DB Sys LAB3
Project Report

Julien THOMAS,
Eliot Calimez

Contents

1 Presentation

This project is used as part of the DBSys course, of the Data Science branch of EURECOM

The objective of this lab is to implement join methods, such as IEJoin or NLJ.
Class implementations of iterators will be in the iterators package, and tests in the 'tests/Lab3Test.java' class. We will analyze the execution time of each method in order to evaluate its evolution according to the number of data, and thus to check which method would be the most asymptotically efficient. All the source code are avaiable on git here : `https://github.com/korrigans84/DBsys_lab3`

2 preliminary

2.1 Input format

The input queries are written in a txt file, and follow this format:

Single Predicate Query:
LINE 1: Rel1_col Rel2_col
LINE 2: Rel1 Rel2
LINE 3: Rel1_col op 1 Rel2_col

Two Predicates Query:
Rel1_col Rel2_col
Rel1 Rel2
Rel1_col op1 Rel2_col
AND/OR
Rel1_col op2 Rel2_col

We therefore need to retrieve the data from this file in order to run the query. The 'QueryFromFile.java' class therefore makes it possible to retrieve the data.

The data is stored in a text file, and we will use the File2Heap method which allows to recover this data.

2.2 Output format

For each query, we have to retrieve the time execution and the size of tables in input.
In the class Lab3Test.java, we have implemented a method which allows to retrieve this data and add them to a csv to be able to analyze the methods in a second step.

We will also retrieve the data from the query, using a projection of the columns selected in the query. This data is written to a text file.

3 Part 1 : Nested Loops Joins

The method is already implemented, so we have to test it. For that, we have recreated a query similar to query 2 in order to test its inequalities operations.

We have to notice that this method compute the next tuple only when we call the get_next method. This avoids all of the storage problems that we will encounter later, because only the input data is stored in memory.

3.1 Task 1a : Test/extend the existing NLJ with single predicate inequality joins.

This task was implemented in 'private void Query1a(String queryFile)' method of Lab3Test.java class.

The problem here was knowing how to implement the operator introduced in the query. Everything therefore goes into the CondExpr_Query1a method, which implements the operator in outFilter, as well as the columns to compare :

```
    private void CondExpr_Query1a(CondExpr[] outFilter, int operator, int outerSymbol, int innerSymbol) {
        outFilter[0].next  = null;
        outFilter[0].op    = new AttrOperator(operator);
        outFilter[0].type1 = new AttrType(AttrType.attrSymbol);
        outFilter[0].type2 = new AttrType(AttrType.attrSymbol);
        outFilter[0].operand1.symbol = new FldSpec(new RelSpec(RelSpec.outer),outerSymbol);
        outFilter[0].operand2.symbol = new FldSpec (new RelSpec(RelSpec.innerRel),innerSymbol);
}
```

The arguments come from 'QueryFromFile.java' class.

3.2 Task 1b : Test/extend NLJ with two predicates inequality join

Just like the simple predicate inequality join, we repeat the test of Query2 by modifying the outFilter:

```
private void CondExpr_Query1b(CondExpr[] outFilter, int operator, int outerSymbol, int innerSymbol, int opera

    outFilter[0].next  = null;
    outFilter[0].op    = new AttrOperator(operator);
    outFilter[0].type1 = new AttrType(AttrType.attrSymbol);
    outFilter[0].type2 = new AttrType(AttrType.attrSymbol);
    outFilter[0].operand1.symbol = new FldSpec(new RelSpec(RelSpec.outer),outerSymbol);
    outFilter[0].operand2.symbol = new FldSpec (new RelSpec(RelSpec.innerRel),innerSymbol);

    outFilter[1].next  = null;
    outFilter[1].op    = new AttrOperator(operator2);
    outFilter[1].type1 = new AttrType(AttrType.attrSymbol);
    outFilter[1].type2 = new AttrType(AttrType.attrSymbol);
    outFilter[1].operand1.symbol = new FldSpec(new RelSpec(RelSpec.outer),outer2Symbol);
    outFilter[1].operand2.symbol = new FldSpec (new RelSpec(RelSpec.innerRel),inner2Symbol);

    outFilter[2] = null;
}
```

The arguments still come from 'QueryFromFile.java' class.

4 Part 2 : IEJoin implementation

For the implementation of IEJoin, we will therefore have to implement it. The algorithm is in the "Fast and Scalable Inequality Joins" paper. We will therefore first work on a single table for the task 2a, and 2b, and we will implement another algorithm to generalize this one to 2 different table

Algorithm 2: IESelfJoin

```
input  : query $Q$ with 2 join predicates $t_1.X$ $\mathsf{op}_1$ $t_2.X$
         and $t_1.Y$ $\mathsf{op}_2$ $t_2.Y$, table $T$ of size $n$
output: a list of tuple pairs $(t_i, t_j)$
let $L_1$ (resp. $L_2$) be the array of column $X$ (resp. $Y$)
if ($\mathsf{op}_1 \in \{>, \geq\}$) sort $L_1$ in ascending order
else if ($\mathsf{op}_1 \in \{<, \leq\}$) sort $L_1$ in descending order
if ($\mathsf{op}_2 \in \{>, \geq\}$) sort $L_2$ in descending order
else if ($\mathsf{op}_2 \in \{<, \leq\}$) sort $L_2$ in ascending order
compute the permutation array $P$ of $L_2$ w.r.t. $L_1$
initialize bit-array $B$ ($|B| = n$), and set all bits to 0
initialize join_result as an empty list for tuple pairs
if ($\mathsf{op}_1 \in \{\leq, \geq\}$ and $\mathsf{op}_2 \in \{\leq, \geq\}$) eqOff $= 0$
else eqOff $= 1$
for ($i \leftarrow 1$ to $n$) do
    pos $\leftarrow P[i]$
    $B[\mathsf{pos}] \leftarrow 1$
    for ($j \leftarrow$ pos + eqOff to $n$) do
        if $B[j] = 1$ then
            add tuples w.r.t. $(L_1[j], L_1[P[i]])$ to
            join_result
return join_result
```

It is important to notify that the IEJoin method must sort the data, and must be traversed several times in the algorithm but the iterators can be traversed only once. In order to simplify this problem, we have therefore decided to use ArrayLists, which will cause memory problems later.

4.1 Task 2a : Implement single predicate self join operation.

Single predicate self join operation is implemented in the SelfJoinPredicate.java class.
Unlike the algorithm, here we don't need to work with 2 operators, but only 1. Thus, it suffices to sort the tuples and for each, retrieve the tuples before or after in the same list, depending on the operator.
We therefore pass N times through the Iterator, which is impossible. So for that we used arraysList

The for loop looks like this:

```
for(int i=0; i<N; i++) {
    for(int j= i+eqOff; j<N; j++) {
        Projection.Join(L1_array.get(j), in1,
         L1_array.get(i), in1,
         JTuple, proj_list, n_out_flds);
        result.add(new Tuple(JTuple));
    }
 }
```

You can find the result of an execution of it algorithm in the output folder.
The use of arrayLists strongly limits us in the execution of queries, which did not allow us to do large-scale tests.

4.2 Task 2b : Extend the single predicate inequality self join (Task 2a) to two predicates in the condition of the inequality self join.

This time, we will implement the algorithm as is, in the SelfJoin.java class, but we will still use ArrayLists.

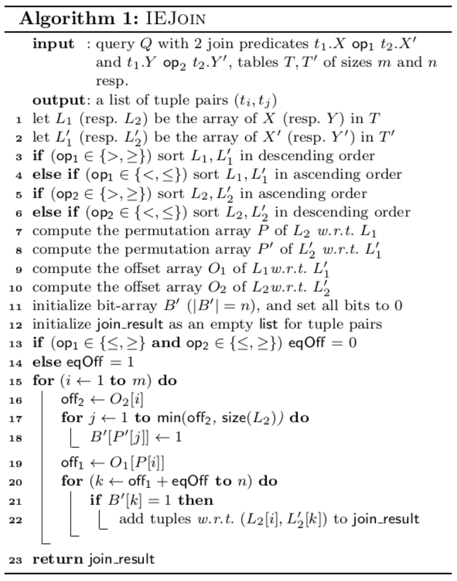
Some improvements have been made. To compute the permutation array, there is a simpler and less expensive method possible. Knowing that we perform a self join, the permutation array will be either the identity, or the inverted identity, depending on the sorting of the 2 lists of tuples. So there is no need to compare every tuples here. We've detect it too late to implement it.

```
for(int i=0; i<N; i++) {
    for(int j=0; j<N; j++) {
        if(TupleUtils.Equal(L1_array.get(i), L2_array.get(j), in1, len_in1)) {
            P[i] = j;
        }
    }
}
//but more simply, we can do

int[] P = new int[N];
if(order.tupleOrder == order2.tupleOrder) {
    for(int i = 0; i<N; i++)
     P[i]=i;
}else {
    for(int i = 0; i<N; i++)
     P[i]=N-i-1;
}
```

4.3 Task 2c : Implement two predicates inequality join.

For the task 2c, we will extend the IEJoin algorithm to join 2 different table, using another algorithm



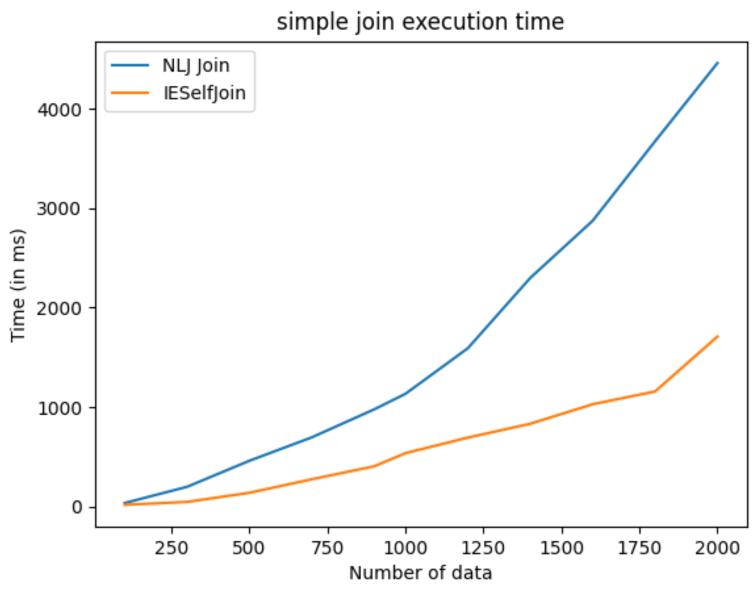
```
Algorithm 1: IEJoin
input  : query $Q$ with 2 join predicates $t_1.X$ op$_1$ $t_2.X'$
         and $t_1.Y$ op$_2$ $t_2.Y'$, tables $T, T'$ of sizes $m$ and $n$
         resp.
output: a list of tuple pairs $(t_i, t_j)$
let $L_1$ (resp. $L_2$) be the array of $X$ (resp. $Y$) in $T$
let $L'_1$ (resp. $L'_2$) be the array of $X'$ (resp. $Y'$) in $T'$
if (op$_1 \in \{>, \geq\}$) sort $L_1, L'_1$ in descending order
else if (op$_1 \in \{<, \leq\}$) sort $L_1, L'_1$ in ascending order
if (op$_2 \in \{>, \geq\}$) sort $L_2, L'_2$ in ascending order
else if (op$_2 \in \{<, \leq\}$) sort $L_2, L'_2$ in descending order
compute the permutation array $P$ of $L_2$ $w.r.t.$ $L_1$
compute the permutation array $P'$ of $L'_2$ $w.r.t.$ $L'_1$
compute the offset array $O_1$ of $L_1 w.r.t.$ $L'_1$
compute the offset array $O_2$ of $L_2 w.r.t.$ $L'_2$
initialize bit-array $B'$ ($|B'| = n$), and set all bits to 0
initialize join_result as an empty list for tuple pairs
if (op$_1 \in \{\leq, \geq\}$ and op$_2 \in \{\leq, \geq\}$) eqOff $= 0$
else eqOff $= 1$
for ($i \leftarrow 1$ to $m$) do
    off$_2 \leftarrow O_2[i]$
    for $j \leftarrow 1$ to min(off$_2$, size($L_2$)) do
        $B'[P'[j]] \leftarrow 1$
    off$_1 \leftarrow O_1[P[i]]$
    for ($k \leftarrow$ off$_1$ + eqOff to $n$) do
        if $B'[k] = 1$ then
            add tuples $w.r.t.$ ($L_2[i]$, $L'_2[k]$) to join_result
return join_result
```

This method gave us several problems, and only works for certain queries, like the one included in the Output folder.
We need to have 4 Iterators in input, to be able to browse the tables 2 times each, if we have to sort it in a different order for each comparison. This strategy should be able to be improved, as it loads each table twice into memory.

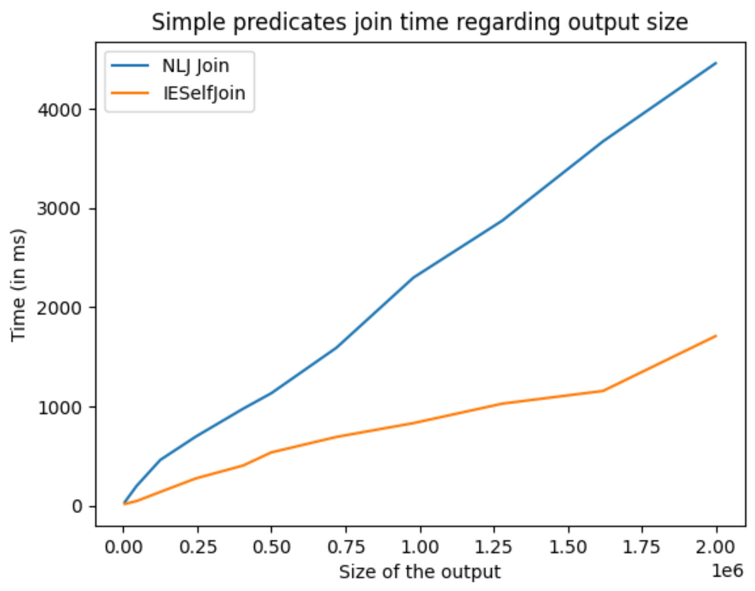
5 Execution analysis

As we said before, using ArrayLists limits the amount of data possible in the query. However, the curves observed make it possible to identify trends.

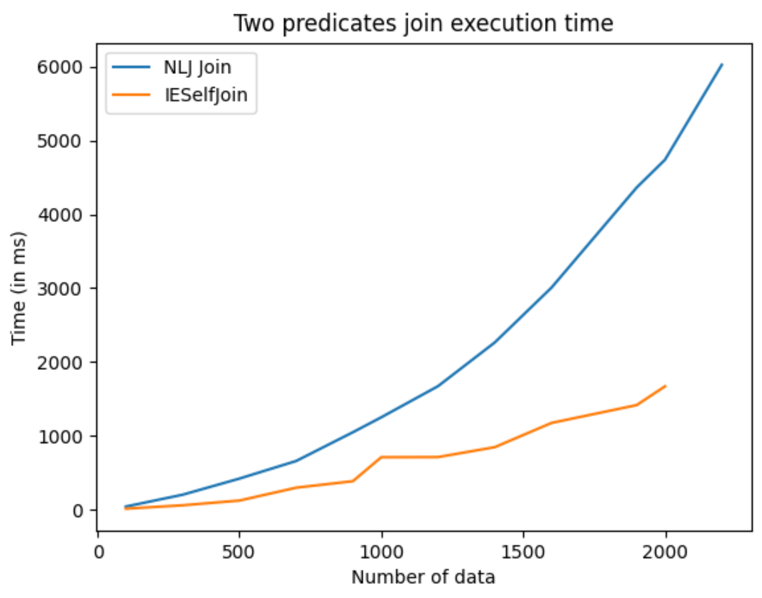
5.1 Simple predicate execution

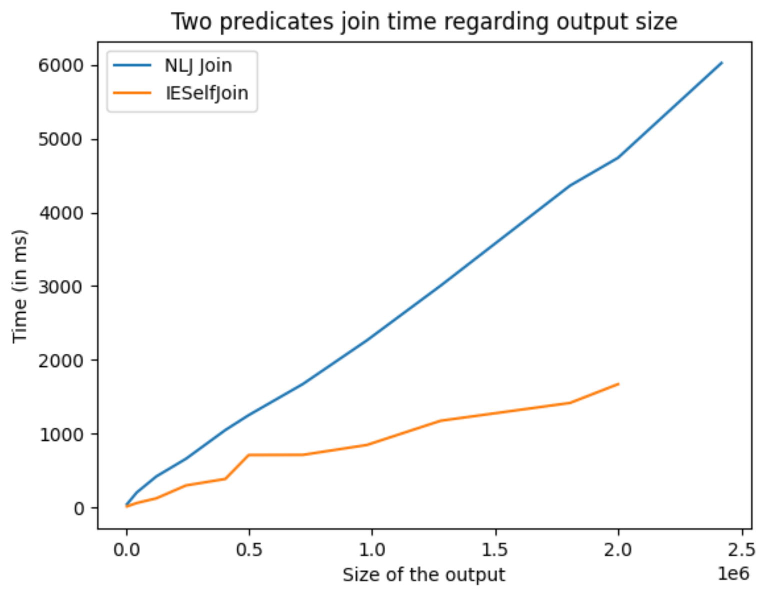


The IEJoin method evolves much less quickly than the NLJ method, which justifies its use.



5.2 Two predicates execution





For the execution with 2 comparisons, the use of IEJoin seems even more justified, because the evolution of the execution time seems linear with the size of the output data, while the NLJ method has an exponential evolution, which prevents scalability.

6 Conclusion

This lab therefore allowed us to play with the implementation of the method of execution of queries in Java. After hours of debugging, and after spending hours understanding the code, we have successfully implemented 2 methods of IESelfJoin. The IEJoin method has been coded, but is still buggy, despite that it works for some operators.
Our methods are very perfectible, because the use of ArrayList strongly prevents their scalability.