

EURECOM

FINAL REPORT

IEJoin Assignment

DBSYS PROJECT
Database Management System Implementation
DBSys

Student Authors
Irina MOSCHINI
Thiziri NAIT SAADA

Teacher
Paolo PAPOTTI

Sommaire

1	Introduction	2
2	The code	2
2.1	The queries	2
2.1.1	Query1a	2
2.1.2	Query1b	3
2.1.3	Query2a	4
2.1.4	Query2b	4
2.1.5	Query2c	4
2.2	Automatization	5
3	Limitations	6
3.0.1	Query1a	6
3.0.2	Query1b	6
3.0.3	Query2a	6
3.0.4	Query2b	6
3.0.5	Query2c	6
4	Scalability	6
5	Correctness	8
5.1	Comparison of NLJ and Self Join with one predicate	8
5.2	Comparison of NLJ and Self Join with two predicates	8
6	README	13

1. Introduction

Optimizing joins for the computing of queries is a common problem that a lot of researchers focused on. Inequality joins are even slower, so techniques have to be found to run such queries in a reasonable time. Some algorithms such as sort merge joins or interval-based indexing exist but can be very slow, even more for queries with inequality joins (it has a quadratic complexity). In this project, we have used/implemented two algorithms, Nested Loop Join (already implemented), and IEJoin (we have implemented) that are more efficient for such queries. They can be used for single predicate inequality joins or two predicates inequality join (with AND or OR conjunctions). Here is a table of the algorithm used, the number of predicate used and the name of the query associated.

	query_1a	query_1b	query_2a	query_2b	query_2c
Algorithm used	Nested Loop Join	Nested Loop Join	IE Self Join	IE Self Join	IEJoin
Number of predicate(s)	1	2	1	2	2

We have written the test to use nested-loop join (NLJ) operation to join two tuples for the given inequality condition (questions **1a** 2.1.1 and **1b** 2.1.2). We then implemented Self Join algorithm for a single predicate (question **2a** 2.1.3), Self Join algorithm for two predicates (question **2b** 2.1.4) and IEJoin (question **2c** 2.1.5). All the tests are run in the class `JoinTest2`, that has the similar use of the `JoinTest` class given at the beginning. The algorithms we implemented are in the classes `SelfJoin1` for the single predicate using Self Join, `SelfJoin2` for the two predicates using Self Join, and `IEJoin` for two predicates using two different tables. We also made an automatisation 2.2 to be able to read any text file containing a query (just need to respect the structure). All will be explained in the second part of our report.

For the algorithms we have implemented (`SelfJoin1`, `SelfJoin2` and `IEJoin`), we noticed some issues, especially for tables containing duplications. We tried to face them and think we succeeded for `SelfJoin1`, but not for `IEJoin` (we think the pseudo-code has to be changed a bit to support tables with duplicate values). We will give the limitations of the algorithms in the third part of the report, section 3.

To see if our central goal (increase the speed of computing the queries) has been achieved, we will compute plots that show the execution time for the same query over the same database of increasing size of input tuples. The plots and analyses will be given in the section 4.

As some algorithms can be used for same queries (Nested Loop Join vs. IEJoin), we will finally compare the results obtained by both the algorithms we used, that can be found in the section 5.

2. The code

2.1. The queries

2.1.1. Query1a

For this first query using Nested Loop Join for a single predicate, what we had to do was understanding the code of `JoinTest` and the examples given (*Query1* to *Query6*). We decided to be inspired by the *Query2* that uses, as we need, Nested Loop Join and used the same structure for our implementation (that is to say create the three functions *RunTest2*, *Query1a* and *Query1a_CondExpr*).

The major thing we had to change was how to access to the database. While in `JoinTest` the table was defined at the beginning of the class, ours were given in text files. This is why we used `Scanner` object that enabled us to read our tables and putted them in a `Vector` object. This method has been used thereafter to read all our text files.

We then understood the function `Query2_CondExpr` (and adapted it for our `Query1a_CondExpr`) that defines the "where" condition by giving the operator to use thanks to this line (where `op` is the number corresponding to the inequality) :

```
expr[0].op = new AttrOperator(op);
```

It also gives the columns to compare thanks to these lines (where `RelSpec.outer` and `RelSpec.innerRel` define the outer and inner relation, and `cond_rel1` and `cond_rel2` give the number of the columns to compare) :

```
expr[0].operand1.symbol = new FldSpec (new RelSpec(RelSpec.outer),cond_rel1);

expr[0].operand2.symbol = new FldSpec (new RelSpec(RelSpec.innerRel),cond_rel2);
```

Then, the understanding of `Query2_CondExpr` was essential to project correctly the queries. The `outFilter` allows to create the adapted size for the condition (always the same for single queries). We then gave the size of the different tables we used (`Rtypes` for the first table for example, that is composed of for columns of integers). Finally, the projection were made by these lines :

```
FldSpec [] projCol1 = {
new FldSpec(new RelSpec(RelSpec.outer), output_rel1),
new FldSpec(new RelSpec(RelSpec.innerRel), output_rel2),
};
```

`RelSpec.outer` gives the relation to consider (inner or outer relation), and `output_rel1` and `output_rel2` gives the columns to keep after the projection.

Once everything is initialized, we just have to call the Nested Loop Join algorithm thanks to this line :

```
nlj = new NestedLoopsJoins (Rtypes, 4, Rsizes,
Stypes, 4, Ssizes,
10,
am, "table2.in",
outFilter, null, projCol1, 2);
```

where the first line give the characteristics of the outer relation (number of columns and types), the second line gives the characteristics of the second relation, the fourth line gives the two table to take into account (`am` is the outer relation that has been previously scanned and `"table2.in"` is a String giving the name of the second relation.

The results are then printed in the console.

2.1.2. Query1b

What we had to change here comparing to the `Query_1a` was how to manage two predicate. We initialized all the rest (types, sizes, projections, how to call Nested Loop Join) exactly as `Query_1a`.

The main thing that has to be changed for two predicates is the outfilter and thus the function `Query1b_CondExpr`. We understood that the outfilter did not have the same dimensions whether an AND or an OR operator is called between the two predicates.

- For an AND, the size of the outfilter 3 (we have just a list). We thus give, for the first position, the first operator and the firsts columns to compare, for the second position we give the second operator and the seconds columns to compare. We finally put a `null` pointer to announce the end of the list for the third position. Please find the code in the function `Query1b_CondExpr` of the class `JoinTest2`.

- For an OR conjunction, we need a linked list for the first position and a `null` pointer for the second position that announces the end of the list. The linked list first gives, for the first position, the first operation and the columns to compare, and then, for the second position, the second operator and the second columns to compare. Please find the code in the function `Query1b_CondExpr` of the class `JoinTest2`.

The last thing to do is then adapt the size of the outfilter in the function *Query1b* depending on the conjunction the query gives.

All the rest is the same as the *Query_1a*.

2.1.3. *Query2a*

In this case, only one condition on one table needs to be satisfied. This case is the simplest one. Instead of using the whole pseudo-code given for two predicates (*Query 2b*), we used the idea given in the paper to just sort a list of tuples, according to the values of the attribute concerned by the condition (the order is given according to the operator that is entered in the query). Then, the trick finally consists in taking a part of this list (the beginning or the end, depending on which order the list has been ordered in). All the tuples taken concerned by this operation corresponds the tuples that satisfy the constraint given by the query. To sort the list of tuples, we did not used the function used by Professor Papotti previously in his *NestedLoopJoin.java* function, that is to say : **Sort**. Instead, we implemented our own customized comparator to do so.

In order to perform well on duplicates in the table, we added some conditions to make sure that they are taken into account. More precisely, when, in the loop, a tuple (*tuple_1*) is met in the sorted list, we systematically check if the next tuples have the same attribute value. If so, we add the couple (*tuple_1*, *tuple_2*) **and** the couple (*tuple_2*, *tuple_1*) to the results, such that we can keep going through the sorted list in the same order as we were, but without forgetting this result. This way, our algorithm is able to find the exact solutions to the problem. We tested it on both non-duplicates and duplicates tables. In both cases, the output results were correct. Please find attached to this report some examples of outputs.

We finally projected onto the attributes specified in the query to get the final result. Please find attached all the code in the file *SelfJoin1.java*.

2.1.4. *Query2b*

In this case, two conditions on the same table need to be satisfied, linked with an **AND**. To deal with it, we used two *ArrayList*, each of both concerning a constrained attributes (given from the query). The specificity of our algorithm is that it performs well on duplicates values :

- We kept track of the records while sorting the two lists. Indeed, rather than letting *list1* and *list2* be having *ArrayLists*, we used instead two *ArrayLists* of *ArrayLists*. Then, for example, if the first condition concerns the *col_1* attribute, *list1* would be an *ArrayList* containing all the *col_1* values, with the *col_2* value associated in the same tuple, and the *rid* of this very tuple from which the attributes were extracted of. We assigned those *rid* in an arbitrary order given by the way the tuples were inserted in the table.
- We computed a permutation array as mentioned in the paper.
- Then, we computed and used our own sorting functions, called **comparator** to sort these *ArrayLists* of *ArrayLists*, according to the attributes values concerned by the operation.
- To deal with duplicates, we read carefully the **Section 3.2** of the paper and given the two operators, we sorted the two *ArrayLists* according to the table from the paper. To do so, our own sorting functions were really useful because we could custom the way the second attribute would be sorted in case of duplicates values for the first attribute. For example, when the first operator (that is to say the operator in the first predicate) was a **<** and the second operator was a **>**, we used our costumed sorting function using **comparator_desc_desc** to sort *list1* in descending order for the first attribute and descending order in equality cases. In this same example, we used the comparator using **comparator_asc_desc** to sort *list2* in ascending order for the first attribute and descending order in equality cases, as mentioned in the paper. In the specific case where both first and second attributes are equal, this is the *rid* of the tuples that gives the order. Therefore, we succeeded in treating all the duplicates cases.
- We finally projected onto the attributes specified in the query to get the final result.

We tested our algorithm on both non-duplicates and duplicates tables. In both cases, the output results were correct. Please find attached to this report some examples of outputs.

Please find attached all the code in the file *SelfJoin2.java*.

2.1.5. *Query2c*

Likewise previously, in this case, two conditions need to be satisfied, linked with an **AND**. However, they concern two different tables. We implemented this function according to the pseudo code given in the paper. As before, we used *ArrayList* of *ArrayLists* (containing the first attribute values, the second, and the *rid* associated to the tuple).

The main changes were to add two offset arrays, and two extra permutation and bit arrays. Indeed, as they concern two different tables, the bit and permutation arrays are not equal anymore from a table to another.

We tested the algorithm and it successfully returns the outputs that were expected only a few times.

To deal with duplicates, we tried to follow the exact same process as for the self join with two predicated by using the costumed comparators and referring to the table given in the paper to know which of them to use depending on the two operators of the query.

But it appears that the algorithm failed to provide the exact solutions we expected. We will discuss on it later on, in the limits sections.

2.2. Automatization

We decided to automate the read and the computation of the queries given in input. This automatisation has been done in the `main` and `RunTest2` functions of the class `JoinTest2`. What we made is that the user just have to enter in the console the number of tables he needs, the path to the tables and the path to the text file containing his query, and the computation is made automatically. Here is a table resuming the number of tables needed and the number of predicate for each query.

	query_1a	query_1b	query_2a	query_2b	query_2c
Number of table(s)	2	2	1	1	2
Number of predicate(s)	1	2	1	2	2

We can notice that the only query that take the same number of tables and the same number of predicates are the `query_1b` and `query_2c`. Thus, by having only the number of tables the user needs and the text file (thus the number of predicates), we can determine if a unique query can be called or if two queries can be called (just in case the number of table is 2 and there are 2 predicates). In this second case, we would demand to the user if he want to run his query thanks to Nested Loop Join (`query_1b`) or IEJoin (`query_2c`).

When the user enters the path go the text file containing his query, we have access to the text inside the file thanks to the object `BufferedReader`. We first computed the number of lines contained in this file to know if it is a simple predicate query, or a double predicate query (3 lines for a simple predicate, 5 lines for double predicates). Here is a table resuming, for each query we have implemented, the number of lines (and thus the number of predicates in the query) that contains the text file taken as input.

	query_1a	query_1b	query_2a	query_2b	query_2c
Number of lines	3	5	3	5	5

Then, in the function `RunTest2`, we did a separating cases whether there are 3 or 5 lines.

- If there are 3 lines, we have a single predicate. Assuming that the outer relation is always the one given on the left (for each line), what we need to know then is on what columns the final projection (the output) has to be made (to be read on line 1), the number of table used (to be read on line 2) and the columns to compare and the operator given (to be read on line 3). We thus created variables for each those information that will be printed in the console. As it is here a single predicate query, the only algorithms that can be called are the ones of `query_1a` and `query_2a`. But as `query_1a` needs 2 tables, while `query_2a` needs only 1 (IE Self Join), we don't have the choice of the algorithm to use. Therefore, we run the only algorithm that corresponds to the query and table(s) given.

- If there are 5 lines, we processed exactly the same as before but by created new variables for the conjunction, the second operator and the columns to compare for the second projection. 5 lines mean two predicates so `query_1b`, `query_2b` and `query_2c` can be called. If the user needs only one table, the only algorithm that can be called is the one of `query_2b`. If he needs to tables, `query_1b` and `query_2c` can be called so we directly asked the user which one he wants.

3. Limitations

3.0.1. Query1a

Everything works well for the `query_1a`. It supports the queries with single predicates that takes two tables as input.

3.0.2. Query1b

Everything works well for the `query_1b`. It supports the queries with double predicates that takes two tables as input. You can use it whether you use an OR or an AND conjunction between the two predicates.

3.0.3. Query2a

We have nothing to say for `query_2a`, everything should work well.

3.0.4. Query2b

As we thought we successfully dealt with duplicates, we tried to go further and to implement the exact same algorithm but that could give the exact output from a query with two predicates linked with an OR. It appears that the pseudo code is not adapted to do so.

3.0.5. Query2c

This algorithm deals with two predicates linked with an AND and not with an OR.

The problem we faced with the correctness of the algorithm while testing it seems to come from the computation of the offset arrays. Indeed, we realized that while computing them, if some elements from `list1` could be inserted in the same exact position in `list1prime`, the algorithm failed to return the exact outputs that were expected. Let `list1` be for example (considering only its first element in each `ArrayList`) : `[1 4 5]` and `list1prime` be `[2 7]`. The offset array1, in the exact sense given in the paper would be `[0 2 2]`, which appears to be an issue in the following of the computation (it exists an iteration where the second bit array is set to 1, and again, there is another iteration where the exact same element of the bit array is set to 1 one more time, ignoring the previous tuple while scanning the rest of the bit array. It is like this duplication in the bit array overwrites some previous steps in the algorithm. This explains the correctness of the algorithm, that will be discussed on in the following section.

Therefore, an idea for further improvement would be to succeed in dealing with these critical cases where multiple bit array elements are set to the same exact value.

4. Scalability

To evaluate the performances of our algorithms, we tested each of them on a table of increasing size (by increasing the number of tuples). To do so, in the `main` function of the `JoinTest2` class, we created an empty text file named `table_test.txt` and add one new tuple composed of four integers for each iteration thanks to the objects `FileWriter`, `BufferedWriter` and the code `bw.write(content);`.

Then, for each algorithm, we first looked for the breaking point, that is to say the first number of tuples for which the algorithm fails (or is too long to be executed). We then computed the time the algorithm took to run the query from 0 tuple until the breaking point (we called the algorithm every 100 or 500 tuples added, depending on the breaking point, to don't wait for too long). Here are the plots obtained. The red stars represent the breaking points.

You can see that for the `query_2a`, we have plotted several lines. Actually, we noticed that the execution time was very slow comparing to the other algorithms. It has an exponential growth as the plot shows. Thus, we thought about increasing the speed of execution by modifying the code in `SelfJoin1`. First, we thought that the comparison between tuples (to deal with duplicate values) was taking a lot of time, so decided to delete these lines (and don't deal with duplicate values anymore) to see if it was faster. Surprisingly enough, it was slower than before. Then, we thought that the `Sort` function was maybe slow as it has to sort all the tuples (and not only the columns to keep), and used another method of sorting, the one we have implemented in `SelfJoin2`. The execution time is a better than before, so we decided to keep this version of the code. Another important thing to notice is that we did not find the breaking point, and decided to stop running the code for 2100 tuples because it was very slow (around

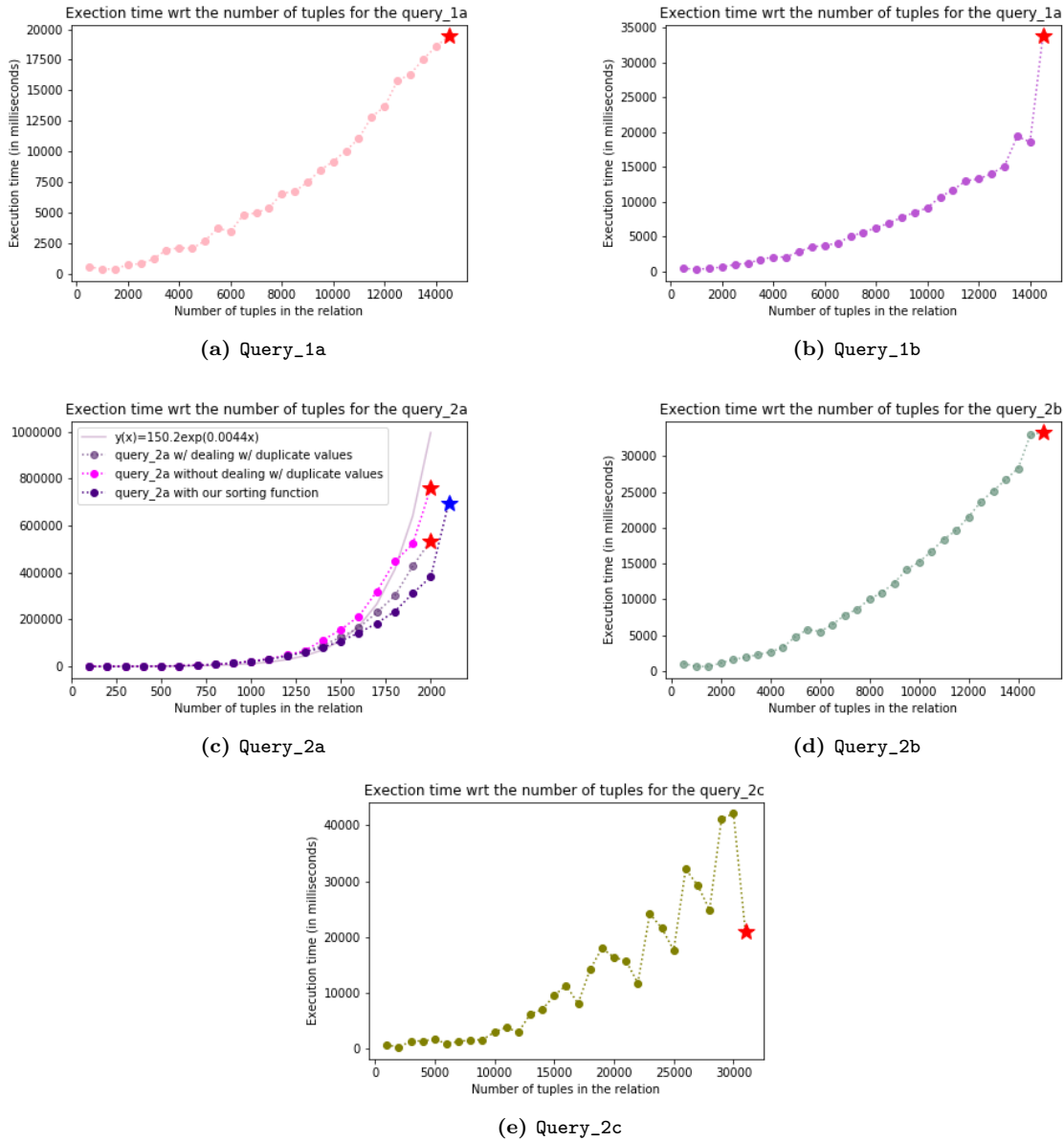


FIGURE 1: Execution time with respect to the size of the dataset (number of tuples) for each query

30min).

To have an idea of the time each query takes to be computed according to the size of the dataset, here is a plot with all the queries (right) and with all of them except **query_2a** (left) to be able to compare them more easily.

Here is finally an histogram reporting the maximum number of tuples supported by each query (or run in a appropriate time).

We can see that the **query_2c** supports a way more tuples than the other (30500 tuples), whereas **query_2a** supports a way less tuples (2100 tuples).

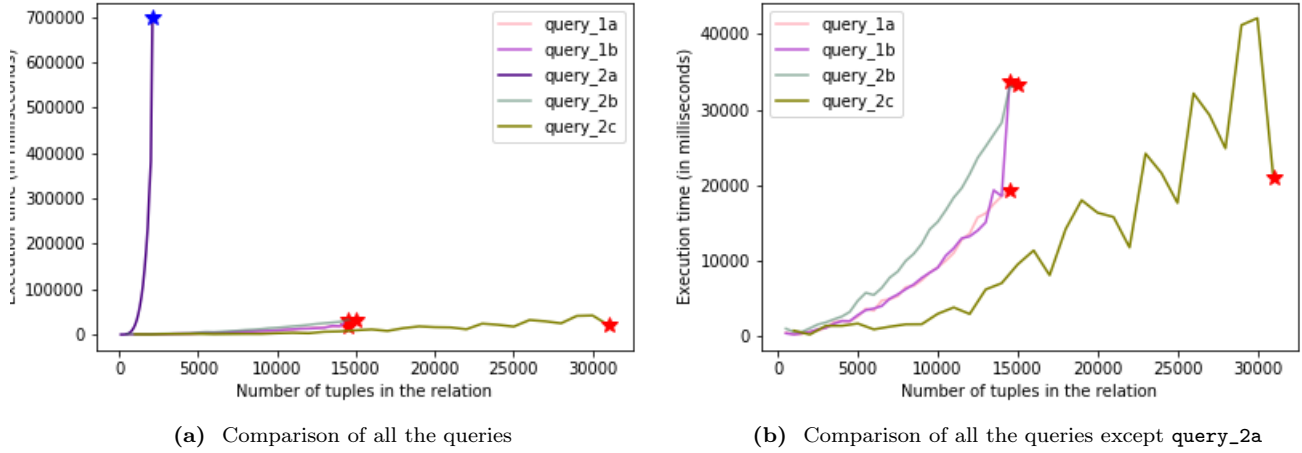


FIGURE 2: Comparison of the execution time between all the queries

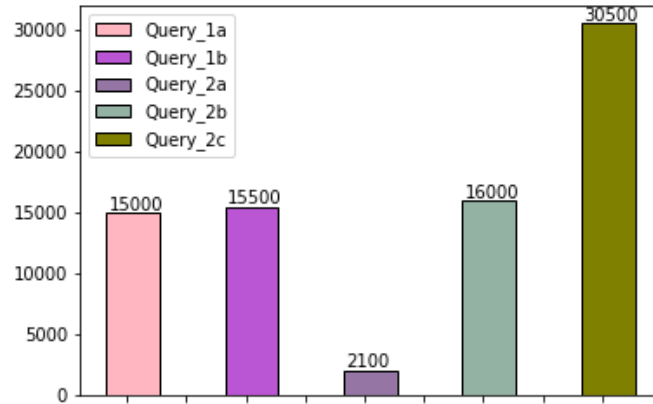


FIGURE 3: Histogram showing the maximum number of tuples supported by each query

5. Correctness

To have an insight of the correctness of the algorithms, we computed the number of outputs for some queries on tables R, S and Q that were given. Please refer to the images.

5.1. Comparison of NLJ and Self Join with one predicate

We obtained the exact same number of outputs. Please see the images below.

5.2. Comparison of NLJ and Self Join with two predicates

We obtained the exact same number of outputs in each case for this query. However, by testing another query, we realized that the algorithms return different number of outputs, which means that they fail to compute the exact outputs that are expected. This behavior could not have been shown for small tables but it has been revealed for larger ones. Please refer to the images.

Here is another example where the outputs are different :

```

Script started on 2020-01-20 16:19:11+0100
-kamets make jointest2
/usr/lib/jvm/java-11-openjdk-amd64/bin/javac -classpath ../ TestDriver.java
Jointest2.java
Note: Jointest2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
/usr/lib/jvm/java-11-openjdk-amd64/bin/java -classpath ../ tests.Jointest2
Replacer: Clock

*****Query1a being read *****
Query:
  SELECT R.col1 S.col1
  FROM   R, S
  WHERE  R.col3 = S.col3

Any resemblance of persons in this database to people living or dead
is purely coincidental. The contents of this database do not reflect
the views of the University, the Computer Sciences Department or the
developers...

*****Query1a starting *****
After Building btree index on Table1.3.

[24956, 589538]
[24956, 324516]
[24956, 1444389]
[24956, 1859882]
[24956, 1138993]

```

(a) First part of the transcript

```

[1987935, 1987155]
[1948029, 1948029]
[1948029, 1944146]
[1948029, 1948310]
[1948029, 1992635]
[1948029, 1972968]
[1948029, 1987155]
[1944146, 1944146]
[1944146, 1948310]
[1944146, 1992635]
[1944146, 1972968]
[1948310, 1948310]
[1948310, 1992635]
[1948310, 1972968]
[1948310, 1987155]
[1972968, 1992635]
[1972968, 1972968]
[1972968, 1987155]
[1987155, 1992635]
[1987155, 1987155]
[1992635, 1992635]
20101

Finished joins testing
join tests completed successfully
-kamets exit

Script done on 2020-01-20 16:19:37+0100

```

(b) Second part of the transcript

FIGURE 4: query_1a using NLJ for one predicate

```

Script started on 2020-01-20 16:20:28+0100
-kamets make jointest2
/usr/lib/jvm/java-11-openjdk-amd64/bin/javac -classpath ../ TestDriver.java
Jointest2.java
Note: Jointest2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
/usr/lib/jvm/java-11-openjdk-amd64/bin/java -classpath ../ tests.Jointest2
Replacer: Clock

*****Query2a being read *****
Query:
  SELECT Q.col1 Q.col1
  FROM   Q
  WHERE  Q.col3 = Q.col3

Any resemblance of persons in this database to people living or dead
is purely coincidental. The contents of this database do not reflect
the views of the University, the Computer Sciences Department or the
developers...

*****Query2a starting *****
After Building btree index on Table1.3.

***** Self Join 1 starting *****
[1992635, 1992635]
[1987155, 1992635]
[1987155, 1987155]
[1972968, 1992635]

```

(a) First part of the transcript

```

[24956, 174711]
[24956, 154178]
[24956, 151959]
[24956, 142845]
[24956, 148214]
[24956, 138806]
[24956, 138996]
[24956, 128314]
[24956, 118361]
[24956, 109869]
[24956, 105639]
[24956, 102597]
[24956, 69723]
[24956, 66742]
[24956, 42889]
[24956, 39239]
[24956, 34826]
[24956, 34476]
[24956, 31895]
[24956, 27652]
[24956, 24956]
20101

Finished joins testing
join tests completed successfully
-kamets exit

Script done on 2020-01-20 16:20:35+0100

```

(b) Second part of the transcript

FIGURE 5: query_2a using Self Join for one predicate

```

1b_test.txt
Script started on 2020-01-20 16:24:18+0100
-kamets make jointest2
/usr/lib/jvm/java-11-openjdk-amd64/bin/javac -classpath ../ TestDriver.java
Jointest2.java
Note: Jointest2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
/usr/lib/jvm/java-11-openjdk-amd64/bin/java -classpath ../ tests.Jointest2
Replacer: Clock

*****Query1b being read *****
Query:
  SELECT R.col1 S.col1
  FROM   R, S
  WHERE  R.col3 = S.col3
        AND R.col4 = S.col4

Any resemblance of persons in this database to people living or dead
is purely coincidental. The contents of this database do not reflect
the views of the University, the Computer Sciences Department or the
developers...

*****Query1b starting *****
After Building btree index on Table1.3.

[24956, 589538]
[24956, 324516]
[24956, 1444300]
[24956, 1859082]

```

(a) First part of the transcript

```

1b_test.txt
[1907935, 1940029]
[1907935, 1944146]
[1907935, 1948310]
[1907935, 1992635]
[1907935, 1972960]
[1907935, 1987155]
[1940029, 1944146]
[1940029, 1948310]
[1940029, 1992635]
[1940029, 1972960]
[1940029, 1987155]
[1944146, 1948310]
[1944146, 1992635]
[1944146, 1972960]
[1944146, 1987155]
[1948310, 1992635]
[1948310, 1972960]
[1948310, 1987155]
[1972960, 1992635]
[1972960, 1987155]
[1987155, 1992635]
number of outputs = 19872

Finished joins testing
join tests completed successfully
-kamets exit

Script done on 2020-01-20 16:24:39+0100

```

(b) Second part of the transcript

FIGURE 6: query_1b using NLJ for two predicates

```

2b_test_bis
Script started on 2020-01-20 16:35:01+0100
-kamets make jointest2
/usr/lib/jvm/java-11-openjdk-amd64/bin/javac -classpath ../ TestDriver.java
Jointest2.java
Note: Jointest2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
/usr/lib/jvm/java-11-openjdk-amd64/bin/java -classpath ../ tests.Jointest2
Replacer: Clock

*****Query2b being read *****
Query:
  SELECT Q.col3 Q.col3
  FROM   Q
  WHERE  Q.col4 = Q.col4
        AND Table1.col1 = Table2.col1

Any resemblance of persons in this database to people living or dead
is purely coincidental. The contents of this database do not reflect
the views of the University, the Computer Sciences Department or the
developers...

*****Query2b starting*****
After Building btree index on Table1.3.

***** Self Join 2 starting *****
[199263500, 198715500]
[198715500, 197296000]

```

(a) First part of the transcript

```

2b_test_bis
[183853200, 2495600]
[184582000, 2495600]
[185342700, 2495600]
[185908200, 2495600]
[186392100, 2495600]
[186635100, 2495600]
[187631000, 2495600]
[188142400, 2495600]
[188632200, 2495600]
[189881700, 2495600]
[190159700, 2495600]
[190425100, 2495600]
[190463600, 2495600]
[190793500, 2495600]
[194002900, 2495600]
[194414600, 2495600]
[194831000, 2495600]
[197296000, 2495600]
[198715500, 2495600]
[199263500, 2495600]
number of outputs = 19888

Finished joins testing
join tests completed successfully
-kamets exit

Script done on 2020-01-20 16:36:53+0100

```

(b) Second part of the transcript

FIGURE 7: query_2b using Self Join for two predicates

```

2c_2_test.txt ~
Script started on 2020-01-20 16:28:15+0100
-kamets make jointest2
/usr/lib/jvm/java-11-openjdk-amd64/bin/javac -classpath ../ TestDriver.java
Jointest2.java
Note: Jointest2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
/usr/lib/jvm/java-11-openjdk-amd64/bin/java -classpath ../ tests.Jointest2
Replacer: Clock

*****Query2c being read *****
Query:
SELECT R.col1 Q.col1
FROM
  R, Q
WHERE R.col3 1 Q.col3
      AND R.col4 4 Q.col4

Any resemblance of persons in this database to people living or dead
is purely coincidental. The contents of this database do not reflect
the views of the University, the Computer Sciences Department or the
developers...

*****Query2c starting *****
AND
[1987155, 1992635]
[1972960, 1987155]
[1972960, 1992635]
[1948310, 1972960]
[1948310, 1987155]

```

(a) First part of the transcript

```

2c_2_test.txt ~
[24956, 1838532]
[24956, 1845820]
[24956, 1853427]
[24956, 1859882]
[24956, 1863921]
[24956, 1866351]
[24956, 1876319]
[24956, 1881424]
[24956, 1881784]
[24956, 1886322]
[24956, 1888817]
[24956, 1901597]
[24956, 1904251]
[24956, 1904636]
[24956, 1907935]
[24956, 1940829]
[24956, 1944146]
[24956, 1948310]
[24956, 1972960]
[24956, 1987155]
[24956, 1992635]
number of outputs = 19872

Finished joins testing
join tests completed successfully
-kamets exit

Script done on 2020-01-20 16:28:37+0100

```

(b) Second part of the transcript

FIGURE 8: query_2c using Ie Join

```

1b_testbis.txt ~
Script started on 2020-01-20 16:37:06+0100
-kamets make jointest2
/usr/lib/jvm/java-11-openjdk-amd64/bin/javac -classpath ../ TestDriver.java
Jointest2.java
Note: Jointest2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
/usr/lib/jvm/java-11-openjdk-amd64/bin/java -classpath ../ tests.Jointest2
Replacer: Clock

*****Query1b being read *****
Query:
SELECT R.col3 S.col3
FROM
  R, S
WHERE R.col4 2 S.col4
      AND R.col1 4 S.col1

Any resemblance of persons in this database to people living or dead
is purely coincidental. The contents of this database do not reflect
the views of the University, the Computer Sciences Department or the
developers...

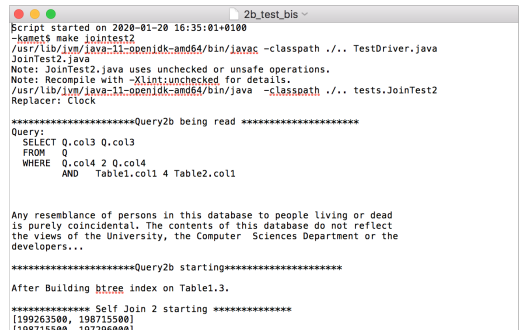
*****Query1b starting *****
After Building btree index on Table1.3.
[34149600, 54325566]
[35052900, 54325566]
[35060100, 54325566]
[35664300, 54325566]
[36168500, 54325566]
[36289600, 54325566]
[37737600, 54325566]
[38707400, 54325566]
[39979200, 54325566]
[40624400, 54325566]
[40649600, 54325566]
[41854800, 54325566]
[42049600, 54325566]
[42049600, 54325566]
[42065800, 54325566]
[43312900, 54325566]
[43400100, 54325566]
[43727600, 54325566]
[44318900, 54325566]
number of outputs = 19

Finished joins testing
join tests completed successfully
-kamets exit

Script done on 2020-01-20 16:37:26+0100

```

FIGURE 9: query_1b using NLJ for two predicates



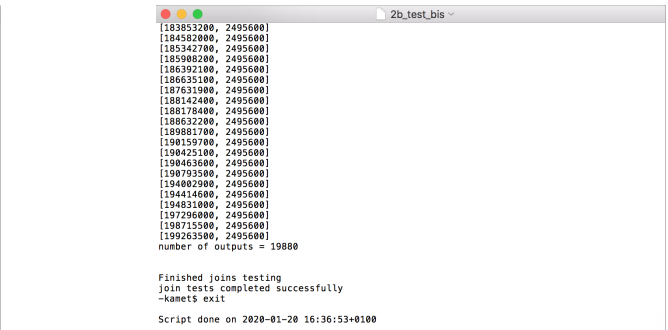
```
Script started on 2020-01-20 16:35:01+0100
-kamets make jointest2
/usr/lib/jvm/java-11-openjdk-amd64/bin/javac -classpath ../ TestDriver.java
Jointest2.java
Note: Jointest2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
/usr/lib/jvm/java-11-openjdk-amd64/bin/java -classpath ../ tests.Jointest2
Replacer: Clock

*****Query2b being read *****
Query:
  SELECT Q.col3 Q.col3
  FROM   Q
  WHERE  Q.col4 2 Q.col4
        AND  Table1.col1 4 Table2.col1

Any resemblance of persons in this database to people living or dead
is purely coincidental. The contents of this database do not reflect
the views of the University, the Computer Sciences Department or the
developers...

*****Query2b starting*****
After Building btree index on Table1.3.

***** Self Join 2 starting *****
[199263500, 198715500]
[198715500, 197296800]
```



```
[183853200, 2495600]
[184582000, 2495600]
[185342700, 2495600]
[185900200, 2495600]
[186392100, 2495600]
[186635100, 2495600]
[187631000, 2495600]
[188142400, 2495600]
[188178400, 2495600]
[188632200, 2495600]
[189881700, 2495600]
[190159700, 2495600]
[190425100, 2495600]
[190463600, 2495600]
[190793500, 2495600]
[194082900, 2495600]
[194414600, 2495600]
[194831000, 2495600]
[197296800, 2495600]
[198715500, 2495600]
[199263500, 2495600]
number of outputs = 19880

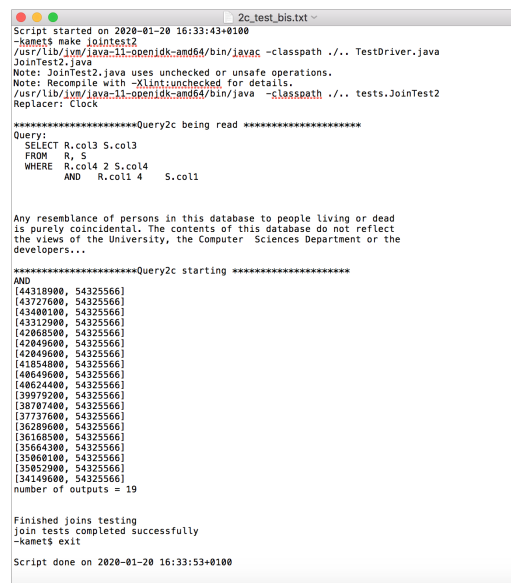
Finished joins testing
join tests completed successfully
-kamets exit

Script done on 2020-01-20 16:36:53+0100
```

(a) First part of the transcript

(b) Second part of the transcript

FIGURE 10: query_2b using Self Join for two predicates



```
Script started on 2020-01-20 16:33:43+0100
-kamets make jointest2
/usr/lib/jvm/java-11-openjdk-amd64/bin/javac -classpath ../ TestDriver.java
Jointest2.java
Note: Jointest2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
/usr/lib/jvm/java-11-openjdk-amd64/bin/java -classpath ../ tests.Jointest2
Replacer: Clock

*****Query2c being read *****
Query:
  SELECT R.col3 S.col3
  FROM   R, S
  WHERE  R.col4 2 S.col4
        AND  R.col1 4 S.col1

Any resemblance of persons in this database to people living or dead
is purely coincidental. The contents of this database do not reflect
the views of the University, the Computer Sciences Department or the
developers...

*****Query2c starting *****
AND
[44318900, 54325560]
[43727600, 54325560]
[43480100, 54325560]
[43312900, 54325560]
[42068500, 54325560]
[42049600, 54325560]
[42049600, 54325560]
[41854000, 54325560]
[40649600, 54325560]
[40624400, 54325560]
[39979200, 54325560]
[38787400, 54325560]
[37737600, 54325560]
[36286600, 54325560]
[36168500, 54325560]
[35664300, 54325560]
[35868100, 54325560]
[35852900, 54325560]
[34149600, 54325560]
number of outputs = 19

Finished joins testing
join tests completed successfully
-kamets exit

Script done on 2020-01-20 16:33:53+0100
```

FIGURE 11: query_2c using Left Join

6. README

The function `main` to run our code is in the class `JointTest2` accessible with the path :

```
/minjava/javaminibase/src/tests/JoinTest2
```

After running the code, you should be asked "the number of tables you need" to compute your query (the number of tables has been given in the queries text files given in the folder `QueriesData_newvalues`, available on My Eurecom). Here is a table resuming the number of table(s) needed for each query.

	query_1a	query_1b	query_2a	query_2b	query_2c
Number of table(s)	2	2	1	1	2

Then, depending on the number of table needed you gave, you should enter the path to the table(s). This path should be given **without quotation marks**, and give the access to the text file you want. For example, if your table, named `table_test.txt`, is in the folder `QueriesData_newvalues`, you should give in the console this path :

```
../../../../QueriesData_newvalues/table_test.txt
```

You should be asked then the path to your query's text file. You should do exactly as before. For example, if your query, named `query_test.txt`, is in the folder `QueriesData_newvalues`, you should give in the console this path :

```
../../../../QueriesData_newvalues/query_test.txt
```

A last thing that can be asked you is the algorithm you want to use if you need two tables, and it is a double predicates case (as explained in section 2.2). You should then enter 1 for Nestead Loop Join (`query_1b`) and 2 for IEJoin (`query_2c`).