Ring-constrained Join: Deriving Fair Middleman Locations from Pointsets via a Geometric Constraint

Man Lung Yiu
Department of Computer Science
Aalborg University
DK-9220 Aalborg, Denmark
mly@cs.aau.dk

Panagiotis Karras
Department of Informatics
University of Zurich
CH-8050 Zurich, Switzerland
karras@ifi.uzh.ch

Nikos Mamoulis
Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong
nikos@cs.hku.hk

ABSTRACT

We introduce a novel spatial join operator, the ring-constrained join (RCJ). Given two sets P and Q of spatial points, the result of RCJ consists of pairs $\langle p, q\rangle$ (where $p \in P$, $q \in Q$) satisfying an intuitive geometric constraint: the smallest circle enclosing p and q contains no other points in P, Q. This new operation has important applications in decision support, e.g., placing recycling stations at fair locations between restaurants and residential complexes. Clearly, RCJ is defined based on a geometric constraint but not on distances between points. Thus, our operation is fundamentally different from the conventional distance joins and closest pairs problems. We are not aware of efficient processing algorithms for RCJ in the literature. A brute-force solution requires computational cost quadratic to input size and it does not scale well for large datasets. In view of this, we develop efficient R-tree based algorithms for computing RCJ, by exploiting the characteristics of the geometric constraint. We evaluate experimentally the efficiency of our methods on synthetic and real spatial datasets. The results show that our proposed algorithms scale well with the data size and have robust performance across different data distributions.

1. INTRODUCTION

In this paper, we identify a novel spatial join operator, called *ring-constrained join* (RCJ). Given two spatial pointsets P and Q, our goal is to find all pairs $\langle p, q\rangle$, such that $p \in P$, $q \in Q$, and the smallest circle enclosing p and q contains no other points in P, Q. This simple geometric constraint captures an interesting and intuitive relationship between two pointsets, as we will see later. Figure 1 depicts two datasets $P = \{p_1, p_2\}$ and $Q = \{q_1, q_2\}$ on a map. The RCJ result pairs are $\langle p_1, q_1\rangle$, $\langle p_2, q_1\rangle$, and $\langle p_2, q_2\rangle$ as their smallest enclosing circles do not contain other points. Observe that $\langle p_1, q_2\rangle$ is not a result pair as its circle contains some other point (e.g., p_2).

From a geometric point of view, each RCJ pair $\langle p_i, q_j\rangle$ corresponds to its smallest enclosing circle, with its center location and radius. Such inherent, derived information is important for decision support applications. In general, given two sets of *facilities*, the center locations of their RCJ pairs can be used for placing *third-party service stations* so that the services can be provided in a convenient manner. To illustrate this, suppose that P and Q contain the locations of cinemas and restaurants respectively. Each RCJ pair $\langle p_i, q_j\rangle$ satisfies the following beneficial properties:

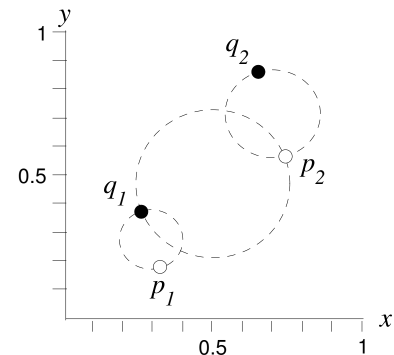


Figure 1: Example of Ring-constrained Join

- **Convenience.** Among all possible locations in the space, the circle center is known to minimize the maximum distances [10] from both p_i (a cinema) and q_j (a restaurant). Thus, the center can be regarded as the most convenient location to both facilities. For instance, a taxi stand or metro station may be built at that center location.
- **Fairness.** The center (e.g., metro station or taxi stand) is equidistant from a cinema and a restaurant. Since it does not have bias towards a particular facility, this is fair for passengers of both examined destinations (e.g., restaurant or cinema).
- **Commercial Advantage.** Metro passengers arriving at the center always consider p_i and q_j to be their nearest cinema and restaurant respectively. It is less likely for the passengers to travel to other cinema or restaurant, provided that they are of similar quality. Besides, people leaving from the cinema/restaurant will probably come back to the same central station.

Besides the above taxi-stand and metro scenarios, we illustrate other important applications of RCJ in the following

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT'08, March 25–30, 2008, Nantes, France.
Copyright 2008 ACM 978-1-59593-926-5/08/0003 ...$5.00.

examples.

- **Recycling Stations.** A recycling station has several designated bins for accepting recyclable materials (e.g., one for aluminum cans, one for glass containers, one for plastic bottles). The city council wants to allocate recycling stations for appropriate pairs between restaurants and residential complexes in the city. Large number of recycling stations are required due to the huge amount of recyclable material (e.g., bottles, biodegradable waste) produced by restaurants and residential complexes. The RCJ results are used for placing each recycling station at a fair distance from the corresponding restaurant and residential complex.

- **Tourist Recommendation.** A tourist asks for RCJ pair(s) such that he/she will be able to visit both the cinema and restaurant conveniently. In particular, the RCJ result set can be sorted in ascending order of the ring diameter so as to facilitate the tourist for making his/her choice with ease. While browsing the sorted list of RCJ results, the tourist has the flexibility to determine whether an RCJ pair satisfies his/her preferences, e.g., (i) he/she is not too far from the center of the pair, or (ii) the qualities of the corresponding cinema and restaurant are not too low.

- **Postboxes.** The post corporation intends to place its post boxes at locations that are convenient to public access. A nice distribution would be to have post boxes located at centers of RCJ pairs between buildings. This is viewed as the self-RCJ problem, where both sets P and Q contain locations of all buildings.

- **School Bus Stops.** A school bus company wishes to allocate its bus stops such that they are close to residential estates of children. Centers of RCJ pairs between estates provide handy locations for placing school bus stops. The RCJ result set can be sorted in descending order of the number of children in the residential estates associated with the RCJ pair. This way, the management of the bus company will be able to discover appropriate locations for its bus stops.

A (binary) spatial join operator takes two spatial datasets P, Q as input and outputs a subset of the Cartesian product $P \times Q$ based on a join predicate. In this paper, we consider the general scenario that P and Q are different datasets (e.g., in the example of recycling stations). In the special case where P and Q are identical (e.g., in the example of postboxes), the operation is called the self-join. Examples of traditional join operators on spatial pointsets include the ϵ-distance join [2, 8] and the k-closest pairs join [6, 3, 13]. However, RCJ has fundamental differences from them.

First, each RCJ pair is semantically equivalent to a geometric object (i.e., enclosing circle), and its inherent information (e.g., circle center, radius) cannot be derived from existing spatial join operators. Second, the join pairs of RCJ adapt to the local data density; nearby pairs in dense regions lead to small circles whereas pairs in sparse regions correspond to large circles. In contrary to conventional spatial join operators, RCJ results do not necessarily obey global distance constraints. For example, in Figure 1, p_2 is not close to q_1, however, $\langle p_2, q_1 \rangle$ belongs to the RCJ result. Third, RCJ does not rely on input parameters, other than the data alone. On the other hand, the ϵ-distance join and the k-closest pairs join rely on the distance parameter ϵ and the cardinality parameter k respectively. Both ϵ and k require domain-expert knowledge to specify at query time, and the parameter-based behavior may vary significantly from region to region, depending on the local density and distribution of data.

Given the importance of RCJ, it is essential to develop effective and efficient techniques for processing RCJ. The computation of RCJ is challenging; a simple *brute-force* solution is to perform a nested loop join between P and Q, examine each possible join pair $\langle p, q \rangle$ and verify whether the pair is a result by issuing range search for its enclosing circle. The brute-force approach requires $O(|P| \cdot |Q|)$ range searches; it is prohibitively expensive and does not scale well for large datasets. To our best knowledge, there is no efficient algorithm for RCJ computation in the literature. Motivated by this, we first study how to reduce the search space, and then present efficient algorithms for processing RCJ.

In summary, our contributions include:

- Introduction of the ring-constrained join (RCJ) operator, with applications to spatial decision support systems

- Development of crucial lemmas for pruning the search space in RCJ computation

- Design of I/O-efficient algorithms for RCJ computation on spatial datasets indexed by R-trees

The rest of the paper is organized as follows. Section 2 discusses the related work on basic spatial queries and spatial joins. Section 3 presents an index nested loop join (INJ) approach for computing RCJ efficiently. Section 4 develops a bulk version of INJ to further optimize its I/O cost. Section 5 experimentally evaluates the efficiency of the proposed algorithms on both real and synthetic spatial pointsets. We also demonstrate the differences between the result set of RCJ and that of existing spatial join operations. Finally, Section 6 concludes the paper.

2. RELATED WORK

Section 2.1 reviews basic query processing techniques on a spatial index and Section 2.2 discusses the existing spatial join types.

2.1 Spatial Query Processing

The most common spatial access method is the R-tree [5], which hierarchically groups objects into disk pages and indexes them by minimum bounding rectangles (MBRs). Figure 2 shows a set of points and an R-tree that indexes them. The root node of this tree has three entries e_1, e_2, e_3, each having a pointer to a leaf node that stores data points. Observe that each non-leaf entry (say, e_2) is associated with its MBR, representing the minimum rectangle that encloses all points in its subtree.

Popular spatial queries such as range queries and nearest neighbor queries can be efficiently answered by R-trees. Given a region W, a *range query* returns all points that intersect W. In the example of Figure 2, a range query asks for the points within distance 3 from q (i.e., the shaded area). The root node of the R-tree is first visited. Then, the query is evaluated by following entries whose MBRs intersect the

query region. For instance, e_1 does not intersect the query region so the subtree of e_1 cannot lead to any query result. On the other hand, the subtree of e_2 is searched recursively and the points in the corresponding node are checked to obtain the result p_7.

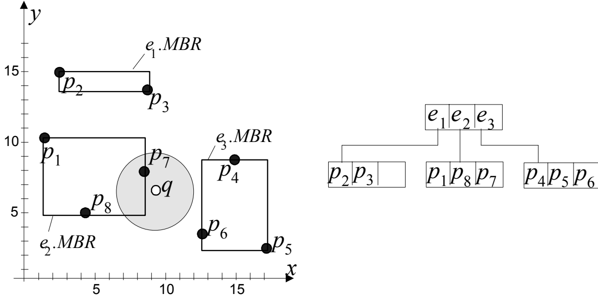


Figure 2: Spatial Queries on R-trees

Given a query point q and a positive integer k, a k nearest neighbor (kNN) query retrieves the k closest points to q. The *incremental nearest neighbor* (INN) algorithm [7] is the state-of-the-art solution for processing kNN query on R-tree. It employs a min-heap H that arranges visited R-tree entries e based on the minimum distance $mindist(q, e)$ [11] of their MBRs from q. First, entries of the root node are inserted into H. In each iteration, the top entry e of H is deheaped. If e is a non-leaf entry, then the child node of e is accessed and all its entries are inserted into H. Otherwise, e is a leaf node entry and the corresponding point is reported as the next NN. The procedure repeats until k points have been found. As an example, we apply the INN algorithm to process the 2-NN query of q in Figure 2. First, the root entries e_1, e_2, e_3 (together with their distances from q) are inserted into H. Then the nearest entry e_2 is retrieved from H, its child node is read, and points p_1, p_7, p_8 are inserted into H. The next nearest entry in H is p_7, which is reported as the first NN of q. Continuing with the above procedure, the next leaf entry deheaped from H (i.e., p_8) is the second NN of q.

The INN algorithm is not only applicable to NN search, but it has also been successfully extended to process other advanced spatial queries such as skyline retrieval [9] and reverse nearest neighbor search [16]. The rationale is that INN serves as a spatial ranking operator for retrieving the points in ascending order of their distances from q. The exact set of retrieved points can be decided on-demand according to specific application requirements (e.g., skyline search).

2.2 Spatial Joins

Given two spatial datasets P and Q, a spatial join returns a set of pairs $\langle p, q\rangle$ (where $p \in P$ and $q \in Q$), subject to the *join predicate*, which selects qualified pairs for the join. Examples of join predicates include the ϵ-distance [2, 8], the k-closest pairs [6, 3, 13], and the k nearest neighbor join [17].

Table 1 summarizes the result sizes and the conditions for a pair $\langle p, q\rangle$ to be qualified in the above spatial join types. The inclusion of a pair $\langle p, q\rangle$ in the result set depends on its pairwise distance $dist(p, q)$ (as compared to others) and user-specified parameters like ϵ, k. For instance, the ϵ-distance join retrieves all pairs $\langle p, q\rangle$ with $dist(p, q)$ below ϵ. On the other hand, the k closest pairs join retrieves k pairs such that their $dist(p, q)$ are smaller than other pairs in the Cartesian product $P \times Q$. Regarding the k nearest neighbor join, a pair $\langle p, q\rangle$ belongs to the result if q is one of the k-th nearest neighbor of p in the dataset Q. Observe that k nearest neighbor join produces asymmetric join pairs (i.e., different results after exchanging P and Q) while the other two return symmetric join pairs.

Join type	Condition for a qualified pair $\langle p, q\rangle$	Result size	Symmetric
ϵ-distance	$dist(p, q) \le \epsilon$	not fixed	yes
k closest pairs	k-th smallest $dist(p, q)$ among pairs in $P \times Q$	k	yes
k nearest neighbor join	$\forall q \in Q$, k-th smallest $dist(p, q)$	$k \cdot \lvert P\rvert$	no

Table 1: Summary of Spatial Join Types

We emphasize that RCJ is fundamentally different from these *distance-based* spatial joins. Their result pairs are decided by pairwise distances and parameters like ϵ, k. In contrast, RCJ is based on an intuitive, non-parametric, and *geometric* constraint, which solely determines the inclusion of a pair in the result (i.e., the enclosing circle of the pair contains no other points). Therefore, current spatial join processing techniques [2, 8, 6, 3, 13] are not applicable to RCJ.

To our knowledge, parameterless spatial join on pointsets has only been studied in [19], which defines the *common influence join* between two pointsets P and Q as pairs $\langle p, q\rangle$ such that the Voronoi cell of p (in P) intersects the Voronoi cell of q (in Q). Unfortunately, result pairs of common influence join cannot be exploited to determine RCJ results effectively — it is not clear how the Voronoi constraint in common influence join can be transformed to the ring constraint in RCJ.

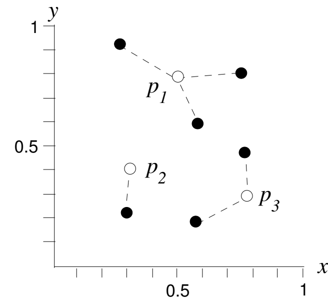


Figure 3: Top-k Influential Site Query

Recently, *influence-based* spatial queries [18, 4] have been proposed for discovering potentially useful points/locations from two spatial datasets. They differ from spatial joins in the sense that: (i) the result is a point/location (as opposed to pairs of points), and (ii) two datasets play different roles [18], where points in one dataset are labeled as *sites* and the others are marked as *objects*. The influence of a site is defined by the number of objects having it as the closest site. In the example of Figure 3, the sites (e.g., cinemas) are white points and the objects (e.g., residential buildings) are black points. Each object is linked to its nearest site by a

dotted line. Based on this concept, the top-k influential site query [18] retrieves the sites with top-k influence values. In Figure 3, the influence values of p_1, p_2, and p_3 are counted as 3, 1, and 2 respectively. Thus, the top-1 influential site is p_1. On the other hand, the optimal location query [4] aims to compute the location (as opposed to known sites) that maximizes the influence value (defined above).

3. INDEX NESTED LOOP JOIN

In our problem setting, we assume that each dataset (i.e., P, Q) is indexed by a disk-based R-tree (i.e., T_P, T_Q). Nevertheless, our methodology is directly applicable to other hierarchical spatial indexes (e.g., point quad-tree) as well.

In order to exploit the R-trees T_P and T_Q, we adopt a well-known query processing technique, called *index nested loop join* [14], for the efficient computation of the RCJ between P and Q. Although this solution sounds straightforward, it generates a number of non-trivial issues to be addressed, as we will show shortly.

Algorithm 1 presents a high-level description of the framework of our Index Nested Loop Join (INJ) solution. At Line 1, the loop iterates for each point q in Q (i.e., each leaf entry in T_Q). Then, we follow a filter-verification approach to compute RCJ pairs for the currently examined point q. In the *filter step* (Line 2), we probe the tree T_P, and retrieve a *candidate set* S of points that potentially form RCJ pairs with q. In the *verification step* (Lines 3-5), we examine each point $p \in S$ and access the trees T_P and T_Q. If the (smallest) enclosing circle of p and q contains no other points in P, Q, then the pair $\langle p, q \rangle$ is reported as a valid RCJ result pair.

Algorithm 1 Index Nested Loop Join Framework for RCJ

```
    algorithm INJ-Framework(R-tree T_Q, R-tree T_P)
    /* search ordering */
1:  for each leaf entry q in the tree T_Q do
        /* filter step */
2:      S:=probe T_P for candidates that match with q;
            /* verification step */
3:      for each p ∈ S do
4:          if ⟨p, q⟩ satisfies the circle constraint then
5:              report ⟨p, q⟩ as a result;
```

In the following, we identify important issues to be investigated in the remainder of this section.

Filter Step.

The implementation of the filter step (Line 2) is essential to the overall performance of our algorithm. A simple approach is to declare the candidate set S as the set of points in P. This technique is ineffective and places a heavy burden on the verification step. Another extreme is to include into S only the nearby points of q (from P). However, this technique is not guaranteed to discover all points of P that form RCJ pairs with q. In Section 3.1, we propose an effective, efficient, and correct implementation of the filter step.

Verification Step.

To verify whether a pair $\langle p, q \rangle$ belongs to the result (Lines 3-5), a simple method is to define the (smallest) enclosing circle of the pair and then perform range searches on both trees T_P and T_Q. In Section 3.2, we develop a more efficient verification technique, without necessarily accessing data points (i.e., leaf entries) in T_P and T_Q.

Correctness of the Algorithm.

In Section 3.3, we elaborate on the Index Nested Loop Join (INJ) Algorithm for RCJ evaluation, by incorporating the above filter and verification steps. In addition, we will establish the correctness of INJ, i.e., (i) no duplicate RCJ pairs are produced, (ii) the result set contains no false positive pairs, (iii) there are no false negatives.

Search Order.

We observe that, the ordering of examining the points in Q (Line 1) also plays an important role in the performance of the algorithm. Section 3.4 discusses an appropriate ordering of visiting the points in Q.

3.1 The Filter Step

We now examine the filter step (at Line 2 of Algorithm 1) and propose efficient techniques for retrieving a candidate set S of points in P that form potential RCJ pairs with a given point $q \in Q$.

Reducing the Candidate Set.

In the following, we devise an effective technique to detect unqualified points in P (that cannot produce RCJ pairs with q) early, so that their expensive verifications can be saved.

Suppose that we have encountered a point $p \in P$ during the search. It is important to exploit p for filtering out other points $p' \in P$ from joining with q. We first define the notations below to facilitate our discussion.

DEFINITION 1. ***Ψ^+ and Ψ^- regions.***
Given a point $q \in Q$ and a point $p \in P$, let $L(q,p)$ be the line passing through p and perpendicular to the line segment $|qp|$. The spatial domain is divided by $L(q,p)$ into two regions: (i) the region $\Psi^+(q,p)$ that contains q, and (ii) the region $\Psi^-(q,p)$ that does not contain q.

We present the following lemma for eliminating a point p' from search, based on the locations of q and p.

LEMMA 1. ***Pruning a point $p' \in P$.***
Let p be a point of P and q be a point of Q. Any other point $p' \in P$ located in $\Psi^-(q,p)$ cannot join with q to form an RCJ pair.

PROOF. Figure 4 shows an exemplary spatial configuration of points p, q, and p'. Let C' be the smallest enclosing circle of q and p'. The diameter of C' (between q and p') must intersect the line $L(q,p)$ at a location z (shown as a gray point). Let C be the smallest enclosing circle of q and z. Since the line segment $|qp'|$ covers the line segment $|qz|$, any point lying on (or inside) C must fall in C'.
As the line segment $|qp|$ is perpendicular to $L(q,p)$, the angle $\angle qpz$ is an right angle (i.e., 90°). Thus, p lies on C, which is inside C' as explained above. As a result, the pair $\langle p', q \rangle$ is not an RCJ pair. □

Maximality of the Pruning Region.

Lemma 1 defines a valid pruning region $\Psi^-(q,p)$ in which any other point p' cannot form RCJ pairs with q. We now investigate whether it is possible to exploit p for deriving a larger pruning region (than $\Psi^-(q,p)$). In other words, does

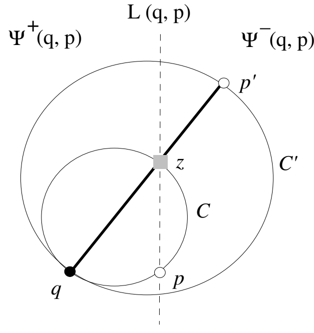


Figure 4: Geometric Construction of Lemma 1

there exist some p' in the region $\Psi^+(q,p)$ that can also be pruned by p? It turns out that the answer is negative, as explained by the lemma below.

LEMMA 2. ***Maximal pruning region.***
Let $p' \in P$ be a point located in the region $\Psi^+(q,p)$. The validity of the pair $\langle p', q \rangle$ (as an RCJ pair) is independent of the point p.

PROOF. By constructing a line parallel to $L(q,p)$ and passing through q, we partition the region $\Psi^+(q,p)$ into three parts, as depicted in Figure 5a. Region I is the area that contains points (e.g., p'_1) between q and p, region II is the area with points (e.g., p'_2) behind q, and region III is the line through q (e.g., covering p'_3).

Figure 5b illustrates the first case. We construct the location z in the same way as the proof of Lemma 1, i.e., z is the location on $L(q,p)$ which is collinear with q and p'. Observe that the enclosing circle C' of the pair $\langle p', q \rangle$ is completely covered by the circle C passing through q, p, and z. As a result, p does not fall in C' and cannot be used to prune the pair $\langle p', q \rangle$.

Figure 5c depicts the second case. Except from the point q, the circles C' and C are disjoint. Again, p cannot fall in C'.

Figure 5d exemplifies the third case, where the line segment $|qp'|$ is parallel to $L(q,p)$. Here, we define the location μ as the center of the enclosing circle C' between q and p'. Since $\angle \mu q p$ is an right angle, the segment $|\mu p|$ must be no shorter than the segment $|\mu q|$. Hence, we can construct a circle C concentric to C' such that C passes through the point p. Note that p does not fall in C', due to the concentric relationship between the circles C and C'.

Combining all three cases, we conclude that the validity of the pair $\langle p', q \rangle$ (as an RCJ pair) is independent of the point p. In other words, the pruning region formulated by Lemma 1 is maximal. □

Saving Filtering I/O Cost.

Having confirmed the completeness of the above pruning rule, we now develop techniques for reducing the filter cost, by accessing as few points as possible from T_P, the R-tree of P. For this purpose, we exploit a discovered point $p \in P$ to prune unqualified subtrees of T_P that cannot contain points for forming RCJ pairs with q. The lemma below shows when an MBR e (for a non-leaf entry in the tree T_P) becomes unqualified for joining with q.

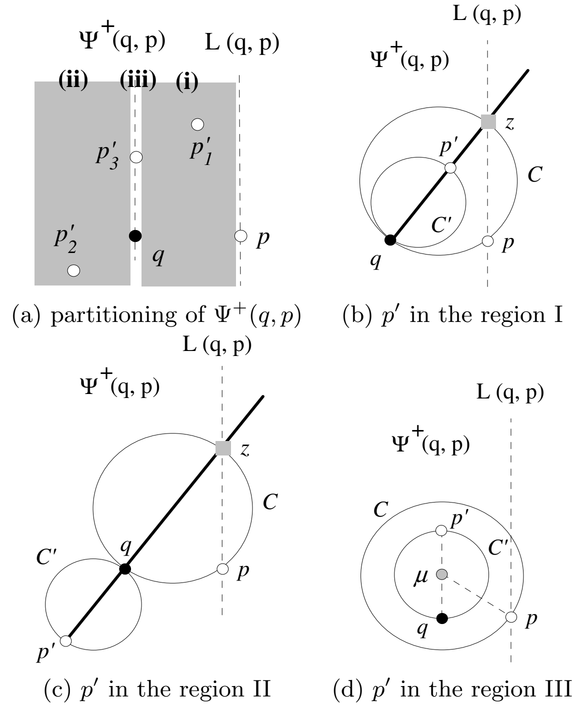


Figure 5: Locations of p' in Unpruned Region $\Psi^+(q,p)$

LEMMA 3. ***Pruning an MBR** e **of some points in** P.*
Given two points $p \in P$, $q \in Q$, and an MBR e of some points in P. If $\Psi^-(q,p)$ contains e, then any point p' inside e cannot join with q to form RCJ pairs.

PROOF. True, due to Lemma 1 and the bounding property of e. □

The Filter Algorithm.

Until now, we have only discussed how to exploit an accessed point p for pruning the search space. We continue to investigate how to obtain such a point p (from P). Observe in Figure 4 that, if p is close to q, then the pruning region becomes large and many unqualified points in P can be pruned away. In order to maximize the pruning power, it is desirable to discover points of P in ascending order of their distances from q. Hence, we integrate the incremental NN algorithm [7] (described in Section 2.1) with our pruning techniques into the filter algorithm.

Algorithm 2 corresponds to the pseudo-code of the filter algorithm, which discovers all potential points in (the tree T_P of) P that can form RCJ pairs with the point $q \in Q$. The set S is used to maintain the candidate points found so far. As in [7], we employ a min-heap H for organizing the entries in the tree T_P to be visited. First, all root entries of the tree T_P are inserted into H with their *mindist* value from q. At Line 6, we deheap an entry e (with the smallest *mindist* value) from H. Recall that points in S come from the tree T_P and they may be used to prune e (by using Lemmas 1 or 3). In case e is pruned, we discard it and continue the loop at Line 5. Otherwise, we check whether e is a non-leaf entry. If e is a non-leaf entry, then we access its child node N' and insert all entries of N' into H. Otherwise, e corresponds to a

point and we simply insert it into S. The process continues until H becomes empty.

Algorithm 2 Filter Algorithm

algorithm Filter(Point q, R-tree T_P)
1: S:=new set;
2: H:=new min-heap (with elements of $(entry, key)$);
3: **for each** entry $e \in T_P.root$ **do**
4: insert $(e, mindist(q, e))$ into H;
5: **while** H is not empty **do**
6: deheap e from H;
7: **if** $\exists p \in S$, $\Psi^{-}(q, p)$ contains e **then**
8: discard the entry e; goto Line 5;
9: **if** e is a non-leaf entry **then**
10: read the child node N' pointed by e;
11: **for each** entry $e' \in N'$ **do**
12: insert $(e', mindist(q, e'))$ into H;
13: **else** ▷ e is a point
14: insert e into S;
15: return S;

Example of the Filter Algorithm.
The graphical example in Figure 6 illustrates how the filter algorithm works. In this example, the R-tree T_P (of the dataset P) has a root node with four non-leaf entries e_1, e_2, e_3, e_4, and each of them points to a leaf node. First, the root entries (e_1, e_2, e_3, e_4) are inserted into H. Then, e_1 is deheaped (as it is the closest to q) and its points p_1, p_2, p_3 are inserted into H. Next, p_1 is deheaped and it is inserted into the set S. After e_2 is deheaped, its points are inserted into H. The next deheaped entry is p_4. Since p_4 does not fall in the pruning region of any point in S (e.g., the region $\Psi^{-}(q, p_1)$), it is inserted into S. Observe that the entries to be deheaped later (i.e., $e_3, e_4, p_2, p_3, p_5, p_6$) will be pruned by some point in $S = \{p_1, p_4\}$. Finally, H becomes empty, the filter algorithm terminates and returns S as the candidate set.

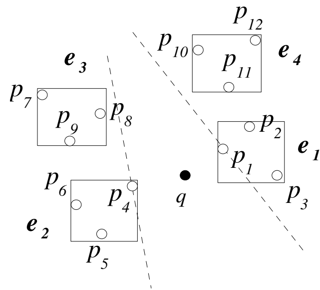


Figure 6: Example of the Filter Step

3.2 The Verification Step

This section elaborates the verification step in our framework (Lines 3–5 of Algorithm 1). Pairs that pass through the filter step need to be verified against both datasets P and Q. Recall that a pair $\langle p_i, q_j \rangle$ is an RCJ result when it satisfies the geometric constraint: the smallest circle enclosing p_i and q_j contains no other points in P, Q. We focus on the verification of the enclosing circle against Q because the verification procedure against P is the same.

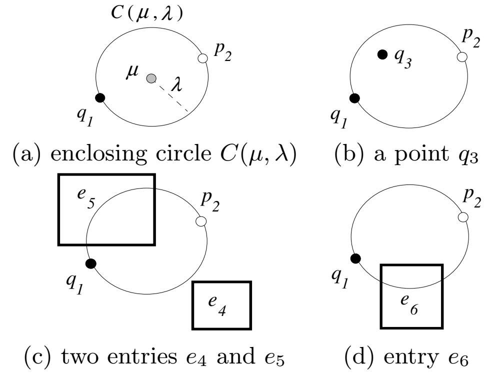
Case Study.
Figure 7a shows a pair $\langle p_2, q_1 \rangle$ and its enclosing circle $C(\mu, \lambda)$, where μ is the circle center and λ is the circle radius. In the following, we illustrate various cases of the verification step. We assume that, the point q_3 and the non-leaf entries e_4, e_5, e_6 in Figures 7b,c,d reside in the R-tree index on Q.

- **Point Inside the Circle.** In Figure 7b, the point q_3 falls in the circle so the pair $\langle p_2, q_1 \rangle$ is pruned.

- **Disjoint Entry.** In Figure 7c, the non-leaf entry e_4 does not intersect the circle. Since e_4 is irrelevant to the verification of the pair, the subtree of e_4 is not accessed.

- **Intersecting Entry.** In Figure 7c, the non-leaf entry e_5 intersects the circle and its subtree may contain some point inside the circle. Thus, the subtree of e_5 needs to be accessed.

- **Entry with a Face Inside the Circle.** In Figure 7d, the non-leaf entry e_6 has one face (i.e., a side in 2D space) inside the circle. According to the property of the minimum bounding rectangle [11, 15], it is guaranteed that the subtree of e_6 must contain a point inside the circle. Therefore, the pair $\langle p_2, q_1 \rangle$ is pruned, without accessing the subtree of e_6.



(a) enclosing circle $C(\mu, \lambda)$ (b) a point q_3
(c) two entries e_4 and e_5 (d) entry e_6

Figure 7: Various Cases of the Verification Step

The Verification Algorithm.
Here, we summarize the above cases and extend the technique for verifying a set of circles concurrently. Algorithm 3 is the pseudo-code of the verification algorithm and its input parameters are: (i) the current node N (of an R-tree) being accessed, and (ii) a set S of circles to be verified. Initially, the node N is set to the root node of the tree of Q (or P).

In case the entry e is a data point (i.e., N is a leaf node), we examine each circle $C \in S$ and discard those containing e.

Otherwise, the entry e is a non-leaf entry (i.e., N is a non-leaf node). First, we eliminate each circle $C \in S$, which contains a face of e. Then, we check whether e intersects any remaining circle $C \in S$. If so, then the subtree of e

may contain some point falling in some circle $C \in S$, and we perform the verification procedure recursively at the child node of e.

It is well known that plane-sweep [2] is an efficient method for detecting the intersection between two groups of rectangles. This technique can be applied to accelerate the comparisons at Lines 3, 5, and 6 of the verification algorithm. For the sake of readability, we do not discuss the details of plane-sweep here.

Algorithm 3 Verification Algorithm

```
    algorithm Verify(Node N, Set S)
1:  for each entry e ∈ N do
2:      if e is a point then
3:          remove each C ∈ S, where C contains e;
4:      else                                  ▷ e is a non-leaf entry
5:          remove each C ∈ S, where C contains a face of e;
6:          if ∃ C ∈ S, C intersects e then
7:              read the child node N' pointed by e;
8:              Verify(N', S);
```

3.3 The INJ Algorithm

Having explained the filter step and the verification step, we are ready to present the Index Nested Loop Join (INJ) Algorithm for RCJ evaluation (see Algorithm 4). It takes the R-trees T_P and T_Q (on datasets P and Q) as input. At Lines 1–2, we iterate each leaf node N of the tree T_Q and examine each of its point q. First, we run the filter algorithm (i.e., Algorithm 2) to obtain the set S' of candidate points from P that may form RCJ pairs with the (currently examined) point q. Then, for each retrieved point $p \in S'$, we define its smallest enclosing circle C with q, and store the circle into the set S. At Line 9, we invoke Algorithm 3 in order to verify the circles of S against the points in tree T_Q. The remaining circles of S are then verified against the points in the tree T_P (at Line 10). Eventually, each circle in S is reported as a RCJ result.

Algorithm 4 Index Nested Loop Join Algorithm for RCJ

```
    algorithm INJ(R-tree T_Q, R-tree T_P)
1:  for each leaf node N of the tree T_Q do
2:      for each point q in N do
3:          S:=∅;                          ▷ set of candidate pairs
4:          S':=Filter(q, T_P);                       ▷ filter step
5:          for each p ∈ S' do
6:              let C be the enclosing circle of q and p;
7:              C.p:=p; C.q:=q;
8:              insert C into S;
9:          Verify(T_Q.root, S);               ▷ verification step
10:         Verify(T_P.root, S);               ▷ verification step
11:         for each C ∈ S do
12:             report ⟨C.p, C.q⟩ as a result;
```

We establish the correctness of the above INJ algorithm, by the following lemma.

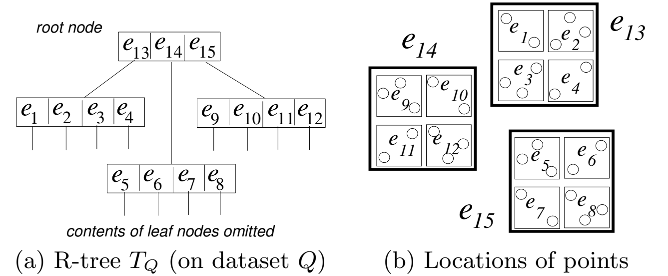
LEMMA 4. ***Algorithm correctness.***
The above algorithm produces exactly all the RCJ results for the point $q \in Q$, i.e., the algorithm has no false negatives, and the reported result set contains no duplicate pairs, no false positive pairs.

PROOF. First, no result is missed (i.e., no false negatives) as only unqualified points in P are eliminated in the filter step (according to the pruning rules in Lemmas 1 and 3). Second, no duplicate result is generated as each point $p \in P$ is considered at most once, with respect to the same point q. Third, each candidate RCJ pair $\langle p_i, q_j \rangle$ is verified against P and Q to ensure no false positives (i.e., the enclosing circle of the pair does not contain other points in P, Q). □

3.4 Search Order

At Line 1 of Algorithm 4, each leaf node N of the tree T_Q is visited in order to compute all RCJ result pairs. However, it does not specify a particular traversal order of accessing leaf nodes from T_Q.

We now investigate the effect of the traversal ordering on the efficiency of the algorithm. Figure 8 shows the structure of the R-tree T_Q and the corresponding locations of points in the dataset Q. Apparently, a random order of accessing the leaf nodes would lead to poor performance. For instance, suppose that we first process the points in e_{11} and then process points in e_2. Since e_{11} and e_2 are far apart, invocations of the filter and verification algorithms on their points may access completely different nodes of the trees T_P and T_Q. In this case, there is no locality of data accesses, leading to expensive join evaluation.



(a) R-tree T_Q (on dataset Q) (b) Locations of points

Figure 8: R-tree Example

In real-world applications, a small memory buffer is employed usually to exploit data access locality and reduce the number of page accesses from the disk. We suggest that an appropriate search ordering is to apply the depth-first traversal on the tree T_Q. We formally present this method in Algorithm 5. The input parameter N specifies the currently examined node of the tree T_Q and it is initially set to the root node of T_Q. If the entry e (in N) is a non-leaf entry, then we continue the search recursively on its child node (see Lines 3-4). Otherwise, the entry e is a data point q and we apply the INJ algorithm on q in order to compute RCJ pairs.

Algorithm 5 Depth-First Index Nested Loop Join for RCJ

```
    algorithm INJ_DF(Node N, R-tree T_Q, R-tree T_P)
1:  for each entry e ∈ N do
2:      if e is a non-leaf entry then
3:          read the child node N' pointed by e;
4:          INJ_DF(N', T_Q, T_P);
5:      else                                  ▷ e is a point q
6:          apply Lines 3–12 of Algorithm 4 for the point q;
```

Figure 8 illustrates the operation of this algorithm; the

leaf nodes are processed in the order: e_1, e_2, e_3, e_4, e_5, etc. Thus, adjacent invocations of the filter and verification procedures may share some common paths in the trees T_P and T_Q. This way, the data access locality is exploited, and the I/O cost can be improved significantly with a small memory buffer.

4. BULK INDEX NESTED LOOP JOIN

In Section 4.1, we apply bulk computation techniques to reduce the cost of the algorithm. Finally, Section 4.2 suggests a powerful pruning rule for further optimizing the algorithm.

4.1 Bulk RCJ Computation

In the last section, RCJ pairs are computed for each point $q \in Q$ separately. Since the filter and verification steps are performed for each q, the total number of R-tree traversals is proportional to $|Q|$. Thus, the processing cost becomes high for large dataset Q.

Fortunately, the number of R-tree traversals can be significantly reduced by performing RCJ computation concurrently for all points q residing in the same leaf node of the tree T_Q. We present the bulk version of Index Nested Loop Join, called BIJ (Algorithm 6), to improve the total I/O cost. It computes RCJ pairs for all points in a leaf node N (of the tree T_Q) concurrently. Note that the algorithm relies on the bulk implementation of the filter algorithm Bulk_Filter which will be elaborated shortly. Set V contains all points in the node N, and each $q \in V$ is associated with its own candidate set $q.S$. We invoke the Bulk_Filter function to retrieve candidates for each point in V. Next, we define enclosing circles for the candidates and verify them against the trees T_Q and T_P. Finally, we report the remaining candidates as RCJ pairs.

Algorithm 6 Bulk Index Nested Loop Join for RCJ

```
    algorithm BIJ(R-tree T_Q, R-tree T_P)
 1: apply depth-first traversal on the tree T_Q;
 2: for each encountered leaf node N do
 3:     S:=∅;                                  ▷ set of candidate pairs
 4:     V:=∅;                                  ▷ set of points
 5:     for each q ∈ N do
 6:         q.S:=∅;          ▷ set of candidate points for point q
 7:         insert q into V;
 8:     Bulk_Filter(V, T_P);                   ▷ filter step
 9:     for each q ∈ V do
10:         for each p ∈ q.S do
11:             let C be the enclosing circle of q and p;
12:             C.p:=p; C.q:=q;
13:             insert C into S;
14:     Verify(T_Q.root, S);                   ▷ verification step
15:     Verify(T_P.root, S);                   ▷ verification step
16:     for each C ∈ S do
17:         report ⟨C.p, C.q⟩ as a result;
```

We proceed to discuss the implementation of Bulk_Filter. Bulk_Filter (Algorithm 7) differs from the original filter (Algorithm 2) in the following aspects:

- A location $\overline{q}$ is defined as the centroid location of points $q \in V$. Data points in the tree T_P are then examined in ascending order of their distances from $\overline{q}$ (instead of each individual $q \in V$).
- At Line 7, if the entry e can be pruned with respect to **all** $q \in V$, then we discard it and continue the loop at Line 5
- At Lines 14-16, the entry e is examined against each $q \in V$ separately; for the currently examined q, e is inserted into the candidate set $q.S$ (of q) when e cannot be pruned with respect to q.

Algorithm 7 Bulk Filter Algorithm

```
    algorithm Bulk_Filter(Set V, R-tree T_P)
 1: H:=new min-heap (with elements of (entry, key));
 2: let q̄ be the centroid location of points q ∈ V;
 3: for each entry e ∈ T_P.root do
 4:     insert (e, mindist(q̄, e)) into H;
 5: while H is not empty do
 6:     deheap e from H;
 7:     if ∀q ∈ V, ∃p ∈ q.S, Ψ⁻(q, p) contains e then
 8:         discard the entry e; goto Line 5;
 9:     if e is a non-leaf entry then
10:         read the child node N′ pointed by e;
11:         for each entry e′ ∈ N′ do
12:             insert (e′, mindist(q̄, e′)) into H;
13:     else                                   ▷ e is a point
14:         for each q ∈ V do
15:             if ∀p ∈ q.S, Ψ⁻(q, p) does not contain e then
16:                 insert e into q.S;
```

4.2 Optimizing the Bulk Filter

In Section 3.1, we have shown how to reduce the search space for a given point $q \in Q$, using an encountered point p from P. We now consider another similar technique for search space reduction. Suppose that we have encountered two points q and q' from Q, but no points from P. The following lemma provides us a search space reduction method by exploiting q'.

LEMMA 5. ***Symmetric pruning rule.***
Given two points q and q' in Q, let $L(q, q')$ be the line passing through q' and perpendicular to the line segment $|qq'|$. Any point $p' \in P$ located in $\Psi^-(q, q')$ cannot join with q as RCJ result.

PROOF. The logic of the proof (see Figure 9) is the same as the one in Lemma 1, except that the point p (in Lemma 1) is replaced by the point q'. □

A practical issue is how the point q' should be chosen (from Q) for applying Lemma 5 on the point q. It is desirable that: (i) q' should be close to q, in order to have high pruning power, and (ii) no extra I/O cost is required for obtaining q'. Next, we demonstrate how Bulk_Filter (Algorithm 7) is adapted to incorporate the above pruning rule for enhancing the filtering effectiveness. In Bulk_Filter, the input parameter V contains a set of points in a leaf node of the tree T_Q. Fortunately, these points satisfy both of the above requirements as they reside in the same leaf node as the point q. In order to utilize the pruning power of points in the set V, at the condition of Lines 7 and 15, we replace the term $p \in q.S$ by the following:

$$p \in q.S \cup (V - \{q\})$$

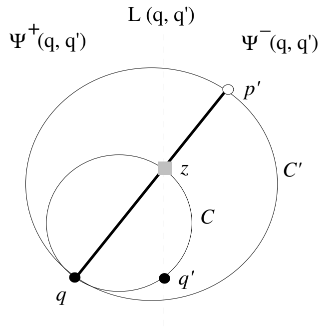


Figure 9: Geometric Construction of Lemma 5

Point p refers to a point that can be used for pruning the search space of point q. In this case, p can be a point in the candidate set $q.S$ or any point in the set V, except from the point q itself. The above pruning condition offers the benefit that, even when the candidate set $q.S$ is empty, the points $q \in V$ can help in shrinking the search space.

5. EXPERIMENTAL EVALUATION

This section evaluates the efficiency of our proposed algorithms using synthetic and real datasets. Uniform synthetic datasets (UI data) were generated by assigning random values to coordinates of points independently. In order to obtain meaningful RCJ results from real datasets, data points of both datasets P and Q should span over the same geographical region. For this purpose, we obtained several real 2D spatial pointsets of U.S., from the U.S. Board on Geographic Names[1]. Table 2 summaries their characteristics and Table 3 depicts the corresponding join combinations with them. Coordinate values in all datasets are normalized to the interval [0, 10000]. Each dataset is indexed by an R*-tree [1] with disk page size of 1K bytes.

ID	Description	Cardinality
PP	Populated Places	177983
SC	Schools	172188
LO	Locales	128476

Table 2: Summary of Real Datasets

Combination	Dataset Q	Dataset P
SP	SC	PP
SP′	PP	SC
LP	LO	PP
LP′	PP	LO

Table 3: Join Combinations

We compare the performance of our proposed RCJ algorithms:

- INJ (Index Nested Loop Join), Algorithm 5
- BIJ (Bulk Index Nested Loop Join), Algorithm 6
- OBJ (Optimized Bulk Index Nested Loop Join), an optimized version of Algorithm 6 with the symmetric pruning rule in Section 4.2

[1]http://geonames.usgs.gov/index.html

The algorithms were implemented in C++ and experiments were performed on a Pentium D 2.8GHz PC with 1GB memory. In practical applications, a small memory buffer is employed to exploit the locality of data accesses and reduce the number of page faults. We set the default size of the memory buffer to 1% of the sum of both tree sizes (for P and Q). We measured both I/O time and CPU time of the algorithms, by charging 10ms per page fault (a typical value [14]). Observe that, I/O time captures the number of page faults while CPU time roughly models the total number (including repeated) of R-tree node accesses.

Section 5.1 demonstrates the differences of the RCJ result set from those of existing spatial join operators. Section 5.2 studies the performance of our RCJ algorithms with respect to different factors.

5.1 Result Sets of RCJ and Other Spatial Joins

From our early discussion in the Introduction, we understand that RCJ is defined based on a geometric constraint whereas existing spatial join operators (e.g., the ϵ-range join, the k closest pairs, the k nearest neighbor join) are defined based on the distances between the points. Due to this fundamental difference, RCJ cannot be reduced into those spatial join operators.

Here, we aim to demonstrate empirically the differences between their result sets. Let S be the result set of RCJ and S' be the result set of another spatial join operator. From the information theoretic point of view, the precision and recall of S' with respect to S are defined as:

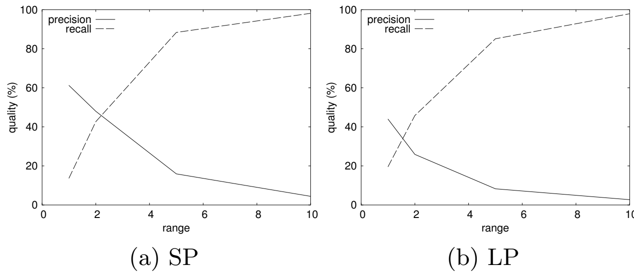
$$precision(S', S) = \frac{|S \cap S'|}{|S'|} \cdot 100\%$$

$$recall(S', S) = \frac{|S \cap S'|}{|S|} \cdot 100\%$$

Observe that the possible values of precision and recall fall between 0% and 100%. S' resembles S when both the recall and precision values are high.

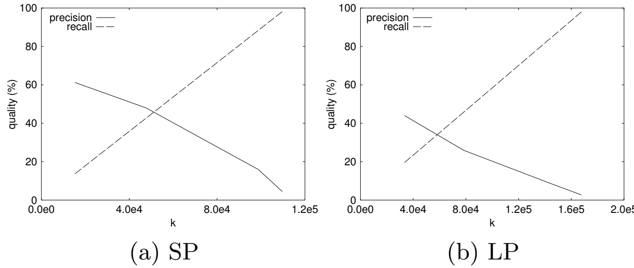
First, we study the effect of the join distance ϵ on the precision and recall values of the ϵ-range join result set with respect to the RCJ result set. Figure 10 shows the precision and recall values as a function of ϵ, for the join combinations SP and LP. There are many false negatives at low ϵ and the number of false positives becomes high at high ϵ. When ϵ increases, the precision decreases and the recall increases. Observe that no single ϵ value achieves both high precision and recall.

Next, we repeat the above experiment for the k-closest pairs join and the k nearest neighbor join. Figure 11 plots the precision and recall values with respect to the parameter k, for the k-closest pairs join. Figure 12 illustrates the precision and recall values as a function of k, for the k nearest neighbor join. Note that the trends in both figures follow the one in Figure 10. In summary, the result set of RCJ is significantly different from that of existing spatial join types, even when their parameters (e.g., ϵ or k) are fine-tuned to maximize the resemblance.



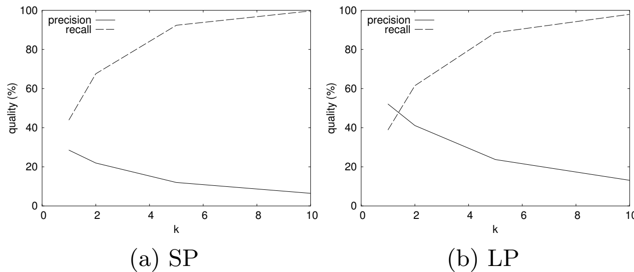
(a) SP (b) LP

Figure 10: Resemblance of ϵ-Range Pairs vs ϵ, Real Data



(a) SP (b) LP

Figure 11: Resemblance of k-Closest Pairs vs k, Real Data



(a) SP (b) LP

Figure 12: Resemblance of k Nearest Neighbor Pairs vs k, Real Data

According to our experimental result, the number of RCJ result pairs are 111763 and 171139 for the join combinations SP and LP respectively. Due to the geometric constraint of RCJ, its result cardinality is comparable to the input data size and does not overwhelm the user. On the other hand, the result size of existing spatial join operator (e.g., the ϵ-range join) is either too small or too large as an appropriate parameter value (e.g., ϵ) cannot be easily determined apriori.

5.2 Efficiency of RCJ Computation

Performance on Real Datasets.

Before studying the performance of our algorithms, we investigate the sizes of their candidate sets on real data. Recall that an efficient algorithm should generate small number of candidates in order to minimize the verification cost. Table 4 shows the candidate size of our RCJ algorithms on real data. As mentioned in the Introduction, the brute-force solution (denoted by BRUTE) takes the complete Cartesian product between P and Q as the candidate set. For the basis of comparison, we also include the actual number of RCJ results. Observe that the candidate size of INJ is four orders of magnitude lower than BRUTE, confirming the effectiveness of our pruning rule in reducing the search space. Since BIJ performs bulk computation for each group of points (in the same leaf node) concurrently, the tree traversal order in filter and verification steps may not be optimal with respect to each individual point in the group. Thus, BIJ has a larger candidate size than INJ. Nevertheless, the bulk computation of BIJ reduces the number of tree traversals significantly and BIJ has better performance than INJ in general, as we will see later. Clearly, the candidate size of OBJ is on average 30% of INJ and it stays very close to the actual number of RCJ results.

Algorithm	**Data combination**	
	SP	LP
BRUTE	3.06E+10	2.28E+10
INJ	767570	571289
BIJ	1161214	1243187
OBJ	175189	227352
RCJ Results	111763	171139

Table 4: Number of Candidate Pairs, Real Data

We proceed to study the performance of INJ, BIJ, and OBJ with respect to different factors. BRUTE is excluded from subsequent experiments due to its huge candidate size.

Figure 13 shows the cost (execution time) of the algorithms for the join combinations in Table 3. The execution time is decomposed into I/O time and CPU time. BIJ performs better than INJ because the bulk computation technique greatly reduces the number of node accesses (and thus saving CPU time from processing them). The reduction in I/O time is not large due to the memory buffer. Since OBJ applies an additional pruning technique (in Section 4.2), it outperforms its competitors. Recall in Section 3.4 that the size of the tree T_Q affects the number of tree traversal operations on the other tree T_P. Hence, the join combination LP (with a smaller T_Q than that of LP$'$) performs better than LP$'$. It is worth noticing that, OBJ has robust performance across different join combinations.

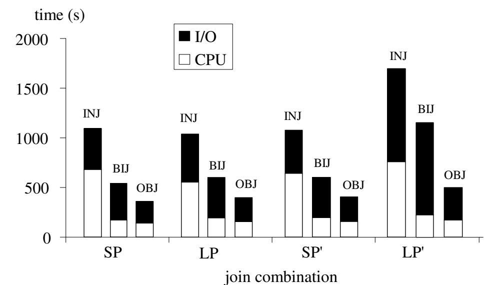


Figure 13: The Effect of Join Combination, Real Data

Experiments on Synthetic Datasets.

All of our RCJ algorithms follow a filter-verification approach for RCJ computation. We now examine the cost of

the verification step relative to the total cost. Figure 14 shows the cost of the algorithms on synthetic UI datasets, at $|P| = |Q| = 200K$. The left column refers to our RCJ algorithms (with both filter and verification steps), whereas the right column represents the variants of our algorithms without the verification step. Observe that the corresponding cost differences between these two columns are small, reflecting the high efficiency of our verification step. Since the filter step in our algorithms is effective in discarding unqualified pairs, only a small number of candidates need to be verified against both datasets P and Q. Thus, the verification step becomes very efficient, incurring less than 25% of the total cost.

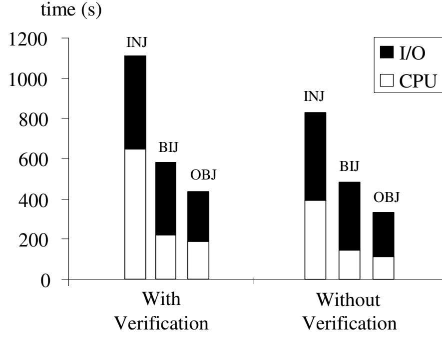


Figure 14: The Cost of RCJ Algorithms, $|P| = |Q| = 200K$, Synthetic UI Data

Figure 15 plots the performance of the algorithms (on synthetic data) as a function of the buffer size, which is expressed as the fraction of total tree sizes. As the buffer size increases, more R-tree nodes can be cached in the buffer and the I/O time of all algorithms falls. OBJ outperforms the other algorithms in all cases. At low buffer size, the performance gap between OBJ and its competitors widens.

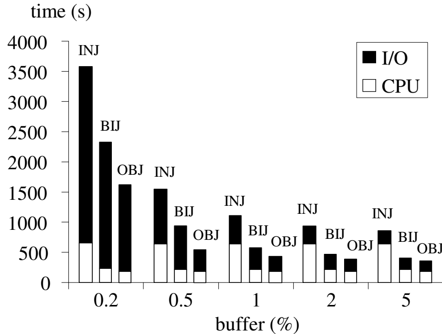
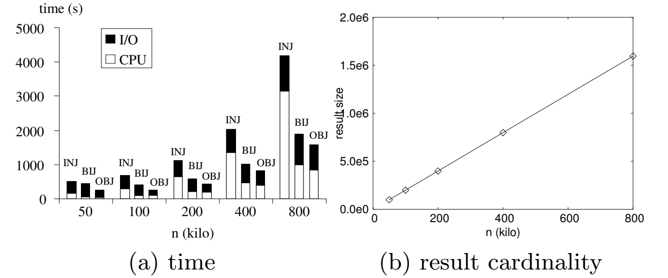


Figure 15: The Effect of Buffer Size, $|P| = |Q| = 200K$, Synthetic UI Data

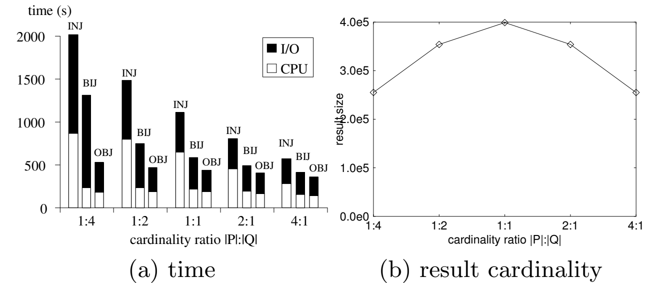
Next, we test the scalability of the algorithms with synthetic datasets. Figure 16a shows the cost of the algorithms with respect to the data size n, where both datasets P and Q have n (kilo) points each. All three algorithms are scalable for large datasets. In particular, the performance gap between OBJ and its competitors widens as n increases. Besides, Figure 16b shows the size of the RCJ result set as a function of n. The result cardinality grows linearly with n.

We then study the performance of the algorithms on synthetic datasets with respect to the cardinality ratio $|P| : |Q|$, by fixing the sum of data sizes $|P| + |Q|$ to 400K points. Figure 16a plots the cost of the algorithms for different cardinality ratio $|P| : |Q|$. As the ratio increases, the size of Q decreases, fewer filter and verification operations have to be executed, and thus the cost decreases. Observe that OBJ has stable performance across different cardinality ratio. Figure 16b shows the result cardinality of RCJ with respect to the ratio $|P| : |Q|$. The result size is maximized when both P and Q have the same size.



(a) time (b) result cardinality

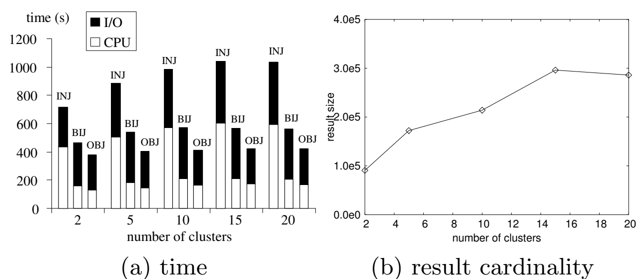
Figure 16: The Effect of Data Size n, $|P| = |Q| = n$, Synthetic UI Data



(a) time (b) result cardinality

Figure 17: The Effect of Cardinality Ratio $|P| : |Q|$, $|P| + |Q| = 400K$, Synthetic UI Data

We proceed to examine the performance of the algorithms with respect to different data distributions. Given a parameter w, we generate w clusters of points in the dataset P (Q) such that (i) all clusters have the same number of points, (ii) the center of each cluster is randomly chosen in the space domain $[0, 10000]^2$, and (iii) points in the same cluster follow Gaussian distribution, with standard deviation 1000 along each dimension. Figure 18a shows the cost of the algorithms on synthetic Gaussian datasets P (and Q), as a function of the number of clusters w, by fixing each data size $|P|$ ($|Q|$) to 200K points. As w increases, the points are distributed more evenly in the space and the datasets become less skewed. Obviously, OBJ outperforms its competitors and its performance is less sensitive to the data distribution. Figure 18b plots the result cardinality of RCJ with respect to w. When w increases, the result size first increases and then tends to be stabilized as the data distribution becomes less skewed.



(a) time (b) result cardinality

Figure 18: The Effect of Number of Clusters w, $|P| = |Q| = 200K$, Synthetic Gaussian Data

6. CONCLUSIONS

In this paper, we have introduced the interesting problem of ring-constrained join (RCJ), which has various applications in decision support systems. Besides, RCJ is a challenging problem as there are no existing I/O-efficient methods for processing it. Motivated by this, we have exploited the characteristics of RCJ and developed a fundamental lemma for reducing the search space effectively. We then applied the lemma for R-tree based search and presented several algorithms for evaluating RCJ.

Our solutions are experimentally evaluated on both real and synthetic datasets. The algorithm OBJ outperforms its competitors in all cases, scales well with large datasets, and its performance is robust across different data distributions (including real datasets). In addition, we demonstrate that the result set of RCJ cannot be well expressed in terms of conventional spatial join types (e.g., the ϵ-distance join, the k-closest pairs, the k nearest neighbor join).

In the future, we first plan to devise accurate I/O cost models for our proposed algorithms, by analyzing the effect of their pruning techniques on search space reduction. Our second research direction is to determine the theoretical upper bound of RCJ result size. In our experimental evaluation, we observed that the result cardinality is linear to the input data size. We intend to study the result size for the "worst" possible data distributions. Yet another important research topic is the generalization of RCJ to other contexts. Recall that our current RCJ definition is built on the concept of circle/ring, a geometric shape exclusively defined in the Euclidean space. Therefore, we need to explore alternative definitions of the circle constraint and its center for other distance metrics: (i) the Manhattan distance (as opposed to the Euclidean distance), and (ii) the shortest path distance along a road network [12] that restricts the locations of points.

7. ACKNOWLEDGEMENT

This work was supported by grant HKU 7155/06E from Hong Kong RGC.

8. REFERENCES

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, 1990.

[2] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient Processing of Spatial Joins Using R-Trees. In *SIGMOD*, 1993.

[3] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest Pair Queries in Spatial Databases. In *SIGMOD*, 2000.

[4] Y. Du, D. Zhang, and T. Xia. The Optimal-Location Query. In *SSTD*, 2005.

[5] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.

[6] G. R. Hjaltason and H. Samet. Incremental Distance Join Algorithms for Spatial Databases. In *SIGMOD*, 1998.

[7] G. R. Hjaltason and H. Samet. Distance Browsing in Spatial Databases. *TODS*, 24(2):265–318, 1999.

[8] N. Koudas and K. C. Sevcik. High Dimensional Similarity Joins: Algorithms and Performance Evaluation. In *ICDE*, 1998.

[9] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive Skyline Computation in Database Systems. *TODS*, 30(1):41–82, 2005.

[10] D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui. Aggregate Nearest Neighbor Queries in Spatial Databases. *TODS*, 30(2):529–576, 2005.

[11] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *SIGMOD*, 1995.

[12] S. Shekhar and S. Chawla. *Spatial Databases: A Tour*. Prentice Hall, 2003.

[13] H. Shin, B. Moon, and S. Lee. Adaptive Multi-Stage Distance Join Processing. In *SIGMOD*, 2000.

[14] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 5th edition, 2005.

[15] A. Singh, H. Ferhatosmanoglu, and A. S. Tosun. High Dimensional Reverse Nearest Neighbor Queries. In *CIKM*, 2003.

[16] Y. Tao, D. Papadias, X. Lian, and X. Xiao. Multi-dimensional Reverse kNN Search. *VLDBJ*, To Appear.

[17] C. Xia, H. Lu, B. C. Ooi, and J. Hu. Gorder: An Efficient Method for KNN Join Processing. In *VLDB*, 2004.

[18] T. Xia, D. Zhang, E. Kanoulas, and Y. Du. On Computing Top-t Most Influential Spatial Sites. In *VLDB*, 2005.

[19] M. L. Yiu, N. Mamoulis, and P. Karras. Common Influence Join: A Natural Join Operation for Spatial Pointsets. In *ICDE*, 2008, To Appear.