K Nearest Neighbor Queries and KNN-Joins in Large Relational Databases (Almost) for Free

Bin Yao, Feifei Li, Piyush Kumar
Computer Science Department, Florida State University, Tallahassee, FL, U.S.A.
{yao, lifeifei, piyush}@cs.fsu.edu

***Abstract*— Finding the k nearest neighbors (kNN) of a query point, or a set of query points (kNN-Join) are fundamental problems in many application domains. Many previous efforts to solve these problems focused on spatial databases or stand-alone systems, where changes to the database engine may be required, which may limit their application on large data sets that are stored in a relational database management system. Furthermore, these methods may not automatically optimize kNN queries or kNN-Joins when additional query conditions are specified. In this work, we study both the kNN query and the kNN-Join in a relational database, possibly augmented with additional query conditions. We search for relational algorithms that require no changes to the database engine. The straightforward solution uses the user-defined-function (UDF) that a query optimizer cannot optimize. We design algorithms that could be implemented by SQL operators without changes to the database engine, hence enabling the query optimizer to understand and generate the "best" query plan. Using only a small constant number of random shifts for databases in any fixed dimension, our approach guarantees to find the approximate kNN with only logarithmic number of page accesses in expectation with a constant approximation ratio and it could be extended to find the exact kNN efficiently in any fixed dimension. Our design paradigm easily supports the kNN-Join and updates. Extensive experiments on large, real and synthetic, data sets confirm the efficiency and practicality of our approach.**

I. Introduction

The k-Nearest Neighbor query (kNN) is a classical problem that has been extensively studied, due to its many important applications, such as spatial databases, pattern recognition, DNA sequencing and many others. A more general version is the kNN-Join problem [7], [8], [11], [31], [33]: Given a data set P and a query set Q, for each point $q \in Q$ we would like to retrieve its k nearest neighbors from points in P.

Previous work has concentrated on the use of spatial databases or stand-alone systems. In these solution methodologies, changes to the database engine maybe necessary; for example, new index structures or novel algorithms need to be incorporated into the engine. This requirement poses a limitation when the data set is stored in a database in which neither the spatial indices (such as the popular R-tree) nor the kNN-Join algorithms are available. Another limitation of the existing approaches that are outside the relational database, is the lack of support for query optimization, when additional query conditions are specified. Consider the query:

```
Retrieve the k nearest restaurants of q
with both Italian food and French wines.
```

The results of this query could be empty (in the case where no restaurants offer both Italian food and French wines) and the kNN retrieval is not necessary at all. It is well known that when the data is stored in a relational database, for various types of queries using only primitive SQL operators, the sophisticated query optimizer built inside the database engine will do an excellent job in finding a fairly good query execution plan [4].

These advantages of storing and processing data sets in a relational database motivate us to study the kNN query and the kNN-Join problem in a relational database environment. Our goal is to design algorithms that could be implemented by the primitive SQL operators and require no changes to the database engine. The benefits of satisfying such constraints are threefold. First, kNN based queries could be augmented with ad-hoc query conditions dynamically, and they are automatically optimized by the query optimizer, without updating the query algorithm each time for specific query conditions. Second, such an approach could be readily applied on existing commercial databases, without incurring any cost for upgrading or updating the database engine, e.g., to make it support spatial indices. We denote an algorithm that satisfies these two constraints as a *relational algorithm*. Finally, this approach makes it possible to support the kNN-Join efficiently. We would like to design algorithms that work well for data in multiple dimensions and easily support dynamic updates without performance degeneration.

The similar relational principle has been observed for other problems as well, e.g., approximate string joins in relational databases [15]. The main challenge in designing relational algorithms is presented by the fact that a query optimizer cannot optimize any user-defined functions (UDF) [15]. This rules out the possibility of using a UDF as a main query condition. Otherwise, the query plan always degrades to the expensive linear scan or the nested-loop join approach, which is prohibitive for large databases. For example,

```
SELECT TOP k * FROM Address A, Restaurant R
WHERE R.Type='Italian' AND R.Wine='French'
ORDER BY Euclidean(A.X, A.Y, R.X, R.Y)
```

In this query, X and Y are attributes representing the coordinates of an Address record or a Restaurant record. "Euclidean" is a UDF that calculates the Euclidean distance between two points. Hence, this is a kNN-Join query. Even though this query does qualify as a relational algorithm, the query plan will be a nested-loop join when there are restaurants satisfying both the "Type" and "Wine" constraints, since the

query optimizer cannot optimize the UDF "Euclidean". This implies that we may miss potential opportunities to fully optimize the queries. Generalizing this example to the kNN-query problem, the UDF-based approach will degrade to the expensive linear scan approach.

Our Contributions. In this work, we design relational algorithms that can be implemented using primitive SQL operators without the reliance on the UDF as a main query condition, so that the query optimizer can understand and optimize. We would like to support both approximate and exact kNN and kNN-Join queries. More specifically,

- We formalize the problems of kNN queries and kNN-Joins in a relational database (Section II).
- We provide a constant factor approximate solution based on the Z-order values from a small, constant number of randomly shifted copies of the database (Section III). We provide the theoretical analysis to show that by using only $O(1)$ random shifts for data in any fixed dimension, our approach gives an expected constant factor approximation (in terms of the radius of the k nearest neighbor ball) with only $\log N$ number of page accesses for the kNN query where N is the size of data set P. Our approach can be achieved with only primitive SQL operators.
- Using the approximate solution, we show how to get exact results for a kNN query efficiently (Section IV). Furthermore, we show that for certain types of data distributions, our exact solution also only uses $O(\log N)$ number of page accesses in any fixed dimension. The exact solution is also easy to implement using SQL.
- We extend our algorithms to kNN-Join queries, which can be achieved in relational databases without changes to the engines (Section V).
- We show that our algorithms easily support float values, data in arbitrary dimension and dynamic updates without any changes to the algorithms (Section VI).
- We present a comprehensive experimental study, that confirms the significant performance improvement of our approach, against the state of the art (Section VII).

In summary, we show how to find constant approximations for kNN queries in logarithm page accesses with a small constant number of random shifts in any fixed dimension; our approximate results lead to highly efficient search of the exact answers, with a simple post-processing of the results. Finally, our framework enables the efficient processing of kNN-Joins. We survey the related work in Section VIII.

II. Problem Formulation

Suppose that the data set P in a d dimensional space is stored in a relational table R_P. The coordinates of each point $p \in P$ are stored in d attributes $\{Y_1, \ldots, Y_d\}$. Each point could associate with other values, e.g., types of restaurants. These additional attributes are denoted by $\{A_1, \ldots, A_g\}$ for some value g. Hence, the schema of R_P is $\{pid, Y_1, \cdots, Y_d, A_1, \cdots, A_g\}$ where pid corresponds to the point id from P.

kNN queries. Given a query point q and its coordinates $\{X_1, \ldots, X_d\}$, let $\mathcal{A}$=kNN(q, R_P) be the set of k nearest neighbors of q from R_P and $|x, y|$ be the Euclidean distance between the point x and the point y (or the corresponding records for the relational representation of the points), then:

$$(\mathcal{A} \subseteq \mathrm{R}_P) \wedge (|\mathcal{A}| = k) \wedge (\forall a \in \mathcal{A}, \forall r \in \mathrm{R}_P - \mathcal{A}, |a, q| \leq |r, q|).$$

kNN-Join. In this case, the query is a set of points denoted by Q, and it is stored in a relational table R_Q. The schema of R_Q is $\{qid, X_1, \cdots, X_d, B_1, \cdots, B_h\}$. Each point q in Q is represented as a record s in R_Q, and its coordinates are stored in attributes $\{X_1, \ldots, X_d\}$ of s. Additional attributes of q, are represented by attributes $\{B_1, \cdots, B_h\}$ for some value h. Similarly, qid corresponds to the point id from Q. The goal of the kNN-Join query is to join each record s from R_Q with its kNN from R_P, based on the Euclidean distance defined by $\{X_1, \ldots, X_d\}$ and $\{Y_1, \ldots, Y_d\}$, i.e., for $\forall s \in Q$, we would like to produce pairs (s, r), for $\forall r \in k\mathrm{NN}(s, \mathrm{R}_p)$.

Other query conditions. Additional, ad-hoc query conditions could be specified by any regular expression, over $\{A_1, \cdots, A_g\}$ in case of kNN queries, or both $\{A_1, \cdots, A_g\}$ and $\{B_1, \cdots, B_h\}$ in case of kNN-Join queries. The relational requirement clearly implies that the query optimizer will be able to *automatically* optimize input queries based on these query conditions (see our discussion in Section VII).

Approximate k nearest neighbors. Suppose q's kth nearest neighbor from P is p^* and $r^* = |q, p^*|$. Let p be the kth nearest neighbor of q for some kNN algorithm A and $r^p = |q, p|$. Given $\epsilon > 0$ (or $c > 1$), we say that $(p, r^p) \in \mathbb{R}^d \times \mathbb{R}$ is a $(1+\epsilon)$-approximate (or c-approximate) solution to the kNN query $k\mathrm{NN}(q, P)$ if $r^* \leq r^p \leq (1+\epsilon) r^*$ (or $r^* \leq r^p \leq c r^*$ for some constant c). Algorithm A is called a $(1+\epsilon)$-approximation (or c-approximation) algorithm.

Similarly for kNN-Joins, an algorithm that finds a kth nearest neighbor point $p \in P$ for each query point $q \in Q$, that is at least a $(1+\epsilon)$-approximation or c-approximation w.r.t $k\mathrm{NN}(q, P)$ is a $(1+\epsilon)$-approximate or c-approximate kNN-Join algorithm. The result by this algorithm is referred to as a $(1+\epsilon)$-approximate or c-approximate join result.

Additional notes. The default value for d in our running examples is two, but, our proofs and algorithms are presented for any fixed dimension, d. We focus on the case where the coordinates for points are always integers. Handling floating points coordinates is discussed in Section VI. Our approach easily supports updates, which is also discussed in Section VI. The number of records in R_P and R_Q are denoted by $N = |\mathrm{R}_P|$ and $M = |\mathrm{R}_Q|$ respectively. We assume that each page can store maximally B records from either R_P or R_Q, and the fan-out of a B+ tree is f. Without loss of generality, we assume that points in P are all in unique locations. For the general case, our algorithms can be easily adapted by breaking the ties arbitrarily.

III. Approximation by Random Shifts

The *z-value* of a point is calculated by interleaving the binary representations of its coordinate values from the most

significant bit (msb) to the least significant bit (lsb). For example, given a point $(2, 6)$ in a 2-d space, the binary representation of its coordinates is $(010, 110)$. Hence, its z-value is $011100 = 28$. The Z-order curve for a set of points P is obtained by connecting the points in P by the numerical order of their z-values and this produces the recursively Z-shaped curve. A key observation for the computation of z-value is that, it only requires simple bit-shift operations which are readily available or easily achievable in most commercial database engines. For a point p, z_p denotes its z-value.

Our idea utilizes the z-values to map points in a multi-dimensional space into one dimension, and then translate the kNN search for a query point q into one dimensional range search on the z-values around the q's z-value. In most cases, z-values preserve the spatial locality and we can find q's kNN in a close neighborhood (say γ positions up and down) of its z-value. However, this is not always the case. In order to get a theoretical guarantee, we produce α, independent, randomly shifted copies of the input data set P and repeat the above procedure for each randomly shifted version of P.

Specifically, we define the "random shift" operation, as shifting all data points in P by a random vector $\overrightarrow{v} \in \mathbb{R}^d$. This operation is simply $p + \overrightarrow{v}$ for all $p \in P$, and denoted as $P + \overrightarrow{v}$. We independently at random generate α number of vectors $\{\overrightarrow{v_1}, \ldots, \overrightarrow{v_\alpha}\}$ where $\forall i \in [1, \alpha]$, $\overrightarrow{v_i} \in \mathbb{R}^d$. Let $P^i = P + \overrightarrow{v_i}$, $P^0 = P$ and $\overrightarrow{v_0} = \overrightarrow{0}$. For each P^i, its points are sorted by their z-values. Note that the random shift operation is executed only once for a data set P and used for subsequent queries. Next, for a query point q and a data set P, let z_p be the successor z-value of z_q among all z-values for points in P. The *γ-neighborhood of q* is defined as the γ points up and down next to z_p. For the special case, when z_p does not have γ points before or after, we simply take enough points after or before z_p to make the total number of points in the γ-neighborhood to be $2\gamma + 1$, including z_p itself. Our kNN query algorithm essentially finds the γ-neighborhood of the query point $q^i = q + \overrightarrow{v_i}$ in P^i for $i \in [0, \alpha]$ and select the final top k from the points in the unioned $(\alpha+1)$ γ-neighborhoods, with a maximum $(\alpha + 1)(2\gamma + 1)$ number of distinct points. We denote this algorithm as the z$^\chi$-kNN algorithm and it is shown in Algorithm 1. It is important to note that in Line 5, we obtain the original point from its shifted version if it is selected to be a candidate in one of the γ-neighborhoods. This step simplifies the final retrieval of the kNN from the candidate set C. It also implies that the candidate sets C^i's may contain duplicate points, i.e., a point may be in the γ-neighborhood of the query point in more than one randomly shifted versions.

The z$^\chi$-kNN is clearly very simple and can be implemented efficiently using only $(\alpha+1)$ one-dimensional range searches, each requires only logarithmic IOs w.r.t the number of pages occupied by the data set (N/B) if γ is some constant. More importantly, we can show that, with $\alpha = O(1)$ and $\gamma = O(k)$, z$^\chi$-kNN gives a constant approximation kNN result in any fixed dimension d, with only $O(\log_f \frac{N}{B} + k/B)$ page accesses.

In fact, we can show these results with just $\alpha = 1$ and $\gamma = k$. In this case, let P' be a randomly shifted version of the point set P, q' be the correspondingly shifted query point and $\mathcal{A}_{q'}$ be the k nearest neighbors of q' in P'. Note that $\mathcal{A}_{q'} = \mathcal{A}$. Let points in P' be $\{p_1, p_2, \ldots, p_N\}$ and they are sorted by their z-values. The successor of $z_{q'}$ w.r.t the z-values in P' is denoted as p_τ for some $\tau \in [1, N]$. Clearly, the candidate set C in this case is simply $C = \{p_{\tau-k}, \ldots, p_{\tau+k}\}$. Without loss of generality, we assume that both $\tau - k$ and $\tau + k$ are within the range of $[1, N]$, otherwise we can simply take additional points after or before the successor as explained above in the algorithm description. We use $B(c, r)$ to denote a ball with center c and radius r and let $rad(p, S)$ be the distance from the point p to *the farthest point* in a point set S. We first show the following lemmas in order to claim the main theorem.

Algorithm 1: z$^\chi$-kNN (point q, point sets $\{P^0, \ldots, P^\alpha\}$)

```
1 Candidates C = ∅;
2 for i = 0, ..., α do
3     Find z_p^i as the successor of z_{q+v_i} in P^i;
4     Let C^i be γ points up and down next to z_p^i in P^i;
5     For each point p in C^i, let p = p − v_i;
6     C = C ⋃ C^i;
7 Let A^χ = kNN(q, C) and output A^χ.
```

Lemma 1 *Let M be the smallest box, centered at q' containing $\mathcal{A}_{q'}$ and with side length 2^i (where i is assumed w.l.o.g to be an integer > 0) which is randomly placed in a quadtree T (associated with the Z-order). If the event $\mathcal{E}_j$ is defined as M being contained in a quadtree box M_T with side length 2^{i+j}, and M_T is the smallest such quadtree box, then*

$$\Pr(\mathcal{E}_j) \leq \left(1 - \frac{1}{2^j}\right)^d \frac{d^{j-1}}{2^{\frac{j^2-j}{2}}}$$

Proof: The proof is in the paper's full version [32]. ■

Lemma 2 *The z$^\chi$-kNN algorithm gives an approximate k-nearest neighbor ball $B(q', rad(q', C))$ using q' and P' where $rad(q', C)$ is at most the side length of M_T. Here M_T is the smallest quadtree box containing M, as defined in Lemma 1.*

Proof: z$^\chi$-kNN scans at least $p_{\tau-k} \ldots p_{\tau+k}$, and picks the top k nearest neighbors to q among these candidates. Let a be the number of points between p_τ and the point with the largest z-value in $B(q', rad(q', \mathcal{A}_{q'}))$, the exact k-nearest neighbor ball of q'. Similarly, let b be the number of points between p_τ and the point with the smallest z-value in $B(q', rad(q', \mathcal{A}_{q'}))$. Clearly, $a + b = k$. Note that $B(q', rad(q', \mathcal{A}_{q'})) \subset M$ is contained inside M_T, hence the number of points inside $M_T \geq k$. Now, $p_\tau \ldots p_{\tau+k}$ must contain a points inside M_T. Similarly, $p_{\tau-k} \ldots p_\tau$ must contain at least b points from M_T. Since we have collected at least k points from M_T, $rad(q', C)$ is upper bounded by the side length of M_T. ■

Lemma 3 *M_T is only constant factor larger than M in expectation.*

Proof: The expected side length of M_T is:

$$E[2^{i+j}] \leq \sum_{j=1}^{\infty} 2^{i+j} \Pr(\mathcal{E}_j) \leq 2^i \sum_{j=1}^{\infty} (1 - \frac{1}{2^j})^d d^{j-1} 2^{\frac{3j-j^2}{2}},$$

where Lemma 1 gives $\Pr(\mathcal{E}_j)$. Using Taylor's approximation:

$$\left(1 - \frac{1}{2^j}\right)^d \leq \left(1 - d2^{-j}\left(1 + 2^{-1-j}\right) + d^2 2^{-2j-1}\right)$$

and substituting it in the expectation calculation, we can show that $E[2^{i+j}]$ is $O(2^i)$. The detail is in the full version [32]. ■

These lemmas lead to the main theorem for z$^\chi$-kNN.

Theorem 1 *Using $\alpha = O(1)$, or just one randomly shifted copy of P, and $\gamma = O(k)$, z$^\chi$-kNN guarantees an expected constant factor approximate kNN result with $O(\log_f \frac{N}{B} + k/B)$ number of page accesses.*

Proof: The IO cost follows directly from the fact that the one dimensional range search used by z$^\chi$-kNN takes $O(\log_f \frac{N}{B} + k/B)$ number of page accesses with a B-tree index. Let q be the point for which we just computed the approximate k-nearest neighbor ball $B(q', rad(q', C))$. From Lemma 2, we know that $rad(q', C)$ is at most the side length of M_T. From Lemma 3 we know that the side length of M_T is at most constant factor larger than $rad(q', \mathcal{A}_{q'})$ in P', in expectation. Note that $rad(q, \mathcal{A}_q)$ in P equals $rad(q', \mathcal{A}_{q'})$ in P'. Hence our algorithm, in expectation, computes an approximate k-nearest neighbor ball that is only a constant factor larger than the true k-nearest neighbor ball. ■

Algorithm 1 clearly indicates that z$^\chi$-kNN only relies on one dimensional range search as its main building block (Line 4). One can easily implement this algorithm with just SQL statements over tables that store the point sets $\{P^0, \ldots, P^\alpha\}$. We simply use the original data set P to demonstrate the translation of the z$^\chi$-kNN into SQL. Specifically, we pre-process the table R_P so that an additional attribute *zval* is introduced. A record $r \in \mathrm{R}_P$ uses $\{Y_1, \ldots, Y_d\}$ to compute its z-value and store it as the *zval*. Next, a *clustered B+-tree index* is built on the attribute *zval* over the table R_P. This also implies that records in R_P are sorted by the *zval* attribute. For a given query point q, we first calculate its z-value (denoted as *zval* as well) and then, we find the *successor* record of q in R_P, based on the *zval* attribute of R_p and q. In the sequel, we assume that such a successor record always exists. In the special case, when q's z-value is larger than all the z-values in R_P, we simply take its predecessor instead. This technical detail will be omitted. The successor can be found by:

```
SELECT TOP 1 * FROM RP WHERE RP.zval ≥ q.zval
```

Note that `TOP` is a ranking operator that becomes part of the standard SQL in most commercial database engines. For example, Microsoft SQL Server 2005 has the `TOP` operator available. Recent versions of Oracle, MySQL, and DB2 have the `LIMIT` operator which has the same functionality. Since table R_P has a clustered B+ tree on the *zval* attribute, this query has only a logarithmic cost (to the size of R_P) in terms of the IOs, i.e., $O(\log_f \frac{N}{B})$. Suppose the successor record is r_s, in the next step, we retrieve all records that locate within γ positions away from r_s. This can be done as:

```
SELECT * FROM
( SELECT TOP γ * FROM RP WHERE RP.zval
  > rs.zval ORDER BY RP.zval ASC
  UNION
  SELECT TOP γ * FROM RP WHERE RP.zval
  < rs.zval ORDER BY RP.zval DESC ) AS C
```

Again, due to the clustered B+ tree index on the *zval*, this query is essentially a sequential scan around r_s, with a query cost of $O(\log_f \frac{N}{B} + \frac{\gamma}{B})$. The first `SELECT` clause of this query is similar to the successor query as above. The second `SELECT` clause is also similar but the ranking is in descending order of the *zval*. However, even in the second case, the query optimizer is robust enough to realize that with the clustered index on the *zval*, no ranking is required and it simply sequentially scans γ records backwards from r_s.

The final step of our algorithm is to simply retrieve the top k records from these $2\gamma + 1$ candidates (including r_s), based on their Euclidean distance to the query point q. Hence, the UDF 'Euclidean' is only applied in the last step with $2\gamma + 1$ number of records. The complete algorithm can be expressed in one SQL statement as follows:

```
1  SELECT TOP k * FROM
2  ( SELECT TOP γ+1 * FROM RP,
3      ( SELECT TOP 1 zval FROM RP
4        WHERE RP.zval ≥ q.zval
5        ORDER BY RP.zval ASC ) AS T
6    WHERE RP.zval≥T.zval
7    ORDER BY RP.zval ASC
8      UNION
9    SELECT TOP γ * FROM RP
10    WHERE RP.zval < T.zval
11   ORDER BY RP.zval DESC ) AS C
12 ORDER BY Euclidean(q.X1,q.X2,C.Y1,C.Y2)   (Q1)
```

For all SQL environments, line 3 to 5 need to be copied into the `FROM` clause in line 9. We omit this from the SQL statement to shorten the presentation. In the sequel, for similar situations in all queries, we choose to omit this. Essentially, line 2 to line 11 in `Q1` select the γ-neighborhood of q in P, which will be the candidate set for finding the approximate kNN of q using the Euclidean UDF by line 1 and 12.

In general, we can pre-process $\{P^1, \ldots, P^\alpha\}$ similarly to get the randomly shifted tables $\{\mathrm{R}_P^1, \ldots, \mathrm{R}_P^\alpha\}$ of R_P, the above procedure could be then easily repeated and the final answer is the top k selected based on applying the Euclidean UDF over the union of the $(\alpha+1)$ γ-neighborhoods retrieved. The whole process could be done in just one SQL by repeating line 2 to line 11 on each randomly shifted table and unioning them together with the SQL operator UNION.

In the work by Liao et al. [23], using Hilbert-curves a set of $d + 1$ deterministic shifts can be done to guarantee a constant factor approximate answer but in this case the space requirement is high, $O(d)$ copies of the entire point set is required, and the associated query cost is increased by a multiplicative factor of d. In contrast, our approach only requires $O(1)$ shifts for any dimension and can be adapted to yield *exact* answers. In practice, we just use the optimal value of $\alpha = \min(d, 4)$ that we found experimentally to

give the best results for any fixed dimension. In addition, z-values are much simpler to calculate than the Hilbert values, especially in higher dimensions, making it suitable for the SQL environment. Finally, we emphasize that randomly shifted tables $\{\mathrm{R}_P^1, \ldots, \mathrm{R}_P^\alpha\}$ are generated only once and could be used for multiple, different queries.

IV. Exact kNN Retrieval

The z^χ-*kNN* algorithm finds a good approximate solution in $O(\log_f \frac{N}{B} + k/B)$ number of IOs. We denote the kNN result from the z^χ-*kNN* algorithm as $\mathcal{A}^\chi$. One could further retrieve the exact kNN results based on $\mathcal{A}^\chi$. A straightforward solution is to perform a range query using the approximate *kth nearest neighbor ball* of $\mathcal{A}^\chi$, $\mathcal{B}(\mathcal{A}^\chi)$. It is defined as the ball centered at q with the radius $rad(q, \mathcal{A}^\chi)$, i.e., $\mathcal{B}(\mathcal{A}^\chi) = B(q, rad(q, \mathcal{A}^\chi))$. Clearly, $rad(q, \mathcal{A}^\chi) \geq r^*$. Hence, the exact kNN points are enclosed by $\mathcal{B}(\mathcal{A}^\chi)$. This implies that we can find the exact kNN for q by:

```
SELECT TOP k * FROM R_P
WHERE Euclidean(q.X1,q.X2,R_P.Y1,R_P.Y2)≤ rad(p,A^χ)
ORDER BY Euclidean(q.X1,q.X2,R_P.Y1,R_P.Y2)  (Q2)
```

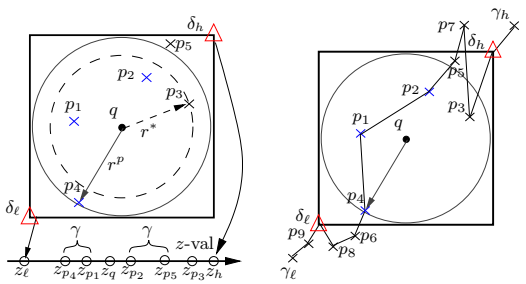
This query does reduce the number of records participated in the final ranking procedure with the Euclidean UDF. However, it still needs to scan the entire table R_P to calculate the Euclidean UDF first for each record, thus becomes very expensive. Fortunately, we can again utilize the z-values to find the exact kNN based on $\mathcal{A}^\chi$ much more efficiently.

We define the kNN box for $\mathcal{A}^\chi$ as the smallest box that fully encloses $\mathcal{B}(\mathcal{A}^\chi)$, denoted as $M(\mathcal{A}^\chi)$. Generalizing this notation, let $\mathcal{A}_i^\chi$ be the kNN result of z^χ-*kNN* when it is applied only on table R_P^i and $M(\mathcal{A}_i^\chi)$ and $\mathcal{B}(\mathcal{A}_i^\chi)$ be the corresponding kNN box and kth nearest neighbor ball from table R_P^i. The exact kNN results from all table R_P^i's are the same and they are always equal to $\mathcal{A}$. In the sequel, when the context is clear, we omit the subscript i from $\mathcal{A}_i^\chi$.

An example of the kNN box is shown in Figure 1(a). In this case, the z^χ-*kNN* algorithm on this table R_P returns the kNN result as $\mathcal{A}^\chi = \{p_1, p_2, p_4\}$ for $k = 3$ and p_4 is the kth nearest neighbor. Hence, $\mathcal{B}(\mathcal{A}^\chi) = B(q, |q, p_4|)$, shown as the solid circle in Figure 1(a). The kNN box $M(\mathcal{A}^\chi)$ is defined by the lower-left and right-upper corner points δ_ℓ and δ_h, i.e., the Δ points in Figure 1(a). As we have argued above, the exact kNN result must be enclosed by $\mathcal{B}(\mathcal{A}^\chi)$, hence, also enclosed by $M(A^\chi)$. In this case, the exact kNN result is $\{p_1, p_2, p_3\}$ and the dotted circle in Figure 1(a) is the exact kth nearest neighbor ball $\mathcal{B}(\mathcal{A})$.

Lemma 4 *For a rectangular box M and its lower-left and upper-right corner points δ_ℓ, δ_h, $\forall p \in M$, $z_p \in [z_\ell, z_h]$, where z_p stands for the z-value of a point p and z_ℓ, z_h correspond to the z-values of δ_ℓ and δ_h respectively (See Figure 1(a)).*

Proof: Consider 2-d points, let $p.X$ and $p.Y$ be the coordinate values of p in x-axis and y-axis respectively. By $p \in M$, we have $p.X \in [\delta_\ell.X, \delta_h.X]$ and $p.Y \in [\delta_\ell.Y, \delta_h.Y]$. Since the z-value of a point is obtained by shuffling bits of its coordinate values from the msb to the lsb in an alternating fashion, this immediately implies that $z_p \in [z_\ell, z_h]$. The case for higher dimensions is similar. ■



(a) kNN box. (b) $\mathcal{A}^\chi = \mathcal{A}$.

Fig. 1. kNN box: definition and exact search.

Lemma 4 implies that the z-values of all exact kNN points will be bounded by the range $[z_\ell, z_h]$, where z_ℓ and z_h are the z-values for the δ_ℓ and δ_h points of $M(\mathcal{A}^\chi)$, in other words:

Corollary 1 *Let z_ℓ and z_h be the z-values of δ_ℓ and δ_h points of $M(\mathcal{A}^\chi)$. For all $p \in \mathcal{A}$, $z_p \in [z_\ell, z_h]$.*

Proof: By $\mathcal{B}(\mathcal{A}) \subset M(\mathcal{A}^\chi)$ and Lemma 4. ■

Consider the example in Figure 1(a), Corollary 1 guarantees that z_{p_i} for $i \in [1, 5]$ and z_q are located between z_ℓ and z_h in the one-dimensional z-value axis. The z^χ-*kNN* essentially searches γ number of points around both the left and the right of z_q in the z-value axis, for α number of randomly shifted copies of P, including P itself. However, as shown in Figure 1(a), it may still miss some of the exact kNN points. Let γ_ℓ and γ_h denote the left and right γ-th points respectively for this search. In this case, let $\gamma = 2$, z_{p_3} is outside the search range, specifically, $z_{p_3} > z_{\gamma_h}$. Hence, z^χ-*kNN* could not find the exact kNN result. However, given Corollary 1, an immediate result is that one can guarantee to find the exact kNN result by considering all points with their z-values between z_ℓ and z_h of $M(\mathcal{A}^\chi)$. In fact, if $z_{\gamma_\ell} \leq z_\ell$ and $z_{\gamma_h} \geq z_h$ in *at least one* of the table R^i's for $i = 0, \ldots, \alpha$, we know for sure that z^χ-*kNN* has successfully retrieved the exact kNN result; otherwise it may have missed some exact kNN points. The case when z_ℓ and z_h are both contained by z_{γ_ℓ} and z_{γ_h} of the $M(\mathcal{A}^\chi)$ in one of the randomly shifted tables is illustrated in Figure 1(b). In this case, in one of the $(\alpha+1)$ randomly shifted tables, the z-order curve passes through z_{γ_ℓ} first before any points in $M(\mathcal{A}^\chi)$ (i.e., $z_{\gamma_\ell} \leq z_\ell$) and it comes to z_{γ_h} after all points in $M(\mathcal{A}^\chi)$ have been visited (i.e., $z_{\gamma_h} \geq z_h$). As a result, the candidate points considered by z^χ-*kNN* include every point in $\mathcal{A}$ and the kNN result from the algorithm z^χ-*kNN* will be exact, i.e., $\mathcal{A}^\chi = \mathcal{A}$.

When this is not the case, i.e., either $z_\ell^i < z_{\gamma_\ell}^i$ or $z_h^i > z_{\gamma_h}^i$ or both *in all tables* R_P^i's for $i = 0, \ldots, \alpha$, $\mathcal{A}^\chi$ might not be equal to $\mathcal{A}$. To address this issue, we first choose one of the table R_P^j such that its $M(\mathcal{A}_j^\chi)$ contains the least number of points among kNN boxes from all tables; then the candidate points for the exact kNN search only need to include all points contained by this box, i.e., $M(\mathcal{A}_j^\chi)$ from the table R_P^j.

To find the exact kNN from the box $M(\mathcal{A}_j^\chi)$, we utilizing Lemma 4 and Corollary 1. In short, we calculate the z_ℓ^j and

Algorithm 2: z-kNN (point q, point sets $\{P^0, \ldots, P^\alpha\}$)

1 Let $\mathcal{A}_i^\chi$ be kNN(q, C^i) where C^i is from Line 4 in the z$^\chi$-kNN algorithm;
2 Let z_ℓ^i and z_h^i be the z-values of the lower-left and upper-right corner points for the box $M(\mathcal{A}_i^\chi)$;
3 Let $z_{\gamma_\ell}^i$ and $z_{\gamma_h}^i$ be the lower bound and upper bound of the z-values z$^\chi$-kNN has searched to produce C^i;
4 **if** $\exists i \in [0, \alpha]$, *s.t.* $z_{\gamma_\ell}^i \le z_\ell^i$ *and* $z_{\gamma_h}^i \ge z_h^i$ **then**
5 Return $\mathcal{A}^\chi$ by z$^\chi$-kNN as $\mathcal{A}$;
6 **else**
7 Find $j \in [0, \alpha]$ such that the number of points in R_P^j with z-values in $\in [z_\ell^j, z_h^j]$ is minimized;
8 Let C_e be those points and return $\mathcal{A} = k\text{NN}(q, C_e)$;

z_h^j of this box and do a range query with $[z_\ell^j, z_h^j]$ on the *zval* attribute in table R_P^j. Since there is a clustered index built on the *zval* attribute, this range query becomes a sequential scan in $[z_\ell^j, z_h^j]$ which is very efficient. It essentially involves logarithmic IOs to access the path from the root to the leaf level in the B+ tree, plus some sequential IOs linear to the number of points between z_ℓ^j and z_h^j in table R_P^j. The next lemma is immediate based on the above discussion. Let z_s^i be the successor z-value to q^i's z-value in table R_P^i where $q^i = q + \overrightarrow{v_i}$, z_p^i be the z-value of a point p in table R_P^i and L^i be the number of points in the range $[z_\ell^i, z_h^i]$ from table R_P^i.

Lemma 5 *For the algorithm* z$^\chi$-kNN*, if there exists at least one* $i \in \{0, \ldots, \alpha\}$*, such that* $[z_\ell^i, z_h^i] \subseteq [z_{\gamma_\ell}^i, z_{\gamma_h}^i]$*, then* $\mathcal{A}^\chi = \mathcal{A}$*; otherwise, we can find* $\mathcal{A} \subseteq [z_\ell^j, z_h^j]$ *for some table* R_P^j *where* $j \in \{0, \ldots, \alpha\}$ *and* $L^j = \min\{L^0, \ldots, L^\alpha\}$.

These discussion leads to a simple algorithm for finding the exact kNN based on z$^\chi$-kNN and it is shown in Algorithm 2. We denote it as the z-kNN algorithm.

Figure 2 illustrates the z-kNN algorithm. In this case, $\alpha = 1$, In both table R_P^0 and R_P^1, $[z_{\gamma_\ell}, z_{\gamma_h}]$ does not fully contain $[z_\ell, z_h]$. Hence, algorithm *z$^\chi$-kNN* does not guarantee to return the exact kNN. Among the two tables, $[z_\ell^1, z_h^1]$ contains less number of points than $[z_\ell^0, z_h^0]$ (in Figure 2, $L^1 < L^0$), hence we do the range query using $[z_\ell^1, z_h^1]$ in table R_P^1 and apply the "Euclidean" UDF on the records returned by this range query to select the final, exact kNN records. Algorithm z-kNN can be achieved using SQL alone. The first step is to check if $\mathcal{A}^\chi = \mathcal{A}$. This is equivalent to check if $[z_\ell^i, z_h^i] \subseteq [z_{\gamma_\ell}^i, z_{\gamma_h}^i]$ for some i. Note that we can calculate the coordinate values of δ_ℓ^i and δ_h^i for any $M(\mathcal{A}_i^\chi)$ easily with addition and subtraction, given q^i and $\mathcal{A}_i^\chi$, and convert them into the z-values (z_ℓ^i and z_h^i) in SQL (we omit this detail for brevity). That said, this checking on table R_P^i can be done via:

```
SELECT COUNT(*) FROM R_P^i AS R
WHERE R.zval ≥ z_ℓ^i AND R.zval ≤ z_h^i
    AND R.pid NOT IN ( SELECT pid FROM R_P^i AS R1
    WHERE R1.zval ≥ z_γℓ^i AND R1.zval ≤ z_γh^i ) (Q3)
```

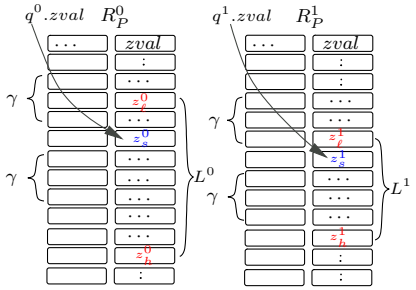


Fig. 2. Exact search of kNN.

If this count equals 0, then $[z_\ell^i, z_h^i] \subseteq [z_{\gamma_\ell}^i, z_{\gamma_h}^i]$. One can easily create one SQL statement to do this checking for all tables. If there is at least one table with a count equal to 0 among $\text{R}_P^0, \ldots, \text{R}_P^\alpha$, then we can can safely return $\mathcal{A}^\chi$ as the exact kNN result. Otherwise, we continue to the next step. We first select the table (say R_P^j) with the smallest L^i value. This is done by `(Q4)`, in which we find the value j. Then we select the final kNN result from R_P^j among the records with z-values between $[z_\ell^j, z_h^j]$ via `(Q5)`.

```
SELECT TOP 1 ID FROM
( SELECT 0 AS ID, COUNT(*) AS L FROM R_P^0 AS R^0
  WHERE R^0.zval ≥ z_ℓ^0 AND R^0.zval ≤ z_h^0
     UNION  ···  UNION
  SELECT α AS ID, COUNT(*) AS L FROM R_P^α AS R^α
  WHERE R^α.zval ≥ z_ℓ^α AND R^α.zval ≤ z_h^α
) AS T ORDER BY T.L ASC                          (Q4)

SELECT TOP k * FROM  R_P^j   //The ID from Q4 is j
WHERE R_P^j.zval ≥ z_ℓ^j AND R_P^j.zval ≤ z_h^j
ORDER BY Euclidean(q^j.X1,q^j.X2,R_P^j.Y1,R_P^j.Y2) (Q5)
```

One can combine `(Q3)`, `(Q4)` and `(Q5)` to get the exact kNN result based on the approximate kNN result given by `(Q1)` (the *z$^\chi$-kNN* algorithm).

Finally, we would like to highlight that theoretically, for many practical distributions, the *z-kNN* algorithm still achieves $O(\log_f \frac{N}{B} + k/B)$ number of page accesses to report the *exact* kNN result in any fixed dimension. When $\mathcal{A}^\chi = \mathcal{A}$, this result is immediate by Theorem 1. When $\mathcal{A}^\chi \neq \mathcal{A}$, the key observation is that the number of "false positives" in the candidate set C_e (line 8 in Algorithm 2) is bounded by some constant $O(k)$ for many practical data distributions. Hence, the number of candidate points that the *z-kNN* algorithm needs to check is still $O(k)$, the same as the *z$^\chi$-kNN* algorithm. Referring back to Figure 1(b), the false positives in a kNN box $M(\mathcal{A}_i^\chi)$ are defined as those points p such that $p \notin M(\mathcal{A}_i^\chi)$ but $z_p \in [z_\ell^i, z_h^i]$. For example, in Figure 1(b), the false positives are $\{p_8, p_6, p_7\}$. Note that p_9 is not a false positive w.r.t this kNN box as it is beyond the range $[z_\ell^i, z_h^i]$.

Let $\mathbb{P}$ be a fixed distribution of points in a fixed d-dimensional space, i.e., d is considered as a constant. Let P be i.i.d. from $\mathbb{P}$ and its size be $N \gg k \ge 1$. We will call this distribution a *Doubling Distribution* if it has the following property: Let S be a d dimensional ball with center $p_i \in P$ and radius r that contains k points. Then, the d dimensional ball S' with center p_i and radius $2r$ has at most νk points, for some $\nu = O(1)$. This is a similar restriction to the doubling metric restriction on metric spaces and has been used before [21].

Note that many distributions of points occurring in real data sets, including uniform distribution satisfy this property.

Theorem 2 *For doubling distributions, the expected number of false positives for a kNN box $M(\mathcal{A}_i^\chi)$ for all $i \in \{0, \ldots, \alpha\}$ is $O(k)$; the number of points that are fully enclosed by $M(\mathcal{A}_i^\chi)$ is also $O(k)$.*

Proof: Without loss of generality, let ι be any fixed value from 0 to α. Let M_T be the smallest quadtree box that contains $M(\mathcal{A}_\iota^\chi)$. Let $d = 1$. We will now show that the number of points in M_T is $O(k)$. The expected number of points in $M(\mathcal{A}_\iota^\chi)$ is at most $k\nu\frac{1}{2} + k\nu^2\frac{1}{2}\frac{3}{4} + k\nu\frac{1}{2}\frac{1}{4}\frac{7}{8} + \ldots \leq k\sum_{j=1}^{\infty}\frac{\nu^j}{2^{(j^2+1)/2}} = O(k)$, since $\nu = O(1)$. A similar argument shows that as the dimension increases (but is still $O(1)$), the expected number of points in $M(\mathcal{A}_\iota^\chi)$ is still $O(k)$. A two level expectation argument shows that the expected number of points in M_T is still $O(k)$. The details of this calculation are omitted for brevity. The points in M_T are consecutive in z-values and the Z-order curve enters the lower-left corner and sweeps through the entire M_T before it leaves through the upper right corner of M_T. Since, $M(\mathcal{A}_\iota^\chi) \subset M_T$, the curve passing through the lower left corner of $M(\mathcal{A}_\iota^\chi)$ and ending at the upper right corner of $M(\mathcal{A}_\iota^\chi)$ can not go out of M_T. This implies that all the false positives are contained in M_T and hence the expected number of false positives is also upper bounded by $O(k)$ in expectation. ■

Corollary 2 *For doubling distributions, the* z-kNN *algorithm, using $O(1)$ number of random shifts, retrieves the exact kNN result with $O(\log_f \frac{N}{B} + k/B)$ number of page accesses for data in any fixed dimension.*

Our Theorem requires just 1 random shift. In practice, we use several random shifts to amplify the probability of getting smaller M_T sizes (which in turn reduces the query cost), at the expense of increasing storage cost. We explore this trade-off in our experiments.

V. kNN-Join and Distance Based θ-Join

An important feature for our approach is that we can easily and efficiently support join queries. The basic principle of finding the k nearest neighbors stays the same for the kNN-Join query over two tables R_Q and R_P. However, the main challenge is to achieve this using a single SQL statement. We still generate $\mathrm{R}_P^0, \ldots, \mathrm{R}_P^\alpha$ in the same fashion. Concentrating on the approximate solution, we need to perform the similar procedure as shown in Section III to joining two tables (R_Q and R_P^i's). The general problem is to join each individual record s_i from R_Q to $2\gamma + 1$ number of records from R_P around s_i's successor (based on the z-value) record $r_s(s_i)$ in R_P; and then for each such group (s_i and the $2\gamma + 1$ records around $r_s(s_i)$) we need to select the top-k records based on their Euclidean distances to s_i. A simple approach is to use a store procedure to implement this idea, i..e, for each record from R_Q, we execute the *z^χ-kNN* query from Section III.

If one would like to implement this join with just one SQL statement, the observation is that the second step above is equivalent to retrieving top k records in each group based on some ranking functions and grouping conditions. This has been addressed by all commercial database engines. For example, in Microsoft SQL Server, this is achieved by the `RANK() OVER (PARTITION BY ... ORDER BY ...)` clause. Conceptually, this clause assigns a rank number to each record involved in one partition or group according to its sorted order in that partition. Hence, we could simply select the tuple with the rank number that is less than or equal to k from each group. Oracle, MySQL and DB2 all have their own operators for similar purposes.

We denote this query as the *z^χ-kNNJ* algorithm. It is *important to note* that in some engines, the query optimizer may not do a good job in optimizing the top-k query for each group. An alternative approach is to implement the same idea with a store procedure, By the same argument as shown in Section III, the following result is immediate.

Lemma 6 *Using $\alpha = O(1)$ and $\gamma = O(k)$, the* z$^\chi$-kNNJ *algorithm guarantees an expected constant approximate kNN-Join result in $O\left(\frac{M}{B}(\log_f \frac{N}{B} + \frac{k}{B})\right)$ number of page accesses.*

We could extend the exact *z-kNN* algorithm from Section IV to derive SQL statements as the *z-kNNJ* algorithm for the exact kNN-Join. We omit it for brevity.

Our approach is quite flexible and supports a variety of interesting queries. In particular, we demonstrate how it could be adopted to support the distance based θ-Join query, denoted as the *θ-DJoin*. Our idea for the *θ-DJoin* is similar to the principle adopted in [24]. This query joins each record $s_i \in \mathrm{R}_Q$ with the set $\mathcal{A}^\theta(s_i)$ which contains all records $r_j \in \mathrm{R}_P$ such that $|s_i, r_j| \leq \theta$ for some specified θ value. Suppose s_i corresponds to a query point q. An obvious observation is that the *furthest* point (or record) to q in $\mathcal{A}^\theta(s_i)$ always has a distance that is at most θ. Hence, the ball $\mathcal{B}(q, \theta)$ completely encloses all points from $\mathcal{A}^\theta(s_i)$. Let the θ-box for a record s_i be the smallest box that encloses $\mathcal{B}(q, \theta)$ and denote it as $M(\mathcal{A}^\theta(s_i))$. Clearly, all points from $\mathcal{A}^\theta(s_i)$ are also fully enclosed by $M(\mathcal{A}^\theta(s_i))$. By Lemma 4, we have for $\forall p \in \mathcal{A}^\theta(s_i)$, $z_p \in [z_\ell, z_h]$, here z_ℓ and z_h are the z-values of the bottom-left and top-right corner points of the box $M(\mathcal{A}^\theta(s_i))$. This becomes exactly the same problem as the exact kNN search and similar ideas from z-kNN could then be applied.

VI. Float Values, Higher Dimensions and Updates

Our method easily supports the floating point coordinates by computing the z-values explicitly for floating point coordinates in the pre-processing phase. This is done via the same bit-interleaving operation. The only problem with this approach is that the number of bits required for interleaving d-single precision co-ordinates is $256d$, assuming IEEE 754 floating point representation. To avoid such a long string we can use the following trick: We fix the length of the interleaved Z-order bits to be at most μd bits, where μ is a small constant. We scale the input data such that all coordinates lie between $(0, 1)$. We then only interleave the first μ bits of each coordinate after the

decimal point. For most practical data sets, $\mu < 32$ (equivalent to more than 9 digits of precision in the decimal system).

There are no changes required to our framework for dealing with data in any dimension. Though our techniques work for any dimension, however, as d increases, the number of bits required for the z-value also increases. This introduces storage overhead as well as performance degradation (the fanout of the clustered B+ tree on the $zval$ attribute drops). Hence, for the really large dimensionality (say $d > 30$) one should consider using techniques that are specially designed for those purposes, for example the LSH-based method [5], [14], [25], [30].

Another nice property of our approach is the easy and efficient support of updates, both insertions and deletions. For the deletion of a record r, we simply delete r based on its pid from all tables $\mathrm{R}^0, \ldots, \mathrm{R}^\alpha$. For an insertion of a record r that corresponds to a point p, we calculate the z-values for $p^0, \ldots, p^\alpha$, recall $p^i = p + \overrightarrow{v_i}$. Next, we simply insert r into all tables $\mathrm{R}^0, \ldots, \mathrm{R}^\alpha$ but with different z-values. The database engine will take care of maintaining the clustered B+ tree indices on the $zval$ attribute in all these tables.

Finally, our queries are parallel-friendly as they execute similar queries over multiple tables with the same schema.

VII. Experiment

We implemented all algorithms in a database server running Microsoft SQL Server 2005. The state of the art algorithm for the exact kNN queries in arbitrary dimension is the iDistance [20] algorithm. For the approximate kNN, we compare against the $Medrank$ algorithm [12], since it is the state of the art for finding approximate kNN in relatively low dimensions and is possible to adapt it in the relational principle. We also compared against the approach using deterministic shifts with the Hilbert-curve [23], however, that method requires $O(d)$ shifts and computing Hilbert values in different dimensions. Both become very expensive when d increases, especially in a SQL environment. Hence, we focused on the comparison against the *Medrank* algorithm. We would like to emphasize they were not initially designed to be relational algorithms that are tailored for the SQL operators. Hence, the results here do not necessarily reflect their behavior when being used without the SQL constraint. We implemented iDistance [20] using SQL, assuming that the clustering step has been pre-processed outside the database and its incremental, recursive range exploration is achieved by a stored procedure. We built the clustered B+ tree index on the one-dimensional distance value in this approach. We used the suggested 120 clusters with the k-means clustering method, and the recommended Δr value from [20]. For the $Medrank$, the pre-processing step is to generate α random vectors, then create α one-dimensional lists to store the projection values of the data sets onto the α random vectors, lastly sort these α lists. We created one clustered index on each list. In the query step, we leveraged on the cursors in the SQL Server as the 'up' and 'down' pointers and used them to retrieve one record from each list in every iteration. The query process terminates when there are k elements have been retrieved satisfying the dynamic threshold (essentially the TA algorithm [13]). Both steps are achieved by using stored procedures. We did not implement the iJoin algorithm [33], the state of the art method for kNN-Joins, using only SQL, as it is not clear if that is feasible. Furthermore, it is based on iDistance and our kNN algorithm significantly outperforms the SQL-version iDistance. All experiments were executed on a Windows machine with an Intel 2.33GHz CPU. The memory of the SQL Server is set to 1.5GB.

Data sets. The real data sets were obtained from [1]. Each data set represents the road-networks for a state in United States. We have tested California, New Jersey, Maryland, Florida and others. They all exhibit similar results. Since the *California* data set is the largest with more than 10 million points, we only show its results. By default, we randomly sample 1 million points from the *California* data set. We also generate two types of synthetic data sets, namely, the uniform (*UN*) points, and the random-clustered (*R-Cluster*) points. Note that the *California* data set is in 2-dimensional space. For experiments in higher dimensional space, we use the *UN* and *R-Cluster* data sets.

Setup. Unless otherwise specified, we measured an algorithm's performance by the *wall clock time* metric which is the *total execution time* of the algorithm, i.e, including both the IO cost and the CPU cost. By default, *100* queries were generated for each experiment and we report the average for *one query*. For both kNN and kNN-Join queries, the query point or the query points are generated uniformly at random in the space of the data set P. The default size of P is $N = 10^6$. The default value for k is 10. We keep $\alpha = 2$, randomly shifted copies, for the *UN* data set and $\alpha = \min\{4, d\}$, randomly shifted copies, for the *California* and *R-Cluster* data sets, and set $\gamma = 2k$. These values of α and γ *do not change for different dimensions*. For the $Medrank$ algorithms, we set its α value as 2 for experiments in two dimensions and 4 for d larger than 2. This is to make fair comparison with our algorithms (with the same space overhead). The default dimensionality is 2.

A. Results for the kNN Query

Impact of α. The number of "randomly shifted" copies has a direct impact on the running time for algorithms *z^χ-kNN* and *z-kNN*, as well as their space overhead. Its effect on the running time is shown in Figure 4. For the *z^χ-kNN* algorithm, we expect its running time to increase linearly with α. Indeed, this is the case in Figure 4. The running time for the exact *z-kNN* algorithm has a more interesting trend. For the uniform *UN* data set, its running time also increases linearly with α, simply because it has to search more tables, and for the uniform distribution more random shifts do not change the probability of $\mathcal{A}^\chi = \mathcal{A}$. This probability is essentially $\Pr(\exists i, [z_\ell^i, z_h^i] \subseteq [z_{\gamma_\ell}^i, z_{\gamma_h}^i])$ and it stays the same in the uniform data for different α values. The number of false

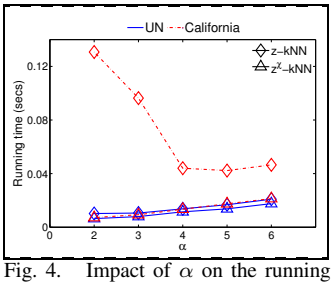
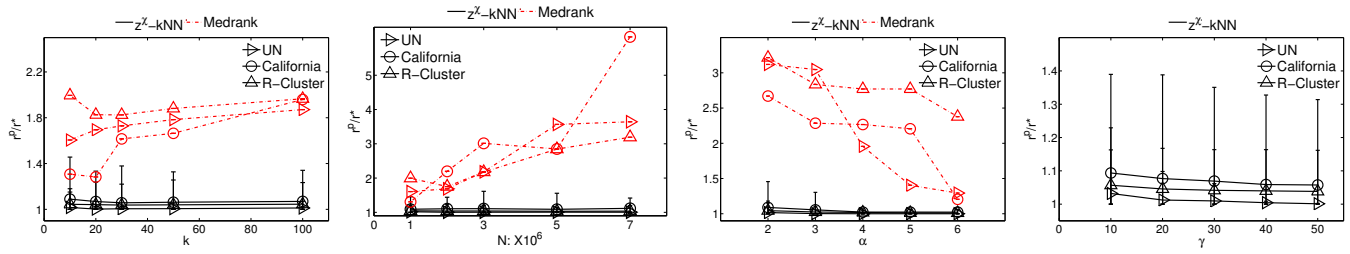


Fig. 4. Impact of α on the running time.



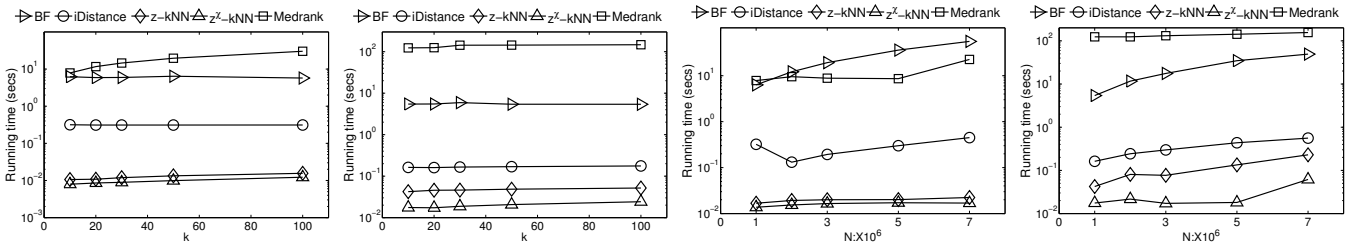
(a) Vary k. (b) Vary N. (c) Vary α. (d) Vary γ.

Fig. 3. The approximation quality of the z^{χ}-*kNN* and *Medrank* algorithm: average and the %5-%95 confidence interval.

positive points also does not change much for the uniform data, when using multiple shifts. On the other hand, for a highly skewed data set, such as *California*, increasing the α value could bring significant savings, even though z-*kNN* has to search more tables. This is due to two reasons. First, in highly skewed data sets, more "randomly shifted" copies increase the probability $\Pr(\exists i, [z_\ell^i, z_h^i] \subseteq [z_{\gamma_\ell}^i, z_{\gamma_h}^i])$. Second, more shifts also help reduce the number of false positive points around the kNN box, in at least one of the shifts. However, larger α values also indicate searching more tables. We expect to see a turning point, where the overhead of searching more tables, when introducing additional shifts, starts to dominate. This is clearly shown in Figure 4 for the z-*kNN* algorithm on *California*, and $\alpha = 4$ is the sweet spot. This result explains our choice for the default value of α. On the storage side, the cost is linear in α. However, since we only keep a small, constant number of shifts in any dimension, such a space overhead is small. The running time and the space overhead of the *Medrank* algorithm increase linearly to α. Since its running time is roughly 2 orders of magnitude more expensive than z^{χ}-*kNN* and z-*kNN*, we omitted it in Figure 4.

Approximation quality. We next study the approximation quality of z^{χ}-*kNN*, comparing against *Medrank*. For different values of k (Figure 3(a)), N (Figure 3(b)), α (Figure 3(c)), and γ (Figure 3(d)), Figure 3 confirms that z^{χ}-*kNN* achieves very good approximation quality and significantly outperforms *Medrank*. For z^{χ}-*kNN*, we plot the average *together with the* $5\% - 95\%$ *confidence interval* for 100 queries, and only the average for the *Medrank* as it has a much larger variance. In all cases, the average approximation ratio of z^{χ}-*kNN* stays below 1.1, and the worst cases never exceed 1.4. This is usually two times better or more than *Medrank*. These results also indicate that our algorithm not only achieves an excellent expected approximation ratio as our theorem has suggested, but also has a very small variance in practice, i.e., its worst case approximation ratio is still very good and much better than the existing method. Figure 3(a) and Figure 3(b) indicate that the approximate ratio of z^{χ}-*kNN* is roughly a constant over k and N, i.e., it has a superb scalability. Figure 3(c) indicates that the approximation quality of z^{χ}-*kNN* improves when more "randomly shifted" tables are used. However, for all data sets, even with $\alpha = 2$, its average approximation ratio is already around 1.1. Figure 3(d) reveals that increasing γ slightly does improve the approximation quality of z^{χ}-*kNN*, but not by too much. Hence, a γ value that is close to k (say $2k$) is good enough. Finally, another interesting observation is that, for the *California* and the *R-Cluster* data sets, four random shifts are enough to ensure z^{χ}-*kNN* to give a nice approximation quality. For the *UN* data set, two shifts already make it give an approximation ratio that is almost 1. This confirms our theoretical analysis, that in practice $O(1)$ shift for z^{χ}-*kNN* is indeed good enough. Among the three data sets, not surprisingly, the *UN* data set consistently has the best approximation quality and the skewed data sets give slightly worse results. Increasing α values does bring the approximation quality of *Medrank* closer to z^{χ}-*kNN* (Figure 3(c)), however, z^{χ}-*kNN* still achieves much better approximation quality.

Running time. Next we test the running time (Figure 5 in next page) of different algorithms for the kNN queries, using default values for all parameters, but varying k and N. Since both *California* and *R-Cluster* are skewed, we only show results from *California*. Figure 5 immediately tells that both the z^{χ}-*kNN* and the z-*kNN* significantly outperform other methods by one to three orders of magnitude, including the brute-force approach *BF* (the SQL query using the "Euclidean" UDF directly), on millions of points and varying k. In many cases, the SQL version of the $Medrank$ method is even worse than the *BF* approach, because retrieving records from multiple tables by cursors in a sorted order, round-by-round fashion is very expensive in SQL; and updating the candidate set in $Medrank$ is also expensive by SQL. The z^{χ}-*kNN* has the best running time followed by the z-*kNN*. Both of them outperform the iDistance method by at least one order of magnitude. In most cases, both the z^{χ}-*kNN* and the z-*kNN* take as little as 0.01 to 0.1 second, for a k nearest neighbor query on *several millions of records, for* $k \leq 100$. Interestingly enough, both Figure 5(a) and 5(b) suggest, that the running time for both the z^{χ}-*kNN* and the z-*kNN* increase very slowly, w.r.t the increment on the k value. This is because the dominant cost for both algorithms, is to identify the set of candidate points, using the *clustered* B+ tree. The sequential scan (with the range depending on the k value) around the successor record is extremely fast, and it is almost indifferent to the k values, unless there is a significant increase. Figure 5(c) and 5(d) indicate that the running time of all algorithms increase with larger N values. For the *California* data set, as the distribution is very skewed, the chance that our algorithms have to scan more false positives and miss exact kNN results is higher.



(a) Vary k: *UN*. (b) vary k: *California*. (c) vary N: *UN*. (d) vary N: *California*.

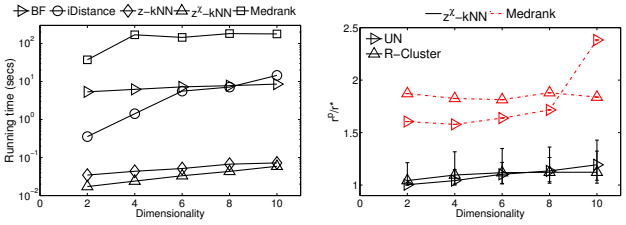
Fig. 5. kNN queries: running time of different algorithms, default $N = 10^6$, $k = 10, d = 2, \alpha = 2, \gamma = 2k$.

Hence, the z^{χ}-*kNN* and the *z-kNN* have higher costs than their performance in the *UN* data set. Nevertheless, Figure 5(c) and 5(d) show that z^{χ}-*kNN* and *z-kNN* have excellent scalability comparing to other methods, w.r.t the size of the database. For example, for 7 million records, they still just take less than 0.1 second.

Effect of dimensionality. We next investigate the impact of dimensionality to our algorithms, compared to the *BF* and the SQL-version of the iDistance and $Medrank$ methods, using the *UN* and *R-Cluster* data sets. The running time for both data sets are similar, hence we only report the result from *R-Cluster*. Figure 6(a) indicates that the SQL version of the *Medrank* method is quite expensive. The *BF* method's running time increases slowly with the dimensionality. This is expected since the IOs contribute the dominant cost. Increasing the dimensionality does not significantly affect the total number of pages in the database. The running time of z^{χ}-*kNN* and *z-kNN* do increase with the dimensionality, but at a much slower pace compared to the SQL-versions of the iDistance, Medrank and BF. When the dimension exceeds eight, the performance of the SQL version iDistance becomes worse than the *BF* method. When dimensionality becomes 10, our algorithms provide more than two orders of magnitude performance gain, compared to the *BF*, the *iDistance* and the *Medrank* methods, for both the uniform and the skewed data sets. Finally, we study the approximation quality of the z^{χ}-*kNN* algorithm, compared against the *Medrank* when dimensionality increases and the result is shown in Figure 6(b). For z^{χ}-*kNN*, we show its average as well as the 5%-95% confidence interval. Clearly, z^{χ}-*kNN* gives excellent approximation ratios across all dimensions and consistently outperforms the *Medrank* algorithm by a large margin. Furthermore, similar to results in two dimension from Figure 3, z^{χ}-*kNN* has a very small variance in its approximation quality across all dimensions. For example, its average approximation ratio is 1.2 when $d = 10$, almost two times better than *Medrank*; and its worst case is below 1.4 which is still much better than the approximation quality of the *Medrank*.

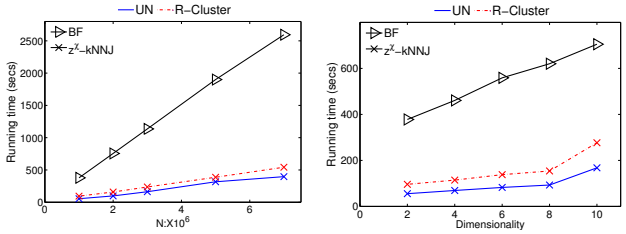
B. Results for the kNN-Join Query

In this section, we study the kNN-Join queries, by comparing the z^{χ}-*kNNJ* algorithm to the *BF* SQL query that uses the UDF "Euclidean" as a major join condition (as shown in Section I). By default, $M = |Q| = 100$. Similar to



(a) Running time:*R-Cluster*. (b) Approximation Quality

Fig. 6. kNN queries: impact of the dimensionality.



(a) Vary N. (b) Vary dimensionality.

Fig. 7. kNN-Join running time: z^{χ}-*kNNJ* vs *BF*.

results between z^{χ}-*kNN* and *z-kNN* in the kNN queries, the exact version of this algorithm (i.e., *z-kNNJ*) has very similar performance (in terms of the running time) to the z^{χ}-*kNNJ*. For brevity, we focus on the z^{χ}-*kNNJ* algorithm. Figure 7 shows the running time of the kNN-Join queries, using either the z^{χ}-*kNNJ* or the *BF* when we vary either N or d on *UN* and *R-Cluster* data sets. In all cases, z^{χ}-*kNNJ* is significantly better than the *BF* method, which essentially reduces to the *nested loop* join. Both algorithms are not sensitive to varying k values up to $k = 100$ and we omitted this result for brevity. However, the *BF* method does not scale well with larger databases as shown by Figure 7(a). The z^{χ}-*kNNJ* algorithm, on the contrary, has a much slower increase in its running time when N becomes larger (up-to 7 million records). Finally, in terms of dimensionality (Figure 7(b)), both algorithms are more expensive in higher dimensions. However, once again, the z^{χ}-*kNNJ* algorithm has a much slower pace of increment. We also studied the approximation quality for z^{χ}-*kNNJ*. Since it is developed based on the same idea as the the z^{χ}-*kNN* algorithm, their approximation qualities are the same. Hence, the results were not shown.

C. Updates and Distance Based θ-Join

We also performed experiments on the distance based θ-Join queries. That too, has very good performance in practice.

As discussed in Section VI, our methods easily supports updates, and the query performance is not affected by dynamic updates. Many existing methods for kNN queries could suffer from updates. For example, the performance of the iDistance method depends on the quality of the initial clusters. When there are updates, dynamically maintaining good clusters is a very difficult problem [20]. This is even harder to achieve in a relational database with only SQL statements. Medrank is also very expensive to support ad-hoc updates [12].

D. Additional Query Conditions

Finally, another benefit of being relational is the easy support for additional, ad-hoc query conditions. We have verified this with both the z^{χ}-*kNN* and *z-kNN* algorithms, by augmenting additional query predicates and varying the query selectivity of those predicates. When databases have meta-data (some statistical information on different attributes) available for estimating the query selectivity, the query optimizer is able to perform further pruning based on those additional query conditions. This is a natural result as both the z^{χ}-*kNN* and *z-kNN* algorithms are implemented by standard SQL operators.

VIII. Related Works

The nearest neighbor query with L_p norms has been extensively studied. In the context of spatial databases, R-tree provides efficient algorithms using either the depth-first [29] or the best-first [18] approach. These algorithms typically follow the branch and bound principle based on the MBRs in an R-tree index [6], [16]. Some commercial database engines have already incorporated the R-tree index into their systems. However, there are still many already deployed relational databases in which such features are not available. Also, the R-tree family has relatively poor performance for data beyond six dimensions and the kNN-join with the R-tree is most likely not available in the engines even if R-tree is available.

Furthermore, R-tree does not provide any theoretical guarantee on the query costs for nearest neighbor queries (even for approximate versions of these queries). Computational geometry community has spent considerable efforts in designing approximate nearest neighbor algorithms [2], [9]. Arya et al. [2] designed a modification of the standard kd-tree, called the Balanced Box Decomposition (BBD) tree that can answer $(1+\epsilon)$-approximate nearest neighbor queries in $O(1/\epsilon^d \log N)$. BBD-tree takes $O(N \log N)$ time to build. BBD trees are certainly not available in any database engine.

Practical approximate nearest neighbor search methods do exist. One is the well-known locality sensitive hashing (LSH) [14], where data in high dimensions are hashed into random buckets. By carefully designing a family of hash functions to be *locality sensitive*, objects that are closer to each other in high dimensions will have higher probability to end up in the same bucket. The drawback of the basic LSH method is that in practice, large number of hash tables may be needed to approximate nearest neighbors well [25] and successful attempts have been made to improve the performance of the LSH-based methods [5], [30]. The LSH-based methods are designed for data in extremely high dimensions (typically $d > 30$ and up-to 100 or more). It is not optimized for data in relatively low dimensions. Our focus in this work is to design *relational algorithms* that are tailored for the later (2 to 10 dimensions). Another approximate method that falls into this class is the Medrank method [12]. In this method, elements are projected to a random line, and ranked based on the proximity of the projections to the projection of the query. Multiple random projections are performed and the aggregation rule picks the database element that has the best median rank. A limitation of both the LSH-based methods and the Medrank method is that they can only give approximate solutions and could not help for finding the exact solutions. Note that for any approximate kNN algorithm, it is always possible to, in a post-processing step, retrieve the exact kNN result by a range query using the distance between q and the kth nearest neighbor from the approximate solution. However, it no longer guarantees any query costs provided by the approximate algorithm, as it was not designed for the range query which, in high dimensional space, often requires scanning the entire database.

Another method is to utilize the space-filling curves and map the data into one dimensional space, represented by the work of Liao et al. [23]. Specifically, their algorithm uses $O(d+1)$ *deterministic* shifted copies of the data points and stores them according to their positions along a Hilbert curve. Then it is possible to guarantee that a neighbor within an $O(d^{1+1/t})$ factor of the exact nearest neighbor, can be returned for any L_t norm with at most $O((d+1)\log N)$ page accesses. Our work is inspired by this approach [10], [23], *however, with significant differences*. We adopted the Z-order for the reason that the Z-value of any point can be computed using only the bit shuffle operation. This can be easily done in a relational database. More importantly, we use *random* shifts instead of deterministic shifts. Consequently, only $O(1)$ shifts are needed to get an expected constant approximation result, using only $O(\log N)$ page accesses for *any fixed dimensions*. In practice, our approximate algorithm gives orders of magnitude improvement in terms of its efficiency, and at the same time guarantees much better approximation quality comparing to adopting existing methods by SQL. In addition, our approach supports *efficient, exact* kNN queries. It obtains the exact kNN for certain types of distributions using only $O(\log N)$ page accesses for any dimension, again using just $O(1)$ random shifts. Our approach also works for kNN-Joins.

Since we are using Z-values to transform points from higher dimensions to one dimensional space, a related work is the UB-Tree [28]. Conceptually, the UB-Tree is to index Z-values using B-Tree. However, they [28] only studied range query algorithms and they did not use random shifts to generate the Z-values. To the best of our knowledge, the only prior work on kNN queries in relational databases was [3]. However, it has a fundamentally different focus. Their goal is to explore ways to improve the query optimizer inside the database engine to handle kNN queries. In contrast, our objective is to design SQL-based algorithms outside the database engine.

The state of the art technique for retrieving the exact k

nearest neighbors in high dimensions is the iDistance method [20]. Data is first partitioned into clusters by any popular clustering method. A point p in a cluster c_i is mapped to a one-dimensional value, which is the distance between p and c_i's cluster center. Then, kNN search in the original high dimensional data set could be translated into a sequence of incremental, recursive range queries, in these one dimensional distance values [20], that gradually expands its search range until kNN is guaranteed to be retrieved. Assuming that the clustering step has been performed outside the database and the incremental search step is achieved in a stored procedure, one can implement a SQL-version of the iDistance method. However, as it was not designed to be a SQL-based algorithm, our algorithm outperforms this version of the iDistance method as shown in Section VII.

Finally, the kNN-Join has also been studied [7], [8], [11], [31], [33]. The latest results are represented by the iJoin algorithm [33] and the Gorder algorithm [31]. The first approach is based on the iDistance, and it extends the iDistance method to support the kNN-Join. The extension is non-trivial and it is not clear how to extend it into a relational algorithm using only SQL statements. Gorder is a block nested loop join method that exploits sorting, join scheduling and distance computation filtering and reduction. Hence, it is an algorithm to be implemented inside the database engine, which is different from our objectives. Other join queries are considered for spatial data sets as well, such as the distance join [17], [18], multiway spatial join [26] and others. Interested readers are referred to the article by Jacox et al. [19]. Finally, the distance-based similarity join in GPU with the help of z-order curves was investigated in [24].

IX. Conclusion

This work revisited the classical kNN-based queries. We designed efficient algorithms that can be implemented by SQL operators in large relational databases. We presented a constant approximation for the kNN query, with logarithmic page accesses in any fixed dimension and extended it to the exact solution, both using just $O(1)$ random shifts. Our approach naturally supports kNN-Joins, as well as other interesting queries. No changes are required for our algorithms for different dimensions, and the update is almost trivial. Several interesting directions are open for future research. One is to study other related, interesting queries in this framework, e.g., the reverse nearest neighbor queries. The other is to examine the relational algorithms to the data space other than the L_p norms, such as the important road networks [22], [27].

X. Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments that improve this work. Bin Yao and Feifei Li are supported in part by Start-up Grant from the Computer Science Department, FSU. Piyush Kumar is supported in part by NSF grant CCF-0643593.

References

[1] Open street map. `http://www.openstreetmap.org`.
[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of ACM*, 45(6):891–923, 1998.
[3] A. W. Ayanso. *Efficient processing of k-nearest neighbor queries over relational databases: A cost-based optimization*. PhD thesis, 2004.
[4] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: a principled and practical approach. In *SIGMOD*, 2005.
[5] M. Bawa, T. Condie, and P. Ganesan. Lsh forest: self-tuning indexes for similarity search. In *WWW*, 2005.
[6] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.
[7] C. Böhm and F. Krebs. High performance data mining using the nearest neighbor join. In *ICDM*, 2002.
[8] C. Böhm and F. Krebs. The k-nearest neighbour join: Turbo charging the kdd process. *Knowl. Inf. Syst.*, 6(6):728–749, 2004.
[9] T. M. Chan. Approximate nearest neighbor queries revisited. In *SoCG*, 1997.
[10] T. M. Chan. Closest-point problems simplified on the ram. In *SODA*, 2002.
[11] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD*, 2000.
[12] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *SIGMOD*, 2003.
[13] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.
[14] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.
[15] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, 2001.
[16] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, 1984.
[17] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD*, 1998.
[18] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2), 1999.
[19] E. H. Jacox and H. Samet. Spatial join techniques. *ACM Trans. Database Syst.*, 32(1), 2007.
[20] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.
[21] D. R. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. In *STOC*, 2002.
[22] M. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *VLDB*, 2004.
[23] S. Liao, M. A. Lopez, and S. T. Leutenegger. High dimensional similarity search with space filling curves. In *ICDE*, 2001.
[24] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A fast similarity join algorithm using graphics processing units. In *ICDE*, 2008.
[25] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *VLDB*, 2007.
[26] N. Mamoulis and D. Papadias. Multiway spatial joins. *ACM Trans. Database Syst.*, 26(4):424–475, 2001.
[27] D. Papadias, M. L. Yiu, N. Mamoulis, and Y. Tao. Nearest neighbor queries in network databases. In *Encyclopedia of GIS*. 2008.
[28] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the ub-tree into a database system kernel. In *VLDB*, 2000.
[29] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, 1995.
[30] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high-dimensional nearest neighbor search. In *SIGMOD*, 2009.
[31] C. Xia, H. Lu, B. C. Ooi, and J. Hu. Gorder: an efficient method for KNN join processing. In *VLDB*, 2004.
[32] B. Yao, F. Li, and P. Kumar. K nearest neighbor queries and knn-joins in large relational databases (almost) for free. Technical report, 2009. `www.cs.fsu.edu/~lifeifei/papers/nnfull.pdf`.
[33] C. Yu, B. Cui, S. Wang, and J. Su. Efficient index-based KNN join processing for high-dimensional data. *Inf. Softw. Technol.*, 49(4):332–344, 2007.