Identifying the Most Interactive Object in Spatial Databases

Daichi Amagata
Osaka University
amagata.daichi@ist.osaka-u.ac.jp

Takahiro Hara
Osaka University
hara@ist.osaka-u.ac.jp

***Abstract*—This paper investigates a new query, called an MIO query, that retrieves the Most Interactive Object in a given spatial dataset. Consider that an object consists of many spatial points. Given a distance threshold, we say that two objects interact with each other if they have a pair of points whose distance is within the threshold. An MIO query outputs the object that interacts with other objects the most, and it is useful for analytical applications e.g., neuroscience and trajectory databases.**

The MIO query processing problem is challenging: a nested loop algorithm is computationally inefficient and a theoretical algorithm is computationally efficient but incurs a quadratic space cost. Our solution efficiently processes MIO queries with a novel index, BIGrid (a hybrid index of compressed Bitset, Inverted list, and spatial Grid structures), with a practical memory cost. Furthermore, our solution is designed so that previous query results and multi-core environments can be exploited to accelerate query processing efficiency. Our experiments on synthetic and real datasets demonstrate the efficiency of our solution.

I. Introduction

Recently, to enable data analysis by spatial query processing, an object has often been modeled by *a set of spatial points*, in, for example, neuroscience [1], computer vision [2], and trajectory databases [3]. Fig. 1 illustrates a real object, namely a rat neuron [4] which is represented by three-dimensional points. In this paper, we focus on such objects and devise a novel analytical query, called MIO, that retrieves the most interactive object in a given spatial dataset.

Consider an object collection O where each object $o \in O$ is a set of three-dimensional (or two-dimensional) points (i.e., $o = P = \{p^1, p^2, ..., p^{|P|}\}$). An MIO query specifies a distance threshold r, and the score of o is defined by the number of objects $o' \in O \backslash \{o\}$ where there exists a pair of points $p \in o$ and $p' \in o'$ such that $dist(p, p') \leq r$. The query outputs the object with the highest score. Assume that o could interact with o' if $\exists p \in o$ and $\exists p' \in o'$ such that $dist(p, p') \leq r$. Then we see that an MIO query identifies *the object that could interact with other objects the most*.

A. Motivating examples

In the following examples, which motivate us to address the problem of MIO query processing, we demonstrate how analytical applications benefit from the problem.

Example 1 (Neuroscience). *Neuroscience simulations have recently been modeling a neuron by three-dimensional points to study neuronal mechanisms [5], [6]. Neurons communicate by transmitting signals through synaptic connections between them, and it is well known that synapses can form only when axons and dendrites (parts of neurons) are within close proximity. Therefore, a simulation specifies a distance threshold r, and assumes that two neurons interact (i.e., can communicate) with each other if they respectively have points p and p' such that $dist(p, p') \leq r$ [1], [7], [8]. It is also known that, like hubs in networks, there are some neurons that have synaptic connections with many neurons [9]. Such neurons play a key role in brain function, thereby identifying and analyzing them is important [10]. Since our problem retrieves the most interactive object, MIO queries enable neuroscientists to obtain important neurons from simulations.*

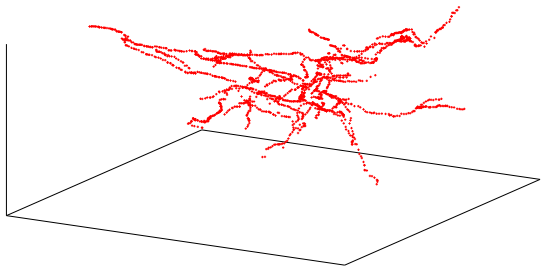


Fig. 1. A neuron represented by 3-dimensional points [4]

Example 2 (Trajectory analysis). *Because of advances in bio-logging technologies, trajectories of many animals have been collected [11]. Because trajectories are effective objects for studying the features and behaviors of animals [12], trajectory analysis has been receiving much attention [13]–[15]. Our problem also could help understanding animal behaviors. For example, it helps to mine motion patterns with spatial constraints. In Fig. 2, we show a red trajectory o, which is identified by an MIO query ($r = 4$[m]) on a bird trajectory set [11]. This trajectory interacts with approximately 30% of all trajectories. We see that some other trajectories have moving patterns similar to that of o. Birds have social relationships with spatial constraints [16], and one example is leader-follower (i.e., many individuals follow the motion pattern of a leader) [17]. How to organize social relationships can be analyzed from the MIO query results, e.g., by extracting the (sub-)trajectories of o and their nearby (sub-)trajectories [18].*

B. Challenge

In the applications described in Section I-A, users would utilize MIO queries while varying the distance threshold r,

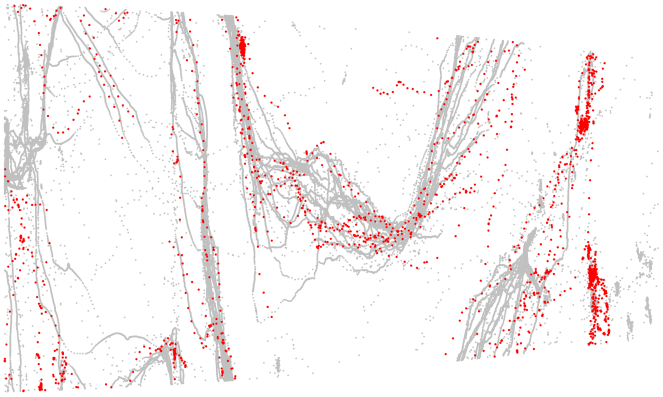


Fig. 2. The red trajectory is identified by our problem and interacts with approximately 30% trajectories in a trajectory set [11], when $r = 4$[m] (x-y coordinates are respectively [1000000, 5000000] and [100000, 700000]).

because they need to obtain deep insights. However, if each MIO query incurs a long processing time, only a limited number of trials (e.g., simulations) may be possible. To avoid such a situation, an algorithm that efficiently processes a given MIO query is required.

Trade-off between computational and memory efficiencies. A naive algorithm for this problem is a nested loop approach: for each object $o \in O$, we execute pairwise distance computation between $p \in o$ and $p' \in o'$ for $\forall p \in o$, $\forall p' \in o'$, and $\forall o' \in O \backslash \{o\}$. Let n and m be the cardinality of O and the average number of points in an object $\in O$, respectively. The nested loop algorithm incurs $\mathcal{O}(n^2m^2)$ time, which is not scalable. As we show in Section II-B, there is an online algorithm that can process an MIO query in $\mathcal{O}(n \log n)$ time. Although this algorithm is theoretically sound, it requires very costly pre-processing and incurs $\mathcal{O}(n^2)$ memory, which is also impractical. These two algorithms, in addition, imply that this problem has a trade-off relationship between computational and memory efficiencies. Therefore, a good solution should process an MIO query with a practical memory cost.

Leveraging previous results and multiple CPU cores. Analytical applications issue many queries, and, in the motivating applications, distance thresholds are usually fine-grained [14], [19]. From this observation, it is intuitively seen that some intermediate results of previously issued MIO queries can be leveraged to accelerate processing a given MIO query. Moreover, recent CPUs are equipped with multi-core processors, and multi-core processing is an option for reducing latency. To obtain its benefit, however, a partitioning approach, which satisfies load balancing, is required. This renders a non-trivial challenge that the solution can make good use of multiple CPU cores.

C. Contribution

We overcome the above challenges and make the following contributions.

- We address the problem of identifying the most interactive object in a spatial dataset (Section II). To the best of our knowledge, this problem has not been tackled so far.
- We propose a novel index, BIGrid (a hybrid index of compressed Bitset, Inverted list, and spatial Grid structures). Given a distance threshold, a BIGrid is built online, thereby our solution does not rely on any pre-processing. This BIGrid facilitates efficient score lower-bounding, upper-bounding, and verification, with a reasonable memory cost. During these operations, we employ a labeling approach, to suggest which points are necessary to exactly compute lower-bound, upper-bound, and score for future queries (Section III).
- Furthermore, we optimize our solution to parallelize BIGrid building, lower-bounding, upper-bounding, and verification, while considering load balancing in multi-core environments (Section IV).
- We conduct extensive experiments on both real and synthetic datasets to evaluate our solution (Section V). The results demonstrate that our solution is basically more than 10 times faster than competitors.

In addition to the above contents, related works are reviewed in Section VI, and Section VII concludes the paper.

II. Preliminary

A. Problem definition

Let O and n respectively be a collection of objects and its cardinality (i.e., $n = |O|$). An object $o_i \in O$ is a set of three-dimensional points[1], that is, $o_i = P_i = \{p_i^1, ..., p_i^{|P_i|}\}$. Each object in O may have a different number of points. Let m be the average point size in O (i.e., $m = \frac{\sum |P_i|}{n}$). We assume that O is memory-resident and static. Given a distance threshold r and two objects o and o', we consider that they have an interaction if they have a pair of points $p \in o$ and $p' \in o'$ such that $dist(p, p') \leq r$. Note that $dist(p, p')$ computes the Euclidean distance between p and p', as employed in [1], [8]. An MIO query provides a score with o by taking into account the above concept[2]. Now we define MIO queries.

Definition 1 (MIO query). *Given a collection of objects O and an MIO query with a user-specified distance threshold $r > 0$, the score of $o_i \in O$, $\tau(o_i)$, is defined as follows.*

$$\tau(o_i) = |O_i|$$

where

$$O_i = \{o_j \mid o_j \in O \backslash \{o_i\}, \exists p \in o_i, \exists p' \in o_j, dist(p, p') \leq r\} \quad (1)$$

The MIO query outputs o^*, the object in O with the highest score. (Ties are broken arbitrarily).

The objective of this paper is to provide the exact answer while minimizing the processing time.

[1]Dealing with two-dimensional points is straightforward, thus its detail is omitted. In addition, as we consider geo-spatial points, higher dimensional points are out of the scope of this paper.

[2]Although we consider only spatial dimension, some applications may require to take a temporal dimension into account. For example, two objects interact with each other iff they have points whose distance is within r and the difference of their generation time is within δ. We can deal with this case without non-trivial extension (see Appendix B)

B. Nested loop and theoretical algorithms

The MIO query processing problem is challenging, due to its scoring function. The score of o_i is obtained from the distance-based relationships between o_i and the other objects. Besides, as illustrated in Figs. 1 and 2, the shapes of objects are complex. From this fact, we see that building minimum bounding rectangle based indices, e.g., R-trees, is not effective, because they would make uselessly large rectangles with large empty spaces. Since this problem has not yet been addressed, we first consider the following solutions.

Nested loop algorithm. Perhaps the most intuitive approach is based on spatial self-join, which retrieves all pairs of points whose distances are within r. It is important to note that we do not have to retrieve all pairs of points. Assume that we compute $\tau(o)$ and now compare o and o'. When we find that $dist(p, p') \leq r$, where $p \in o$ and $p' \in o'$, we no longer have to find other pairs of points between o and o'.

Algorithm 1: Nested loop (NL)

```
   Input: O, r
1  o* ← ∅
2  for ∀o_i ∈ O do
3      for ∀o_j ∈ O such that j > i do
4          f ← 0
5          for ∀p_i^k ∈ o_i do
6              for ∀p_j^l ∈ o_j do
7                  if dist(p_i^k, p_j^l) ≤ r then
8                      τ(o_i) ← τ(o_i) + 1, τ(o_j) ← τ(o_j) + 1
9                      f ← 1
10                     break
11             if f = 1 then
12                 break
13     if τ(o*) < τ(o_i) then
14         o* ← o_i
15 return o*
```

Algorithm 1 details NL, a non-indexed nested loop algorithm. NL does not require any pre-processing, which is an advantage. However, to obtain $\tau(o)$, $\mathcal{O}(nm^2)$ time is required. NL hence incurs $\mathcal{O}(n^2m^2)$ time, which is very costly.

Theoretical algorithm. We next consider a theoretically faster algorithm than NL and show that there is an online algorithm that outputs o^* in $\mathcal{O}(n \log n)$ time.

THEOREM 1. *There is an online algorithm that outputs o^* in $\mathcal{O}(n \log n)$ time.*

PROOF. Assume that for an object $o_i \in O$, we have an array A_i that stores the distances of the closest point pairs between o_i and the other objects in O. If the distance of the closest point pair between o_i and o_j is within a distance threshold r, it is guaranteed that $o_j \in O_i$ (see Equation (1)). Assume furthermore that A_i is sorted in ascending order of distance. Given a distance threshold r, $\tau(o_i)$ is obtained by a binary search on A_i, which requires $\mathcal{O}(\log n)$ time. To compute o^*, we execute the same operation for all $o_j \in O$. This algorithm therefore can output o^* in $\mathcal{O}(n \log n)$ time. □

Although this algorithm is theoretically sound and the index (arrays) is general to any r, there are some critical drawbacks in practice. First, its space cost is $\mathcal{O}(n^2)$, which is derived from the arrays for all objects in O and is not scalable. Second, its pre-processing cost is significant. To build A_i, we have to retrieve the closest point pairs for all $o_j \in O \backslash \{o_i\}$. Given o_i and o_j, finding their closest pair requires at least $\mathcal{O}(|P_i| \log |P_j|)$ time [20], whose amortized cost is $\mathcal{O}(m \log m)$. Finding the closest pairs between o_i and the other objects thereby requires $\mathcal{O}(nm \log m)$. Besides, sorting them incurs $\mathcal{O}(n \log n)$ time. We now see that building A_i requires $\mathcal{O}(n(m \log m + \log n))$ time, thus the pre-processing cost is $\mathcal{O}(n^2(m \log m + \log n))$.

In summary, NL is computationally inefficient. The theoretical algorithm is space inefficient and its pre-processing cost is prohibitive, thus cannot be a practical option for analytical applications that want to obtain some insights as soon as data is available [21]. Motivated by these results, we design an algorithm that can efficiently process an MIO query with a practical memory cost.

III. MIO QUERY PROCESSING

The main bottleneck of MIO query processing is score computation, since, when $\tau(o)$ is computed, o has to be compared with the other objects. Therefore, to minimize processing time, it is important to reduce the number of score computations. The challenge is how to prune unnecessary score computations without sacrificing correctness. We overcome this challenge and propose an efficient solution.

Main idea. To prune unnecessary score computations, we devise lower- and upper-bounding techniques. A natural question is how to compute a lower-bound and an upper-bound for a given object *with reasonable costs*. Our answer is derived from the facts that (i) *the score is the set size* and (ii) a pair of points whose distance *can be* or *certainly is* within r is efficiently obtained by employing spatial grids. For points $p \in o$, if we know the distances between p and points of other objects are certainly within r and the corresponding objects are represented by bitsets, we can obtain a lower-bound of $\tau(o)$ by using a fast bitwise OR operation. An upper-bound of $\tau(o)$ is also obtained similarly. Our solution employs this idea and a novel grid-based index that enables such lower- and upper-bounding. A nice property of our lower- and upper-bounding is that they do not need distance computation.

Algorithm 2: Framework

```
   Input: O, r
1  GRID-MAPPING(O, r)                                  ▷build the BIGrid
2  τ_max^low ← LOWER-BOUNDING(O, r)                    ▷lower-bound comp.
3  O_cand ← UPPER-BOUNDING(O, r, τ_max^low)            ▷upper-bound comp.
     and pruning
4  o* ← VERIFICATION(O_cand, r)                        ▷exact score comp.
5  return o*
```

Framework. Algorithm 2 describes the overview of our solution. To start with, we build our novel index BIGrid online and then employ a filter-and-verification framework. More specifically, given an MIO query, we first execute GRID-MAPPING(O, r) that builds a BIGrid. Then, for each $o \in O$,

we compute its lower-bound score by utilizing the BIGrid and obtain the maximum lower-bound $\tau^{low}_{\max}$ in LOWER-BOUNDING(O, r). We next compute an upper-bound score for each $o \in O$, prune objects that cannot be o^*, and insert non-pruned objects into O_{cand}, in UPPER-BOUNDING$(O, r, \tau^{low}_{\max})$. After that, in VERIFICATION(O_{cand}, r), we compute the exact score of $o \in O_{cand}$ if it needs to be verified.

In the following, we elaborate the detail of BIGrid in Section III-A. Our bounding technique is presented in Section III-B, and the verification step is described in Section III-C. Besides, in Section III-D, we introduce how to use previous results to accelerate MIO query processing.

A. BIGrid structure

We devise a novel index *BIGrid* (a hybrid index of compressed Bitset, Inverted list, and spatial Grid structures), to enable efficient bounding and score computation. A BIGrid consists of two uniform grids, small- and large-grids, which are formally defined below.

DEFINITION 2 (SMALL-GRID). *The small-grid for an MIO query with r is a set of cells c^s_K, where K represents a key, and is implemented by a hash table. The width of each cell is $\frac{r}{\sqrt{3}}$. (Given a point $p \in o_i$, its key for the small-grid is obtained from its spatial coordinates and $\frac{r}{\sqrt{3}}$.) Each cell c^s_K has a compressed bitset $\mathbf{b}(c^s_K)$ whose i-th bit is 1, iff o_i has a point whose key is K.*

DEFINITION 3 (LARGE-GRID). *The large-grid for an MIO query with r is a set of cells c^l_K, where K represents a key, and is implemented by a hash table. The width of each cell is $\lceil r \rceil$. (Given a point $p \in o_i$, its key for the large-grid is obtained from its spatial coordinates and $\lceil r \rceil$.) Each cell c^l_K has an inverted list $I(c^l_K)$ and two compressed bitsets $\mathbf{b}(c^l_K)$ and $\mathbf{b}^{adj}(c^l_K)$. $I(c^l_K)$ is a collection of posting lists $I(c^l_K)[o_i]$, each of which maintains a set of points p^j_i whose keys are K. For $\mathbf{b}(c^l_K)$, iff $I(c^l_K)[o_i]$ exists, the i-th bit is 1. On the other hand, $\mathbf{b}^{adj}(c^l_K) = \bigvee \mathbf{b}(c^l_{K'})$, where K' is K or the key of the adjacent cell of c^l_K.*

Fig. 3 illustrates the BIGrid structure[3], and we show an example of a cell of the large-grid below.

EXAMPLE 3. *Fig. 4 illustrates a cell c^l_K of the large-grid, where points p^1_i, p^4_i, p^2_{i+1}, and p^1_{i+2}, whose keys are K, are mapped into the cell. These points are maintained by the inverted list $I(c^l_K)$, and the i-th, $(i+1)$-th, and $(i+2)$-th bits of $\mathbf{b}(c^l_K)$ are 1 (we assume that they are compressed).*

It can be seen that the small-grid does not maintain points and the large-grid does with inverted lists. The small- and large-grids do not share their keys, since, given a point p, its key for the small-grid (large-grid) is obtained from its coordinates and $\frac{r}{\sqrt{3}}$ ($\lceil r \rceil$). Note that, given a cell of the large-grid, accessing its adjacent cell takes $\mathcal{O}(1)$ amortized time.

[3]BIGrid is orthogonal to any compressed bitset, and we use EWAH [22] as our implementation. Selecting an optimal compressed bitset for a given dataset is beyond the scope of this paper.

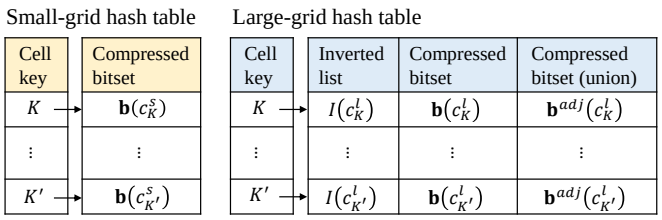


Fig. 3. Illustration of the BIGrid structure

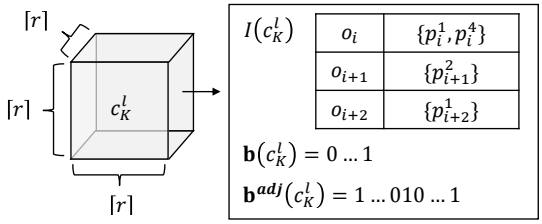


Fig. 4. Illustration of a cell c^l_K of the large-grid

An intuition behind the idea of employing compressed bitset is its effectiveness for skewed datasets. It is well known that there are dense and sparse spaces in real datasets [6], as illustrated in Fig. 2. In general, a compressed bitset compacts a sequence of bits if each bit of the sequence is the same, such as 00...0 (corresponding to a sparse space) and 11...1 (corresponding to a dense space). In both cases, the compressed bitset becomes compact, which is not only space efficient but also computationally efficient. If we use uncompressed bitsets, each cell *always* needs n bits, which incurs a large amount of memory[4]. (The compressed bitset(s) of each cell may or may not have different bit length due to the compression.) The small-grid (large-grid) is exploited for lower-bounding (upper-bounding and verification), and we show why we use a ceiling function for large-grid in Section III-D.

GRID-MAPPING(O, r). Definitions 2 and 3 describe that a BIGrid is specific to a user-specified distance threshold r. Therefore, the BIGrid is built online. Algorithm 3 details the BIGrid construction. For all $o_i \in O$, we execute the following operations for each $p^j_i \in o_i$.

Building the small-grid. We first compute the key of p^j_i for the small-grid, and let the key be K. If c^s_K exists in the small-grid, we set the i-th bit of $\mathbf{b}(c^s_K)$ as 1[5]. (If it does not exist, we simply create the cell, update the bit, and insert it into the small-grid.) Let $|\mathbf{b}(c^s_K)|$ be the number of bits set as 1 in $\mathbf{b}(c^s_K)$. We confirm whether $|\mathbf{b}(c^s_K)|$ becomes 2 or more. If it becomes 2, the cell has some points of two objects, and assume that the i'-th bit is also 1. We make o_i and $o_{i'}$ maintain the key K with key list (denoted by $o_i.L$ and $o_{i'}.L$), which is utilized in the lower-bounding step. If it becomes more than 2, we update only $o_i.L$, because the other corresponding objects already contain K in their key lists.

[4]In the case of the default setting of our experiment, the compression ratio of the compressed bitsets is more than 80% and at most 99.9% in bytes, compared with uncompressed bitsets.

[5]When the i-th bit is set as 1, its lower bits, except the i'-th ($i' < i$) ones where $o_{i'}$ has a point whose key is K, are considered to be 0, and compressed (if possible).

Algorithm 3: GRID-MAPPING(O, r)

```
1  for ∀o_i ∈ O do
2      for ∀p_i^j ∈ o_i do
3          /* Building the small-grid */
4          K ← the key of p_i^j for the small-grid
5          if c_K^s exists in the small-grid then
6              Set the i-th bit of b(c_K^s) as 1
7              if |b(c_K^s)| becomes 2 (the i'-th bit is already 1) then
8                  o_i'.L ← o_i'.L ∪ {K}, o_i.L ← o_i.L ∪ {K}
9              if |b(c_K^s)| becomes more than 2 then
10                 o_i.L ← o_i.L ∪ {K}
11         else
12             Create c_K^s and set the i-th bit of b(c_K^s) as 1
13             Insert c_K^s into the small-grid
14         /* Building the large-grid */
15         K ← the key of p_i^j for the large-grid
16         if c_K^l exists in the large-grid then
17             Set the i-th bit of b(c_K^l) as 1
18             I(c_K^l)[o_i] ← I(c_K^l)[o_i] ∪ {p_i^j}
19         else
20             Create c_K^l and execute lines 17–18
21             Insert c_K^l into the large-grid
```

Building the large-grid. The operations for building the large-grid are essentially the same as those of the small-grid. The difference is to update inverted lists, which is described in lines 18 and 20, instead of maintaining key lists. Note that we do not create $\mathbf{b}^{adj}(c_K^l)$ in this step, to avoid a significant cell access cost. The bitset is created in the upper-bounding step.

It is important to note the following observations. (i) No empty cells: because we create cells when they are needed, empty cells are never generated. This is an important requirement for main-memory processing [6]. (ii) No replication: a point is mapped to only a single cell. (iii) Efficient building: each operation in Algorithm 3 takes a constant time. The time complexity of GRID-MAPPING(O, r) is thus $\mathcal{O}(nm)$.

Discussion about offline index building. One may consider an approach that builds BIGrid with a certain distance threshold r'. We show that this approach is not efficient in Appendix A. (It may be intuitively seen, because r is an arbitrary value and not known in advance, suggesting that finding good r' for any r is hard.)

B. Lower- and upper-bounding

We next focus on how to obtain a lower-bound and an upper-bound scores of a given object $o \in O$ with the BIGrid. Thanks to the BIGrid structure, it is easy to compute them.

Lower-bounding. We compute a lower-bound score for each object in O. Given $o_i \in O$, we exploit its key list to obtain a lower-bound of $\tau(o_i)$. Recall that, given a cell c_K^s of the small-grid, $|\mathbf{b}(c_K^s)|$ is the number of bits set as 1 in the compressed bitset of c_K^s. We utilize the following lemma for lower-bounding.

LEMMA 1 (LOWER-BOUND). *Given an object o_i and the BIGrid built by Algorithm 3, the following inequality holds.*

$$|\bigvee_{\forall K \in o_i.L} \mathbf{b}(c_K^s)| - 1 \leq \tau(o_i) \tag{2}$$

Algorithm 4: LOWER-BOUNDING(O, r)

```
1  τ_max^low ← 0
2  for ∀o_i ∈ O do
3      b(o_i) ← ∅        ▷b(o_i) is a temporal compressed bitset
4      for ∀K ∈ o_i.L do
5          b(o_i) ← b(o_i) ∨ b(c_K^s)        ▷bitwise OR operation
6      τ^low(o_i) ← |b(o_i)| − 1
7      if τ^low(o_i) > τ_max^low then
8          τ_max^low ← τ^low(o_i)
9  return τ_max^low
```

PROOF. Recall that the width of a cell of the small-grid is $\frac{r}{\sqrt{3}}$. Given two 3-dimensional points p and p' which are mapped into the same cell of the small-grid, we can guarantee that $dist(p, p') \leq r$. Therefore, if the i-th and j-th bits of $\mathbf{b}(c_K^s)$ are 1, we have $o_j \in O_i$ (see Equation (1)), because they have at least one pair of points $p \in o_i$ and $p' \in o_j$ such that $dist(p, p') \leq r$. Now we see that $\bigvee_{\forall K \in o_i.L} \mathbf{b}(c_K^s)$ represents a union of sets of objects that certainly have point pairs with o_i such that $dist(p, p') \leq r$. Note that the i-th bit of $\mathbf{b}(c_K^s)$ is 1, so we need the subtraction "-1". □

Let $\tau^{low}(o_i)$ be $|\bigvee_{\forall K \in o_i.L} \mathbf{b}(c_K^s)| - 1$. We describe why $o_i.L$ is enough for computing $\tau^{low}(o_i)$. A straightforward approach to computing $\tau^{low}(o_i)$ is that for all $p_i^j \in o_i$, we access its corresponding cell of the small-grid and execute a bitwise OR operation. Unfortunately, this approach may access unnecessary cells. Assume that $|\mathbf{b}(c_K^s)| = 1$ and the i-th bit is 1. From Equation (2), the compressed bitset of this cell does not contribute to $\tau^{low}(o_i)$. Algorithm 3 avoids maintaining such cells (i.e., $o_i.L$ does not have such K).

LOWER-BOUNDING(O, r). In Algorithm 4, we describe the detail of the lower-bounding step. We first set $\tau_{\max}^{low}$ as 0. Next, for each $o_i \in O$, we create $\mathbf{b}(o_i)$ (a temporal compressed bitset) and compute its lower-bound by accessing all cells whose keys are maintained by $o_i.L$ and using a bitwise OR operation between $\mathbf{b}(o_i)$ and $\mathbf{b}(c_K^s)$. We then update $\tau_{\max}^{low}$ if $\tau_{\max}^{low} < \tau^{low}(o_i)$. At the end of this algorithm, we have $\tau_{\max}^{low} = \text{argmax}_O\, \tau^{low}(o_i)$.

Let $\mathsf{cost}(\mathbf{b},\mathbf{b'})$ be the cost of bitwise operation between two compressed bitsets $\mathbf{b}$ and $\mathbf{b'}$. The time complexity of computing $\tau^{low}(o_i)$ is $\mathcal{O}(\sum_{o_i.L} \mathsf{cost}(\mathbf{b}(o_i),\mathbf{b}(c_K^s)))$. Therefore, Algorithm 4 takes $\mathcal{O}(n \sum_{o_i.L} \mathsf{cost}(\mathbf{b}(o_i),\mathbf{b}(c_K^s)))$ time[6].

Upper-bounding. This step is very similar to the lower-bounding step. To obtain $\tau^{upp}(o_i)$, an upper-bound score of o_i, we utilize the following lemma.

LEMMA 2 (UPPER-BOUND). *Let $o_i.L'$ be the set of all keys of the points for the large-grid in o_i. Given an object o_i and the BIGrid built by Algorithm 3, the following inequality holds.*

$$\tau(o_i) \leq |\bigvee_{\forall K \in o_i.L'} \mathbf{b}^{adj}(c_K^l)| - 1$$

[6]In our implementation, $\mathsf{cost}(\mathbf{b},\mathbf{b'}) = \mathcal{O}(\mathsf{size}(\mathbf{b})+\mathsf{size}(\mathbf{b'}))$, where $\mathsf{size}(\mathbf{b})$ is the size of $\mathbf{b}$ in bytes [23]. Actually, this size can be regarded as a constant in practice for skewed datasets. In this case, we have $\mathcal{O}(n \sum_{o_i.L} \mathsf{cost}(\mathbf{b}(o_i),\mathbf{b}(c_K^s))) \approx \mathcal{O}(nm')$, where m' is the average size of $o_i.L$.

Algorithm 5: UPPER-BOUNDING$(O, r, \tau^{low}_{\max})$

```
1  O_cand ← ∅                          ▷ O_cand is a set of candidate objects for o*
2  K ← ∅                               ▷ K is a set of keys
3  for ∀o_i ∈ O do
4      b(o_i) ← ∅                      ▷ b(o_i) is a temporal compressed bitset
5      for ∀p_i^j ∈ o_i do
6          K ← the key of p_i^j for the large-grid
7          if K ∉ K then
8              K ← K ∪ {K}
9              Compute b^adj(c_K^l)                    ▷ see Definition 3
10         b(o_i) ← b(o_i) ∨ b^adj(c_K^l)              ▷ bitwise OR operation
11     τ^upp(o_i) ← |b(o_i)| − 1
12     if τ^upp(o_i) ≥ τ^low_max then
13         O_cand ← O_cand ∪ {o_i}
14 Sort O_cand in descending order of upper-bound
15 return O_cand
```

PROOF. Recall that $\mathbf{b}^{adj}(c^l_K) = \bigvee \mathbf{b}(c^l_{K'})$, where K' is K or the key of the adjacent cell of c^l_K. Assume that a point p^j_i is mapped to c^l_K. Since the width of each cell of the large-grid is $\lceil r \rceil$, points p, which *may* satisfy $dist(p^j_i, p) \leq r$, exist in c^l_K or its adjacent cells. The remaining discussion is essentially the same as the proof of Lemma 1. □

Now we have the following theorem.

THEOREM 2. *Consider an object o_i and $\tau^{low}_{\max}$ obtained by Algorithm 4. If $\tau^{upp}(o_i) < \tau^{low}_{\max}$, $o_i \neq o^*$.*

PROOF. From Lemmas 1 and 2. □

UPPER-BOUNDING$(O, r, \tau^{low}_{\max})$. We utilize Lemma 2 and Theorem 2 to compute $\tau^{upp}(o_i)$ and prune unnecessary score computations. Algorithm 5 outlines our upper-bounding. Let O_{cand} and $\mathcal{K}$ respectively be sets of candidate objects for o^* and keys, which are empty sets at first. Given $o_i \in O$, we create a temporal compressed bitset $\mathbf{b}(o_i)$ as well as lower-bounding. Then, for all $p^j_i \in o_i$, we compute its key K for the large-grid and confirm $K \in \mathcal{K}$ or not. If not, $\mathbf{b}^{adj}(c^l_K)$ has not been computed, so we compute it by accessing c^l_K and its adjacent cells and then insert K into $\mathcal{K}$. We update $\mathbf{b}(o_i)$ by a bitwise OR operation between it and $\mathbf{b}^{adj}(c^l_K)$, and obtain $\tau^{upp}(o_i)$ by Lemma 2. If $\tau^{upp}(o_i) \geq \tau^{low}_{\max}$, we insert o_i into O_{cand}, and otherwise it is pruned by Theorem 2. The above operations are repeated for all $o_i \in O$. Finally, we sort O_{cand} in descending order of upper-bound.

Let the number of cells of the large-grid be g. Computing $\mathbf{b}^{adj}(c^l_K)$ for each cell of the large-grid takes $\mathcal{O}(g \sum \mathsf{cost}(\mathbf{b}^{adj}(c^l_K), \mathbf{b}(c^l_{K'})))$ time. (The number of adjacent cells of each cell is bounded by a constant, e.g., 26 in a three-dimensional space.) Similar to lower-bounding, the time complexity of computing an upper-bound for all objects in O is $\mathcal{O}(n \sum_{o_i.L'} \mathsf{cost}(\mathbf{b}(o_i), \mathbf{b}^{adj}(c^l_K)))$. Therefore, the time complexity of Algorithm 5 is obtained by summing the two time complexities. (The sorting cost is dominated by them, and our experiments show that computing $\tau^{upp}(o_i)$ is faster than computing $\tau(o_i)$ by a factor at least one order of magnitude.)

C. Verification

In this step, we obtain the answer of the MIO query, o^*. To this end efficiently, we employ a best-first approach, due to the following corollary which is obtained from Lemma 2.

COROLLARY 1 (EARLY TERMINATION). *Consider O_{cand} as a queue. We dequeue the front object of O_{cand} and compute its exact score. Let o be the object whose score is the best among a set of objects that have been dequeued so far. Besides, let o' be the current front object. If $\tau(o) \geq \tau^{upp}(o')$, we have $o^* = o$.*

The BIGrid structure is effective not only for the bounding techniques but also for exact score computation. We show that it can avoid unnecessary cell accesses and pairwise distance computations.

VERIFICATION(O_{cand}, r). Let o be an intermediate result, and it is initialized at $\varnothing$ (so, $\tau(o)$ is 0 at first). In what follows, the detail of this step is described.

1) We dequeue the front object of O_{cand}, and let the object be o_i. If $\tau^{upp}(o_i) \leq \tau(o)$, it is guaranteed that $o = o^*$ by Corollary 1, thus we terminate query processing. Otherwise, we proceed to the next operation.
2) As with the bounding step, we create a temporal compressed bitset $\mathbf{b}(o_i)$ and set its i-th bit as 1. Given $p \in o_i$, we do the following. We compute its key K for the large-grid and access c^l_K. Let $\mathbf{b}$ be $\mathbf{b}^{adj}(c^l_K) - \mathbf{b}(o_i)$. If $|\mathbf{b}| = 0$, $\mathbf{b}(o_i)$ already contains all objects (by bit representation) some of whose points are mapped to c^l_K and its adjacent cells. (Recall that for a point p whose key for the large-grid is K, the points p', where $dist(p, p') \leq r$, exist at c^l_K or its adjacent cells.) In this case, we do not need to access the adjacent cells. If $|\mathbf{b}| > 0$, we have to access the adjacent cells $c^l_{K'}$ but can avoid unnecessary score computation by leveraging inverted lists. It is important to note that candidate objects that can be in O_i are represented by $\mathbf{b}$. Specifically, if the j-th bit of $\mathbf{b}$ is 1, o_j may be in O_i. We therefore access only posting lists $I(c^l_{K'})[o_j]$ such that the j-th bit of $\mathbf{b}$ is 1, to avoid unnecessary distance computation. If there is a point $p' \in I(c^l_{K'})[o_j]$ such that $dist(p, p') \leq r$, we set the j-th bit of $\mathbf{b}(o_i)$ ($\mathbf{b}$) as 1 (0). Again, if $|\mathbf{b}| = 0$, the adjacent cells of c^l_K have no more candidate objects, thereby we deal with the next point. This operation is executed for all $p \in o_i$.
3) Then we have $\tau(o_i) = |\mathbf{b}(o_i)| - 1$. If $\tau(o) < \tau(o_i)$, we replace the intermediate result and go back to the first operation.

Algorithm 6 summarizes this step.

Discussion about top-k variant. Although Definition 1 focuses on the object with the highest score, our solution can easily deal with its top-k variant. If applications require top-k objects with the highest score, we compute the k-th highest lower-bound instead of $\tau^{low}_{\max}$ in the lower-bounding step and use it in the upper-bounding step. In the verification step, after computing the scores of k objects, we set a threshold. Then Corollary 1 is applied accordingly.

D. Leveraging previous results

Algorithms 5 and 6 may incur some unnecessary computations. This is because there may exist some points in o that

Algorithm 6: VERIFICATION(O_{cand}, r)

```
1  o ← ∅                                   ▷ o is an intermediate result
2  while |O_cand| > 0 do
3      o_i ← the front object of O_cand    ▷ dequeue the front object
4      if τ^upp(o_i) ≤ τ(o) then
5          break
6      b(o_i) ← ∅                          ▷ b(o_i) is a temporal compressed bitset
7      Set the i-th bit of b(o_i) as 1
8      for ∀p ∈ o_i do
9          K ← the key of p for the large-grid
10         b ← b^adj(c^l_K) − b(o_i)
11         if |b| > 0 then
12             for ∀K' in the set of K and its adjacent cell keys do
13                 for ∀o_j ∈ I(c^l_K') s.t. the j-th bit of b is 1 do
14                     if ∃p' ∈ I(c^l_K')[o_j] s.t. dist(p, p') ≤ r then
15                         Set the j-th bit of b(o_i) (b) as 1 (0)
16                     if |b| = 0 then
17                         break
18                 Execute lines 16–17
19     τ(o_i) ← |b(o_i)| − 1
20     if τ(o_i) > τ(o) then
21         o ← o_i
22 o* ← o
23 return o*
```

do not contribute to computing $\tau^{upp}(o)$ and/or $\tau(o)$. If we know such points in advance, we can skip accessing them in the processing of a given MIO query. In other words, by providing each point p in O with a label that shows whether p is necessary for obtaining $\tau^{upp}(o)$ and/or $\tau(o)$ during the processing of a given MIO query, we can improve the performance of future MIO queries.

Rationale. To implement the above idea, we have to know when we face redundant computation. We describe such cases below.

OBSERVATION 1. *Consider that $|\mathbf{b}^{adj}(c^l_K)| = 1$ after executing line 9 of Algorithms 5. For this case, assume that only the i-th bit is 1 and p^j_i is mapped to c^l_K. This case means that, for all $o_{i'} \in O \backslash \{o_i\}$, $\nexists p \in o_{i'}$ such that $dist(p^j_i, p) \leq r$. That is, even if we ignore p^j_i, we can still obtain $\tau^{upp}(o_i)$.*

OBSERVATION 2. *Notice that two objects may have multiple pairs of points whose distances are within r. At line 10 of Algorithm 5, $\mathbf{b}(o_i)$ may not be varied, due to the above observation. In this case, executing this line is not necessary.*

OBSERVATION 3. *For Algorithm 6, consider line 10. If $|\mathbf{b}| = 0$, we can prune unnecessary cell accesses. However, it still incurs a hash look-up and a bitwise operation.*

If we can skip the above cases, the performances of Algorithms 5 and 6 are improved. Assume that we know points p with at least one of Observations 1–3 w.r.t. an MIO query with r. Assume further that we are given a new MIO query with r' where $\lceil r \rceil = \lceil r' \rceil$. Since, given O, the large-grid is the same for all r' such that $\lceil r \rceil = \lceil r' \rceil$, we can skip accessing p in upper-bounding and/or verification. As noted in Section I, analytical applications may issue queries by varying r, and r is often fine-grained [14], [19]. This is the reason why $\lceil r \rceil$ is employed as the width of the large-grid.

Now we propose a labeling approach, and define the label.

DEFINITION 4 (LABEL). *Consider an MIO query with r on O. The label of each point p in O, $\mathsf{label}(p)$, consists of three bits and is initialized at* 111. *Let c^l_K be the large-grid cell into which p is mapped.*

- Labeling-1: *If $|\mathbf{b}^{adj}(c^l_K)| = 1$, $\mathsf{label}(p) = 0**$, where "$*$" is* 0 *or* 1.
- Labeling-2: *If p encounters the case in Observation 2, $\mathsf{label}(p) = 10*$.*
- Labeling-3: *If p encounters the case in Observation 3, $\mathsf{label}(p) = 1*0$.*

When to provide labels. As can be seen from Definition 4, Labeling-1, Labeling-2, and Labeling-3 are executed after line 9 of Algorithm 5, line 10 of Algorithm 5, and line 10 of Algorithm 6, respectively. In addition, labels are outputted in post-processing. The space cost of the labels for MIO queries with $\lceil r \rceil$ is $\mathcal{O}(nm)$. Note that the number of MIO queries issued cannot be bounded. For practical use, therefore, labels should be resident in external memory. In this case, the I/O cost of loading labels for a given MIO query is $\mathcal{O}(\frac{nm}{B})$, where B is a block size[7].

How to leverage labels. We verify that our solution is friendly to use intermediate results of previous MIO queries. Given an MIO query with r, we first check whether there are labels for $\lceil r \rceil$. This can be done in $\mathcal{O}(1)$ time by using a hash table. If no, we simply execute Algorithms 3–6. Otherwise, we leverage the labels and elaborate how to do this below. Note that the access order of objects and points has to be the same to keep correctness.

GRID-MAPPING-WITH-LABEL(O, r). We first consider BI-Grid building with labels. The basic operation is the same, but we can ignore points p where $\mathsf{label}(p) = 0**$, due to the following lemma.

LEMMA 3 (POINT PRUNING). *Consider a point $p \in o_i$, where $\mathsf{label}(p) = 0**$. Even if we do not update the bitset of its corresponding small-grid cell and do not map p to its corresponding large-grid cell, $\tau^{low}(o_i)$, $\tau^{upp}(o_i)$, and $\tau(o_i)$ can be obtained exactly.*

PROOF. From Definitions 2, 3, and 4. □

LOWER-BOUNDING-WITH-LABEL(O, r). This step also does the same operation as with the original one. A difference is that we maintain $\mathbf{b}(o_i)$ to utilize this in the verification step.

UPPER-BOUNDING-WITH-LABEL$(O, r, \tau^{low}_{\max})$. In this step, we replace line 5 of Algorithm 5 with "$\forall p^j_i \in o_i$ such that $\mathsf{label}(p) = 11*$."

VERIFICATION-WITH-LABEL(O_{cand}, r). This step has three differences from the original one. First, $\mathbf{b}(o_i)$ is initialized at the compressed bitset obtained in LOWER-BOUNDING-WITH-LABEL(O, r). Because we do not need to do labeling, this initialization does not lose correctness. Furthermore, we can prune more cells, because we tend to have $|\mathbf{b}| = 0$ at line 11 in an early iteration. Second, we replace line 8 of Algorithm 6

[7]Our experiments demonstrate that this I/O cost is dominated by the cost of query processing.

with "$\forall p \in o_i$ such that $\mathsf{label}(p) = 1 * 1$." Last, it is important to note that after the upper-bounding step, cells c^l_K may not obtain $\mathbf{b}^{adj}(c^l_K)$, because we skip some points by labeling in the upper-bounding step. At line 10 of Algorithm 6, therefore, if $\mathbf{b}^{adj}(c^l_K)$ has not been computed, we compute it first.

IV. Parallel MIO query processing

We shift our interest to multi-core processing. We parallelize Algorithms 3–6, and discuss how to achieve load balancing.

Parallel-Grid-Mapping(O, r). There are two loops in Grid-Mapping(O, r): lines 1 and 2. Which line should we parallelize? Our answer is to do line 2. The amortized time for mapping all points of $o_i \in O$ is $\mathcal{O}(m)$, and each operation in this loop is independent of the other points of o_i. A hash-partitioning approach therefore enables $\mathcal{O}(\frac{m}{t})$ time for mapping them, where t is the number of available CPU cores. On the other hand, parallelizing line 1 faces an NP-complete problem. Each object may have a different sized point set. For load balancing, we have to assign a subset of objects to each core so that all cores have (almost) the same number of points. We prove that this is hard.

Theorem 3. *Parallelizing line 1 of Algorithm 3 so that the load difference between CPU cores is minimized is NP-complete.*

Proof. To optimally assign objects in O to each core, we have to solve the following problem. Consider a set of integers $\{|P_1|, |P_2|, ..., |P_n|\}$. When we divide it into t disjoint subsets, the sum of the integers in each subset, denoted by $\sum(t_i)$ where t_i is the i-th core, is obtained. The problem is to minimize $\max_t \sum(t_i) - \min_t \sum(t_i)$. This problem is called multi-way number partitioning, which is NP-complete [24]. □

Theorem 3 suggests that parallelizing line 1 cannot provide $\mathcal{O}(\frac{nm}{t})$ time.

Parallel-Lower-bounding(O, r). This lower-bounding also has two options: parallelizing lines 2 and 4 of Algorithm 4. Theorem 3 implies that the first option has to tackle an NP-complete problem. However, the second option is not a perfect solution, and they have both advantages and disadvantages.

• **Dividing O for parallel lower-bounding.** Since optimally dividing O is not practical, we employ a heuristic algorithm. Let T_i be a subset of O, which is assigned to the i-th core t_i and initialized at $\varnothing$. We incrementally update T_i by a greedy approach. Given $o \in O$, we obtain T = argmin $\sum_{T_i} |o_j.L|$ and insert o into T. This is done for all $o \in O$. Note that T can be obtained in $\mathcal{O}(1)$ time by parallel processing, so this algorithm takes $\mathcal{O}(n)$ time. After partitioning, each core runs Algorithm 4.

• **Dividing P_i for parallel lower-bounding.** The second approach is to divide P_i by a hash-partitioning. However, simple hash-partitioning suffers from synchronization at updating $\mathbf{b}(o_i)$ (see line 5). This degrades the performance of parallel processing. To avoid this, we assign a local compressed bitset to each core, and each core updates this bitset in the loop. The final result is obtained by merging the local bitsets.

The disadvantage of the first approach is that optimal load balancing is hard to obtain, although the greedy approach provides a good partitioning in practice. The advantage is that no synchronization is required to update the bitsets. On the other hand, the advantage of the second approach is the optimal load balancing. The disadvantage is that if $|o_i.L|$ is small, parallel processing does not give much gain and the merging can be bottleneck. Our empirical study verifies that the first approach is better than the second one.

Parallel-Upper-bounding$(O, r, \tau^{low}_{\max})$. In this step, we parallelize line 5 of Algorithm 5. Recall that if $\mathbf{b}^{adj}(c^l_K)$ has not been computed, we have to compute it at first. That is, the cost of dealing with $p \in o_i$ becomes different if p encounters the above case. To minimize the cost difference between each core, we propose a cost-based heuristic partitioning for parallel upper-bounding.

Assume that points in o_i are grouped by their keys, which can be done in building the large-grid. Let $P_{i,K}$ be the set of points of o_i whose keys are K, and we have $P_{i,K} \cap P_{i,K'} = \varnothing$. The key-based point grouping provides a way to design a cost of a point set, and we assign $P_{i,K}$ to a core based on the following cost model. Assuming that the cost of updating a compressed bitset, $\mathsf{cost}(\mathbf{b})$, is a constant, the cost of $P_{i,K}$, $C(P_{i,K})$, is formalized as follows.

$$C(P_{i,K}) = \begin{cases} \mathsf{cost}(\mathbf{b}) + |P_{i,K}| & (\text{if } |\mathbf{b}(c^l_K)| = 0) \\ 27 \cdot \mathsf{cost}(\mathbf{b}) + |P_{i,K}| & (\text{otherwise}) \end{cases} \quad (3)$$

Note that $|P_{i,K}|$ corresponds to the labeling cost for each point in $P_{i,K}$. Using the above cost model, the same greedy partitioning approach as in the parallel lower-bounding, and local bitsets, we assign $P_{i,K}$ to the core with the minimum cumulative cost.

In this approach, only a single core independently computes $\mathbf{b}^{adj}(c^l_K)$ and the other cores do not access the bitset, since points with the same keys are assigned to a single core. This is an advantage of multi-core processing, since we do not need any synchronization for the computation. When the labels are utilized in this step, $|P_{i,K}|$ in Equation (3) is omitted, since we do not label points.

Parallel-Verification(O_{cand}, r). In this step, we parallelize line 8 of Algorithm 6, and propose a heuristic partitioning approach. (After partitioning, each core runs the loop of line 8.) Recall that we use the intermediate result $\mathbf{b}(o_i)$ for cell pruning. Due to this pruning, this step may be the most difficult to be parallelized with load balancing. Our approaches take into account this observation.

• **Without label case.** We use a local compressed bitset for each core to avoid synchronization. (The bitsets are merged after the loop of line 8.) That is, each core prunes unnecessary cells by using the local bitset. To achieve load balancing, the pruning rate and the number of distance computations of each core should be similar. Because they are unknown in advance, we propose a heuristic partition. The idea of this partition is to assign points with different keys of the large-grid uniformly to each core. Specifically, for each $P_{i,K}$, we partition it into

t disjoint subsets, each of which is assigned to a core. (If $|P_{i,K}| < t$, each point is assigned to the cell with the minimum sized point set.)

• **With label case.** Given $p_i^j \in o_i$, whose key for the large-grid is K, $\mathbf{b}^{adj}(c_K^l)$ might not be computed. Given o_i, we first compute the bitsets of such cells, which can be done in parallel. Let $\mathbf{b}^{low}(o_i)$ be the compressed bitset obtained in the lower-bounding step. We next prune unnecessary cells by $\mathbf{b}^{low}(o_i)$, which is also done in parallel. During this, we store the keys K of cells that have not been pruned in $\mathcal{K}$. For all $P_{i,K}$ such that $K \in \mathcal{K}$, we do the same partitioning as the without label case.

V. Empirical study

A. Setting

All experiments were conducted on a machine with a 12-core Intel Xeon E5-2687W v4 processor (3.0GHz), 512GB RAM and 1TB SSD with 4096B block size. (This external memory is used only for labels, and a given dataset is memory-resident, as mentioned in Section II-A). For multi-threading, we used OpenMP. All evaluated algorithms were implemented in C++. In the experiments, we measured query processing time and memory usage. Note that we terminated the experiments if query processing did not terminate within 8 hours.

Evaluated algorithms. In the experiments, we evaluated the following algorithms.

- BIGrid: the algorithm proposed in Sections III-A–III-C.
- BIGrid-label[8]: the algorithm proposed in Section III-D.
- NL: the algorithm introduced in Section II-B[9].
- SG: a simple grid algorithm. Given a distance threshold r, SG first builds a spatial grid (the width of each cell is r) while mapping each point into the corresponding cell. Then, for each object $o \in O$, SG computes $\tau(o)$ by utilizing the grid, similarly to BIGrid. SG corresponds to a state-of-the-art main memory spatial-join algorithm TOUCH [5] (but is optimized for our problem). This algorithm also builds an index (a hierarchical tree) online and then compares points that can be within a distance. Since SG computes distances between points if they can be within r, such a hierarchical index is not necessary and SG incurs less index access overheads.

We do not compare our solutions with the theoretical algorithm introduced in Section II-B, because its pre-processing spent more than 8 hours.

Datasets. We used four real datasets, *Neuron*, *Neuron-2*, *Bird*, and *Bird-2*, that have large numbers of points (i.e., nm)[10].

[8]To function BIGrid-label, we made BIGrid output the labels of points for each parameter setting.

[9]We also tested a variant of NL. This variant maintains the points of an object by a kd-tree, so the time complexity of this algorithm is $\mathcal{O}(n^2 m \log m)$. However, this variant shows a similar performance to those of NL and cannot beat our solutions. We therefore omit the result of this variant for conciseness.

[10]It is important to note that datasets with large nm are suitable for our experiments, because the complexities of the evaluated algorithms are derived from the number of *points* in O.

TABLE I
Dataset statistics

Dataset	n	m	nm	Unit of r
Neuron	776	7,960	6,176,960	Micrometer
Neuron-2	5,493	848	4,657,696	Micrometer
Bird	143,042	50	7,152,100	Meter
Bird-2	29,247	100	2,924,700	Meter
Syn	851,519	52	44,266,671	-

Neuron and Neuron-2 are sets of rat neuron objects [4]. Bird and Bird-2 are sets of bird trajectories [11]. We generated these datasets by dividing long trajectories so that each trajectory contains approximately m points [14]. In addition, we generated a synthetic dataset, *Syn*, so that its score distribution follows a power law, based on a human-brain network [25]. The statistics of these datasets are shown in TABLE I.

B. Result: Single core processing

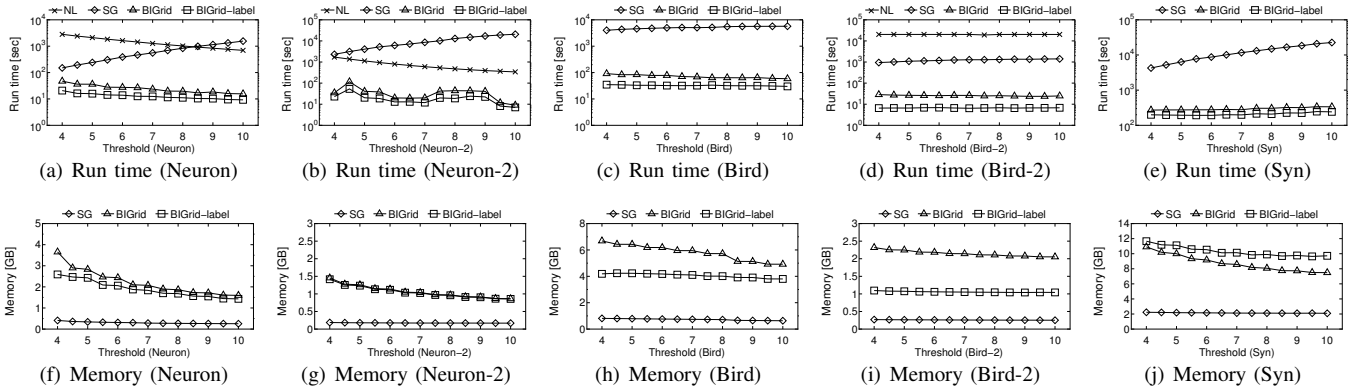
Varying r. We study the impact of the distance threshold r, and varied r from 4 to 10 based on [7]. Fig. 5 shows the experimental results.

From Figs. 5(a)–5(d), we see that NL and SG have opposite results. That is, NL decreases its processing time as r increases, while SG increases its processing time as r increases. The reason is as follows. As r increases, two objects tend to have many pairs of points whose distances are within r. Therefore, NL can find such pairs in an early iteration, thereby reduces the number of pairwise distance computations. On the other hand, w.r.t. SG, as r increases, the number of points in a cell tends to increase, so SG needs more pairwise distance computations for a point.

The second observation is that BIGrid clearly outperforms NL and SG. When $r = 4$, for the datasets of Neuron, Neuron-2, and Bird-2, BIGrid is 60 (3), 49 (67), and 711 (33) times faster, respectively, than NL (SG). Also, for Bird and Syn, BIGrid is 44 and 16 times faster, respectively, than SG, when $r = 4$. Furthermore, BIGrid-label successfully accelerates the query processing efficiency. To better understand, we show run time of each operation of BIGrid and BIGrid-label in TABLE II. We see that loading labels is not an overhead for BIGrid-label. In addition, Grid-Mapping-with-Label$(\cdot,\cdot)$, Lower-bounding-with-Label$(\cdot,\cdot)$, Upper-bounding-with-Label$(\cdot,\cdot,\cdot)$, and Verification-with-Label$(\cdot,\cdot,\cdot)$ incur similar or (significantly) less time compared with the corresponding ones in BIGrid. Lower- and upper-bounding incur significantly less costs than computing the exact score of an object. This can be seen by comparing the lower- and upper-bounding times with the run time of SG.

The third observation is that, in Neuron-2, BIGrid does not provide a consistent result. The upper-bound and exact score distributions are affected by r. For some r, there are many objects with higher upper-bound scores than $\tau(o^*)$, which results in a longer run time.

Figs. 5(f)–5(j) show the memory usage of SG, BIGrid, and BIGrid-label. (The memory cost of NL is dominated by the memory usage for a given dataset, so is omitted.) As r increases, their memory usages decrease, because their grids



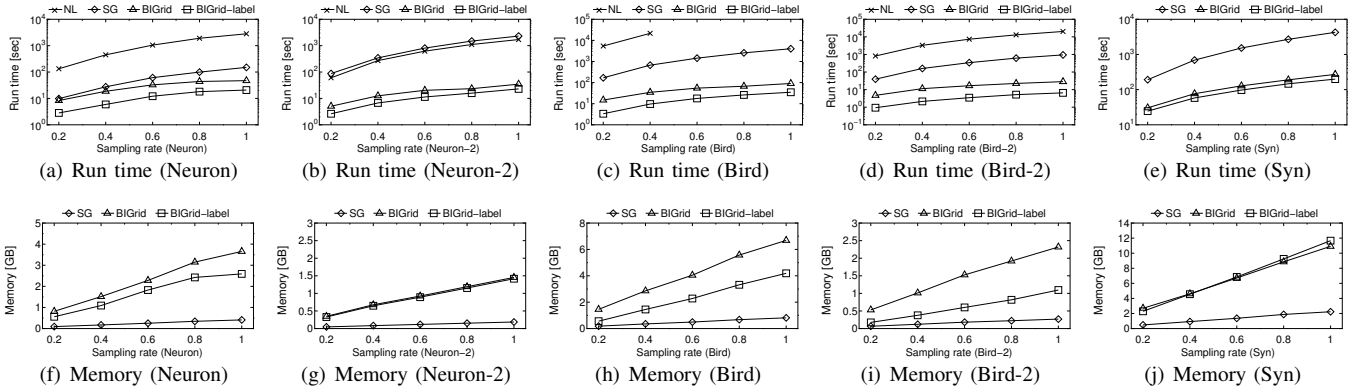
(a) Run time (Neuron) (b) Run time (Neuron-2) (c) Run time (Bird) (d) Run time (Bird-2) (e) Run time (Syn)

(f) Memory (Neuron) (g) Memory (Neuron-2) (h) Memory (Bird) (i) Memory (Bird-2) (j) Memory (Syn)

Fig. 5. Impact of threshold (r)

TABLE II
DECOMPOSED TIME [SEC] WHEN $r = 4$

Dataset	Neuron		Neuron-2		Bird		Bird-2		Syn	
Algorithm	BIGrid	BIGrid-label	BIGrid	BIGrid-label	BIGrid	BIGrid-label	BIGrid	BIGrid-label	BIGrid	BIGrid-label
Label-Input	-	0.401	-	0.325	-	0.775	-	0.240	-	4.959
Grid-Mapping	17.122	15.452	8.164	8.400	25.403	15.810	8.882	3.819	59.927	59.059
Lower-bounding	2.677	2.895	5.836	5.911	1.696	1.656	0.898	0.842	44.036	41.960
Upper-bounding	25.061	1.286	6.891	1.031	62.800	15.122	18.713	1.580	156.697	90.181
Verification	2.291	0.566	13.829	7.123	0.804	0.614	0.022	0.005	10.792	2.958



(a) Run time (Neuron) (b) Run time (Neuron-2) (c) Run time (Bird) (d) Run time (Bird-2) (e) Run time (Syn)

(f) Memory (Neuron) (g) Memory (Neuron-2) (h) Memory (Bird) (i) Memory (Bird-2) (j) Memory (Syn)

Fig. 6. Impact of sampling rate (s)

have less cells. Although the memory usage of BIGrid is larger than that of SG, it is affordable for recent main-memory systems. Besides, since BIGrid-label prunes points p where $\mathsf{label}(p) = 0**$, its memory usage is less than that of BIGrid. Fig. 5(j) shows an exception. Because Syn has less points p such that $\mathsf{label}(p) = 0**$, the memory cost of the labels becomes a little bit overhead.

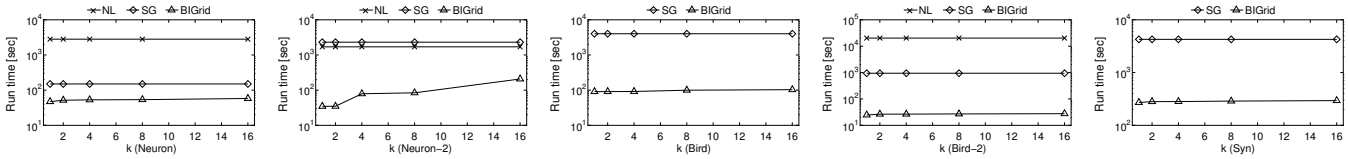
Scalability test. We next investigate the scalability of NL, SG, BIGrid, and BIGrid-label. For each dataset, we select $s \times n$ objects, where s is a sampling rate. Fig. 6 shows the experimental results. We see from Figs. 6(a)–6(e) that NL and SG do not scale well, and the computational costs of BIGrid and BIGrid-label are (much) better than those of NL and SG. This is because the costs of GRID-MAPPING$(\cdot,\cdot)$, LOWER-BOUNDING$(\cdot,\cdot)$, and UPPER-BOUNDING$(\cdot,\cdot,\cdot)$ are almost linear and the pruning rate of score computation is high. Figs. 6(f)–6(j) show that the memory costs of BIGrid and BIGrid-label have linear scalability.

Varying k. In this section, we finally investigate the run time of BIGrid for its top-k variant problem, which is shown in Fig. 7. Recall that NL and SG compute the scores of all objects, and their performances are independent on k. Figs. 7(a)–7(e) verify that our solution is still efficient for the top-k case. As k increases, its threshold becomes smaller. The run time therefore increases as k increases, but BIGrid effectively prunes unnecessary score computations.

C. Result: Multi-core processing

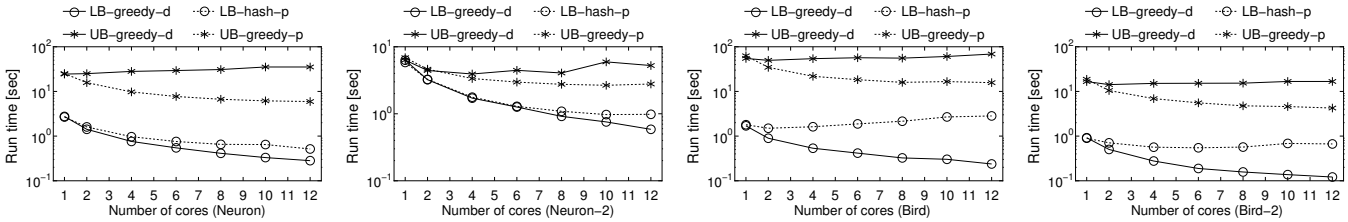
Evaluation of partitioning approaches. We first evaluate the efficiency of partitioning approaches for parallel lower- and upper-bounding by using the real datasets. We use LB-greedy-d and LB-hash-p to denote parallel lower-bounding by dividing O and P_i, respectively. Also, we use UB-greedy-p to denote parallel upper-bounding by the cost-based greedy P_i partition. As a competitor for parallel upper-bounding, we employ an approach that greedily partitions O based on $|P_i|$, which is denoted by UB-greedy-d.

Fig. 8 shows the results of the experiments with varying the number of available CPU cores. We see that LB-greedy-d and UB-greedy-p scale well w.r.t. the number of cores, demonstrating the effectiveness of our cost-based partitioning.



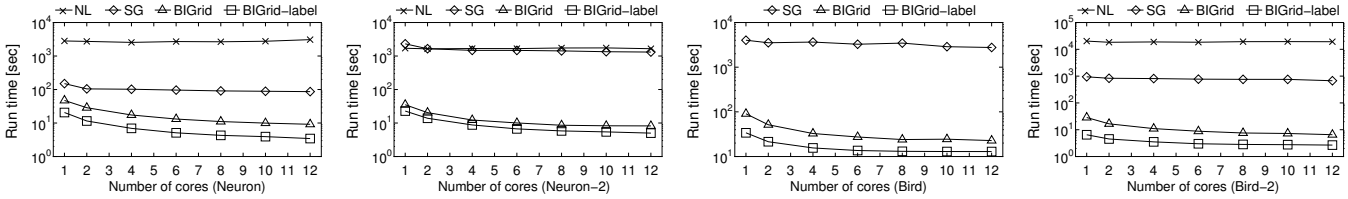
(a) Run time (Neuron) (b) Run time (Neuron-2) (c) Run time (Bird) (d) Run time (Bird-2) (e) Run time (Syn)

Fig. 7. Impact of result size (k)



(a) Bounding time (Neuron) (b) Bounding time (Neuron-2) (c) Bounding time (Bird) (d) Bounding time (Bird-2)

Fig. 8. Evaluation of partitioning approaches



(a) Run time (Neuron) (b) Run time (Neuron-2) (c) Run time (Bird) (d) Run time (Bird-2)

Fig. 9. Impact of number of cores (t)

Meanwhile, the other approaches do not exploit the available cores. LB-greedy-d works well in Neuron and Neuron-2 but does not in the other datasets. Because the size of a key list, $|o_i.L|$, is small in the cases of Bird and Bird-2, LB-greedy-d loses its advantage and the local bitset merging becomes overhead. UB-greedy-d consistently shows poor performance, because its partitioning approach does not consider the real cost of dealing with a point.

Comparison with parallel NL and SG. We parallelize NL and SG, to compare BIGrid with them. Fig. 9 shows the experimental results on the real datasets.

Although NL parallelizes line 3 of Algorithm 1, NL is not only computationally inefficient but also hard to achieve load balancing. Algorithm 1 shows that there are three loops to obtain $\tau(o_i)$, but the cost of each loop is hard to estimate, since the number of distance computations is unknown. Therefore, NL cannot achieve much time reduction. SG has a similar result. SG computes $\tau(o_i)$ in parallel by hash-partitioning, but its run time does not decrease so much. In general, hash-partitioning functions only when each task has the same cost. Due to the skewed distributions, the number of distance computations for a point is usually different. SG therefore does not achieve load balancing. On the other hand, BIGrid and BIGrid-label reduce their run times with increasing of the number of cores. This result demonstrates that, as well as parallel lower- and upper-bounding, PARALLEL-GRID-MAPPING$(\cdot,\cdot)$ and PARALLEL-VERIFICATION$(\cdot,\cdot)$ exploit the available cores. TABLE III, which shows the speedup ratio against single core version on Neuron and Bird, also confirms this result. We can observe that, although the speedup depends on datasets, which is derived from that fact that the compression statuses of bitsets depend on the data distributions, larger number of cores provides a shorter run time.

TABLE III
SPEEDUP RATIO AGAINST SINGLE CORE VERSION WHEN $r = 4$

Dataset	Neuron		Bird	
Algorithm	BIGrid	BIGrid-label	BIGrid	BIGrid-label
$t = 2$	1.648	1.789	1.763	1.584
$t = 4$	2.717	2.968	2.770	2.181
$t = 6$	3.585	4.036	3.313	2.492
$t = 8$	4.265	4.809	3.513	2.598
$t = 10$	4.750	5.251	3.708	2.626
$t = 12$	5.154	5.985	3.984	2.629

VI. RELATED WORK

Spatial query processing is used in many applications, such as the location selection problem [26] and data mining [27]. In this section, we review some spatial data processing techniques that are related to our solutions.

Spatial query processing. Given a set of points and a distance threshold, the problem of in-memory spatial join (self-join case) is to retrieve all pairs of points whose distances are within the threshold [28]. The time complexity of this problem is essentially quadratic, thus to reduce execution time, many approaches, which prune unnecessary distance computations between points, have been proposed. Examples are a hierarchical grid-based algorithm [29], TOUCH [5], and a space-partitioning algorithm [30], to name a few. [31] demonstrated that TOUCH is effective for real and skewed datasets among them. Our experiments have verified that spatial self-join based approaches, like NL and SG, is not efficient for our problem.

The closest point pair computation problem retrieves the pair of two points whose distance is the shortest among

all pairs in a given point set [32]. In Section II-B, we discussed how to employ the technique of the closest point pair computation and showed that it is not practical.

Hybrid index for spatial data. Recently, spatial objects have often been associated with some attributes, e.g., keywords [33] and social relationships [34]. To retrieve such objects, spatio-textual and geo-social queries have been devised, along with efficient retrieval algorithms. One representative index for spatio-textual queries is IR-tree [35], which is an R-tree with an inverted list for each node. The BIGrid structure is actually inspired by IR-tree and its variants, but has a clear difference. IR-tree aims at pruning objects that cannot be the top-k answer in a batch w.r.t. a given query point with keywords. This query considers that an object is a point, thereby MBR-based hierarchical indexing functions. On the other hand, the BIGrid is designed so that we can efficiently compute lower-bound, upper-bound, and the exact scores of objects.

Leveraging previous query results. If applications require quick data-to-insight time, building an index by costly pre-processing is not tolerant. In this case, leaving some query results for future queries is promising. Database cracking [36], which incrementally updates indices as a side-effect of query processing, has been proposed to achieve this. Although database cracking techniques were focused on relational databases [37], recent studies have proposed cracking techniques for in-memory and external-memory spatial datasets [21], [38]. Their techniques deal with spatial range queries, and it is not trivial to apply them to MIO queries.

VII. Conclusion

This paper addressed the problem of identifying the most interactive object in spatial datasets. To the best of our knowledge, this paper is the first work of this problem. We first showed that a nested-loop and a theoretical algorithms do not scale. Motivated by this fact, we proposed an efficient solution with a novel data structure BIGrid. The BIGrid structure efficiently facilitates lower-bounding, upper-bounding, and verification. Our solution is carefully designed so that it can use intermediate results of previous MIO queries, deal with the top-k variant, and be parallelized. Finally, our experiments demonstrate that our solution is significantly faster than competitors.

There may exist applications that want to find the most interactive object in a high-dimensional space, although it is beyond the scope of this paper. For such applications, our solution can be still used but may not function well, as it is well-known that grid structures are not effective for high-dimensional data. We leave designing a robust index for high-dimensional spaces for future work.

Acknowledgment. This research is partially supported by JSPS Grant-in-Aid for Scientific Research (A) Grant Number 18H04095, JST CREST Grant Number J181401085, and ACT-I, JST. We really thank Dr. Ken-ichi Kawarabayashi for his great comments.

References

[1] J. van Pelt and A. van Ooyen, "Estimating neuronal connectivity from axonal and dendritic density fields," *Frontiers in computational neuroscience*, vol. 7, pp. 160:1–160:19, 2013.
[2] H. Fan, H. Su, and L. J. Guibas, "A point set generation network for 3d object reconstruction from a single image," in *CVPR*, vol. 2, no. 4, 2017, pp. 605–613.
[3] Q. Fan, D. Zhang, H. Wu, and K.-L. Tan, "A general and parallel platform for mining co-movement patterns over large-scale trajectories," *PVLDB*, vol. 10, no. 4, pp. 313–324, 2016.
[4] http://neuromorpho.org/index.jsp.
[5] S. Nobari, F. Tauheed, T. Heinis, P. Karras, S. Bressan, and A. Ailamaki, "Touch: in-memory spatial join by hierarchical data-oriented partitioning," in *SIGMOD*, 2013, pp. 701–712.
[6] F. Tauheed, T. Heinis, and A. Ailamaki, "Thermal-join: A scalable spatial join for dynamic workloads," in *SIGMOD*, 2015, pp. 939–950.
[7] A. van Ooyen, A. Carnell, S. de Ridder, B. Tarigan, H. D. Mansvelder, F. Bijma, M. de Gunst, and J. van Pelt, "Independently outgrowing neurons and geometry-based synapse formation produce networks with realistic synaptic connectivity," *PloS one*, vol. 9, no. 1, p. e85858, 2014.
[8] R. Badhwar and G. Bagler, "A distance constrained synaptic plasticity model of c. elegans neuronal network," *Physica A: Statistical Mechanics and its Applications*, vol. 469, pp. 313–322, 2017.
[9] S. Nigam, M. Shimono, S. Ito, F.-C. Yeh, N. Timme, M. Myroshnychenko, C. C. Lapish, Z. Tosi, P. Hottowy, W. C. Smith *et al.*, "Rich-club organization in effective connectivity among cortical neurons," *Journal of Neuroscience*, vol. 36, no. 3, pp. 670–684, 2016.
[10] P. Bonifazi, M. Goldin, M. A. Picardo, I. Jorquera, A. Cattani, G. Bianconi, A. Represa, Y. Ben-Ari, and R. Cossart, "Gabaergic hub neurons orchestrate synchrony in developing hippocampal networks," *Science*, vol. 326, no. 5958, pp. 1419–1424, 2009.
[11] https://www.datarepository.movebank.org/.
[12] Z. Li, J. Han, M. Ji, L.-A. Tang, Y. Yu, B. Ding, J.-G. Lee, and R. Kays, "Movemine: Mining moving object data for discovery of animal movement patterns," *TIST*, vol. 2, no. 4, pp. 37:1–37:32, 2011.
[13] J.-G. Lee, J. Han, and K.-Y. Whang, "Trajectory clustering: a partition-and-group framework," in *SIGMOD*, 2007, pp. 593–604.
[14] Z. Shang, G. Li, and Z. Bao, "Dita: Distributed in-memory trajectory analytics," in *SIGMOD*, 2018, pp. 725–740.
[15] X. Ding, L. Chen, Y. Gao, C. S. Jensen, and H. Bao, "Ultraman: a unified platform for big trajectory data management and analytics," *PVLDB*, vol. 11, no. 7, pp. 787–799, 2018.
[16] P. Laube, M. van Kreveld, and S. Imfeld, "Finding remo - detecting relative motion patterns in geospatial lifelines," in *Developments in spatial data handling*, 2005, pp. 201–215.
[17] M. Andersson, J. Gudmundsson, P. Laube, and T. Wolle, "Reporting leaders and followers among trajectories of moving point objects," *GeoInformatica*, vol. 12, no. 4, pp. 497–528, 2008.
[18] N. Mavridis, N. Bellotto, K. Iliopoulos, and N. Van de Weghe, "Qtc3d: Extending the qualitative trajectory calculus to three dimensions," *Information Sciences*, vol. 322, pp. 20–30, 2015.
[19] M. W. Reimann, J. G. King, E. B. Muller, S. Ramaswamy, and H. Markram, "An algorithm to predict the connectome of neural microcircuits," *Frontiers in computational neuroscience*, vol. 9, no. 120, pp. 1–18, 2015.
[20] P. M. Vaidya, "An $O(n \log n)$ algorithm for the all-nearest-neighbors problem," *Discrete & Computational Geometry*, vol. 4, no. 2, pp. 101–115, 1989.
[21] M. Pavlovic, D. Sidlauskas, T. Heinis, and A. Ailamaki, "Quasii: Query-aware spatial incremental index," in *EDBT*, 2018, pp. 325–336.
[22] D. Lemire, O. Kaser, and K. Aouiche, "Sorting improves word-aligned bitmap indexes," *Data & Knowledge Engineering*, vol. 69, no. 1, pp. 3–28, 2010.
[23] O. Kaser and D. Lemire, "Compressed bitmap indexes: beyond unions and intersections," *Software: Practice and Experience*, vol. 46, no. 2, pp. 167–198, 2016.
[24] R. E. Korf, "Multi-way number partitioning." in *IJCAI*, 2009, pp. 538–543.
[25] http://networkrepository.com/bn.php.
[26] D. Amagata and T. Hara, "Monitoring maxrs in spatial data streams." in *EDBT*, 2016, pp. 317–328.
[27] D.-W. Choi and C.-W. Chung, "Nearest neighborhood search in spatial databases," in *ICDE*, 2015, pp. 699–710.

[28] B. Sowell, M. V. Salles, T. Cao, A. Demers, and J. Gehrke, "An experimental analysis of iterated spatial joins in main memory," *PVLDB*, vol. 6, no. 14, pp. 1882–1893, 2013.
[29] N. Koudas and K. C. Sevcik, "Size separation spatial join," in *SIGMOD Record*, vol. 26, no. 2, 1997, pp. 324–335.
[30] J. M. Patel and D. J. DeWitt, "Partition based spatial-merge join," in *SIGMOD Record*, vol. 25, no. 2, 1996, pp. 259–270.
[31] S. Nobari, Q. Qu, and C. S. Jensen, "In-memory spatial join: The data matters!" in *EDBT*, 2017, pp. 462–465.
[32] M. I. Shamos and D. Hoey, "Closest-point problems," in *FOCS*, 1975, pp. 151–162.
[33] L. Chen, G. Cong, C. S. Jensen, and D. Wu, "Spatial keyword query processing: an experimental evaluation," in *PVLDB*, vol. 6, no. 3, 2013, pp. 217–228.
[34] K. Mouratidis, J. Li, Y. Tang, and N. Mamoulis, "Joint search by social and spatial proximity," *TKDE*, vol. 27, no. 3, pp. 781–793, 2015.
[35] G. Cong, C. S. Jensen, and D. Wu, "Efficient retrieval of the top-k most relevant spatial web objects," *PVLDB*, vol. 2, no. 1, pp. 337–348, 2009.
[36] S. Idreos, M. L. Kersten, S. Manegold *et al.*, "Database cracking." in *CIDR*, vol. 7, 2007, pp. 68–78.
[37] F. M. Schuhknecht, A. Jindal, and J. Dittrich, "An experimental evaluation and analysis of database cracking," *The VLDB Journal*, vol. 25, no. 1, pp. 27–52, 2016.
[38] M. Pavlovic, E. T. Zacharatou, D. Sidlauskas, T. Heinis, and A. Ailamaki, "Space odyssey: efficient exploration of scientific data," in *Exploratory Search in Databases and the Web*, 2016, pp. 12–18.

APPENDIX

A. Analysis of offline BIGrid building

We demonstrate that offline BIGrid building does not provide any advantages over online approach.

First, we assume that BIGrid is build based on a certain distance threshold r'. Given an MIO query with r, lower- and upper-bounding become ineffective or incorrect. (i) If $r > r'$, lower-bonding does not lose correctness (but lower-bound scores become loose). However, upper-bounding loses correctness if we do not provide any extension. Recall that, in the original algorithm, for a point p_i^j with key K for the large-grid, we access c_K^l and its neighbor cells (say $c_{K'}^l$). To obtain upper-bound score (i.e., to guarantee correctness) w.r.t. an MIO query with $r > r'$, in addition to these cells, we have to access at least their ($c_{K'}^l$'s) neighbor cells. It is important to note that this approach increases the number of access cells exponetially. (ii) If $r < r'$, we see that small-grid does not provide correct lower-bound scores, because points p and p' in the same cell do not satisfy $dist(p, p') \leq r$.

Second, we can re-construct BIGrid (decompose/merge grid cells) if we have $r = \alpha \cdot r'$ where $\alpha \neq 0$ (i.e., very limited cases). Recall that we need to assign cell key for each point in a given object set O and need to update compressed bitsets if cells are updated. Therefore, decomposing/merging cells incurs the same cost as that of the original approach. We see that this approach does not provide any gain.

B. Incorporating a temporal dimension

If we consider a temporal dimension, we have to define an object o_i as $o_i = \{\langle p_i^1, t_i^1\rangle, ..., \langle p_i^{|P_i|}, t_i^{|P_i|}\rangle\}$, where t_i^j is the time when p_i^j is generated. Then, O_i (see Equation (1)) is defined as

$$O_i = \{o_j \mid o_j \in O \backslash \{o_i\}, \exists\langle p_i^{i'}, t_i^{i'}\rangle \in o_i, \exists\langle p_j^{j'}, t_j^{j'}\rangle \in o_j, dist(p_i, p_j) \leq r, |t_i^{i'} - t_j^{j'}| \leq \delta\},$$

where δ is a threshold for the temporal dimension. Without loss of generality, let us assume that the temporal dimension of a given dataset has a time domain $[0, T]$. Given an MIO query with r and δ, we first decompose $[0, T]$ into disjoint sub-domains, $[0, \delta)$, $[\delta, 2\delta)$, ..., $[T - \delta, T]$. Then, for each sub-domain, we build BIGrid based on the generation time of points.

For lower-bounding, assume that we have $\langle p_i^j, t_i^j\rangle \in o_i$, where $t_i^j = \delta$, and the key for small-grid of p_i^j is K. Now the small-grid on the sub-domain $[\delta, 2\delta)$ has a compressed bitset $\mathbf{b}(c_K^s)$ whose i-th bit is 1. We see that, to obtain points p certainly satisfying $dist(p_i^j, p) \leq r$ and $|t_i^j - t| \leq \delta$, we need to access only c_K^s on the sub-domain $[\delta, 2\delta)$.

Upper-bounding approach is similar to the above one. Assume that we have $\langle p_i^j, t_i^j\rangle \in o_i$, where $t_i^j = \delta$, and the key for large-grid of p_i^j is K. To obtain points p which may satisfy $dist(p_i^j, p) \leq r$ and $|t_i^j - t| \leq \delta$, we need to access c_K^l and its neighbor cells on the sub-domains $[0, \delta)$, $[\delta, 2\delta)$, and $[2\delta, 3\delta)$ (i.e., $[\delta, 2\delta)$ and its neighbor sub-domains). The verification also follows this upper-bounding approach.

Note that $\delta = 0$ is a special case. In this case, we build a BIGrid for each generation time of points in O. More specifically, if we have $\langle p_i^j, t_i^j\rangle \in o_i$, a BIGrid with generation time t_i^j is built. Bitwise OR operations are conducted only on the BIGrid with the same generation time.