**INEQUALITY JOINS**

Inequality joins report with their comparative analysis

**Group 27**

Omkar Kaptan [1209385070]

Kavita Korgaonkar [1209283721]

Aarav Madan [1209374462]

Laxmikant Patil [1209141566]

Nisar Shaik [1209364620]

**Table of Contents**

[**ABSTRACT** 2](#_Toc446527318)

[**INTRODUCTION** 2](#_Toc446527319)

[**IMPLEMENTATION** 3](#_Toc446527320)

[**INEQUALITY SELF JOIN** 3](#_Toc446527321)

[**INEQUALITY JOIN** 3](#_Toc446527322)

[**OPTIMIZATION** 3](#_Toc446527323)

[**EXPERIMENT** 3](#_Toc446527324)

[**CONCLUSION** 4](#_Toc446527325)

[**APPPENDIX** 4](#_Toc446527326)

[**BIBLIOGRAPHY** 4](#_Toc446527327)

# **ABSTRACT**

Minibase is a database management system intended for educational use. It has a parser, optimizer, buffer pool manager, storage mechanisms and a disk space management system. The goal is not just to have a functional DBMS, but to have a DBMS where the individual component can be studied and implemented. Here MiniBASE Java is used to experiment with join operations – a fundamental database operation that requires both memory and time for its successful completion.

Join operation combines related tuples from same or different relations on different attributes schemes. Inequality joins, which join relational tables on inequality conditions are used in various applications. In this report, we have introduced fast inequality join algorithms with one or two predicate conditions. We have put columns to be joined in sorted arrays and used positional permutation arrays to encode positions of tuples in one sorted array w.r.t. the other sorted array. We have implemented a centralized version of these algorithms on top of PostgreSQL, and a distributed version on top of Spark SQL. We have compared against well-known optimization techniques for inequality joins and show that our solution is more scalable and several orders of magnitude faster. and compare its performance with sort-merge join and nested loop join with inequality predicates. It is evident from results that IEJoin outperforms other techniques for both single and two predicate inequality joins.

# **INTRODUCTION**

Joins are one of the most crucial operators in a database. Based on the join predicates and the algorithm used for the join, the result generation may range from a few seconds to few days. Thus, careful analysis of the the join techniques is essential for implementing the most efficient algorithm for the relations and the predicates in consideration.

Minibase supports Nested loop join and Sort-Merge Join [3]. To optimize the performance of selections and projection on attributes, these operations are implemented while reading data from files. To access data from the files containing records, Iterators are used which can return data in desired sorted order.

Join is generally performed by doing a selection or projection on a Cartesian product operation. This operation is time consuming and hence some databases use indices such as B+trees. Using B+trees may not result in full performance as the index is unclustered.

As an extension to the existing algorithms, we implement the space efficient inequality join ( IE Join). Two variants of the IE join exist: one for the generic two predicate join between two tables and the other one for two predicated self join. These queries contain predicate of the form where are attributes in relation R & S respectively and op is an inequality operator in {<,>,}..

# **IMPLEMENTATION**

Given data tables –

Sailor {sid, sname, rating, age} 🡪 S

Boats {bid, bname, color} 🡪 B

Reserves {sid, bid, date} 🡪 R

## **INEQUALITY SELF JOIN**

The SQL SELF JOIN is used to join a table to itself as if the table were two tables. Queries with self join contain predicate of the form & where are attributes in relation R & S respectively and op is an inequality operator in {<,>,}. For such join, we have implemented the algorithm. It takes self-join inequality query as an input with tables R and S.

Queries are stored in input file in the below format.

R\_1 S\_2 Columns to be displayed

R Relation names

R\_3 1 R\_3 Join column

When query starts executing, the algorithm sorts the join attribute column value list. After sorting, next step is comparison. Record ID and join attribute value are getting fetched from table and stored in separate array. 2 records are get`ompared in that array based on join attribute values. If value matches then it is getting printed. Comparison includes computing permutation array which maps the occurrences of records in one sort order to the occurrences of those records in another sorted order. The permutation array is scanned sequentially and a track of the scanned records is maintained in a bit array which allows for space efficient calculation of the joins.

Pseudo code –

**Input –** Query Q with 2 join predicates on tables with size n.

**Output –** a list of join records.

Sorted array

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

Let be the sorted array of columns R & S.

If ()

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

Sort in ascending order

Sorted array

Else if ()

Sort in descending orde

B

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 0 |

If () P

Sort in descending order

Else if ()

Sort in ascending order

Compute permutation array P of w.r.t.

Initialize bit-array B (|B| = n) and set all bits to 0

If ( && eqOff=0

Else eqOff=1

For (i1 to n)

Pos p[i]

B[pos]i

For (jpos+eqOff to n) do

If B[j]=1 then

Add tuples w.r.t. ( to

Join\_result

Return join\_result

Complexity – Time complexity of IESelfJoin in O() & space complexity is o(n).

## **INEQUALITY JOIN**

The INEQUALITY JOIN is used to join a table with other table with join attribute. Inequality join algorithm takes a query with one or two inequlity join conditions as input and returns a set of result records. Queries with inequality join contain predicate of the form & where are attributes in relation R & S respectively and op is an inequality operator in {<,>,}. For such join, we have implemented the algorithm. It takes inequality query as an input with tables R and S.

Queries are stored in input file in the below format.

R\_1 S\_2 Columns to be displayed

R Relation names

R\_3 1 R\_3 Join column

It first sorts the attribute values to be joined, computes permutation array for both the tables by proceeding in the same way as in case of two predicate IE join. Two offset arrays are also created for both join attributes in the respective predicates. These offset arrays map the position of records sorted on X attribute in R table to the record sorted on X’ attribute in S table where X maps to X’. Finally, a bit array is maintained as in case of self-join for computing the join tuples. The offset arrays along with the permutation arrays help in accessing the records according to constraints imposed by the specified predicates.

Pseudo code –

**Input –** Query Q with 2 join predicates && , tables T,T’ of sizes m and n respecively

**Output –** a list of join records.

Sorted array

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

Let be the sorted array of columns X & Y in T.

Let be the sorted array of columns X’ & Y’ in T’

If ()

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

Sort in descending order

Sorted array

Else if ()

Sort , in ascending orde

B

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 0 |

If () P

Sort in ascending order

Else if ()

Sort , in descending order

Compute permutation array P of w.r.t.

Compute permutation array P’ of ’ w.r.t. ’

Compute the offset array of w.r.t.

Compute the offset array of w.r.t.

Initialize bit-array B’ (|B’| = n) and set all bits to 0

If ( && eqOff=0

Else eqOff=1

For (i1 to m) do

[i]

For j 🡨 1 to do

B’[P’[j]]1

P[[i]]

For (k 🡨 + eqOff to n) do

If B’[k]1 then

Add tuples w.r.t. () to join result

Return join\_result

Complexity – Time complexity of IESelfJoin in O() & space complexity is O(m + n).

# **OPTIMIZATION**

While working with bit array (B), in case of low selectivity it becomes difficult to scan all 0’s on right of the current position every time. So for optimization of Bit-array scan, we have used Bloom filter. Given a bit array B with size n and a predefined chunk size c, we have created new bit array with size [n/c] where each bit corresponds of new array corresponds to a chunk in B with 1 indicating that the chunk contains atleast a 1 and 0 otherwise. It saves time of scanning huge bit array every time.

# **EXPERIMENT**

# **CONCLUSION**

# **APPPENDIX**

# **BIBLIOGRAPHY**

[1] Gehrke, Ramakrishnan: “Database Management Systems”

[2] Z. Khayyat, W. Lucia, M. Singh, M. Ouzzani, P. Papotti, J. Quiane-Ruiz, N. Tang, P. Kalnis. [Lightning Fast and Space Efficient Inequality Joins.](https://williamlucia.github.io/publications/) In: Proceedings of the 42nd International Conference on Very Large Data Bases (VLDB), pp. 2074-2085, New Delhi, India (2016)

[3] <http://research.cs.wisc.edu/coral/mini_doc/joins/j.html>