b1) ? b : b1)) as b,((b is null) ? e1 : ((e < e1) ? e : e1)) as e;

C = DISTINCT B;

X = FILTER C by b <= e;
```

SAMPLE

```
A = SAMPLE Sa 0.5;
```

SPLIT

```
SPLIT Sa INTO X IF b<7, Y IF e==5, Z IF (b<6 OR b>6);
```

TEMPORAL GROUP

```
A = FOREACH Sa generate b as b1;

B = FOREACH Sa generate e as e1;

C = UNION A,B;

D = CROSS Sa , C;

E = FILTER D by b1 <= e and b1 >= b;

F = FOREACH E generate name, b1;

G = DISTINCT F;
```

X = GROUP G by b1;

UNION

X = UNION S_a, S_b;

APPENDIX B

Coalesce . java

```
package myudfs;



import java.io.*;

import java.util.*;

import org.apache.pig.EvalFunc;

import org.apache.pig.backend.executionengine.ExecException;

import org.apache.pig.data.Tuple;

import org.apache.pig.data.TupleFactory;

import org.apache.pig.impl.logicalLayer.schema.Schema;

import org.apache.pig.data.DataType;

import org.apache.pig.data.BagFactory;

import org.apache.pig.data.DataBag;



public class Coalesce extends EvalFunc<Tuple> {

        ArrayList<Integer> lower = new ArrayList<Integer>();

        ArrayList<Integer> higher = new ArrayList<Integer>();

        ArrayList<Tuple> result = new ArrayList<Tuple>();
        /*
         * First argument is key by which the data is grouped. In this case it is name
         * Second argument consist of a databag which has grouped tuples essentially array of (name,
start time, end time)
         */
        public Tuple exec(Tuple input) throws IOException {
                if (input == null || input.size() < 2) { // if input from pig doesnt have
                   //two arguments that is group key and databag then it should return null
                        return null;
                }
                try {
                        Tuple output = TupleFactory.getInstance().newTuple(3);
                        DataBag db = (DataBag) input.get(1); //gets the databag of a group which needs
to be processed
                        Iterator it = db.iterator();
                        while (it.hasNext()) {
                                Tuple t = (Tuple) it.next();
                                lower.add(Integer.parseInt(t.get(1).toString()));    //adding    the    lower
timestamp to lower array list
```

```
                higher.add(Integer.parseInt(t.get(2).toString()));    //    adding    higher
timestamp to higher array list
            }
            //sort according to lower
            sort();
            //make the groups and store it in results
            group();
            //iterate through results and add into db
            DataBag db1 = BagFactory.getInstance().newDefaultBag();
            Iterator it1 = result.iterator();
            while (it1.hasNext()) {
                Tuple t = (Tuple) it1.next();
                db1.add(t);
            }
            //clear lower and higher and result
            lower.clear();
            higher.clear();
            result.clear();
            output.set(0, input.get(0));
            output.set(1, db1);
```

```
                        return output;

                } catch (Exception e) {

                        System.err.println("Failed to process input; error - " + e.getMessage());

                        return null;

                }

        }

        public Schema outputSchema(Schema input) {

                try {

                        Schema tupleschema = new Schema();

                        tupleschema.add(new
Schema.FieldSchema("group",DataType.CHARARRAY));

                        ArrayList<Schema.FieldSchema>          inner          =          new
ArrayList<Schema.FieldSchema>();

                        inner.add(new Schema.FieldSchema("st", DataType.INTEGER));

                        inner.add(new Schema.FieldSchema("et", DataType.INTEGER));

                        Schema bagSchema = new Schema(inner);

                        bagSchema.setTwoLevelAccessRequired(true);

                        Schema.FieldSchema          bagFs          =          new
Schema.FieldSchema("timestamps",bagSchema, DataType.BAG);

                        tupleschema.add(bagFs);
                        return new Schema( new Schema.FieldSchema ( getSchemaName(  this.getClass
() . getName () . toLowerCase () , input ), tupleschema, DataType.BAG));
                } catch (Exception e) {
                        return null;
                }
        }
        public void sort() {
                for (int i = 1; i <= lower.size(); i++) {
                        for (int j = 0; j < lower.size() - 1; j++) {
                                if (lower.get(j + 1) < lower.get(j)) {
                                        int temp = lower.get(j);
                                        lower.set(j, lower.get(j + 1));
                                        lower.set(j + 1, temp);
                                        int temp1 = higher.get(j);
                                        higher.set(j, higher.get(j + 1));
                                        higher.set(j + 1, temp1);
                                }
                        }
                }
        }
```

```
//This function is responsible for grouping the tuples inside the databag and puts the result in
results array list

public void group() throws ExecException {

    int current_lower = lower.get(0), current_higher = higher.get(0);

    for (int i = 1; i < lower.size(); i++) {

        if (current_higher >= lower.get(i)) {

            if (current_higher < higher.get(i)) {

                current_higher = higher.get(i);

            }

        } else {

            Tuple tobepushed = TupleFactory.getInstance().newTuple(2);

            tobepushed.set(0, current_lower);

            tobepushed.set(1, current_higher);

            result.add(tobepushed);

            current_lower = lower.get(i);

            current_higher = higher.get(i);

        }

    }

    Tuple tobepushed = TupleFactory.getInstance().newTuple(2);

    tobepushed.set(0, current_lower);
```

```
                tobepushed.set(1, current_higher);

                result.add(tobepushed);

        }

}
```