Noname manuscript No.
(will be inserted by the editor)

Fast and Scalable Inequality Joins

Zuhair Khayyat · William Lucia · Meghna Singh · Mourad Ouzzani · Paolo Papotti · Jorge-Arnulfo Quiané-Ruiz · Nan Tang · Panos Kalnis

Abstract Inequality joins, which is to join relations with inequality conditions, are used in various applications. Optimizing joins has been the subject of intensive research ranging from efficient join algorithms such as sort-merge join, to the use of efficient indices such as B^+-tree, R^*-tree and Bitmap. However, inequality joins have received little attention and queries containing such joins are notably very slow. In this paper, we introduce fast inequality join algorithms based on sorted arrays and space efficient bit-arrays. We further introduce a simple method to estimate the selectivity of inequality joins which is then used to optimize multiple predicate queries and multi-way joins. Moreover, we study an incremental inequality join algorithm to handle scenarios where data keeps changing. We have implemented a centralized version of these algorithms on top of PostgreSQL, a distributed version on top of Spark SQL, and an existing data cleaning system, NADEEF. By comparing our algorithms against well known optimization techniques for inequality joins, we show our solution is more scalable and several orders of magnitude faster.

Z. Khayyat
King Abdullah University of Science and Technology,
Saudi Arabia
E-mail: zuhair.khayyat@kaust.edu.sa

W. Lucia
Qatar Computing Research Institute, HBKU, Qatar
E-mail: williamlucia.wl@gmail.com

M. Singh
Qatar Computing Research Institute, HBKU, Qatar
E-mail: mesingh@qf.org.qa

M. Ouzzani
Qatar Computing Research Institute, HBKU, Qatar
E-mail: mouzzani@qf.org.qa

P. Papotti
Arizona State University, USA
E-mail: ppapotti@asu.edu

J. Quiané-Ruiz
Qatar Computing Research Institute, HBKU, Qatar
E-mail: jquianeruiz@qf.org.qa

N. Tang
Qatar Computing Research Institute, HBKU, Qatar
E-mail: ntang@qf.org.qa

P. Kalnis
King Abdullah University of Science and Technology,
Saudi Arabia
E-mail: panos.kalnis@kaust.edu.sa

1 Once Upon a Time ...

Bob[1], a data analyst working for an international provider of cloud services, wanted to analyze revenue and utilization trends from different regions. In particular, he wanted to find out all those transactions from the West-Coast that last longer and produce smaller revenues than any transaction in the East-Coast. In other words, he was looking for any customer from the West-Coast who rented a virtual machine for more hours than any customer from the East-Coast, but who paid less. Figure 1 illustrates a data instance for both tables. He wrote the following query for such a task:

Q_t : SELECT east.id, west.t_id
FROM east, west
WHERE east.dur < west.time AND east.rev > west.cost;

Bob first ran Q_t over 200K transactions on the distributed system storing the data (System-X). Given that the input dataset is ∼1GB, he expected a minute response time or so. However, he waited for more than three hours without seeing any result. He immediately thought that this problem comes from System-X and killed the query. He then used an open-source DBMS-X

[1] We motivate the problem with a real-life story. Names and queries have been changed for confidentiality reasons.

east	id	dur	rev	cores
r_1	100	140	9	2
r_2	101	100	12	8
r_3	102	90	5	4

west	t_id	time	cost	cores
s_1	404	100	6	4
s_2	498	140	11	2
s_3	676	80	10	1
s_4	742	90	5	4

Fig. 1 East-Coast and West-Coast transactions

to run his query. Although join is by far the most important and most studied operator in relational algebra [1], Bob had to wait for over two hours until DBMS-X returned the results. He found that Q_t is processed by DBMS-X as a *Cartesian product* followed by a *selection* predicate, which is problematic due to the huge number of unnecessary intermediate results.

In the meantime, Bob heard that a big DBMS vendor was in town to highlight the power of their recently released distributed DBMS to process big data (DBMS-Y). So he visited them with a small (few KBs) dataset sample of the tables to run Q_t. Surprisingly, DBMS-Y could not run Q_t for even that small sample! He spent 45 minutes waiting while one of the DBMS-Y experts was trying to solve the issue. Bob left the query running and the vendor never contacted him again. In fact, DBMS-Y is using underneath the same open-source DBMS-X that Bob tried before. He thus understood that a simple distribution of the process does not solve his problem. Afterwards, Bob decided to call one of his friends working for a very famous DBMS vendor. His friend kindly accepted to try Q_t on their DBMS-Z, which is well reputed to deal with terabytes of data. A couple of days later, his friend came back to him with several possible ways (physical plans) to run Q_t on DBMS-Z. Nonetheless, all these query plans still had the quadratic complexity of a Cartesian product with its inherent inefficiency.

Despite the prevalence of this kind of queries in applications, such as temporal and spatial databases, and data cleaning, no off-the-shelf efficient solutions exist. There have been countless techniques to optimize the different flavors of joins in various settings [19]. In the general case of a theta join, these techniques are mostly based on two assumptions: (i) one of the relations is small enough to fit in memory, and (ii) queries contain selection predicates or an equijoin with a high selectivity, which would reduce the size of the relations to be fed to the inequality join. The first assumption does not hold when joining two big relations. The second assumption is not necessarily true with low-selectivity predicates, such as gender or region, where the obtained relations are still very large. Furthermore, similar to Q_t, there is a large spectrum of applications where the above two assumptions do not necessarily hold. For example, for *data analytics* in a temporal database, a typical query would be to find all employees and managers that overlapped while working in a certain company [18]. In data cleaning, when *detecting violations* based on denial constraints, an example of a query is to find all pairs of tuples such that one individual pays more taxes but earns less than another individual [10].

Bob then started looking at alternatives. Common ways of optimizing such queries include sort-merge joins [13] and interval-based indexing [9, 17, 22]. Sort merge join reduces the search space by sorting the data based on the joining attributes and merging them. However, it still has a quadratic complexity for queries with inequality joins only. Interval-based indexing reduces the search space of such queries even further by using bitmap interval indexing [9]. However, such indices require large memory space [36] and long index building time. Moreover, Bob would have to create multiple indices to cover all those attributes referenced in his query workload. Such indices can be built at query time, but their long construction time renders them impractical.

With no hope in the horizon, Bob decided to talk with his friends who happen to do research in data analytics. They happily started working on this interesting problem. After several months of hard work, they came out with IEJoin, a new algorithm that utilizes bit-arrays and positional permutation arrays to achieve fast inequality joins. Given the inherent quadratic complexity of inequality joins, IEJoin follows the *RAM locality is King* principle coined by Jim Gray. The use of memory-contiguous data structures with small footprint leads to orders of magnitude performance improvement over the prior art. The basic idea of our proposal is to create a sorted array of tuples for each inequality comparison and compute their intersection, which would output the join results. The prohibitive cost of the intersection operation is alleviated through the use of a permutation array to encode positions of tuples in one sorted array *w.r.t.* the other sorted array (assuming that there are only two conditions). A bit-array is then used to emit the join results.

This work extends [25] along several lines including dealing with the not equal operator, improving the bitmap, optimizing multi-predicate queries and query planning for multi-way joins through selectivity estimation, and an incremental version of the algorithm.

Contributions. (1) We present novel, fast and space efficient inequality join algorithms (Sections 2 and 3). Furthermore, we discuss two optimization techniques to significantly speed up the computation (Section 3.3).

Specifically, we exploit bitmaps to reduce the search space, and reorganize data to improve data locality.

(2) We discuss selectivity estimation for optimizing multi-predicate queries and query planning for multi-way joins (Section 4).

(3) We devise incremental inequality join algorithms to deal with dynamic data (Section 5). We show that our proposed algorithms, leveraging Packed-memory array [5], can be easily adapted to handle data updates.

(4) To handle very large datasets, we present a distributed version of our algorithm that can be deployed on systems such as Spark SQL [4]. In particular, we use attribute metadata (*e.g.,* min and max values) to greatly reduce data shuffling (Section 6).

(5) We implement our algorithms in three existing systems, namely PostgreSQL, Spark SQL, and NADEEF (Section 7). We conduct an extensive experimental study by comparing against well known optimization techniques. The results show that our proposed solution is more general, scalable, and orders of magnitude faster than the state-of-the-art (Section 8).

Furthermore, we discuss related work in Section 9 and conclude the paper in Section 10.

2 Solution Overview

In this section, we restrict our discussions to queries with inequality predicates only. Each predicate is of the form: A_i op B_i where A_i (resp., B_i) is an attribute in a relation R (resp., S) and op is an inequality operator in the set $\{<, >, \leq, \geq\}$.

Example 1 [Single predicate] Consider the west table in Figure 1 and an inequality self-join query Q_s:

Q_s : SELECT s_1.t_id, s_2.t_id
FROM west s_1, west s_2
WHERE s_1.time > s_2.time;

Query Q_s returns a set of pairs $\{(s_i, s_j)\}$ where s_i takes more time than s_j; the result is $\{(s_2, s_1), (s_2, s_3), (s_2, s_4), (s_1, s_3), (s_1, s_4), (s_4, s_3)\}$.

A natural idea to handle inequality join on one attribute is to leverage a sorted array. For instance, we sort west tuples on time in ascending order in array L_1:$\langle s_3, s_4, s_1, s_2\rangle$. We denote by $L[i]$ the i-th element in array L, and $L[i, j]$ its sub-array from position i to position j. Given a tuple s, any tuple at $L_1[k]$ ($k \in [1, i-1]$) has time value less than $L_1[i]$, the position of s in L_1. Consider Example 1, tuple s_1 in position $L_1[3]$ joins with tuples in positions $L_1[1, 2]$, namely s_3 and s_4.

Example 2 [Two predicates] Let us now consider a self-join with two inequality conditions:

Q_p : SELECT s_1.t_id, s_2.t_id
FROM west s_1, west s_2
WHERE s_1.time > s_2.time AND s_1.cost < s_2.cost;

Q_p returns pairs (s_i, s_j) where s_i takes more time but pays less than s_j; the result is $\{(s_1, s_3), (s_4, s_3)\}$.

Similar to attribute time in Example 1, we also sort attribute cost in ascending order into an array L_2:$\langle s_4, s_1, s_3, s_2\rangle$, as shown below.

L_1	$s_3(80)$	$s_4(90)$	$s_1(100)$	$s_2(140)$	(sort ↑ on time)
L_2	$s_4(5)$	$s_1(6)$	$s_3(10)$	$s_2(11)$	(sort ↑ on cost)

Thus, given a tuple s whose position in L_2 is j, any tuple $L_2[l]$ ($l \in [j+1, n]$) has a higher cost than s, where n is the size of the input relation. Our observation here is as follows. For any tuple s', to form a join result (s, s') with tuple s, the following two conditions must be satisfied: (i) s' is on the left of s in L_1, *i.e., s* has a larger value for time than s', and (ii) s' is on the right of s in L_2, *i.e., s* has a smaller value for cost than s'. Thus, all tuples in the intersection of $L_1[1, i-1]$ and $L_2[j+1, n]$ satisfy these two conditions and belong to the join result. For example, s_4's position in L_1 (resp. L_2) is 2 (resp. 1). Hence, $L_1[1, 2-1] = \langle s_3\rangle$ and $L_2[1+1, 4] = \langle s_1, s_3, s_2\rangle$, and their intersection is $\{s_3\}$, producing (s_4, s_3). To obtain the final result, we simply repeat the above process for each tuple.

The challenge is how to perform the aforementioned intersection operation in an efficient manner. There already exist several indices, such as R-tree and B^+-tree, that can possibly help. R-tree is ideal for supporting two or higher dimensional range queries. However, the non-clustered nature of R-trees makes them inadequate for inequality joins; we cannot avoid random I/O access when retrieving join results. B^+-tree is a clustered index. The bright side is that for each tuple, only sequential disk scan is required to retrieve relevant tuples. However, we need to repeat this n times, where n is the number of tuples, which is prohibitively expensive. When confronted with such problems, one common practice is to use space-efficient and CPU-friendly indices; in this paper, we employ bit-arrays.

As discussed earlier, the idea to handle an inequality join on one attribute is to leverage a sorted array, as shown in Example 1. When two different attributes appear in the join, in order to leverage the similar idea, a natural solution is to use a permutation array between two sorted arrays L_1 and L_2. Given the i-th element in L_1, a permutation array can tell its corresponding position in L_2 in constant time. Moreover, when visiting items in L_1, we need to keep track of the items we have

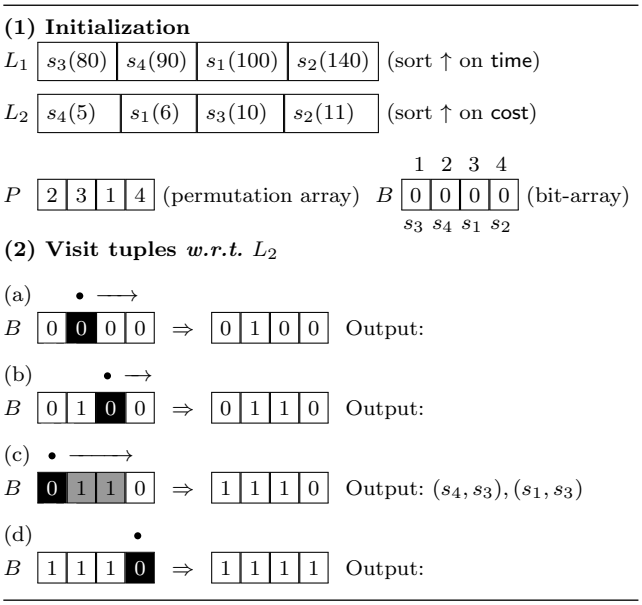


Fig. 2 IEJoin process for query Q_p

seen so far, for which we use a bit-array to make such a connection.

Generally speaking, our method, namely IEJoin, sorts relation west on time and cost, creates a permutation array for cost *w.r.t.* time, and leverages a bit-array to emit join results. We will briefly present the algorithm below, and defer detailed discussion to Section 3. Figure 2 depicts the process.

S1. [Initialization] Sort both time and cost values in ascending order, as depicted by L_1 and L_2, respectively. While sorting, compute a permutation (reordering) array of elements of L_2 in L_1, as shown by P. For example, the first element of L_2 (*i.e.,* s_4) corresponds to position 2 in L_1. Hence, $P[1] = 2$. Initialize a bit-array B with length n and set all bits to 0, as shown by B with array indices reported above the cells and corresponding tuples reported below them.

S2. [Visit tuples in the order of L_2] Scan the permutation array P and operate on the bit-array.

(a) *Visit* $P[1]$. First visit tuple s_4 (1^{st} element in L_2) and check in P what is the position of s_4 in L_1 (*i.e.,* position 2). Then go to $B[2]$ and scan all bits in positions higher than 2. As all $B[i] = 0$ for $i > 2$, there is no tuple that satisfies the join condition of Q_p *w.r.t.* s_4. Finish this visit by setting $B[2] = 1$, which indicates that tuple s_4 has been visited.

(b) *Visit* $P[2]$. This is for tuple s_1. It processes s_1 in a similar manner as s_4, without emitting any result.

(c) *Visit* $P[3]$. This visit corresponds to tuple s_3. Each non-zero bit on the right of s_3 (highlighted by grey cells) corresponds to a join result, because each marked cell corresponds to a tuple that pays less cost (*i.e.,* being visited first) but takes more time (*i.e.,* on the right side of its position). It thus outputs (s_4, s_3) and (s_1, s_3).

(d) *Visit* $P[4]$. This visit corresponds to tuple s_2 and does not return any result.

The final result of Q_p is the union of all the intermediate results from the above steps, *i.e.,* $\{(s_4, s_3), (s_1, s_3)\}$.

A few observations make our solution appealing. First, there are many efficient techniques for sorting large arrays, *e.g.,* GPUTeraSort [20]. In addition, after getting the permutation array, we only need to sequentially scan it once. Hence, we can store the permutation array on disk, in case there is not enough memory. Only the bit-array is required to stay in memory, to avoid random disk I/Os. Thus, to execute queries Q_s and Q_p on 1 billion tuples, theoretically, we only need 1 billion bits (*i.e.,* 125 MB) of memory.

3 Centralized Algorithms

In this section, we describe our inequality join algorithms using permutation arrays and bit-arrays. We start by discussing the case with two relations and operators in $\{<, >, \leq, \geq\}$, followed by describing the not equal operator (*i.e.,* $\neq$) (Section 3.1). We then discuss the special case of self-joins (Section 3.2). We close this section by some optimization techniques (Section 3.3).

3.1 IEJoin

We present our join algorithm with two inequality predicates involving two relations. We refer the reader to Section 4 on how we deal with more than two predicates and inequality joins over multiple relations.

Algorithm. The algorithm, IEJoin, is shown in Algorithm 1. 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 (lines 3-6), computes the permutation array (lines 7-8) and two offset arrays (lines 9-10). We defer details of computing the permutation arrays to Section 7. 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 sets up the bit-array (line 11) as well as the result set (line 12). In addition, it sets an offset variable to distinguish between the inequality operators with or without equality conditions (lines 13-14). It then visits the values in L_2 in the appropriate order, which is to sequentially scan the permutation array from left to right (lines 15-22). For each tuple visited in L_2, it first sets all bits for those t in T' whose Y' values are smaller than

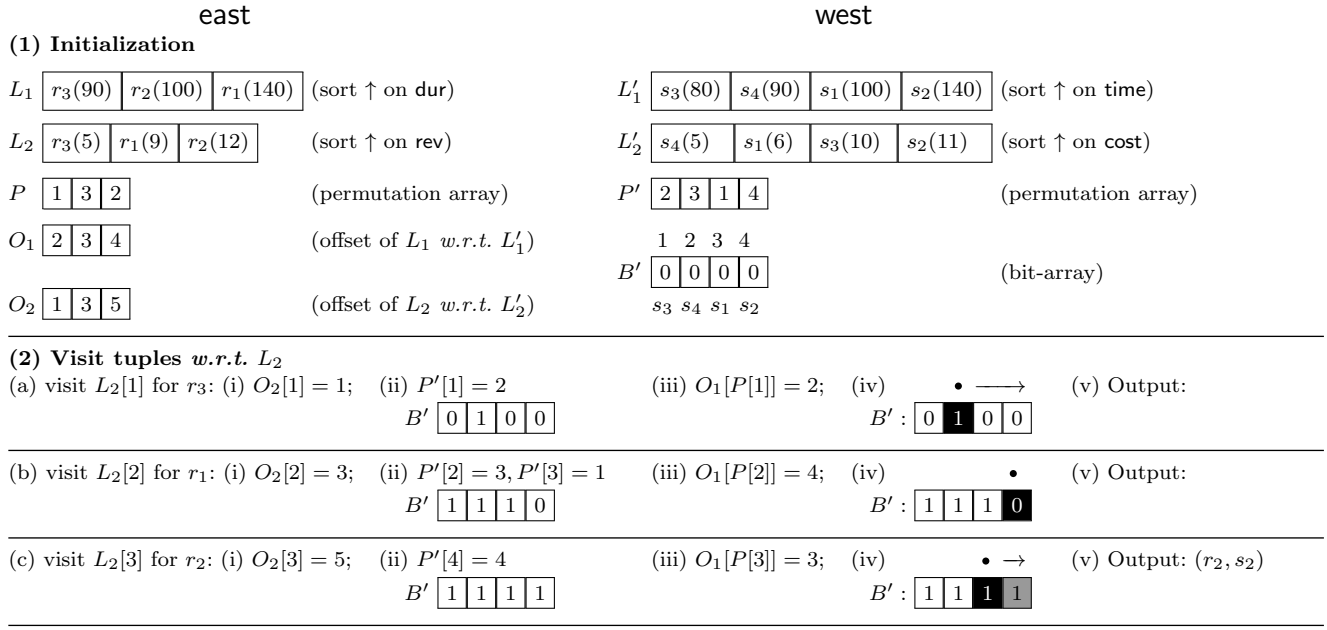


Fig. 3 IEJoin process for query Q_t

Algorithm 1: IEJoin

```
input : query Q with 2 join predicates t1.X op1 t2.X' and t1.Y op2 t2.Y', tables T, T' of sizes m and n resp.
output: a list of tuple pairs (ti, tj)
1  let L1 (resp. L2) be the array of X (resp. Y) in T
2  let L'1 (resp. L'2) be the array of X' (resp. Y') in T'
3  if (op1 ∈ {>, ≥}) sort L1, L'1 in descending order
4  else if (op1 ∈ {<, ≤}) sort L1, L'1 in ascending order
5  if (op2 ∈ {>, ≥}) sort L2, L'2 in ascending order
6  else if (op2 ∈ {<, ≤}) sort L2, L'2 in descending order
7  compute the permutation array P of L2 w.r.t. L1
8  compute the permutation array P' of L'2 w.r.t. L'1
9  compute the offset array O1 of L1 w.r.t. L'1
10 compute the offset array O2 of L2 w.r.t. L'2
11 initialize bit-array B' (|B'| = n), and set all bits to 0
12 initialize join_result as an empty list for tuple pairs
13 if (op1 ∈ {≤, ≥} and op2 ∈ {≤, ≥}) eqOff = 0
14 else eqOff = 1
15 for (i ← 1 to m) do
16     off2 ← O2[i]
17     for j ← 1 to min(off2, size(L2)) do
18         B'[P'[j]] ← 1
19     off1 ← O1[P[i]]
20     for (k ← off1 + eqOff to n) do
21         if B'[k] = 1 then
22             add tuples w.r.t. (L2[i], L'2[k]) to join_result
23 return join_result
```

the Y value of the current tuple in T (lines 16-18), *i.e.*, those tuples in T' that satisfy the second join condition. It then uses the other offset array to find those tuples in T' that also satisfy the first join condition (lines 19-22). It finally returns all join results (line 23).

Example 3 Figure 3 shows how Algorithm 1 processes Q_t (from Section 1). In initialization (step 1), L_1, L_2, L'_1 and L'_2 are sorted; P (resp. P') is the permutation array between L_1 and L_2 (resp. L'_1 and L'_2), where the details of computation are given in implementation details for PostgreSQL in Section 7.1 Figure 7; O_1 (resp. O_2) is the offset array of L_1 relative to L'_1 (resp. L_2 relative to L'_2), *e.g.*, $O_1[1] = 2$ means that the relative position of value $L_1[1] = 90$ in L'_1 is 2. Clearly, O_1 (resp. O_2) can be computed by sequentially scanning L_1 and L_1 (resp. L_2 and L'_2); and B' is the bit-array with all bits initialized to be 0.

After the initialization, the algorithm starts visiting tuples *w.r.t.* L_2 (step(2)). For example, when visiting the first item in L_2 (r_3) in step (2)(a), it first finds its relative position in L'_2 at step (2)(a)(i). Then it visits all tuples in L'_2 whose cost values are no larger than r_3[rev] at step (2)(a)(ii). Afterwards, it uses the relative position of r_3[dur] at L'_1 (step (2)(a)(iii)) to populate all join results (step (2)(a)(iv)). The same process applies to r_1 (step (2)(b)) and r_2 (step (2)(c)), and the only result is returned at step (2)(c)(v).

Correctness. The algorithm terminates and the results satisfy the join condition. For any tuple pair (r_i, s_j) that should be a result, s_j will be visited first and its corresponding bit is set to 1 (lines 17-18). Afterwards, r_i will be visited and the result (r_i, s_j) will be identified (lines 20-22) by the algorithm.

Complexity. Sorting arrays and computing their permutation array together are in $O(m \cdot log\, m + n \cdot log\, n)$ time, where m and n are the sizes of the two input relations (lines 3-8). Computing the offset arrays

will take linear time using sort-merge (lines 9-10). The outer loop will take $O(m \cdot n)$ time (lines 15-22). Hence, the total time complexity of the algorithm is $O(m \cdot log\, m + n \cdot log\, n + m \cdot n)$. It is straightforward to see that the total space complexity is $O(m + n)$.

Not equal operator. In the case of the not equal "$\neq$" operator, we simply rewrite the query as two queries, with the ($>$) and ($<$) operators, respectively. We then merge the results of these two queries through the UNION ALL SQL command.

Example 4 Consider the following query with one $\neq$ condition:

```
Q_k : SELECT r.id, s.id
      FROM Events r, Events s
      WHERE r.start ≤ s.end AND r.end ≠ s.start;
```

We translate Q_k into the union of two queries as follows:

```
Q'_k : SELECT r.id, s.id
       FROM Events r, Events s
       WHERE r.start ≤ s.end AND r.end < s.start
       UNION ALL
       SELECT r.id, s.id
       FROM Events r, Events s
       WHERE r.start ≤ s.end AND r.end > s.start;
```

The performance of IEJoin for queries with a "$\neq$" operator strongly depends on the attribute domain itself. Attributes with smaller ranges will typically lead to higher selectivity, while larger ranges to lower selectivity. Note that as higher selective attributes produce fewer results, compared to lower ones, they have a faster IEJoin runtime.

Outer joins. IEJoin can also support left, right, and full outer joins. For left outer joins, any tuple that does not find any matches while scanning the bit-array (lines 20-22 in Algorithm 1) is paired with a Null value. The right outer join is processed by flipping the order of input relations, executing a normal left outer join and then reversing the order of the join result. The IEJoin output should return $(L'_2[k], L_2[i])$ or $(NULL, L_2[i])$ instead of the original order (line 22 in Algorithm 1) to generate the correct result for the right outer join. The full outer join is translated into one left outer join that includes the normal IEJoin output, and one right outer join that only emits results with null values.

3.2 IESelfJoin

In this section, we present the algorithm for self-join queries with two inequality operators.

Algorithm. IESelfJoin (Algorithm 2) takes a self-join inequality query Q and returns a set of result pairs. The algorithm first sorts the two lists of attributes to be joined (lines 2-5), computes the permutation array (line 6), and sets up the bit-array (line 7) as well as the result set (line 8). It also sets an offset variable to distinguish inequality operators with or without equality (lines 9-10). It then visits the values in L_2 in the desired order, *i.e.,* sequentially scan the permutation array from left to right (lines 11-16). For each tuple visited in L_2, it needs to find all tuples whose X values satisfy the join condition. This is performed by first locating its corresponding position in L_1 via looking up the permutation array (line 12) and marked 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 they have been visited before (line 15). Such tuples will be joined with the tuple currently being visited as results (line 16). It finally returns all join results (line 17).

Note that the different sorting orders, *i.e.,* ascending or descending for attribute X and Y in lines 2-5, are chosen to satisfy various inequality operators. One may observe that if the database contains duplicated values, when sorting one attribute X, its corresponding value in attribute Y should be considered, and vice versa, in order to preserve both orders for correct join result. Hence, in IESelfJoin, when sorting X, we use an algorithm that also takes Y as the secondary key. Specifically, when some X values are equal, their sorting orders are decided by their Y values (lines 2-3), similarly for the other way around (lines 4-5). Note that if both attributes are duplicates, we make sure that the sorting of the attributes is consistent by relying on the inter-

Algorithm 2: IESelfJoin

```
   input  : query Q with 2 join predicates t1.X op1 t2.X
            and t1.Y op2 t2.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 ∈ {>, ≥}) sort L1 in ascending order
 3 else if (op1 ∈ {<, ≤}) sort L1 in descending order
 4 if (op2 ∈ {>, ≥}) sort L2 in descending order
 5 else if (op2 ∈ {<, ≤}) sort L2 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 ∈ {≤, ≥}) 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]]) to
               join_result
17 return join_result
```

		op_2 sorting order			
		$<$	$>$	$\leq$	$\geq$
op_1 sort order	$<$	Asc/Des	Des/Des	Asc/Asc	Des/Asc
	$>$	Asc/Asc	Des/Asc	Asc/Des	Des/Des
	$\leq$	Des/Des	Asc/Des	Des/Asc	Asc/Asc
	$\geq$	Des/Asc	Asc/Asc	Des/Des	Asc/Des

Table 1 Secondary key sorting order for Y/X in op_1/op_2

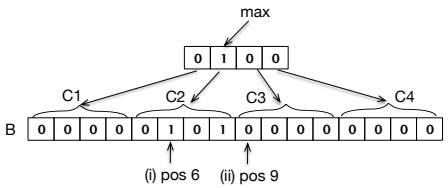


Fig. 4 Example of using a bitmap

nal record (tuple) ids. In particular, when the values are equal both in X and Y, we sort them in increasing order according to their internal id.

Moreover, we show in Table 1 the sorting orders for op_1's secondary key (Y) and op_2's secondary key (X) when the dataset contains duplicate values. For example, if op_1's condition is ($\leq$) and op_2's condition is ($>$), according to Table 1, equal values in op_1 are sorted based on ascending order of their Y values while op_2's equal values are sorted on descending order of their X values. Please refer to the example in Section 2 for query Q_p using IESelfJoin.

Correctness. It is easy to check that the algorithm will terminate and each result in join_result satisfies the join condition. For *completeness*, observe the following. For any tuple pair (t_1, t_2) that should be in the result, t_2 is visited first and its corresponding bit is set to 1 (line 13). Afterwards, t_1 is visited and the result (t_1, t_2) is identified (lines 15-16) by IESelfJoin.

Complexity. Sorting two arrays and computing their permutation array is in $O(n \cdot log\, n)$ time (lines 2-8). Scanning the permutation array and scanning the bit-array for each visited tuple run in $O(n^2)$ time (lines 11-16). Hence, in total, the time complexity of IESelfJoin is $O(n^2)$. It is easy to see that the space complexity of IESelfJoin is $O(n)$.

3.3 Enhancements

We discuss two techniques to improve performance: (i) Indices to improve the lookup performance for the bit-array. (ii) Union arrays to improve data locality and reduce the data to be loaded into the cache.

Bitmap index to improve bit-array scan. An analysis on both IEJoin and IESelfJoin shows that for each visited value (*i.e.,* lines 20-22 in Algorithm 1 and lines 14-16 in Algorithm 2), we need to scan all the bits on the right of the current position. When the query selectivity is high, this is unavoidable for producing the correct results. However, when the query selectivity is low, iteratively scanning a long sequence of 0's will be a performance bottleneck. We thus adopt a bitmap to guide which parts of the bit-array should be visited.

Given a bit-array B of size n and a predefined chunk size c, our *bitmap* is a bit-array with size $\lceil n/c \rceil$ where each bit corresponds to a chunk in B, with 1 indicating that the chunk contains at least a 1, and 0 otherwise.

Example 5 Consider the bit-array B in Figure 4. Assume that the chunk size $c = 4$. The bit-array B will be partitioned into four chunks C_1–C_4. Its bitmap is shown above B in the figure and consists of 4 bits. We consider two cases.

Case 1: visit $B[6]$, in which case we need to find all the 1's in $B[i]$ for $i > 6$. The bitmap tells that only chunk 2 needs to be checked, and it is safe to ignore chunks 3 and 4.

Case 2: visit $B[9]$, the bitmap tells that there is no need to scan B, since there cannot be any $B[j]$ where $B[j] = 1$ and $j > 9$.

To further improve the bitmap, we avoid unnecessary scanning by stopping at the maximum modified value in the bitmap index. For example, in Figure 4, we maintain a max scan index variable (*MaxIndex*) with value 2 to stop the bitmap from unnecessarily scanning bitmap indices 3 and 4. The initial value of this variable is 0, which means that both the bitmap and the bit-array are empty. As we iteratively modify the bit-array (line 18 in Algorithm 1 and line 13 in Algorithm 2) and edit the bitmap, we make sure to update the max scan index variable if the current updated filter index is larger than the last recorded max filter index.

Union arrays on join attributes. In testing Algorithm 1, we found that there are several cache loads and stores. A deeper analysis shows that the extra cache loads and stores may be caused by cache misses when sequentially visiting different arrays. Take Figure 3 for example. In step (2)(a), we visit arrays L_2, O_2, P', P and O_1 in sequence, with each causing at least one cache miss. Step (2)(b) and step (2)(c) show a similar behavior. An intuitive solution is to merge the arrays on join attributes and sort them together. Again, consider Figure 3. We can merge L_1 and L'_1 into one array and sort them, which will result in $\langle s_3(80), r_3(90), s_4(90), r_2(100), s_1(100), r_1(140), s_2(140)\rangle$. Similarly, we can merge L_2 and L'_2, and P and P'. Also, O_1 and O_2 are not needed in this case, and B' needs to be extended to be aligned with the merged arrays. This solution is similar to IESelfJoin (Section 3.2. However, we need to prune join results for tuples that come

Algorithm 3: IEJoin Selectivity Estimation

```
   input  : join predicates t1.X op1 t2.X' and
            t1.Y op2 t2.Y', sample tables Ts & Ts', block
            size n
   output: Number of overlapping blocks
1  overlappingBlocks ← 0
2  if (op1 ∈ {<, ≤}) then
3      sort Ts & Ts' in descending order on X and X'
4  else if (op1 ∈ {>, ≥}) then
5      sort Ts & Ts' in ascending order on X and X'
6  Partition Ts & Ts' into B&B' blocks of size n
7  for (i ← 1 to B) do
8      for (j ← 1 to B') do
9          if Ts_i ⋂ Ts'_j then
10             increment overlappingBlocks by 1
11 return overlappingBlocks
```

from the same table. This can be easily done using a Boolean flag for each position, where 0 (resp. 1) denotes that the corresponding value is from the first (resp. the second) table. Our experiments (Section 8.5) show that this simple union can significantly reduce the number of cache misses, and thus improve execution time.

4 Query Optimization

We introduce an approach to estimate the selectivity of inequality join predicates that is then used to optimize inequality join queries. In particular, we tackle the cases of selecting the best join predicate when joining two relations on more than two join predicates and selecting the best join order for a multi-way IEJoin. Note that our goal is not to build a full-fledged query optimizer and implement it in an existing system, but rather to propose a simple, yet efficient, approach for optimizing joins with inequality conditions.

4.1 Selectivity Estimation

To estimate the selectivity of an inequality join predicate, we simply count the number of overlapping sorted blocks obtained from a sample of the input tables. We assume that the join has two inequality predicates as in Algorithm 1. We first obtain a sample from the two tables to join, using uniform random sampling[2]. We give the samples as input to Algorithm 3. We then sort the two samples based on the attributes involved in one of the join predicates (lines 2-5), while using the remaining join attributes to sort duplicate tuples. Next, we divide each sample into smaller blocks (line 6) and test all combinations of block pairs for potential overlaps based on min/max values of all of the join attributes in each block (lines 7-10). The idea is that overlapping blocks may generate results for a given query while non-overlapping blocks cannot generate results at all. Finally, we return the number of overlapping blocks as the selectivity estimation of the inequality join (line 11). The lower the number of overlapping blocks, the higher the selectivity of the inequality join condition.

While we cannot determine the exact size of the join output in overlapping blocks before execution, sorting helps with good estimates. This is because sorting blocks naturally increases the locality of the input relations as we consider inequality join conditions. Thus, the number of overlapping blocks is a good approximation of the selectivity of an inequality join condition.

4.2 Join Optimization with Multiple Predicates

Given an inequality join query on two relations and with more than two join predicates, we need to determine which two predicates should be joined first to minimize the cost. The remaining predicates will be evaluated on the result of the join. To this end, we need to estimate the selectivity of all pair combinations of the join predicates in the query. For example, we compare the selectivity estimation of three pair combinations ($\bowtie_1$-$\bowtie_3$) for Q_w,

Q_w : SELECT s_1.t_id, s_2.t_id
FROM west s_1, west s_2
WHERE s_1.time $>$ s_2.time and s_1.cost $<$ s_2.cost
and s_1.totalUsers $<$ s_2.totalUsers;

as follows:

$\bowtie_1$: s_1.time $>$ s_2.time & s_1.cost $<$ s_2.cost
$\bowtie_2$: s_1.time $>$ s_2.time & s_1.totalUsers $<$ s_2.totalUsers
$\bowtie_3$: s_1.cost $<$ s_2.cost & s_1.totalUsers $<$ s_2.totalUsers

We need to test $\binom{N}{2}$ combinations to choose the best join predicates for a given query with N input join predicates. Clearly, a large N would increase the overhead of the selectivity estimation. To lower this overhead, we reuse the sorted input relations in multiple instances of Algorithm 3 that share a similar join predicate. For example, IEJoin can sort on attribute time (or totalUsers) to reuse it in join predicates ⟨time, totalUsers⟩ and ⟨time, cost⟩ (resp. ⟨time, totalUsers⟩ and ⟨cost, totalUsers⟩) as they both share attribute time (resp. totalUsers). Note that sorting based on one attribute or another does not impact the selectivity estimation of the pair combination.

4.3 Multi-way Join Optimization

For multi-way inequality joins, we follow a common approach in optimizing such joins, such as in [31]. Gener-

[2] We experimentally show in Section 8.7 that our algorithm requires only 1% of the input data to be accurate.

ally speaking, we execute a multi-way inequality join as a series of two-way inequality joins formed as a left-deep plan. We adopt a greedy approach where we choose the order of the two-way joins based on their estimated selectivity, *i.e.,* the two-way joins with the higher selectivity (lowest number of overlapping blocks as computed by Algorithm 3) are pushed down in the plan.

Algorithm 4 builds the execution plan for a multi-way join query. We first estimate the selectivity of all possible combinations of two-way joins using Algorithm 3 (lines 3-6). We discard Cartesian products. At level one of the left-deep plan, we pick the two relations with the most selective inequality join (lines 8-11). We then proceed by selecting the one relation, among the remaining ones, that would deliver the most selective inequality join if joined with the previous level in the plan (lines 12-18). This is performed by selecting the relation that has the smallest number of overlapped blocks when joined with a relation from the previous level. We only consider joins that share an inequality predicate with one relation in the previous levels (lines 14-16). In each level, we append the most selective two-way join to the final execution plan (lines 17-18) and remove it from the pool of available joins (line 19). By repeating this process until there is no more relations to join, we compute a full IEJoin order. Algorithm 4 returns an array that describes the left-deep plan. The plan is obtained by joining the relations in the order they appear in the array, *i.e.,* first join $plan[1]$ with $plan[2]$, then the result with $plan[3]$ and so on.

Algorithm 4: Multi-way IEJoin planner

```
   input  : input relations R
   output: join plan
1  n ← |R|
2  Est ← empty set
3  for (i ← 1 to n) do
4      for (j ← i + 1 to n) do
5          if R_i & R_j share IEJoin predicates then
6              Est ← selectivity estimation of R_i ⋈ R_j
7  for (planIndex ← 1 to n) do
8      if planIndex = 1 then
9          min ← smallest estimation ∈ Est
10         plan[1] ← R_i, where i ∈ min
11         plan[2] ← R_j, where j ∈ min
12     else if planIndex > 2 then
13         Est' ← empty set
14         foreach R_i ⋈ R_j estimation ∈ Est do
15             if (R_i|R_j) ∈ plan[1, planIndex − 1] and
               (R_i ∧ R_j) ∉ plan[1, planIndex − 1] then
16                 Est' ← selectivity estimation of R_i ⋈ R_j
17         min ← smallest estimation ∈ Est'
18         plan[planIndex] ← R_i (or R_j), where
           (i ∧ j) ∈ min and R_i (or R_j)
           ∉ plan[1, planIndex − 1]
19     remove min from Est
20 return plan
```

5 Incremental Inequality Joins

In this section, we present algorithms for *incremental inequality joins* when the data keeps changing with insertions and deletions. We compute $Q(D \oplus \Delta D)$, where Q is an inequality join on D that contains either one or two relations, and ΔD is the data updates that are insertions ΔD^+ or deletions ΔD^-. Adapting our algorithms for incremental computation faces two challenges: (i) maintaining the sorted arrays; and (ii) only computing results *w.r.t.* ΔD.

Maintain a sorted array. Given a sorted array L and an unsorted array ΔL, the problem is to compute a sorted array for elements in $L \oplus \Delta L$.

(1) *Sort-merge [27].* The straightforward solution is to first sort ΔL and then merge two sorted arrays L and ΔL in linear time. This approach is appropriate for *batch updates.*

(2) *Packed-memory array [5].* A widely adopted approach for maintaining a dynamic set of N elements in sorted order in $\Theta(N)$-sized array is to use Packed-memory array (PMA). The idea is to insert $\Theta(N)$ empty spaces among the elements of the array such that only a small number of elements need to be shifted around per update. Thus, the number of element moves per update is only $O(log_2 N)$. This approach is ideal for continuous query answering in a streaming fashion.

Note that, PMA leaves empty spaces in the array. For instance, the arrays L_1 and L_2 in Figure 5 are stored using PMAs, where "|"'s denote the empty spaces. Also, PMAs handle deletions [5]. Although we use PMAs to maintain the sorted arrays by leaving gaps (*i.e.,* the "|"'s) for future updates, our algorithms are unchanged by simply ignoring these gaps.

Let us start by discussing the incremental inequality join on one relation with insertions. We will then discuss the case with deletions, and then on two relations.

Incremental IESelfJoin**.** We illustrate the idea of computing incremental results on one relation with *insertions* by an example.

Example 6 Consider query Q_p from Example 2 and the data T used in Figure 2. Computing $Q_p(T)$ is the same as described in Figure 2, shown as step (1) in Figure 5. Consider a new tuple insertion $\Delta D^+ = \{s_5(95, 8)\}$. As shown in Figure 5 step (2), it finds the right positions

in both sorted arrays, *i.e.,* 95 in position 4 of L'_1, 8 in position 4 of L'_2 and the permutation array is updated.

Similar to the process described in Section 2, it visits tuples in L'_2 from left to right (Figure 5 step (3)).

(a) For all tuples whose time values are less than s_5 that is 8, set all corresponding bits as 1's since they satisfy one join condition on attribute time.

(b) Visit s_5 and output all results for (s_5, s_i) whose s_i is on the right of s_5 in B (*i.e.,* satisfying both join conditions), which is $\{(s_5, s_1)\}$.

(c) For other tuples on the right of s_5 in L'_2, output a new join result if it contains the new tuple s_5. When visiting s_3, output $\{(s_3, s_5)\}$.

(d) When visiting s_2, the process is similar to (c), with an empty output.

From the above example, we can see that only new results are produced, *i.e.,* those coming from ΔD^+. The incremental algorithm using PMA is a simple adaption of Algorithm 2, which is thus omitted here. Moreover, when the update contains a set of insertions, the procedure for each one is the same as described in Example 6.

For *deletions* ΔD^-, we use a similar methodology as discussed above to compute updated results. The difference is that, instead of adding these new results, we remove them from old results.

Incremental IEJoin**.** We now discuss how to extend Algorithm 1 on two relations R and S to support incremental processing. We will focus on insertions only (ΔR^+ and ΔS^+), since deletions are similar to insertions by only removing results.

Take Figure 3 for reference, we perform the following three steps to run IEJoin: (1) maintain the sorted lists *w.r.t.* the insertions ΔR^+ and ΔS^+; (2) maintain the offsets for $R \oplus \Delta R^+$ relative to $S \oplus \Delta S^+$; and (3) compute the new results.

Step (1) is the same as discussed in IESelfJoin. Step (2) can be performed in a way similar to the sorted merge join by linearly scanning both sorted arrays in $R \oplus \Delta R^+$ and $S \oplus \Delta S^+$ to set the corresponding offsets. Step (3) is to run IEJoin by only outputting results related to ΔR^+ or ΔS^+, *i.e.,* (ΔR^+, S), (R, ΔS^+) or (ΔR^+, ΔS^+). If insertions are only ΔR^+, IEJoin ends by visiting the last element of ΔR^+ in the bit-array. Otherwise, it continues until all tuples in $R \oplus \Delta R^+$ are visited. When processing both insertions and deletions, results from (ΔR^+,ΔS^-) or (ΔR^-,ΔS^+) are ignored.

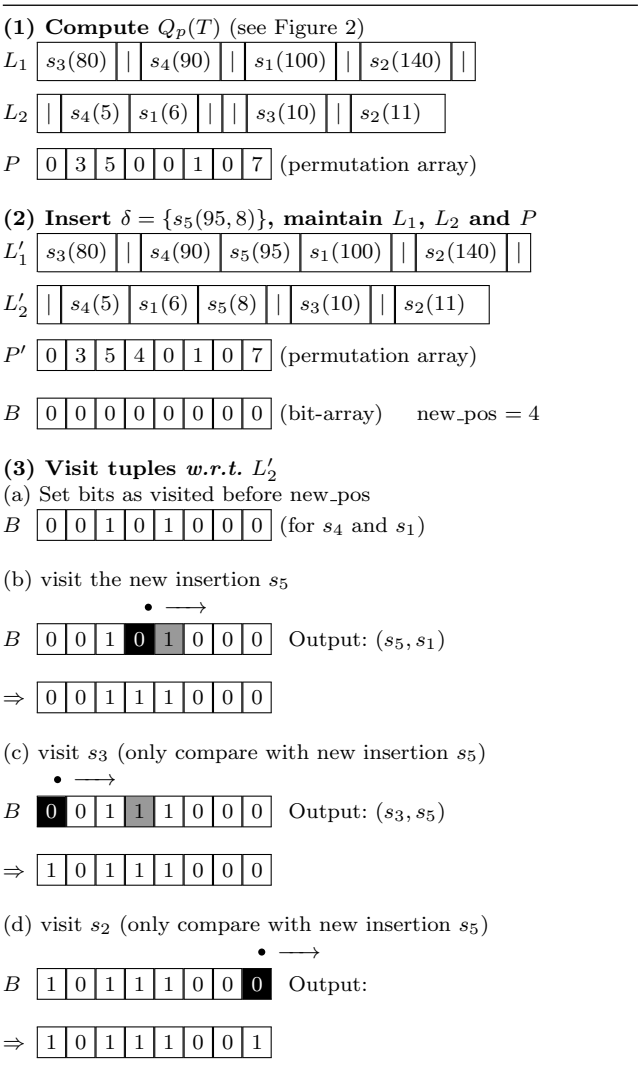


Fig. 5 Incremental IESelfJoin process for query Q_p

6 Scalable Inequality Joins

We present a scalable version of IEJoin along the same lines of state-of-the-art general purpose distributed data processing systems, such as Hadoop's MapReduce [12] and Spark [37]. Our goal is twofold: (i) scale our algorithm to very large input relations that do not fit into the main memory of a single machine and (ii) minimize processing overheads to improve the efficiency even further.

A simple approach for scaling IEJoin is to (i) construct k data blocks of each input relation, (ii) apply Cartesian product (or self-Cartesian product for a single relation input) on the data blocks, and (iii) run IEJoin (either on a single table or two tables input) on all remaining data block pairs (up to k^2). This approach suffers from a high processing overhead caused by the excessive block replication. In particular, this excessive data replication increases the CPU overhead by scheduling a large number of tasks and redundantly processing replicated blocks (*i.e.,* sorting identical blocks in different threads). It also causes high

memory overheads where different tasks maintain identical copies of the same data. In distributed settings, this approach additionally causes large network overheads as it transfers identical copies of the data to different workers. A naïve solution would be to reduce the number of blocks by increasing their size. However, very large blocks introduce work imbalance and require larger memory for each worker.

6.1 Scalable IEJoin

We solve the above challenges through efficient *pre-processing* and *post-processing* phases that reduce data replication by minimizing the number of required data block pairs. We achieve this by pruning unnecessary data block pairs early before wasting any resources. The pre-processing phase generates space-efficient data blocks for the input relation(s), predicts which pair of data blocks may report query results, and only materializes useful pairs of data blocks. IEJoin, in its scalable version, returns the join results as a pair of tupleIDs instead of returning the actual tuples. It is the responsibility of the post-processing phase to materialize the final results by resolving the tupleIDs into actual tuples. We use the internal tupleIDs of existing systems to uniquely identify different tuples. We summarize in Algorithm 5 the implementation of the scalable algorithm when processing two input tables.

Algorithm 5: Scalable IEJoin

```
   input : Query Q with 2 join predicates t1.X op1 t2.X'
           and t1.Y op2 t2.Y', Table t1,Table t2
   output: Table t_out
1  //Pre-processing
2  foreach tuple r ∈ t1 and t2 do
3      r ← global unique ID
4  DistT1 ← sort t1 on t1.X and partition to n blocks
5  DistT2 ← sort t2 on t2.X' and partition to m blocks
6  for (i ← 1 to n) do
7      D1_i ← all X and Y values in DistT1_i
8      MT1_i ← min and max of X and Y in D1_i
9  for (j ← 1 to m) do
10     D2_j ← all X' and Y' values in DistT2_j
11     MT2_j ← min and max of X' and Y' in D2_j
12 Virt ← MT1 × MT2
13 forall the (MT1_i,MT2_j) pairs ∈ Virt do
14     if MT1_i ⋂ MT2_j then
15         (MT1_i, MT2_j) ← (D1_i, D2_j)
16     else
17         Remove (MT1_i, MT2_j) from Virt
18 //IEJoin function
19 forall the block pairs (D1_i, D2_j) ∈ Virt do
20     TupleIDResult ← IEJoin(Q,D1_i,D2_j)
21 //Post-processing
22 forall the tupleID pairs (v, w) ∈ TupleIDResult do
23     tuple_v ← tuple id v in DistT1
24     tuple_w ← tuple id w in DistT2
25     t_out ← (tuple_v,tuple_w)
```

Pre-processing. After assigning unique tupleIDs to each input tuple (lines 2-3), the pre-processing step globally sorts each relation on a common attribute of one of the IEJoin predicates (*i.e.,* salary in Q_1). Then, it partitions each sorted relation to $k = \lceil \frac{M}{b} \rceil$ equally-sized partitions, where M is the relation input size and b is the default block size (lines 4-5). Note that global sorting before partitioning maximizes data locality within partitions, which in turn decreases the overall runtime. This is because global sorting partially answers one of the inequality join conditions, where it physically moves tuples closer to their candidate pairs. In other words, global sorting increases the efficiency of block pairs that generate results, while block pairs that do not produce results can be filtered out before actually processing them. After that, for each sorted partition, we generate a single data block that stores only the attribute values referenced in the join conditions in a list. Following the semi-join principle, these data blocks do not store the entire tuples thus allowing to reduce the memory, disk I/O, and network overheads. We also extract metadata that contain the block ID and the min/max values of each referenced attribute value from each data block (lines 6-11). Then, we create $n_{MT1} \times m_{MT2}$ virtual block combinations and filter out block combinations with non-intersecting min-max values since they do not produce results (lines 12-17). Figure 6 illustrates the pre-processing of two relations R and S. It starts by sorting and partitioning R into three blocks and S into two blocks. It then generates the metadata blocks and executes Cartesian product on all metadata blocks. Next, it removes all non-overlapping blocks. Finally, it recovers the original content for all overlapping metadata blocks, *i.e.,* μR_1 recovers its data from block R_1.

IEJoin. We now have a list of overlapping block pairs. We simply run an independent IEJoin for each of these pair blocks in parallel. Specifically, we merge the attribute values in $D1$ and $D2$ and run IEJoin over the merged block. The permutation and bit arrays generation are similar to the centralized version. However, the scalable IEJoin does not have access to the actual relation tuples. Therefore, each parallel IEJoin instance outputs a pair of tupleIDs that represents the joined tuples (lines 19-20).

Post-processing. In the final step, we materialize the result pairs by matching each tupleID-pair from the output of the distributed IEJoin with the tupleIDs of

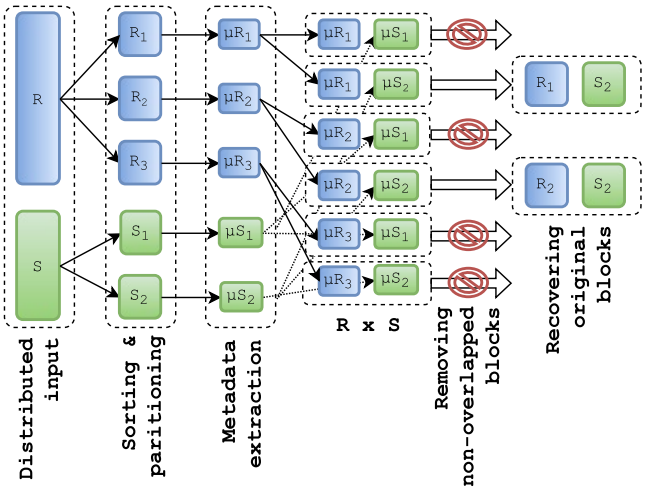


Fig. 6 An example of the IEJoin pre-processing stage with two relations R and S

*DistT*1 and *DistT*2 (lines 22-25). We run this post-processing phase in parallel, as a scalable hash join based on the tupleIDs, to speed up the materialization of the final join results.

6.2 Multithreaded and Distributed IEJoin

The scalable solution we just described can be adapted to a multithreaded setting as follows. For the pre-processing phase, we sort and partition inputs in parallel. If the input does not fit into memory, we apply external sorting. Next, block metadata are extracted by all threads, while each thread examines an independent set of block metadata pairs to eliminate non-overlapping blocks in parallel. The remaining block metadata pairs are then materialized, from either a cached copy in memory or from disk, into block pairs. Each thread then applies the IEJoin algorithm on a different block pair to generate partial join results. Once the complete result is computed, we materialize the output using multi-threaded hash-based join. In a distributed setting, we follow the same process, but we use compute nodes as the main processing units instead of threads. We discuss in Section 7.2 our distributed version on top of Spark SQL.

7 Integration into Existing Systems

We describe the integration of our algorithms into three existing systems: PostgreSQL, a popular open-source DBMS (Section 7.1); Spark SQL, a popular SQL-like engine on top of Spark (Section 7.2); and Nadeef [24], a distributed data cleaning system (Section 7.3).

7.1 PostgreSQL

PostgreSQL processes queries in three stages: *parsing*, *planning*, and *execution*. Parsing extracts relations and predicates and creates query parse trees. Planning creates query plans and invokes the query optimizer to select a plan with the smallest estimated cost. Execution runs the selected plan and emits the output.

Parsing and Planning. PostgreSQL uses merge and hash join operators for equijoins and naïve nested loop for inequality joins. PostgreSQL looks for the most suitable join operator for each join predicate. We extend this check to verify that a join is IEJoin-able by checking if a predicate contains a scalar inequality operator. If it is the case, we save the operator's oid in the data structure associated with the predicate. For each operator and ordered pair of relations, we create a list of predicates that the operator can handle. For example, two equality predicates over the same pair of relations are associated to one hash join. In the presence of inequality joins, we make sure that the PostgreSQL optimizer chooses our IEJoin algorithm, while optimizations for other operators are done by PostgreSQL optimizer on top of the optimized IEJoin.

Next, the Planner estimates the execution cost for possible join plans. In the presence of both equality and inequality joins, the optimizer will delay all inequality joins as they are usually less selective than equality joins. More specifically, every node in the plan has a base cost, which is the cost of executing the previous nodes, plus the cost for the actual node. We added a cost function for our operator; it is evaluated as the sum of the cost for sorting inner and outer relations, CPU cost for evaluating all output tuples (approximated based on PostgreSQL's default inequality joins estimation), and the cost of evaluating additional predicates for each tuple (*i.e.,* the ones that are not involved in the actual join). Next, PostgreSQL selects the plan with the lowest cost.

Execution. The executor stores incoming tuples from outer and inner relations into arrays of type *TupleTableSlot*, which is PostgreSQL's default data structure that stores the relation tuples. These copies of the tuples are required as PostgreSQL may not have the content of the tuple at the same pointer location when the tuple is sent for the final projection. This step is a platform-specific overhead that is required to produce an output. The outer relation (of size N) is parsed first, followed by the inner relation (of size M). If the inner join data is identical to the corresponding outer join data (self-join), we drop the inner join data and the data structure has size N instead of $2N$. If there are

Sort1	idx	time	cost	pos
s_1	3	80	10	1
s_2	4	90	5	2
s_3	1	100	6	3
s_4	2	140	11	4

Sort2	idx	time	cost	pos
s_1	4	90	5	2
s_2	1	100	6	3
s_3	3	80	10	1
s_4	2	140	11	4

Fig. 7 Permutation array creation for self-join Q_p

more than two IEJoin predicates, then we follow the procedure explained in Section 4.2, *i.e.,* we pass to the algorithm the pair of predicates with the highest selectivity.

We illustrate in Figure 7 the data structure and the permutation array computation with an example for self-join Q_p. We initialize the data structure with an index (idx) and a copy of the attributes of interest (time and cost for Q_p). Next, we sort the data on the first predicate (time) using system function *qsort* with special comparators (defined in Algorithm 1) to handle cases where two values for a predicate are equal. The result of the first sort is reported at the left-hand side of Figure 7. The last column (pos) is now filled with the ordering of the tuples according to this sorting. As a result, we create a new array to store the index values for the first predicate. We use this array to select tuple IDs at the time of projecting tuples. The tuples are then ordered again according to the second predicate (cost), as reported in the right-hand side of Figure 7. After the second sorting, the new values in pos are the values for the permutation array.

Finally, we create and traverse a bit-array B of size $(N + M)$ (N in case of self-join) along with a bitmap, as discussed in Section 3.3. If the traversal finds a set bit, the corresponding tuples are sent for projection. Predicates, not selected by the optimizer in the case of multi-predicate IEJoin, are evaluated at this stage and, if the conditions are satisfied, tuples are projected.

7.2 Spark SQL

Spark SQL [4] allows users to query structured data on top of Spark [37]. It stores the input data as a set of in-memory Resilient Distributed Datasets (RDD). Each RDD is partitioned into smaller cacheable blocks, where each block fits in the memory of a single machine. Spark SQL takes as input the datasets location(s) in HDFS and an SQL query, and outputs an RDD that contains the query result. The default join operation in Spark SQL is inner join. When passing a join query to Spark SQL, the optimizer searches for equality join predicates that can be used to evaluate the inner join operator as a hash-based physical join operator. If there are no equality join predicates, the optimizer translates the inner join physically to a Cartesian product followed by a selection predicate.

We implemented the distributed version of IEJoin as a new Spark SQL physical join operator. To make the optimizer aware of the new operator, we added a new rule to recognize inequality conditions. The rule passes all inequality conditions to the IEJoin operator. If the operator receives more than two inequality join conditions, it deploys the IEJoin optimizer to find the two highest selective inequality conditions. It executes the join using such conditions and evaluates the rest of the join conditions as a post selection operation on the output. Similar to the PostgreSQL case, in the presence of both equality and inequality joins, it orders inequality joins after all equality joins. The distributed operator utilizes Spark RDD operators to run both the IEJoin and its optimizer. As a result, distributed IEJoin depends on Spark's default memory management to partition and store the user's input relation. If the result does not fit in the memory of a single machine, we temporarily store the result into HDFS. After all IEJoin instances finish writing into HDFS, the distributed operator passes the HDFS file to Spark, which constructs a new RDD of the result and passes it to Spark SQL.

Figures 8 and 9 show how the distributed IEJoin is processed in Spark. First, we globally sort the two relations using Spark RDD sort (Figure 8(a)). Next, we generate a set of distributed data blocks for each relation through the RDD `mapPartitionsWithIndex()` function (Figure 8(b)). As described in Section 6, the block transformation does not store the actual tuples; it only stores the attributes in the join predicates. We then transform the data blocks into a metadata blocks using their statistics (Figure 8(c)). Afterwards, we apply Cartesian product on the block metadata of R and S and remove non-overlapping block metadata through RDD `filter()` operator (Figure 8(d)). Next, we join the remaining blocks' metadata with the original data blocks to recover their content (Figure 8(e)) through two RDD `join()` operators; one for each input relation. We then apply an independent instance of IEJoin on every block pair in parallel by using RDD `flatMapToPair()` operator (Figure 9(a)). Finally, we join the IEJoin result with the original relations R and S, using two RDD `join()` operators, to recover the full attribute information of the result (Figure 9(b)).

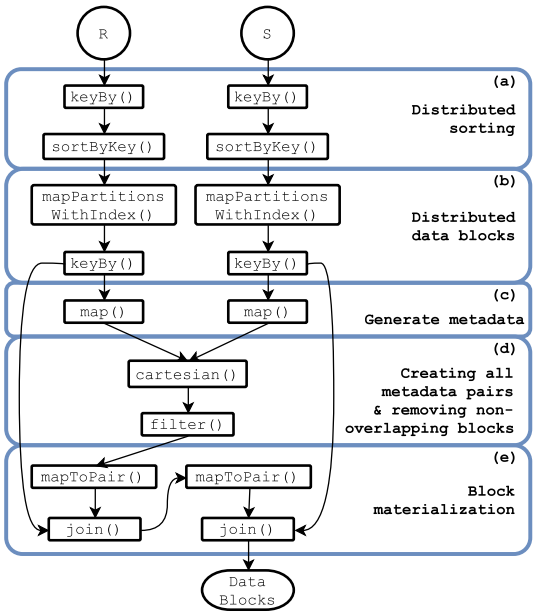


Fig. 8 Spark's DAG for the pre-processing phase

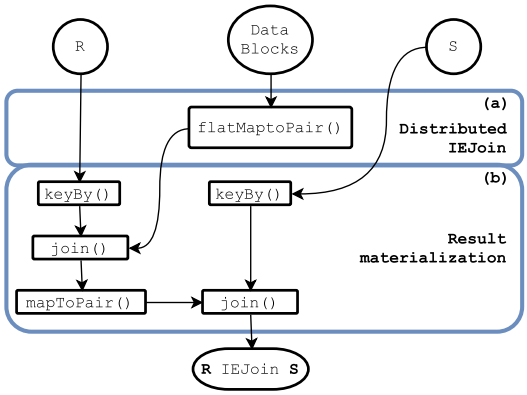


Fig. 9 Spark's DAG for IEJoin the post-processing phase

7.3 NADEEF

NADEEF [11, 15, 16, 24] is a generalized data cleaning system[3], implemented on top of PostgreSQL and Rheem [3]. Rheem is a data processing framework that provides independence from and interoperability among existing data processing platforms. Given a dirty dataset where errors are detected as violations of some data quality rules (*e.g.,* integrity constraints), NADEEF outputs a data instance free of violations. NADEEF operates in two phases: violation detection and data repair. In the detection phase, it finds all data errors *w.r.t.* the data quality rules; in the repair phase, it corrects data errors by using a repair algorithm, *i.e.,* an algorithm that updates the data instance to make it consistent *w.r.t.* the rules.

[3] http://github.com/daqcri/NADEEF

Dataset	Number of tuples	Size
Employees	10K – 500M	300KB – 17GB
Employees2	1B – 8B	34GB – 287GB
Events	10K – 500M	322KB – 14GB
Events2	1B – 6B	32GB – 202GB
MDC	24M	2.4GB
Cloud	470M	28.8GB
Grades	100K	1.7MB

Table 2 Size of the datasets

In the data violation detection phase, NADEEF has to generate all possible violation candidates for verification. We can see this candidate generation like a self-join operation as it has to check each input tuple for possible violations with other tuples. For example, a rule might state that given a dataset for employees in the same State, for every two distinct individuals, the one earning a lower salary should have a lower tax rate. In this case, NADEEF avoids performing a cross product by using a distributed sort-merge join-like approach [24]. In a nutshell, it first range partitions the input dataset and sorts each of the resulting range partitions in order to sort-merge join overlapping data partitions in a distributed fashion. It is in this last joining step that we integrate the IEJoin algorithm. That is, instead of performing a simple sort-merge join, we perform our algorithm as explained in Section 3. More specifically, we implemented the distributed version of IEJoin in NADEEF by (i) extending the Rheem framework in order to expose the new join as a physical operator, and by (ii) implementing the distributed version of IEJoin as an execution operator for Spark. Notice that the IEJoin execution operator is similar to its implementation in Spark SQL discussed in Section 7.2. In addition, we extended the NADEEF optimizer to take this new join operator into consideration.

8 Experimental Study

We evaluate IEJoin along several dimensions: (i) Effect of sorting and caching (Section 8.5); (ii) IEJoin in a centralized environment (Section 8.6); (iii) Query optimization techniques (Section 8.7); (iv) Incremental algorithms (Section 8.8); and (v) IEJoin in a distributed environment (Section 8.9).

8.1 Datasets

We used both synthetic and real-world data (summarized in Table 2) to evaluate our algorithms.

(1) EMPLOYEES contains employees' salary and tax information [6] with eight attributes: state, married, dependents, salary, tax, age, and three others for notes.

The relation has been populated with real-life data for tax rates, income brackets, and exemptions which were then used to generate synthetic tax records. EMPLOYEES2 in Table 2 is a group of larger input datasets with up to 6 Billion records, but with only 0.001% random changes to tax values. We lower the percentage of changes to test the distributed algorithm on large input files while avoiding extremely large output files.

(2) EVENTS is a synthetic dataset that contains start and end time information for a set of independent events. Each event contains the name of the event, event ID, the number of attending people, and the sponsor ID. To make sure we generate output for a given query, we extended end values for 10% random events. EVENTS2 contains larger datasets with up to 6 Billion records and 0.001% extended random events.

(3) MOBILE DATA CHALLENGE (MDC) is a 50GB real dataset [26, 28] that contains behavioral data of nearly 200 individuals collected by Nokia Research (`https://www.idiap.ch/dataset/mdc`). The dataset contains physical locations, social interactions, and phone logs of the participating individuals.

(4) CLOUD [33] is a real dataset that contains cloud reports from 1951 to 2009, through land and ship stations (`ftp://cdiac.ornl.gov/pub3/ndp026c/`).

(5) GRADES is a synthetic dataset for testing the not equal ($\neq$) join predicate. It is composed of a list of students with attributes id, name, gender, grade, and age. We encode gender values by using numeric values to be able to compare them using conditions less than ($<$) and greater than ($>$). The dataset size is fixed (100K tuples), but we vary the ratio of female students between 0.01% and 50% to diversify the selectivity of the query.

8.2 Algorithms

We compare our algorithms with several centralized as well as distributed algorithms

Centralized Systems. For our centralized setting, we use the following systems:

(1) C++ IEJOIN. This is a standalone C++ implementation of IEJOIN to run experiments that cannot be executed within a DBMS. For example, we use this implementation to evaluate the incremental experiments of IEJOIN (Section 8.8). The implementation uses optimized data structures from the Boost library[4] to maximize the performance gain of IEJOIN through a faster array scanning.

[4] http://www.boost.org/

(2) PG-IEJOIN. We implemented IEJOIN inside PostgreSQL v9.4, as discussed in Section 7.1. We compare it against the baseline systems below.

(3) PG-ORIGINAL. We used PostgreSQL v9.4 as a baseline. We tuned it with *pgtune* to maximize the benefit from large main memory.

(4) PG-BTREE & PG-GiST. We also used two variants of PostgreSQL using indices. PG-BTREE uses a B-tree index for each attribute in a query. PG-GiST uses the GiST access method in PostgreSQL, which considers arbitrary indexing schemes and automatically selects the best technique for the input relation.

(5) MONETDB. We used MonetDB Database Server Toolkit v1.1 (Oct2014-SP2), an open-source column-oriented database, in a disk partition of size 669GB.

(6) DBMS-X. We used a leading commercial centralized relational database.

Distributed Systems. We used the following systems:

(1) SPARK SQL-IEJOIN. We implemented IEJOIN inside Spark SQL v1.0.2 (`https://spark.apache.org/sql/`) as detailed in Section 7.2. We evaluated the performance of IEJOIN against the baseline systems below.

(2) SPARK SQL & SPARK SQL-SM. Spark SQL is the default implementation in Spark SQL. Spark SQL-SM [24] is an optimized version based on distributed sort-merge join (Section 7.3). We also improve the above method by pruning the non-overlapping partitions to be joined.

(3) DPG-BTREE & DPG-GiST. We used a commercial version of PostgreSQL with distributed query processing. This allows us to compare SPARK SQL-IEJOIN to a distributed version of PG-BTREE and PG-GiST.

8.3 Queries

We evaluate our algorithms from different perspectives and using several queries with inequality join conditions. It is worth noting that our main goal in the experiments is to show the value of optimizing inequality queries using our approach irrespective of the presence of other operators.

For our first experiment, we used the following self-join query over Employees to find violations of a data quality rule [10]:

Q_1 :
```
SELECT r.id, s.id
FROM Employees r, Employees s
WHERE r.salary < s.salary AND r.tax > s.tax;
```

The query returns a set of employee pairs, where one employee earns higher salary than the other but pays less tax. To make sure that we generate output for Q_1,

we selected 10% random tuples and increased their tax values. We also used a self-join query that collects pairs of overlapping events (in the Events dataset):

Q_2 : SELECT r.id, s.id
FROM Events r, Events s
WHERE r.start $\leq$ s.end AND r.end $\geq$ s.start
AND r.id $\neq$ s.id;

We extended end values for 10% random events to make sure we generate output for Q_2. Throughout all of our experiments, we either use these two queries or slightly modified versions thereof. For example, we added a third join predicate to Q_1 over age to get Q_1' and extended Q_1 and Q_2 to get Q_{mw} to study how our algorithms deal with *multi-predicate conditions* and with *multi-way joins*, respectively.

Q_1' : SELECT r.id, s.id
FROM Employees r, Employees s
WHERE r.salary $<$ s.salary AND r.tax $>$ s.tax
AND r.age $>$ s.age;

Q_{mw} : SELECT count(*)
FROM R r, S s, T t, V v, W w
WHERE r.salary $>$ s.salary AND r.tax $<$ s.tax
AND s.start $\leq$ t.end AND s.end $\geq$ t.start
AND t.salary $>$ v.salary AND t.tax $<$ v.tax
AND v.start $\leq$ w.end AND v.end $\geq$ w.start;

We also used slightly modified versions of Q_1 and Q_2 for our comparison with baselines using indexes. This is because although Q_1 and Q_2 appear to be similar, they require different data representation to be indexed with GiST. The inequality attributes in Q_1 are independent, each condition forms a single open interval, while in Q_2 they are dependent, together they form a single closed interval. Thus, we convert salary and tax attributes into a single geometric point data type SalTax to get Q_{1i}. Similarly for Q_2, we convert start and end attributes into a single range data type StartEnd to get Q_{2i}.

Q_{1i} : SELECT r.id, s.id
FROM Employees r, Employees s
WHERE r.SalTax $>^{\wedge}$ s.SalTax
AND r.SalTax $\gg$ s.SalTax;

Q_{2i} : SELECT r.id, s.id
FROM Events r, Events s
WHERE r.StartEnd && s.StartEnd AND r.id $\neq$ s.id;

In the rewriting of these queries for PG-GiST, operator "$>^{\wedge}$" corresponds to "*is above?*", operator "$\gg$" means "*is strictly right of?*", and operator "&&" indicates "*overlap?*". For geometric and range types, GiST uses a Bitmap index to optimize its data access with large datasets.

In addition to the above two main queries, we used Q_3 to evaluate our algorithms when producing different output sizes. This query looks for all persons that are close to a shop up to a distance c along the x-axis (xloc) and the y-axis (yloc):

Q_3 : SELECT s.name, p.name
FROM Shops s, Persons p
WHERE s.xloc $-$ c $<$ p.xloc AND s.xloc $+$ c $>$ p.xloc
AND s.yloc $-$ c $<$ p.yloc AND s.yloc $+$ c $>$ p.yloc;

We also used a self-join query Q_4, similar to Q_3, to compute all stations within distance $c = 10$ for every station. Since the runtime in Q_3 and Q_4 is dominated by the output size, we mostly used them for scalability analysis in the distributed case.

Furthermore, we used query Q_5 for our *not equal join predicates* experiment.

Q_5 : SELECT r.name, s.name FROM Grades r, Grades s
WHERE r.gender $\neq$ s.gender AND r.grade $>$ s.grade;

Notice that PostgreSQL executes Q_5 with nested loops. In our solution, the query is rewritten to Q_5'.

Q_5' : SELECT r.name, s.name FROM Grades r, Grades s
WHERE r.gender $<$ s.gender AND r.grade $>$ s.grade
UNION ALL
SELECT r.name, s.name FROM Grades r, Grades s
WHERE r.gender $>$ s.gender AND r.grade $>$ s.grade;

As attribute gender in Q_5 has only two distinct values, it might not provide the complete picture of the performance of ($\neq$) IEJoins. Thus, we additionally used Q_6 and Q_7 to analyze the effect of non-binary attributes on IEJoin with ($\neq$) as the join predicates.

Q_6 : SELECT r.name, s.name FROM Grades r, Grades s
WHERE r.age $\neq$ s.age AND r.grade $>$ s.grade;

Q_7 : SELECT r.name, s.name FROM Grades r, Grades s
WHERE r.gender $\neq$ s.gender AND r.grade $\neq$ s.grade;

8.4 Setup

For the centralized evaluation, we used a Dell Precision T7500 with two 64-bit quad-core Intel Xeon X5550 and 58GB RAM. For our distributed experiments, we used a cluster of 17 Shuttle SH55J2 machines (1 master with 16 workers) with Intel i5 processors with 16GB RAM, and connected to a high-end 1 Gigabit switch. For both settings, all arrays are stored in memory.

8.5 Parameter Setting

We start our experimental evaluation by showing the effect of the two optimizations (Section 3.3), as well as the effect of global sorting (Section 6).

Bitmap. Bitmaps are used when big array scanning is expensive. The chunk size is an optimization parameter that is machine-dependent. We run query Q_2 on 10M tuples with size 322MB to show the performance gain of using a bitmap based on three different implementations of IEJoin: the centralized C++, the C implementation of PostgreSQL, and the Java implementation of

Chunk size (bits)	C++ (Sec)	PostgreSQL (Sec)	Spark SQL (Sec)
1	> 1 day	> 1 day	> 1 day
64	2333	3139	1623
256	**558**	**817**	**896**
1024	**158**	**242**	**296**
4096	**81**	**117**	**158**
16384	139	142	232

Table 3 Bitmaps on 10M tuples (Events data)

Max Index?	C++ (Sec)	PostgreSQL (Sec)	Spark SQL (Sec)
No	81	117	158
Yes	23	47	46

Table 4 Bitmaps on 10M tuples with max scan index optimization (Events data)

Spark SQL. For Spark SQL, we only ran a single instance of IEJoin in a centralized setting.

Results are shown in Table 3. Intuitively, the larger the chunk size, the better. However, a very large chunk size defeats the purpose of using bitmaps to reduce the bit-array scanning overhead. The experiment shows that the performance gain is 3X between 256 bits and 1,024 bits and around 1.8X between 1,024 bits and 4,096 bits. Larger chunk sizes show worse performance, as shown with chunk size of 16,384 bits. The experiment in Table 4 shows the performance of the three implementations of IEJoin when adding the **max scan index** optimization to the bitmap. This index optimization improves the performance of IEJoin around 3.5 times compared to the best results achieved by the bitmap in Table 3.

Union arrays. To study the impact of the union optimization, we run IEJoin with and without the union array using 10M tuples in the Events dataset. We collect the following statistics with Spark SQL-IEJoin, as shown in Table 5: (i) L1 data caches (dcache), (ii) last level cache (LLC), and (iii) data translation lookaside buffer (dTLB). The optimized algorithm with union arrays is 2.6 times faster than the original one. The performance gain in the optimized version is due to the lower number of cache loads and stores (L1-dcache-loads, L1-dcache-stores, dTLB-loads and TLB-stores), which is 2.7 to 3 times smaller than the original algorithm. This behavior is expected since the optimized IEJoin has fewer arrays *w.r.t.* the original version.

Global sorting on distributed IEJoin**.** As presented in Algorithm 5, the distributed version of our algorithm applies global sorting in the pre-processing phase (lines 6-7). We report in Table 6 a detailed performance comparison for Q_1 and Q_2 with and without global sorting on 100M tuples from Employees and Events datasets. The pre-processing time includes data loading from HDFS, global sorting, partitioning, and block-pairs materialization. One may think that the global sorting impairs the performance of distributed IEJoin as it shuffles data through the network. However, global sorting improves the performance of the distributed algorithm by 2.4 to 2.9 times. More specifically, the runtime for the pre-processing phase with global sorting is at least 30% faster compared to the case without global sorting. Moreover, we also note that the time required by IEJoin is one order of magnitude faster when using global sorting. This is because global sorting enables to filter out block-pair combinations that do not generate results. This greatly reduces the network overhead and increases the memory locality in the block combinations that are passed to our algorithm.

Parameter (M/sec)	**IEJoin** (union)	**IEJoin**
cache-references	6.5	8.4
cache-references-misses	3.9	4.8
L1-dcache-loads	**459.9**	**1,240.6**
L1-dcache-load-misses	8.7	10.9
L1-dcache-stores	**186.8**	**567.5**
L1-dcache-store-misses	1.9	1.9
L1-dcache-prefetches	4.9	7.0
L1-dcache-prefetches-misses	2.2	2.7
LLC-loads	5.1	6.6
LLC-load-misses	2.9	3.7
LLC-stores	3.8	3.7
LLC-store-misses	1.1	1.2
LLC-prefetches	3.1	4.1
LLC-prefetch-misses	2.2	2.9
dTLB-loads	**544.4**	**1,527.2**
dTLB-load-misses	0.9	1.6
dTLB-stores	**212.7**	**592.6**
dTLB-store-misses	0.1	0.1
Total time (sec)	**125**	**325**

Table 5 Cache statistics on 10M tuples (Events data)

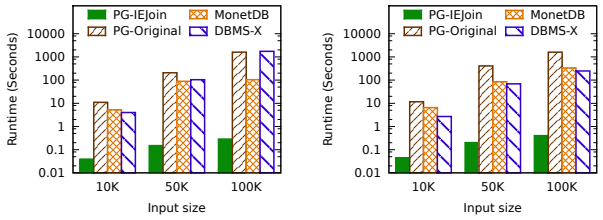
Query	Pre-process	IEJoin	Post-process	Total
With global sorting (Seconds)				
Q_1	632	**162**	519	1,313
Q_2	901	**84**	391	1,376
Without global sorting (Seconds)				
Q_1	1,025	1,714	426	3,165
Q_2	1,182	1,864	349	3,395

Table 6 IEJoin time analysis on 100M tuples and 6 workers

Based on the above experiments, in the following tests we used 1,024 bits as the default chunk size, the max scan index optimization, union arrays, and global sorting for distributed IEJoin.

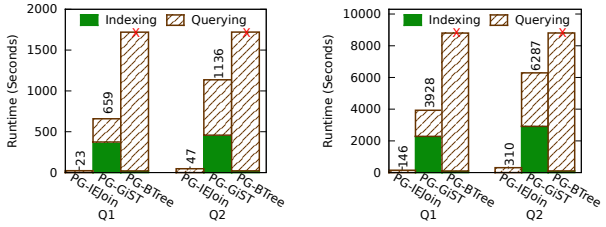
8.6 Single-node Experiments

In this set of experiments, we study the efficiency of IEJoin on datasets that fit the main memory of a single compute node and compare its performance with alternative centralized systems.



(a) Q_1 (b) Q_2

Fig. 10 IEJoin vs baseline systems (centralized)



(a) 10M tuples (b) 50M tuples

Fig. 11 IEJoin vs. BTree and GiST (centralized)

IEJoin **vs. baseline systems.** Figure 10 shows the results for queries Q_1 on Employees dataset and Q_2 on Events dataset in a centralized environment using 10K, 50K and 100K tuples. The x-axis represents the input size in terms of the number of tuples and the y-axis represents the corresponding running time in seconds. The figure reports that PG-IEJoin outperforms all baseline systems by more than one order of magnitude for both queries and for every reported dataset input size. In particular, PG-IEJoin is up to more than three (resp., two) orders of magnitude faster than PG-Original and MonetDB (resp., DBMS-X). Clearly, the baseline systems cannot compete with PG-IEJoin since they all use the classic Cartesian product followed by a selection predicate. In fact, this is the main reason why they cannot run for bigger datasets.

IEJoin **vs. indexing.** We now consider the two variants of PostgreSQL, PG-BTree & PG-GiST, to evaluate the efficiency of our algorithm on bigger datasets. We run again Q_1 and Q_2 on 10M and 50M records using both Employees and Events datasets. Figure 11 shows the results. In both experiments, IEJoin is more than one order of magnitude faster than PG-GiST. In fact, IEJoin is more than three times faster than the GiST indexing time alone. We stopped PG-BTree after 24 hours of runtime. Our algorithm performs better than these two baseline indices because it better utilizes the memory locality.

Memory consumption. The memory consumption for MonetDB increases exponentially with the input size. For example, MonetDB uses 419GB for an input dataset with only 200K records. In contrast to MonetDB, IEJoin makes better use of the memory. Table 7 shows that IEJoin uses around 200MB for Q_1 and Q_2 for an input dataset of 200K records, while MonetDB requires two orders of magnitude more memory. In Table 7, we also report the overall memory used by sorted attribute arrays, permutation arrays, and the bit-array. Moreover, although IEJoin requires 8.2GB of memory for an input dataset of 10M records, it runs to completion in about 8 hours (28,928 seconds) for a dataset producing more than 50 billion output records.

Query	Input	Output	Time(secs)	Mem(GB)
Q_1	100K	9K	0.30	0.1
Q_1	**200K**	1.1K	0.5	**0.2**
Q_1	1M	29K	2.79	0.9
Q_1	10M	3M	27.64	8.8
Q_2	100K	0.2K	0.34	0.1
Q_2	**200K**	0.8K	0.65	**0.2**
Q_2	1M	20K	3.38	0.9
Q_2	10M	2M	59.6	9.7
Q_4	100K	6M	2.8	0.1
Q_4	200K	25M	10.6	0.2
Q_4	1M	0.4B	186	0.9
Q_4	**10M**	**50.5B**	**28,928**	**8.2**

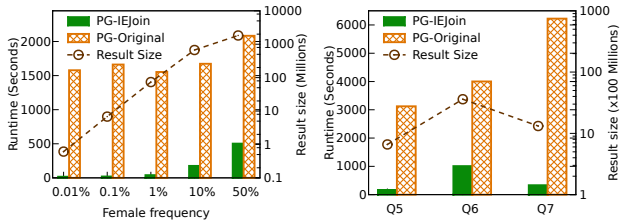
Table 7 Runtime and memory usage (PG-IEJoin)

Query	Data reading	Data sorting	Bit-array scanning	Total time
C++ IEJoin.				
Q_1	82	31	4	117
Q_2	85	31	3	119
PG-IEJoin.				
Q_1	46	94	4	146
Q_2	48	238	24	310
Spark SQL-IEJoin (Single Node).				
Q_1	158	240	165	563
Q_2	319	332	215	866

Table 8 Time breakdown on 50M tuples, all times in seconds

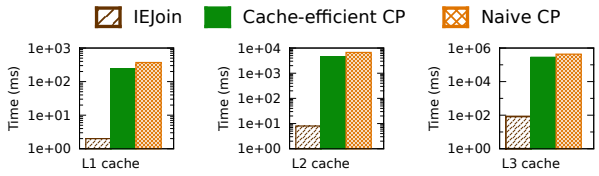
Time breakdown. We further analyze the breakdown time of IEJoin on Employees and Events datasets with 50M tuples. Table 8 shows that, by excluding the time required to load the dataset into memory, scanning the bit-array takes only 10% of the overall execution time in C++ IEJoin, 5% in PG-IEJoin, and 40% in Spark SQL-IEJoin, while the rest is mainly for sorting. This shows the high efficiency of our algorithm.

IEJoin **with the not equal ($\neq$) join predicate.** We tested the performance of IEJoin with the not equal join predicate with Q_5 on the Grades dataset. PG-Original runs the query by using nested loops, while PG-IEJoin uses two instances of IEJoin to process Q_5'. As shown in Figure 12(a), PG-IEJoin is from four times to two orders of magnitude faster than PG-Original. Note that the query output becomes larger while increasing the Female frequency in the dataset.



(a) Q_5 (b) Q_5, Q_6 and Q_7

Fig. 12 Q_5 runtime with different Female distributions, and Q_5, Q_6 and Q_7 runtimes with 10% Female distribution



(a) L1 cache (b) L2 cache (c) L3 cache

Fig. 13 Q_1 runtime for data that fits caches

We also tested the effect of a non-binary attribute by using queries Q_5, Q_6, and Q_7. For PG-IEJoin, the queries were transformed into a union of inequality joins as discussed before. Figure 12(b) shows that running PG-IEJoin on Q_5 and Q_7 is an order of magnitude faster than PG-Original, while on Q_6 it is four times faster. This is because the join attribute in Q_5 and one of the join attributes in Q_7 generate far less results than Q_6. In fact, the output of Q_6 is five times higher than the one of Q_5 and three times higher than the one of Q_7. This difference on Q_6 is expected since both attributes of the not equal join predicates are not binary.

IEJoin **vs. cache-efficient Cartesian product.** We further push our evaluation to better highlight the memory locality efficiency on Q_1 using Employees dataset. We compare the performance of C++ IEJoin with both naïve and cache-efficient Cartesian product joins for Q_1 on datasets that fit the L1 cache (256 KB), L2 cache (1 MB), and L3 cache (8 MB) of the Intel Xeon processor. We used 10K tuples for L1 cache, 40K tuples for L2 cache, and 350K tuples for L3 cache. We do not report results for query Q_2 because they are similar to Q_1. In Figure 13, we see that when the dataset fits in the L1 cache, IEJoin is two orders of magnitude faster than both the cache-efficient Cartesian product and the naïve Cartesian product. Furthermore, as we increase the dataset size for Q_1 to be stored at the L2 and L3 caches, we see that IEJoin becomes almost three and four orders of magnitude faster than the Cartesian product. This is because of the delays of L2 and L3 caches and the complexity of the Cartesian product.

Single-node summary. IEJoin outperforms existing baselines by at least an order of magnitude for two main reasons: it avoids the use of Cartesian product and it exploits memory locality by using memory-contiguous data structures with small footprint. In other words, our algorithm avoids as much as possible going to memory to fully exploit the CPU speed.

Join Predicates	**Estimation on a % sample** (number of overlapping blocks)			
	1%	**5%**	**10%**	**20%**
salary & tax	**501**	**2535**	**5130**	**10K**
salary & age	125250	3.1M	12M	50M
tax & age	125250	3.1M	12M	50M

Table 9 Q_1''s selectivity estimation on 50M tuples Employees dataset with **low selectivity** age attribute

Join Predicates	**Estimation on a % sample** (number of overlapping blocks)			
	1%	**5%**	**10%**	**20%**
salary & tax	**500**	**2545**	**5144**	**10K**
salary & age	503	2559	5237	11K
tax & age	124750	3M	12M	50M

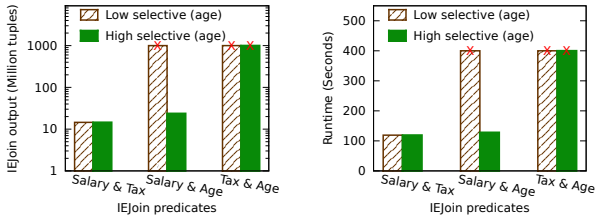
Table 10 Q_1''s selectivity estimation on 50M tuples Employees dataset with **high selectivity** age attribute

8.7 IEJoin Optimizer Experiments

In this section, we evaluate the accuracy of the IEJoin optimizer when dealing with multi-predicate and multi-way queries. We use Q_1' for multi-predicate IEJoin and Q_{mw} for multi-way IEJoin. For the multi-predicate experiment, we evaluate the performance of IEJoin by using the highest selective predicates, found by Algorithm 3, compared to using other predicates. For the multi-way IEJoin, we compare the runtime of the plan generated by Algorithm 4 against other plans.

Multi-predicate IEJoin. We evaluate the accuracy of the selectivity estimation algorithm on query Q_1' usinmg C++ IEJoin. We first use Algorithm 3 to calculate the selectivity estimation for all three predicate pairs using the join attributes salary, tax, and age.

To evaluate the estimation algorithm, we generated two different distributions of attribute age for query Q_1': a low selectivity distribution that generates a large output with salary and tax attributes, and a high selectivity distribution that generates a small output with attribute salary. The low selectivity distribution injects random noise on age attribute in each tuple, generating a large output for the IEJoin on (r.salary $<$ s.salary AND r.age $>$ s.age) or on (r.tax $>$ s.tax AND r.age $>$ s.age). For the high selectivity distribution, however, we carefully assign a value proportional to the salary attribute to generate a small output for the IEJoin on (r.salary $<$ s.salary AND r.age $>$ s.age). We show the selectivity estimation



(a) Result size (b) Runtime

Fig. 14 Result size and runtime for Q'_1

for both distributions of the age attribute in Tables 9 and 10. The tables report selectivity estimations when using input sample of size 1%, 5%, 10%, and 20%. We determine the best join predicate pair for Q'_1 by selecting the predicate pair with the minimum overlapping blocks. According to Tables 9 and 10, the (salary, tax) join predicate pair has the highest selectivity in all input samples. Although the difference between predicates pairs (salary, tax) and (salary, age) in Table 10 is relatively small, we notice that it gets larger as we increase the size of the sample input. To validate this observation, we execute IEJoin on each join predicate pair and compare the output size and runtime values.

We show in Figure 14(a) that choosing (salary, tax) is the right decision. In fact, the output size of (salary, tax) with the low selectivity age attribute is at least two orders of magnitude lower than the results with other predicates, and it is 50% lower than (salary, age) with the high selectivity age attribute. In Figure 14(b), we show the runtime difference among predicate pairs with the high selectivity age attribute. Join predicate pair (salary, tax) is orders of magnitude faster than predicate pair (tax, age), but only a couple of seconds faster than predicate pair (salary, age). Although the performance difference between (salary, tax) and (salary, age) pairs are small in the dataset with high selectivity age, the IEJoin query optimizer was able to detect that using (salary, tax) is faster than using (salary, age). We also notice that the runtime for the low selectivity age dataset is orders of magnitude faster with predicate pair (salary, tax) compared with other combinations.

Multi-way IEJoin. We test the multi-way IEJoin optimization in Algorithm 4 with the five relations in Q_{mw}. Each relation in Q_{mw} contains a random number of tuples from both Employees and Events datasets. We summarize in Table 11 the size of each relation, the selectivity estimation (computed with Algorithm 3 on 1% sample size), and the join result size based on the inequality conditions in Q_{mw}. Table 11 shows that the selectivity estimation is consistent with the actual join result size; the lower the estimation the smaller the output size and the higher the estimation the larger the

	Number of tuples	**Join Selectivity**		
		Join	Estimation	Result size
R	2M	R⋈S	388	576961
S	38M	S⋈T	6.7M	31.5M
T	28M	T⋈V	1999	876513
V	10M	V⋈W	0.5M	12.5M
W	22M	–	–	–

Table 11 Relation sizes, selectivity estimations and actual output for individual joins in Q_{mw}

1^{st} **Join**		2^{nd} **Join**		3^{rd} **Join**		4^{th} **Join**		**Total**
R⋈S	**106s**	⋈T	**77s**	⋈V	**21s**	⋈W	**53s**	**257s**
S⋈T	390s	⋈R	41s	⋈V	21s	⋈W	51s	503s
		⋈V	67s	⋈R	4s	⋈W	51s	512s
				⋈W	55s	⋈R	4s	516s
T⋈V	**103s**	⋈S	108s	⋈R	4s	⋈W	53s	268s
				⋈W	58s	⋈R	4s	273s
		⋈W	57s	⋈S	106s	⋈R	3s	269s
V⋈W	319s	⋈T	91s	⋈S	108s	⋈R	4s	522s

Table 12 Runtime of different multi-way join plans for Q_{mw}

output size. Based on Algorithm 4, the optimal plan for Q_{mw} is $((((R \bowtie S) \bowtie T) \bowtie V) \bowtie W)$ according to the selectivity estimations in Table 11. To evaluate the quality of the optimal plan, we evaluate the performance of all multi-way IEJoin plans for Q_{mw} with PG-IEJoin, and report the results in Table 12. Plan $(((R \bowtie S) \bowtie T) \bowtie V) \bowtie W$ that was generated by Algorithm 4 is indeed the fastest one. Although $(R \bowtie S)$ has higher selectivity than $(T \bowtie V)$, evaluating $(T \bowtie V)$ is slightly faster than $(R \bowtie S)$. Indeed $(R \bowtie S)$ produces less results compared to $(T \bowtie V)$ (Table 11), but since sorting dominates the performance for IEJoin, as shown in Table 8, $(R \bowtie S)$ becomes slightly slower with a larger number of tuples to sort (40M tuples) compared to $(T \bowtie V)$ (38M tuples). Nevertheless, $((((R \bowtie S) \bowtie T) \bowtie V) \bowtie W)$ remains the fastest plan because it eliminates more tuples earlier-compared to the rest.

Optimizer experiments summary. With only 1% of the data, selectivity estimation in Algorithm 3 is able to accurately distinguish between high and low selective IEJoins. Note that the 1% sample size may not be applicable for other datasets since the minimum sample size depends on the distribution of the join attributes. The performance degradation of not selecting the optimal query execution plan for multi-predicate and multi-way IEJoins varies between 5% and orders of magnitude performance drop. With negligible overhead, our selectivity estimation allows a DBMS to tune its query optimizer to select the optimal plan for multi-predicate and multi-way IEJoins.

8.8 Incremental IEJoin Experiments

We developed the incremental algorithm in Section 5 using C++ IEJoin and tested it on Q_1. Again, we do not report results for Q_2 since they were similar to those of Q_1. We used five different implementations for the experiments in this section: (1) Non-incremental (our original implementation), (2) incremental non-batched sort-merge (ΔInc), (3) incremental batched sort-merge (B-ΔInc), (4) incremental with the packed-memory array (PMA) data structure to dynamically maintain the cached input (P-ΔInc), and (5) σInc which is similar to B-ΔInc but returns the full output. The three variations of ΔInc return results that correspond only to the updates while σInc generates results from both the updates and the cached input.

Incremental IEJoin with small updates. We first study the advantage of our P-ΔInc algorithm on small updates. In Figure 15, we compare the runtime of ΔInc, B-ΔInc, non-incremental, and P-ΔInc by using 80M tuples as cached input. We consider ΔInc as the baseline for incremental IEJoin. In Figure 15, ΔInc is twice faster than the non-incremental IEJoin for update of size 1. As the size of the update increases, ΔInc slows down because it processes each update individually. To avoid this overhead, we use P-ΔInc to increase the efficiency of individual updates and B-ΔInc to process updates in a batched fashion. Since the update sizes are relatively small, the runtime of both B-ΔInc and the non-incremental algorithm across different update sizes remain constant. Figure 15 shows that B-ΔInc is always twice faster than the non-incremental algorithm, while P-ΔInc is 30% better than B-ΔInc and three times faster than the non-incremental with update of size 1. The processing overhead of P-ΔInc is directly proportional to the update size, and its performance declines as the update size increases. Although P-ΔInc is up to two orders of magnitude faster than ΔInc on higher update sizes, B-ΔInc is better than P-ΔInc on updates larger than 50. The limitation of P-ΔInc on updates larger than 50 is inherited from the design of PMA which works better with individual updates. Note that P-ΔInc has an extra 60% memory overhead, compared to the non-incremental IEJoin, due to the extra empty spaces maintained by the PMA.

Incremental IEJoin with large updates. In these experiments, we focus on B-ΔInc and σInc since both ΔInc and P-ΔInc do not work well with large updates. We show in Figures 16(a), 16(b), and 16(c) a comparison between the runtime (without data loading) of B-ΔInc and non-incremental IEJoin with different update and cached input sizes. In Figure 16(a), we start with 10M tuples as cached input. B-ΔInc is 40% faster on the update of size 1M tuples and has 5 times less output compared to the non-incremental algorithm. For updates larger than 1M, however, we notice that the performance of B-ΔInc significantly drops as the difference in the output sizes between the B-ΔInc and the non-incremental becomes insignificant. Similar behavior can be observed in Figures 16(b) and 16(c) with cached input of 20M and 30M tuples, respectively. ΔInc is 50% and 30% faster than the non-incremental algorithm on updates of sizes 1M and 5M tuples, respectively, while its performance drops on updates larger than 5M tuples in both cases. We also notice that B-ΔInc on the 1M tuples update has 10 times smaller output in Figure 16(b) and 15 times smaller output in Figure 16(c) compared to the non-incremental algorithm. From the above three figures, B-ΔInc clearly shows significant improvement over the non incremental algorithm (up to 50%) with update sizes that generate significantly smaller output. In this experimental setup, an efficient update size for B-ΔInc does not exceed 25% of the cached input size.

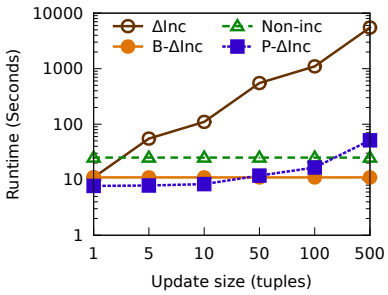
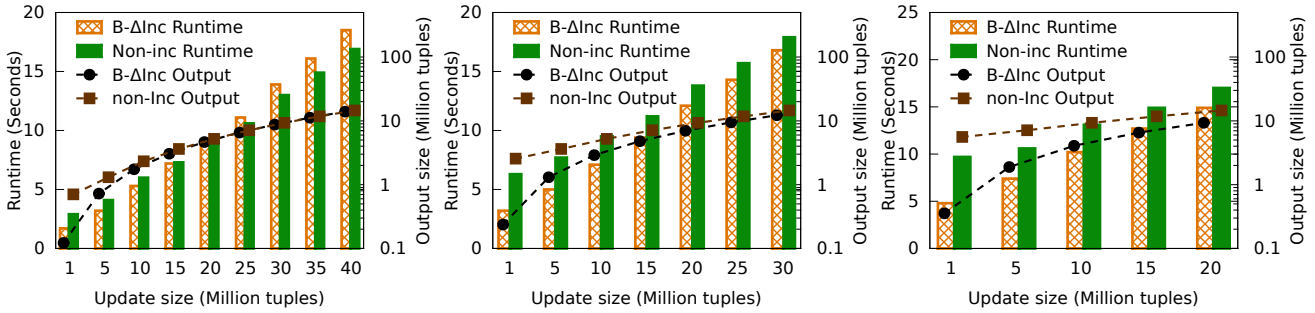


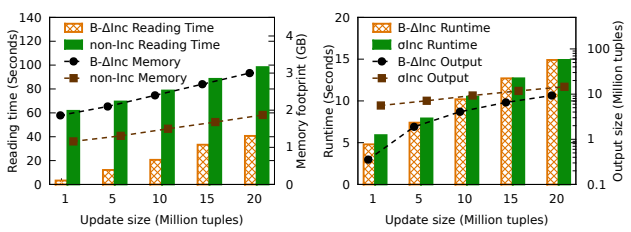
Fig. 15 Runtime of ΔInc, B-ΔInc, non-incremental, and P-ΔInc on small update sizes

We further tested the effect of update sizes on disk reading and memory consumption on B-ΔInc when using 30M tuples as cached input. Based on Figure 17(a), B-ΔInc gains up to 90% performance increase in disk reading time compared to the non-incremental algorithm with small updates sizes. The downside of the incremental algorithm is that it has 60% higher memory overhead compared to the non-incremental one, caused by data structures required to enable fast IEJoin updates. We also compare the performance difference between B-ΔInc and σInc in Figure 17(b) using 30M tuples as cached input, where the only difference between them is the size of the output. σInc is at most 20% slower than B-ΔInc when the output difference between them is large on the 1M tuples update. However, the performance gap between B-ΔInc and σInc becomes negligible as the output difference gets smaller on larger update sizes.



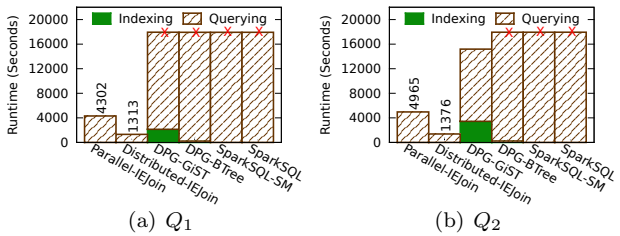
(a) 10M cached input (b) 20M cached input (c) 30M cached input

Fig. 16 Runtime and output size of non-incremental and *B-ΔInc* IEJoin on big update sizes



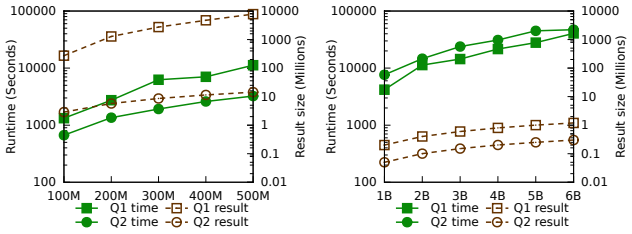
(a) Memory & I/O overhead (b) *B-ΔInc* vs. *σInc*

Fig. 17 Memory and I/O overhead of *B-ΔInc*, and runtime and output size of *B-ΔInc* and *σInc* (30M cached input)



(a) Q_1 (b) Q_2

Fig. 18 Distributed IEJoin (100M tuples, 6 nodes/threads)



(a) Employees & Events (b) Employees2 & Events2

Fig. 19 Distributed IEJoin, 6B tuples, 16 nodes

Incremental experiments summary. When the update size is smaller than 50, the PMA-based incremental algorithm performs 30% better than the sort-merge incremental algorithm and three times better than the original IEJoin. As we increase the size of the updates, the batched sort-merge-based incremental algorithm becomes more efficient than the PMA-based one. For update sizes that do not exceed 25% of the cached input size, the batched sort-merge-based incremental algorithm is twice faster than the original algorithm.

8.9 Multi-node Experiments

We now evaluate our proposal in a distributed environment and by using larger datasets.

Scalable IEJoin vs. baseline systems. We should note that we had to run these experiments on a cluster of 6 compute nodes only due to the limit imposed by the free version of the distributed PostgreSQL system. Additionally, in these experiments, we stopped the execution of any system that exceeded 24 hours. We test the scalable IEJoin using the parallel IEJoin (with 6 threads in a single machine while enabling disk caching) and the distributed IEJoin (on 6 compute nodes). Figure 18 shows the results of all distributed systems we consider for queries Q_1 and Q_2. This figure shows again that both versions of our algorithm significantly outperform all baselines. It is on average more than one order of magnitude faster. In particular, we observe that only DPG-GiST could terminate before 24 hours for Q_2. The distributed IEJoin is twice faster than the time required to run GiST indexing alone. Moreover, distributed IEJoin is, as expected, faster than the parallel multi-threaded version. This is because the multi-threaded version has a higher processing overhead due to resource contention. These results show the high superiority of our algorithm over all baseline systems.

Scaling input size. We further push the evaluation of the efficiency in a distributed environment with bigger input datasets: from 100M to 500M records with large results size (Employees and Events), and from 1B to 6B records with smaller results size (Employees2 and Events2). As we now consider IEJoin only, we run this experiment on our entire 16 compute nodes cluster. Figure 19 shows the runtime results as well as the output sizes. We observe that IEJoin gracefully scales along with input dataset size in both scenarios. We also observe in Figure 19(a) that, when the output size is large,

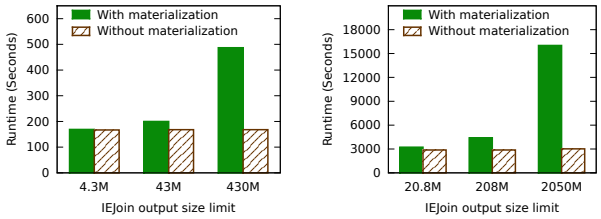
the runtime increases accordingly as it is dominated by the materialization of the results. In Figure 19(a), Q_1 is slower than Q_2 as its output is three orders of magnitude larger. When the output size is relatively small, both Q_1 and Q_2 scale well with increasing input size (see Figure 19(b)). Below, we study in more details the impact of the output size on performance.

Scaling dataset output size. We test our system's scalability in terms of the output size using two real datasets (MDC and Cloud) as shown in Figure 20. To have full control on this experiment, we explicitly limit the output size from 4.3M to 430M for MDC, and 20.8M to 2050M for Cloud. The figures clearly show that the output size affects runtime; the larger the output size, the longer it will take to produce them. They also show that materializing a large number of results is costly. Take Figure 20(a) for example, when the output size is small (*i.e.,* 4.3M), materializing them or not will have similar performance. However, when the output size is big (*i.e.,* 430M), materializing the results takes almost 2/3 of the entire running time.

In order to run another set of experiments with a much bigger output size, we created two variants of Q_3 for MDC dataset by keeping only two predicates over four (less selectivity). Figure 21 shows the scalability results of these experiments with no materialization of results. For Q_{3a}, IEJoin produced more than 1,000B records in less than 3,000 seconds. For Q_{3b}, we stopped the execution after 2 hours with more than 5,000B tuples in the temporary result. This demonstrates the good scalability of our solution.

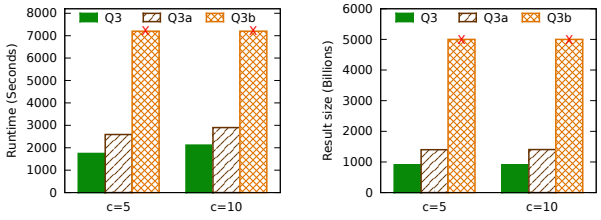
Speedup and scaleup. We also test speedup and scaleup efficiency of the distributed IEJoin by using Employees2 dataset and query Q_1. Figure 22(a) shows that our algorithm has outstanding speedup thanks to the scalability optimizations. IEJoin was only 4%, 3% and 16% slower than the ideal speedup when processing 8B rows on 4, 8 and 16 workers respectively. Figure 22(b) shows the scaleup efficiency of IEJoin as we proportionally increase the cluster size and input size. We observe that distributed IEJoin also has good scaleup: on 4 workers (2B rows) and 8 workers (4B rows) it was only 5% and 20% slower than the ideal scaleup. However, due to the increase in dataset size, the sorting overhead in IEJoin becomes larger. This explains why scalable IEJoin, on 16 workers with 8B rows input, is 46% slower than the ideal scaleup.

Multi-node summary. Similarly to the centralized case, IEJoin outperforms existing baselines by at least one order of magnitude. In particular, we observe that it gracefully scales in terms of input (up to 6B tuples). This is because our algorithm first join the metadata,



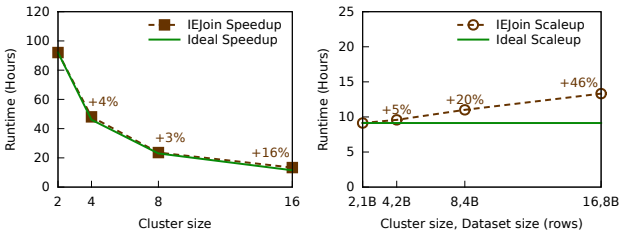
(a) MDC - Q_3 (b) Cloud - Q_4

Fig. 20 Runtime of IEJoin ($c = 10$)



(a) Runtime (b) Output size

Fig. 21 Without result materialization ($c = 5, 10$)



(a) Speedup (b) Scaleup

Fig. 22 Speedup (8B rows) & Scaleup on Q_1

which are orders of magnitude smaller than the actual data. As a result, it shuffles only those data partitions that can potentially produce join results. Typically, IEJoin processes a small number of data partitions.

9 Related work

Several cases of inequality joins have been studied in the literature; these include band joins, interval joins and, more generally, spatial joins. IEJoin is specially optimized for joins with at least two predicates in {"<", ">", "≤", "≥"}.

A band join [13] of two relations R and S has a join predicate that requires the join attribute of S to be within some range of the join attribute of R. The join condition is expressed as $R.A - c_1 \leq S.B$ & $S.B \leq R.A + c_2$, where c_1 and c_2 are constants. The band-join algorithm [13] partitions the data from relations R and S into partitions R_i and S_i respectively, such that for every tuple $r \in R$, all tuples of S that join with r appear in S_i. It assumes that R_i fits into memory. Contrary to IEJoin, band join is limited to a single inequality condition type, involving one single attribute from

each column. IEJoin works for any inequality conditions and attributes from the two relations. While band join queries can be processed using our algorithm, not all IEJoin queries can be reduced to band join.

Interval joins are frequently used in temporal and spatial data. The work in [17] proposes the use of the relational Interval Tree to optimize joining interval data. Each interval intersection is represented by two inequality conditions, where the lower and upper times of any two tuples are compared to check for overlaps. This work optimizes non-equijoins on interval intersections, where they represent each interval as a multi-value attribute. Compared to our work, they only focus on improving interval intersection queries and cannot process general purpose inequality joins.

Spatial indexing is widely used in several applications with multidimensional datasets, such as bitmap indices [8, 30], R-trees [21] and space filling curves [7]. In PostgreSQL, support for spatial indexing algorithms is provided through a single interface known as Generalized index Search Tree [22] (GiST). From this collection of indices, bitmap index is the most suitable technique to optimize multiple attribute queries that can be represented as 2-dimensional data. Examples of 2-dimensional datasets are intervals (*e.g.,* start and end time in Q_2), GPS coordinates (*e.g.,* Q_3), and any two numerical attributes that represent a point in an XY plot (*e.g.,* salary and tax in Q_1). The main disadvantage of the bitmap index is that it requires large memory footprint to store all unique values of the composite attributes [9,36]. Bitmap index is a natural baseline for our algorithm, but, unlike IEJoin, it does not perform well with high cardinality attributes, as demonstrated in Figure 8. R-trees, on the other hand, are not suitable because an inequality join corresponds to window queries that are unbounded from two sides, and consequently intersect with a large number of internal nodes of the R-tree, generating unnecessary disk accesses.

The patent in [32] also presents an algorithm to optimize the Cartesian product when joining two tables based on a single inequality condition. The algorithm partitions the input relations into smaller blocks based on the value distribution and min/max values of the join predicate. It then applies Cartesian product on a subset of the input partitions, where it eliminates unnecessary partitions depending on the join condition. Compared with our approach, this algorithm optimizes the Cartesian product through partitioning based on only one single inequality join predicate.

Several other proposals have been made to speed-up join executions in MapReduce (*e.g.,* [14]). However, they focus on joins with equalities thus requiring massive data shuffling to be able to compare each tuple with each other. There have been few attempts to devise efficient implementation of theta-join in MapReduce [33,38]. [33] focuses on pair-wise theta-join queries. It partitions the Cartesian product output space with rectangular regions of bounded sizes. Each partition is mapped to one reducer. The proposed partitioning guarantees correctness and workload balance among the reducers while minimizing the overall response time. [38] further extends [33] to solve multi-way theta-joins. It proposes an I/O and network cost-aware model for MapReduce jobs to estimate the minimum time execution costs for all possible decomposition plans for a given query, and selects the best plan given a limited number of computing units and a pool of possible jobs. We propose a new algorithm to do the actual inequality join based on sorting, permutation arrays, and bit arrays. The focus in these previous proposals is on efficiently partitioning the output space and on providing a cost model for selecting the best combination of MapReduce jobs to minimize response time. In both proposals, join is performed with existing algorithms, where inequality conditions correspond to Cartesian product followed by selection.

A large number of approaches focused on selectivity estimation. However, most existing work on selectivity estimation has focused on equijoins [2, 23, 29, 34, 35]. There are few proposals for the general case of theta-join [38] and spatial join [31]. Nevertheless, most of these proposals estimate the selectivity of the inequality join to be $O(n^2)$ because it is evaluated as Cartesian product. IEJoin significantly differs from these works as it does not consider Cartesian product. It uses an efficient selectivity estimation technique that computes the number of overlapping sorted blocks obtained from a sample of the input relations.

10 . . . The End

To help Bob with his inequality join queries, we proposed novel algorithms for efficiently evaluating these queries. We rely on auxiliary data structures that enable efficient computations and require a small memory footprint. Our algorithms exploit data locality to achieve orders of magnitude computation speedup. We introduced selectivity estimation to support multi-predicate and multi-way join queries. We devised incremental versions to deal with continuous queries on changing data. We implemented these algorithms on both a centralized and a distributed system, namely PostgreSQL and Spark SQL, respectively. We additionally implemented IEJoin in Nadeef, an open source data cleaning system. Our experiments demonstrate that IEJoin is superior to baseline systems: it is 1.5

to 3 orders of magnitude faster than commercial and open-source centralized databases; and at least 2 orders of magnitude faster than the original Spark SQL. While the algorithm does not break the theoretical quadratic time bound, our experiments show performance results that are proportional to the size of the output.

11 Acknowledgments

Portions of the research in this paper used the MDC Database made available by Idiap Research Institute, Switzerland and owned by Nokia.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. F. N. Afrati and J. D. Ullman. Optimizing Joins in a Map-reduce Environment. In *EDBT*, pages 99–110, 2010.
3. D. Agrawal, S. Chawla, A. K. Elmagarmid, Z. K. M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, and M. J. Zaki. Road to Freedom in Big Data Analytics. In *EDBT*, pages 479–484, 2016.
4. M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.
5. M. A. Bender and H. Hu. An Adaptive Packed-memory Array. *TODS*, 32(4), 2007.
6. P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional Functional Dependencies for Data Cleaning. In *ICDE*, pages 746–755, 2007.
7. C. Böhm, G. Klump, and H.-P. Kriegel. XZ-Ordering: A Space-Filling Curve for Objects with Spatial Extension. In *SSD*, pages 75–90, 1999.
8. C.-Y. Chan and Y. E. Ioannidis. Bitmap Index Design and Evaluation. In *SIGMOD*, pages 355–366, 1998.
9. C.-Y. Chan and Y. E. Ioannidis. An Efficient Bitmap Encoding Scheme for Selection Queries. In *SIGMOD*, pages 215–226, 1999.
10. X. Chu, I. F. Ilyas, and P. Papotti. Holistic Data Cleaning: Putting Violations into Context. In *ICDE*, pages 458–469, 2013.
11. M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: A Commodity Data Cleaning System. In *SIGMOD*, 2013.
12. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.
13. D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An Evaluation of Non-Equijoin Algorithms. In *VLDB*, pages 443–452, 1991.
14. J. Dittrich, J. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB*, 3(1):515–529, 2010.
15. A. Ebaid, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, J. Quiané-Ruiz, N. Tang, and S. Yin. NADEEF: A Generalized Data Cleaning System. *PVLDB*, 6(12):1218–1221, 2013.
16. A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, J. Quiané-Ruiz, N. Tang, and S. Yin. NADEEF/ER: Generic and Interactive Entity Resolution. In *SIGMOD*, pages 1071–1074, 2014.
17. J. Enderle, M. Hampel, and T. Seidl. Joining Interval Data in Relational Databases. In *SIGMOD*, pages 683–694, 2004.
18. D. Gao, C. S. Jensen, R. T. Snodgrass, and M. D. Soo. Join Operations in Temporal Databases. *VLDB J.*, 14(1):2–29, 2005.
19. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems*. Pearson Education, 2009.
20. N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *SIGMOD*, pages 325–336, 2006.
21. A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, pages 47–57, 1984.
22. J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *VLDB*, pages 562–573, 1995.
23. A. Kemper, D. Kossmann, and C. Wiesner. Generalised Hash Teams for Join and Group-by. In *VLDB*, pages 30–41, 1999.
24. Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and S. Yin. BigDansing: A System for Big Data Cleansing. In *SIGMOD*, pages 1215–1230, 2015.
25. Z. Khayyat, W. Lucia, M. Singh, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and P. Kalnis. Lightning Fast and Space Efficient Inequality Joins. *PVLDB*, 8(13):2074–2085, 2015.
26. N. Kiukkonen, B. J., O. Dousse, D. Gatica-Perez, and L. J. Towards Rich Mobile Phone Datasets: Lausanne Data Collection Campaign. In *ICPS*, 2010.
27. D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.
28. J. K. Laurila, D. Gatica-Perez, I. Aad, B. J., O. Bornet, T.-M.-T. Do, O. Dousse, J. Eberle, and M. Miettinen. The Mobile Data Challenge: Big Data for Mobile Computing Research. In *Pervasive Computing*, 2012.
29. G. Lohman, C. Mohan, L. Haas, D. Daniels, B. Lindsay, P. Selinger, and P. Wilms. Query Processing in R*. In *Query Processing in Database Systems*, pages 31–47. 1985.
30. T. L. Lopes Siqueira, R. R. Ciferri, V. C. Times, and C. D. de Aguiar Ciferri. A Spatial Bitmap-based Index for Geographical Data Warehouses. In *SAC*, pages 1336–1342, 2009.
31. N. Mamoulis and D. Papadias. Multiway Spatial Joins. *TODS*, 26(4):424–475, 2001.
32. J. Morris and B. Ramesh. Dynamic Partition Enhanced Inequality Joining Using a Value-count Index, 1 2011. US Patent 7,873,629 B1.
33. A. Okcan and M. Riedewald. Processing Theta-Joins using MapReduce. In *SIGMOD*, pages 949–960, 2011.
34. D. A. Schneider and D. J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-nothing Multiprocessor Environment. In *SIGMOD*, 1989.
35. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, pages 23–34, 1979.
36. K. Stockinger and K. Wu. Bitmap Indices for Data Warehouses. *Data Warehouses and OLAP: Concepts, Architectures and Solutions*, 5:157–178, 2007.
37. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, pages 10–10, 2010.
38. X. Zhang, L. Chen, and M. Wang. Efficient Multi-way Theta-Join Processing Using MapReduce. *PVLDB*, 5(11):1184–1195, 2012.