Spatial Queries with k-Nearest-Neighbor and Relational Predicates

Ahmed M. Aly
Purdue University
West Lafayette, IN
aaly@cs.purdue.edu

Walid G. Aref
Purdue University
West Lafayette, IN
aref@cs.purdue.edu

Mourad Ouzzani
Qatar Computing Research Institute
Doha, Qatar
mouzzani@qf.org.qa

ABSTRACT

The ubiquity of location-aware devices and smartphones has unleashed an unprecedented proliferation of location-based services that require processing queries with both spatial and relational predicates. Many algorithms and index structures already exist for processing k-Nearest-Neighbor (kNN, for short) predicates either solely or when combined with textual keyword search. Unfortunately, there has not been enough study on how to efficiently process queries where kNN predicates are combined with general relational predicates, i.e., ones that have selects, joins and group-by's. One major challenge is that because the kNN is a ranking operation, applying a relational predicate before or after a kNN predicate in a query evaluation pipeline (QEP, for short) can result in different outputs, and hence leads to different query *semantics*. In particular, this renders classical relational query optimization heuristics, e.g., pushing selects below joins, inapplicable. This paper presents various query optimization heuristics for queries that involve combinations of kNN select/join predicates and relational predicates. The proposed optimizations can significantly enhance the performance of these queries while preserving their semantics. Experimental results that are based on queries from the TPC-H benchmark and real spatial data from OpenStreetMap demonstrate that the proposed optimizations can achieve orders of magnitude enhancement in query performance.

1. INTRODUCTION

The widespread use of location-aware devices has led to countless location-based services that embed complex queries with spatial as well as relational predicates. This demands spatial query processors that can efficiently process spatial queries of various complexities.

One of the most ubiquitous spatial predicates is the kNN-Select, e.g., find the k-closest restaurants to my location (or any focal point). Another important spatial predicate is the kNN-Join, e.g., find for each gas station the k-closest fire departments. A kNN-Join can also be useful when multiple kNN-Select predicates are to be executed on the same dataset. To share the execution, exploit data locality, and avoid multiple yet unnecessary scans of the underlying data (e.g., as in [12]), all the query focal points of the kNN-Select predicates are treated as an outer relation and processing is performed in a single kNN-Join.

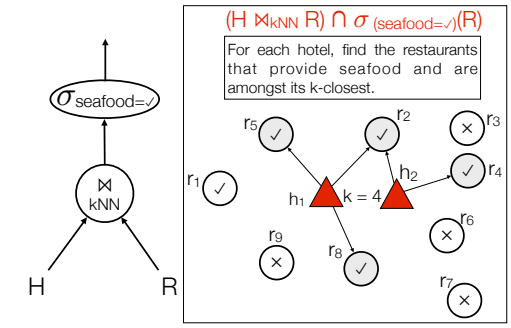
This paper studies the class of queries that combine spatial kNN predicates with relational predicates. Such queries arise frequently in practice. Examples of such queries include: (i) Find a restaurant that is close to my location (kNN-Select predicate) and is within my budget (relational select); (ii) Find (hotel, restaurant) pairs that are close to each other (kNN-Join predicate) such that the hotel is a 5-star hotel (relational select), or that the restaurant offers vegetarian food (relational select), or that the hotel is one of my preferred ones (relational join with my-preferred hotels table).

Queries that involve both kNN and relational predicates raise important challenges. To illustrate, consider the following example for two queries that involve kNN-Join and relational predicates. Assume that we have a Restaurant Table, say R, and an Hotel Table, say H, with the schemas: `(id, location, seafood)` and `(id, location)`, respectively. Attribute `R.seafood` is Boolean and takes the value *`True`* if the restaurant provides seafood, and *`False`* otherwise. The k-NN-Join operates on Attributes `R.location` and `H.location` that represent the locations of the tuples in the two-dimensional space.

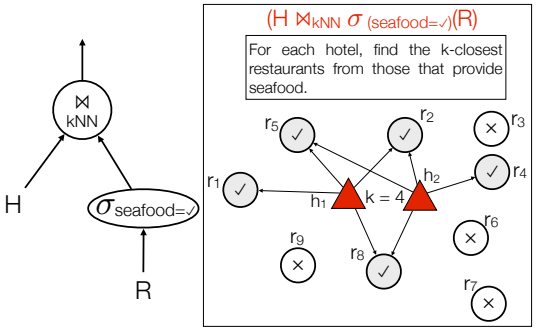
EXAMPLE 1.
(a) For each hotel, find the restaurants that provide seafood and are amongst the hotel's k-closest restaurants.
(b) For each hotel, find the k-closest restaurants from those that provide seafood.

Figure 1 gives the QEPs, relational algebraic expressions, and the output corresponding to the two queries of Example 1 for $k = 4$. The triangles denoted by h_1 and h_2 represent the locations of the tuples in the Hotel Table, and the circles represent the locations of the tuples in the Restaurant Table, denoted by r_1 through r_9. For each Restaurant tuple, the value of the Boolean attribute `R.seafood` is indicated inside the corresponding circle (An 'X' indicates that `seafood=`*`False`* while a checkmark indicates that `seafood=`*`True`*). As the figure demonstrates, the two QEPs produce different outputs.

Although the above queries seem similar, they have different semantics. In the expression: $(\sigma_B(A \bowtie_{kNN} B))$, conceptually, the kNN-Join should be performed *before* filtering any restaurants. In contrast, in the expression: $(H \bowtie_{kNN} \sigma(R))$, conceptually, the kNN-Join should be applied *after* all the non-seafood restaurants are filtered out. Nowadays spatial query processors do not disambiguate these semantics. We refer to the semantics of the former expression as **Pre-Filter** and the semantics of the latter expression as **Post-Filter**. Similar semantics arise for queries with kNN-Select and relational predicates.



(a) A query with a Pre-Filter kNN-Join. The output pairs are: $(h1,\ r2)$, $(h1,\ r5)$, $(h1,\ r8)$, $(h2,\ r2)$, and $(h2,\ r4)$.



(b) A query with a Post-Filter kNN-Join. The output pairs are: $(h1,\ r1)$, $(h1,\ r2)$, $(h1,\ r5)$, $(h1,\ r8)$, $(h2,\ r2)$, $(h2,\ r4)$, and $(h2,\ r5)$, $(h2,\ r8)$.

Figure 1: The Pre-Filter and Post-Filter semantics for queries with kNN-Join and relational predicates.

It is up to the user to choose the proper semantics for the query. While in the above example it might be more intuitive that the user requires the Post-Filter semantics (i.e., retrieve exactly k restaurants for each hotel regardless of the closeness between the hotel and the restaurants), in some other scenarios, the closeness of an entity can have high significance on the intended semantics of the query. To illustrate, consider the following example:

EXAMPLE 2.
For each school, we need to make sure that the nearest hospital has an urgent care unit. Assume that we want to find the schools that do not satisfy this condition, i.e., schools with the nearest hospital not having an urgent care unit.

A possible solution to the above query is to select for each school the nearest hospital, i.e., a kNN-Join ($k = 1$), and then apply a relational predicate (has urgent care = *False*) to each resulting pair. Finally, the list of intended schools is retrieved. Observe that if the relational predicate is applied first before the kNN-Join, during the evaluation of the kNN-Join, every school will be joined with a hospital that does not have an urgent care. In this case, the final result will be all the schools, which is clearly not the correct query answer because some schools may have a neighboring hospital that does have an urgent care unit. These schools were filtered out (too early) by the relational predicate.

In addition to the challenge of disambiguating the semantics of queries with kNN and relational predicates, a ***performance*** challenge emerges. Assume that a user intends to apply the Pre-Filter semantics for the query in Figure 1(a). Similarly to relational joins, the kNN-Join is a relatively costly operation. A well-known heuristic for optimizing a join is to push the select(s) below the join in the QEP (e.g., see [19]). However, if we apply such heuristic to the QEP of Figure 1(a), it will lead to the Post-Filter semantics. The lack of such optimization calls for new optimization techniques that can still leverage the pruning effect of selection without compromising the correctness of evaluation according to the intended semantics.

In addition to the above case of interaction between kNN-Join and relational predicates, we study the cases of interaction between kNN-Select and relational predicates. For each case, we identify the possible semantics, and present optimization techniques that can enhance the query performance while preserving the semantics. To arbitrate between the possible optimization alternatives and the possible QEPs for each query, we utilize our techniques in [5] to estimate the cost of the kNN predicates and decide the cheapest QEPs to execute.

The contributions of this paper are summarized as follows:

- We identify the various semantics for queries with kNN and relational predicates.
- We present various optimization heuristics for queries with kNN-Select/Join and relational predicates that can enhance the query performance while preserving the semantics (Sections 4 and 5).
- We extend our cost model in [5] to show how a query optimizer can arbitrate between the various QEPs that queries with kNN and relational predicates can have (Section 6).
- We conduct extensive experiments using queries from the TPC-H benchmark [2] and real spatial datasets from OpenStreetMap [1]. The experimental results demonstrate that the optimization heuristics coupled with the cost model can achieve a query performance gain of up to three orders of magnitude (Section 7).

2. RELATED WORK

A large body of research has tackled location-based queries. We categorize the related work that is in line with the scope of this paper into three main classes: (1) processing single-operator spatial queries, (2) processing queries that involve spatial and non-spatial operators, and (3) estimating the cost and the selectivity of spatial operators.

In the first class, several research efforts have targeted the efficient processing of queries with single spatial or spatio-temporal operators, e.g., k-nearest-neighbor, reverse nearest-neighbor, aggregate nearest-neighbor, range, and spatial-join operators (e.g., see [7, 17, 23, 28, 37, 40]). However, queries in this class are purely spatial with no relational predicates.

In the second class, two lines of research have been pursued:

1. *Processing spatial queries with relational operators:* Secondo and the BerlinMOD benchmark [15, 16] study the processing of queries that involve combinations of spatial range predicates and relational predicates. However, they address neither the *optimization* of these queries nor the processing of queries that combine kNN and relational predicates. [41] studies the processing of the spatial kNN-Select/Join predicates inside a relational database, but does not address the optimization of such predicates when combined with relational predicates in the same query.

2. *Spatial keyword queries:* [13] surveys the state-of-the-art for geo-textual indices that address spatial keyword queries.

Usually, these indices have two components: (i) a spatial index, e.g., an R-tree and (ii) a text index, e.g., an inverted file. Some indices loosely combine these two components while others integrate them tightly resulting in hybrid indices, e.g., the IR-Tree [14, 25, 39]. Although useful, these indices are optimized only for spatio-textual search and cannot be directly used for general predicates on relational attributes, e.g., hotel price and customer rating of a restaurant, etc.

In the third class, several research efforts (e.g., see [3, 6, 8, 9, 11, 22, 26, 27, 36]) tackled the estimation of the selectivity and cost of the spatial join and range operators. [5, 38] study the cost of the kNN predicates. In this paper, we adopt our cost model in [5] to optimize the execution of queries that combine kNN and relational predicates.

3. PRELIMINARIES

We assume that the data consists of points in the two-dimensional space. For simplicity, we use the Euclidean distance as the distance metric. We do not assume a specific indexing structure. The algorithms can be applied to a quadtree, an R-tree, or any of their variants, e.g., [10, 18, 20, 24, 29]. These are hierarchical spatial data structures that recursively partition the underlying space/points into blocks until the number of points inside a block satisfies some criterion (e.g., being less than some threshold). We just assume that whatever index is adopted, it will provide a collection of leaf blocks where the **count** of points in each block is maintained.

We make extensive use of the MINDIST and MAXDIST metrics [28]. Refer to Figure 2 for illustration. The MINDIST (or MAXDIST) between a point, say p, and a block, say b, refers to the minimum (or maximum) possible distance between p and any point in b. Similarly, the MINDIST (or MAXDIST) between two blocks is the minimum (or maximum) possible distance between them. In some scenarios, we process the blocks in a certain order according to their MINDIST from a certain point. An ordering of the blocks based on the MINDIST from a certain point or block is termed MINDIST ordering.

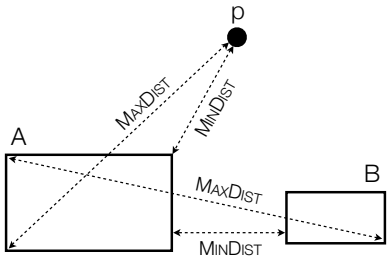


Figure 2: The MINDIST and MAXDIST metrics.

In this paper, we focus on the variants of the kNN operations given below. Assume that we have two tables, say R and S, where each tuple represents a point in the two-dimensional space and contain some other attributes that describe the point. For example, for an hotel table, hotel location is the 2D-point spatial attribute, while hotel name, hotel address, hotel rating, and hotel amenities are attributes that describe an hotel.

- ***k*NN-Select**: Given a query-point q, $\sigma_{k,q}(R)$ returns the k-closest to q from the set of points in R.
- ***k*NN-Join**: $R \bowtie_{kNN} S$ returns all the pairs (r, s), where $r \in R$ and $s \in S$, and s is among the k-closest points to r.

Observe that the kNN-Join is an asymmetric operation, i.e., the two expressions: $(R \bowtie_{kNN} S)$ and $(S \bowtie_{kNN} R)$ are not equivalent. In the expression $(R \bowtie_{kNN} S)$, we refer to Relation R as the outer relation and to Relation S as the inner relation.

We assume that a query is compiled into a binary tree-structured QEP of pipelined operators that follow a lazy evaluation scheme using operator iterators. Each operator is aware of its left and right child operators. A unary operator, e.g., select, has only one child operator. All the operators are pull-based. Starting from the root, each operator in the QEP calls the `getNext()` method of its direct child operator(s) to get a tuple. Then, the same method is recursively called down the QEP by the underlying operators until the table-scan operators at the leaf level are reached. Once an operator gets its next tuple, it performs its designated operator logic, be it a select, a join, or a group-by, and reports its output to its parents to respond to the parent's `getNext()` call. For more details on pull-based query evaluation pipelines, the reader is referred to [19].

4. KNN-SELECT WITH RELATIONAL PREDICATES

As discussed in Section 1, depending on the order of evaluating a kNN-Join in a QEP, a query with a kNN-Join and relational predicates can have two semantics, namely, the Post-Filter and Pre-Filter semantics. Similarly, queries with kNN-Select and relational predicates have the same two semantics. To illustrate, consider the following example. We use the same schema as in Example 1.

EXAMPLE 3.
(a) From the k-closest restaurants to my location q, find the restaurants that provide seafood.
(a) From the restaurants that provide seafood, find the k-closest to my location q.

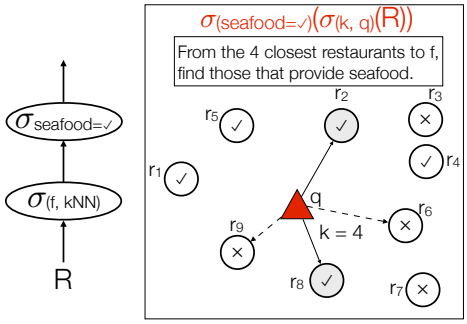
Figure 3 gives the QEPs, relational algebraic expressions, and the output corresponding to the two queries of Example 3 for $k = 4$. We use the same notations as in Figure 1. As the figure illustrates, the two queries produce different results, and hence they have different semantics. We refer to the semantics of the query in Example 3(a) as Pre-Filter, and the those of the query in Example 3(b) as Post-Filter.

As mentioned in Section 1, it is up to the user to choose which semantics to apply. In the rest of this section, we study alternative query optimization heuristics that can enhance the execution of the above queries while preserving their semantics.

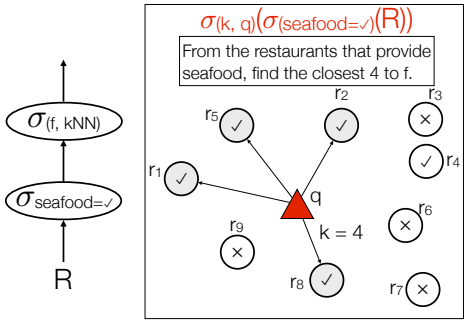
4.1 Pre-Filter kNN-Select

Given an input Relation, say R_s, a query with a Pre-Filter kNN-Select retrieves the tuples that qualify a set of relational predicates from among the tuples that are k-closest to a certain query-point, say q. The relational predicates can arbitrarily involve selects, joins, or group-by's on R_s as well as other relations, say R_1 through R_n. Based on the semantics of the query, tuples from R_s that are amongst the nearest-neighbors say q are the only tuples that should contribute to the result of the query. Intuitively, to process such query, the kNN-Select should be performed early in the QEP, and then the relational predicates should be applied afterwards. Refer to QEP(a) in Figure 4 for illustration.

Because the cost of the kNN-Select depends on many parameters, e.g., the value of k and the location of q, performing the kNN-Select early in the QEP may be costly and can lead to suboptimal overall query performance. This is especially true when the selectivity of the relational predicates is high. In such case, applying the relational predicates first and selecting the tuples that belong to



(a) A query with a Pre-Filter kNN-Select. The output is: r_2 and r_8.



(b) A query with a Post-Filter kNN-Select. The output is: r_1, r_2, r_5, and r_8.

Figure 3: The Pre-Filter and Post-Filter semantics for queries with kNN-Select and relational predicates.

the kNN may result in better performance. An interesting problem is how to select the nearest-neighbors after applying the relational predicates knowing that if a regular kNN-Select predicate is applied, it will lead to the Post-Filter kNN semantics. To address this problem, we introduce the k-in-circle optimization heuristic. For each tuple, say t, that qualifies the relational predicates, we count the number of points that are included inside the circle centered at t and whose radius equals the distance between q and t. If the number of points in the circle is $\geq k$, we determine that t does not belong to the answer of the query, and vice versa. Refer to Figure 5 for illustration. Assume that points t_1 and t_2 qualify the relational predicates. When $k = 4$, t_1 belongs to the nearest-neighbors of q because the corresponding circle encloses 2 points; in contrast, t_2 is ignored because the corresponding circle encloses 4 points.

Notice that the counting process is robust. Since we store the count of points in each block of the index, when a block is completely included within the circle, the count of points in that block is simply added to the total count without examining any of the points in that block. Similarly, when a block is completely outside the circle, that block is pruned.

To further enhance the k-in-circle optimization, instead of performing the counting for every tuple that qualifies the relational predicates, we apply the counting only for the k-closest to the query location. These k-closest tuples can be efficiently computed using a priority queue. Afterwards, we scan these k-closest tuples in decreasing order of their distance from the query point (by successive retrievals of the top element in the priority queue). Given the top element in the priority queue, we do the counting process and if the corresponding count is $\geq k$, we ignore that tuple and get the next tuple from the priority queue. If the count is $< k$, we determine that this tuple and *all* the remaining tuples in the priority queue belong

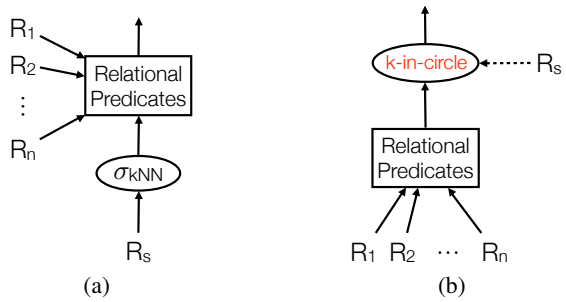


Figure 4: Different QEPs for queries with Pre-Filter kNN-Select.

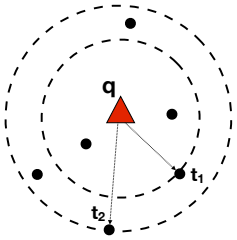


Figure 5: The k-in-circle optimization

to the answer of the query. For instance, in Figure 5, if both t_1 and t_2 qualify the relational predicates and $k = 5$, t_2 is scanned first because it is is farther to q than t_1. Since the circle corresponding to t_2 encloses 4 points, we conclude that t_2 belongs to the answer and consequently t_1 does.

As we show in Section 6, depending on the cost of the relational predicates as well as the cost of the kNN-Select, we arbitrate between QEP(a), QEP(b).

4.2 Post-Filter kNN-Select

Given an input Relation, say R_s, a query with a Post-Filter kNN-Select retrieves the k-closest tuples to a certain query-point, say q, from the tuples that qualify a set of relational predicates. The relational predicates can arbitrarily involve selects, joins, or group-by's on R_s as well as other relations, say R_1 through R_n.

According to the semantics of this query, pushing the kNN-Select below the relational predicates in the QEP produces incorrect results because it leads to the Pre-Filter semantics. Conceptually, the relational predicates should be evaluated first. Afterwards, tuples that qualify these predicates are processed to find the k-closest to the q. Refer to QEP(a) of Figure 6 for illustration.

Observe that the kNN-Select operator in QEP(a) operates with no index information because the tuples that qualify the relational predicates are computed as part of the evaluation of the query, and hence have no corresponding index. Furthermore, when the selectivity of the relational predicates is low, i.e., when many tuples qualify the relational predicates, QEP(a) can be costly. The reason is that in this case, the relational predicates are applied to *all* the tuples and many tuples of these tuples will be compared according to their closeness to q, however, in the end, only k of these tuples comprise the answer to the query. Thus, although QEP(a) is legitimate, it can have bad performance. One important observation is that the k tuples that comprise the answer of the query are likely to be located around or within the locality of q. Hence, when the value of k is relatively small, applying the relational predicates to tuples that are far from the query-point is redundant and can be avoided

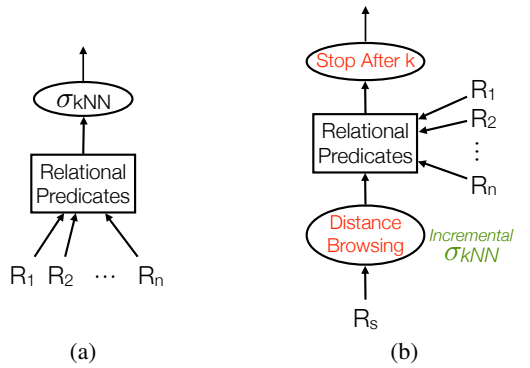


Figure 6: Different QEPs for queries with Post-Filter kNN-Select.

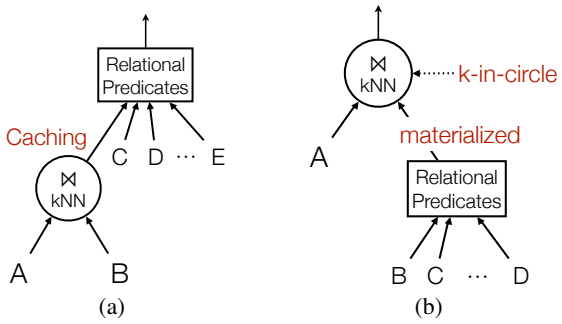


Figure 7: Different QEPs for queries with Pre-Filter kNN-Join.

without affecting the correctness of evaluation of the query.

To address this issue, [21] presents the *Distance-Browsing* approach for processing kNN-Select. The main idea of this approach is that it can progressively retrieve the *next* nearest-neighbor of a query point through its `getNextNearest()` method. The Distance-Browsing approach is incremental; every time the `getNextNearest()` method is invoked, it does not redo the search from scratch. Instead, an internal state is kept to enable the efficient retrieval of the next nearest point without re-scanning the data points.

To process a query with a Post-Filter kNN-Select, one can apply QEP(b) of Figure 6. Each time the `getNextNearest()` of the kNN-Select operator is invoked, the next nearest tuple is returned, and then the relational predicates are applied to that tuple. Once k tuples qualify the relational predicates, the execution terminates (see the 'Stop After k' operator at the root of the QEP). In Section 6, we show how to choose between QEP(a) and QEP(b).

Although the Distance-Browsing approach is good for queries with Post-Filter kNN-Select, it is inefficient when applied to queries with Post-Filter kNN-Joins because it operates on a tuple-by-tuple basis while processing a kNN-Join is more efficient when applied on a block-by-block basis. In Section 5.2, we present a block-by-block approach for processing Post-Filter kNN-Joins.

5. KNN-JOIN WITH RELATIONAL PREDICATES

In this section, we study the various cases for interleaving a kNN-Join with relational predicates. [34] studies the conceptual evaluation of queries that involve similarity predicates [32], e.g., similarity joins including ϵ-join and kNN-joins [33]. [34] provides equivalence transformation rules that guarantee the correctness of evaluation for such queries. According to [34],

$$\sigma_A(A \bowtie_{kNN} B) \equiv \sigma(A) \bowtie_{kNN} B.$$

In other words, when there is a relational predicate, e.g., select on the *outer* relation of a kNN-Join, applying the relational predicate after computing the kNN-Join, i.e., $(\sigma_A(A \bowtie_{kNN} B))$ is equivalent to applying the predicate to the outer relation before computing the kNN-Join, i.e., $(\sigma(A) \bowtie_{kNN} B)$. The reason is that applying the relational predicate before the kNN-Join will filter out some tuples (points) from the outer relation that would also be filtered out anyway if the relational predicate is applied after the kNN-join (Refer to [4, 34] for more detail). Thus, pushing the relational predicates below the outer relation of a kNN-Join is a valid query optimization heuristic. In contrast, when there is a relational predicate, e.g., a relational select on the *inner* relation of a kNN-Join, applying the relational predicate after computing the kNN-Join is not equivalent to applying the relational predicate to the inner relation before computing the kNN-Join, i.e.,

$$\sigma_B(A \bowtie_{kNN} B) \not\equiv A \bowtie_{kNN} \sigma(B).$$

As explained in Example 1 (see Figure 1 in Section 1), we refer to the kNN-Join in Figure 1(a) as Pre-Filter and the one in Figure 1(b) as Post-Filter. In the rest of this section, we present various optimization alternatives that can enhance the execution of queries with kNN-Join and relational predicates while preserving the semantics of these queries.

5.1 Pre-Filter kNN-Join

In this section, we study queries with Pre-Filter kNN-Join, where the relational predicates are on the inner relation of the kNN-Join. Conceptually, the kNN-Join should be applied first, then the resulting tuples are filtered out by the relational predicates. Refer to QEP(a) of Figure 7 for illustration.

Although QEP(a) is legitimate, it can suffer from repeated redundant computations. It is highly likely that a point in the inner relation corresponds to the k-nearest-neighbors of more than one point in the outer relation, e.g., Point r_2 in Figure 1(b). [35] shows that for $k = 1$, the number of the reverse nearest neighbors of a point can reach up to six. For higher values of k, the number of reverse nearest neighbors of every point can be much higher, leading to a large number of repeated computations. To avoid this redundancy, we use a cache (i.e., a hash table) that stores for every processed tuple whether it qualifies or disqualifies the relational predicates. If a tuple is already processed (i.e., exists in the hash table) and qualifies the relational predicates, it is directly emitted to the output, otherwise, it is ignored.

QEP(b) of Figure 7 is another alternative for evaluating queries with Pre-Filter kNN-Join. In this QEP, the relational predicates are applied to the inner relation first and the qualifying tuples are materialized into a temporary relation. Afterwards, the k-in-circle optimization heuristic is applied as we explain below. Notice that this optimization allows for pushing relational select(s) below the kNN-Join without compromising the correctness of evaluation.

In QEP(b), for every tuple, say p_o, in the outer relation, the k-closest tuples in the materialized relation are determined and scanned in a decreasing order of their distance from p_o. As discussed earlier in Section 4, this can be efficiently achieved using a priority queue. For each pair (p_o, p_i), where $p_i \in$ the materialized relation, we count the number of points from the inner relation that are contained within the circle centered at p_o and whose radius is the distance between p_o and p_i. If the number of points within that circle is $\geq k$, p_i is ignored, otherwise, the pair (p_o, p_i) is added to

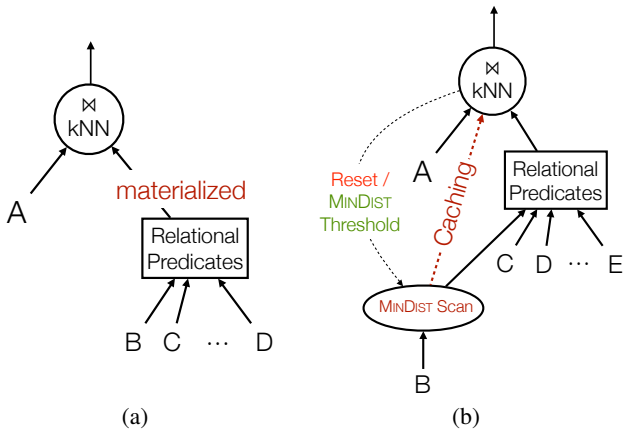


Figure 8: Different QEPs for queries with Post-Filter kNN-Join.

the answer of the query as well as all the pairs (p_o, p_r), where p_r is any point that is remaining in the priority queue.

Observe that in QEP(a), any index-based algorithm for processing the kNN-Join (e.g., [31]) can be applied. In contrast, in QEP(b), the materialized relation has no corresponding index. In this case, the kNN-Join is executed using a nested-loops join [19]. In Section 6, we show how to choose between QEP(a) and QEP(b).

5.2 Post-Filter kNN-Join

In this section, we study queries with Post-Filter kNN-Join, where the relational predicates are on the inner relation of the kNN-Join. Conceptually, the relational predicates should be applied to the inner relation first and the qualifying tuples are materialized in a temporary relation.[1] Afterwards, the kNN-Join can be performed between the outer relation and the materialized relation. Refer to QEP(a) of Figure 8 for illustration.

QEP(b) of Figure 8 is another alternative for evaluating queries with Post-Filter kNN-Join. In this QEP, the index on the inner relation is used with no need to materialize the tuples that qualify the relational predicates. Given a block from the outer relation, say b_o, we scan the blocks of the inner relation in MinDist order from b_o. We apply the relational predicates to the points in these blocks from the inner relation and add those tuples that qualify the relational predicates to a list, say $qualifyingList$. Once the size of $qualifyingList$ reaches or exceeds k, we record the highest MaxDist, say $highestMaxDist$, from b_o of the encountered blocks. We continue scanning until a block is encountered whose MinDist is greater than $highestMaxDist$. Afterwards, for every point in b_o, the nearest neighbors in $qualifyingList$ are determined and are added to the result of the query. Observe that when a scanning round corresponding to a block from the outer relation is completed, the scan operator is reset to start a new round of scan according to the MinDist of the next block; For illustration, refer to the dashed line from the kNN-Join operator to the Scan operator in QEP(b) of Figure 8.

Similarly to QEP(b) of Figure 7, repeated computations of the relational predicates can occur for points from the inner relation that belong to the k-nearest-neighbors of multiple points in the outer relation. We apply the same caching mechanism to solve this problem. In particular, if a tuple is already processed (i.e., exists in the hash table) and qualifies the relational predicates, it is directly passed to the kNN-Join operator, otherwise, it is ignored. For illustration, refer to the dashed line from the Scan operator to the kNN-Join operator in QEP(b) of Figure 8.

[1]Similar to QEP(a) of Figure 7, the materialized relation has no corresponding index that the kNN-Join operation can leverage.

6. COST-BASED OPTIMIZATION

In order to decide which query processing strategy to use for a query with either a Pre-Filter or a Post-Filter kNN-Select/Join, estimating the cost of these kNN predicates is essential. For instance, for a query optimizer to choose between QEP(a) and QEP(b) in Figure 6, the cost of the kNN-Select predicate needs to be determined. We adopt our cost estimation techniques in [5] for estimating the cost of the kNN-Select and kNN-Join operators. Given a kNN operator, in [5], we estimate the number of blocks that are going to be scanned during the processing of this operator. The main idea of our cost model is to maintain a compact set of *catalog* information that can be kept in main-memory to enable fast estimation via lookups. In this paper, we apply the Staircase and Catalog-Merge techniques [5] to estimate the cost of the kNN-Select and kNN-Join, respectively.

In the rest of this section, we show how the cost model can be used to arbitrate between the alternative QEPs that each query can have.

Pre-Filter kNN-Select

Refer to Figure 4 for illustration. The cost of the kNN-Select in QEP(a) can be directly estimated using the Staircase technique in [5]. However, in QEP(b), the kNN-Select operates with no spatial index, and hence the cost is determined by the cost of the relational operators. Notice that the CPU cost of the kNN operator applies the k-in-circle optimization is $O(n \log k)$ in CPU time, where n is the number of tuples that qualify the relational predicates below the operator.

Depending on the cost of the relational predicates as well as the cost of the kNN-Select, we choose either QEP(a) or QEP(b).

Post-Filter kNN-Select

Refer to Figure 6 for illustration. Estimating the cost of the kNN-Select operator in QEP(b) can be described as follows. Because there is no correlation between the location of a tuple and whether it qualifies the relational predicates, we can assume that the tuples that qualify the relational predicates are uniformly distributed in the space. If the selectivity of the relational predicates is $\rho \leq 1$, then $\frac{k}{(1-\rho)}$ tuples will contain k tuples that qualify the relational predicates. Thus, we substitute the value of k by $\frac{k}{(1-\rho)}$ and then apply the Staircase technique in [5] to estimate the cost of the kNN-Select. However, in QEP(a), the kNN-Select operates with no spatial index, and hence the cost is determined by the cost of the relational operators.

Depending on selectivity and cost of the relational predicates and the value of k, we choose either QEP(a) or QEP(b) of Figure 6.

Pre-Filter kNN-Join

Refer to Figure 7 for illustration. The cost of the kNN-Join in QEP(a) is straightforward. In this case, the kNN-Join operates in a standard way, and hence its cost can be estimated by directly using the Catalog-Merge technique in [5]. However, in QEP(b), the kNN-Join operates with no index on the inner relation, and hence its cost is equivalent to the cost of a nested-loops join [19]. The cost of the relational predicates in both QEP(a) and QEP(b) is the same because QEP(a) applies a caching technique that avoids any repeated computations. Depending on the relational selectivity and the value of k, we choose either QEP(a) or QEP(b).

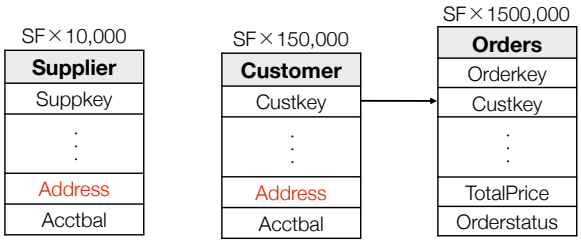


Figure 9: Part of the schema of the TPC-H benchmark.

Post-Filter kNN-Join

Refer to Figure 8 for illustration. In QEP(a), the kNN-Join operates with no spatial index, and hence its cost is the same as the cost of a nested-loops join. The overall cost of this QEP equals the cost of performing the relational predicates plus the cost of a nested-loops join [19]. Estimating the cost of the kNN-Join operator in QEP(b) can be described as follows. Similarly to kNN-Select queries, we assume no correlation between the location of a tuple and whether it qualifies the relational predicates, and hence we assume that the tuples that qualify the relational predicates are uniformly distributed in the space. If the selectivity of the relational predicates is $\rho \leq 1$, we substitute the value of k by $\frac{k}{(1-\rho)}$, and then apply the Catalog-Merge technique in [5] to estimate the cost of the kNN-Join.

Depending on the selectivity and cost of the relational predicates and the value of k, we choose either QEP(a) or QEP(b) of Figure 8.

7. EXPERIMENTS

In this section, we evaluate the performance of the proposed optimization techniques. We realize a testbed in which we implement the state-of-the-art kNN-Select and kNN-Join algorithms as described in [21, 31]. In addition, we employ our Staircase and Catalog-Merge techniques as described in [5] to estimate the cost of the kNN operators. Our implementation is based on a region-quadtree index [30], where each node in the quadtree represents a region of space that is recursively decomposed into four equal quadrants, subquadrants, and so on with each leaf node containing points that correspond to a specific subregion. We choose the quadtree because the blocks of a quadtree do not spatially overlap. This property leads to robust performance for kNN queries. In our testbed, we tried an R-tree implementation, but it does not yield the same good performance as the quadtree even for the baseline kNN queries due to the spatial overlap of the R-tree blocks. All implementations are in Java. Experiments are conducted on a machine running Mac OS X on Intel Core i7 CPU at 2.3 GHz and 8 GB of main memory.

The datasets used in the experiments are based on the TPC-H benchmark [2]. We choose to use the TPC-H benchmark because it is a well-crafted source of relational data with well-defined schema and queries with complex relational expressions. We generate various instances of the data using the TPC-H generator with different scale factors. Figure 9 gives part of the relational schema of the TPC-H benchmark tables. The size of each table is displayed as multiples of the scale factor (SF). For instance, if SF = 10, then the size of the Orders Table is $10 \times 1.5M = 15M$ tuples. For illustration, only the attributes that are relevant to the queries we use are displayed.

In Figure 9, each of the Customer and Supplier tables has an address attribute. The TPC-H-generated values for this attribute are randomly generated strings. We replace them with real location data from OpenStreetMap [1]. The location of each point in the data represents the $(lat, long)$ coordinates of real GPS data collected world-wide. Figure 10 displays a sample of the data that we plot through a visualizer that we build as part of our test bed. The figure also displays a region-quadtree that is built on top of the data.

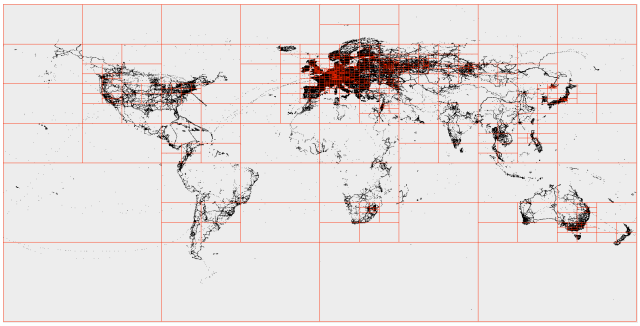


Figure 10: Sample of OpenStreetMap GPS data with a region-quadtree built on top

The queries we use in the experiments are derived from Query **Q13** specified by the TPC-H benchmark. We choose this query for the following reasons: 1) it contains a reference to the address attribute in a way that makes it natural to use with a kNN predicate, 2) it has relatively complex relational constructs, e.g., group-by, and joins, allowing us to demonstrate that the proposed optimizations work well with relatively complex relational predicates, and 3) the relational constructs in the query allow us to try different selectivity values, and hence demonstrate how our cost estimation model can be used to make the right choice for a QEP.

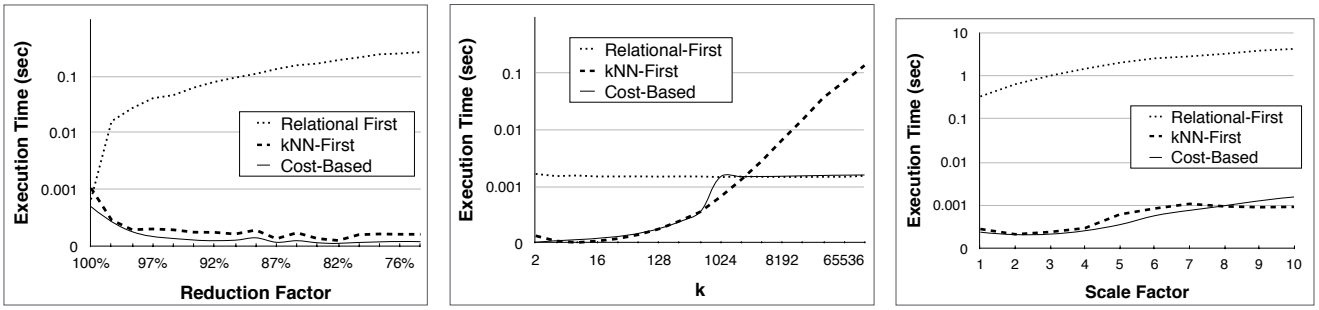
For each customer, say c, Q13 retrieves the number of orders c has made. To enable different selectivities for our corresponding relational predicates, we change the query to retrieve a customer if he made at least one order with price $> t$. This threshold represents a tuning parameter for controlling the selectivity or the *reduction factor* of the relational expression. The modified query can be expressed in SQL as follows:

```
SELECT C.Custkey, COUNT(O.Orderkey) AS N
FROM Customer C, Orders O
WHERE C.Custkey = O.Custkey
AND O.TotalPrice > t
GROUP BY C.custkey
HAVING N > 0;
```

We try all the possible values of t and map each value to the corresponding reduction factor using histograms for the TotalPrice Attribute. Furthermore, to speedup the execution of the above query, we assume the existence of a hash index on Custkey Attribute of Table Orders (to speed up the join between the Tables Customer and Orders) as well as a B+Tree index on the TotalPrice Attribute. Furthermore, we assume that both Supplier and Customer Tables are indexed using region quadtrees based on Attribute Address.

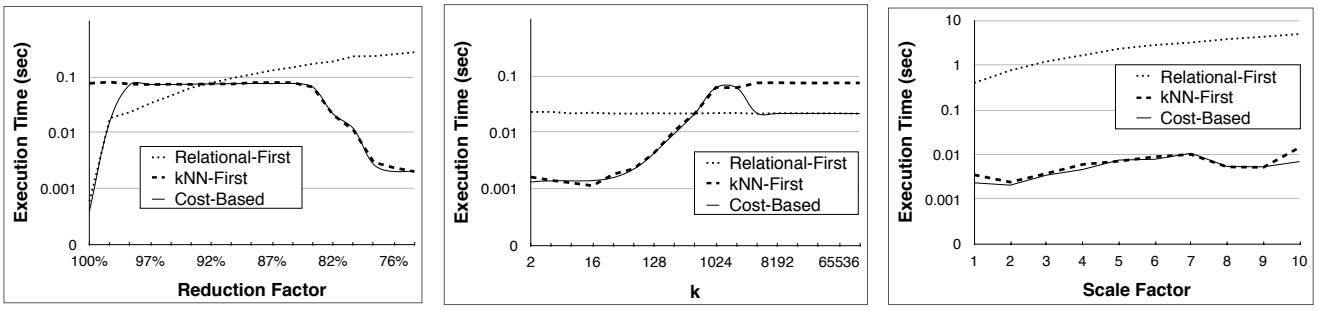
We embed a kNN-Select or a kNN-Join into the above query. For each possible query semantics, we have two possible QEPs. Furthermore, we estimate the cost of each these two QEPs and choose the one that has the least cost. We refer to the QEP that automatically chooses the best QEP based on the estimated cost as the **Cost-Based** QEP.

For the queries we study, our performance metric is the execution time. We monitor this metric after varying: 1) the value of k, 2) the selectivity (i.e., reduction factor) of the relational expression, and 3) the scale factor, i.e., the size of the database.



(a) k = 10. Scale Factor = 1. (b) Reduction Factor = 90%. Scale Factor = 1. (c) k = 10. Reduction Factor = 70%.

Figure 11: Performance of queries with Pre-Filter kNN-Select.



(a) k = 10. Scale Factor = 1. (b) Reduction Factor = 90%. Scale Factor = 1. (c) k = 10. Reduction Factor = 70%.

Figure 12: Performance of queries with Post-Filter kNN-Select.

7.1 Pre-Filter kNN-Select

In this experiment, we study a query with a Pre-Filter kNN-Select. Given a threshold, say t, and a query-point, say q, that is chosen at random, the query chooses from the k-nearest customers to q those who have made at least one order of TotalPrice $> t$. The query can be expressed in SQL as follows:

```
SELECT C.Custkey, COUNT(O.Orderkey) AS N
FROM Customer C, Orders O
WHERE C.Custkey = O.Custkey
AND O.TotalPrice > t AND C is kNN q
GROUP BY C.custkey
HAVING N > 0;
```

We examine two QEPs for the query; Refer to QEP(a) and QEP(b) of Figure 4 for illustration. We refer to QEP(a) as kNN-First and QEP(b) as Relational-First. In addition, we examine the Cost-Based QEP that automatically chooses either QEP(a) and QEP(b) based on the estimated cost.

Figure 11 gives the performance of the three QEPs and shows how the Cost-Based QEP succeeds to choose the best QEP in most of the cases. Figure 11(a) illustrates that the kNN-First QEP has almost constant performance irrespective of the value of the reduction factor because the value of k is constant. Because the value of k is small in this case, it is cheap to apply the kNN-Select first and apply the relational predicates afterwards to only k tuples, and hence the kNN-First QEP has better performance (by up to two orders of magnitude). However, as Figure 11(b) demonstrates, the performance of the kNN-First QEP degrades as the value of k increases, which is a natural result. In contrast, the Relational-First QEP has constant performance; because the reduction factor is constant (90%), the k-in-circle optimization enables the QEP to process the same number of tuples regardless of the value of k.

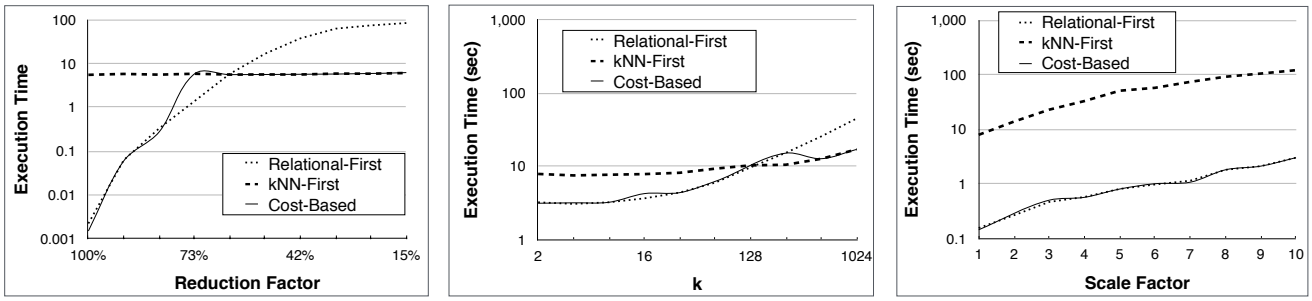
Figure 11(c) illustrates that for different scale factors, when the value of k is small, the Cost-Based QEP always makes the right choice of choosing the kNN-First QEP, leading to three orders of magnitude gain compared to the Relational-First QEP.

Last but not least, the Cost-Based QEP succeeds to choose the best QEP in most of the cases. Note that there are some cases where the Cost-Based QEP does not make the right choice, but this happens near the points of intersection between the two curves of the performance of the Relational-First and kNN-First QEPs. In this case, the two QEPs have almost the same performance; it does not matter which QEP to choose. In contrast, Figure 11(c) demonstrates that when the difference in performance between the two QEPs is significant, i.e., three orders of magnitude, the Cost-Based QEP always succeeds in making the right choice.

7.2 Post-Filter kNN-Select

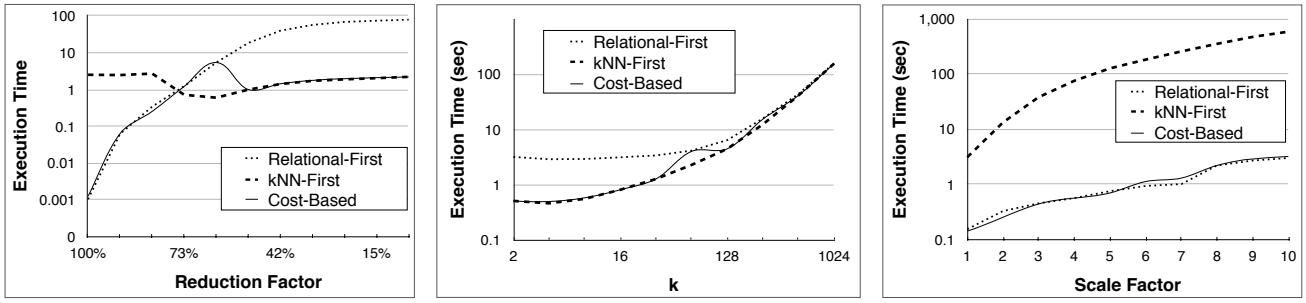
In this experiment, we study a query with a Post-Filter kNN-Select. Given a threshold, say t, and a query-point, say q, that is chosen at random, the query chooses the k-closest customers to q from those customers who have made at least one order of TotalPrice $> t$. The query can be expressed in SQL as follows:

```
SELECT C.Custkey, COUNT(O.Orderkey) AS N
FROM Customer C, Orders O
WHERE C.Custkey = O.Custkey
AND O.TotalPrice > t
GROUP BY C.custkey
HAVING N > 0
ORDER BY distance(C.address, q) LIMIT k;
```



(a) k = 10. Scale Factor = 1. (b) Reduction Factor = 70%. Scale Factor = 1. (c) k = 10. Reduction Factor = 90%.

Figure 13: Performance of queries with Pre-Filter kNN-Join.



(a) k = 10. Scale Factor = 1. (b) Reduction Factor = 70%. Scale Factor = 1. (c) k = 10. Reduction Factor = 90%.

Figure 14: Performance of queries with Post-Filter kNN-Join.

We examine two QEPs for the query; Refer to QEP(a) and QEP(b) of Figure 6 for illustration. Similarly to the above experiments, we refer to QEP(a) as Relational-First, QEP(b) as kNN-First, and the QEP that automatically chooses either QEP(a) and QEP(b) as the Cost-Based QEP. For the Relational-First QEP, we use the B+Tree index on the TotalPrice Attribute to retrieve the tuples of TotalPrice $< t$.

Figure 12 gives the performance of the three QEPs. As illustrated in the figure, there is a variability in the dominance of one QEP over the other. For instance, in Figure 12(a), when the reduction factor is high, the performance of the Relational-First QEP is better by more than two orders of magnitude. In this case, tuples of TotalPrice $< t$ are quickly retrieved from the B+Tree and the k-closest tuples are quickly selected from them. However, the kNN-First QEP terminates only when k tuples that qualify the relational predicates (TotalPrice $< t$) are retrieved (refer to the 'Stop After k' operator at the top of the kNN-First QEP). Hence, for the same case of high reduction factor, early termination is unlikely to happen for the kNN-First QEP as the MINDIST Scan operator will keep scanning almost all the blocks in the entire space. In contrast, a low value of the reduction factor implies that k matching tuples will be found around q, and hence, early termination is more likely to happen; the kNN-First QEP will dominate in this case.

Similarly, Figure 12(b) shows that for small k values, the kNN-First QEP performs better because in this case, early termination is more likely to happen. The opposite happens when the value of k increases because early termination is unlikely to occur in this case. In either case, the Relational-First QEP blindly processes the input table without leveraging the spatial locality, and hence has almost constant performance. Figure 12(c) illustrates that for different scale factors, when the value of k is small, the Cost-Based QEP always makes the right choice of choosing the kNN-First QEP.

7.3 Pre-Filter and Post-Filter kNN-Joins

In this experiment, we use a kNN-Join between the Supplier and Customer tables, where a relational expression is applied on the Customer Table (i.e., the inner table of the kNN-Join). In particular, we study two queries:

1. A query with a *Pre-Filter* kNN-Join: Given a threshold t, retrieve for each Supplier, from the k-closest customers those who have made at least one order with TotalPrice $> t$.
2. A query with a *Post-Filter* kNN-Join: Given a threshold t, retrieve for each Supplier, the k-closest customers from those who have made at least one order with TotalPrice $> t$.

Similarly to the above experiments, for each of the above queries, we examine two QEPs as well as the Cost-Based QEP that automatically chooses the QEP with the least cost. We refer to QEP(a) and QEP(b) of Figure 7 as kNN-First and Relational-First, respectively. Similarly, we refer to QEP(a) and QEP(b) of Figure 8 as Relational-First and kNN-First, respectively.

Figures 13 and 14 give the performance of the various QEPs at different parameter settings. Figures 13(a) (and similarly Figure 14(a)) shows that when the reduction factor is high, the Relational-First QEP performs better (by three orders of magnitude). In this case, customers that qualify the relational predicate are materialized into a small relation, and hence, the kNN-Join will be easy to perform, either with the k-in-circle optimization for a Pre-Filter query or with a standard kNN priority queue for a Post-Filter query. The kNN-First QEP blindly applies the kNN-Join without leveraging the pruning effect of the high relational selectivity. In contrast, when the reduction factor is low, the kNN-First QEP performs better (by two orders of magnitude) because the size of the materialized relation is large in this case, which degrades the performance of the Relational-First QEP.

Figure 13(b) (and similarly Figure 14(b)) shows that the performance of both the Relational-First and kNN-First QEPs degrades as the value of k increases, which is a natural result because the k-nearest-neighbors have to be determined in either QEPs.

Figure 13(c) (and similarly Figure 14(c)) shows that the difference in performance between the QEPs is maintained for different scale factors when the other parameters are fixed. In this case, because the reduction factor is high, the Relational-First QEP outperforms the kNN-First QEP by almost two orders of magnitude.

Last but not least, the Cost-Based QEP succeeds in selecting the best QEP in most of the cases except near the points of intersection between the two curves. This is especially true when the difference in performance is significant, e.g., three orders of magnitude as in Figures 13(c) and 14(c). This proves the robustness of our cost estimation model.

8. CONCLUDING REMARKS

In this paper, we present a comprehensive study for the various cases of queries with kNN and relational predicates. Such queries embed two challenges: 1) a semantics disambiguation challenge, and 2) a performance challenge. We disambiguate the different semantics that emerge from the coexistence of kNN-Select/Join and relational predicates within the same query. We present various optimization heuristics to enhance the performance of these queries while preserving their semantics. For each query, we apply a cost-based estimation model that arbitrates between the various optimizations and possible QEPs that the query can have. Our experiments that are based on the TPC-H benchmark and real spatial datasets from OpenStreetMap demonstrate orders of magnitude enhancement in query performance.

9. REFERENCES

[1] OpenStreetMap bulk gps point data. `http://blog.osmfoundation.org/2012/04/01/bulk-gps-point-data/`.

[2] TPC-H benchmark version 2.14.4. `http://www.tpc.org/tpch/`.

[3] S. Acharya, V. Poosala, and S. Ramaswamy. Selectivity estimation in spatial databases. In *SIGMOD Conference*, pages 13–24, 1999.

[4] A. M. Aly, W. G. Aref, and M. Ouzzani. Spatial queries with two kNN predicates. *PVLDB*, 5(11):1100–1111, 2012.

[5] A. M. Aly, W. G. Aref, and M. Ouzzani. Cost estimation of spatial k-nearest-neighbor operators. In *EDBT*, pages 457–468, 2015.

[6] N. An, Z.-Y. Yang, and A. Sivasubramaniam. Selectivity estimation for spatial joins. In *ICDE*, pages 368–375, 2001.

[7] W. G. Aref. Pipelined spatial join processing for quadtree-based indexes. In *GIS*, pages 49:1–49:4, 2007.

[8] W. G. Aref and H. Samet. Estimating selectivity factors of spatial operations. In *FMLDO*, pages 31–43, 1993.

[9] W. G. Aref and H. Samet. A cost model for query optimization using R-Trees. In *ACM-GIS*, pages 60–67, 1994.

[10] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, pages 322–331, 1990.

[11] A. Belussi and C. Faloutsos. Estimating the selectivity of spatial queries using the 'correlation' fractal dimension. In *VLDB*, pages 299–310, 1995.

[12] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 203–214, 2002.

[13] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: An experimental evaluation. *PVLDB*, 6(3):217–228, 2013.

[14] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.

[15] V. T. de Almeida, R. H. Güting, and T. Behr. Querying moving objects in secondo. In *MDM*, page 47, 2006.

[16] C. Düntgen, T. Behr, and R. H. Güting. Berlinmod: a benchmark for moving object databases. *VLDB Journal*, 18(6):1335–1368, 2009.

[17] H. G. Elmongui, M. F. Mokbel, and W. G. Aref. Continuous aggregate nearest neighbor queries. *GeoInformatica*, 17(1):63–95, 2013.

[18] M. Y. Eltabakh, R. Eltarras, and W. G. Aref. Space-partitioning trees in postgresql: Realization and performance. In *ICDE*, page 100, 2006.

[19] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation*. Prentice-Hall, 2000.

[20] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[21] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[22] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. A cost model for estimating the performance of spatial joins using R-trees. In *SSDBM*, pages 30–38, 1997.

[23] J. Jin, N. An, and A. Sivasubramaniam. Analyzing range queries on spatial data. In *ICDE*, pages 525–534, 2000.

[24] H.-P. Kriegel, P. Kunath, and M. Renz. R*-tree. In *Encyclopedia of GIS*, pages 987–992. 2008.

[25] Z. Li, K. C. K. Lee, B. Zheng, W.-C. Lee, D. L. Lee, and X. Wang. IR-Tree: An efficient index for geographic document search. *IEEE Trans. Knowl. Data Eng.*, 23(4):585–599, 2011.

[26] N. Mamoulis and D. Papadias. Selectivity estimation of complex spatial queries. In *SSTD*, pages 155–174, 2001.

[27] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD Conference*, pages 294–305, 1996.

[28] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD Conference*, pages 71–79, 1995.

[29] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., 2006.

[30] H. Samet and R. E. Webber. Storing a collection of polygons using quadtrees. *ACM Trans. Graph.*, 4(3):182–222, 1985.

[31] J. Sankaranarayanan, H. Samet, and A. Varshney. A fast all nearest neighbor algorithm for applications involving large point-clouds. *Computers & Graphics*, 31(2):157–174, 2007.

[32] Y. N. Silva, A. M. Aly, W. G. Aref, and P.-Å. Larson. Simdb: a similarity-aware database system. In *SIGMOD Conference*, pages 1243–1246, 2010.

[33] Y. N. Silva, W. G. Aref, and M. H. Ali. The similarity join database operator. In *ICDE*, pages 892–903, 2010.

[34] Y. N. Silva, W. G. Aref, P.-Å. Larson, S. S. Pearson, and M. H. Ali. Similarity queries: their conceptual evaluation, transformations, and processing. *VLDB Journal*, pages 1–26, 2012.

[35] I. Stanoi, D. Agrawal, and A. El Abbadi. Reverse nearest neighbor queries for dynamic databases. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 44–53, 2000.

[36] C. Sun, D. Agrawal, and A. El Abbadi. Selectivity estimation for spatial joins with geometric selections. In *EDBT*, pages 609–626, 2002.

[37] G. Tang, J. E. Córcoles, and N. Jing. NSJ: an efficient non-blocking spatial join algorithm. In *GIS*, pages 235–242, 2006.

[38] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *IEEE Trans. Knowl. Data Eng.*, 16(10):1169–1184, 2004.

[39] D. Wu, G. Cong, and C. S. Jensen. A framework for efficient spatial web object retrieval. *VLDB J.*, 21(6):797–822, 2012.

[40] W. Wu, F. Yang, C. Y. Chan, and K.-L. Tan. FINCH: evaluating reverse k-nearest-neighbor queries on location data. *PVLDB*, 1(1):1056–1067, 2008.

[41] B. Yao, F. Li, and P. Kumar. K nearest neighbor queries and kNN-joins in large relational databases (almost) for free. In *ICDE*, pages 4–15, 2010.