

# Surrounding Join Query Processing in Spatial Databases

Lingxiao Li<sup>(✉)</sup>, David Taniar, Maria Indrawan-Santiago, and Zhou Shao

Monash University, Melbourne, Australia

lli278@student.monash.edu,

{david.taniar,maria.indrawan,joe.shao}@monash.edu

**Abstract.** Spatial join queries play an important role in spatial database, and mostly all the distance-based join queries are based on the *range* search and nearest neighbour (NN), namely *range join query* and *kNN join query*. In this paper, we propose a new join query which is called *surrounding join query*. Given two point datasets  $Q$  and  $P$  of multidimensional objects, the *surrounding* query retrieves for each point in  $Q$  its all surrounding points in  $P$ . As a new spatial join query, we propose algorithms that are able to process such query efficiently. Evaluation on multiple real world datasets illustrate that our approach achieves high performance.

**Keywords:** Spatial join · Spatial indexing · Spatial database · Nearest neighbour

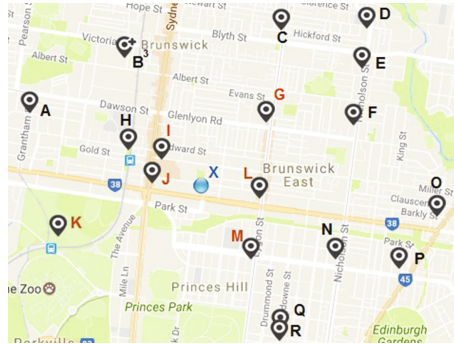
## 1 Introduction

The spatial join query involves two datasets  $Q$  and  $P$  retrieves the object pairs from the Cartesian Product  $Q \times P$  which satisfy a spatial predicate, From the theoretical point of view, the spatial join is similar as join that in the traditional database system domain. The main difference is join predicate, which can be intersection, topological, directional or distance, rather than simply equijoin. The intersection and distance-based join queries have been widely studied. A typical example of an intersection join is “*find all suburbs that are crossed by Southern Link Highway (M1), Western Link (M2) and East Link Highway (M3) in the city of Melbourne*”. In the example, we regard highway and suburb as spatial objects, line and polygon respectively. On the other hand, a case of distance-based join could be “*find all pairs of hotels and restaurants within 1 km apart*”. Both hotel and restaurant denote spatial point.

In this paper, we only focus on the distance-based join queries, most common join queries in this category are range join and  $k$ NN join. More specifically, the range join is a query for each query point finds all the target points that within the pre-specified range  $\epsilon$ . In contrast, the  $k$ NN join query retrieves  $k$  nearest neighbours for each query points. However, both above queries have some main problems. For range join query, the result set cardinality is difficult to control.

If the distance is defined too small or too large, the size of result set will change enormously in some situations. The problem of range join can be overcome by  $k$ NN join, which make sure each point in one dataset exactly combined with its  $k$  closest neighbours in the other dataset. However, if there is a cluster of points near all the query points, then the  $k$ NN join result is restricted to that scope [15]. Besides, before the query processing, we have to specify the range distance  $\epsilon$  for range join and value of  $k$  for  $k$ NN join.

Inspired by these limitations, we introduce a new join query, called the Surrounding Join (SJ) query and propose efficient query processing techniques. This new join query is based on the surrounding query. A surrounding query is a query to retrieve all the nearest objects that surround the query object. Figure 1 shows an example of surrounding query. In the figure, the blue point  $X$  denotes a user's position, and the black dots ( $A$  to  $R$ ) are all groceries in this suburb. From the perspective of this user, the query of surrounding groceries are the points  $\{G, I, J, L, K, M\}$ . If this user picks a surrounding grocery (for example  $I$ ), which means that she doesn't need to know the groceries ( $A, B, H$ ) behind the grocery  $I$ . In this case,  $A, B$  and  $H$  are dominated by  $I$ .



**Fig. 1.** An example of surrounding query (Color figure online)

In summary, the contributions of this paper are summarized as follows:

- We introduce surrounding join (SJ) query in spatial databases, which belongs to the distance-based join queries and involves spatial point data type.
- To solve the SJ queries, We propose two approaches; the first one is a straightforward algorithm that relies on a Voronoi diagram; The second approach is a hierarchical algorithm which prunes unnecessary nodes for obtaining the surrounding points. Meanwhile, it has higher performance.
- We have conducted extensive experimental studies on two real datasets that demonstrate the efficiency of our algorithms.

## 2 Related Work

In geospatial domain, the nearest neighbour (NN) query is to retrieve the points in the target dataset  $P$  that has shortest distance to a query point  $q$ . It has been widely used in many different type of queries, such as  $k$  Nearest Neighbor ( $k$ NN) [1, 6, 12], Reverse  $k$  Nearest Neighbor ( $Rk$ NN) [7, 14, 16, 17] and skyline query [2, 9, 13]. The existing NN algorithms always assume that the target dataset is indexed by an R-tree. In the R-tree index, the data point is completely and properly enclosed by a minimum bounding rectangle (MBR). In [11], Roussopoulos et al. proposed an algorithm to find the nearest neighbour object to a point, which is called *branch and bound R-tree traversal*. The metrics *MINDIST* denotes the minimum distance of point  $p_i$  to  $q$ ,  $p_i \in P$ . The algorithm access the R-tree in a depth first (DF) manner. Starting from root node, and the entry with the smallest *MINDIST* is accessed first. The process is repeated recursively until the leaf node is visited where a potential NN is found. An optimal NN algorithm has been introduced in [10]. Consider query point  $q$  is center point and radius equals to the distance from  $q$  to its NN, then a *query circle* is created. In this case, the algorithm only traverses nodes whose MBR intersect with *query circle*.

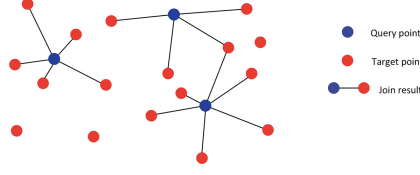
The Voronoi diagram (VD) of a given set  $G$  of  $k$  points  $\{g_1, g_2, \dots, g_m\}$  in a Euclidean plane partitions the space  $\mathbb{R}^d$  into  $k$  regions. Each region contains a point  $g_i$  ( $g_i \in G$ ) that is regard as generator point. The Euclidean distance from any other point in its region to  $g_i$  is smaller than to any other generator. Two generator points shares a Voronoi edge and three generator points form a Voronoi vertex. Existing algorithms for generating VD can be briefly divided into tree categories. The first category of algorithms are incremental algorithms, which create the VD by inserting a point at a time [5]. The second are divide and conquer algorithms. The set of points is divided into multiple parts, and VD of each part constructed recursively [19]. The last category of algorithms compute VD by implementing the sweepline technique [4].

## 3 Problem Definition

The surrounding join (SJ) query is defined as below:

**Definition 1 (*SJ Queries*).** *Given a set  $Q$  of  $m$  query points  $q_1, q_2, \dots, q_m$  and a set  $P$  of  $n$  target points  $p_1, p_2, \dots, p_n$ , a SJ query  $Q \bowtie_{SJ} P$  returns for each query point  $q_i \in Q$ , a sub-set  $P_i \subseteq P$ . In terms of the sub-set  $P_i$ ,  $\forall p_{j'} \in P_i$  to the query point  $q_i$  has the shortest Euclidean distance in a particular direction. Meanwhile,  $\forall p_{j'} \in P_i$  is not dominated by the other points in  $P_i$ .*

For a surrounding join query, we are going to find all the nearest target points that just surround each query point. As depicted in Fig. 2, three query points are respectively connected to its surrounding points.

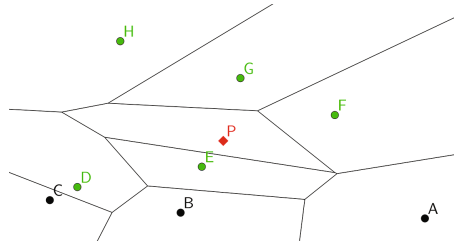


**Fig. 2.** Surrounding join

## 4 A Sketch-First Approach

Inspired by the Voronoi Diagram (VD), we can instantly get an idea that all the surrounding target points of each query point look like the adjacent vertexes in the VD [18]. Towards addressing this idea, a possible solution could be like this: Firstly, create a VD based on all the query and target points, and then for each query point  $q_i$  retrieves all the adjacent vertexes which surround this query point. Here, we assume that the adjacent vertexes are same as the surrounding points, which are what we need for the join query. To creating a VD, we apply the Fortune’s sweep line algorithm [4], which guarantees the  $O(n \log n)$  worst-case running time and uses  $O(n)$  space. However, there are some limitations of this straightforward approach: (i) the data from two datasets need to be merged first and then sorted as the input of the algorithm. If two datasets are very big, which contains millions of spatial points; The sorting phase will take a substantial amount of computing resources and very inefficient. (ii) When we add, update, delete points in any dataset, the VD will be changed accordingly. It means that we have to create a new VD for the join queries.

Figure 3 depicts an example of the VD processing. The target points are denoted as  $A, B, C, \dots, H$ . For simplicity and clarity, we only specify one query point  $P$  that is represented as red square. Obviously, the surrounding points of point  $P$  should be all adjacent vertexes, namely points  $\{D, E, F, G, H\}$ . In Fig. 3, we observe that each of these green points share a VD *edge* with query point  $P$ .



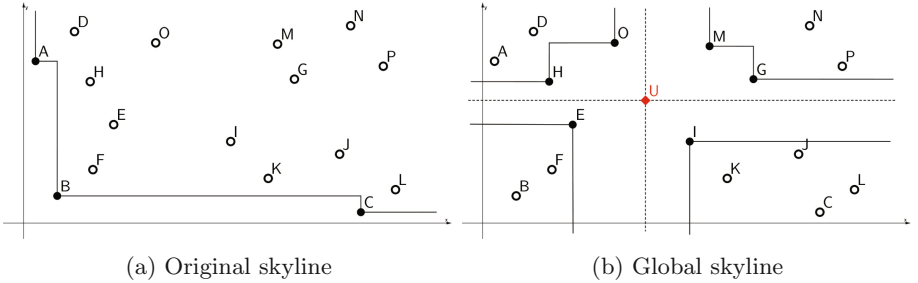
**Fig. 3.** VD approach (Color figure online)

## 5 Our Proposed Approach

In this section, we present our approach for the *surrounding join* queries computation, which is mainly composed of two parts: *Filter* and *Refinement*. More specifically, in *Filter* phase, we implement the global Branch and Bound skyline (GBBS) to prune all dominated points in target dataset. GBBS is an enhanced customization of the original *Branch and Bound Skyline* BBS algorithm [8]. Then, based on the skyline points from phase 1, a VD is created. The purpose the VD here is to help us retrieve all the adjacent points which surround the query point. We start by introducing the skyline and its variation *global-skyline*, and then we continue with a description of algorithms in *Filter* phase and *Refinement* phase.

### 5.1 Skyline

Give a set  $T$  of  $d$ -dimensional points, the original skyline operation returns all points in  $T$  are not dominated by any other point. More specifically, assume a point  $t_j$  is dominated by another point  $t_i$ , the condition is that coordinate of  $t_i$  on any axis is not greater than the corresponding coordinate of  $t_j$ , and strictly smaller in at least one axis. Informally, this implies that point  $t_i$  is preferable to  $t_j$  based on any real scenario. Figure 4(a) shows an example of original skyline in a two-dimensional space. Three solid dots  $A$ ,  $B$ ,  $C$  are skyline points which are dominating all the other points. If we refer to  $x$  and  $y$  axis to distance and price attribute, retroactively, and assume all the dots denote different restaurants. For instance, because point  $B$  dominates the point  $F$ , we can say restaurant  $B$  is better than restaurant  $F$ . The reason is that restaurant  $B$  is cheaper and closer than restaurant  $F$ . In short, the skyline of a multi-dimensional dataset encloses the *best* points according to any preference function that is monotone in each dimension [3].



**Fig. 4.** Skyline

Original skyline considers the static attribute values of each data point in the dimensional space, and only examines one direction. Meanwhile, the query

point is not involved into the operation. Since our aim is surrounding join, which means we need to consider all the directions rather than one direction. Besides, both query and target points should be taken into account. Accordingly, we apply *global-skyline* to solve our problem. As a variations of original skyline, *global-skyline* concerns about the potential targets points for each a query point, and returns all the points that are not *globally* dominated by other points. In other words, the *global-skyline* considers the directions of the processing, and the minimization of the coordinate distances between a query point and the target point is taken into account.

Figure 4(b) illustrates the *global-skyline* of query point  $U$  contains six points,  $H, O, M, G, I, E$ , which dominates the other points on all directions. Notice that these dominating points surround the query point  $U$ , which means no other target point is better than one of them with regard to  $U$ . Actually, these dominating points are the initial result of the surrounding join query. In the following section, we present the algorithm to generate these *global-skyline* points.

## 5.2 Filter Phase: GBBS Process

Same as NN and BBS, the GBBS join algorithm is also based on the nearest neighbour [11] search. Although all of these algorithms could be implemented by using data partition method, in this paper we use R-tree as index for target dataset due to its simplicity and popularity in spatial area. The set of 2-dimensional data points are used in Fig. 3, which is organized in the R-tree of Fig. 5.

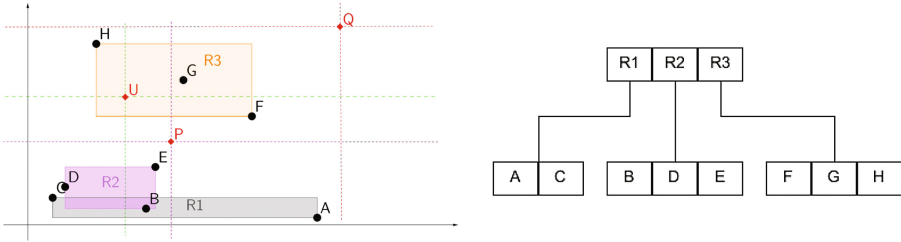


Fig. 5. R-tree

For the query processing, we take point  $P$  as the query point to describe the detail of the algorithm. GBBS starts from the root node of the R-tree and inserts all its entries ( $R_1, R_2, R_3$ ) in a empty heap. The element in the heap is sorted by the Euclidean distance from point  $P$  as ascending order. Then, the entry with the minimum distance  $R_3$  is expanded. This expansion prunes  $R_3$  from the heap and add its children ( $F, G, H$ ). Currently, the elements in the heap are ( $R_2, R_1, F, G, H$ ). Then, the entry  $R_2$  with minimum distance is expanded, and insert its children ( $E, B, D$ ), in which the first nearest data point  $E$  appears.

**Algorithm 1.** GBBS Join Algorithm

---

**Input:** query points  $Q$ , R-tree  $R$  of all target points  
**Output:**  $S$ : list of target points

```

1 initialization:  $i \leftarrow \{\}$ ,  $S \leftarrow \{\}$ , insert all entries of  $R$  in  $H$ ;
2 while  $H.size() \neq 0$  do
3    $e \leftarrow$  poll first element of  $H$ ;
4   if  $e$  is not leaf then                                     // MBR, intermediate node
5     for each child  $e_i$  of  $e$  do
6       if  $e_i$  is not dominate by any item in  $S$  then
7         insert  $e_i$  to  $H$ ;
8       end
9     end
10  else if  $e$  is leaf then                                     // leaf node
11    for each child  $e_i$  of  $e$  do
12      if  $e_i$  is not dominate by any item in  $S$  then
13        insert  $e_i$  to  $H$ ;
14      end
15    end
16  else
17    insert  $e_i$  to  $S$ ;
18  end
19 end
20 Return result list  $S$ 

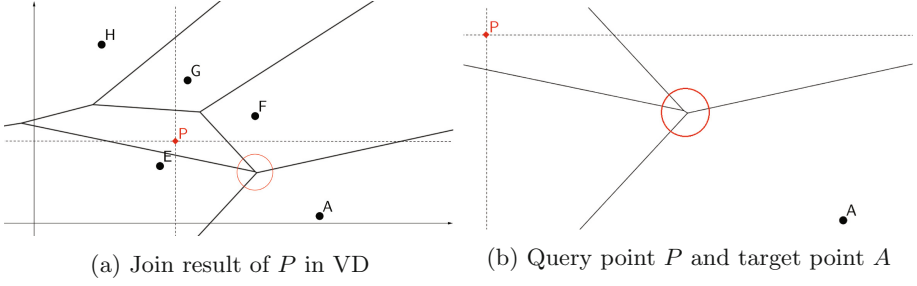
```

---

Point  $E$  belongs to the global skyline and is added to result list  $S$ . After we moved  $E$  from the heap to  $S$ . The first element in the heap is  $R1$ , which still is an intermediate node, not the real data node. GBBS proceeds with the  $R1$  and inserts its children ( $A$ ,  $C$ ). The heap now becomes ( $G$ ,  $B$ ,  $F$ ,  $D$ ,  $C$ ,  $H$ ,  $A$ ), and  $S = \{E\}$ . The algorithm processes in the same way until the heap becomes empty thus all global skyline points are added in the result list  $S$ . The join result of  $P$  in phase 1 is a list  $\{E, F, G, H, A\}$ . The join operation between the other two query point  $U$ ,  $Q$  and target points will be processed as the same manner, and the result about  $U$ ,  $Q$  are the list  $\{B, E, F, G, H, D\}$  and  $\{A, F, G, H\}$ , respectively. The pseudo-code of GBBS is shown in Algorithm 1.

### 5.3 Refinement Phase: VD Process

In this phase, we consider the join result from step 1 is candidate result which needs further process. Therefore, we create VD based on the query point and skyline points which come from step 1. This means, as soon as we find the skyline  $S$  of query point  $q$ , we check if the point  $t$  in  $S$  is the adjacent of the  $q$ . If this is the case, we add this  $t$  to the final join result. Otherwise, we can safely prune point  $t$ . Note that the number of target points is much smaller than the number of the original dataset and consequently, the VD can be created much faster. For generating VD, we still use Fortune's sweep line algorithm [4].

**Fig. 6.** Candidate result in VD

In the following discussion, we continue to use query point  $P$  to describe the detail of the refinement algorithm. So far, we have already got the skyline points of  $P$  is a list  $\{E, F, G, H, A\}$ . Therefore, we merge  $p$  and five skyline points together and create a VD. The VD is shown in the Fig. 6(a). Next, we retrieve the points around the cell of  $P$ . If a point share a same edge with  $P$ , then we add this point to reset list. For example, each of point  $E, F, G$  and  $H$  shares a VD edge with  $P$ , we can say the surrounding join result of  $P$  is the list of target points  $\{E, F, G, H\}$ . Note that, the point  $A$  is pruned in this phase. The reason is obvious, because  $A$  is not the adjacent of  $P$ . In Fig. 6(b), We can see there is an edge between  $P$  and  $A$ . For the other query points, the same process can be followed to get the join result. The pseudo-code of refinement is shown in Algorithm 2.

---

**Algorithm 2.** Refinement Algorithm
 

---

**Input:** query points  $Q$ , Skyline points  $S$ 
**Output:** Result set  $R$ 

```

1 initialization: Merger  $Q$  and Skyline points, generate  $VD$ ;
2 for each edge  $e$  of  $VD.edges$  do
3   if  $e.left$  is equal to  $Q$  then
4     insert  $e.right$  into  $R$ ;
5   else if  $e.right$  equals  $Q$  then
6     insert  $e.left$  into  $R$ ;
7 end
8 Return result set  $R$ 

```

---

At this point, we get the join result of query points  $U, P, Q$  and target points  $A, B, C, D, E, F, G, H$ . For Point  $P$ , if implement sketch-first approach, which is a pure VD approach, then we get the join result  $P \rightarrow \{D, E, F, G, H\}$ . The detail is shown in Sect. 4, Fig. 3. In contrast, in our improved approach, the join result of  $P$  is the  $\{E, F, G, H\}$ . If we compare these two lists, we can find that the first approach contains extra point  $D$ . Consider this case based on the



perspective of the global-Skyline; we can see  $D$  is dominated by  $E$ . Note that,  $E$  is a surrounding point of  $P$ . Since there are no other existing approaches to answer the surrounding join query, and we are the first to propose these two possible solutions. Therefore, if we only consider these approaches, the second approach is more accurate than sketch-first approach.

## 6 Experiments

### 6.1 Experimental Setup

According to our literature research, in addition to our two approaches, there are no prior methods to process surrounding join queries. Therefore, we compare two proposed algorithms with each other to evaluate their performance. We refer to our sketch-first approach, improved approach as the *VDS* and *SVDS* in the following evaluation report, respectively. The experiments are performed on the real datasets which are road network of San Francisco and California. Both datasets are retrieved from the website<sup>1</sup>. For the input data, we randomly obtain 2000 points and set them as query points. Then, get rid of those 2000 points, we randomly generate five target datasets which contain 2000, 5000, 8000, 11000 and 14000 points, respectively. The experiments are repeated 100 times, and the average value is reported. All algorithms were implemented in JAVA and experiments were conducted on a Linux PC with 16 Intel Xeon E312xx 2 GHz CPUs and 64 GB main memory.

### 6.2 Experimental Results

In this section, we will evaluate two approaches of surrounding join query from four different perspectives, namely index construction, a diverse number of target points, a diverse number of query points and the detail of the second approach.

**Evaluation on Index Construction:** In the first approach *VDS*, we create a Voronoi Diagram first, then conduct the join operation on this VD. We assume the VD that stored in the main memory is a kind of index. Besides, The second approach *SVDS* is based on the R-Tree. Therefore, we compare the time and space consumption of the index construction of two approaches. Figure 7(a) illustrates the runtime of two index construction on same datasets. We specify the number of query point equals to 2000 and gradually increase the number of target points. The index creation time of *SVDS* is slighter faster than *VDS*. However, the VD-index need more memory space to store index as shown in Fig. 7(b). Specifically, the size of VD-index is about seven times that of *SVDS*-index, which dues to that VD-index involves both query and target datasets, and the structure is not good as R-Tree.

**Evaluation on the Varied Number of Query and Target Points:** We evaluate the overall performance of *VDS* and *SVDS* from two perspectives,

<sup>1</sup> <http://www.cs.utah.edu/~lifeifei/SpatialDataset.htm>.

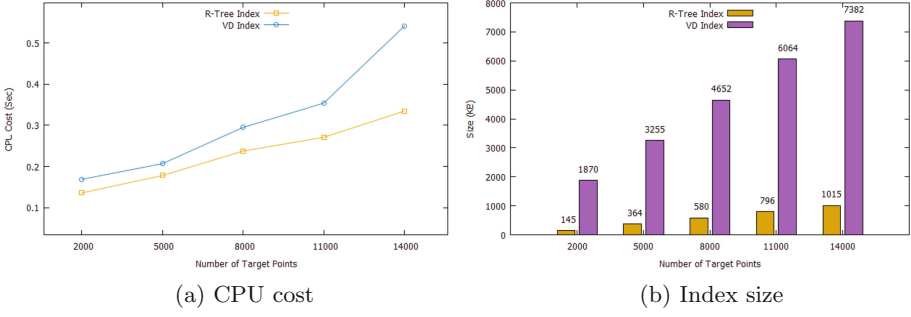


Fig. 7. Index construction

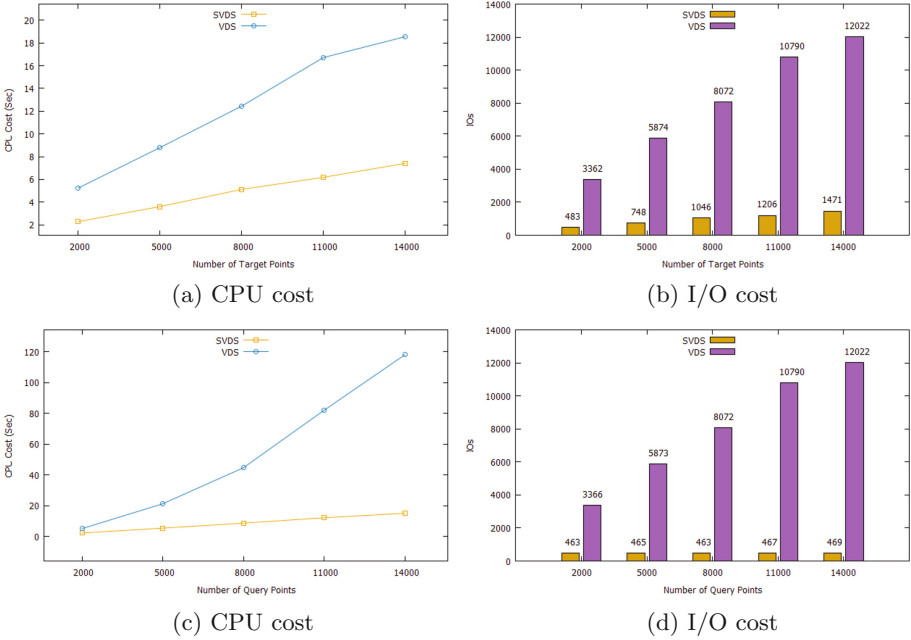


Fig. 8. Effect of varying number of points

CPU cost and average I/O cost. Figure 8(a) shows the CPU cost and Fig. 8(b) shows the I/O cost of each method for increasing the number of the target point. The number of query point is specified as 2000. When the number of target point increases, the CPU and I/O cost of both two approaches increase correspondingly. However, with the increment of the number of target points, the cost of *VDS* rises rapidly because it has to access all the points. On the other hand, we set target point equals to 2000 and increase the number of the query point. For this case, Fig. 8(c) and (d) show the processing time and I/O cost, respectively. The experimental results still indicate that *SVDS* is more efficient

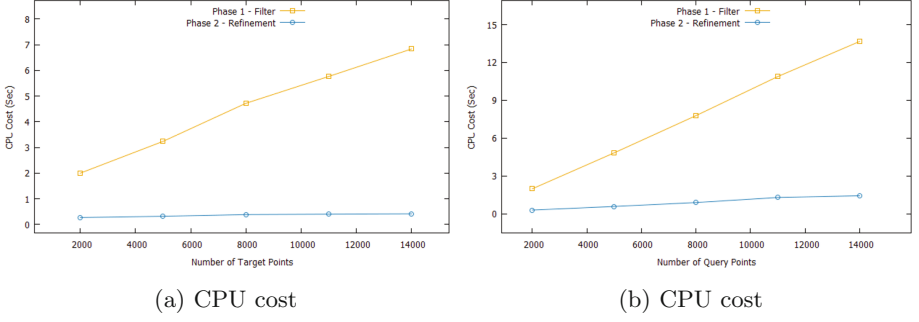


Fig. 9. Performance comparison on two phases of SVDS

than *VDS*. The main reason behind this is *SVDS* prunes all the dominated points in the filter phase. Nevertheless, *VDS* always access all the points.

**Evaluation of *SVDS*:** Based on the above evaluation, we understand the performance of *SVDS* is much better than *VDS*. Evaluating the *filter* and *refinement* phase in *SVDS* is necessary. In Fig. 9(a), we observe that the running time of *filter* phase increases gradually with the increase of the number of the target point. In contrast, the *refinement* phase remains almost constant during the whole experiment. The CPU cost of two phases is displayed in Fig. 9(b), which roughly illustrates the similar characteristics as Fig. 9(a). The reason is obvious. The candidate join result as the input for refinement phase which comes from filter phase is much smaller than the original size of target points.

## 7 Conclusion

In this paper, we introduced a new type of query, namely *Surrounding Join Query* that enables for each query point to identify the surrounding target points. It enriches the semantics of the conventional distance-based spatial join query. To efficiently process a *Surrounding Join Query*, we proposed two approaches. The first one, *VDS*, relies on the Voronoi Diagram. In contrast, the second approach, *SVDS* that combines the skyline and Voronoi Diagram to answer the query. Our experiments also illustrate that our algorithm has the capability to process the query efficiently. In the future, we are going to implement the surrounding for range join query and other spatial data types.

## References

1. Berchtold, S., Ertl, B., Keim, D.A., Kriegel, H.-P., Seidl, T.: Fast nearest neighbor search in high-dimensional space. In: Proceedings of the 14th International Conference on Data Engineering, pp. 209–218. IEEE (1998)
2. Borzsony, S., Kossmann, D., Stocker, K.: The skyline operator. In: Proceedings of the 17th International Conference on Data Engineering, pp. 421–430. IEEE (2001)

3. Dellis, E., Seeger, B.: Efficient computation of reverse skyline queries. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 291–302. VLDB Endowment (2007)
4. Fortune, S.: A sweepline algorithm for Voronoi diagrams. In: Proceedings of the Second Annual Symposium on Computational Geometry, pp. 313–322. ACM (1986)
5. Green, P.J., Sibson, R.: Computing dirichlet tessellations in the plane. *Comput. J.* **21**(2), 168–173 (1978)
6. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. *ACM Trans. Database Syst. (TODS)* **24**(2), 265–318 (1999)
7. Korn, F., Muthukrishnan, S.: Influence sets based on reverse nearest neighbor queries. *ACM SIGMOD Rec.* **29**, 201–212 (2000). ACM
8. Papadias, D., Tao, Y., Fu, G., Seeger, B.: An optimal and progressive algorithm for skyline queries. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 467–478. ACM (2003)
9. Papadias, D., Tao, Y., Greg, F., Seeger, B.: Progressive skyline computation in database systems. *ACM Trans. Database Syst. (TODS)* **30**(1), 41–82 (2005)
10. Papadopoulos, A., Manolopoulos, Y.: Performance of nearest neighbor queries in R-trees. In: Afrati, F., Kolaitis, P. (eds.) *ICDT 1997*. LNCS, vol. 1186, pp. 394–408. Springer, Heidelberg (1997). doi:[10.1007/3-540-62222-5\\_59](https://doi.org/10.1007/3-540-62222-5_59)
11. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. *ACM SIGMOD Rec.* **24**, 71–79 (1995). ACM
12. Shao, Z., Cheema, M.A., Taniar, D., Lu, H.: Vip-tree: an effective index for indoor spatial queries. *PVLDB* **10**(4), 325–336 (2016)
13. Sharifzadeh, M., Shahabi, C.: The spatial skyline queries. In: Proceedings of the 32nd International Conference on Very Large Data Bases, pp. 751–762. VLDB Endowment (2006)
14. Stanoi, I., Agrawal, D., El Abbadi, A.: Reverse nearest neighbor queries for dynamic databases. In: *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pp. 44–53 (2000)
15. Taniar, D., Rahayu, W.: A taxonomy for nearest neighbour queries in spatial databases. *J. Comput. Syst. Sci.* **79**(7), 1017–1039 (2013)
16. Taniar, D., Safar, M., Tran, Q.T., Rahayu, W., Park, J.H.: Spatial network RNN queries in GIS. *Comput. J.* **54**(4), 617–627 (2011)
17. Tao, Y., Papadias, D., Lian, X.: Reverse KNN search in arbitrary dimensionality. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, vol. 30, pp. 744–755. VLDB Endowment (2004)
18. Xuan, K., Zhao, G., Taniar, D., Safar, M., Srinivasan, B.: Voronoi-based multi-level range search in mobile navigation. *Multimedia Tools Appl.* **53**(2), 459–479 (2011)
19. Yap, C.K.: Ano (n logn) algorithm for the Voronoi diagram of a set of simple curve segments. *Discrete Comput. Geom.* **2**(1), 365–393 (1987)

Databases Theory and Applications

28th Australasian Database Conference, ADC 2017,

Brisbane, QLD, Australia, September 25–28, 2017,

Proceedings

Huang, Z.; Xiao, X.; Cao, X. (Eds.)

2017, XXVII, 286 p. 81 illus., Softcover

ISBN: 978-3-319-68154-2