Common Influence Join: A Natural Join Operation for Spatial Pointsets

Man Lung Yiu #, Nikos Mamoulis *, and Panagiotis Karras †

#*Department of Computer Science, Aalborg University*
DK-9220 Aalborg, Denmark
mly@cs.aau.dk

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

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

***Abstract*— We identify and formalize a novel join operator for two spatial pointsets P and Q. The *common influence join* (CIJ) returns the pairs of points $(p, q), p \in P, q \in Q$, such that there exists a location in space, being closer to p than to any other point in P and at the same time closer to q than to any other point in Q. In contrast to existing join operators between pointsets (i.e., ϵ-distance joins and k-closest pairs), CIJ is parameter-free, providing a natural join result that finds application in marketing and decision support. We propose algorithms for the efficient evaluation of CIJ, for pointsets indexed by hierarchical multi-dimensional indexes. We validate the effectiveness and the efficiency of these methods via experimentation with synthetic and real spatial datasets. The experimental results show that a non-blocking algorithm, which computes intersecting pairs of Voronoi cells on-demand, is very efficient in practice, incurring only slightly higher I/O cost than the theoretical lower bound cost for the problem.**

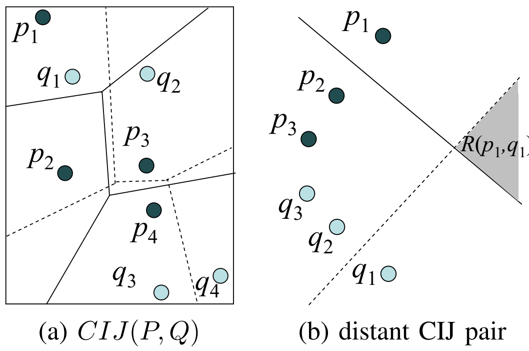


Fig. 1. Examples of CIJ results

I. Introduction

Given a dataset P of points, we can define the *influence region* of each point p in P as the set of locations that are closer to p than to any other point in P. Geometrically, this region corresponds to the Voronoi cell $V(p, P)$ of p in the Voronoi diagram [1] $Vor(P)$ of P. In this paper, we define and study the *common influence join* (CIJ), an interesting spatial data analysis operation related to Voronoi diagrams. Given two pointsets P and Q, CIJ computes all pairs (p, q), $p \in P$, $q \in Q$, such that there exists a common location r inside both $V(p, P)$ and $V(q, Q)$. Figure 1a illustrates two datasets P and Q on the same map. The solid lines define $Vor(P)$, whereas the dotted ones form $Vor(Q)$. $CIJ(P, Q)$ consists of pairs of points whose Voronoi cells intersect: $\{(p_1, q_1)$, (p_1, q_2), (p_2, q_1), (p_2, q_3), (p_3, q_1), (p_3, q_2), (p_3, q_3), (p_3, q_4), (p_4, q_3), $(p_4, q_4)\}$.

Traditional join operations on spatial pointsets are the distance join [2] and the closest pairs join [3], [4]. Given two pointsets P and Q, the ϵ-distance join returns all pairs $(p, q) \in P \times Q$ with their distance $dist(p, q)$ at most ϵ. The k-closest pairs join finds the k pairs in $P \times Q$ with the smallest distance. In contrast to the above operations, CIJ results do not necessarily have distance bounds or distance ordering constraints. For example, in Figure 1a, p_2 is closer to q_2 than to q_3, however, (p_2, q_3) is in $CIJ(P, Q)$, whereas (p_2, q_2) is not. Also, in the example of Figure 1b, p_1 joins with q_1, although p_1 is the furthest neighbor of q_1 in P (and vice versa); whether (p, q) is a CIJ pair does not depend solely on the distance between the two points, but also on the distances and relative locations of p (q) to other points in P (Q). Thus, CIJ cannot be reduced to distance-related spatial joins. Also, CIJ evaluation is more challenging due to the necessity of expensive Voronoi cell computations. Another important difference is that the CIJ result is parameter-free, unveiling the inherent relationship between P and Q. On the other hand, the results of distance joins and closest pairs joins are affected by parameters ϵ and k, respectively. Setting appropriate values to these parameters imposes a burden on the data analyst, who may seek for point pairs that have a *natural* join relationship. Applications of CIJ are illustrated below.

Collaborative Promotion Consider a set of restaurants P and a set of cinemas Q. An advertisement company can compute $CIJ(P, Q)$ and direct collaborative advertisements to the residents in the common influence region of each CIJ pair (p, q); e.g., p offers 5% dinner discount for customers who watched movies in q and q offers free pop-corn to customers who dined in p, within the same week. For example, for the pair (p_1, q_1) of Figure 1b, such a promotion can be directed to region $R(p_1, q_1)$. In addition, for each $(p, q) \in CIJ(P, Q)$, by analyzing $R(p, q)$ (i.e., the intersection of $V(p, P)$ and $V(q, Q)$), we can target a specific marketing focus. For example, if area $R(p_1, q_1)$ in Figure 1b corresponds to a neighborhood where residents have high average age, the joint promotion of p_1 and q_1 for $R(p_1, q_1)$ could be on gourmet food and classic movies.

Decision Support Following the example above, assume that an investor wants to choose a particular cinema to run. By examining the CIJ results, she could assess the potential of each cinema $q \in Q$ with respect to the quality or peculiarities of restaurants in P having common influence with q. If for example the restaurants which join with a cinema q have low revenue or bad service, she may decide not to select q, after inferring that customers may avoid the neighborhood around q. Or, the investor may decide to choose q, based on the fact that restaurants pairing with it lack a popular service, which she might add into the cinema's facilities.

Grouped Nearest Neighbors Typically, the set L of houses on a map is much larger than the set P (Q) of hospitals (parks). A data analyst may be interested in searching, for each house, the nearest hospital and the nearest park. A related `GROUP-BY` analysis operation is to find, for each hospital-park pair, the number of houses having them as their nearest neighbors. These problems can be solved by two All Nearest Neighbor (AllNN) joins [5] of L with P and Q, respectively; however, this is very expensive. An alternative solution is to compute the CIJ between P and Q, and then find the points of L falling in each CIJ region. This is more efficient because (i) not all hospital-park combinations participate in the result (i.e., pairs not in the CIJ result definitely do not appear in the `GROUP-BY` result) and (ii) post-processing (grouping) the numerous AllNN results is avoided. In fact, previous work [6] has shown that intersection of Voronoi diagrams can compute fast location-allocation decision support queries.

Customized Multi-objective Search Personalized filtering can be applied on CIJ pairs to obtain customized results. Examples include (i) a tourist office finds CIJ regions $R(p, q)$ such that both restaurant p and cinema q are above three star, to recommend hotels there and (ii) a tenant searches for housing only in CIJ regions $R(p, q)$, such that hospital p has a coronary intensive care unit and park q has a pool.

Bandwidth Allocation The CIJ between the communication cells of different wireless service providers can be post-processed to design appropriate sharing of bandwidth inside the CIJ regions.

Note that CIJ computation cannot be reduced to simple NN or RNN queries [7] using *only* P and Q. For instance, it is not clear how one can derive the (distant) CIJ pair (p_1, q_1) of Figure 1b, by applying NN or RNN queries on $P \cup Q$ only. Therefore, there is a need for direct CIJ computation algorithms. An intuitive approach is to compute the intersection join of two Voronoi diagrams that have been pre-computed and indexed. However, such a method has high maintenance cost, especially in applications where data updates are frequent compared to CIJ computations. In this paper, we study the problem for the more typical case, where the join inputs P and Q are pointsets indexed by hierarchical spatial access methods,[1] like the R-tree [8]. We propose and evaluate three CIJ algorithms. The first method is an intuitive one that computes $Vor(P)$ and $Vor(Q)$; the Voronoi diagrams of P and Q. It then creates R-tree indexes for them, and finally joins them using an off-the-shelf spatial join algorithm [9]. The construction of each Voronoi diagram and the creation of the corresponding index are performed at low disk access cost, by exploiting the existing R-trees on pointsets P and Q. The second algorithm computes and indexes the Voronoi diagram of P only, and, while computing the Voronoi cells for the points of Q, it probes them at the index of $Vor(P)$ to retrieve their CIJ pairs, in a block index nested loops fashion. Our third algorithm avoids the materialization of complete Voronoi diagrams, but while computing the Voronoi cells for the points of Q, it performs a specialized probing at the R-tree that indexes P, generates only a small subset of $Vor(P)$ on-demand, and computes CIJ pairs using it. As we demonstrate by experimentation, the third algorithm outperforms the other solutions by a wide margin and its I/O cost is close to the lowest possible. The contributions of this paper are summarized as follows:

- We identify the common influence join (CIJ) as a *natural* join operation for pointsets and demonstrate its applicability.
- We propose and evaluate efficient on-demand algorithms to process CIJ for pointsets indexed by R-trees.
- As a side contribution, we develop an optimized R-tree based algorithm for Voronoi cell computation, which subsumes earlier techniques [7], [10] for this problem.

The rest of the paper is organized as follows. Section II provides background and reviews related work to CIJ. Section III presents our optimized algorithm for computing the Voronoi cell for a single point (or a group of points) by applying a single tree traversal. In addition, it describes two intuitive CIJ algorithms that rely on computation and materialization of one or both Voronoi diagrams. Section IV proposes the third and most efficient CIJ algorithm, which evaluates the join directly on the existing trees. A thorough experimental evaluation of our methods is conducted in Section V. Finally, Section VI concludes the paper.

II. Background and Related Work

The Common Influence Join is equivalent to a spatial intersection join between two Voronoi diagrams. Therefore, our work is related to spatial join evaluation and computation techniques for Voronoi diagrams and Voronoi cells. In this section, we review work related to these problems and provide the essential background for our CIJ evaluation techniques that follow.

[1]Spatial access methods can be updated much more efficiently compared to Voronoi diagrams and at the same time they are useful for additional spatial operations like range queries and conventional spatial joins.

A. Evaluation of Spatial Joins

Given two datasets R and S and a spatial predicate θ, the spatial join $R \bowtie_\theta S$ is defined as the subset of the Cartesian product $R \times S$, such that for each pair $(r,s) \in R \bowtie_\theta S$, $r\ \theta\ s$ is true. The typical predicate for spatial joins between two sets of objects with extent is *intersection*. The most efficient method that computes the intersection join between two datasets (R and S) indexed by R-trees (R_R and R_S, respectively) is the *Synchronous Traversal* (ST) algorithm [9]. The idea is to traverse both trees concurrently following entry pairs whose minimum bounding rectangles (MBRs) intersect; if a non-leaf entry e_R from R_R is disjoint with a non-leaf entry e_S from R_S, then there can be no pair of objects (r,s) that intersect, such that r (s) is in the subtree pointed to by e_R (e_S).

For joins between pointsets P and Q, θ is most commonly a *distance* constraint ϵ; the ϵ-distance join returns the pairs of points $(p,q), p \in P, q \in Q$, such that $dist(p,q) \le \epsilon$, where $dist()$ is a distance metric (i.e., usually Euclidean distance). Assuming that P and Q are indexed by two R-trees, we can adapt the synchronous traversal algorithm of [9] (described above) to follow entry pairs (e_P, e_Q) with $mindist(e_P, e_Q) \le \epsilon$. Here, $mindist(e_P, e_Q)$ denotes the minimum possible distance between any pair of points $p \in$ MBR(e_P) and $q \in$ MBR(e_Q). Another popular join between pointsets is the closest pairs join [3], [4], which takes as input a number k and returns the k pairs (p,q) with the smallest distance. This problem can be solved by combining ideas from nearest neighbor search algorithms [11] with the synchronous traversal join algorithm [9].

Previous work on spatial join computation cannot directly be applied for CIJ evaluation. The main challenge to address is that whether a point $p \in P$ participates in a CIJ result depends not only on the locations of points in Q, but also on the locations of other points in P.

B. Computation of Voronoi Diagrams and Cells

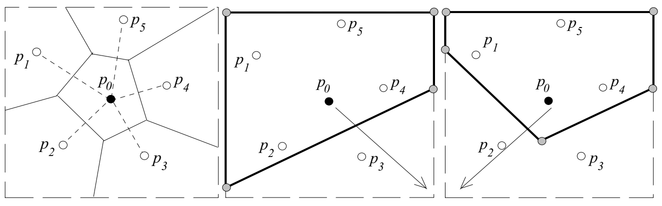
Given two points p_i and p_j on the plane, the halfplane $\perp_{p_i}(p_i, p_j)$ is defined by the locations closer to p_i than p_j:

$$\perp_{p_i}(p_i, p_j) = \{\ a \mid dist(p_i, a) \le dist(p_j, a)\ \} \qquad (1)$$

The *Voronoi cell* of p_i in pointset P is defined by the intersection of all halfplanes with other points in P:

$$V(p_i, P) = \bigcap_{p_j \in P, p_j \neq p_i} \perp_{p_i}(p_i, p_j) \qquad (2)$$

The *Voronoi diagram* [1] $Vor(P)$ of P is the space partitioning formed by the cells $V(p_i, P)$ of all $p_i \in P$. Figure 2a shows $Vor(P)$ for a set P of six points. Observe that the border between two adjacent cells corresponds to their perpendicular bisector and each Voronoi cell is a convex polygon. The Voronoi diagram can be computed in main memory in $O(n \log n)$ time [1]. For large pointsets that do not fit in memory, the Voronoi diagram can be computed by a complex 3D convex hull algorithm with same asymptotic cost as external sorting [12]. Isenburg et al. [13] showed how to compute the Delaunay Triangulation of a pointset P (convertible to $Vor(P)$) at only three passes over P. Although this method exploits spatial properties to reduce memory consumption, its peak memory size depends on the data distribution and is not known apriori.



(a) Voronoi diagram (b) TP-VOR, step 1 (c) TP-VOR, step 2

Fig. 2. Voronoi cell computation

Our CIJ evaluation techniques use single Voronoi cell computation as a building block, so we discuss work related to this problem in detail. According to Equation 2, a simple method is to scan all data points $p \in P$ and take the intersection of halfplanes $\perp_{p_i}(p_i, p_j)$ for all $p_j \neq p_i$. This approach is impractical since only the points surrounding p_i contribute to $V(p_i, P)$ (on the average the number of such points is only 6). [14] proposed a main-memory method for computing an approximation of $V(p_i, P)$ with asymptotic bounds on approximation quality and space complexity. Assuming that the pointset P is indexed by an R-tree, [7] developed a method for computing $V_c(p_i)$, an approximation (superset) of $V(p_i, P)$, by finding the nearest neighbors of p_i at each of the four quadrants defined by rectilinear lines passing p_i. This can be done by issuing four concurrent *constrained* nearest neighbor queries. $V_c(p_i)$ is then defined by taking the bisectors of p_i and its NN at each quadrant.

[10] proposed a technique for *exact* computation of $V(p_i, P)$. First, an approximation $V_c(p_i)$ of $V(p_i, P)$ is initialized to the whole space domain. As Figure 2b illustrates, a *time-parameterized* nearest neighbor (TPNN) query [15] is issued towards a vertex (e.g., bottom-right corner) of $V_c(p_0)$, to find the NN of p_0 along that direction. In our example, point p_3 is discovered and used to refine $V_c(p_0)$ (by intersecting the current $V_c(p_0)$ with $\perp_{p_0}(p_0, p_3)$). The above procedure is repeated for other vertices of $V_c(p_0)$. In Figure 2c, a TPNN query is issued towards the bottom-left corner of $V_c(p_0)$, and the retrieved point p_2 is used to refine $V_c(p_0)$. After TPNN queries have been issued towards all vertices in $V_c(p_0)$, the algorithm returns $V_c(p_0)$ as the exact $V(p_0, P)$. Note that, during the refinement of $V_c(p_0)$, the next TPNN queries to be issued depend on the results of previous TPNN queries (since the vertices of $V_c(p_0)$ change after iteration). Thus, these operations cannot be combined to a single traversal of the R-tree. As a result, the algorithm requires multiple R-tree traversals, one for each TPNN query.

III. Multi-stage CIJ Computation

In this section, we first propose an R-tree based Voronoi cell computation algorithm, which has better performance than the methods in [7], [10]. We then extend the technique for batch Voronoi cell computation of a group of points close to each other. Finally, we propose two algorithms for CIJ based on the computation of Voronoi diagram(s) using our Voronoi cell computation technique.

A. Efficient Voronoi Cell Computation

As discussed in Section II-B, existing R-tree based algorithms for Voronoi cell computation are either approximate [7] or require multiple tree traversals [10]. Our goal is to compute the *exact* $V(p_i, P)$ for a point $p_i \in P$ at a *single traversal* of the R-tree R_P, i.e., accessing each tree node of R_P at most once. We propose an algorithm that achieves this goal at *low I/O cost*.

The idea is to start with an approximation $V_c(p_i)$ of $V(p_i, P)$ (initially $V_c(p_i)$ is set to the whole space domain) and progressively refine it to the exact cell. While traversing R_P, we should determine, for each visited entry e, whether the subtree pointed to by e may contain points that potentially refine $V_c(p_i)$. Let $\Gamma_c(p_i)$ be the set of vertices of the current $V_c(p_i)$. The following lemma utilizes a geometric observation to determine whether a point $p_j \in P$ can refine the current $V_c(p_i)$:

Lemma 1: Given a point $p_j \in P$ ($p_i \neq p_j$), if $\forall \gamma \in \Gamma_c(p_i)$, $dist(p_j, \gamma) \geq dist(\gamma, p_i)$, then the current $V_c(p_i)$ cannot be refined by p_j.

Proof: Observe that the polygon of $V_c(p_i)$ is the convex hull of all vertices $\gamma \in \Gamma_c(p_i)$. According to Theorem 1 of Ref. [16], for any point p_j inside $V_c(p_i)$, there exists a vertex $\gamma \in \Gamma_c(p_i)$ such that $dist(p_j, \gamma) \leq dist(\gamma, p_i)$. Thus, any point p_j satisfying the condition of this lemma, must fall outside $V_c(p_i)$.

Since $\forall \gamma \in \Gamma_c(p_i)$, $dist(p_j, \gamma) \geq dist(\gamma, p_i)$, all vertices of $V_c(p_i)$ must be included in the refined $V'_c(p_i)$ by p_j. Thus the refined cell $V'_c(p_i)$ is a convex polygon, containing all vertices in $V_c(p_i)$. Combining this with the property $V'_c(p_i) \subseteq V_c(p_i)$, we conclude that $V'_c(p_i) = V_c(p_i)$. ■

The above lemma can be generalized to determine whether a non-leaf entry e of R_P (and any point in the subtree pointed to by e) can refine $V_c(p_i)$. Let $mindist(e, p)$ be the minimum possible distance between a point p and any point in the MBR of an R-tree entry e.

Lemma 2: Given a non-leaf entry e of R_P, if $\forall \gamma \in \Gamma_c(p_i)$, $mindist(e, \gamma) \geq dist(\gamma, p_i)$, then $V_c(p_i)$ cannot be refined by any point in e.

Proof: True, due to Lemma 1 and lower-bounding property of $mindist$: $\forall\ p_j \in e,\ dist(p_j, \gamma) \geq mindist(e, \gamma)$. ■

In addition, we should set an appropriate order for visiting nodes that would minimize the I/O cost. We decide to prioritize the visit of entries e according to $mindist(e, p_i)$ (i.e., in the same order as the best-first incremental NN algorithm of [11]). This way, it becomes more likely to discover early points near p_i that refine $V_c(p_i)$ to a tight approximation of $V(p_i, P)$. Algorithm 1 puts everything together to a method for computing the exact $V(p_i, P)$ efficiently, by accessing each tree node at most once and minimizing the number of accessed nodes. Initially, $V_c(p_i)$ is set to the space domain $\mathbb{U}$. The algorithm then browses the tree entries in ascending order of their distances from p_i. Whenever a point is discovered, it is used to refine $V_c(p_i)$. In addition, our pruning technique is incorporated at Line 7, for entries (and their subtrees) that cannot refine $V_c(p_i)$ further. The method continues to examine the next entry until H becomes empty. Eventually, $V_c(p_i)$ is returned as the exact Voronoi cell $V(p_i, P)$.

Algorithm 1 Single Voronoi Cell Computation

```
    algorithm SingleVoronoi(Point $p_i$, R-Tree $R_P$)
 1: $H$:=new min-heap ($mindist$ from $p_i$ as the key);
 2: $V_c(p_i)$:=the space domain $\mathbb{U}$;                ▷ current Voronoi cell
 3: for all entries $e \in R_P.root$ do
 4:     insert $\langle e, mindist(e, p_i)\rangle$ into $H$;
 5: while $H$ is not empty do
 6:     deheap $\langle e, mindist(e, p_i)\rangle$ from $H$;
 7:     if $\exists\ \gamma \in \Gamma_c(p_i)$, $mindist(e, \gamma) < dist(\gamma, p_i)$ then
 8:         if $e$ is a point $p_j$ then
 9:             update $V_c(p_i)$ by $\perp_{p_i}(p_i, p_j)$;        ▷ update cell
10:         else
11:             read the child node $N'$ pointed to by $e$;
12:             for all entries $e' \in N'$ do
13:                 insert $\langle e', mindist(e', p_i)\rangle$ into $H$;
14: return $V_c(p_i)$;
```

B. Batch Voronoi Cell computation

Suppose that we need to compute the Voronoi cells for a subset G of P with closely located points. A simple solution is to invoke Algorithm 1 for each point in G. However, this may result in accessing some nodes of the tree multiple times, since Voronoi cells of nearby points are defined by points in the same region. In order to avoid this redundancy, we propose Algorithm 2, a batch method for computing the Voronoi cells of all points in G concurrently.

Algorithm 2 Batch Voronoi Cell Computation

```
    algorithm BatchVoronoi(Set $G$, R-Tree $R_P$)
 1: $\overline{G}$:=centroid of all points in $G$;
 2: $H$:=new min-heap ($mindist$ from $\overline{G}$ as the key);
 3: for all points $p_i \in G$ do
 4:     $V_c(p_i)$:=the space domain $\mathbb{U}$;          ▷ current Voronoi cell
 5: for all entries $e \in R_P.root$ do
 6:     insert $\langle e, mindist(e, \overline{G})\rangle$ into $H$;
 7: while $H$ is not empty do
 8:     deheap $\langle e, mindist(e, \overline{G})\rangle$ from $H$;
 9:     if $\exists p_i \in G, \exists \gamma \in \Gamma_c(p_i)$, $mindist(e, \gamma) < dist(\gamma, p_i)$ then
10:         if $e$ is a point $p_j$ then
11:             for all points $p_i \in G$ do
12:                 if $\exists \gamma \in \Gamma_c(p_i)$, $dist(p_j, \gamma) \leq dist(\gamma, p_i)$ then
13:                     update $V_c(p_i)$ by $\perp_{p_i}(p_i, p_j)$;
14:         else
15:             read the child node $N'$ pointed to by $e$;
16:             for all entries $e' \in N'$ do
17:                 insert $\langle e', mindist(e', \overline{G})\rangle$ into $H$;
18: return $V_c(p_i)$, $\forall p_i \in G$;
```

BatchVoronoi has the following modifications compared to Algorithm 1. First, entries e are deheaped from H in ascending distance order from $\overline{G}$, the centroid of all points in G. Second, an entry e is pruned when it cannot lead to the refinement of the Voronoi cell of any point $p_i \in G$. Third, before updating the Voronoi cell of a point $p_i \in G$ (at Line 13), we check whether e may refine such a cell (not all points could use e for their refinement). Next, we show how this algorithm is used by two intuitive CIJ evaluation techniques.

C. Computing CIJ pairs

Our first solutions for CIJ evaluation are based on the intuitive idea of first computing the Voronoi diagrams for the datasets P and Q, and then their intersection join.

Full materialization Algorithm 3 is the pseudo-code of a *full materialization* CIJ algorithm (FM-CIJ). First, FM-CIJ performs depth-first traversal on the tree R_P of P. For each leaf node N_P encountered, the exact Voronoi cells of all points in N_P are computed concurrently, using Algorithm 2. Afterwards, these Voronoi cells are inserted into another R-tree R'_P. The same procedure is applied for creating R'_Q; an R-tree that indexes the Voronoi diagram of Q. Finally, the intersection join algorithm of [9] is performed between R'_P and R'_Q to obtain intersecting pairs of Voronoi cells, which correspond to the CIJ result.

Algorithm 3 Full Materialization Algorithm

```
    algorithm FM-CIJ(R-Tree $R_P$, R-Tree $R_Q$)
 1: $R'_P$:=new R-tree; $R'_Q$:=new R-tree;
 2: apply depth-first traversal to $R_P$;
 3: for all visited leaf nodes $N_P$ do
 4:     $G_P$:=$\{p \in N_P\}$;
 5:     $V_P$:=BatchVoronoi($G_P, R_P$);
 6:     insert contents of $V_P$ into $R'_P$;
 7: apply depth-first traversal to $R_Q$;
 8: for all visited leaf nodes $N_Q$ do
 9:     $G_Q$:=$\{q \in N_Q\}$;
10:     $V_Q$:=BatchVoronoi($G_Q, R_Q$);
11:     insert contents of $V_Q$ into $R'_Q$;
12: perform intersection join between $R'_P$ and $R'_Q$;
```

Partial materialization Algorithm 4 is the pseudo-code of a *partial materialization* CIJ algorithm (PM-CIJ). Unlike FM-CIJ, this method saves I/O accesses by building only one R-tree (instead of two). First, PM-CIJ performs depth-first traversal on R_P, computes the Voronoi cells of all points in P and indexes them by a tree R'_P (i.e., exactly like FM-CIJ). Then, it traverses R_Q, and for each leaf node $N_Q \in R_Q$, the Voronoi cells of the set of points G_Q in it are computed in batch. However, instead of inserting them to another tree R'_Q, PM-CIJ immediately probes them to R'_P to find their CIJ join pairs in P. The latter operation is performed as a single range query to R'_P, with the query region enclosing all Voronoi cells of G_Q. Thus, PM-CIJ operates like a block index nested loops algorithm, performing probes to R'_P, for batches of Voronoi cells from Q. Intuitively, if consecutive probes have high spatial locality and an LRU buffer is used, PM-CIJ will be cheaper than FM-CIJ.

Optimized construction of R'_P and R'_Q We now discuss the construction of Voronoi R-trees R'_P (and R'_Q), which are used by FM-CIJ and PM-CIJ. We avoid individual insertion of the computed cells into the trees, in order to reduce the construction cost. Note that each cell has at least three vertices and not all cells have the same number of vertices. In order to create leaf nodes (for R'_P) of fixed page size, we tune the depth-first traversal of R_P, so that the entries are accessed in the order of Hilbert values [17] of their centroids. This way, successively created Voronoi cells are close in space. These Voronoi cells are *sequentially packed* into leaf nodes of the tree R'_P so as to bulk-load the tree in a bottom-up fashion. This technique has several advantages: (i) expensive node splits for R'_P are avoided and the I/O cost of tree construction is exactly the cost of writing the nodes of R'_P to disk, (ii) disk space is fully utilized, and (iii) packed leaf nodes contain Voronoi cells close in space and the tree has good search performance (similar to Hilbert R-tree [18]).

Algorithm 4 Partial Materialization Algorithm

```
    algorithm PM-CIJ(R-Tree $R_P$, R-Tree $R_Q$)
 1: $R'_P$:=new R-tree;
 2: apply depth-first traversal to $R_P$;
 3: for all visited leaf nodes $N_P$ do
 4:     $G_P$:=$\{p \in N_P\}$;
 5:     $V_P$:=BatchVoronoi($G_P, R_P$);
 6:     insert contents of $V_P$ into $R'_P$;
 7: apply depth-first traversal to $R_Q$;
 8: for all visited leaf nodes $N_Q$ do
 9:     $G_Q$:=$\{q \in N_Q\}$;
10:     $V_Q$:=BatchVoronoi($G_Q, R_Q$);
11:     apply batch range search on $R'_P$, using $V_Q$;
```

IV. Non-blocking CIJ Computation

The techniques proposed in the previous section are *blocking* in the sense that they require the creation of at least one complete Voronoi diagram (i.e., $Vor(P)$), which is then indexed by an R-tree (i.e., R'_P). In this section, we propose a more efficient and *non-blocking* technique for CIJ evaluation. Its main innovation is that, unlike PM-CIJ, it does not create $Vor(P)$; instead, for each computed Voronoi cell $V(q, Q)$ of a $q \in Q$, it determines the CIJ pairs that include q by probing the cell $V(q, Q)$ to the original R-tree R_P that indexes P. In Section IV-A, we elaborate on this probing methodology. Then, in Section IV-B, we describe our third CIJ algorithm, which is founded on this operation.

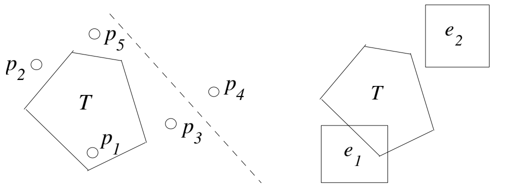
A. Voronoi Cell Intersection Search

Assume that we know the Voronoi cell $T = V(q, Q)$ of a point q in Q and that we want to find the points in $p_i \in P$ for which $V(p_i, P)$ intersects T (i.e., (p_i, q) is a CIJ result). Figure 3a exemplifies T and a pointset P. We can distinguish three cases depending on the position of p_i with respect to T and other points in P.

1) p_i *is inside* T. In this case, $V(p_i, P)$ must intersect T. For such point (e.g., p_1 in Figure 3a) we report (p_1, q) as a CIJ result without extra computation.
2) p_i *is outside* T *and exists* $p_j \in P$, *such that* $\perp_{p_i}(p_i, p_j)$ *does not contain* T. Note that $\perp_{p_i}(p_i, p_j)$

is a superset of $V(p_i, P)$ (see Eq. 2). If T lies outside $\perp_{p_i}(p_i, p_j)$ then p_i can be pruned, since $V(p_i, P)$ cannot intersect T. In Figure 3a, T lies outside the halfplane region $\perp_{p_4}(p_4, p_3)$, therefore q does not join with p_4.

3) *p_i is outside T and there does not exist $p_j \in P$, such that $\perp_{p_i}(p_i, p_j)$ does not intersect T.* In this case, (p_i, q) may be a CIJ result (depending on the locations of other points in P). For example, points p_2, p_3, and p_5 of Figure 3a may form join pairs with q.



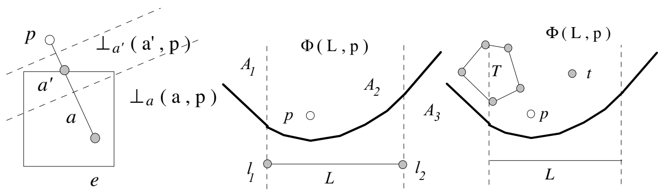
(a) leaf entries (b) non-leaf entries

Fig. 3. Conditional voronoi cell intersection

The last case is the most expensive to verify exactly; it is actually more expensive than computing $V(p_i, P)$ (e.g., using our method from Section III-A). Given $T = V(q, Q)$, our goal is to traverse the R-tree R_P that indexes P and identify the CIJ pairs that include q with as few node accesses to R_P as possible. We perform this search in two phases. During the *filter* phase, we traverse R_P and eliminate points and subtrees that cannot contain points that join with q (falling in case 2 above), while constructing a set C_P of *candidate* points $p \in P$ may join with q. In the second, *refinement* phase, we compute the exact Voronoi cells for the points in C_P and test them for intersection against T (except from the points that fall in Case 1 above). During the filter phase, C_P is used to prune points from R_P as follows. Let p be a newly examined point (not in C_P). We compute $V(p, C_P)$; the approximate (superset) Voronoi cell of p using only the points in C_P (recall that this is a spatial superset of the actual $V(p, P)$, since $C_P \subseteq P$). If $V(p, C_P)$ does not intersect T, then the point p is eliminated. Otherwise, we include p in C_P, since it may be necessary to compute the exact $V(p, P)$ in order to check whether it intersects T.

Pruning subtrees of R_P We now elaborate on the pruning of entries from R_P during the filter phase. Given the MBR e of a non-leaf entry e that indexes a subset of points from P, our goal is to determine whether e may contain some point whose Voronoi cell intersects region $T = V(q, Q)$. Consider the example in Figure 3b where e_1 and e_2 are non-leaf entries of R_P. If a non-leaf entry (e.g., e_1) intersects T, then it may contain some points inside T (these definitely join with q). Thus, the entry cannot be pruned; we need to access its child node.

In case a non-leaf entry e (e.g., e_2 in Figure 3b) does not intersect T, we know that all points in it must be outside T. In this case, we check only the MBR of e, to determine whether the subtree pointed to by e may contain a point p with $V(p, P)$ intersecting T. This can be done with the help of other points p seen from R_P so far (i.e., the candidate set C_P). Figure 4a shows an entry $e \in R_P$ and another point p from P. For any point $a \in e$, region $\perp_{a'}(a', p)$ always encloses $\perp_a(a, p)$, where a' is the intersection of segment ap with the boundary of e. Therefore, the Voronoi cells of potential points on the boundary of e represent the largest possible extent of Voronoi cells of points in e, and we confine our study to these points only.



(a) reduction (b) $\Phi(L, p)$ (c) geometric filter

Fig. 4. Pruning a non-leaf entry e

Given a single side L of e, we define region $\Phi(L, p)$, as the set of all locations closer to p than to any location on L:

$$\Phi(L, p) = \{\ b \mid dist(p, b) \leq mindist(L, b)\ \} \tag{3}$$

Figure 4b shows a point p, a line segment $L = l_1 l_2$, and the region $\Phi(L, p)$. The dotted lines are perpendicular to L and divide the space domain into three partitions A_1, A_2, and A_3. Note that any point in A_1 (A_3) has l_1 (l_2) as its closest location on L. Thus, in A_1 (A_3), $\Phi(L, p)$ is bounded by the perpendicular bisector between p and l_1 (l_2). Inside partition A_2, $\Phi(L, p)$ is bounded by the parabolic bisector of p and line L. Overall, the boundary of $\Phi(L, p)$ can be represented by a piecewise function: a quadratic function in A_2 and two linear functions in A_1 and A_3. Thus, we can check whether any point t falls in $\Phi(L, p)$ in constant time, by evaluating the corresponding function depending on the region where t falls (e.g., A_2 for the example of Figure 4c). To find out whether a convex polygon T completely falls in $\Phi(L, p)$, we can perform the check for each of its vertices as the following lemma suggests:

Lemma 3: Given a convex polygon T, a point p, and a line segment L, if each vertex of T falls in $\Phi(L, p)$, then every location in T falls in $\Phi(L, p)$.

Proof: True, since both $\Phi(L, p)$ and T are convex. ■

Lemma 3 can be used to verify whether a non-leaf entry e (from R_P) can contain points whose Voronoi cells intersect T; if it cannot, it is pruned from search. Recall that during the filter phase of the search, we maintain a set of candidate points C_P, which are likely to form join pairs with q. If there exists a point $p \in C_P$ such that T falls in $\Phi(L, p)$ for all sides L of e, then we can safely prune e, since the Voronoi cell of any point in e cannot intersect T.

The algorithm Algorithm 5 is a pseudocode for the procedure that implements the filter phase of our search. ConditionalFilter computes a candidate set C_P of points p from R_P, for which $V(p, P)$ possibly intersects a convex polygon T. It employs a heap like the incremental NN search [11] algorithm, in order to retrieve the points $p \in P$ in ascending order of their distances from $\overline{T}$, the centroid of T. When a leaf entry of R_P is deheaped (i.e., a point $p \in P$), we compute its approximate Voronoi cell $V(p, C_P)$ using only the current contents of C_P. Point p is inserted into C_P only when $V(p, C_P)$ intersects T. When a non-leaf entry e is deheaped, we attempt to prune e using the geometric checking technique described in the previous paragraph. If e cannot be pruned, its child node N' is loaded and all entries of N' are inserted into H. When H becomes empty, C_P contains the set of points that pass the filter step. In the refinement phase, we examine the entries in C_P and test whether their exact Voronoi cells intersect T. Points from C_P falling in T are immediately reported as true hits without further processing. The exact Voronoi cells of the remaining candidates need to be computed (by Algorithm 1) in order to check whether they intersect T.

Algorithm 5 Conditional Filter

```
    algorithm ConditionalFilter(Polygon $T$, R-Tree $R_P$)
 1: $\overline{T}$:=centroid of $T$;
 2: $H$:=new min-heap ($mindist$ from $\overline{T}$ is the key);
 3: $C_P$:=$\varnothing$;                               ▷ set of candidate points
 4: for all entries $e \in R_P.root$ do
 5:     insert $\langle e, mindist(e, \overline{T})\rangle$ into $H$;
 6: while $H$ is not empty do
 7:     deheap $\langle e, mindist(e, \overline{T})\rangle$ from $H$;
 8:     if $e$ is a point $p$ then
 9:         compute the approximate Voronoi cell $V(p, C_P)$;
10:         if $V(p, C_P)$ intersects $T$ then
11:             insert $p$ into $C_P$;
12:     else
13:         if $\exists p \in C_P, \forall L \in e$, $T$ completely falls in $\Phi(L, p)$ then
14:             go to Line 6;                            ▷ $e$ is pruned
15:         read the child node $N'$ pointed to by $e$;
16:         for all entries $e' \in N'$ do
17:             insert $\langle e', mindist(e', \overline{T})\rangle$ into $H$;
18: return $C_P$;
```

Batch conditional filter In Section III-B we extended the Voronoi cell computation algorithm to apply for a set of points. Likewise, for a group G of convex polygons (i.e., Voronoi cells from $Vor(Q)$), we can apply a BatchConditionalFilter(G, R_P) process, which computes a subset C_P of P, containing points whose Voronoi cells may intersect *any* polygon in G. For this, we make three modifications to Algorithm 5. First, the entries are deheaped from H in ascending order of their distances from $\overline{G}$, the centroid of all polygons in G. Second, at Lines 10–11, p is inserted into C_P when it intersects *any* polygon in G. Third, the non-leaf entry e is pruned at Line 13 when *all* polygons $T \in G$ fall in $\Phi(L, p)$ for all L at the boundary of e.

B. Computing CIJ pairs

Algorithm 6 is a pseudocode of our *no materialization CIJ* method (NM-CIJ), which is based on the (batch) Voronoi cell intersection search operation. NM-CIJ is similar to PM-CIJ, in that it traverses R_Q and for each leaf node N_Q it encounters, it computes the Voronoi cells of all points $q \in N_Q$ (in batch) and finds the CIJ pairs that include these points by probing the group to an index of P. However (unlike PM-CIJ), NM-CIJ neither computes nor indexes the complete Voronoi diagram of P, but finds the CIJ pairs by searching R_P directly, using the two-phase approach described in Section IV-A. Therefore, NM-CIJ avoids the preprocessing of P and the creation of the R'_P index at the expense of on-demand (potentially multiple) computations of Voronoi cells of points in P.

Algorithm 6 No Materialization Algorithm

```
   algorithm NM-CIJ(R-Tree $R_P$, R-Tree $R_Q$)
1: apply depth-first traversal to $R_Q$;
2: for all visited leaf nodes $N_Q$ do
3:     $G_Q$:={$q \in N_Q$};
4:     $V_Q$:=BatchVoronoi($G_Q$, $R_Q$);
5:     $C_P$:=BatchConditionalFilter($V_Q$,$R_P$);           ▷ filter phase
6:     BatchVoronoi($C_P$, $R_P$);                         ▷ refinement phase
7:     for all $q \in G_Q$, $p \in C_P$ such that $V(p, P)$ intersects $V(q, Q)$
           do
8:         report $(p, q)$ as a CIJ result;
```

Reuse of Voronoi cells in NM-CIJ A careful examination of Algorithm 6 reveals that even though the Voronoi cell of each point in Q is computed exactly once, Voronoi cells of points in P may be repeatedly computed, if they are used by points in Q at different leaf nodes N_Q of R_Q. We observed that nearby leaf nodes in R_Q share a fraction of common items in their candidate sets C_P. In order to take advantage of this observation, we employ a buffer B which maintains the Voronoi cells of C_P from the previous loop. If the exact cell $V(p, P)$ of a point $p \in C_P$ at the current loop already resides in B, we avoid computing it again. B is updated to contain the cells of the current C_P, in order to be used in the next loop.

V. Experimental Evaluation

In this section, we experimentally evaluate the efficiency of CIJ algorithms on synthetic and real datasets. Uniform synthetic datasets were generated by assigning random locations to points. We obtained real datasets of geographical features from the *U.S. Board on Geographic Names*[2]; their descriptions are shown in Table I. Attribute values of all datasets are normalized to the interval [0, 10000]. Each dataset is indexed by an R-tree with a disk page size of 1K bytes. By default, both P and Q have $n = 100$K points each. We used an LRU memory buffer whose default size is set to 2% of the data size on disk. All algorithms (FM-CIJ, PM-CIJ, and NM-CIJ) were implemented in C++. Experiments were performed on a Pentium D 2.8GHz PC with 1GB memory.

A. Voronoi Cell Computation

In the first experiment, we compare our single-traversal technique BF-VOR (Algorithm 1) against the multiple-traversal method TP-VOR [10], on a synthetic (uniform) dataset P with 100K points indexed by an R-tree. Figure 5a shows the R-tree node accesses incurred for computing

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

TABLE I

REAL DATASETS OF US

Dataset	Contents	Data cardinality
PP	Populated Places	177983
SC	Schools	172188
CE	Cemeteries	124336
LO	Locales	128476
PA	Parks	58312

the Voronoi cells of individual queries (100 query points randomly chosen from the dataset). BF-VOR outperforms TP-VOR and has stable performance across different query instances because BF-VOR accesses a tree node at most once and employs an effective pruning rule to discard early entries unqualified for refining the Voronoi cell of the query point. Figure 5b displays the CPU cost of the algorithms, which is directly proportional to the node accesses.

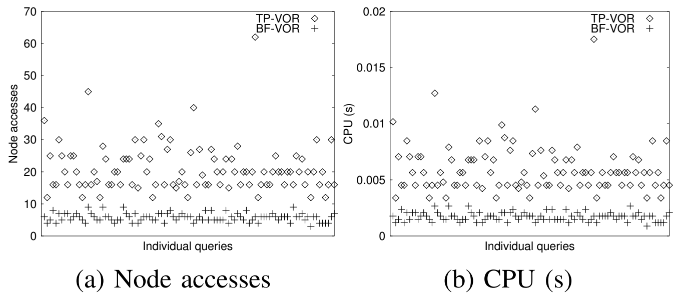


Fig. 5. Cost of individual queries, $n = 100$K

Next, we test the following methods for Voronoi diagram computation on a uniform dataset P (indexed by an R-tree R_P): (i) an ITER method, which traverses R_P in depth-first order and for each point p computes its $V(p, P)$, using Algorithm 1, and (ii) a BATCH method which computes Voronoi cells of points in the same leaf node concurrently, using Algorithm 2. The comparison also includes LB, which represents the lowest possible I/O cost bound of these methods; that of just traversing the complete R_P.

Figure 6 shows the cost of all Voronoi cell computation as a function of the datasize. ITER and BATCH have similar I/O cost as LB. Both of them have much lower I/O cost than external memory Voronoi diagram computation algorithms (the theoretical one [12] and the practical one [13]), which require multiple reads and writes on the data (i.e., with I/O cost as a few times of LB). Regarding CPU cost, the performance gap between ITER and BATCH widens as the datasize increases. Table II shows the performance of BATCH on computing the Voronoi diagrams of the real datasets. Observe that the I/O cost of BATCH may vary between datasets of similar sizes (e.g., PP and SC). The algorithm is slightly more expensive when adjacent Voronoi cells have large size deviation in area. In such cases, the points located close to the boundary between small and large Voronoi cells are frequently accessed. Still, BATCH is I/O-efficient for all tested datasets. In addition, the CPU cost of BATCH scales well with the size of real datasets.

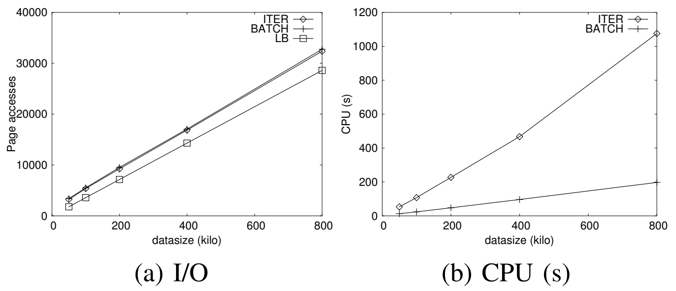


Fig. 6. The effect of datasize n

TABLE II

PERFORMANCE OF BATCHVORONOI

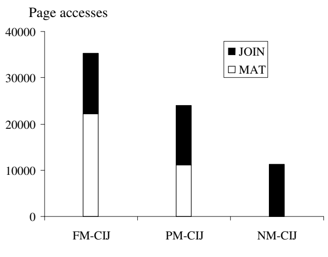
Dataset	Page accesses	CPU (s)
PP	10177	57.3
SC	13220	56.0
CE	10504	37.0
LO	8134	41.0
PA	6406	20.3

B. CIJ Computation

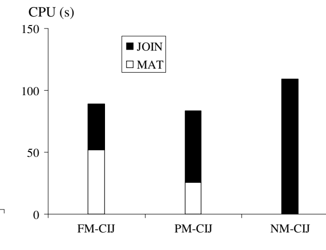
The next set of experiments compare the performance of the three CIJ algorithms (FM-CIJ, PM-CIJ, and NM-CIJ) for a wide range of settings. We first examine the case of joining uniform synthetic datasets. Figure 7 shows the cost breakdown of I/O and CPU cost of the algorithms for the default setting (i.e., $|Q| = |P| = 100$K, buffer size at 2%). The three methods have similar I/O costs for JOIN (join processing cost) but different costs for MAT (materialization cost). The results show that NM-CIJ saves the creation and materialization of the Voronoi R-trees, at no I/O penalty to its join cost. Regarding CPU cost, the three methods have similar performance, with PM-CIJ being slightly cheaper than the other methods. NM-CIJ is expected to have the highest CPU time since it uses more sophisticated techniques for pruning (i.e., the BatchVoronoiFilter method). Nevertheless, its significant savings in terms of I/O compensate this slight computational cost overhead. Note that (if we charge a typical 10ms for each random disk page access), the algorithms are I/O sensitive, with the exception of NM-CIJ whose I/O cost (at the presence of a buffer) is very low and similar to the computational cost of all algorithms.

As the basis for comparison, in the following experiments we include the I/O cost of LB, i.e., the cost of traversing both trees once, which provides a theoretical lower bound I/O cost of CIJ computation using R-trees.[3] Figure 8a shows the I/O performance of the algorithms as a function of the buffer size (%). NM-CIJ outperforms its competitors, for the reasons we already explained. As the buffer size increases, more nodes

[3]*Every* point in P and Q should participate in the CIJ (since each $p \in P$ is contained in a cell of $Vor(Q)$ and vice-versa). Therefore it is essential to visit all points in P and Q during CIJ computation.



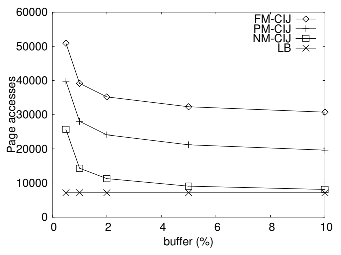
(a) I/O



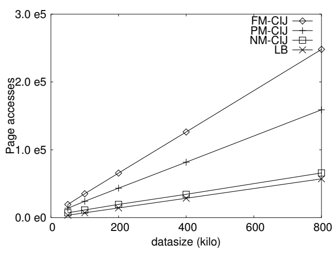
(b) CPU (s)

Fig. 7. Cost breakdown, $|Q| = |P| = 100\text{K}$

can be effectively cached, thus the I/O cost of all methods decreases. The cost of NM-CIJ converges rapidly to that of LB; at 2% buffer size the difference is only 30%. NM-CIJ incurs low I/O cost because it makes excellent use of the buffer, by (i) visiting the leaf nodes of Q in an order with high locality and (ii) computing Voronoi cells and their join pairs by examining only nearby nodes of R_Q and R_P, which exist in the buffer with high probability. Figure 8b plots the cost of the algorithms with respect to the datasize n. All algorithms scale well for large datasets. NM-CIJ has the lowest I/O cost, which is very close to LB. The CPU time of NM-CIJ is only slightly higher than that of PM-CIJ and FM-CIJ. Their relative CPU-time difference (graph omitted) remains constant. In the remaining experiments we do not plot the CPU cost, since all three algorithms have similar computational performance with the CPU cost of NM-CIJ being only slightly (10%-20%) higher.



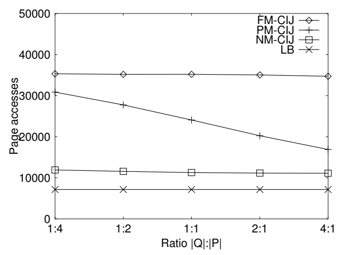
(a) buffer effect



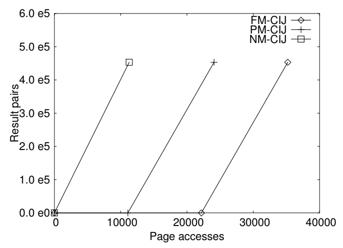
(b) scalability to $|P| = |Q|$

Fig. 8. Effect of memory and database size

Figure 9a shows the I/O cost of the algorithms as a function of the cardinality ratio $|Q| : |P|$ for a constant number of total points $|Q|+|P|$ =200K. As the cardinality ratio increases, $|P|$ decreases and PM-CIJ incurs less I/O cost on materializing the Voronoi cells of points in P. For FM-CIJ, the total number of materialized Voronoi cells remains constant for different cardinality ratios. As we will verify in another experiment, the ratio does not have much impact on the number of redundant Voronoi cell computations in NM-CIJ, which remains constant as well. Figure 9b plots the number of CIJ results produced by the algorithms, as a function of their current I/O cost (progressiveness). Both FM-CIJ and PM-CIJ are blocking; they generate CIJ pairs after one or two Voronoi R-trees are built. On the other hand, NM-CIJ is a non-blocking algorithm, starting to produce CIJ pairs immediately.



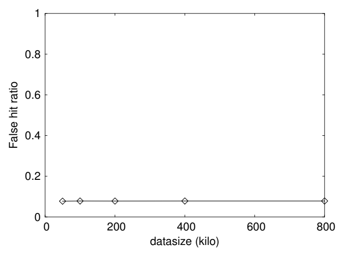
(a) ratio $|Q|$:$|P|$,$|Q|$+$|P|$=200K



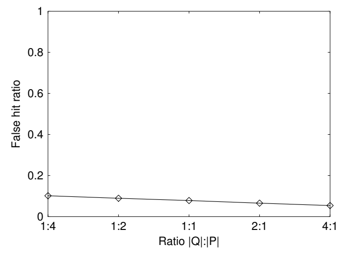
(b) output progress

Fig. 9. Cardinality ratio and progress

The effectiveness of the filter step (Line 5 of Algorithm 6) directly affects the performance of NM-CIJ. At the i-th loop of NM-CIJ (indicating the i-th leaf node N_Q of R_Q), let s_i be the size of C_P and s'_i be the number of candidates that actually join with at least one Voronoi cell of a point in N_Q. The false hit ratio of the filter step is defined as: $FHR = \frac{\sum_{i=1}^{m} s_i - \sum_{i=1}^{m} s'_i}{\sum_{i=1}^{m} s'_i}$, where m is the total number of leaf nodes in R_Q. Figure 10a shows that the FHR is very low and not sensitive to the join input size. Figure 10b plots the false hit ratio with respect to the cardinality ratio $|Q| : |P|$, for $|Q| + |P|$=200K. For small $|Q| : |P|$, $|P|$ is large and many points in P lie close to border of Voronoi cells in Q; these cannot be pruned and result in redundant computations of the corresponding Voronoi cells, increasing the overall FHR. Nevertheless, the filter step of NM-CIJ is still effective as the false hit ratio for all cases remains below 0.1.



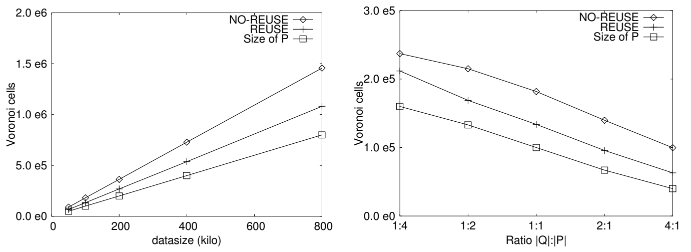
(a) vs n, $|Q| = |P| = n$



(b) vs $|Q| : |P|$, $|Q| + |P|$=200K

Fig. 10. False hit ratio (in NM-CIJ)

We now study the benefit of reusing Voronoi cells in NM-CIJ (see Section IV-B). Figure 11 plots the number of exact Voronoi cells computed for points in P by two versions of NM-CIJ: (i) REUSE, the standard version, and (ii) NO-REUSE, which does not reuse exact Voronoi cells of P computed in the last batch. The figure shows these numbers in parallel to the total number of points in P; the number of redundant cell computations in REUSE and NO-REUSE can be viewed as the difference between their lines and that of $|P|$ in the plots. Note that the relative benefit of REUSE over NO-REUSE is insensitive to the database size and the cardinality ratio. In general, the REUSE heuristic proves valuable, since it reduces by 50% (on the average) the redundant Voronoi cell computations of NO-REUSE.



(a) vs n, $|Q| = |P| = n$ (b) vs $|Q| : |P|$, $|Q| + |P|$=200K

Fig. 11. Reuse of Voronoi cells (in NM-CIJ)

Table III shows the join output size and the I/O costs of the CIJ algorithms on various pairs of real datasets. In general, the results are consistent with the experiments on synthetic data; NM-CIJ outperforms PM-CIJ which, in turn, is more efficient than FM-CIJ. The relative difference between the algorithms is not stable, for instance, at CE $\bowtie$ LO the relative difference between NM-CIJ and PM-CIJ is not as high as at PA $\bowtie$ PP. Observe that the join output size is comparable to input data size and not overwhelmingly large. In general, the number of join pairs increases slightly with the data skew.

TABLE III

RESULT SIZE AND PAGE ACCESSES OF CIJ

			Page accesses		
Q	**P**	**CIJ pairs**	**FM-CIJ**	**PM-CIJ**	**NM-CIJ**
SC	PP	583169	60476	40644	26613
CE	LO	422986	45310	28602	23160
CE	SC	426516	54885	41021	29082
LO	PP	623989	50754	36495	21992
PA	SC	344771	43468	36989	22612
PA	PP	383518	41334	34863	19744

VI. CONCLUSIONS

In this paper we identified a *natural* spatial join operation between pointsets; the common influence join (CIJ), which finds application in practical spatial data analysis tasks. Nevertheless, CIJ cannot be computed from traditional distance joins, due to the essential computation of Voronoi cells. We proposed an optimized technique for computing the Voronoi cell of a given point and extended it for batch computation. Based on this module, we develop three efficient CIJ algorithms that operate on pointsets indexed by R-trees. An experimental evaluation with synthetic and real datasets shows that our NM-CIJ algorithm is highly efficient, incurring only slightly higher I/O cost than the theoretical lower bound cost for the problem. Yet another advantage of NM-CIJ is that it is non-blocking; it starts producing CIJ pairs after few disk pages are accessed.

In the future, we will extend our solutions for 3D points, with the intuition that, the convex polygon $V_c(p_i)$ (of Lemma 1) in 2D space is analogous to a convex polyhedron in 3D space. Also, we plan to generalize CIJ computation for multiple pointsets and develop multiway CIJ algorithms.

ACKNOWLEDGEMENT

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

REFERENCES

[1] A. Okabe, B. Boots, K. Sugihara, and S. Chiu, *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*, 2nd ed. Wiley, 2000.
[2] C. Böhm, B. Braunmüller, F. Krebs, and H.-P. Kriegel, "Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data," in *SIGMOD*, 2001.
[3] G. R. Hjaltason and H. Samet, "Incremental Distance Join Algorithms for Spatial Databases," in *SIGMOD*, 1998.
[4] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, "Closest Pair Queries in Spatial Databases," in *SIGMOD*, 2000.
[5] J. Zhang, N. Mamoulis, D. Papadias, and Y. Tao, "All-Nearest-Neighbors Queries in Spatial Databases," in *SSDBM*, 2004.
[6] M. L. Yiu, X. Dai, N. Mamoulis, and M. Vaitis, "Top-k Spatial Preference Queries," in *ICDE*, 2007.
[7] I. Stanoi, M. Riedewald, D. Agrawal, and A. E. Abbadi, "Discovery of Influence Sets in Frequently Updated Databases," in *VLDB*, 2001.
[8] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," in *SIGMOD*, 1984.
[9] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient Processing of Spatial Joins Using R-Trees," in *SIGMOD*, 1993.
[10] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. Lee, "Location-based Spatial Queries," in *SIGMOD*, 2003.
[11] G. R. Hjaltason and H. Samet, "Distance Browsing in Spatial Databases," *TODS*, vol. 24(2), pp. 265–318, 1999.
[12] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter, "External-Memory Computational Geometry," in *FOCS*, 1993.
[13] M. Isenburg, Y. Liu, J. Shewchuk, and J. Snoeyink, "Streaming Computation of Delaunay Triangulations," in *SIGGRAPH*, 2006.
[14] S. Arya and A. Vigneron, "Approximating a Voronoi cell," *HKUST Research Report HKUST-TCSC-2003-10*, 2003.
[15] Y. Tao and D. Papadias, "Time Parameterized Queries in Spatio-Temporal Databases," in *SIGMOD*, 2002.
[16] M. Sharifzadeh and C. Shahabi, "The Spatial Skyline Queries," in *VLDB*, 2006.
[17] A. R. Butz, "Alternative Algorithm for Hilbert's Space-Filling Curve," *IEEE Trans. Comput.*, vol. C-20, no. 4, pp. 424–426, 1971.
[18] I. Kamel and C. Faloutsos, "Hilbert R-tree: An Improved R-tree using Fractals," in *VLDB*, 1994.