Department of Informatics, University of Zürich

MSc Thesis

Implementing Inequality and Interval Overlap Joins

Youngseo Kim

Matrikelnummer: 20-744-470

Email: youngseo.kim@uzh.ch

November 27, 2024

supervised by Dr. Sven Helmer and Jamal Mohammed

University of Zurich[UZH]

Department of Informatics

Dedicate to my parents, who supported me unconditionally

Acknowledgements

First and foremost, I would like to express my gratitude to my two advisors, Dr. Sven Helmer and Jamal Mohammed. Working with these amazing researchers has been an incredible journey into algorithm development. Their support and guidance throughout this process have been invaluable. They assisted me whenever I encountered difficulties, taught me the essence of academic research, and helped me develop skills I lacked. Thanks to them, I thoroughly enjoyed this entire learning experience.

I would also like to extend my heartfelt thanks to my family. Mum, your support until the very end has been my pillar of strength. Dad, thanks a lot for your encouragement and belief in my career. Lastly, I am grateful to my best friends for their constant moral support.

Abstract

Joining relational tables using inequality predicates is usually slow, often requiring quadratic runtime due to a nested-loop join. A study conducted by Khayyat et al. introduced an optimized sort-merge join by constructing a bit array on the sorted tuples. It implemented a Bloom filter algorithm that used a bit-array index for the join filter. Another challenging join predicate involves checking whether two intervals overlap. Dignös et al. investigated a method which combined interval-overlap predicates with an equality join predicate. They formulated joins as unions of range joins with an additional equality condition. However, these approaches do not work if the equality predicate is changed to inequality, highlighting the limitations of the current methods. Therefore, this study attempts to overcome these limitations by proposing a new approach that introduces additional inequalities to both interval-overlap join and inequality join algorithms and compares their efficiency.

Zusammenfassung

Die Verknüpfung von relationalen Tabellen mit Hilfe von Ungleichheitsprädikaten ist in der Regel langsam und erfordert aufgrund einer verschachtelten Schleifenverknüpfung oft quadratische Laufzeiten. In einer von Khayyat et al. durchgeführten Studie wurde eine optimierte Sort-Merge-Verknüpfung eingeführt, indem ein Bit-Array auf den sortierten Tupeln konstruiert wurde. Es wurde ein Bloom-Filter-Algorithmus implementiert, der einen Bit-Array-Index für den Verknüpfungsfilter verwendet. Ein weiteres anspruchsvolles Join-Prädikat ist die Überprüfung, ob sich zwei Intervalle überschneiden. Dignös et al. untersuchten eine Methode, die Intervall-Überlappungsprädikate mit einem Gleichheits-Join-Prädikat kombinierte. Sie formulierten Joins als Vereinigungen von Range-Joins mit einer zusätzlichen Gleichheitsbedingung. Dieser Ansatz funktioniert jedoch nicht, wenn das Gleichheitsprädikat in eine Ungleichheit geändert wird, was die Grenzen der derzeitigen Methoden aufzeigt. Daher wird in dieser Studie versucht, diese Einschränkungen zu überwinden, indem ein neuer Ansatz vorgeschlagen wird, der zusätzliche Ungleichheiten in die Algorithmen für Intervall-Überlappungs-Joins und Ungleichheits-Joins einführt und ihre Effizienz vergleicht.

Contents

List of Figures

List of Tables

1 Introduction

Joins are frequent yet resource-intensive tasks in database systems [DBG+22]. They are essential for many database applications that contain numerous relational tables. Nevertheless, as the size of the database increases, processing relational joins becomes challenging. Hence, improving join processing performance is of interest in database study [RLT20].

However, joining with inequality predicates (e.g. $\neq, <, >, \leq, \geq$) is prohibitively slow because typically, it requires nested loops as the database cannot leverage efficient indexing or hashing, unlike joining with an equality predicate ($=$) [GMUW08]. In an inequality join, every tuple in the outer relation is compared with every tuple in the inner relation [Che24]. Therefore, it leads to a quadratic time complexity, $O(n \times m)$, where n and m respresent the number of rows in two different tables, respectively.

Several attempts have been made to address this issue. One of the attempts was made by Khayyat et al., suggesting a novel inequality join using an optimized sort-merge join. This is because a sort-merge join narrows the search area by organizing the data according to the joining attributes and then combining them. They constructed a space-efficient bit-array on the sorted tuples and implemented a Bloom filter algorithm using bit-array indexes for a join filter [KLS+17]. However, this study tends to overlook the fact that inequality joins are often a key component in more complex joins, such as interval overlap joins, where additional optimization strategies are needed. An even more challenging join predicate involves checking whether intervals overlap. Hence, Dignös et al. investigated a method that combines interval-overlap predicates with an equality predicate. To accomplish this, they developed an overlap join algorithm with the union of two disjoint range joins [DBG+22].

Despite these improvements, joining relational tables with inequalities has limitations. For instance, the above-mentioned studies address some aspects, but they are limited to comparing two inequality predicates. Hence, the process needs to be addressed when multiple inequality predicates are combined. Therefore, this study reviews the inequality join algorithm with bloom filter and the interval overlap join algorithm, aiming to provide a novel way of joining intervals with further inequalities. More specifically, the study introduces a novel method to add an inequality predicate to compare indexes in addition to two inequality predicates for overlapping time intervals. A holistic approach is utilized, integrating additional inequalities in both inequality and overlap join algorithms.

This study describes the design and implementation of a new methodology that merges additional inequality predicates into the inequality and overlap merge algorithms. This proposed methodology provides new insights into joining relational tables with multiple inequalities. The thesis is composed of the following outline.

Outline It begins by the previous studies related to this research, providing a solid foundation for the subsequent chapters. The third chapter delves into a detailed introduction to the concepts and definitions necessary for understanding this thesis. The fourth section introduces the brute force nested loop join algorithm, which is the baseline of this study. Chapter Five analyzes the concepts of inequality join with Bloom filters for the bit-array scan introduced by Khayyat et al., related algorithms from the paper, and the implementation of the newly modified inequality join algorithm. Chapter Six, similar to chapter five, presents the interval overlap join algorithms studied by Dignös et al. regarding its concept and algorithms, focusing on how this study adopted additional inequality predicates into the existing algorithm. The evaluation part deals with the selectivity and complexity of attributes for the different approaches regarding run time and memory footprint. Lastly, in the last chapter, we conclude the findings and limitations of this study and offer future suggestions for further research.

2 Related Work

The related work section is divided into two parts to explain the research that is the basis of this thesis. First, we examine the overall concept of the inequality join. Second, we review interval overlap merge join.

Inequality joins are utilized in different applications to join relations with inequality predicates. However, inequality joins have been studied less than equality joins. Moreover, queries containing such joins are slow compared to other join algorithms [KLS+17]. The problem stems from its quadratic execution time because inequality joins require nested-loops to compare each value from both relational tables [Che24]. Hence, Khayyat et al. propose a new method based on bit-arrays and positional permutation arrays for inequality join, and named it as IEJOIN. The central concept involves forming a sorted array of tuples for each inequality comparison and determining their intersection to produce results. The permutation array reduces the cost of locating intersections by encoding the positions of tuples in one sorted array relative to another sorted array, given that there are only two conditions. Then, a bit-array helps generate join results. With this method, it achieves significant performance improvement compared to previous methods as it leverages contiguous memory structures [KLS+17].

We now turn to the second part of the related work section, focusing on overlap joins associated with time using equality predicates. Overlap joins rely on inequalities, which makes them challenging to compute efficiently. In the present study, we assume two relational tables **r** and **s** with the interval attribute **P**, including the start boundary B and the end boundary E. Given these two relations with intervals, SQL interprets overlap as follows:

```
r.P OVERLAPS s.P ≡ r.B < s.E AND s.B < r.E
```

However, optimizing this query is challenging because it contains two inequality predicates over four different attributes, and each condition is only bounded on one side. For example, in $r.B < s.E$, $s.E$ is an only bound for $r.B$. Therefore, Dignös et al. devised a range-merge join algorithm which performs the range join twice and returns a union of each result, which would be the following:

```
r.P OVERLAPS s.P ≡ r.B <= s.B < r.E OR s.B < r.B < s.E
```

Firstly, they perform range join twice because they use two disjoint range predicates. Thus, they produce results with no duplicates, and we can evaluate them independently. Secondly, we can assess range predicates efficiently via sorting as each time range is restricted by lower and upper bounds, e.g., in $r.B \leq s.B < r.E, s.B$ has a lower and an upper bound to restrict the search space. As a result, it provides a simpler way to join [DBG+22].

3 Preliminaries

The chapter illustrates concepts and definitions of join algorithms, which form the basis of this study.

We begin with the explanation of notations. This study defines that there is a tuple r with schema $R = (A_1, ..., A_m)$ and a tuple s with schema $S = (A'_1, ...A'_k)$, and $\circ$ refers to the concatenation of tuples. Thus, $r \circ s$ returns a tuple with schema $(A_1, ...A_m, A'_1, ..., A'_k)$ and the corresponding values. Also, in this study, **P** refers to the interval attribute; B indicates the start boundary, whilst E indicates end boundary. With these definitions, Ov(**P**$_1$, **P**$_2$) represents the overlap between two intervals **P**$_1$ and **P**$_2$ that evaluate to true, if and only if the intervals of **P**$_1$ and **P**$_2$ overlap in at least one common point. For two relations **r** and **s**, and a set of common attributes **C** = $\{C_1, ..., C_k\}$, we use **r.C = s.C** to represent the equality over all attributes in **C**. Furthermore, $\uplus$ symbolizes the union of two relations, which contain duplicates like SQL's `UNION ALL` [DBG+22]. Also, when comparing two attributes, we use the notation of r_i op s_i, where r_i (resp. s_i) is an attribute in a relation **r** (resp., **s**) and op is an inequality operator in the set $\{<, >, \leq, \geq\}$. We denote $L[i]$ the i-th element in array L, and $L[i, j]$ its sub-array from position i to position j [KLS+17]. Throughout this thesis, ω is used to refer to the end of the sequence in pseudocodes.

Accordingly, the study formally specifies interval notations and boundary types as follows, borrowing the definitions from Dignös et al.

Definition 1 *An interval attribute is specified as* ***P*** **=** (B, E)*, where* $B = (v_s, b_s)$ *and* $E = (v_e, b_e)$ *represent the start and end boundary, respectively. Given there are four boundary types '[', ']', '(', ')', one can describe that* $v_s, v_e \in \mathbb{R}, b_s \in$ *{'[', '('}, and* $b_e \in$ *{']', ')'}. Accordingly, a general interval includes a start boundary* $B = (v_s, b_e)$ *and an end boundary* $E = (v_e, b_e)$*, each consisting of a boundary value and a type. The type demonstrates whether the boundary value is included ('[', ']')or excluded ('(', ')') in the interval.*

Before exploring the definitions of join algorithms covered in this study, it starts with defining equi join and non-equi join, defined by Silberschatz et al. [SKS19].

Definition 2 *An* ***equi join*** *is a join in relational databases where tables are joined based on equality between specified columns. The equi join of* ***r*** *and* ***s*** *on a common attribute* $A_i = A'_j$ *is represented by:* ***r*** $\bowtie_{A_i = A'_j}$ ***s*** $= \{r \circ s | r \in$ ***r*** $\wedge\ s \in$ ***s*** $\wedge\ r.A_i = s.A'_j\}$*, where* $r.A_i$ *denotes the value of attribute* A_i *in tuple* r *and* $s.A'_j$ *denotes the value of attribute* A'_j *in tuple* s*.*

Definition 3 *A* ***non-equi join*** *joins two tables based on a condition that is not equality. Typical conditions are inequalities (e.g.,* $>, <, \neq$*). Given relations* ***r*** *and* ***s****, the non-equi join on a condition* θ *(where* θ *could be* $>, <, \neq$*) is denoted as:* ***r*** $\bowtie_{\theta(A_i, A'_j)}$ ***s*** $= \{r \circ s | r \in$ ***r*** $\wedge\ s \in$ ***s*** $\wedge\ r.A_i \theta s.A'_j\}$*. It means the non-equi join returns tuples where the values of* A_i *and* A'_j *meet the specified condition* θ*.*

As above-mentioned, there are two types of joins regarding equality conditions. Firstly, we describe one type of non-equi join, inequality join, based on Khayyat et al.

Definition 4 *An **inequality join** is an operation joining two relations, **r** and **s**, where an inequality rather than an equality defines the join condition. Given relations **r** with attributes $A_1, ..., A_m$ and **s** with attributes $A'_1, ..., A'_n$, an inequality join operates under a condition $C(r, s)$ such that $C(r, s) : f(A_i)\theta g(A'_j)$, where f and g are attribute functions on **r** and **s** respectively, and θ is an inequality operator (e.g., $<, >, \leq, \geq$). This operation results in a relation containing all pairs (r, s) from the Cartesian product **r** $\times$ **s** that satisfies the inequality condition.*

Next, we define range merge join and overlap merge join based on Dignös et al.

Definition 5 *A **range join** is a type of equi join where the join condition checks whether a value from one relation lies within the range of two values from another relation. Let **r** be a relation with schema R and **s** with schema S, and $C \in R \cap S$ be the attributes for joining. Also, we define attribute $B \in R$ and $E \in R$ represents a range in **r**, while $B' \in S$ be an attribute with the same domain as B and $E' \in S$ with the same domain as E. Moreover, $\prec^S \in \{<, \leq\}$ and $\prec^E \in \{<, \leq\}$. Given these, this study defines a range join between **r** and **s** as follows: **r** $\bowtie_{\mathbf{r.C = s.C} \wedge \mathbf{r}.B \prec^S \mathbf{s}.B' \prec^E \mathbf{r}.E}$ **s**. The operators $\prec^S$ and $\prec^E$ compare B' to B and E, respectively.*

Definition 6 *An **overlap join** is a type of join where the join condition checks whether the interval of one relation overlaps with another interval. Let **r** be a relation with schema R and **s** with S. Given $C \subseteq R \cap S$ is the attributes for joining, we define the overlap join between **r** and **s** as follows:* **r** $\bowtie_C^{Ov(\mathbf{r.P}, \mathbf{s.P})}$ **s** $= \{r \circ s | r \in \mathbf{r} \wedge s \in \mathbf{s} \wedge r.\mathbf{C} = s.\mathbf{C} \wedge Ov(r.\mathbf{P}, s.\mathbf{P})\}$

4 Baseline Join Algorithm

4.1 Overview

The present study employs brute-force nested loop join for the baseline join algorithm. The brute force join, also known as a nested-loop join, is a method in database systems for performing a join operation by exhaustively comparing each tuple in one relation with each tuple in another relational table. Given two relations **r** and **s** with $|\mathbf{r}| = m$ and $|\mathbf{s}| = n$ tuples, respectively, the brute force join examines each pair (r, s), where $r \in \mathbf{r}$ and $s \in \mathbf{s}$, to determine whether the join condition holds [SKS19]. Although the brute force join is inefficient for large dataset, it is straightforward to implement. Thus, it is generally used as a baseline comparison for other optimized join techniques, such as hash joins, and sort-merge joins, especially when indexing is unavailable or when datasets are small [SKS19]. Another advantage of brute-force join is that it can handle arbitrarily complex predicates. Therefore, the present study utilizes the brute force approach to analyze the efficiency of modified inequality join and interval-overlap merge join which will be discussed further in Chapter 7.

Example 1 *Consider there are two relational tables* ***r*** *and* ***s*** *which are identical, as shown in Table 4.1, and you intend to execute the following query Q_t:*

```
SELECT r.idx, s.idx
FROM r, s
WHERE r.idx > s.idx AND r.B < s.E AND r.E > s.E;
```

With the brute force join algorithm, the outer loop would first loop through each element in **r**. Then, the inner loop would loop through each element in **s** of each component in **r** for comparison. Consequently, the algorithm returns results after exhaustively examining each pair (r, s) to determine whether it meets the conditions, which would be the same as the `WHERE` clause in Q_t. The implementation of brute force join to execute Q_t is shown in Algorithm 1.

r

	P	
idx	[B	E]
1	1	3
2	2	4
3	4	6
4	0	2

s

	P	
idx	[B	E]
1	1	3
2	2	4
3	4	6
4	0	2

Table 4.1: Tabular representation: **r** and **s**

4.2 Implementation

Algorithm 1: BFJOIN

input : query Q with 2 join predicates $r.B < s.E$ and $r.E > s.B$, tables **r** and **s**, both sorted by idx, respectively
output: a list of tuple pairs (r_i, s_j)

```
1  initialize join_result as an empty list for tuple pairs
2  r ← first(r)
3  while r ≠ ω do
4      s ← first(s)
5      while s ≠ ω ∧ r.idx > s.idx do
6          if r.B < s.E ∧ r.E > s.B then
7              add tuples w.r.t. (r.idx, s.idx) to join_result
8          s ← next(s)
9      r ← next(r)
10 return join_result
```

Algorithm The brute force join algorithm (BFJOIN) is shown in Algorithm 1. It takes two sorted input relations, relation **r**, with idx indicating the index of each row, start point B and endpoint E, and relation **s** with idx indicating the index of each row, start point B and endpoint E as **r** does. Also, both B and E are inclusive as indicated in Table 4.1 Input relations **r** and **s** must be sorted according to index idx.

The algorithm first initializes an empty list join_result to append results (line 1) and reads the first tuple from **r** (line 2). Then, as long as the **r** has not yet been fully read, it scans the inner while loop (lines 3-9) and reads the first tuple from **s** to scan all tuples from **s** which have smaller idx compared to tuples from **r** (line 4). The inner while loop is executed as long as the **s** has not yet been fully read and $r.idx$ is greater than $s.idx$ (lines 5-8). Inside of the while loop, it $r.B$ is smaller than $r.E$ and $r.E$ is greater than $s.B$, it adds tuples with regard to $(r.idx, s.idx)$ to join_result (lines 6-7). It keeps reads the next tuple in **s** until it breaks the condition in the innner while loop (line 8). Once it is done with reading the inner while loop, the outer while loop moves to the next tuple in **r** (line 9).

Complexity We can analyze that this approach has a time complexity of $O(m \cdot n)$. At first, the outer loop of Algorithm 1 iterates over each tuple in table **r**. The inner loop iterates over each tuple in table **s** for each tuple in **r** until $r.idx > s.idx$ is no longer valid and it reaches the end of **s**. Therefore, if |**r**| is m and |**s**| is n, the time complexity is $O(m \cdot n)$.

In terms of space complexity, we should consider that the algorithm stores the resulting tuple pairs in a list. In the worst case, if every pair of tuples from **r** and **s** satisfies the join condition, the space complexity would be $O(m \cdot n)$. However, if the number of resulting pairs is less than $m \cdot n$, the space complexity would be proportional to the number of resulting pairs.

5 Inequality Join Based on Sort-Merge with Bloom Filters

5.1 Inequality Join Algorithm

As defined in Definition 4 in Chapter 3, an inequality join is a join defined by inequality predicates. Let's assume we would like to execute the following query Q_s:

```
SELECT r.idx, s.idx
FROM r, s
WHERE r.idx > s.idx AND r.B < s.E;
```

The traditional DBMS would first create a Cartesian product **r** $\times$ **s** and go through selection predicates $(r.idx > s.idx) \wedge (r.B < s.E)$. However, this approach is effective when datasets are small enough to be stored in memory and queries with selection conditions or highly selective equi joins. Therefore, these conditions are unsuitable for joining two large relational tables or for queries with low selectivity predicates, such as those based on gender or region [KLS$^+$17].

In such cases, sort-merge joins and inverval-based indexing are the most preferred methods to solve this issue. A major advantage of sort-merge join is that it minimizes the search space. It sorts data by joining attributes and merging them. Despite this, it still has quadratic complexity regarding inequality joins. Another preferred method, interval-based indexing, further narrows the search space by utilizing bitmap interval indexing. Nevertheless, these indices demand significant space and time for building memory and indexes [KLS$^+$17].

Therefore, Khayyat et al. applied bit-map arrays and positional permutation arrays, and named it as IEJOIN.

Example 2 *Given the relational tables* ***r*** *and* ***s*** *in Table 4.1, consider that we would like to execute Q_s. IEJOIN takes the steps as follows (also, see Figure 5.1):*

[Initialization] We sort **r** tuples on idx in descending order in the array $L_1 :< 4, 3, 2, 1 >$, an array sorted by values and storing indexes of the values. Like attribute idx, we also sort attribute B in descending order into an array $L_2 :< 3, 2, 1, 4 >$. Similarly, sort **s** tuples on idx in descending order in the array L'_1 and on E in descending order in the array L'_2. To join two different attributes, we calculate a relative position of a sorted array L_2 (resp. L'_2) regarding L_1 (resp. L'_1). It yields two permutation arrays $P :< 2, 3, 4, 1 >$ and $P' :< 2, 3, 4, 1 >$. $P[2] = 3$ means that the second element of L_2 ($i.e.$, 2) corresponds to position 3 in L_1. Then, it computes an offset array O_1 (resp. O_2) of L_1 relative to L'_1 (resp. L_2 relative to L'_2) by comparing

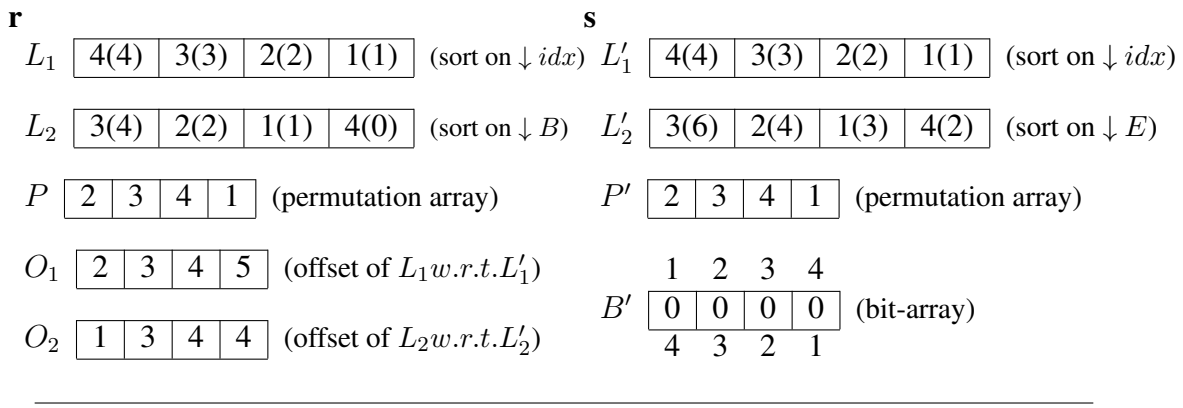


Figure 5.1: IEJOIN process for query Q_s

values between them concerning operators between two attributes. For instance, $O_1[1] = 2$ means that $L'_1[2]$ is the first true element, making $L_1[1] = 4$ greater than its value. $O_2[1] = 1$ implies that the first element of L'_2 is the last true value, smaller than $L_2[1] = 4$. Finally, we create a bit-array B' of length n and set all bits to 0. [Visit tuples $w.r.t.$ L_2] It starts to scan the permutation array P' and operate on the bit-array.

(a) *Visit L_2[1] for idx 3* First, visit the first element in L_2 (3). Then, it visits all tuples in L'_2 whose E values are greater than $3[B]$. This is achieved by scanning P' from the initial position up to the first position specified in O_2. Hence, it sets $B'[2]$ as 1. Afterwards, it uses the relative position of $3[idx]$ at L'_1 to populate all join results. This is done by going to $B'[3]$ and scanning all bits in positions higher than 3. As all $B'[k] = 0$ for $k \geq 3$, no tuple satisfies the join condition of $Q_s w.r.t.$ idx 3. Finish this visit by setting B' of position $P'[1]$, which indicates that the first element of L'_2 has been visited.

(b) *Visit L_2[2] for idx 2* Visit the second item 2 of L_2. Then, it visits all tuples in L'_2 where $2[B]$ is no larger than $2[E]$. As the relative position of $2[idx]$ is 3, it scans B' from $O_1[3]$ ($i.e.$, 4) to n ($i.e.$, 4). As $B'[4] = 1$, it outputs $(L_2[2], L'_1[4])$, which is (2, 1). Finish this visit by setting $B'[k] = 1$, where k is no larger than $P'[3]$ (i.e., $O_2[2] = 3$) and greater than $P'[1]$ to indicate that the second element of L'_2 has been visited.

(c) *Visit L_2[3] for idx 1* This visit corresponds to tuple 1. Given that the relative position of $1[idx]$ is 5 ($i.e.$, $1[idx] = 4, O_1[4] = 5$), it scans all bits in positions higher than 5. However, as the size of B' is 4, it returns no results. Set the elements of $B'[k] = 1$, where k is no larger than $P'[4] (i.e., O_2[3] = 4)$.

(d) *Visit L_2[2] for idx 4* In this step, the relative position of $4[idx]$ is 2. Hence, it yields results if $B'[k] = 1$, where $k \geq 2$ regarding $L_2[4]$. It, thus, outputs (4,3), (4,2), and (4,1).

Then, query Q_s returns a set of pairs $\{(2, 1), (4, 1), (4, 2), (4, 3)\}$.

5.2 Algorithms

This section describes a permutation array (Sect 5.2.1) and offset arrays (Sect 5.2.2) to establish a basis for inequality joins. Then, the study reviews IEJOIN with two inequality predicates, using operators like $\{<, >, \leq, \geq\}$, between two relations (Sect 5.2.3). Finally, it analyzes another inequality join, IESELFJOIN, regarding two inequalities between attributes of the same relational table (Sect 5.2.4).

5.2.1 Permutation Array

Algorithm 2: PERMUTATIONARRAY

input : arrays L_1 and L_2 of size m
output: a permutation array P

```
1 let L2Index be the array of indexes of L2
2 initialize index_map as an empty dictionary
3 initialize P as an empty list
4 for (i ← 1 to m) do
5 |  add L1[i] as a key and i as a value to index_map
6 for (j ← 1 to m) do
7 |  add index_map[L2Index[j]] to P
8 return P
```

Algorithm A permutation array (see Algorithm 2) is used in both IEJOIN (see Algorithm 5 lines 7-8) and IESELFJOIN (see Algorithm 6 line 6). The permutation array tells the corresponding position in L_2, given the i-th element in L_1. Therefore, the algorithm takes arrays L_1 and L_2 of size m. The algorithm first reads the indexes of L_2 (line 1) into L_2Index. Then, it initializes index_map as an empty dictionary and P as an empty list (lines 2-3). It scans L_1 and stores the value of $L_1[i]$ as a key and i as a value to index_map (lines 4-5). Finally, the second for-loop produces a permutation array P by accessing the relative position of L_2 regarding L_1 through values of index_map (lines 6-8).

Complexity Provided that the size of L_1 and L_2 is m, it gives a complexity of $O(m)$. The initialization of index_map and P is $O(1)$. For both loops, adding a key-value pair to a dictionary or retrieving a value from a dictionary and appending it to a list takes $O(1)$. However, as both loops scan through an array of size m, their total time complexity is $O(m)$. Therefore, the overall time complexity of Algorithm 2 is $O(m)$. The space complexity of a permutation array is $O(n)$ since it stores n elements.

5.2.2 Offset Array

The offset array is one of the main parts of IEJOIN (see Algorithm 5 lines 9-10). An offset array is a sort-merge-based array computed by a linear scan of a sorted array L_1 (resp. L_2) from **r** and another sorted array L'_1 (resp. L'_2) from **s** [KLS+17]. There are two algorithms for offset arrays, depending on which part of the selection predicates they address.

Algorithm 3: OFFSETARRAY1

input : arrays L_1 of size m and L'_1 of size n, and the inequality join condition $\mathsf{op_1}$
output: an offset array O_1

```
1 initialize O_1 as an empty list for relative positions of elements in L_1 w.r.t. L'_1
2 for (i ← 1 to m) do
3     for (j ← 1 to n) do
4         if (L_1[i] op_1 L'_1[j]) then
5             add j to O_1
6         if (L_1[i] op_1 L'_1[j] = False) ∧ (j = n) then
7             add j + 1 to O_1
8 return O_1
```

Algorithm 4: OFFSETARRAY2

input : arrays L_2 of size m and L'_2 of size n, and the inequality join condition $\mathsf{op_2}$
output: an offset array O_2

```
1 initialize O_2 as an empty list for relative positions of elements in L_2 w.r.t. L'_2
2 for (i ← 1 to m) do
3     for (j ← 1 to n) do
4         if (L_2[i] op_2 L'_2[j] = False) then
5             add j − 1 to O_2
6         if (L_2[i] op_2 L'_2[j]) ∧ (j = n) then
7             add j to O_2
8 return O_2
```

Algorithm The algorithm 3 tackles the inequality relation between L_1 and L'_1 based on $\mathsf{op_1}$ to output an offset array O_1. It takes L_1 from **r**, L'_1 from **s**, and an inequality join condition $\mathsf{op_1}$ as inputs. Firstly, it initializes O_1 as an empty list (line 1). Then, it starts scanning all the elements in L_1 with the outer loop to compare it with each value in L'_1 (lines 2-7). In the inner for loop, it scans each element in L'_1 to check whether the condition is satisfied (lines 3-7). The if conditions compare the value of $L_1[i]$ to $L'_1[j]$ according to $\mathsf{op_1}$ (lines 4-7). If $L_1[i]\mathsf{op_1}L'_1[j]$ returns the first true, then it records the relative position j from L_1 in L'_1 (lines

4-5). However, if the condition is not met and j has reached the final position n, it adds the relative position $j+1$ to O_1 as we assume it would return true at the $n+1$ position (lines 6-7). The algorithm returns the final result O_1 in line 8.

The algorithm 4 takes arrays L_2 from **r** and L'_2 from **s** and an inequality join condition $\mathsf{op_2}$ as inputs to return an offset array O_2. It initializes O_2 as an empty list (line 1). Then, it iterates through each element in L_2 of length m to compare with values in L'_2 respectively (lines 2-3). In the inner for loop, it scans each element in L'_2 to check whether the condition is satisfied (lines 3-7). The if conditions compare the value of $L_2[i]$ to $L'_2[j]$ depending on $\mathsf{op_2}$ (lines 4-7). If the comparison between $L_2[i]$ $\mathsf{op_2}$ $L'_2[j]$ is false, it adds $j-1$ to O_2 (lines 4-5). However, if the comparison result is true and reaches the last element of L'_2, we add j to O_2 (lines 6-7). The algorithm returns the final result O_2 in line 8.

Complexity Given that m is the size of L_1, and n is the size of L'_1, we can analyze the complexity of Algorithm 3 as follows. As the outer loop iterates over L_1, it has the complexity of $O(m)$ (lines 2-7). Meanwhile, the inner loop iterates over L'_1 for each element of L_1 (lines 3-7). For each pair (i, j), it checks two conditions and may add an element to O_1. As the complexity per iteration for the checks and additions is $O(1)$, the total complexity for the inner loop is $O(n)$. Since the inner loop runs for every iteration of the outer loop, the total complexity of the nested loops is $O(m) \times O(n) = O(m \cdot n)$. Thus, the overall time complexity is $O(m \cdot n)$.

We can take a similar approach to analyze the time complexity of Algorithm 4. The outer loop's time complexity is $O(m)$ as it iterates over L_2 of size m (lines 2-7). The inner loop iterates over L'_2 of size n for each element of L_2 (lines 3-7). For each pair (i, j), it checks two conditions and may add an element to O_2. As the complexity per iteration for the checks and additions is $O(1)$, the total complexity for the inner loop is $O(n)$. Since the inner loop runs for every iteration of the outer loop, the total complexity of the nested loops is $O(m) \times O(n) = O(m \cdot n)$. Thus, the overall time complexity of Algorithm 4 is also $O(m \cdot n)$.

The space complexity of an offset array is $O(n)$ as it stores n elements.

5.2.3 IEJoin

Algorithm IEJOIN, is shown in Algorithm 5. It takes a query Q with two inequality join conditions as input and returns a set of result pairs. It first sorts the attribute values to be joined either in ascending or descending order depending on an operator (lines 3-6). Afterwards, it computes a permutation array (lines 7–8) and two offset arrays (lines 9–10). We deferred details of a permutation array in Sect. 5.2.1 and offset arrays in Sect. 5.2.2. Each element of an offset records the relative position from L_1 (resp. L_2) in L'_1 (resp. L'_2). The offset array is computed by a linear scan of both sorted arrays (e.g., L_1 and L'_1). The algorithm also intializes a bit-array (line 11) and a result set (line 12). Then, it visits the values in L_2 in the sequential order by scanning the permutation array from left to right (lines 13-20). For each tuple visited in L_2, it first sets all bits for those s in **s** whose Y' values are smaller than the Y value of the current tuple in **r** (lines 14-16), i.e., those tuples in **s** that satisfy the second join condition. It then uses the other offset array to find those tuples in **s** that also satisfy the first join condition

Algorithm 5: IEJOIN

input : query Q with 2 join predicates $r.X$ op_1 $s.X'$ and $r.Y$ op_2 $s.Y'$, tables **r**, **s** of sizes m and n resp.
output: a list of tuple pairs (r_i, s_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 ($\mathsf{op}_1 \in \{>, \geq\}$) **then** sort L_1, L'_1 in descending order
else if ($\mathsf{op}_1 \in \{<, \leq\}$) **then** sort L_1, L'_1 in ascending order
if ($\mathsf{op}_2 \in \{>, \geq\}$) **then** sort L_2, L'_2 in ascending order
else if ($\mathsf{op}_2 \in \{<, \leq\}$) **then** 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
for ($i \leftarrow 1$ **to** m) **do**
    $\mathsf{off}_2 \leftarrow O_2[i]$
    for ($j \leftarrow 1$ **to** $\mathsf{min}(\mathsf{off}_2, \mathsf{size}(L'_2))$) **do**
        $B'[P'[j]] \leftarrow 1$
    $\mathsf{off}_1 \leftarrow O_1[P[i]]$
    for ($k \leftarrow \mathsf{off}_1$ to n) **do**
        if $B'[k] = 1$ **then**
            add tuples *w.r.t.* $(L_2[i], L'_1[k])$ to join_result
return join_result

(lines 17-20). It finally returns all join results (line 21).

Complexity Let us now turn to the time complexity of IEJOIN. Firstly, it extracts arrays L_1, L_2, L'_1, L'_2, and this takes $O(m+n)$ (lines 1-2). Then, it sorts arrays, which in turn returns the time complexity of $O(m \log m)$ for L_1 and L_2, and $O(n \log n)$ for L'_1 and L'_2. This yields the total cost of $O(m \log m + n \log n)$. As described in Sect. 5.2.1, creating P (resp. P') has the time complexity of $O(m)$ (resp. $O(n)$). And both O_1 and O_2 has the time complexity of $O(m \cdot n)$, as described in Sect. 4. Hence, the overall computation cost of permutation arrays and offset arrays is $O(m \cdot n)$. The outer loop has the time complexity of $O(m)$. However, given the first inner loop iterates up to (min(off_2, size(L'_2))), its worst case complexity is $O(n)$. Similarly, the second inner loop has the time complexity of $O(n)$ in the worst case, since it scans from off_1 to n. Hence, for both loops, it gives the complexity $O(m \cdot n)$. Among these terms, the dominant one is $O(m \cdot n)$, thus, the overall time complexity is $O(m \cdot n)$.

With respect to the space complexity, we consider all the arrays involved. The input arrays L_1, L_2, L'_1 and L'_2 store m or n elements, contributing $O(m+n)$ space. The permutation arrays P and P', and the offset arrays O_1 and O_2 also require $O(m+n)$ space since they store m or n elements. The bitmap array B' has n bits, contributing $O(n)$ space. The list join_result stores the output tuple pairs. In the worst case, this could store up to $O(m \cdot n)$ pairs. Therefore, the space complexity of the IEJOIN is dominated by the space required for the join result list, resulting in the space complexity of $O(m \cdot n)$.

5.2.4 IESelfJoin

Algorithm IESELFJOIN is shown in Algorithm 6. Unlike IEJOIN, it compares inequalities between two different attributes in the same relational table. Therefore, it takes a self-join inequality query Q and returns a set of result pairs.

Firstly, the algorithm sorts the two lists of attributes (lines 2-5), computes a permutation array (line 6) and builds a bit array (line 7). It also initializes a result set (line 8) and specifies an offset variable to determine whether inequality operators have an equality operator $=$ (lines 9-10). After initializations, it visits the values in L_2 by sequentially scanning the permutation array from left to right (lines 11-16). Each tuple visited in L_2 needs to find all tuples whose X values satisfy the join condition. The joining is performed by first locating its corresponding position in L_1 via exploring the permutation array (line 12) and marking it in the bit array (line 13). Since the bit-array and L_1 have a one-to-one positional correspondence, the tuples on the right of pos will satisfy the join condition on X (lines 14-16), and these tuples will also satisfy the join condition on Y if the loop visited them before (line 15). Such tuples will be joined with the visited tuple (line 16). It finally returns all join results (line 17).

Complexity In IESELFJOIN, we first consider sorting and computing permutation arrays (lines 2-8). This step has a time complexity of $O(n \log n)$, as sorting takes $O(n \log n)$ and computing a permutation array takes $O(n)$. Scanning the permutation array and the bit-array for each visited tuple takes $O(n^2)$ time (lines 11-16). Therefore, the overall time complexity of IESELFJOIN is $O(n^2)$.

Algorithm 6: IESELFJOIN

input : query Q with 2 join predicates $t_1.X$ op_1 $t_2.X$ and $t_1.Y$ op_2 $t_2.Y$, table T of size n
output: a list of tuple pairs (t_i, t_j)

```
1  let L1 (resp. L2) be the array of column X (resp. Y)
2  if (op1 ∈ {>, ≥}) then sort L1 in ascending order
3  else if (op1 ∈ {<, ≤}) then sort L1 in descending order
4  if (op2 ∈ {>, ≥}) then sort L2 in descending order
5  else if (op2 ∈ {<, ≤}) then sort L1 in ascending order
6  compute the permutation array P of L2 w.r.t. L1
7  initialize bit-array B (|B| = n), and set all bits to 0
8  initialize join_result as an empty list for tuple pairs
9  if (op1 ∈ {≤ , ≥} and op2 ∈ {≤ , ≥}) then eqOff = 0
10 else eqOff = 1;
11 for (i ← 1 to n) do
12     pos ← P[i]
13     B[pos] ← 1
14     for (j ← pos + eqOff to n) do
15         if B[j] = 1 then
16             add tuples w.r.t. (L1[j], L1[P[i]])
17 return join_result
```

With respect to the space complexity, we consider the following components. The arrays L_1 and L_2 and the permutation array P store n elements and thus require $O(n)$ for space. Similarly, the bit map array B has n bits, contributing $O(n)$ space. The final results store the output tuple pairs. In the worst case, it could store up to $O(n^2)$ pairs, contributing $O(n^2)$ space. Taking these into consideration, the overall space complexity of the IESELFJOIN is $O(n^2)$.

5.3 Modifications and Implementations

Algorithm 7: MODIFIEDIEJOIN

input : query Q with 3 join predicates $r.idx > s.idx$, $r.B < s.E$ and $r.E > s.B$, tables **r**, **s** of sizes m and n resp.
output: a list of tuple pairs (r_i, s_j)

```
1  let L1 (resp. L2) be the array of B (resp. E) in r
2  let L1' (resp. L2') be the array of E (resp. B) in s
3  let L3 (resp. L3') be the array of positions of L2 (resp. L1')
4  sort L1, L1' in ascending order
5  sort L2, L2' in ascending order
6  compute the permutation array P of L2 w.r.t. L1
7  compute the permutation array P' of L2' w.r.t. L1'
8  compute the offset array O1 of L1 w.r.t. L1'
9  compute the offset array O2 of L2 w.r.t. L2'
10 initialize bit-array B' (|B'| = n), and set all bits to 0
11 initialize join_result as an empty list for tuple pairs
12 for (i ← 1 to m) do
13     off2 ← O2[i]
14     for (j ← 1 to min(off2, size(L2'))) do
15         B'[P'[j]] ← 1
16     off1 ← O1[P[i]]
17     for (k ← off1 to n) do
18         if (B'[k] = 1) ∧ (L3[i] > L3'[k]) then
19             add tuples w.r.t. (L2[i], L1'[k]) to join_result
20 return join_result
```

5.3.1 Challenges

IEJOIN achieves its efficiency by various methods. An offset array is one of them as it decides which positions to read in the bit-map array. According to Khayyat et al., an offset array records the relative position from L_1 (resp. L_2) in L_1' (resp. L_2') by linearly scanning both

sorted arrays (lines 9-10). In addition, the original IEJOIN sets an offset variable to distinguish between the inequality operators with or without equality conditions (lines 13–14). Moreover, the computation of offset array is supposed to have loglinear complexity.

However, a major problem occurs with this clue. It is obscure whether one should construct an offset array by sorting unique elements from both L_1 (resp. L_2) and L'_1 (resp. L'_2) and get the indexes of all the elements from L_1 (resp. L_2) to construct an offset array. Another idea could be placing one element from L_1 (resp. L_2) in L'_1 (resp. L'_2), checking its relative position, and then getting an index. For instance, given L_2 and L'_2 in Figure 5.2, the first assumption would return $O_2 =< 2, 4, 5, 6 >$, while the second assumption would return $O_2 =< 2, 4, 5, 5 >$. However, all of these ideas are misleading as they cannot properly handle what should be activated in the bit array, and consequently, lead to wrong results. Even the offset values cannot prevent returning an incorrect result as the issue is innate.

Therefore, we suggest two different approaches to find offset arrays for L_1 and L'_1, which are compared by op_1 (see Algorithm 3) and L_2 and L'_2 compared by op_2 (see Algorithm 4) to handle the linear scan properly, although the time complexity is different from the original algorithm. Algorithm 3 assumes that when comparing a value of L_1 to the values of L'_1 based on op_1, $n + 1$th position is always true. On the other hand, Algorithm 4 compares a value of L_2 to the values of L'_2 based on op_2, 0th position is always true. With these assumptions, it does not require offset values, as it already handled the distinction between the inequality operators with or without equality conditions. Technical details are already handled in Sect. 5.2.2.

Another challenge lies in producing the results. The original IEJOIN algorithm adds $L'_2[k]$ to join_result (Algorithm 5 line 20). However, this also misleads to a wrong result considering the fact that bit-map array is actually dealing with the positional indexes of L'_1. Thus, this study modified it to return $(L_2[i], L'_1[k])$ (Algorithm 5 line 20).

5.3.2 Implementation

The adaptations against the challenges described in Sect 5.3.1 are reflected on the newly modified IEJOIN in Algorithm 7.

Example 3 *Consider two relational tables* ***r*** *and* ***s*** *in Table 4.1; let us assume we would like to execute a query Q_t according to Algorithm 7.*

The MODIFIEDJOIN firstly interprets Q_t as the execution of a query Q_p as follows according to IEJOIN with an additional condition $r.idx > s.idx$.

```
SELECT r.idx, s.idx
FROM r, s
WHERE r.B < s.E AND r.E > s.B;
```

In initialization, Algorithm 7 sorts **r** tuples on B and E respectively in ascending order in the array $L_1 :< 4, 1, 2, 3 >$ and $L_2 :< 4, 1, 2, 3 >$. Similarly, sort **s** tuples on E and B respectively in ascending order in the array L'_1 and L'_2. The unique part of MODIFIEDIEJOIN is to construct L_3 (resp. L'_3) to store indexes of L_2 (resp. L'_1) for the idx comparison. To join two different

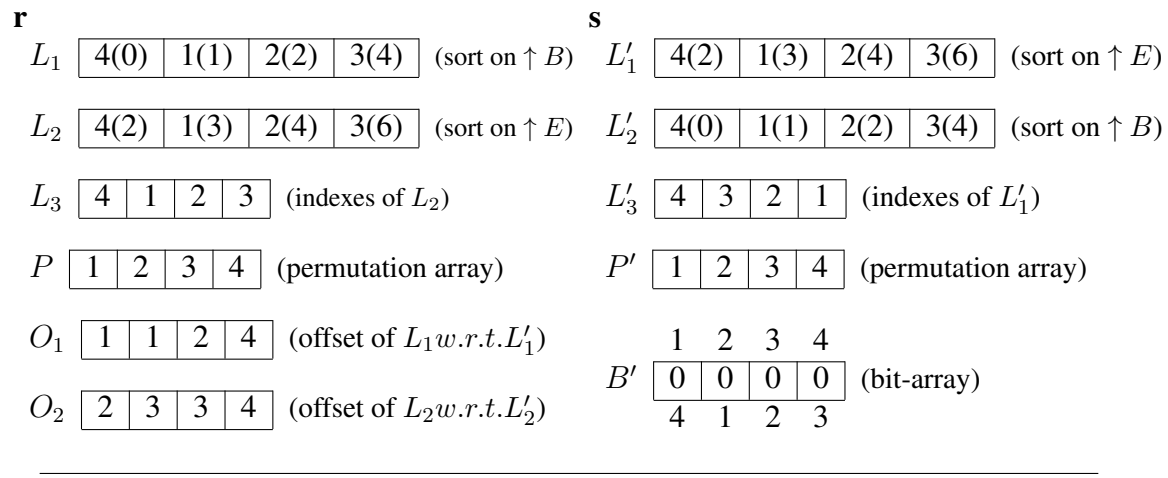


Figure 5.2: MODIFIEDIEJOIN process for query Q_t

attributes, we calculate a relative position of a sorted array L_2 (resp. L'_2) regarding L_1 (resp. L'_1): $P :< 1, 2, 3, 4 >$ and $P' :< 1, 2, 3, 4 >$. Then, it computes an offset array O_1 (resp. O_2) of L_1 relative to L'_1 (resp. L_2 relative to L'_2): $O_1 :< 1, 1, 2, 4 >, O_2 :< 2, 3, 3, 4 >$. Finally, we create a bit-array B' of length n and set all bits to 0.

After the initialization, it visits tuples concerning L_2. It scans the permutation array P' and operates on the bit-array like IEJOIN. For example, when it visits the first element of L_2 (4), it checks all tuples in L'_2 whose B values are less than $4[E]$. Afterwards, it uses the relative position of $4[E]$ at L'_1 to populate all join results. This is done by going to $B'[4]$ and scanning all bits in positions higher than 4. The main unique part of MODIFIEDIEJOIN is that when populating results, it checks the condition whether $r.idx > s.idx$. For this, the algorithm checks whether the first item of L_3 is greater than the k-th position L'_3. The exact process applies to other elements of L_2. Then, the query Q_t returns a set of pairs $\{(4, 1), (2, 1)\}$.

Algorithm The MODIFIEDIEJOIN, is shown in Algorithm 7. It takes a query Q with three join predicates as input and returns a set of result pairs. Similar to IEJOIN, it starts with setting up L_1 (resp. L'_1) and L_2 (resp. L'_2) (lines 1-2). The difference is that it creates L_3 to store indexes of L_2 and L'_3 for the indexes of L'_1 for comparison and output (line 3). The arrays, L_1, L'_1, L_2, L'_2, are sorted in ascending order as the algorithm assumes op_1 is always < since $r.B < s.E$, and op_2 is always > as $r.E > s.B$ (lines 4-5). Afterwards, it computes a permutation array P and P' (lines 6-7) and two offset arrays (lines 8-9). It also intializes a bit-array (line 10) and a result set (line 11). Then, it visits the values in L_2 in the sequential order by scanning the permutation array (lines 13-20). For each tuple visited in L_2, it first sets all bits for those s in **s** whose B values are smaller than the E value of the current tuple in **r** (lines 13-15), i.e., those tuples in **s** that satisfy the second join condition. It then uses the other offset array to find those tuples in **s** that also satisfy the first join condition $r.B < s.E$ (lines 16-19). However, we added an additional condition $r.idx > s.idx$ by comparing L_3 and L'_3 to return results (lines 18-19). It finally returns all join results (line 20).

Complexity Although MODIFIEDIEJOIN added additional arrays L_3 and L'_3 for additional condition, its time complexity still remains the same as IEJOIN. MODIFIEDIEJOIN sorts arrays and computes their permutation arrays, where m and n are the sizes of the two input relations (lines 3-8), its time complexity is $O(m)$ for P and $O(n)$ for P'. It also computes the offset arrays in quadratic time using sort-merge (lines 9-10), and the same applies to the outer loop, which in turn gives the complexity of $O(m \cdot n)$. Therefore, considering the dominant term, the total time complexity is $O(m \cdot n)$.

Regarding the space complexity, it is also similar with IEJOIN. In MODIFIEDIEJOIN, it requires additional memory as it has L_3 and L'_3 which require $O(m)$ and $O(n)$ space. However, the dominant term is still the final results like IEJOIN. In the worst case, join_result could store the mixture of all the tuple pairs from both relations $m \times n$, requiring $O(m \cdot n)$ space. Consequently, the overall space complexity is $O(m \cdot n)$.

6 Range Joins for Interval Overlap Merge Join

6.1 Range Merge Join and Overlap Merge Join

In Sect. 2, the study stated that Dignös et al. developed a new approach for overlap predicates. They interpreted $Ov(r.\mathbf{P}, s.\mathbf{P}) \equiv r.B \leq s.E \wedge s.B \leq r.E$ as a disjunction of two terms, allowing them to be computed separately without generating duplicates. The overlaps predicate, designed by them, is as follows: $Ov(r.\mathbf{P}, s.\mathbf{P}) \equiv (r.B \leq s.B < r.E) \vee (s.B < r.B < s.E)$. They also named it as RMJ(range merge join), and the union of two RMJs as OMJ (overlap merge join). Although Dignös et al. define $Ov(r.\mathbf{P}, s.\mathbf{P})$ as $(r.B \leq s.B \leq r.E) \vee (s.B < r.B \leq s.E)$ as well, to reduce the confusion, we only consider the version defined in Chapter 2 and definition mentioned above.

Example 4 *Consider a relation **t** and **u** over an interval **P** in Table 6.1. Assume we would like to execute the following query Q_w:*

```
SELECT t.idx, u.idx
FROM t, u
WHERE t.idx = u.idx AND t.P OVERLAPS u.P;
```

t

	P	
idx	[B	E)
2	1	6
1	2	5
2	4	8
1	9	11

u

	P	
idx	[B	E)
1	1	11
2	1	6
2	6	10

Table 6.1: Tabular representation: **t** and **u**

Given relation **t** with schema T and **u** with schema U, we start to sort the relations by an equality attribute **C** and the start point B. Then, we compute range merge join twice and returns a union as follows:

1. Sort **t** and **u** by (**C**, B)
2. Compute RMJ(**t**, **u**, **C**, $B, \leq, B, <, E, T \circ U$) $\uplus$ RMJ(**u**, **t**, **C**, $B, <, B, <, E, T \circ U$)

In this example, we define an equality attribute as idx. Hence, we start with sorting both **t** and **u** by (idx, B). Next, we combine the result of range merge joins: RMJ(**t**, **u**, $\{idx\}$, $B, \leq, B, <, E, T \circ U$) and RMJ(**u**, **t**, $\{idx\}$, $B, <, B, <, E, T \circ U$), where both T and U are (idx, B, E), respectively.

t

	idx	P $[B$	$E)$
t_1	1	2	5
t_2	1	9	11
t_3	2	1	6
t_4	2	4	8

u

	idx	P $[B$	$E)$	t_1	t_2	t_3	t_4
u_1	1	1	11	-			
u_2	2	1	6	x	x	✓	-
u_3	2	6	10			$\perp$	✓
							$\perp$

(a) RMJ(**t**,**u**,idx,$B, \leq, B, <, E, T \circ U$)

u

	idx	P $[B$	$E)$
u_1	1	1	11
u_2	2	1	6
u_3	2	6	10

t

	idx	P $[B$	$E)$	u_1	u_2	u_3
t_1	1	2	5	✓	-	
t_2	1	9	11	✓	-	
t_3	2	1	6	$\perp$	-	-
t_4	2	4	8		✓	
					$\perp$	

(b) RMJ(**s**,**r**,idx,$B, <, B, <, E, R \circ S$)

Figure 6.1: Range merge joins for our running example: ✓indicates a match, $\perp$ the end of matches, - an inner skip, and x an outer skip

Figure 6.1a provides an overview of the first RMJ. For the first tuple t_1 in sorted order from **t**, the relation **u** is scanned from the beginning in sorted order. Tuples t_1 and u_1 share the same idx value. However, since the start boundary of t_1 is not less than or equal to the start boundary of u_1, u_1 is skipped (indicated by "-"). Next, tuple u_2 is checked. This time, the idx value of t_1 is less than that of u_2, so t_1 is skipped (indicated by "x"), and tuple t_2 is fetched. t_2 is also skipped for the same reason, and tuple t_3 is retrieved and compared with u_2. They have the same idx, and the B value of t_3 is less than or equal to the B value of u_2. Tuple u_2 is marked, and since the start boundary u_2 is less than the end boundary of t_3, an output (t_3, u_2) is produced (indicated by "✓"), and the inner relation advances to u_3. Tuple u_3 has the same idx, but its B value does not fall within t_3's interval. Thus, the next outer tuple t_4 is fetched (indicated by "$\perp$"), and the inner relation is restored to u_2. Tuples t_4 and u_2 have the same idx, but the B value of u_2 is not greater than or equal to the B of t_4, so it skips u_2. Next, u_3 is fetched. As t_4 and u_3 have the same idx and the start boundary of t_4 is less than or equal to the start boundary of u_3, it marks u_3. The $u_3.B$ is less than the end boundary of t_4. Therefore, it produces an output (t_4, u_3). Then, it reaches the end of the inner relation. The outer relation advances, and the inner relation is restored to u_3. Since the outer relation's end is reached, the algorithm terminates. Overall, this yields two result tuples $\{(t_3, u_2), (t_4, u_3)\}$.

The processing of the second range merge join RMJ(**u**, **t**, $\{idx\}, B, <, B, <, E, T \circ U$) is summarized in Figure 6.1b. Aside from swapping the positions of the two input relations, it is similar to the first RMJ operation. The critical difference is using the comparison operator $<$ for the start time instead of $\leq$. For example, when processing tuple u_2, the inner tuple t_3 is skipped. This is to prevent the generation of a duplicate result for (t_3, u_2), which was already generated in the previous RMJ. As a result, the second RMJ returns $\{(t_1, u_1), (t_2, u_1), (t_4, u_2)\}$.

Therefore, with this example, we get the result tuples $\{(t_3, u_2), (t_4, u_3), (t_1, u_1), (t_2, u_1), (t_4, u_2)\}$.

6.2 Algorithms

This section illustrates the algorithmic basis of range merge join and overlap merge join.

6.2.1 Range Merge Join

Algorithm 8: RMJ (**r, s, C**, $B, \prec^S, X, \prec^E, E, O$)

input : Relations **r** sorted by (**C**, B) and **s** sorted by (**C**, X), equality attributes **C**, start point B in **r**, comparison operator $\prec^S \in \{<, \leq\}$ for B and X, attribute X in **s**, comparison operator $\prec^E \in \{<, \leq\}$ for $X$and E, end ponit E in **r**, output schema O

output: Result of $\mathbf{r} \bowtie_{\mathbf{r.C = s.C} \wedge \mathbf{r}.B \prec^S \mathbf{s}.X \prec^E \mathbf{r}.E} \mathbf{s}$

```
1  r ← first(r)
2  s ← first(s)
3  while r ≠ ω ∧ s ≠ ω do
4      if r.C < s.C then
5          r ← next(r)
6      else if r.C = s.C ∧ r.B ≺^S s.X then
7          marked ← s
8          while s ≠ ω ∧ r.C = s.C ∧ s.X ≺^E r.E do
9              output r and s according to schema O
10             s ← next(s)
11         r ← next(r)
12         s ← marked
13     else
14         s ← next(s)
```

Algorithm RMJ is shown in Algorithm 8. The algorithm operates on two sorted input relations **r** with B and E, and **s** with an attribute X. It also considers equality attributes C and two

comparison operators $\prec^S, \prec^E$ that define whether the range join includes or excludes the start and/or end points (as per Definition 1). RMJ takes relation **r** sorted by the equality attributes C and the start point B and **s** sorted by C and X.

The algorithm begins by reading the first tuple from each relation (lines 1-2). It continues processing until both relations are fully read, executing one of the if conditions (lines 3-14). The first if condition (skip outer) is triggered if the equality attributes C in r are smaller than those in s (lines 4-5). In such cases, the current tuple in r is skipped (line 5). If **C** is empty, this step is never executed, as $r.\mathbf{C} = s.\mathbf{C}$ is assumed to be true. The second if condition (join match) occurs when the equality attributes C match and the start time B of r is less than or equal to the value of X in s (lines 6-12). This may generate result tuples (line 9). The position of the current tuple in s is marked, as it could match subsequent tuples in r (line 7). A while-loop then produces results for all subsequent tuples in s that satisfy the join condition with r (lines 8-10). Once all matches for the current tuple in r are processed, the next tuple in r is retrieved, and the position in s is reset to the marker (lines 11-12). The last step (skip inner) is executed if C in r is greater than C in s, or if C values are equal but X in s is less than the start time B in r (lines 13-14). In such cases, the current tuple in s is skipped, moving to the next tuple (line 14). These processes ensure efficient matching of tuples based on the specified conditions.

Complexity Algorithm 8 has three main conditions, each of which moves the pointer for one relation forward by one step. It contributes $O(n + m)$ to the complexity. The while loop in lines 8–10 also generates the result tuples. Since there are z result tuples, the loop runs $O(z)$ times. Therefore, the overall time complexity of RMJ is $O(n + m + z)$.

With regard to the space complexity, it could be calculated as $O(m \cdot n)$. This is because if $|\mathbf{r}| = m$ and $|\mathbf{s}| = n$, it requires $m \times n$ space, as in the worst case, it could return all the tuples pairs from both relations.

6.2.2 Overlap Merge Join

Algorithm 9: OMJ (**r, s, C**)

input : Relation **r** with schema R, relation **s** with schema S, and equality attributes **C**
output: Result of $\mathbf{r} \bowtie_{\mathbf{r.C = s.C} \wedge Ov(\mathbf{r.P, s.P})} \mathbf{s}$

1 **r'** $\leftarrow$ **r** sorted by (**r.C**, **r**.B)
2 **s'** $\leftarrow$ **s** sorted by (**s.C**, **s**.B)
3 $z_1 \leftarrow$ RMJ(**r'**, **s'**, **C**, $B, \leq, B, <, E, R \circ S$)
4 $z_2 \leftarrow$ RMJ(**s'**, **r'**, **C**, $B, <, B, <, E, R \circ S$)
5 **return** $z_1 \uplus z_2$

Algorithm Algorithm 9 introduces a sort-merge approach, called OMJ, for performing the overlap join as defined in Definition 6. Initially, the two input relations are sorted based on the attributes in the equality predicate and the start point of the period attribute (lines 1–2). Next,

two range-merge joins (RMJs) are performed, with the roles of the input relations swapped for the second join (lines 3-4). Finally, the algorithm returns the union of the results from the two joins (line 5).

Complexity The complexity of the overlap join using the RMJ algorithm is $O(n \cdot \log n + m \cdot \log m + z)$, where n and m represent the cardinalities of the two input relations, and z is the size of the result. The process begins by sorting both input relations. If relation **r** has size n and relation **s** has size m, this sorting step requires $O(n \cdot \log n + m \cdot \log m)$. Then, we consider RMJ (see Algorithm 8). Since we add up two RMJs, combining these components, the total complexity is: $O(n \cdot \log n + m \cdot \log m) + O(n + m) + O(z) = O(n \cdot \log n + m \cdot \log m + z)$.

Regarding the space complexity, as it requires the space for $m \times n$ tuples in the worst case, the overall space complexity of OMJ is $O(m \cdot n)$.

6.3 Modifications and Implementations

6.3.1 Challenges

As range join is an equi join, RMJ compares the equality predicate between two relations (see Algorithm 8 lines 4, 6, 13). However, as this study aims to find overlaps among intervals where $r.C$ is greater than $s.C$, this method cannot be used. Hence, instead of comparing the common attributes in three steps (skip outer, join match, skip inner), the study restructured them into two. It skips outer in case the common attribute value of r is smaller or equal to the one of s. Otherwise, it returns a join match and skips inner. Additionally, dividing conditions to return results is essential, as checking conditions in one line could lead to a similar approach with brute force join, and this approach makes RMJ distinct.

Another obstacle arises when adapting RMJ: swapping the position of **r** and **s** to execute results without duplicates. It is because RMJ is computed twice, each checking different boundaries. Switching the position of two relations **r** and **s** does not affect the overall execution of Algorithm 8 if it compares the equality. It returns join matches if $r.C$ and $s.C$ are the same, and the rest would lead to either the inner or outer skip. However, this study examines whether $r.C$ is greater than $s.C$ and whether their intervals overlap. Thus, it is essential to distinguish them in two different algorithms, which will be explained further in Algorithms 10 and 11. Also, we keep the approach of bounding the interval on two sides as it is a unique property of RMJ.

Lastly, the original OMJ sorts relations **r** and **s** by **C** and B (see Algorithm 9 lines 1-2). However, this study takes arrays sorted by idx as inputs. Consequently, if we sort relations as Algorithm 9, it would cost $O(n \log n)$ given that RMJ is a sort merge-based algorithm. It would eventually lead to a slower execution of the result. Therefore, the modified version of OMJ takes sorted relations as input and does not sort them inside the algorithm like the original OMJ.

6.3.2 Implementation

Example 5 *Consider a relation* **r** *and* **s** *over an interval* ***P*** *in Table 4.1 (Sect. 4). Assume we would like to execute the query Q_t (see Example 1 in Chapter 4) with* MODIFIEDOMJ.

For inputs, we take **r** with schema R and **s** with schema S that are already sorted by idx, respectively. With both relations, we compute range merge join twice, and get the union of them: MODIFIEDRMJ1(**r**, **s**, $R \circ S$) $\uplus$ MODIFIEDRMJ2(**s**, **r**, $R \circ S$), where both R and S are (idx, B, E), respectively.

Figure 6.2b demonstrates the processing of the MODIFIEDRMJ2. For the first tuple 1 in **r** and the tuple 1 in **s**, it compares the values of the common attribute idx between them. As tuple $r[1]$ and $s[1]$ have the same idx values, r is advanced to the next tuple (indicated by "x"). Next, the idx value of $r[2]$ is greater than the idx value of $s[1]$; thus, we scan **s** from the beginning until it reaches the current s. Since the B value of $r[2]$ ($i.e.$, 2) is greater than the B value of $s[1]$ ($i.e.$, 1), but less than $s.E$ ($i.e.$, 3), it outputs a pair of tuples (2,1) (indicated by "✓"). The comparison stops as s is at tuple 1, and s is advanced to tuple 2 (indicated by "-"). With $r[2]$ and $s[2]$, r is advanced to the tuple 3, as they have the same idx values. When $r[3]$, the idx value of r is greater than the one of $s[2]$. Thus, the comparison begins and scans relation **s** from the beginning. However, from tuple 1 to 2, none has B, smaller than B of $r[3]$, and E, greater than B of $r[3]$. So the comparison terminates (indicated by "$\perp$"), and s is advanced to tuple 3. This time, r is advanced to tuple 4 as the idx values of r and s are 3. When $r[4]$, it scans s from tuple 1 to tuple 3. Since none of the tuples from 1 to 3 in **s** satisfy the condition to output a result, it terminates and advances s to tuple 4. Lastly, r and s have the same idx values, so r is advanced, but since tuple 4 is the last element, the loop is terminated.

The approach of MODIFIEDRMJ1 is similar to MODIFIEDRMJ2 as shown in Figure 6.2a. The critical difference is that r bounds $s.B$ on two sides, and $\leq$ is used for the start time. At first, it advances r to tuple 2 because tuple 1 in **r** and tuple 1 in **s** have the same idx values. The same process is repeated if $r.idx$ is greater than $s.idx$. When it is at $r[4]$ and $s[3]$, it scans **s** from tuple 1 to tuple 3. When scanning $s[1]$, its B value is 1, which satisfies the condition that is greater or equal to $r.B$ ($i.e.$, 0) and less than $r.E$ ($i.e.$, 2). Thus, it outputs a pair of tuples (4, 1). It keeps checking the condition, but the rest yield no results.

Consequently, MODIFIEDOMJ returns the union of the results from both algorithms: {(2,1), (4,1)}.

Range Merge Join Both Algorithms 10 and 11 operate on two input relations **r** and **s** with B and E as their attributes. It also considers a common attribute idx and two comparison operators $\prec^S, \prec^E$ that define whether the range join includes or excludes the start and/or endpoints (as per Definition 1). For $\prec^S$, $\leq$ is applied when it is Algorithm 10, but < when Algorithm 11. For $\prec^E$, we consider only < for both algorithms. This study sets the output schema O to (idx, B, E). The algorithm begins by reading the first tuple from each relation (lines 1-2). It continues processing until **r** is fully read, executing one of the if conditions (lines 3-12). The first if condition is triggered if the attribute idx in r is smaller or equal to the one in s (lines 4-5). In such cases, the current tuple in r is skipped (line 5). It enters the second if condition when the attribute idx in r is greater than s (line 6). Since it aims to compare all the values

r

	P	
idx	$[B$	$E]$
1	1	3
2	2	4
3	4	6
4	0	2

s

	P					
idx	$[B$	$E]$	1	2	3	4
1	1	3	x	-	$\perp$	✓
2	2	4		x	-	$\perp$
3	4	6			x	-
4	0	2				x

(a) MODIFIEDRMJ1(**r**,**s**,$R \circ S$)

s

	P	
idx	$[B$	$E]$
1	1	3
2	2	4
3	4	6
4	0	2

r

	P					
idx	$[B$	$E]$	1	2	3	4
1	1	3	x			
2	2	4	✓	x		
3	4	6	$\perp$	-	x	
4	0	2	$\perp$	$\perp$	-	x

(b) MODIFIEDRMJ2(**s**,**r**,$R \circ S$)

Figure 6.2: Modified range merge joins for our running example: ✓indicates a match, $\perp$ the end of matches, - an inner skip, and x an outer skip

from **s** which has a smaller idx value than r, it marks the first tuple in **s** as $sscan$ (line 7). A while-loop then scans **s** until $sscan$ reaches where the current s is marked and produces results for all subsequent tuples in $sscan$ that satisfy the join condition (lines 8-11). Both algorithms execute results in a certain condition with different bounds. For MODIFIEDRMJ1, it checks whether $sscan.B$ is greater or equal to $r.B$ and smaller than $r.E$ (see Algorithm 10 line 9). For MODIFIEDRMJ2, it checks whether $r.B$ is greater than $sscan.B$ and smaller than $sscan.E$ (see Algorithm 11 line 9). Once all matches for the current tuple in s are processed, the next tuple in s is retrieved (line 12). These methods guarantee that tuples are matched effectively according to the given criteria.

Overlap Merge Join Algorithm 12 shows the modified version of overlap merge join. It takes **r** with schema R and **s** with schema S. As mentioned in Sect. 6.3.1, it does not sort **r** and **s**. With the inputs, it computes MODIFIEDRMJ1 and MODIFIEDRMJ2 (lines 1-2). Finally, it returns the union of results from both algorithms (line 3).

Complexity The complexity of MODIFIEDRMJ1 can be analyzed as follows. The complexity of the algorithm depends on the operations and the size of the relations **r** and **s**. Let m and n represent the number of tuples in **r** and **s**, respectively. Firstly, the outer loop iterates over all tuples in **r**, running m times (line 3). Once entering the while loop, the conditions check whether the current tuple r has idx greater than the current tuple s. If false, the algorithm advances r. This comparison takes $O(1)$. If true, it enters the inner while loop. In this case,

Algorithm 10: MODIFIEDRMJ1

input : Relations **r** and **s** with attributes idx, B, E, and output schema O
output: Result of $\mathbf{r} \bowtie_{\mathbf{r}.idx > \mathbf{s}.idx \wedge \mathbf{r}.B \leq \mathbf{s}.B < \mathbf{r}.E} \mathbf{s}$

```
1  r ← first(r)
2  s ← first(s)
3  while r ≠ ω do
4      if r.idx ≤ s.idx then
5          r ← next(r)
6      else
7          sscan ← first(s)
8          while sscan ≤ s do
9              if (r.B ≤ sscan.B) ∧ (sscan.B < r.E) then
10                 output r and sscan according to schema O
11             sscan ← next(sscan)
12         s ← next(s)
```

Algorithm 11: MODIFIEDRMJ2

input : Relations **r** and **s** with attributes idx, B, E, and output schema O
output: Result of $\mathbf{r} \bowtie_{\mathbf{r}.idx > \mathbf{s}.idx \wedge \mathbf{s}.B < \mathbf{r}.B < \mathbf{s}.E} \mathbf{s}$

```
1  r ← first(r)
2  s ← first(s)
3  while r ≠ ω do
4      if r.idx ≤ s.idx then
5          r ← next(r)
6      else
7          sscan ← first(s)
8          while sscan ≤ s do
9              if (sscan.B < r.B) ∧ (r.B < sscan.E) then
10                 output r and sscan according to schema O
11             sscan ← next(sscan)
12         s ← next(s)
```

Algorithm 12: MODIFIEDOMJ

input : Relation **r** with schema R, relation **s** with schema S
output: Result of $\mathbf{r} \bowtie_{\mathbf{r.idx} > \mathbf{s.idx} \wedge Ov(\mathbf{r.P}, \mathbf{s.P})} \mathbf{s}$

```
1  z1 ← MODIFIEDRMJ1(r, s, R ∘ S)
2  z2 ← MODIFIEDRMJ2(s, r, R ∘ S)
3  return z1 ⊎ z2
```

for each r, it iterates over all tuples in **s** up to s. This loop can iterate over n tuples for each r. Inside the inner loop are condition checks and potentially an output operation, which are $O(1)$. The algorithm returns an output only if it meets the conditions. It is a constant-time operation for each pair of tuples that satisfies the condition. In the worst case, the inner loop runs n times for each m tuple in **r**. The complexity of the inner loop is $O(n)$, and the outer loop iterates m times. Therefore, the overall time complexity is $O(m \cdot n)$. The same applies to MODIFIEDRMJ2.

Both Algorithm 10 and Algorithm 11 traverse **r** and **s** and temporary variables for comparison and output. As this could lead to $m \times n$ tuples to output in the worst case, the space complexity of both algorithms is $O(m \cdot n)$.

Therefore, MODIFIEDOMJ can be analyzed as follows. Algorithm 12 utilizes MODIFIEDRMJ1 and MODIFIEDRMJ2, which have a complexity of $O(m \cdot n)$, where $m = |\mathbf{r}|$ and $n = |\mathbf{s}|$ (lines 1-2). The algorithm concatenates the results z_1 and z_2 using $\uplus$. Concatenating two lists of sizes $O(m \cdot n)$ involve a simple append operation, which takes $O(k)$, where k is the total size of the results. Since $k = O(m \cdot n) + O(m \cdot n) = O(m \cdot n)$, the cost of the union is $O(m \cdot n)$. Hence, the overall complexity is $O(m \cdot n) + O(m \cdot n) = O(m \cdot n)$

Regarding the space complexity, we need $O(1)$ additional space, assuming in-place processing and that output size is not included in space analysis. If the output size $O(m \cdot n)$ is included, the space complexity is $O(m \cdot n)$.

7 Experimental Evaluation

The study runs experimental cases to evaluate our approach empirically in different combinations of inequality predicates in IEJOIN along two dimensions: (i) data size and (ii) the level of overlapping among intervals. Based on the results, we compare the execution of MODIFIEDIEJOIN, MODIFIEDOMJ and the baseline join algorithm.

7.1 Datasets

The study uses synthetic data for the evaluation. Synthetic datasets are generated based on the schema as follows:

- n: Represents the number of intervals to generate. It defines the total number of unique intervals in the dataset.
- w: Specifies the width of each interval. All intervals have a uniform length determined by this parameter.
- p: Represents the overlap factor, a float in the range $[0, 1]$. This parameter determines how much consecutive intervals overlap. A higher value indicates more overlapping intervals.
- c: Defines the overlap case with three possible values
  - $c = 0$: No overlap. All intervals are spaced apart.
  - $c = 1$: Full overlap. All intervals start at the same point, overlapping entirely.
  - $c = 2$: Partial overlap. Consecutive intervals overlap based on the overlap factor p.

After generation, intervals are shuffled randomly to remove any inherent order. This flexible approach allows the creation of diverse datasets with varying levels of interval overlap.

7.2 Algorithms and Queries

For $n = 1000$, we conducted the evaluation on IEJOIN firstly. Afterwards, based on the results, we either used the MODIFIEDIEJOIN or restructured the condition (Algorithm 7 lines 4-5, 18). For $n = 5000, 10000$, we run the experiment only on IEJOIN.

We evaluate our algorithms using several queries with inequality join conditions, as we aim to show how the inequality queries can be optimized depending on cases c. For our first experiment, we used Q_s (Sect. 5.1). For the second experiment, we used Q_k:

n	w	p	c
	2	0.5	0
	2	0.5	1
1000	2	0.1	2
	2	0.5	2
	2	0.9971	2
	2	0.5	0
	2	0.5	1
5000	2	0.1	2
	2	0.5	2
	2	0.9993	2
	2	0.5	0
	2	0.5	1
10000	2	0.1	2
	2	0.5	2
	2	0.9997	2

Table 7.1: Parameter settings by data size

```
SELECT r.idx, s.idx
FROM r, s
WHERE r.idx > s.idx AND r.E > s.B;
```

For our third experiment, we used Q_p (Sect. 5.3.2). Finally, based on the execution time of each query using IEJOIN, Q_t (Example 1 Sect. 4) is tested on MODIFIEDIEJOIN and the study compares the results with other algorithms, which also execute Q_t.

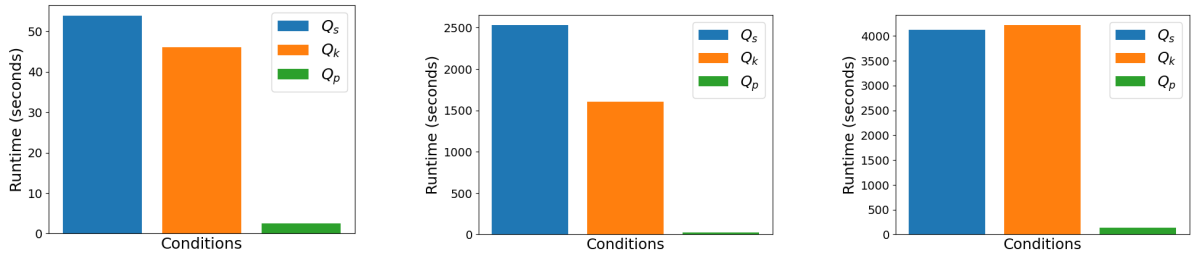
7.3 Setup and Parameter Setting

For our centralized evaluation setup, we used a personal computer equipped with an 11th Gen Intel Core i7-1165G7 processor (64-bit, quad-core), paired with 16GB RAM. All arrays are generated by each experiment and stored in memory.

We start our experimental evaluation by running queries Q_s, Q_k, Q_p on IEJOIN on different data sizes. The details are described in Table 7.1. In this study, we evaluate the algorithms on $n = 1000, 5000, 10000$. Regarding $c = 2$, the study uses $p = 0.1, 0.5$ and > 0.9 to test whether it would return a similar result with $c = 0$ or $c = 1$ depending on the extent of overlapping. The study selects the minimum value for $p > 0.9$ in $c = 2$, which yields a similar result with $c = 1$ after iterative experiments.

7.4 IEJoin Optimizer Experiments

In this set of experiments, we test the efficiency of IEJOIN by data size depending on the combinations of inequality predicates. The study applies Q_s, Q_k, Q_p to IEJOIN to evaluate



(a) Case 0 in $n = 1000$ (b) Case 0 in $n = 5000$ (c) Case 0 in $n = 10000$

Figure 7.1: Case 0 (no overlap) in IEJOIN by data size

Case	Query	Input	Output	Time (s)
0	Q_s	1K	252.8K	53.79
0	Q_k	1K	246.7K	45.99
0	Q_p	1K	1K	2.45
0	Q_s	5K	6.2M	2528.75
0	Q_k	5K	6.3M	1604.69
0	Q_p	5K	5K	27.05
0	Q_s	10K	24.7M	4118.38
0	Q_k	10K	25.2M	4212.59
0	Q_p	10K	10K	132.99

Table 7.2: Case 0 – Runtime and output size of IEJOIN

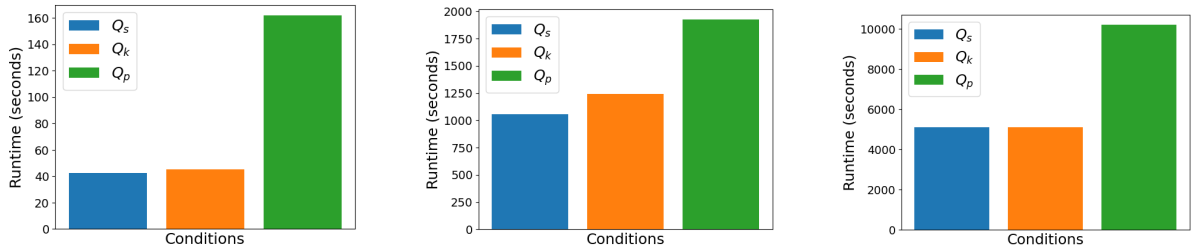
runtime and results to discover the highest selective predicates by different cases.

7.4.1 Case 0: No Overlap

Figure 7.1 and Table 7.2 show the results of queries Q_s, Q_k, Q_p on datasets with different sizes. The x-axis of each plot in Figure 7.1 represents the different combinations of inequality predicates ($i.e.$, queries), and the y-axis represents runtime to run queries. The figure reports that in all data sizes, Q_p shows the highest selectivity compared to other queries with different combinations of inequality predicates. For instance, when the data size is 1000, Q_s takes 53.79 seconds to execute the results, whilst Q_p takes only 2.45 seconds. In Table 7.2, Q_p is the fastest and has the highest selectivity regarding the number of outputs in all data sizes. Also, closer inspection of Table 7.2 shows that for $n = 10000$, the runtimes and the number of executed outputs for Q_s and Q_k are similar.

7.4.2 Case 1: All Overlap

Figure 7.2 and Table 7.3 present the outcomes of queries Q_s, Q_k, Q_p across datasets of varying sizes. The figure indicates that for all dataset sizes, Q_s and Q_k exhibit the highest selectivity compared to Q_p. To execute Q_s (resp. Q_k) with a data size of 5000, it takes 17 minutes 35 seconds (resp. 20 minutes 43 seconds), while Q_p takes 32 minutes 4 seconds. Although both



(a) Case 1 with $n = 1000$ (b) Case 1 with $n = 5000$ (c) Case 1 with $n = 10000$

Figure 7.2: Case 1 (all overlap) in IEJOIN by data size

Case	Query	Input	Output	Time (s)
1	Q_s	1K	499.5K	42.39
1	Q_k	1K	499.5K	45.42
1	Q_p	1K	1M	161.72
1	Q_s	5K	12.5M	1055.38
1	Q_k	5K	12.5M	1243.40
1	Q_p	5K	25M	1923.73
1	Q_s	10K	50.0M	5095.11
1	Q_k	10K	50.0M	5097.98
1	Q_p	10K	100M	10193.09

Table 7.3: Case 1 – Runtime and output size of IEJOIN

queries have slight differences in runtime, Q_s and Q_k return the same number of output results as shown in Table 7.3. Accordingly, both have the highest selectivity, given the runtimes and the number of outputs in all data sizes.

7.4.3 Case 2: Some Overlap

Case 2: $p = 0.1$ Figure 7.3 and Table 7.4a display the results of queries Q_s, Q_k and Q_p on datasets of different sizes. Figure 7.3 shows that Q_p has the highest selectivity across all dataset sizes when 10% of datasets overlap, similar to case 0 (no overlap), compared to Q_s and Q_k. Executing Q_p with a data size of 10000 takes 1 minute 42 seconds, while for Q_s 2 hours 7 minutes and Q_k 1 hour 13 minutes. This notable difference is also reflected in the number of tuples returned. Table 7.4a shows Q_p is very selective as it returns about 30.0K tuples. Therefore, $r.B < s.E \wedge r.E > s.B$ is the inequality predicate with the highest selectivity for case 2 with $p = 0.1$. This outcome is noteworthy as it bears similarity to case 0, potentially attributable to the minimal overlap present among the intervals. Furthermore, in Table 7.4a, there is a clear trend of similarity between Q_s and Q_k in terms of the number of outputs and runtimes when the input size is 5000 and 10000.

Case 2: $p = 0.5$ Figure 7.4 and Table 7.4b present the outcomes of queries Q_s, Q_k and Q_p on various dataset sizes. According to Figure 7.4, Q_p demonstrates the highest selectivity

Case	Query	Input	Output	Time (s)
2	Q_s	1K	247.2K	37.61
2	Q_k	1K	253.3K	87.34
2	Q_p	1K	3.0K	5.29
2	Q_s	5K	6.4M	787.73
2	Q_k	5K	6.1M	778.28
2	Q_p	5K	15.0K	27.43
2	Q_s	10K	25.1M	7625.31
2	Q_k	10K	24.9M	4391.16
2	Q_p	10K	30.0K	102.30

(a) Case 2 ($p = 0.1$)

Case	Query	Input	Output	Time (s)
2	Q_s	1K	246.0K	103.42
2	Q_k	1K	254.5K	108.20
2	Q_p	1K	3.0K	5.28
2	Q_s	5K	6.2M	778.01
2	Q_k	5K	6.4M	807.09
2	Q_p	5K	15.0K	26.90
2	Q_s	10K	25.1M	3129.08
2	Q_k	10K	24.9M	4134.84
2	Q_p	10K	30.0K	354.64

(b) Case 2 ($p = 0.5$)

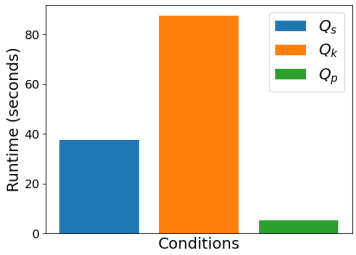
Case	Query	Input	Output	Time (s)
2	Q_s	1K	385.9K	59.72
2	Q_k	1K	401.8K	79.47
2	Q_p	1K	577.3K	88.72
2	Q_s	5K	9.5M	2353.89
2	Q_k	5K	9.4M	3911.78
2	Q_p	5K	12.8M	3714.08
2	Q_s	10K	37.8M	12240.05
2	Q_k	10K	37.7M	4730.19
2	Q_p	10K	51.1M	6740.57

(c) Case 2 ($p > 0.9$)

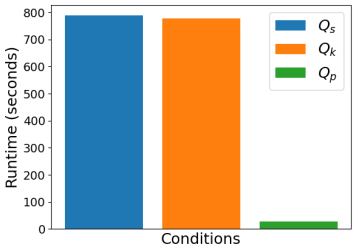
Table 7.4: Case 2 – Runtime and output size of IEJOIN

across all dataset sizes when there is a 50% overlap, similar to the scenarios with no overlap (case 0) and with $p = 0.1$ (case 2), in comparison to Q_s and Q_k. Running Q_p on a dataset of 10000 records takes 5 minutes and 55 seconds, whereas Q_s takes 52 minutes and 9 seconds, and Q_k takes 1 hour and 8 minutes. This time difference is also evident in the number of tuples returned. Table 7.4b indicates that Q_p is highly selective, returning 30000 tuples, while the other two queries return nearly 25 million tuples, given the data size is 10000. Consequently, the inequality predicates $r.B < s.E \wedge r.E > s.B$ demonstrate the highest selectivity for case 2 with $p = 0.5$. Moreover, from Table 7.4b, it can be seen that Q_s and Q_k return a similar number of results in a similar amount of times across all data sizes.

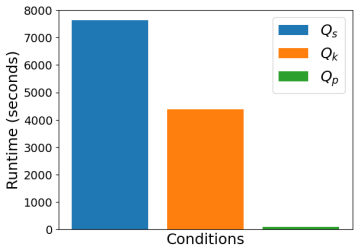
Case 2: $p > 0.9$ Figure 7.5 and Table 7.4c demonstrate the results of queries Q_s, Q_k and Q_p on datasets of different sizes. Q_s and Q_k have the highest selectivity across all dataset sizes when more than 90% of intervals overlap, compared to Q_p. For $n = 1000$, executing Q_s takes 59.72 seconds, while Q_k takes 1 minute and 19 seconds. For larger datasets ($n = 5000$ and $n = 10000$), the execution times for Q_s and Q_k differ slightly. For example, with a dataset size of 5000, Q_s takes 39 minutes and 13 seconds, and Q_k takes 1 hour and 5 minutes. For $n = 10000$, Q_k takes 1 hour and 18 minutes, whereas Q_s takes 3 hours and 24 minutes.



(a) Case 2 ($p = 0.1$) with $n = 1000$

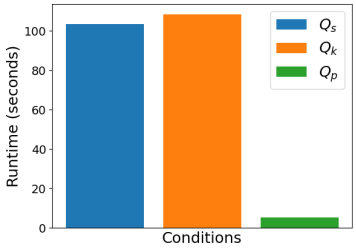


(b) Case 2 ($p = 0.1$) with $n = 5000$

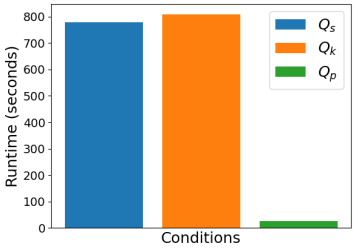


(c) Case 2 ($p = 0.1$) with $n = 10000$

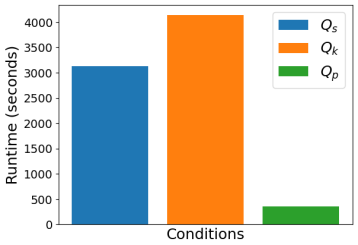
Figure 7.3: Case 2 ($p = 0.1$) in IEJOIN by data size



(a) Case 2 ($p = 0.5$) in $n = 1000$

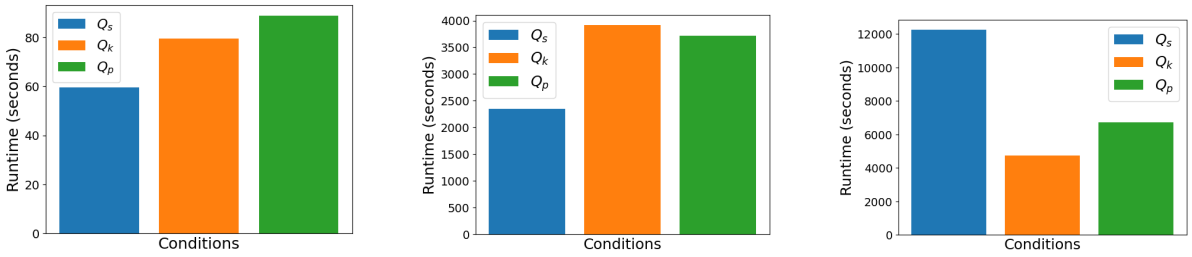


(b) Case 2 ($p = 0.5$) in $n = 5000$



(c) Case 2 ($p = 0.5$) in $n = 10000$

Figure 7.4: Case 2 ($p = 0.5$) in IEJOIN by data size



(a) Case 2 ($p = 0.9971$) in $n = 1000$

(b) Case 2 ($p = 0.9993$) in $n = 5000$

(c) Case 2 ($p = 0.9997$) in $n = 10000$

Figure 7.5: Case 2 ($p > 0.9$) in IEJOIN by data size

Despite these differences, both Q_s and Q_k return a similar number of tuples across all dataset sizes, as shown in Table 7.4c. Therefore, the inequality predicates $r.idx \wedge r.B < s.E$ and $r.idx > s.idx \wedge r.E > s.B$ are the highest selectivity for case 2 with $p > 0.9$.

7.5 Experiments on Optimized Algorithms for Comparison

This section analyzes the implementation of the combined approaches, MODIFIEDIEJOIN and MODIFIEDOMJ, comparing with the baseline algorithm (Algorithm 1), experimentally. Regarding IEJOIN, we combined inequality predicates differently depending on the results from Sect. 7.4 for the adoption of the modified version of IEJOIN. It indicates this study picks the two inequality predicates with the most selective inequality join, which return the fewest outputs and spend the least time (see Algorithm 7 lines 13-15). Then, an algorithm supposedly proceeds by the last inequality predicate in the if condition (see Algorithm 7 line 18). Given this, the research uses Algorithm 7 for case 0 and case 2 with $p = 0.1$. This experiment excluded case 2 ($p = 0.5$) for evaluation as the range of p, which returns similar results with $p = 0.1$, is broad, thus, close to 0.9. Although case 1 and case 2 ($p > 0.9$) supposedly show the best efficiency in Q_s and Q_k, they show the difference in execution time because of the limitation in dataset generation. Therefore, we choose the query returning the fastest runtime for inequality join and proceed with the remaining inequality condition. For case 1, the runtime of Q_s (see Table 7.3) and for case 2 with $p > 0.9$, the runtime of Q_s (see Table 7.4c) were the fastest. Hence, the study adopts Algorithm 13. The detailed implementation of Algorithms 13 and 14 is described in Chapter 9. The study checks the accuracy of results from each algorithm by comparing the results with the baseline algorithm, BFJOIN (Algorithm 1).

Figure 7.6 presents the result of executing Q_t by case with different algorithms when the dataset is $n = 1000$. When no intervals overlap (case 0), it reveals that MODIFIEDIEJOIN is the fastest (3.19 seconds) among all algorithms. On the other hand, MODIFIEDOMJ returns the query in 95.25 seconds, making it the slowest among all, whilst BFJOIN does in 43.77 seconds.

For case 1 (all overlap), the baseline algorithm was the fastest (38.48 seconds), while

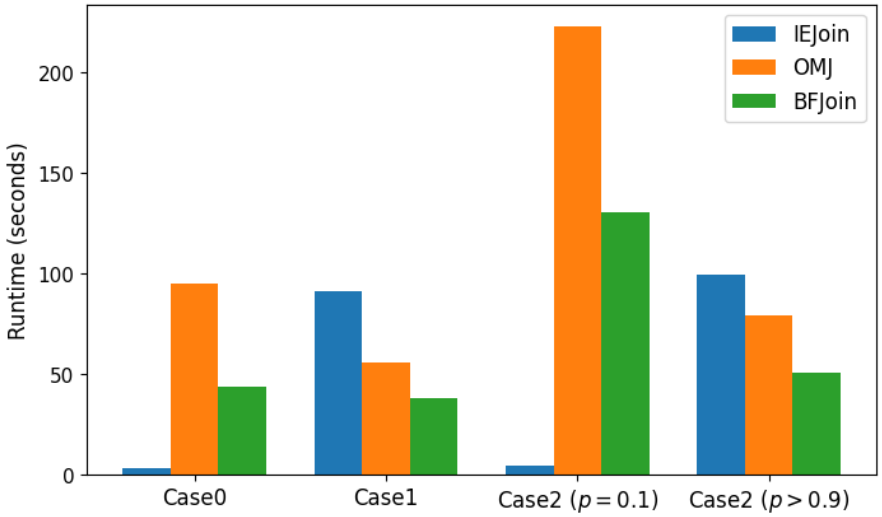


Figure 7.6: MODIFIEDIEJOIN versus MODIFIEDOMJ versus Brute-force Join (n = 1000)

MODIFIEDIEJOIN was the slowest (1 minute 31 seconds).

If 10% of intervals in datasets overlap (case 2 ($p = 0.1$)), MODIFIEDIEJOIN is significantly more efficient than the other two algorithms, recording 4.45 seconds for executing Q_t. Both algorithms take more than 2 minutes, denoting about 3 minutes 43 seconds for MODIFIEDOMJ and 2 minutes and 10 seconds for BFJOIN.

However, when more than 90% of the data overlap (case 2 ($p > 0.9$)), the results are similar to those in case 1. The baseline algorithm is the most efficient in producing the result, showing 50.69 seconds of runtime.

Therefore, it is analyzed that MODIFIEDIEJOIN performs the best when no data overlap or up to 50% of intervals overlap, MODIFIEDOMJ performs the worst regarding memory footprint. Furthermore, the baseline algorithm is the most efficient when all intervals overlap or most of the intervals ($p > 0.9$) overlap, following MODIFIEDOMJ and MODIFIEDIEJOIN.

8 Conclusion and Future Work

This study set out to find a new method for inequality join between relations with overlapping intervals and index processing. The study implemented a baseline join algorithm using brute force nested loop join to compare it to the other two optimized algorithms and assess their efficiency. Also, the research has developed optimized join algorithms based on the existing inequality join algorithms, such as IEJOIN, RMJ and OMJ. For instance, MODIFIEDIEJOIN joins relations concerning two inequality predicates like the original IEJOIN. However, when returning the results, the algorithm checks whether a tuple meets the additional inequality join condition. The order of combining inequality predicates was determined by evaluating the efficiency of executing results. Moreover, both MODIFIEDRMJ1 and MODIFIEDRMJ2 firstly examine whether the current tuple meets the condition: $r.idx > s.idx$ before checking intervals overlapping. If it meets the condition, they proceed with matching the conditions for interval overlapping. MODIFIEDOMJ runs both algorithms to return the union of results. Based on these algorithms, the study conducted experimental evaluations on efficiency by case with different sizes of datasets. The investigation has shown that when no data intervals or up to 50% of intervals overlap, Q_p performs the best concerning runtime. Hence, Algorithm 7 was applied for comparison with other algorithms in case 0 and case 2 with $p = 0.1$ and 0.5. The experiments also confirmed that when all data intervals or more than 90% of intervals overlap, Q_s and Q_k perform best. Therefore, we applied Algorithm 13 for case 1 and case 2 with $p > 0.9$. The results of this investigation among algorithms show that MODIFIEDIEJOIN is the most efficient in case 0 and case 2 with $p = 0.1$ and 0.5. For cases where more than 90% intervals overlap, the baseline algorithm was the most efficient, followed by MODIFIEDOMJ, due to its index-based approach, and MODIFIEDIEJOIN.

Taken together, these results suggest that indexed-based filtering ($i.e., r.idx > s.idx$) is more efficient when most intervals overlap, such as case 1 and case 2 ($p > 0.9$). This is because index comparison is simpler than checking the actual interval boundaries. Given that many intervals overlap, checking the interval boundaries first would be more exhausting, leading to the inequality join condition's low selectivity. Since most intervals overlap, the index-based filtering quickly narrows down the candidates for further checks.

On the other hand, when fewer intervals overlap like case 0 and case 2 ($p = 0.1$), interval-based filtering ($i.e., r.B < s.E \wedge r.E > s.B$) is more efficient. This is because the index-based filtering might not significantly reduce the number of comparisons needed. Instead, directly checking the interval boundaries can quickly eliminate non-overlapping intervals, making the process more efficient.

Similarly, the results of comparing algorithms suggest that optimized algorithms' performance is determined by whether index-based filtering or interval-based filtering precedes. MODIFIEDIEJOIN (Algorithm 7) excels with fewer overlapping intervals due to its interval-based filtering. It leverages interval-based filtering to quickly screen and join tuples based on

inequality predicates, which reduces the number of comparisons needed. Based on this, it can be interpreted that MODIFIEDRMJ ranked second in runtime, as it scans indexes first before checking overlapping intervals. Nevertheless, when there is a high degree of overlap ($i.e.$, case 1, case 2 ($p > 0.9$)), the brute force join algorithm is faster due to its index-based filtering. Although MODIFIEDIEJOIN1 (resp. MODIFIEDIEJOIN2) checks indexes in Algorithm 13 (resp. Algorithm 14), it needs to handle not just indexes but also another inequality condition for intervals together.

Furthermore, this evaluation's results reveal that the runtimes for the equivalent selectivity ratio, Q_s and Q_k, are similar across all cases. In Sect. 7.4, it is observed that cases that perform the best with index-based filtering, as well as cases 0 and 2 with $p = 0.1$ and 0.5, tend to execute Q_s and Q_k in comparable runtime, which both involve an index inequality predicate ($i.e., r.idx > s.idx$). Consequently, it can be asserted that the overall performance of the algorithms will be consistent, provided that the queries applied have an equivalent selectivity ratio, regardless of the size of the dataset and the extent of overlap.

The importance of this study lies in its novel approach to optimizing inequality joins between relations with overlapping intervals. The findings highlight that the choice between index-based and interval-based filtering is crucial, depending on the degree of interval overlap. Moreover, the study's experimental results provide a clear understanding of when each algorithm performs best, offering practical guidelines for database practitioners. This contributes to the broader field of knowledge by enhancing the performance of inequality joins, which are fundamental operations in many database applications.

A limitation of this study is that it was not able to reproduce an offset array algorithm with the time complexity of $O(n \log n)$. As described in Sect. 5.3.1, Khayyat et al. lacked explanations about the implementation of an offset array, which is the critical part of IEJOIN. The current offset array algorithms have the time complexity of $O(m \cdot n)$, as it compares each element from both arrays in nested loops. However, the quadratic time complexity can make it slow, especially when dealing with large datasets. As a result, the performance may not be suitable in certain conditions. Furthermore, the inability to achieve a time complexity of $O(n \log n)$ suggests that there may be underlying constraints or complexities within the problem domain that have yet to be addressed. This gap emphasizes the need for further research to identify potential strategies or modifications.

An additional uncontrolled factor is the possibility that the current study has yet to achieve maximum efficiency when modifying range merge joins. The range merge joins intentionally bound an interval on two sides, supposedly limiting the returned number of tuples, which leads to high selectivity. Hence, in theory, it should outperform the baseline algorithm, as it checks intervals over four attributes like SQL. Nevertheless, the experimental evaluation revealed that the brute force join demonstrated better performance in case 1 and case 2 ($p > 0.9$). Both algorithms scan indexes first, then proceed with inequality predicates for overlapping intervals. A possible explanation for this might be the distiction between two algorithms characterized by the bounds involved and the number of computations. The brute-force join algorithm is computed once and bounds intervals by four attributes: $r.B < s.E \wedge r.E > s.B$. These inequality predicates are checked simultaneously to see if r and s meet the inequality condition. Conversely, MODIFIEDOMJ performs the computation twice, as it involves two MODIFIEDRMJs. Each algorithm checks the different conditions due to the constraints imposed by upper and

lower bounds: $r.B \leq s.B < r.E \vee s.B < r.B < s.E$. This approach ensures no duplicates within outcomes. In summary, the brute-force join algorithm is characterized by its simplicity, involving a single computation, whereas MODIFIEDOMJ is more complex, incorporating additional computations to prevent duplicate results. Therefore, this suggests that although RMJ intentionally adopts several bounds to restrict the number of outcomes, this additional complexity over boundaries and overhead of the modified range merge join may not be justified under the conditions tested. A natural progression of this work is to analyze the range merge join with additional inequality predicate, identify scenarios where it could outperform brute-force join and optimize its implementation for better efficiency.

Notwithstanding these limitations, this thesis has provided a deeper insight into optimizing inequality joins between relations with overlapping intervals and additional inequality predicate. By developing and evaluating new algorithms, such as MODIFIEDIEJOIN, MODIFIEDRMJ and MODIFIEDOMJ, this research has significantly advanced the efficiency of query processing in database management systems. The findings highlight that the choice between index-based and interval-based filtering is crucial, depending on the degree of interval overlap. This insight can guide future database optimization strategies, ensuring that the most efficient algorithm is applied based on the data's specific characteristics. Moreover, the study's experimental results clearly explain when each algorithm performs best, offering practical guidelines for join operations. Thus, the present study lays the groundwork for future research into enhancing the performance of inequality joins, which are fundamental operations in many database applications.

9 Appendix

9.1 IEJoin with Different Combinations of Inequality Predicates

Here, we show two different versions of the modified IEJOIN for Chapter 7. The overall logic is the same as Algorithm 7 except for what it takes as $L_1, L_2, L'_1, L'_2, L_3, L'_3$, and accordingly, conditions to check before returning an output.

Algorithm 13 takes $r.idx$ (resp. $s.idx$) as L_1 (resp. L'_1) and $r.B$ (resp. $s.E$) as L_2 (resp. L'_2). It checks the additional condition $r.E > s.B$ in line 18.

Algorithm 13: MODIFIEDIEJOIN1

input : query Q with 3 join predicates $r.idx > s.idx$, $r.B < s.E$ and $r.E > s.B$, tables **r**, **s** of sizes m and n resp.
output: a list of tuple pairs (r_i, s_j)

1 let L_1 (resp. L_2) be the array of idx (resp. B) in **r**
2 let L'_1 (resp. L'_2) be the array of idx (resp. E) in **s**
3 let L_3 (resp. L'_3) be the array of positions of L_2 (resp. L'_1)
4 sort L_1, L'_1 in descending order
5 sort L_2, L'_2 in ascending order
6 compute the permutation array P of L_2 *w.r.t.* L_1
7 compute the permutation array P' of L'_2 *w.r.t.* L'_1
8 compute the offset array O_1 of L_1 *w.r.t.* L'_1
9 compute the offset array O_2 of L_2 *w.r.t.* L'_2
10 initialize bit-array B' ($|B'| = n$), and set all bits to 0
11 initialize join_result as an empty list for tuple pairs
12 **for** ($i \leftarrow 1$ **to** m) **do**
13 $\text{off}_2 \leftarrow O_2[i]$
14 **for** ($j \leftarrow 1$ **to** min(off_2, size(L'_2))) **do**
15 $B'[P'[j]] \leftarrow 1$
16 $\text{off}_1 \leftarrow O_1[P[i]]$
17 **for** ($k \leftarrow \text{off}_1$ to n) **do**
18 **if** $(B'[k] = 1) \wedge (\mathbf{r}[L_3[i]].B < \mathbf{s}[L'_3[k]].E)$ **then**
19 add tuples *w.r.t.* $(L_2[i], L'_1[k])$ to join_result
20 **return** join_result

Algorithm 14 takes $r.idx$ (resp. $s.idx$) as L_1 (resp. L'_1) and $r.E$ (resp. $s.B$) as L_2 (resp. L'_2). It checks the additional condition $r.B < s.E$ in line 18.

Algorithm 14: MODIFIEDIEJOIN2

input : query Q with 3 join predicates $r.idx > s.idx$, $r.B < s.E$ and $r.E > s.B$, tables **r**, **s** of sizes m and n resp.
output: a list of tuple pairs (r_i, s_j)

```
1  let L_1 (resp. L_2) be the array of idx (resp. E) in r
2  let L'_1 (resp. L'_2) be the array of idx (resp. B) in s
3  let L_3 (resp. L'_3) be the array of positions of L_2 (resp. L'_1)
4  sort L_1, L'_1 in descending order
5  sort L_2, L'_2 in descending order
6  compute the permutation array P of L_2 w.r.t. L_1
7  compute the permutation array P' of L'_2 w.r.t. L'_1
8  compute the offset array O_1 of L_1 w.r.t. L'_1
9  compute the offset array O_2 of L_2 w.r.t. L'_2
10 initialize bit-array B' (|B'| = n), and set all bits to 0
11 initialize join_result as an empty list for tuple pairs
12 for (i ← 1 to m) do
13     off_2 ← O_2[i]
14     for (j ← 1 to min(off_2, size(L'_2))) do
15         B'[P'[j]] ← 1
16     off_1 ← O_1[P[i]]
17     for (k ← off_1 to n) do
18         if (B'[k] = 1) ∧ (r[L_3[i]].E > s[L'_3[k]].B) then
19             add tuples w.r.t. (L_2[i], L'_1[k]) to join_result
20 return join_result
```

Bibliography

[Che24] Alvin Cheung. Iterators and joins. Database Systems, `https://cs186berkeley.net/notes/note9/`, 2024. Accessed: September 13, 2024.

[DBG+22] Anton Dignös, Michael H. Böhlen, Johann Gamper, Christian S. Jensen, and Peter Moser. Leveraging range joins for the computation of overlap joins. *The VLDB Journal*, 31:75–99, 2022.

[GMUW08] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Pearson, Upper Saddle River, NJ, 2nd edition, 2008.

[KLS+17] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. Fast and scalable inequality joins. *The VLDB Journal*, 26:125–150, 2017.

[RLT20] Ran Rui, Hao Li, and Yi-Cheng Tu. Efficient join algorithms for large database tables in a multi-gpu environment. In *Proceedings of the VLDB Endowment*, volume 14, 2020.

[SKS19] Avi Silbershcatz, Henry F. Korth, and S. Sudarshan. *Database Management Systems*. McGraw-Hill, New York, NY, 7th edition, 2019.