Minibase Join Operator

EURECOM - DBSys Fall 2018/2019
Authors:
Alaa Boukhary & Oussama Kandakji

I. Introduction

The objective of this project is to extend the code and capabilities of Minibase [1] to support fast and scalable Inequality Joins as well as Query Parsing. The java code in this project is scaled to fit and support join operations on large relations. The default join operation implemented in Java Minibase is a nested loop join which is considered a naive approach for handling joins on large relations. The join we implemented in this project is based on the paper "Fast and Scalable Inequality Joins"[2]. The runtime and performance for each of the implemented join operators will be studied, compared, and tested.

II. File & Code Structure

The main directory of the project is composed of three primary directories: Code, Output, Report.

i. The Code Directory

This directory contains all the source code classes running the project. Above the *Minibase* source code, the newly added or updated classes and code files are in two directories: /*iterator* and /tests

/iterator contains the following files:

- *SelfJoinOnePredicate.java:* This class contains an implementation of the self inequality joins with one predicate (inherits from Iterator.java class).
- *SelfInequalityJoinTwoPredicate.java:* This class contains an implementation of the self inequality joins with two predicates (inherits from Iterator.java class).
- *SelfInequalityJoinTwoPredicateOptimized.java:* This class contains an implementation of the optimized self inequality joins with two predicates using bitmap index (inherits from Iterator.java class).
- *InequalityJoinTwoPredicates.java:* This class contains an implementation of the inequality joins with two predicates (inherits from Iterator.java class).
- *InequalityJoinTwoPredicatesOptimized.java:* This class contains an implementation of the inequality joins with two predicates (inherits from Iterator.java class).

- *Utils.java:* This class contains some functions that are used by the inequality joins.
- *Row.java:* This class is used to represent a tuple to be sorted in an array.
- *RowWithTuple.java:* This class inherits from the class *Row* and stores the tuple field to be selected.
- Iterator.java: this class was updated to sequentially read the maximum number of tuples into a heapfile.
- *SortAscending.java:* This class is created to be used by the function Collections.Sort to sort an array of tuples (Row objects) in ascending order.
- *SortDescending.java:* This class is created to be used by the function Collections.Sort to sort an array of tuples (Row objects) in ascending order.
- NestedLoopsJoins.java: this class was updated in the way it handles the join termination.

/tests contains the following files:

- *QueryParser.java*: this class is responsible for parsing the query file to identify the parameters of the query, such as the involved relations, the output fields, the types of the fields, etc.
- *plot1.java, plot2.java, plot3.java*: are classes that generate plots and runtimes for the join operators in the project.
- Task1a.java: this class runs Task 1a of the assignment.
- Task1b.java: this class runs Task 1b of the assignment.
- Task2a.java: this class runs Task 2a of the assignment.
- Task2b.java: this class runs Task 2b of the assignment.
- Task2c.java, Task2c_1.java. Task2c_2.java: these classes run Task 2c of the assignment.
- *Task2d_2b.java, Task2d_2c, Task2d_2c_1, Task2d_2c_2*: these classes run Task 2d
- *ParsetTestIEJoin.java*: this class uses the *QueryParser.java* class that extracts the arguments used by the inequality joins and calls the specified inequality join class (depending on task we are executing).

ii. The Output Directory

The Output directory contains the relation files (R, S, & Q) as well as the query files that are to be parsed and executed. In addition, this directory will be used to store the output files of the queries in the done tasks.

iii. The Report Directory

This directory contains the report of this project and experiment results.

III. High Level Details

- The Query Parser: QueryParser.java is a class that takes a query text file and collects the information needed about the query and its concerned relations. This parser computes the number and offset of the output fields, the types of the fields inside the relations, the characteristics of the predicates, as well as the outer and inner relation fields. The arguments computed by this class are used by all the join operators (the nested loop join and the inequality joins).
- The Iterator: Iterator.java is the base class of the join operator, and since the relation Q doesnt fit in one heapfile (doesn't fit in memory) when the relation is too large, tuples have to be read sequentially in chunks, in such a way that the maximum number of tuples is read at each time. All the inequality joins' constructors call the function *getNextHeapFile()* to retrieve a new heapfile each time it is called until all the tuples are read from the text file that contains the tuple of a certain relation.
- Since we are using many heap files sequentially to read the relation Q, we added a number called *heapfile_number* to make sure tuples have unique record ids.

IV. Task 1a & 1b

These cover the implementation of Nested-Loop-Join in single and double predicates respectively. The two tasks can be run through *Task1a.java* and *Task1b.java*. The latter classes will use *ParserTestNLJ.java* which implements a both small and large-scale NLJ.

What is basically done is to make an NLJ operation for each 2 heapfiles at a time in such a way that the two heapfiles fit in memory. The two heapfile are iterated to cover all the records in the input relation(s).

V. Task 2a & 2b

These cover the implementation of *Self Inequality-Join* in single and double predicates respectively. The two tasks can be run through *Task2a.java* and *Task2b.java*. The classes will use *SelfOnePredicate.java* and *SelfInequalityJoinTwoPredicate.java* that will be called using the interface class *ParserTestIEJoin.java.*

In the *selfOnePredicate.java* we don't use a bit array and a permutation array, since we simply emit all elements before the current element being read.

To deal with problem of duplicates in the self inequality joins we created a function (*Utils.predicate_evaluate*) that is used just before emitting a join result. This function makes sure that the tuples to be emitted satisfy the predicate(s) of the inequality join.

VI. Task 2c

This covers the implementation of *inequality-Join* in double predicates. The task can be run through *Task2c.java, Task2c_1.java, Task2c_2.java*. The classes will use the class *InequalityJoinTwoPredicates.java* that will be called using the interface class *ParserTestIEJoin.java.*

VII. Task 2d

This covers the implementation of *the optimized self* and *non self inequality-Join* in double predicates. The task can be run through *Task2d_2b.java, Task2d_2c, Task2d_2c_1, Task2d_2c_2*. The classes will use *SelfInequalityJoinTwoPredicateOptimized.java* and *InequalityJoinTwoPredicatesOptimized.java* that will be called using the interface class *ParserTestIEJoin.java.*

VIII. Experimentation and Performance Evaluation

i. Runtime NLJ vs Self IE Join Single Predicate

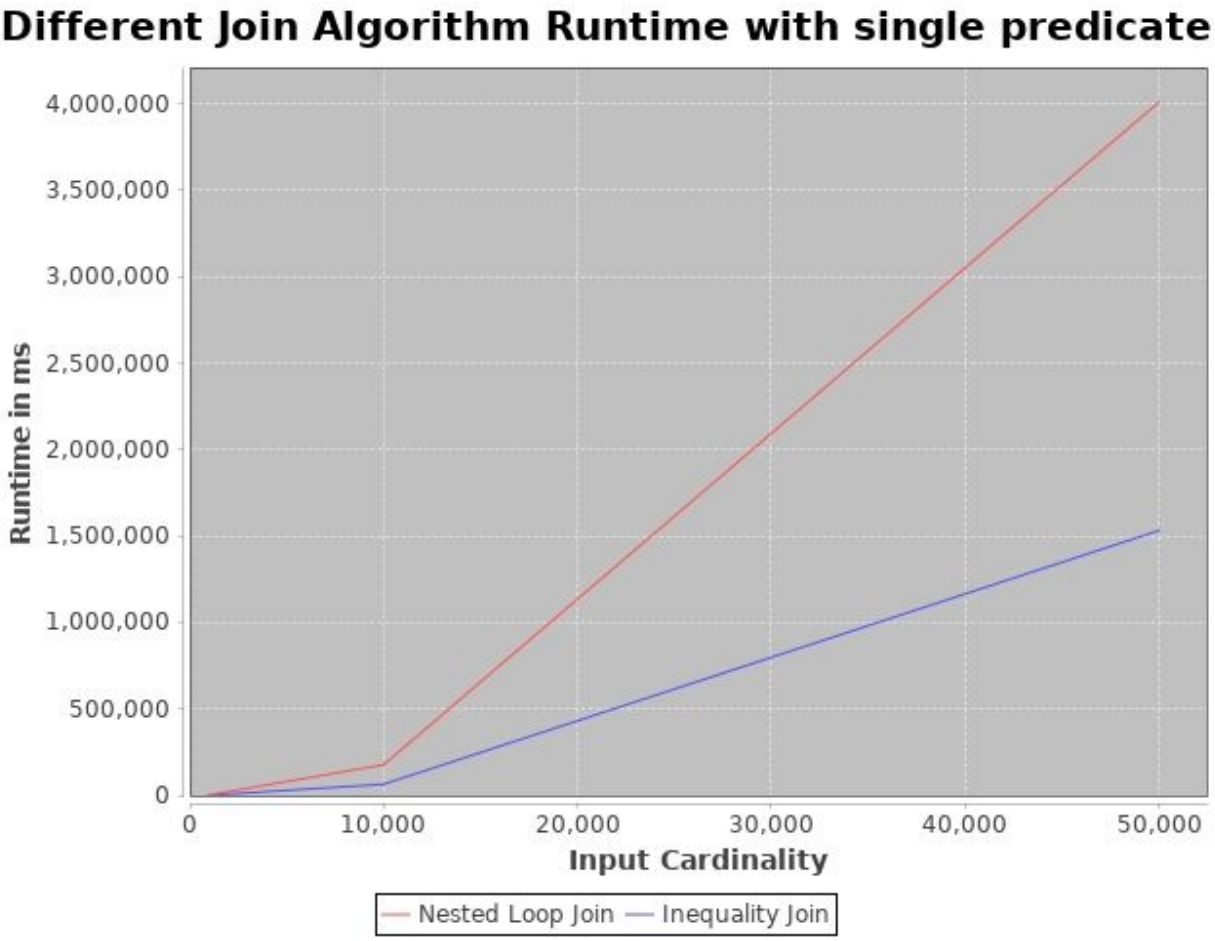

Figure 1 - Runtime of Several Join Algorithms as a Function of Input Cardinality

In the case of a single predicate, the runtimes at low input cardinalities is very close among NLJ and Self IE Join (1000 input tuples). However, with increasing input tuples (20000 and above), we start to observe a drastic difference in runtime and performance in the favor of Self IE join which has proven to be more scalable than NLJ.

Nevertheless, in this implementation of IE Join the bit array, as well as, the tuple arrays (*L1 & L2)* are in memory. However, this is not the case in commercial DBMS' where we cannot assume that *L1 & L2* can fit in RAM and thus are flushed to disk. Therefore, it is more interesting to see the runtime on this algorithm when L1 and L2 are on disk.

ii. Runtime Optimized vs Unoptimized Self IE Join with Two Predicates

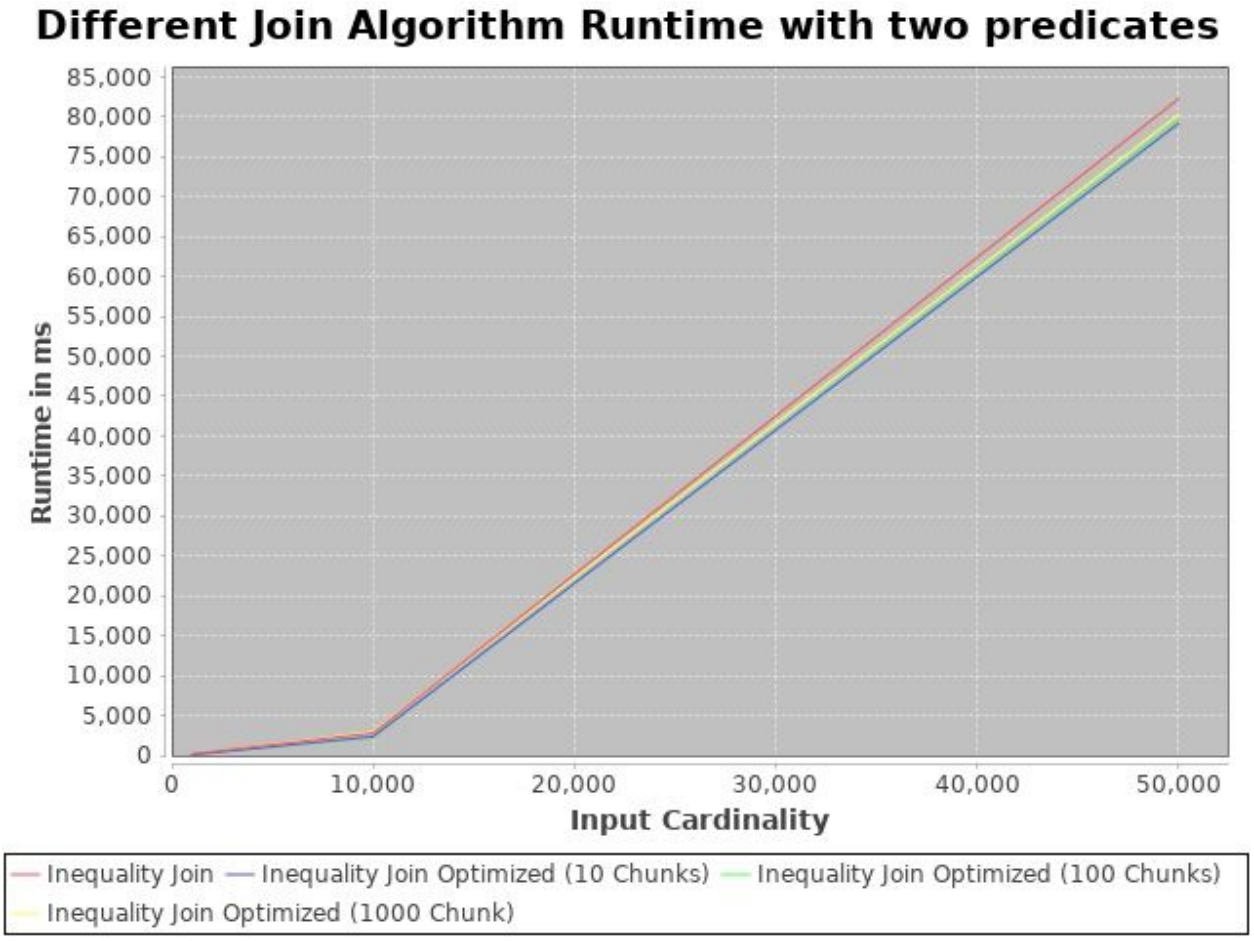

Figure 2 - Runtime of Optimized and Unoptimized Self IE Join

To emphasize, at low input cardinality, the runtime has not much of a difference among the variations of the join algorithms. As the input cardinality increases we start seeing slight difference in the runtime among the different inequality joins with the best being the *Inequality join* with 10 chunks bitmap. So with sufficiently large input cardinality the difference in runtime will become more significant assuming the selectivity stays the same.

iii. Runtime of Optimized Self IE Join with Two Predicates at Various Chunk Sizes with Constant Input Cardinality

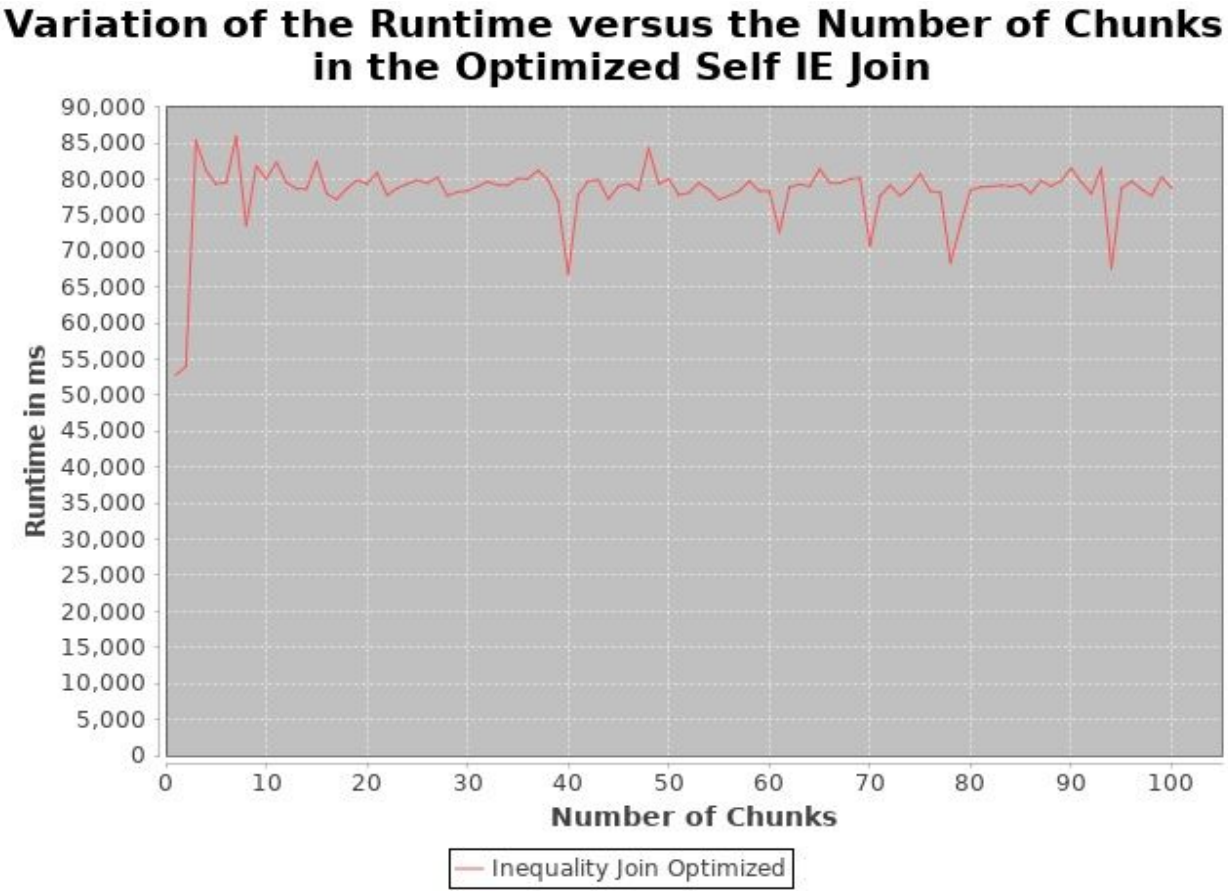

Figure 3 - Runtime of Optimized Self IE Join at Various Chunk Sizes<100

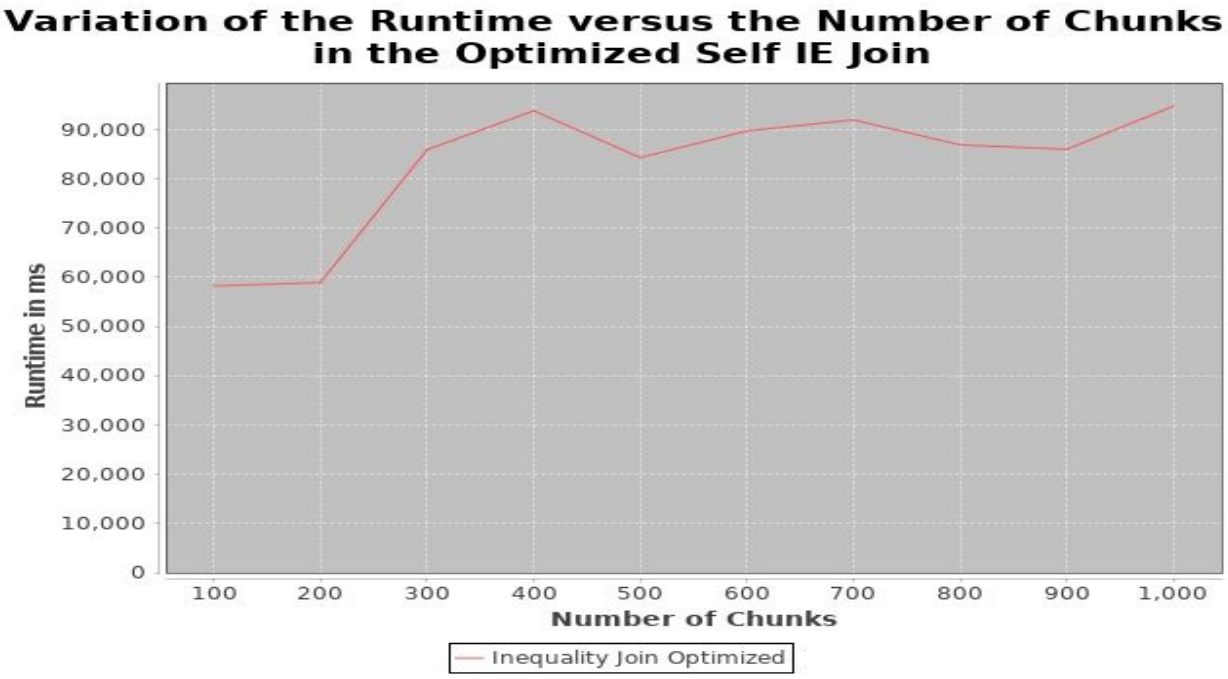

Figure 3 - Runtime of Optimized Self IE Join at Various Chunk Sizes>100

We can observe from the above plots that when chunk size equal to 1 we achieve the lowest runtime. The optimized inequality join with chunk size equal to 1 is equivalent to the unoptimized inequality join. So, in this case the optimized join is introducing an additional overhead to the algorithm and isn't optimizing it. The reason behind this result is that the selectivity of the used query is high so it is unavoidable to scan all the second array which makes the unoptimized version of the algorithm faster. So to properly use this algorithm we have to estimate to the selectivity of the query and decide from histogram plots and estimate if the proper method to be used is the optimized or the unoptimized version of the inequality join.

IX. Conclusion

In conclusion, in this project we've extended the capabilities of *Minibase* when it comes to parsing query files, implementing large-scale nested-loop-joins, supporting Self Inequality Joins, and Regular Inequality Joins, and their corresponding optimization. As a result, how the join operator implemented affects hugely the performance of the Database Management System. There are algorithms for Inequality Join that surpass the naive Nested Loop Join algorithm significantly when it comes to runtime and resource consumption. However, these algorithms' performance depends on several factors including the selectivity of the query, the input cardinality, and the number of chunks in case of the optimized IE Join.

References

[1] Minibase is a database management system intended for educational use. The goal is not just to have a functional DBMS, but to have a DBMS where the individual components can be studied and implemented by students.

[2] "Fast and Scalable Inequality Joins" is a paper by Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quian´e-Ruiz, Nan Tang, and Panos Kalnis that introduces fast inequality join algorithms based on sorted arrays and space efficient bit-arrays as well as a simple method to estimate the selectivity of inequality joins which is used to optimize multiple predicate queries and multi-way joins.