Spatial Approximate String Search

Feifei Li *Member, IEEE*, Bin Yao, Mingwang Tang, Marios Hadjieleftheriou

Abstract—This work deals with the approximate string search in large spatial databases. Specifically, we investigate range queries augmented with a string similarity search predicate in both Euclidean space and road networks. We dub this query the spatial approximate string (SAS) query. In Euclidean space, we propose an approximate solution, the MHR-tree, which embeds min-wise signatures into an R-tree. The min-wise signature for an index node u keeps a concise representation of the union of q-grams from strings under the sub-tree of u. We analyze the pruning functionality of such signatures based on the set resemblance between the query string and the q-grams from the sub-trees of index nodes. We also discuss how to estimate the selectivity of a SAS query in Euclidean space, for which we present a novel adaptive algorithm to find balanced partitions using both the spatial and string information stored in the tree. For queries on road networks, we propose a novel exact method, RSASSOL, which significantly outperforms the baseline algorithm in practice. The RSASSOL combines the q-gram based inverted lists and the reference nodes based pruning. Extensive experiments on large real data sets demonstrate the efficiency and effectiveness of our approaches.

Index Terms—approximate string search, range query, road network, spatial databases

1 INTRODUCTION

Keyword search over a large amount of data is an important operation in a wide range of domains. Felipe et al. has recently extended its study to spatial databases [17], where keyword search becomes a fundamental building block for an increasing number of real-world applications, and proposed the IR^2-Tree. A main limitation of the IR^2-Tree is that it only supports exact keyword search. In practice, keyword search for retrieving approximate string matches is required [3], [9], [11], [27], [28], [30], [36], [43]. Since exact match is a special case of approximate string match, it is clear that keyword search by approximate string matches has a much larger pool of applications. Approximate string search is necessary when users have a fuzzy search condition, or a spelling error when submitting the query, or the strings in the database contain some degree of uncertainty or error. In the context of spatial databases, approximate string search could be combined with any type of spatial queries. In this work, we focus on range queries and dub such queries as *Spatial Approximate String* (SAS) queries. An example in the Euclidean space is shown in Figure 1, depicting a common scenario in location-based services: find all objects within a spatial range r (specified by a rectangular area) that have a description that is similar to "theatre". We denote SAS queries in Euclidean space as (ESAS) queries. Similarly, Figure 2 extends SAS queries to road networks (referred as RSAS queries). Given a query point q and a network distance r on a road network, we want to retrieve all objects within distance r to q and with the description similar to "theatre", where the distance between two points is the length of their shortest path.

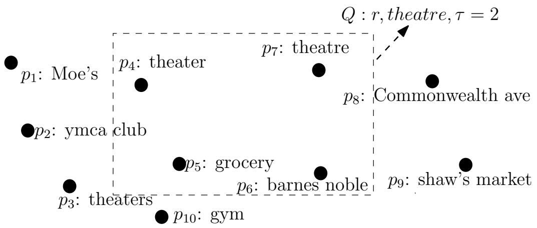


Fig. 1. An example of ESAS query.

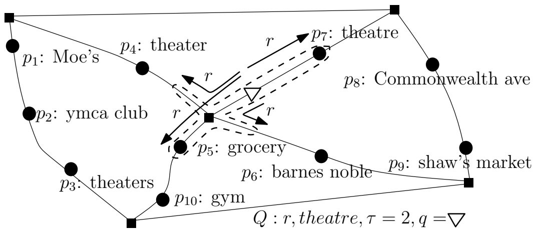


Fig. 2. An example of RSAS query.

A key issue in SAS queries is to define the similarity between two strings. The *edit distance* metric is often adopted [3], [9], [11], [27], [28], [30], [36], [43]. Specifically, given strings σ_1 and σ_2, the edit distance between σ_1 and σ_2, denoted as $\varepsilon(\sigma_1, \sigma_2)$, is defined as the minimum number of *edit operations* required to transform one string into the other. The *edit operations* refer to an insertion, deletion, or substitution of a single character. Clearly, ε is symmetric, i.e., $\varepsilon(\sigma_1, \sigma_2) = \varepsilon(\sigma_2, \sigma_1)$. For example, let $\sigma_1 =$'theatre' and $\sigma_2 =$'theater', then $\varepsilon(\sigma_1, \sigma_2) = 2$, by substituting the first 'r' with 'e' and the second 'e' with 'r'. We *do not* consider the generalized edit distance in which the transposition operator (i.e., swapping two characters in a string while keeping others fixed) is also included. The standard method for computing $\varepsilon(\sigma_1, \sigma_2)$ is a dynamic programming formulation. For two strings with lengths n_1 and n_2 respectively, it has a complexity of $O(n_1 n_2)$. That said, given the edit distance threshold $\tau = 2$, the answer to the ESAS query in Figure 1 is $\{p_4, p_7\}$; the answer to the RSAS query in Figure 2 is $\{p_7\}$.

- *Feifei Li and Mingwang Tang are with the School of Computing, University of Utah. E-mail: {lifeifei, tang}@cs.utah.edu. Bin Yao is with the Department of Computer Science and Engineering, Shanghai Jiao Tong University (contact author). E-mail: yaobin@cs.sjtu.edu.cn. Marios Hadjieleftheriou is with the AT&T Labs Research. E-mail: marioh@research.att.com.*

A straightforward solution to any SAS query is to use any existing techniques for answering the spatial component of an SAS query and verify the approximate string match predicate either in post-processing or on the intermediate results of the spatial search. We refer to them as the *spatial solution*.

In the Euclidean space, we can instantiate the *spatial solution* using R-trees. While being simple, this *R-tree solution* could suffer from unnecessary node visits (higher IO cost) and string similarity comparisons (higher CPU cost). To understand this, we denote the exact solution to an ESAS query as $\mathcal{A}$ and the set of candidate points that have been visited by the R-tree solution as $\mathcal{A}_c$. An intuitive observation is that it may be the case that $|\mathcal{A}_c| \gg |\mathcal{A}|$, where $|\cdot|$ denotes set cardinality. In an extreme example, considering an ESAS query with a query string that does not have any similar strings within its query range, $\mathcal{A} = \emptyset$. So ideally, this query should incur a minimum query cost. However, in the worst case, an R-tree solution could possibly visit all index nodes and data points from the R-tree. In general, the case that r contains a large number of points could lead to unnecessary IO and CPU overhead, since computing the edit distance between two strings has a quadratic complexity (to the length of the string). The fundamental issue here is that the possible pruning power from the string match predicate has been completely ignored by the R-tree solution. Clearly, in practice a combined approach that prunes simultaneously based on the string match predicate and the spatial predicate will work better.

For RSAS queries, the baseline spatial solution is based on the Dijkstra's algorithm. Given a query point q, the query range radius r, and a string predicate, we expand from q on the road network using the Dijkstra algorithm until we reach the points distance r away from q and verify the string predicate either in a post-processing step or on the intermediate results of the expansion. We denote this approach as the *Dijkstra solution*. Its performance degrades quickly when the query range enlarges and/or the data on the network increases. This motivates us to find a novel method to avoid the unnecessary road network expansions, by combining the prunings from both the spatial and the string predicates simultaneously.

Similarly, another straightforward solution in both ESAS and RSAS queries is to build a string matching index and evaluate only the string predicate, completely ignoring the spatial component of the query. After all similar strings are retrieved, points that do not satisfy the spatial predicate are pruned in a post-processing step. We dub this the *string solution*. First, the string solution suffers the same scalability and performance issues (by ignoring one dimension of the search) as the *spatial solution*. Second, we want to enable the efficient processing of standard spatial queries (such as nearest neighbor queries, etc.) while being able to answer SAS queries additionally in existing spatial databases, i.e., a spatial-oriented solution is preferred in practice in spatial databases.

Another interesting problem is the *selectivity estimation* for SAS queries. The goal is to accurately estimate the size of the results for an SAS query with cost significantly smaller than that of actually executing the query itself. Selectivity estimation is very important for query optimization purposes and data analysis and has been studied extensively in database research for a variety of approximate string queries and spatial range queries [1], [33].

Thus, our main contributions are summarized as follows:

- We formalize the notion of SAS queries and the related selectivity estimation problem in Section 2.
- We introduce a new index for answering ESAS queries efficiently in Section 3.2, which embeds min-wise signatures of q-grams from sub-trees into the R-tree nodes and converts the problem into that of evaluating set resemblance using min-wise signatures.
- We present a novel and robust selectivity estimator for ESAS queries in Section 3.3. Our idea is to leverage an adaptive algorithm that finds balanced partitions of nodes from any R-tree based index based on both the spatial and string information in the R-tree nodes. The identified partitions are used as the buckets of the selectivity estimator.
- We design RSASSOL in Section 4 for RSAS queries. The RSASSOL method partitions the road network, adaptively searches relevant subgraphs, and prunes candidate points using both the string matching index and the spatial reference nodes. Lastly, an adapted multi-points ALT algorithm (MPALT) is applied, together with the exact edit distances, to verify the final set of candidates.
- We demonstrate the efficiency and effectiveness of our proposed methods for SAS queries using a comprehensive experimental evaluation in Section 5. For ESAS queries, our experimental evaluation covers both synthetic and real data sets of up to 10 millions points and 6 dimensions. For RSAS queries, our evaluation is based on two large, real road network datasets, that contain up to 175,813 nodes, 179,179 edges, and 2 millions points on the road network. In both cases, our methods have significantly outperformed the respective baseline methods.

We survey the related work in Section 6. The paper concludes with Section 7.

2 PROBLEM FORMULATION

Formally, a spatial database P contains points with strings. Each point in P may be associated with one or more strings. For brevity and without loss of generality, here we assume that each point in P has one associated string. Our methods can be easily generalized to handle multiple strings per point (see online Appendix D [31]). Hence, a data set P with N points is the following set: $\{(p_1, \sigma_1), \ldots, (p_N, \sigma_N)\}$. Different points may contain duplicate strings. In the sequel, when the context is clear, we simply use a point p_i to denote both its geometric coordinates and its associated string.

In the Euclidean space, each point is specified by its geometric coordinates in a d dimensional space. In a road network G, we have $G = (V, E)$, where V (E) denotes the set of nodes (edges) in G. We index nodes in G by unique ids, and specify an edge by its two end-nodes, placing the node with the smaller id first. That said, each point $p_i \in P$ resides on an edge $(n_i, n_j) \in E$, where $n_i, n_j \in V$ and $n_i < n_j$. We locate p_i by (n_i, n_j) and its distance offset to n_i.

A *spatial approximate string* (SAS) query Q consists of two parts: the spatial predicate Q_r and the string predicate Q_s. In this paper we concentrate on using *range* queries as the spatial

Symbol	Description
P	the set of points with strings
g_σ	the set of q-grams of the string σ
τ	the edit distance threshold
$\varepsilon(\sigma_1, \sigma_2)$	the edit distance between strings σ_1 and σ_2
$\rho(A, B)$	the set resemblance of two sets A and B
$\widehat{\rho}(A, B)$	an unbiased estimator for $\rho(A, B)$
$s(g_p)$	the min-wise signature of g_p
$\Theta(b), \prod(b)$	the area, perimeter of a block b
k	number of buckets
$G = (V, E)$	road network with vertex (edge) set V (E)
$d(p_1, p_2)$	network distance between p_1 and p_2
V_R	the set of reference nodes
$d^-(p_1, p_2)$	a lower bound of $d(p_1, p_2)$
$d^+(p_1, p_2)$	an upper bound of $d(p_1, p_2)$

Fig. 3. Frequently used notations.

predicate. In the Euclidean space, Q_r is defined by a query rectangle r; in road networks, it is specified by a query point q and a radius r. In both cases, the string predicate Q_s is defined by a string σ and an edit distance threshold τ.

Let the set $\mathcal{A}_r = \{p_x | p_x \in P \wedge p_x$ is contained in $r\}$ if P is in the Euclidean space; or, $\mathcal{A}_r = \{p_x | p_x \in P \wedge d(q, p_x) \leq r\}$ if P is in a road network and $d(q, p)$ is the network distance between two points q and p.

Let the set $\mathcal{A}_s = \{p_x | p_x \in P \wedge \varepsilon(\sigma, \sigma_x) \leq \tau\}$. We define the SAS query as follows:

Definition 1 (SAS query) An SAS query $Q : (Q_r, Q_s)$ retrieves the set of points $\mathcal{A} = \mathcal{A}_r \cap \mathcal{A}_s$.

The problem of *selectivity estimation for an* SAS *query Q* is to efficiently (i.e., faster than executing Q itself) and accurately estimate the size $|\mathcal{A}|$ of the query answer.

We use σ_p to denote the associated string of a point p and assume that P is static. Extending our techniques to the general case with multiple strings per point, or with other spatial query types, and dealing with dynamic updates will be discussed in the online Appendix D [31]. Figure 3 summarizes the notations frequently used in the paper.

3 THE ESAS QUERIES

3.1 Preliminaries

Let Σ be a finite alphabet of size $|\Sigma|$. A string σ of length n has n characters (possibly with duplicates) in Σ^*.

3.1.1 Edit distance pruning

Computing edit distance exactly is a costly operation. Several techniques have been proposed for identifying candidate strings within a small edit distance from a query string fast [4], [11], [33]. All of them are based on q-grams and a q-gram counting argument.

For a string σ, its q-grams are produced by sliding a window of length q over the characters of σ. To deal with the special case at the beginning and the end of σ, that have fewer than q characters, one may introduce special characters, such as "#" and "$", which are not in Σ. This helps conceptually extend σ by prefixing it with $q-1$ occurrences of "#" and suffixing it with $q-1$ occurrences of "$". Hence, each q-gram for the string σ has exactly q characters.

Example 1 The q-grams of length 2 for the string $theatre$ are $\{\#t, th, he, ea, at, tr, re, e\$\}$. The q-grams of length 2 for the string $theater$ are $\{\#t, th, he, ea, at, te, er, r\$\}$.

To handle duplicates in the q-grams of a string, we associate a counter with each *unique* q-gram to indicate the number of times it appears in the string. For example:

Example 2 The q-grams of length 2 for the string $aabaa$ are $\{(\#a, 1), (aa, 1), (ab, 1), (ba, 1), (aa, 2), (a\$, 1)\}$.

Clearly, a string of length n will have $n-q+1$ q-grams with each q-gram having length q. Let g_σ be the set of q-grams of the string σ. It is also immediate from the above example that strings within a small edit distance will share a large number of q-grams. This intuition has been formalized in [19], [42] and others. Essentially, if we substitute a single character in σ_1 to obtain σ_2, then their q-gram sets differ by at most q q-grams (the length of each q-gram). Similar arguments hold for both the insertion and deletion operations. Hence,

Lemma 1 *[From [19]] For strings σ_1 and σ_2 of length $|\sigma_1|$ and $|\sigma_2|$, if $\varepsilon(\sigma_1, \sigma_2) = \tau$, then $|g_{\sigma_1} \cap g_{\sigma_2}| \geq \max(|\sigma_1|, |\sigma_2|) - 1 - (\tau - 1) * q$.*

3.1.2 The min-wise signature

The min-wise independent families of permutations were first introduced in [6], [12]. A family of *min-wise independent permutations* $\mathcal{F}$ must satisfy equation (1) below. Let the universe of elements be $\mathcal{U}$, for any set X that is defined by elements from $\mathcal{U}$, i.e., $X \subseteq \mathcal{U}$, for any $x \in X$, when π is chosen at random in $\mathcal{F}$ we have:

$$\Pr(\min\{\pi(X)\} = \pi(x)) = \frac{1}{|X|}. \tag{1}$$

In equation (1), $\pi(X)$ produces a permutation of X and $\pi(x)$ is the location value of x in the resulted permutation, and $\min\{\pi(X)\} = \min\{\pi(x) | x \in X\}$. In other words, all elements of any fixed set X have an equal probability to be the minimum value for set X under permutation π from a min-wise independent family of permutations. The min-wise independent family of permutations is useful for estimating *set resemblance*. The set resemblance of two sets A and B is defined as:

$$\rho(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

Broder et al. has shown in [6] that a min-wise independent permutation π could be used to construct an unbiased estimator for $\rho(A, B)$, specifically, let:

$$\widehat{\rho}(A, B) = \Pr(\min\{\pi(A)\} = \min\{\pi(B)\}).$$

Then $\widehat{\rho}(A, B)$ is an unbiased estimator for $\rho(A, B)$. Based on this, one can define the *min-wise signature* of a set A using ℓ min-wise independent permutations from a family $\mathcal{F}$ as:

$$s(A) = \{\min\{\pi_1(A)\}, \min\{\pi_2(A)\}, \ldots, \min\{\pi_\ell(A)\}\}, \tag{2}$$

then, $\widehat{\rho}(A, B)$ could be estimated as:

$$\widehat{\rho}(A, B) = \frac{|\{i | \min\{\pi_i(A)\} = \min\{\pi_i(B)\}\}|}{\ell}.$$

The above can be easily extended to k sets, $A_1, \ldots, A_k$:

$$\widehat{\rho}(A_1, \ldots, A_k) = \frac{|\{i| \min\{\pi_i(A_1)\} = \cdots \min\{\pi_i(A_k)\}\}|}{\ell}. \quad (3)$$

Implementation of min-wise independent permutations requires generating random permutations of a universe and Broder et al. [6] showed that there is no efficient implementation of a family of hash functions that guarantees equal likelihood for any element to be chosen as the minimum element of a permutation. Thus, prior art often uses linear hash functions based on Rabin fingerprints to simulate the behavior of the min-wise independent permutations since they are easy to generate and work well in practice [6]. Let

$$h(x) = (\alpha_\lambda x^\lambda + \alpha_{\lambda-1} x^{\lambda-1} + \cdots + \alpha_1 x + \alpha_0) \mod p,$$

for large random prime p and λ. We generate independently at random ℓ linear hash functions $h_1, \ldots, h_\ell$ and let any $\pi_i = h_i$ for $i = 1, \ldots, \ell$.

3.2 The MHR-tree

Suppose the disk block size is B. The R-tree [21] and its variants (e.g., R*-tree [5]) share a similar principle. They first group $\leq B$ points that are in spatial proximity into a minimum bounding rectangle (MBR), which is stored in a leaf node. When all points in P are assigned into MBRs, the resulting leaf node MBRs are then further grouped together recursively till there is only one MBR left. Each node u in the R-tree is associated with the MBR enclosing all the points stored in its subtree, denoted by MBR(u). Each internal node also stores the MBRs of all its children.

For a range query r, we start from the root and check the MBR of each of its children, then recursively visit any node u whose MBR intersects or falls inside r. When a leaf node is reached, all the points that are inside r are returned. R*-trees achieve better performance in general than the original R-trees. In the following, we used R*-trees in our construction and simply use the notation R-tree to denote an R*-tree.

To incorporate the pruning power of edit distances into the R-tree, we can utilize the result from Lemma 1. The intuition is that if we store the q-grams for all strings in a subtree rooted at an R-tree node u, denoted as g_u, given a query string σ, we can extract the query q-grams g_σ and check the size of the intersection between g_u and g_σ, i.e., $|g_u \cap g_\sigma|$. Then we can possibly prune node u by Lemma 1, even if u does intersect with the query range r. Formally,

Lemma 2 *Let g_u be the set for the union of q-grams of strings in the subtree of node u. For a* SAS *query with $Q_s = (\sigma, \tau)$, if $|g_u \cap g_\sigma| < |\sigma| - 1 - (\tau - 1) * q$, then the subtree of node u does not contain any element from $\mathcal{A}_s$.*

Proof: g_u is a set, thus, it contains distinct q-grams. The proof follows by the definition of g_u and Lemma 1. □

By Lemma 2, we can introduce string-based pruning into the R-tree by storing sets g_u for all R-tree nodes u. However, the problem of doing this is that g_u becomes extremely large for nodes located in higher levels of the tree. This not only introduces storage overhead, but more importantly, it drastically reduces the fan-out of the R-tree and increases the query cost. To address this issue, we embed the min-wise signature of g_u in an R-tree node, instead of g_u itself. The min-wise signature $s(g_u)$ has a constant size (see Equation 2; assuming ℓ is some constant), and this means that $|s(g_u)|$ (its size) is independent of $|g_u|$. We term the combined R-tree with $s(g_u)$ signatures embedded in the nodes as the *M*in-wise signature with linear *Hashing R*-tree (MHR-tree). The rest of this section explains its construction and query algorithms.

3.2.1 The construction of the MHR-tree

For a leaf level node u, let the set of points contained in u be $\mathbf{u}_p$. For every point p in $\mathbf{u}_p$, we compute its q-grams g_p and the corresponding min-wise signature $s(g_p)$. Note that in order to compute the min-wise signature $s(g_u)$ for the node u, we do not need to compute g_u (which can be costly, space-wise). One can do this much more efficiently. Specifically, let $s(A)[i]$ be the ith element for the min-wise signature of a set A, i.e., $s(A)[i] = \min\{\pi_i(A)\}$. Given the set of min-wise signatures $\{s(A_1), \ldots, s(A_k)\}$ of k sets, by Equation 2, we have:

$$s(A_1 \cup \cdots \cup A_k)[i] = \min\{s(A_1)[i], \ldots, s(A_k)[i]\}, \quad (4)$$

for $i = 1, \ldots, \ell$, since each element in a min-wise signature always takes the smallest image for a set.

We can obtain $s(g_u)$ using Equation 4 and $s(g_p)$'s for every point $p \in \mathbf{u}_p$, directly. We store all (p, $s(g_p)$) pairs in node u, and $s(g_u)$ in the index entry that points to u in u's parent.

For an index level node u, let its child entries be $\{c_1, \ldots, c_f\}$ where f is the fan-out of the R-tree. Each entry c_i points to a child node w_i of u, and contains the MBR for w_i. We also store the min-wise signature of the node pointed to by c_i, i.e., $s(w_i)$. Clearly, $g_u = \cup_{i=1,\ldots,f} g_{w_i}$. Hence, $s(g_u) = s(g_{w_1} \cup \cdots \cup g_{w_f})$; based on Equation 4, we can compute $s(g_u)$ using $s(g_{w_i})$'s. This implies that we do not have to explicitly produce g_u to get $s(g_u)$, i.e., there is no need to store g_{w_i}'s.

This procedure is recursively applied in a bottom-up fashion until the root node of the R-tree has been processed.

3.2.2 Query algorithms for the MHR-tree

The query algorithms for the MHR-tree generally follow the same principles as the corresponding algorithms for the spatial query component. However, we would like to incorporate the pruning method based on q-grams and Lemma 2 without the explicit knowledge of g_u for a given R-tree node u. We need to achieve this with the help of $s(g_u)$. Thus, the key issue boils down to estimating $|g_u \cap g_\sigma|$ using $s(g_u)$ and σ.

We can easily compute g_σ and $s(g_\sigma)$ from the query string once, using the same hash functions that were used for constructing the MHR-tree. When encountering a node u, let g refer to $g_u \cup g_\sigma$ (g cannot be computed explicitly as g_u is not available). We compute $s(g) = s(g_u \cup g_\sigma)$ based on $s(g_u)$, $s(g_\sigma)$ and Equation 4. Next, we estimate the set resemblance $\rho(g, g_\sigma)$ between g and g_σ as follows:

$$\widehat{\rho}(g, g_\sigma) = \frac{|\{i| \min\{h_i(g)\} = \min\{h_i(g_\sigma)\}\}|}{\ell}. \quad (5)$$

Equation 5 is a direct application of Equation 3. Note that:

$$\rho(g, g_\sigma) = \frac{|g \cap g_\sigma|}{|g \cup g_\sigma|} = \frac{|(g_u \cup g_\sigma) \cap g_\sigma|}{|(g_u \cup g_\sigma) \cup g_\sigma|} = \frac{|g_\sigma|}{|g_u \cup g_\sigma|}. \quad (6)$$

Algorithm 1: QUERY-MHR(MHR-tree R, Range r, String σ, int τ)

```
1  Let $L$ be a FIFO queue initialized to $\emptyset$, let $\mathcal{A} = \emptyset$;
2  Let $u$ be the root node of $R$; insert $u$ into $L$;
3  while $L \neq \emptyset$ do
4      Let $u$ be the head element of $L$; pop out $u$;
5      if $u$ is a leaf node then
6          for every point $p \in \mathbf{u}_p$ do
7              if $p$ is contained in $r$ then
8                  if $|g_p \cap g_\sigma| \geq \max(|\sigma_p|, |\sigma|) - 1 - (\tau - 1) * q$ then
9                      if $\varepsilon(\sigma_p, \sigma) < \tau$ then Insert $p$ in $\mathcal{A}$;
10     else
11         for every child entry $c_i$ of $u$ do
12             if $r$ and MBR$(w_i)$ intersect then
13                 Calculate $s(g = g_{w_i} \cup g_\sigma)$ based on $s(g_{w_i})$, $s(g_\sigma)$ and Equation 4;
14                 Calculate $|\widehat{g_{w_i} \cap g_\sigma}|$ using Equation 9;
15                 if $|\widehat{g_{w_i} \cap g_\sigma}| \geq |\sigma| - 1 - (\tau - 1) * q$ then
16                     Read node $w_i$ and insert $w_i$ into $L$;
17 Return $\mathcal{A}$.
```

Based on Equations 5 and 6 we can estimate $|g_u \cup g_\sigma|$ as:

$$|\widehat{g_u \cup g_\sigma}| = \frac{|g_\sigma|}{\widehat{\rho}(g, g_\sigma)}. \tag{7}$$

Finally, we can estimate $\rho(g_u, g_\sigma)$ by:

$$\widehat{\rho}(g_u, g_\sigma) = \frac{|\{i | \min\{h_i(g_u)\} = \min\{h_i(g_\sigma)\}\}|}{\ell}. \tag{8}$$

Note that $\rho(g_u, g_\sigma) = |g_u \cap g_\sigma| / |g_u \cup g_\sigma|$. Hence, based on Equations 7 and 8 we can now estimate $|g_u \cap g_\sigma|$ as:

$$|\widehat{g_u \cap g_\sigma}| = \widehat{\rho}(g_u, g_\sigma) * |\widehat{g_u \cup g_\sigma}|. \tag{9}$$

Given the estimation $|\widehat{g_u \cap g_\sigma}|$ for $|g_u \cap g_\sigma|$, one can then apply Lemma 2 to prune nodes that cannot possibly contain points from $\mathcal{A}_s$. Specifically, an R-tree node u could be pruned if $|\widehat{g_u \cap g_\sigma}| < |\sigma| - 1 - (\tau - 1) * q$. Since $|\widehat{g_u \cap g_\sigma}|$ is only an estimation of $|g_u \cap g_\sigma|$, the pruning based on $|\widehat{g_u \cap g_\sigma}|$ may lead to false negatives (if $|\widehat{g_u \cap g_\sigma}| < |g_u \cap g_\sigma|$). However, empirical evaluation in Section 5 suggests that when a reasonable number of hash functions have been used in the min-wise signature (our experiment indicates that $\ell = 50$ is good enough for large databases with 10 million points), the above estimation is very accurate.

The ESAS query algorithm is presented in Algorithm 1. When the object is a data point (line 6), we can obtain $|g_p \cap g_\sigma|$ exactly; g_p is not stored explicitly in the tree, but can be computed on the fly by a linear scan of σ_p. We also know the lengths of both σ_p and σ at this point. Hence, in this case, Lemma 1 is directly applied in line 8 for better pruning power. When either σ_p or σ is long, calculating $|g_p \cap g_\sigma|$ exactly might not be desirable. In this case, we can still use $s(g_p)$ and $s(g_\sigma)$ to estimate $|\widehat{g_p \cap g_\sigma}|$ using Equation 9. When the object is an R-tree node, we apply Equation 9 and Lemma 2 to prune (lines 14-17), in addition to the pruning by the query range r and the MBR of the node (line 13).

3.3 Selectivity estimation for ESAS queries

Another interesting topic for approximate string queries in spatial databases is selectivity estimation. Several selectivity estimators for approximate string matching have been proposed, none though in combination with spatial predicates. Various techniques have been proposed specifically for edit distance [25], [28], [33]. A state of the art technique based on q-grams and min-wise signatures is *VSol* [33]. It builds inverted lists with q-grams as keys and string ids as values; one list per distinct q-gram in input strings. Each list is summarized using the min-wise signature of the string ids in the list. The collection on min-wise signatures and their corresponding q-grams (one signature per distinct q-gram) is the *VSol* selectivity estimator for a data set P.

VSol uses the *L-M* similarity for estimating selectivity. $L = |\sigma| - 1 - (\tau - 1) * q$ is the number of matching q-grams two strings need to have for their edit distance to be possibly smaller than τ (based on Lemma 1). M is the number of q-grams in *VSol* that match some q-grams in the query string σ. The *L-M* similarity quantifies the number of string ids contained in the corresponding M inverted lists that share at least L q-grams with the query. Clearly, if a given data string shares at least L q-grams with σ, then the corresponding string id should appear in at least L of these M lists. Identifying the number of such string ids (in other words the selectivity of the query), amounts to estimating the number of string ids appearing in L lists, for all M choose L combinations of lists (each has L lists from M candidate lists). Denote the set of string ids that appear in all L lists in the i-th combination with L_i, $1 \leq i \leq \binom{M}{L}$. The *L-M* similarity is defined as:

$$\rho_{LM} = | \cup L_i|. \tag{10}$$

If we can estimate ρ_{LM}, we can estimate the selectivity as $\frac{\rho_{LM}}{|P|}$. Computing ρ_{LM} exactly is very expensive, as it requires storing inverted lists for all q-grams in the database explicitly and also enumerating all M choose L combinations of lists. As it turns out, estimating ρ_{LM} using the inverted list min-wise signatures of *VSol* is straightforward. Further details appear in [33] and are beyond the scope of this paper.

A key observation in [33] is that the number of neighborhoods (denote it with η) in the data set P greatly affects *VSol*'s performance. A neighborhood is defined as a cluster of strings in P that have a small edit distance to the center of the cluster. For a fixed number of strings, say N, the smaller the value of η is, the more accurate the estimation provided by *VSol* becomes. We refer to this observation as the *minimum number of neighborhoods principle*.

However, *VSol* does not address our problem where selectivity estimation has to be done based on both the spatial and string predicates of the query. The general principle behind accurate spatial selectivity estimation is to partition the spatial data into a collection of buckets so that data within each bucket is as close as possible to a uniform distribution (in terms of their geometric coordinates). We denote this as the *spatial uniformity principle*. Every bucket is defined by the MBR of all points enclosed in it. Each point belongs to only one bucket and buckets may overlap in the areas they cover. Given a range query r, for each bucket b that intersects with r we compute the area of intersection. Then, assuming uniformity, the estimated number of points from b that also fall into r is directly proportional to the total number of points in b, the total area of b and the area of intersection between b and r. This principle has been successfully applied by several work [1], [20], which mostly differ on how buckets are formed.

That said, the basic idea of our selectivity estimator is to build a set of buckets $\{b_1, \ldots, b_k\}$ for some budget k. Let the number of points in the i-th bucket be n_i and its area be $\Theta(b_i)$. For each bucket b_i, we build a *VSol* estimator *VSol*$_i$ based on the min-wise signatures of the q-gram inverted lists of the strings contained in the bucket. The selectivity estimation for an ESAS query $Q = \{r, (\sigma, \tau)\}$ is done as follows. For every bucket b_i that intersects with r, we calculate the intersection area $\Theta(b_i, r)$ and the *L-M* similarity ρ^i_{LM} of strings in b_i with σ, using *VSol*$_i$. Let $\mathcal{A}_{b_i}$ denote the set of points from b_i that satisfy Q, then $|\mathcal{A}_{b_i}|$ is estimated as:

$$\widehat{|\mathcal{A}_{b_i}|} = n_i \frac{\Theta(b_i, r)}{\Theta(b_i)} \frac{\rho^i_{LM}}{n_i} = \frac{\Theta(b_i, r)}{\Theta(b_i)} \rho^i_{LM}. \quad (11)$$

Our challenge thus becomes how to integrate the minimum number of neighborhoods principle from *VSol* into the spatial uniformity principle effectively when building these k buckets.

3.3.1 The partitioning metric

Formally, given a data set P, we define η as the number of neighborhoods in P. The strings associated with the points in one neighborhood must have an edit distance that is less than τ' from the neighborhood cluster center. We can use any existing clustering algorithm that does not imply knowledge of the number of clusters (e.g., correlation clustering [14]) to find all neighborhoods in P (notice that edit distance without character transpositions is a metric, hence any clustering algorithm can be used). Given a rectangular bucket b in d-dimensions, let n_b be the number of points in b, η_b the number of neighborhoods, and $\{X_1, \ldots, X_d\}$ the side lengths of b in each dimension. The *neighborhood and uniformity quality* of b is defined as:

$$\Delta(b) = \eta_b n_b \sum_{1,\ldots,d} X_i \quad (12)$$

Intuitively, $\Delta(b)$ measures the total "uncertainty" of all points in bucket b along each dimension and each neighborhood of b, if we use b, n_b and η_b to succinctly represent points assuming a uniform probability of a point belonging to any neighborhood in every dimension. For a bucket b, a larger value of $\Delta(b)$ leads to larger errors for estimating string selectivity over b using Equation 11. Intuitively, the larger the perimeter of a bucket, the more error the spatial estimation for the point's location introduces; the larger the number of neighborhoods the larger the error of *VSol* becomes.

Thus, our problem is to build k buckets $\{b_1, \ldots, b_k\}$ for the input data set P and minimize the sum of their *neighborhood and uniformity qualities*, i.e., $\min \sum_{i=1}^{k} \Delta(b_i)$, where k is a budget specified by the user.

Once such k buckets are found, we build and maintain their *VSol* estimators and use Equation 11 to estimate the selectivity. Unfortunately, we can show that for $d > 1$, this problem is NP-hard (proof in online Appendix A [31]).

Theorem 1 *For a data set $P \in \mathbb{R}^d$ and $d > 1$, $k > 1$, let $\mathbf{b}_{i,p}$ be the set of points contained in b_i. Then, the problem of finding k buckets $\{b_1, \ldots, b_k\}$, s.t. $\forall i, j \in [1, k], i \neq j$, $\mathbf{b}_{i,p} \cap \mathbf{b}_{j,p} = \emptyset$, $b_i = MBR(\mathbf{b}_{i,p})$, b_i, b_j are allowed to overlap, and $\bigcup_{i=1}^{k} \mathbf{b}_{i,p} = P$, $\min \sum_{i=1}^{k} \Delta(b_i)$ is NP-hard.*

Given this negative result, in what follows, we present effective heuristics that work well in practice as alternatives.

3.3.2 The adaptive R-tree algorithm

We first illustrate our main ideas using a simple, greedy principle, which proceeds in multiple iterations. In each iteration, one bucket is produced. At the ith iteration, we start with a seed as the first point in b_i, randomly selected from the unassigned points (points not chosen by existing buckets), and keep adding points to b_i until no reduction to the overall "uncertainty" of the current configuration can be introduced. The overall "uncertainty" of a given configuration is estimated by the sum of the uncertainty of existing buckets (b_1 to b_i), and the assumption that the remaining, unassigned points are uniformly distributed into $k - i$ buckets. The detail of this algorithm is discussed in online Appendix B [31].

Directly applying the greedy algorithm on a large spatial database is very expensive. Note that the R-tree is a data partitioning index and its construction metrics are to minimize the overlap among its indexing nodes as well as the total perimeter of its MBRs. Hence, the MBRs of the R-tree serve as an excellent starting point for building the buckets for our selectivity estimator. This section presents a simple adaptive algorithm that builds the buckets based on the R-tree nodes, instead of constructing them from scratch. We term this algorithm the *Adaptive R-Tree Algorithm*.

Given an R-tree and a budget k, descending from the root, we find the first level in the tree that has more than k nodes, say it has $\kappa > k$ nodes $\{u_1, \ldots, u_\kappa\}$. Our task is to group these κ nodes into k buckets, with the goal of minimizing the sum of their *neighborhood and uniformity qualities*.

We follow an idea similar to the greedy algorithm and produce one bucket in each round. In the pre-processing step we find the number of neighborhoods for each R-tree node, denoted with $\{\eta_{u_1}, \ldots, \eta_{u_\kappa}\}$, and the number of points that are enclosed by each node, denoted with $\{n_{u_1}, \ldots, n_{u_\kappa}\}$. Let n_i and η_i be the number of points and the number of neighborhoods in bucket b_i respectively. In the i-th round, we select the node with the left-most MBR (by the left vertex of the MBR) from the remaining nodes as the initial seed for bucket b_i. Next, we keep adding nodes, one at a time, until the overall value for the neighborhood and uniformity quality for b_i and the remaining nodes cannot be reduced. When adding a new node u_j to b_i, we update the number of neighborhoods for b_i by clustering points covered by the updated b_i again. This could be done in an incremental fashion if we know the existing clusters for both b_i and u_j [15].

For remaining nodes, we assume that they are grouped into $k - i$ buckets in a uniform and independent fashion. Let remaining nodes be $\{u_{x_1}, \dots, u_{x_\ell}\}$ for some $\ell \in [1, \kappa - i]$, and each $x_i \in [1, \kappa]$. Then, the number of points that are covered by these nodes is $\overline{n} = \sum_{i=1}^{\ell} n_{u_{x_i}}$. We let $\overline{\eta}$ be the average number of neighborhoods for the remaining buckets, i.e., $\overline{\eta} = \sum_{i=1}^{\ell} \eta_{u_{x_i}}/(k - i)$. Similar to the greedy algorithm, we define the uncertainty for the current configuration as:

$$U'(b_i) = \eta_i \cdot n_i \cdot \Pi(\mathbf{b}_{i,p}) + \overline{\eta} \cdot \overline{n} \cdot \left(\frac{\Theta(\bigcup_{i=1}^{\ell} \mathbf{u}_{x_i,p})}{k - i} \right)^{1/d} \cdot d \quad (13)$$

$\Pi(\mathbf{b}_{i,p})$ is the perimeter of the MBR of $\mathbf{b}_{i,p}$, $\Theta(\bigcup_{i=1}^{\ell} \mathbf{u}_{x_i,p})$ is the area for the MBR of the remaining points covered by the remaining nodes $\{u_{x_1}, \dots, u_{x_\ell}\}$. We can find this MBR easily by finding the combined MBR of the remaining nodes.

Given Equation 13, the rest of the adaptive R-tree algorithm follows the same grouping strategy as the greedy algorithm. Briefly, we calculate the values of $U'(b_i)$ by adding each remaining node to the current b_i. If no node addition reduces the value of $U'(b_i)$, the i-th round finishes and the current b_i becomes the i-th bucket. Otherwise, we add the node that gives the smallest $U'(b_i)$ value to the bucket b_i, and repeat.

When there are $k - 1$ buckets constructed, we group all remaining nodes into the last bucket and stop the search. Finally, once all k buckets have been identified, we build the *VSol* estimator for each bucket. For the i-th bucket, we keep the estimator, the total number of points and the bucket MBR in our selectivity estimator. Given an ESAS query, we simply find the set of buckets that intersects with the query range r and estimate the selectivity using Equation 11.

4 THE RSAS QUERIES

In this case, since the locations of points are constrained by the road network and represented by the edge holding the point and the distance offset to the edge end, the MHR-tree is not applicable in this context. In order to handle large scale datasets, we adopt a disk-based road network storage framework and develop external-memory algorithms.

4.1 Disk-based road network representation

We adopt a disk-based storage model to our setting that groups network nodes based on their connectivity and distance, as proposed in [39]. Figure 4 demonstrates an instance of our model for the network shown in Figure 2. In our model, the adjacency list and the points are stored in two separate files, each is then indexed by a (separate) B+-tree. To facilitate our query algorithm, a small set V_R of nodes from V is selected as the *reference nodes*. The distance between two nodes, two points, or a node and a point is the length of the shortest (network) path connecting two objects of concern.

For each node n_i, we store its distance to each of the reference nodes in V_R (collectively denoted by RDIST$_i$ in Figure 4) and the number of adjacent nodes of n_i (e.g. 3 for n_1 in Figure 4) at the beginning of its adjacency list. How to select V_R and compute RDIST for each node will be discussed in Section 4.2. For each adjacent node n_j of n_i, we store the adjacent node ID, the length of the edge $e = (n_i, n_j)$ (NDIST(n_i, n_j) in Figure 4) and a pointer to the points group on e. If e does not contain any point, a null pointer is stored. A B+-tree is built on the adjacency list file. The key of this tree is the node id and the value is a pointer to its adjacency list. For example, in Figure 4, given the node id n_1, we can find its adjacency list from the B+-tree which contains n_1's RDIST, number of adjacent nodes, and each of its adjacent nodes (n_2, n_3, and n_5): their distances to n_1 and the pointers to the points group on each corresponding edge.

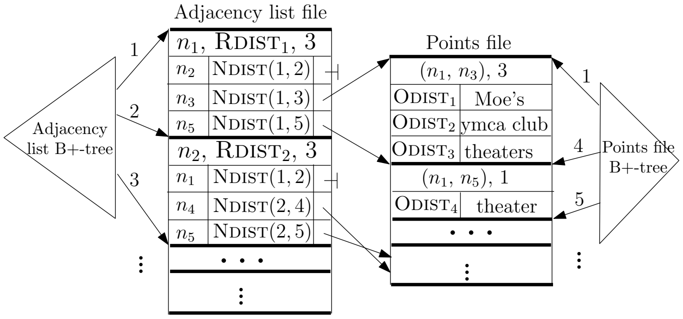


Fig. 4. Disk-based storage of the road network.

In the points file, the ids of points are assigned in such a way that for points on the same edge (n_i, n_j), points are stored by their offset distances *to the node with smaller id* in ascending order, and their ids are then sequentially assigned (crossing over different edges as well). Note that for any edge e defined by two nodes n_i and n_j, we represent e by always placing the node with the smaller id first. That said, if $n_i < n_j$, then in the adjacency list of n_j, the entry for n_i will have its pointer to the points group pointing to the points group of (n_i, n_j) (i.e., no duplication of points group will be stored).

We also store other information associated with a point (i.e., strings) after the offset distance. We store the points on the same edge in a *points group*. At the beginning of the points group, we also store the edge information (i.e., (n_i, n_j)) and the number of points on the edge. The groups are stored in a *points file* in the ascending order of the node ids defining the edges. Then a B+-tree is built on this file with keys being the first point id of each points group and values being the corresponding points group. For example, the points file in Figure 4 partially reflects the example in Figure 2. The ODIST$_i$ is the offset distance of point p_i.

Our storage model supports our query algorithms seamlessly and efficiently. Our design was inspired by the adjacency list module in [35], [39]. The design of the points file is motivated by our query algorithms. Lastly, in order to support the efficient approximate string search on a collection of strings, which is used as a component in our query algorithm, we integrate the *FilterTree* from [30] into our storage model. Please refer to online Appendix C [31] for details.

4.2 The RSASSOL algorithm

We partition a road network $G = \{V, E\}$ into m edge-disjoint subgraphs $G_1, G_2, \dots, G_m$, where m is a user parameter, and build one string index (FilterTree) for strings in each subgraph. We also select a small subset V_R of nodes from V as *reference* *nodes*: they are used to prune candidate points/nodes whose distances to the query point q are out of the query range r.

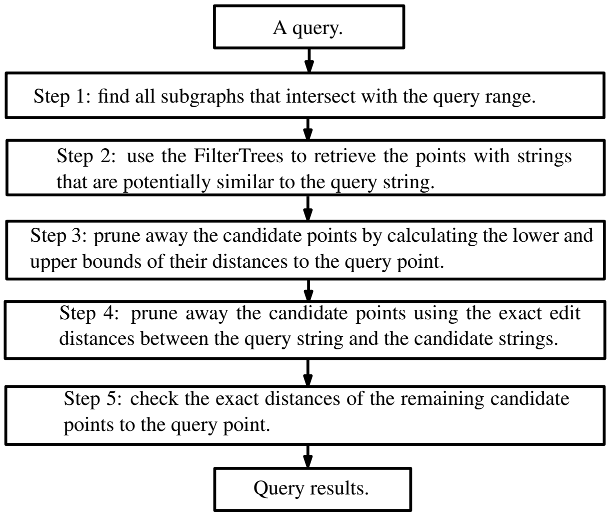

A query.
Step 1: find all subgraphs that intersect with the query range.
Step 2: use the FilterTrees to retrieve the points with strings that are potentially similar to the query string.
Step 3: prune away the candidate points by calculating the lower and upper bounds of their distances to the query point.
Step 4: prune away the candidate points using the exact edit distances between the query string and the candidate strings.
Step 5: check the exact distances of the remaining candidate points to the query point.
Query results.

Fig. 5. Overview of the RsasSol algorithm.

Conceptually, our Rsas query framework consists of five steps (refer to Figure 5 and the comments in Algorithm 2). Given a query, we first find all subgraphs that intersect with the query range. Next, we use the FilterTrees of these subgraphs to retrieve the points whose strings are potentially similar to the query string. In the third step, we prune away some of these candidate points by calculating the lower and upper bounds of their distances to the query point, using V_R. The fourth step is to further prune away some candidate points using the exact edit distance between the query string and strings of remaining candidates. After this step, the string predicate has been fully explored. In the final step, for the remaining candidate points, we check their exact distances to the query point and return those with distances within r.

We dub this algorithm RsasSol and the rest of this section presents the details of this algorithm. We use $d(o_1, o_2)$ to denote the network distance of two objects o_1, o_2 (where an object can be a network vertex, or a point on the network).

Pre-processing. To partition the network, in a nutshell, we randomly select m seeds from P (points of interest residing on the network) and construct the voronoi-diagram-like partition of the network using these seeds. We denote this approach as the *RPar* algorithm. Specifically, given a network $G = \{V, E\}$ and the dataset P on G, *RPar* randomly samples a small subset P_s of m seed points from P. Then, it first initializes m empty subgraphs, and assigns each point in P_s as the "center" of a distinct subgraph. Next, for each node $n \in V$, *RPar* finds n's nearest neighbor p in P_s, and computes $d(n, p)$. This can be done efficiently using Erwig and Hagen's algorithm [16], with G and P_s as the input. Next, for each edge $e \in E$ with $e = (n_l, n_r)$, *RPar* inserts e into the subgraph whose center p minimizes $\min\{d(p, n_l), d(p, n_r)\}$ among all subgraphs. When all edges are processed, *RPar* returns the m edge-disjoint subgraphs constructed. For each subgraph G_i, we collect the strings associated with points residing on edges in G_i and build a FilterTree as in last section.

We also select a small subset V_R of nodes from V as the reference nodes. This is to help us leverage the reference-nodes based distance pruning in a road network [18], [37], [38]. They also help us in our algorithm to compute the shortest paths between the query point and the final set of candidate points. How to select V_R greatly affects its effectiveness, and we adopt the best selection strategy proposed in [18]. Essentially, a reference node should be picked up on the boundary of the road network and as far away from each other as possible. It is also shown in [18] that a small constant number of reference nodes (e.g., 16) will be enough to achieve excellent performance even for large road networks and this only introduces a small linear space overhead, which is acceptable in most applications.

```
Algorithm 2: RsasSol(network $G$, $Q_r = (q, r)$, $Q_s = (\sigma, \tau)$)
1  /* step 1: find subgraphs intersecting the query range */
2  Find the set $X$ of ids of all subgraphs intersecting $(q, r)$;
3  Set $\mathcal{A} = \emptyset$, $\mathcal{A}_c = \emptyset$;
4  for each subgraph id $i \in X$ do
5      /* step 2: use the FilterTrees to retrieve points with strings that
       are potentially similar to the query string */
6      Find all point ids in $G_i$ whose associated strings $\sigma'$
       may satisfy $\varepsilon(\sigma', \sigma) \le \tau$ using FilterTree$_i$, and insert
       them into $\mathcal{A}_c$;
7  /* step 3: prune away points by calculating the lower and
   upper bounds of their distances to the query point using $V_R$ */
8  for every point $p_i \in \mathcal{A}_c$ do
9      calculate $d^+(p_i, q)$ and $d^-(p_i, q)$ as discussed;
10     if $d^+(p_i, q) \le r$ then
11         if $\varepsilon(\sigma_i, \sigma) \le \tau$ then
12             move $p_i$ from $\mathcal{A}_c$ to $\mathcal{A}$;
13         /* step 4: prune points using the exact edit distance
           between the query string and the candidate string */
14         else
15             delete $p_i$ from $\mathcal{A}_c$;
16     else
17         if $d^-(p_i, q) > r$ then
18             delete $p_i$ from $\mathcal{A}_c$;
19 /* step 4: same pruning as the step 4 above */
20 for every point $p_i \in \mathcal{A}_c$ do
21     if $\varepsilon(\sigma_i, \sigma) > \tau$ then
22         delete $p_i$ from $\mathcal{A}_c$;
23 /* step 5: check the exact distances of the remaining candidate
   points to the query point */
24 Use the mpALT algorithm to find all points $p$'s in $\mathcal{A}_c$
   such that $d(p, q) \le r$, push them to $\mathcal{A}$;
25 Return $\mathcal{A}$.
```

Query processing. The RsasSol algorithm is presented in Algorithm 2. First, we find all subgraphs that intersect with the query range. We employ the Dijkstra's algorithm (using the Fibonacci heap) to traverse nodes in G (note that we ignore points on G), starting from the query point q. Whenever this traversal meets the *first node of a new subgraph*, we examine that subgraph for further exploration (line 2). The algorithm terminates when we reach the boundary of the query range (defined by the distance r to q). For each subgraph G_i to be examined, we use the approximate string search over G_i's

FilterTree as the next pruning step (line 6), to find points from G_i that may share similar strings to the query string.

Then we further prune the candidate points using the spatial predicate, by computing lower and upper bounds on their distances to q using V_R, in a similar way to the ALT algorithm [18]. Recall that we have pre-computed and stored the distance of every network node to every node in V_R (the set RDIST$_i$ in Figure 4). Given a candidate point p on an edge $e = (n_i, n_j)$, the shortest path from p to a reference node n_r must pass through either n_i or n_j. Thus, the network distance $d(p, n_r) = \min(d(p, n_i) + d(n_i, n_r), d(p, n_j) + d(n_j, n_r))$. Note that $d(n_i, n_r)$ and $d(n_j, n_r)$ are available from RDIST$_i$ and RDIST$_j$, respectively; $d(p, n_i)$ is the distance offset of p to n_i which is available in the adjacency list and then points file of n_i (ODIST in Figure 4); and $d(p, n_j)$ =NDIST$(n_i, n_j) - d(p, n_i)$ where NDIST(n_i, n_j) is available in the adjacency list of n_i as well. We compute $d(p, n_r)$ on the fly rather than explicitly storing the distance between a point and a reference node since the number of points is much larger than the number of the nodes in G. By doing so, we avoid significant space blowup. We can also compute $d(q, n_r)$ in a similar way (only once). In summary, our structure allows us to compute $d(p, n_r)$ for any point and any reference node efficiently.

Given $d(p, n_r)$ and $d(q, n_r)$ for every $n_r \in V_R$, we then obtain the distance lower and upper bounds between p and q using the triangle inequality. Each reference node yields such a pair of lower and upper bounds. We take the maximum (minimum) value from the lower (upper) bounds of all reference nodes as the final lower (upper) bound of $d(p, q)$, denoted as $d^-(p, q)$ and $d^+(p, q)$ respectively. If $d^+(p, q) \leq r$, we know for sure p satisfies the spatial predicate and we only need to check the exact edit distance as the last measure (lines 10-15); if $d^-(p, q) > r$, we can safely remove p (lines 17-18); otherwise, we need to check both the exact edit distance and compute $d(p, q)$ to complete the verification on p.

After the pruning by $d^-(p, q)$ and $d^+(p, q)$, we compute the exact edit distances on the remaining candidate points in $\mathcal{A}_c$ (lines 21-22) and prune away points whose edit distances to the query string σ are larger than τ. Note that for a point p satisfying $d^+(p, q) \leq r$ and the exact edit distance threshold, it has already been removed from $\mathcal{A}_c$ and pushed to $\mathcal{A}$ before this step. For all other remaining candidates $\mathcal{A}_c$, we only need to compute the exact network distances between them and q to complete the algorithm. The naive solution is to apply the ALT algorithm for every $p \in A_c$ and q to find their shortest path [18]. However, this can be prohibitive when $|\mathcal{A}_c|$ is still large. Next, we introduce an improvement, *the* MPALT *algorithm*, which computes multiple shortest paths, within the query range, simultaneously at once between a single source point s and multiple destination points $\{t_1, \ldots, t_m\}$.

We leverage on the distances computed and stored in our storage model between a node to all reference nodes, which allows us to compute lower and upper distance bounds for $d(n, t)$, for any given node n and any destination point t, during the expansion. They are computed in similar fashion to that of the distance lower and upper bounds above for $d(p, q)$.

The MPALT algorithm minimizes the access to the network by avoiding the nodes that will not be on any shortest path between s and any destination t_i. It also avoids repeatedly access to the explored part of the network when calculating multiple shortest paths to multiple destinations. The basic idea works as follows. We start the expansion of the network from s with the two nodes from the edge containing s, and always expand the network from an explored node n (by adding adjacent nodes of n to a priority queue and checking points on corresponding edges) that has the shortest possible distance to any one of the destinations. We also avoid inserting an adjacent node n' of n to the priority queue if $d^+(n', s) > r$ (but we do check points on the edge (n, n')).

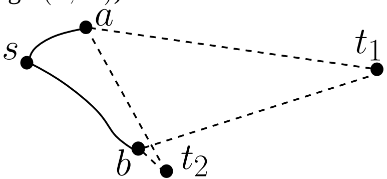


Fig. 6. Expansion in MPALT.

The algorithm terminates when the priority queue becomes empty (i.e., we have reached the boundary of the query range) or all the destination points are already met. Figure 6 illustrates the basic idea of the MPALT algorithm in choosing the next node for expansion, where we want to find the shortest paths from s to $\{t_1, t_2\}$. Instead of finding these two paths independently by applying the ALT algorithm twice on the same network, MPALT does this simultaneously at once. Suppose the solid curves represent the paths found so far and the dashed straight lines represent the estimated lower distance-bounds, i.e., we have at this point $d(s, a)$, $d(s, b)$, $d^+(s, t_1)$, $d^-(s, t_1)$, $d^+(s, t_2)$, and $d^-(s, t_2)$. In this case, the MPALT will choose b as the next node to expand as $d(s, b) + d^-(b, t_2)$ is the best potential candidate for the shortest path (compared to $d(s, b) + d^-(b, t_1)$, $d(s, a) + d^-(a, t_1)$, $d(s, a) + d^-(a, t_2)$).

Recall that we apply the MPALT algorithm using q as s and $\mathcal{A}_c$ (candidates after the exact edit distance pruning) as destination points $\{t_1, \ldots, t_m\}$. Thus, whenever a destination point is met in MPALT, it is output as one of the final answers.

Lastly, we would like to point out, unlike the MHR-tree in the Euclidean space which is an approximation solution, RSASSOL is an exact algorithm for RSAS queries.

4.3 Selectivity estimation of RSAS queries

Selectivity estimation of range queries on road networks is a much harder problem than its counterpart in the Euclidean space. Several methods were proposed in [41]. However, they are only able to estimate the number of nodes and edges in the range. None can be efficiently adapted to estimate the number of points in the range. One naive solution is to treat points as nodes in the network by introducing more edges. This clearly increases the space consumption significantly (and affects the efficiency) since the number of points is typically much larger than the number of existing nodes. Then we also have the challenges of integrating the spatial selectivity estimator with the string selectivity estimator in an effective way, as we did for the Euclidean space. This turns out to be non-trivial and we leave it as an open problem for future work.

5 EXPERIMENTAL EVALUATION

For ESAS queries, we implemented the R-tree solution and the MHR-tree, using the widely adopted spatial index library. The adaptive R-tree algorithm for the ESAS selectivity estimator seamlessly works for both the R-tree and the MHR-tree. For RSAS queries, we implemented the Dijkstra solution and the RSASSOL method, based on the disk-based storage model and the Flamingo package (http://flamingo.ics.uci.edu/, for the FilterTree). The default page size is 4KB and the fill factor of all indexes is 0.7. All experiments were executed on a Linux machine with an Intel Xeon CPU at 2GHz and 2GB of memory. Note that by ignoring one search dimension completely, *string solutions* suffer the similar issues as the *spatial solutions*. We focus on using the *spatial solutions* as the baseline methods for comparison, since they are preferred in spatial databases than *string solutions* (as discussed in Section 1 and Appendix D) and due to the space limitation.

Datasets. For ESAS queries, the real datasets were obtained from the open street map project. Each dataset contains the streets for a state in the USA. Each point has its longitude and latitude coordinates and several string attributes. We scale the points' coordinates into $(0, 10000)$ in any dimension. We combine the state, county and town names for a point as its associated string. For our experiments, we have used the Texas (*TX*) and California (*CA*) datasets, since they are the largest few in size among different states. The *TX* dataset has 14 million points and the *CA* dataset has 12 million points. The real datasets are in two dimensions. To test the performance of our algorithms on different dimensions for ESAS queries, we use two synthetic datasets. In the *UN* dataset, points are distributed uniformly in the space and in the *RC* dataset, points are generated with random clusters in the space. For both the *UN* and the *RC* datasets, we assign strings from our real datasets randomly to the spatial points generated. The default size N of the dataset P is 2 million. For the *TX* and *CA* datasets, we randomly sample 2 million points to create the default datasets. For all datasets the average length of the strings is approximately 14.

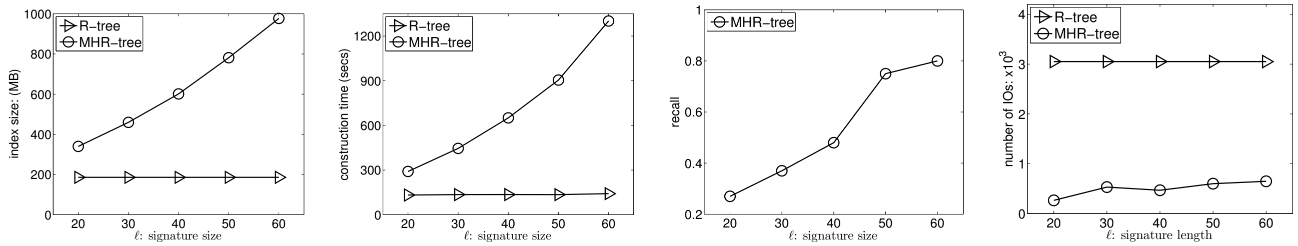
To test RSAS queries, we use two real road network datasets, *NAN* and *CAN*, obtained from the *Digital Chart of the World Server*. In particular, *NAN* (*CAN*) captures the road network in North America (California), and contains 175,813 nodes and 179,179 edges (21,048 nodes and 21,693 edges). We obtain a large number of real locations with text information in North America (California) from the open street map project and assign strings into the road network based on their coordinates. In the default datasets, we randomly sample 2 million points and assign them into the *NAN* and *CAN* road networks.

Setup. The spatial range predicate r (a rectangle) in an ESAS query is generated by randomly selecting a center point c_r and a query area that is specified as a percentage of total space, denoted as $\theta = \text{area of r}/\Theta(P)$. To make sure that the query will return non-empty results, we select the query string as the associated string of the nearest neighbor of c_r from P. The default value for θ is 3%. The default size of the signature is $\ell = 50$ hash values (200 bytes). The spatial range predicate of an RSAS query is generated by choosing a point q on a randomly selected edge and the radius r is a network distance value. For RSAS queries, the default setup is $r = 500$. Similarly, the query string is the same to the associated string of the nearest neighbor of q from P. The default number of subgraphs is $m = 100$ and the number of reference nodes is 4. For all query performance experiments we report averages over 100 randomly generated queries. In all cases, our experiments indicate that two-grams work the best. Hence, the default q-gram length is 2. In all cases, the default edit distance threshold is $\tau = 2$ and the default $N = |P|$ is $2,000,000$. *TX* and *NAN* are the default dataset for ESAS and RSAS queries, respectively. For all experiments, unless otherwise specified, we varied the values of one specific parameter of interest in the x-axis, while using the default values for all other parameters.

5.1 The ESAS queries

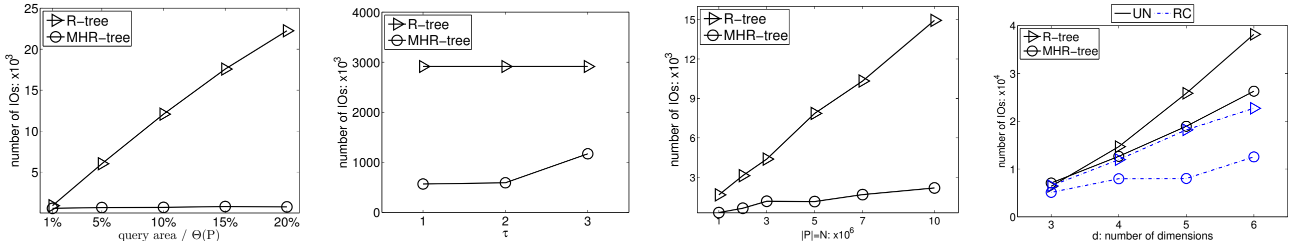
We first study the impact of the signature size on the performance of the MHR-tree. Figure 7 summarizes the results using the *TX* dataset. The results from the *CA* dataset are similar. The first set of experiments investigates the construction cost of the two indexes. Since the MHR-tree has to store signatures in its nodes, it has a smaller fan-out and a larger number of nodes compared to the R-tree. This indicates that the MHR-tree will need more space to store the nodes, and a higher construction cost to compute the signatures and write more nodes to the disk. This is confirmed by our experiments. Figure 7(a) indicates that the size of the MHR-tree increases almost linearly with the increase of the signature size. Similar trend holds for its construction cost as shown by Figure 7(b). Both of its size and construction cost are approximately $\frac{\ell}{10}$ times more expensive than the R-tree. A larger ℓ value leads to a higher overhead for the construction and storage of the MHR-tree, but it improves its query accuracy. Recall that the min-wise signature may underestimate the size of the intersection used for pruning in Lemma 2. This could result in pruning a subtree that contains some query results. Thus, an important measurement reflecting the accuracy of MHR-tree query results is the *recall*, the percentage of actual correct answers returned. Let $\mathcal{A}_x$ be the result returned by the MHR-tree for an ESAS query, then the recall for this query is simply $\frac{|\mathcal{A}_x|}{|\mathcal{A}|}$. Note that $\mathcal{A}_x$ will always be a subset of the correct result $\mathcal{A}$, as the MHR-tree will never produce any false positives (all points that pass the threshold test will be finally pruned by their exact edit distances to the query string). In other words, its precision is always 1. Figure 7(c) shows the recall of the MHR-tree using various signature sizes. Not surprisingly, larger signatures lead to better accuracy. When $\ell = 50$ recall reaches about 80%. Finally, the query cost of the MHR-tree, for various signature sizes, is always significantly lower than the cost of the R-tree, as shown in Figure 7(d). Its query cost does increase with a larger signature, however, the pace of this increment is rather slow. This means that we can actually use larger signatures to improve the query accuracy without increasing the query cost by much. However, this does introduce more storage overhead and construction cost. Hence, for our experiments, the default is $\ell = 50$.

Note that linear hashing is very cheap to compute in today's CPU. Hence, when comparing query performance in R-trees and MHR-trees, the IO cost dominates and is already a very good indicator of the overall query performance. As a result,



(a) index size. (b) construction time. (c) recall. (d) query performance.

Fig. 7. Impact of the signature size: TX dataset.



(a) vary θ, TX dataset. (b) vary τ, TX dataset. (c) vary N, TX dataset. (d) vary d, UN and RC datasets.

Fig. 8. Query performance, $\ell = 50$.

we have omitted the running time comparison for queries because of the space limit.

Using the default signature size of $\ell = 50$, we further investigate the improvement in query performance of MHR-tree compared to the R-tree. The results are summarized by Figure 8. Figure 8(a) shows the average number of IOs for various query area sizes, θ, from 1% to 20%, using the *TX* dataset. Clearly, R-tree becomes more and more expensive compared to the MHR-tree when the query area increases. For example, when $\theta = 10\%$, R-tree is 20 times more expensive in terms of IO compared to MHR-tree, and this gap enlarges to more than one order of magnitude for larger area sizes. This is due to the fact that the cost of range queries for the R-tree is proportional to the query area. On the other hand, the cost of the MHR-tree increases rather slowly due to the additional pruning power provided by the string predicate.

Next, we study the effect of the edit distance threshold, for query area $\theta = 3\%$. For $\tau = 1, 2, 3$, the MHR-tree always significantly outperforms the R-tree in Figure 8(b). Of course, when τ keeps increasing, the R-tree cost remains constant, while the MHR-tree cost increases. Hence, for large τ values, R-tree will be a better choice. Nevertheless, most approximate string queries return interesting results only for small τ values.

The next experiment, Figure 8(c), studies the scalability of the MHR-tree by varying the data size N. Using the *TX* dataset, N ranges from 1 to 10 million. The MHR-tree scales much better w.r.t. N. The IO difference between the MHR-tree and R-tree enlarges quickly for larger datasets. For example, Figure 8(c) shows that when N reaches 10 million, the IO cost for a query with $\tau = 2, \theta = 3\%$ for the MHR-tree is more than 10 times smaller than the cost of the R-tree. Similar results were observed for the *CA* dataset when we vary θ, τ and N.

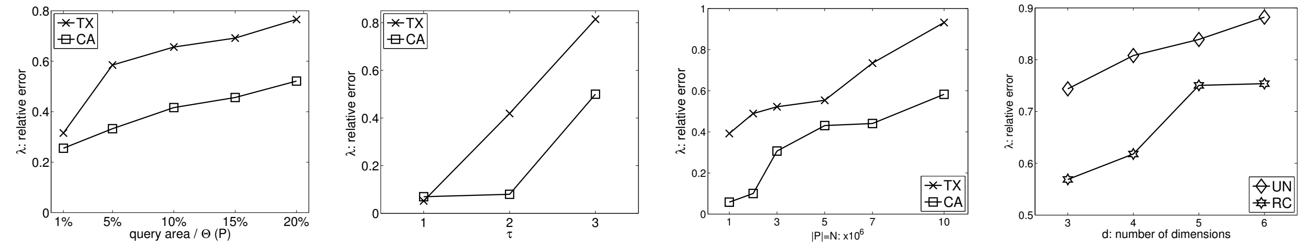
We study the scalability of the two indexes for higher dimensions. Using the default *UN* and *RC* datasets, for $d = 3, 4, 5, 6$. Figure 8(d) shows that the MHR-tree always outperforms the R-tree in all dimensions and also enjoys better scalability w.r.t. the dimensionality of the dataset, i.e., the MHR-tree outperforms the R-tree by larger margins in higher dimensions. For all these experiments, with $\ell = 50$, the recall of the MHR-tree stays between 70% to 80%.

Finally, the relationship between the size of the MHR-tree and its construction cost with respect to that of the R-tree is roughly a constant, as we vary N and d. For $\ell = 50$, they are roughly 5 times the respective cost of the R-tree.

5.2 Selectivity estimation of ESAS queries

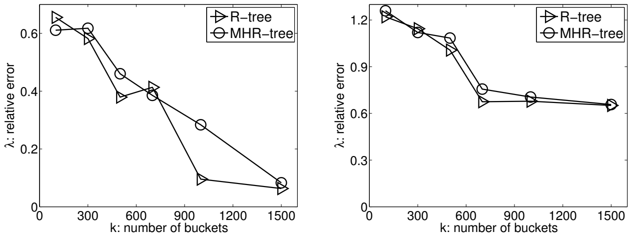
This section presents the experimental evaluation of our selectivity estimator for ESAS queries. We have implemented both the greedy algorithm and the adaptive R-tree algorithm. The adaptive algorithm is much cheaper and works almost as well in practice. Thus, we only report the results for the adaptive algorithm. We refer to the estimator built by the R-tree based adaptive algorithm as the *adaptive estimator*. An important measure that is commonly used when measuring the accuracy of a selectivity estimator is its *relative error* λ. Specifically, for an ESAS query Q, let its correct answer be $\mathcal{A}$, and the number of results estimated by a selectivity estimator be ξ, then $\lambda = \frac{|\xi - |\mathcal{A}||}{|\mathcal{A}|}$. Lastly, k denotes the number of buckets that the selectivity estimator is allowed to use.

Our first experiment is to study the relationship between k and λ. Intuitively, more buckets should lead to better accuracy. This is confirmed by the results in Figure 9 when we vary the number of buckets from 100 to 1500, on both the *CA* and *TX* datasets. Note that our adaptive algorithm works for any R-tree based index. Hence, in this experiment, we have tested it on both the R-tree and the MHR-tree. Clearly, when more buckets were used, the accuracy of the adaptive estimator improves. On the *CA* dataset, λ improves from above 0.5 to below 0.1 when k changes from 100 to 1500 (Figure 9(a)). On the *TX* dataset, λ improves from above 1.2 to 0.6 for the same setting (Figure 9(b)). The results also reflect that our algorithm is almost independent of the underlying R-tree variant, i.e., the results from the R-tree and the MHR-tree are similar. More importantly, these results reveal that the adaptive estimator achieves very good accuracy on both datasets when a small number of buckets is used, say $k = 1000$ on 2 million points.



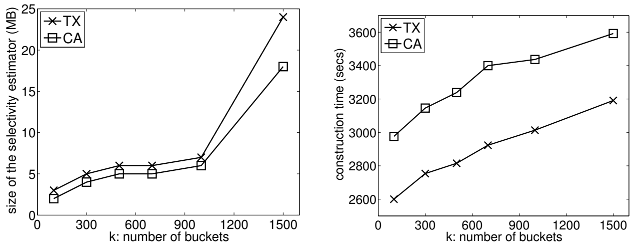
(a) vary θ, CA and TX datasets. (b) vary τ, CA and TX datasets. (c) vary N, CA and TX datasets. (d) vary d, UN and RC datasets.

Fig. 11. Relative errors of the adaptive estimator, $k = 1000$.



(a) CA dataset. (b) TX dataset.

Fig. 9. Errors of the adaptive estimator for ESAS queries.



(a) size. (b) construction time.

Fig. 10. Size, construction cost of the adaptive estimator.

Specifically, when $k = 1000$, on the R-tree index, the adaptive estimator has approximately only 0.1 relative error on the *CA* dataset (Figure 9(a)) and approximately 0.6 relative error on the *TX* dataset. The higher λ value from the *TX* dataset is mainly due to the larger errors produced by *VSol* on strings in the *TX* dataset. Note that accurate selectivity estimation for approximate string search is a challenging problem in itself. For example, when being used as a stand-alone estimator, *VSol* on average has a relative error between 0.3 to 0.9, depending on the dataset used, as has been shown in [33]. Thus, relatively speaking, our algorithm by combining the insights from both the spatial and string distributions, works very well in practice.

Since the adaptive estimator has a slightly better accuracy on the R-tree, in the sequel, we concentrate on the results from the R-tree; the results from the MHR-tree are very similar.

The higher accuracy delivered by using a larger number of buckets comes at the expense of space overhead and higher construction costs. Figure 10 investigates these issues in detail. Not surprisingly, the size of the selectivity estimator increases with more buckets, as shown in Figure 10(a). Since the *VSol* of each bucket b has to maintain the min-wise signatures of all distinct q-grams of the strings contained in b and each min-wise signature has a constant size, the size of each bucket b entirely depends on the distinct number of q-grams it contains (denoted as g_b). It is important to understand that the length of the inverted list for every q-gram does not affect the size of each bucket, as the list is not explicitly stored. When more buckets are used, the sum of g_bs over all buckets increases. This value will increase drastically when a large number of buckets is used, indicated by the point $k = 1500$ in Figure 10(a). However, even in that case, due to the constant size for the min-wise signatures, the overall size of the adaptive estimator is still rather small, below 25 MB for the CA and TX datasets for 2 million points. When $k = 1000$, the size of the adaptive estimator is about 6 MB for both datasets. On the other hand, the construction cost of the adaptive estimator is almost linear to the number of buckets k as shown in Figure 10(b). Note that the construction cost is a one time expense.

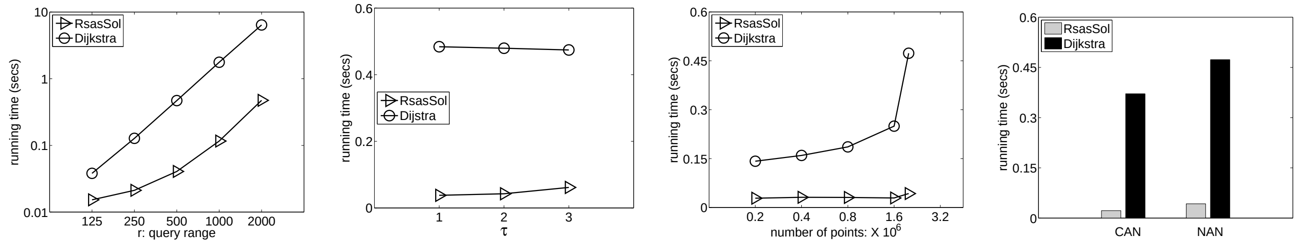
The small number of buckets and small size of the adaptive estimator indicates that it can be easily stored in memory for selectivity estimations. Thus, using it for selectivity estimation incurs much less cost than executing the query itself on disk-based datasets. We omitted these comparisons for brevity.

We further investigate the accuracy of the adaptive estimator when varying other parameters. Given the results from Figures 9 and 10, we set the default number of buckets to 1000. The results are reported in Figure 11. These results indicate that the adaptive estimator provides very good accuracy in a large number of different settings. In general, the adaptive estimator gives better accuracy in the *CA* dataset compared to the *TX* dataset. The estimation, with a fixed number of buckets, is more accurate for smaller query ranges (Figure 11(a)), smaller edit distance thresholds (Figure 11(b)), smaller datasets (Figure 11(c)), and lower dimensions (Figure 11(d)).

5.3 The RSAS queries

We study the effectiveness of the RSASSOL algorithm for RSAS queries in this section. Firstly, we investigate the effect of the selection and the number of reference nodes, and the number of subgraphs built by the *RPar* algorithm, in the pre-processing step, to the RSASSOL algorithm.

The impacts of reference nodes selection on the query performance come in two folds. First, different selection strategy will end up in selecting different reference nodes. Inspired by the results shown in [18] and our own observations, the *optimized planar* algorithm from [18] is always the best strategy, given the same number of reference nodes. Second, the number of reference nodes is also critical. Clearly, the construction time and space overhead of having more reference nodes expect to increase linearly with $|V_R|$ (since each node n has to compute and store $d(n, n_r)$ for every $n_r \in V_R$). This is exactly what we have observed in our experiments and these results were omitted for brevity. On the other hand, having more reference nodes, the lower and upper distance-bounds tend to be tighter, leading to better pruning. But,



(a) vary query range, NAN dataset. (b) vary τ, NAN dataset. (c) vary N, NAN dataset. (d) vary dataset.

Fig. 14. Query performance for RSAS queries.

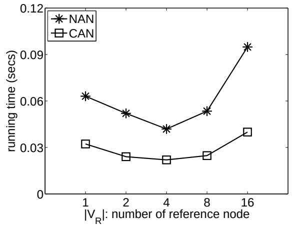


Fig. 12. |reference nodes|.

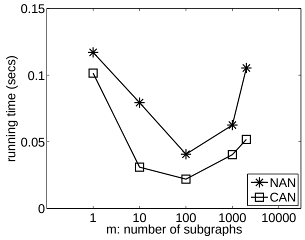


Fig. 13. |subgraphs|.

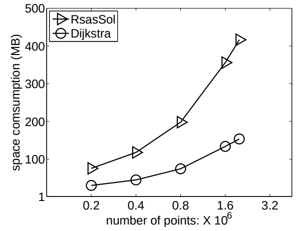


Fig. 15. Space of RSASSOL.

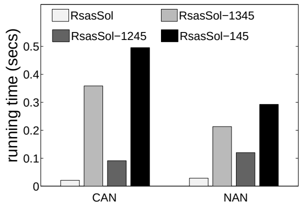


Fig. 16. Steps in RSASSOL.

more reference nodes also lead to higher computation costs to calculate the lower and upper distance-bounds during query processing. Therefore, we expect to see a sweet point of the overall running time when we vary $|V_R|$. Figure 12 shows exactly this trend using the average running time of 100 queries. It shows that 4 reference nodes provide the best running time for both the *NAN* and *CAN* datasets. Thus, we use $|V_R| = 4$ across the board.

Next, we study the effect of m, number of subgraphs, on the running time of RSASSOL (using the average of 100 queries). Figure 13 shows that, when $m \leq 100$, the running time decreases with more subgraphs. The reason is that more subgraphs can refine the intersection area with a query range, which avoids unnecessary string retrievals. However, having more and more subgraphs also means more access to smaller FilterTrees, which introduces query-overhead when searching for approximate strings. Eventually, such overheads dominate over the benefit of pruning more areas using more and smaller subgraphs. Hence, the query cost increases when $m > 100$ in Figure 13. For both *NAN* and *CAN* datasets, $m = 100$ achieves a good balance, which justifies our choice of the default value for m. *RPar* is very efficient for partitioning nodes into these subgraphs, and it is a one-time pre-processing cost. Hence, we omitted these results due to the space limitation.

Next, we study the query performance of RSASSOL compared against the baseline method, the *Dijkstra* solution, in Figure 14. The default dataset is *NAN* with 2 million points, since it has more nodes and edges.

Figure 14(a) shows the average running time when r varies from 125 to 2000. Clearly, RSASSOL outperforms Dijkstra, especially for larger query ranges; and the gap between the two widens when r increases. This is due to the fact that the cost of the Dijkstra solution is strictly proportional to the query range. On the other hand, the cost of the RSASSOL algorithm increases rather slowly because of the combined spatial and string-based pruning power. For example, when $r = 500$, Dijkstra is 10 times more expensive than RSASSOL; and this gap enlarges to more than one order of magnitude for larger ranges. In the follows, the default value of r is 500.

Figure 14(b) shows the effect of different τ values for $\tau = 1, 2, 3$. RSASSOL has always outperformed the Dijkstra solution. The next experiment, shown in Figure 14(c), investigates the scalability of both algorithms by varying the size of the dataset P on the road network, where $N = |P|$ changes from $200,000$ to $2,000,000$. Clearly, RSASSOL has much better scalability compared to the Dijkstra solution. For example, when N reaches 2 million, RSASSOL is 10 times faster than the Dijkstra solution.

Next, we tested their performance on the *CAN* dataset as well. The trends are very similar. Figure 14(d) compares the average running time of one query for RSASSOL and Dijkstra on *CAN* and *NAN*, side-by-side, using default values for r, τ, and N. RSASSOL outperforms Dijkstra by 10 to 15 times.

Figure 15 shows the space usage of RSASSOL compared against the *Dijkstra* solution, using the *NAN* dataset. Both of them introduce a space consumption that is linear to input datasets. RSASSOL has a higher space consumption since it additionally utilizes the FilterTrees to accelerate the approximate string matching and stores the distances from the nodes to the reference nodes (e.g. RDIST_i) for the third step pruning. Nevertheless, these overhead are still linear to the input datasets in the worst case. That said, the size of our storage structure for RSASSOL is less than 2.5 times of the space consumption for the *Dijkstra* solution (which has almost no space overhead) in the worst case when we have increased the number of points on the road network from 200,000 points to 2 million points.

Lastly, we demonstrate the effectiveness of different prunings used in RSASSOL in Figure 16. In particular, we show the pruning power of steps 2 and 3, namely the FilterTrees and the lower and upper distance bounds, on the datasets *CAN* and *NAN*. In Figure 16, RSASSOL-1345, RSASSOL-1245, and RSASSOL-145 stand for the RSASSOL solution without the pruning of step 2, or without step 3, or without both of steps 2 and 3, respectively. On both datasets, the step 2 is very effective, which prunes away most of the candidates. On *CAN*, step 3 prunes away about 30% candidates. However, step 3 has little pruning power on *NAN*. These results show that combining these different prunings seamlessly into one solution (i.e., the RSAS solution) is effective and necessary.

6 Related Work

The IR^2-tree was proposed in [17] to perform exact keyword search with kNN queries in spatial databases. The IR^2-tree cannot support spatial approximate string searches, neither their selectivity estimation was addressed therein. Authors in [45], [46] study the m-closest keywords query in Euclidean space, where the proposed bR*-tree cannot handle the approximate string search neither. Two other relevant studies appear in [7], [13] where ranking queries that combine both the spatial and text relevance to the query object were investigated.

Another related work appears in [2] where the LBAK-tree was proposed to answer location-based approximate-keyword queries which are similar to our definition of spatial approximate string queries in the Euclidean space. The basic idea in the LBAK-tree is to augment a tree-based spatial index (such as an R-tree) with q-grams of subtree nodes to support edit-distance based approximate string/keyword searches. The LBAK-tree was proposed after our study on SAS queries in the Euclidean space [44] (the conference version of this work), and it has been compared against the MHR-tree in [2]. Their results have shown that the LBAK-tree has achieved better query time than the MHR-tree, but using more space. Note that the LBAK-tree returns exact answers for the ESAS queries, and the MHR-tree returns approximate answers. We did not compare to the LBAK-tree with the MHR-tree in this work since detailed comparison of the two was already available in [2]. That said, for ESAS queries, the LBAK-tree should be adopted when exact answers are required; when space consumption must be small and approximate solutions are acceptable, the MHR-tree is the candidate.

To the best of our knowledge, RSAS queries and selectivity estimation of SAS queries have not been explored before.

Approximate string search alone has been extensively studied in the literature [3], [8], [9], [11], [19], [27], [30], [32], [36], [40], [42], [43]. These works generally assume a similarity function to quantify the closeness between two strings. There are a variety of these functions such as edit distance and Jaccard. Many approaches leverage the concept of q-grams. Our main pruning lemma is based upon a direct extension of q-gram based pruning for edit distance that has been used extensively in the field [19], [40], [42]. Improvements to the q-grams based pruning has also been proposed, such as v-grams [43], where instead of having a fixed length for all grams variable length grams were introduced, or the two-level q-gram inverted index [26].

Note that in the literature “approximate string matching” also refers to the problem of finding a pattern string approximately in a text, which is the problem surveyed in Navarro’s paper [34] in 2001. The problem in our paper is different: we want to search in a collection (unordered set) of strings to find those similar to a single query string (“selection query”). We used “approximate string search” to refer to our problem, see an excellent tutorial on this topic [22]; q-grams based solution for the edit-distance metric has become the norm, see [3], [9], [11], [27], [28], [30], [32], [36], [43] and references therein. The state-of-the-art in [30], [32] have conclusively shown that the q-grams based solution is the best method for the edit-distance based metric.

The q-grams based solution for edit-distance is also especially effective for the relatively short strings [30], which is the case for our problem (e.g. a large set of geo-tags rather than a long text document). That said, applying other metrics and/or other approximate string matching methods is definitely an interesting open problem to investigate.

Another well-explored topic is the selectivity estimation of approximate string queries [10], [23], [24], [28], [29], [33]. Most of them use the edit distance metric and q-grams to estimate selectivity. Other work uses clustering [25]. Our selectivity estimation builds on the *VSol* estimator proposed in [33]. Finally, special treatment was provided for selectivity of approximate string queries with small edit distance [28] and substring selectivity estimation was examined in [23], [29].

Our effort for selectivity estimation in ESAS queries is also related to selectivity estimation for spatial range queries [1], [20]. Typically, histograms and partitioning based methods are used. Our approach is based on similar principles but we also take into account the string information and integrate the spatial partitioning with the knowledge of string distribution. We did not address the selectivity estimation of RSAS queries. As discussed in Section 4.3, Selectivity estimation of range queries on road networks is a much harder problem than its counterpart in the Euclidean space. Several methods for selectivity estimation for range queries on road networks were proposed in [41]. However, they are only able to estimate the number of nodes and edges in the range (not the number of points residing on the network in the range). How to extend these techniques for points and combine them with string predicates presents interesting challenges for future work.

7 Conclusion

This paper presents a comprehensive study for spatial approximate string queries in both the Euclidean space and road networks. We use the edit distance as the similarity measurement for the string predicate and focus on the range queries as the spatial predicate. We also address the problem of query selectivity estimation for queries in the Euclidean space. Future work include examining spatial approximate sub-string queries, designing methods that are more update-friendly, and solving the selectivity estimation problem for RSAS queries.

References

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

[2] S. Alsubaiee, A. Behm, and C. Li. Supporting location-based approximate-keyword queries. In *GIS*, pages 61–70, 2010.

[3] A. Arasu, S. Chaudhuri, K. Ganjam, and R. Kaushik. Incorporating string transformations in record matching. In *SIGMOD*, pages 1231–1234, 2008.

[4] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[5] 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*, pages 322–331, 1990.

[6] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *STOC*, pages 327–336, 1998.

[7] X. Cao, G. Cong, and C. S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. *Proc. VLDB Endow.*, 3:373–384, 2010.

[8] K. Chakrabarti, S. Chaudhuri, V. Ganti, and D. Xin. An efficient filter for approximate membership checking. In *SIGMOD*, pages 805–818, 2008.

[9] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, pages 313–324, 2003.

[10] S. Chaudhuri, V. Ganti, and L. Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *ICDE*, pages 227–238, 2004.

[11] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 5–16, 2006.

[12] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, 1997.

[13] 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.

[14] E. D. Demaine, D. Emanuel, A. Fiat, and N. Immorlica. Correlation clustering in general weighted graphs. *Theoretical Computer Science*, 361(2):172–187, 2006.

[15] C. Ding and X. He. Cluster merging and splitting in hierarchical clustering algorithms. In *ICDM*, pages 139–146, 2002.

[16] M. Erwig and F. Hagen. The graph voronoi diagram with applications. *Networks*, 36:156–163, 2000.

[17] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, pages 656–665, 2008.

[18] A. V. Goldberg and C. Harrelson. Computing the shortest path: A^* search meets graph theory. In *SODA*, pages 156–165, 2005.

[19] 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*, pages 491–500, 2001.

[20] D. Gunopulos, G. Kollios, J. Tsotras, and C. Domeniconi. Selectivity estimators for multidimensional range queries over real attributes. *The VLDB Journal*, 14(2):137–154, 2005.

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

[22] M. Hadjieleftheriou and C. Li. Efficient approximate search on string collections. *PVLDB*, 2(2):1660–1661, 2009.

[23] H. V. Jagadish, R. T. Ng, and D. Srivastava. Substring selectivity estimation. In *PODS*, pages 249–260, 1999.

[24] L. Jin and C. Li. Selectivity estimation for fuzzy string predicates in large data sets. In *VLDB*, pages 397–408, 2005.

[25] L. Jin, C. Li, and R. Vernica. Sepia: estimating selectivities of approximate string predicates in large databases. *The VLDB Journal*, 17(5):1213–1229, 2008.

[26] M.-S. Kim, K.-Y. Whang, J.-G. Lee, and M.-J. Lee. n-gram/2l: a space and time efficient two-level n-gram inverted index structure. In *VLDB*, pages 325–336, 2005.

[27] N. Koudas, A. Marathe, and D. Srivastava. Flexible string matching against large databases in practice. In *VLDB*, pages 1078–1086, 2004.

[28] H. Lee, R. T. Ng, and K. Shim. Extending q-grams to estimate selectivity of string matching with low edit distance. In *VLDB*, pages 195–206, 2007.

[29] H. Lee, R. T. Ng, and K. Shim. Approximate substring selectivity estimation. In *EDBT*, pages 827–838, 2009.

[30] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[31] F. Li, B. Yao, M. Tang, and M. Hadjieleftheriou. Spatial approximate string search. Technical report, School of Computing, University of Utah, 2011. http://www.cs.utah.edu/~lifeifei/papers/sas.pdf.

[32] G. Li, J. Feng, and C. Li. Supporting search-as-you-type using sql in databases. *TKDE*, To Appear, 2011.

[33] A. Mazeika, M. H. Böhlen, N. Koudas, and D. Srivastava. Estimating the selectivity of approximate string queries. *ACM TODS*, 32(2):12–52, 2007.

[34] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33:31–88, 2001.

[35] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB*, pages 802–813, 2003.

[36] S. Sahinalp, M. Tasan, J. Macker, and Z. Ozsoyoglu. Distance based indexing for string proximity search. In *ICDE*, pages 125–136, 2003.

[37] J. Sankaranarayanan and H. Samet. Distance oracles for spatial networks. In *ICDE*, pages 652–663, 2009.

[38] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *PVLDB*, 2:1210–1221, 2009.

[39] S. Shekhar and D. ren Liu. Ccam: A connectivity-clustered access method for networks and network computations. *IEEE Transactions on Knowledge and Data Engineering*, 9:410–419, 1997.

[40] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *ESA*, pages 327–340, 1995.

[41] E. Tiakas, A. N. Papadopoulos, A. Nanopoulos, and Y. Manolopoulos. Node and edge selectivity estimation for range queries in spatial networks. *Inf. Syst.*, 34:328–352, 2009.

[42] E. Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theor. Comput. Sci.*, 92(1):191–211, 1992.

[43] X. Yang, B. Wang, and C. Li. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *SIGMOD*, pages 353–364, 2008.

[44] B. Yao, F. Li, M. Hadjieleftheriou, and K. Hou. Approximate string search in spatial databases. In *ICDE*, pages 545 – 556, 2010.

[45] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *ICDE*, pages 688–699, 2009.

[46] D. Zhang, B. C. Ooi, and A. Tung. Locating mapped resources in web 2.0. In *ICDE*, pages 521–532, 2010.

PLACE
PHOTO
HERE

Feifei Li received the BS degree in computer engineering from the Nanyang Technological University in 2002 and the PhD degree in computer science from the Boston University in 2007. He was an assistant professor in the Computer Science Department, Florida State University between 2007 and 2011. Since Aug 2011, He has been an assistant professor in the School of Computing, University of Utah. His research interests include databases, data management, and big data analytics.

PLACE
PHOTO
HERE

Bin Yao received the BS degree and the MS degree in computer science from the South China University of Technology in 2003 and 2007, and the PhD degree in computer science from the Florida State University in 2011. He has been an assistant professor in the Department of Computer Science and Engineering, Shanghai Jiao Tong University since 2011. His research interestes are management and indexing of large databases, and scalable data analytics.

PLACE
PHOTO
HERE

Mingwang Tang received the BS degree and the MS degree in computer science from the Sichuan University in 2006 and 2009 respectively. He was a PhD student in the Computer Science Department, Florida State University from Sep 2009 to Aug 2011. After that, he has been a PhD student in School of Computing, University of Utah, since Sep 2011. His research interests include query processing, query optimization, and data analytics for large data.

PLACE
PHOTO
HERE

Marios Hadjieleftheriou received the electrical and computer engineering diploma in 1998 from the National Technical University of Athens, Greece, and the PhD degree in computer science from the University of California, Riverside, in 2004. He has worked as a research associate at Boston University. Currently, he is working for AT&T Labs Research. His research interests are in the areas of databases and data management in general.

Electronic Appendix to “Spatial Approximate String Search”

Feifei Li *Member, IEEE*, Bin Yao, Mingwang Tang, Marios Hadjieleftheriou

Appendix A: Proof of Theorem 1

Proof: When $\eta_b = 1$ in Equation 12, our problem is identical to the following problem (Definitions $1, 2$ in [3]): given a set P of points, we want to find k partitions of P with k MBRs to minimize the information loss of P. Each partition is represented by the MBR and the number of points in the MBR. The information loss for a point p in the ith partition b_i is defined as the amount of “uncertainty” in each dimension of p, which is $\sum_{1,\dots,d} X_i$. And the information loss of P is the summation of information loss for all points in P. This problem is proved to be an NP-hard problem (Theorem 2 in [3]) by reducing the PLANAR 3-SAT, which is an NP-complete problem [2], to this problem.

In a special case of our problem, we assume that: (i) all associated strings are identical, then $\eta_{b_i} = 1$ for any b_i; (ii) $d = 2$. Then Equation 12 is equivalent to the definition of information loss for each bucket in the above. Hence, any instance of the above problem, which is NP-hard, can be reduced to an instance of our problem. □

Appendix B: Esas Selectivity Estimator: The Greedy Algorithm

We first illustrate our main ideas using a simple, greedy principle. The algorithm proceeds in multiple iterations. In each iteration, one bucket is produced. At every iteration, we start with a seed, randomly selected from the unassigned points (the points that have not been covered by existing buckets). Let $\Pi(P)$ and $\Theta(P)$ be the perimeter and area of the MBR that encloses a set of points P, and $\overline{P}$ be the unassigned points at any instance.

At the beginning of the i-th iteration $\overline{P} = P - \bigcup_{j=1}^{i-1} \mathbf{b}_{j,p}$. We also find the number of neighborhoods $\overline{\eta}$ in $\overline{P}$ and store the memberships of points in these neighborhoods. For the i-th iteration, we initialize $\mathbf{b}_{i,p}$ with a seed randomly selected from $\overline{P}$, and set $n_i = 1$ and $\eta_i = 1$. Then, we try to add points to $\mathbf{b}_{i,p}$ from $\overline{P}$ in a greedy fashion. Whenever we remove a point $p \in \overline{P}$ and add it to $\mathbf{b}_{i,p}$, we update n_i, η_i, and $\overline{\eta}$ accordingly. We do not recompute the neighborhoods in $\overline{P}$ after a point is moved. Rather, we simply remove the corresponding point from the already computed neighborhood. Since we have stored the neighborhoods, updating $\overline{\eta}$ after removing a point is easy, i.e., we simply decrease $\overline{\eta}$ by one when the last point for some neighborhood has been removed. We re-compute the neighborhoods of $\mathbf{b}_{i,p}$ after inserting a new point, or we can simply add new points to existing neighborhoods and rebuild neighborhoods periodically.

At any step, the total amount of “uncertainty” caused by the current configuration of b_i and $\overline{P}$ is estimated as:

$$U(b_i) = \eta_i \cdot n_i \cdot \Pi(\mathbf{b}_{i,p}) + \frac{\overline{\eta} \cdot |\overline{P}|}{k-i} \cdot \left(\frac{\Theta(\overline{P})}{k-i}\right)^{1/d} \cdot d \qquad (14)$$

In the above equation, the first term is simply $\Delta(b_i)$. The second term estimates the “uncertainty” for the remaining buckets by assuming that unassigned points are evenly and uniformly distributed into the remaining $k - i$ buckets. More specifically, each remaining bucket has an extent of $\left(\frac{\Theta(\overline{P})}{k-i}\right)^{1/d} \cdot d$, i.e., a square with an area of $\frac{\Theta(\overline{P})}{k-i}$. It has $|\overline{P}|/(k-i)$ points and $\overline{\eta}\frac{\Theta(\overline{P})/(k-i)}{\Theta(\overline{P})}$ neighborhoods, where $\Theta(\overline{P})/(k-i)$ is the average area of each bucket (i.e., the number of neighborhoods an unassigned bucket covers is proportional to its area). Based on this information, we can estimate $\Delta(b)$ for any unassigned bucket over $\overline{P}$ and sum over all of them yields the second term in Equation 14.

When deciding which point from $\overline{P}$ to add into b_i, we iterate through every point $p_j \in \overline{P}$. For each p_j, let b_i^j be a bucket containing points $\mathbf{b}_{i,p} \cup \{p_j\}$ and $\overline{P}' = \overline{P} - \{p_j\}$. We compute $U(b_i^j)$ with $\overline{P}'$ as the unassigned points according to Equation 14. We select the point p_j with the minimum $U(b_i^j)$ among all points satisfying $U(b_i^j) < U(b_i)$ and move p_j from $\overline{P}$ to b_i. If no such point exists, i.e., for all p_j, $U(b_i^j) \geq U(b_i)$, we let the current b_i be the i-th bucket and proceed with bucket $i+1$ by repeating the same process. The intuition is that further quality improvement is not likely to occur by adding more points to the i-th bucket. After the $(k-1)$-th bucket is constructed, the unassigned points form the k-th bucket. If fewer than k buckets are created, we find all buckets with more than one neighborhood and repeat the process. If for a given bucket no more than one point can be added (an outlier), we select a different seed.

The greedy algorithm has $k - 1$ rounds. In each round, we have to repeatedly check all points from $\overline{P}$ and select the best

- *Feifei Li and Mingwang Tang are with the School of Computing, University of Utah. E-mail: {lifeifei, tang}@cs.utah.edu. Bin Yao is with the Department of Computer Science and Engineering, Shanghai Jiao Tong University (contact author). E-mail: yaobin@cs.sjtu.edu.cn. Marios Hadjieleftheriou is with the AT&T Labs Research. E-mail: marioh@research.att.com.*

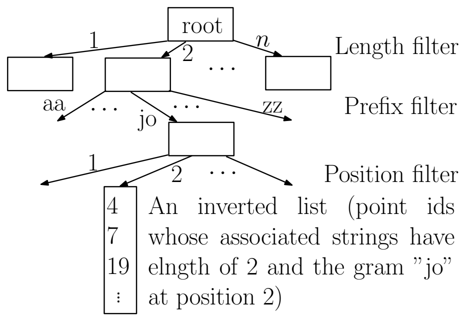

root
1
2
n
. . .
Length filter
aa
. . .
jo
. . .
zz
Prefix filter
1
2
. . .
Position filter
4
7
19
⋮
An inverted list (point ids whose associated strings have elngth of 2 and the gram "jo" at position 2)

Fig. 17. An example of a FilterTree from [1].

point to add to the current bucket. $\overline{P}$ has $O(N)$ size, and in the worst case, we may check every point $O(N)$ times. Hence, the complexity of the greedy algorithms is $O(kN^2)$.

Appendix C: Efficient Approximate String Search

In order to support the efficient approximate string search on a collection of strings, which is used as a component in our query algorithm, we leverage on the FilterTree from [1]. The FilterTree combines several string filters (e.g., length filter, position filter) with the q-gram inverted lists for a collection of strings and organizes these filters in a tree structure. An example is shown in Figure 17. The nodes in the first level of the tree are grouped by the string length. The second level nodes are grouped by different grams (in this example, 2-grams). Then we decide the children of each gram by the position of that gram in the strings. Each such child node maintains an inverted list of string ids (that contain this gram at a particular position with a given length). We integrate the FilterTree with our disk-based storage model. Specifically, in each inverted list, we store the point ids that correspond to the associated strings. In query processing, these point ids can be used to search the B+-tree of the points file to retrieve the corresponding strings and other information.

To answer an approximate string query on the collection of strings indexed by the FilterTree, we traverse the FilterTree from the root to leaves. We retrieve those inverted lists that satisfy the length, prefix, and position filters. Then we call one of the merging algorithms in [1], so that strings contain enough number of common q-grams to the query string can be identified (an improved version of Lemma 1 by incorporating the position and length filters as well).

Appendix D: Other Issues

Multiple strings. In the general case, points in P and/or the query may contain multiple strings. For RSAS queries, as long as a point has at least one string similar to the query string, it will be returned for further processing. For a data point with multiple strings in ESAS queries, we build one min-wise signature for each string and take the union of these signatures when computing the signature for the leaf node containing this point. For a query with multiple strings and corresponding thresholds (conjunctive logic), we apply the string-related prunings in Algorithms 1 and 2 for each query string. As soon as there is one string that does not satisfy the pruning test, the corresponding node/point can be pruned.

Other spatial query types. It is possible to adapt our query processing techniques to work with other spatial query types. In this work, we show how to do this for nearest neighbor queries, and leave the study of other query types as interesting future works. Both our solutions in the Euclidean space and the road networks are spatial-oriented, i.e., the underlying structures of our algorithms are constructed based on spatial queries, rather than string queries. Hence, they can easily answer standard spatial queries (by just ignoring the string predicate pruning), or be adapted to answer an approximate string search integrated with other spatial query types (instead of the range query we have investigated). Note that for this reason, these approaches are preferred than a *string solution* (as discussed in Section 1) in spatial databases. For example, using a *string solution* to answer an approximate string search with k nearest neighbor queries, one has to collect all points with similar strings to the query string, and then in the post-precessing step find the k nearest neighbors to the query point among this set. Without the help of a spatial index (since this set varies for different queries, it is not possible to build any structure in a pre-processing step prior to the query), this post-precessing step can be very expensive (especially when this set is large or data is in higher dimensions, both are likely in practice).

That said, to deal with nearest neighbor queries combined with approximate string search, our solution works as follows. In ESAS queries, one can also revise the kNN algorithm for the normal R-tree to derive the kNN-MHR algorithm. The basic idea is to use a priority queue that orders objects in the queue with respect to the query point using the MinDist metric. However, only nodes or data points that can pass the string pruning test (similar to lines 14-17 and lines 8-9 respectively in Algorithm 1) will be inserted into the queue. Whenever a point is removed from the head of the queue, it is inserted in $\mathcal{A}$. The search terminates when $\mathcal{A}$ has k points or the priority queue becomes empty.

For RSAS queries, one can also revise the RSASSOL algorithm to derive the kNN-RSASSOL algorithm. The basic idea is to use a priority queue to order subgraphs or points with respect to the possible minimum distance (i.e. the lower bound or exact network distance) to the query point. The entry in the priority queue can be a subgraph with distance to the query point, or a point with a lower-bound distance or the exact distance to the query point. If a subgraph is popped out, then we get the candidate set from line 4 in Algorithm 2) for this subgraph and calculate lower bounds for all the candidate points and push them into the priority queue. Whenever the priority queue pops out a point with the lower bound, then we apply the edit distance filtering on it. If it satisfies the string predicate, we calculate the exact network distance using the ALT algorithm and re-insert it to the priority queue. Otherwise, we can safely delete it. The search terminates when k points with exact distances are popped out or the priority queue becomes empty. We can also utilize the MPALT algorithm by popping out a certain amount of points at one time when we compute the exact distances.

Note that an in-depth study of the nearest neighbor queries combined with the approximate string search, that is not possible in this work due to the space limit, is still needed as a future work.

Updates. In the Euclidean space, coupling the R-tree nodes with the min-wise signatures in the MHR-tree complicates dynamic updates. For the insertion of a new object, we follow the R-tree insertion algorithm, then, compute the signature for the newly inserted point and union its signature with the signature for the leaf node that contains it, by taking the smaller value for each position in the two signatures. For those positions that the signature of the leaf node changes, the changes propagate to the parent nodes in similar fashion. The propagation stops when the values of the signature on the affected positions from the children node are no longer smaller than the corresponding elements for the signature of the parent. On the other hand, deletion is a bit more involved. If some positions in the signature of the deleted point have the same values as the corresponding positions in the signature of the leaf node that contains the point, then we need to find the new values for these positions, by taking the smallest values from the corresponding positions of the signatures of all points inside this node. These updates may propagate further up in the tree and a similar procedure is needed as that in the insertion case. It is important to note here that the addition of the min-wise signatures does not affect the update performance of the R-tree since signature updates never result in structural updates. Hence, the update properties and performance of the MHR-tree is exactly the same as that of the R-tree (subject to a very small computation overhead of computing the signatures, linear hashing, of some affected nodes).

Second, maintaining good selectivity estimators under dynamic updates is a challenging problem in general. Most existing work for either spatial-alone or string-alone selectivity estimation (see Section 6) concentrate on static datasets. For our estimator, we can simply update the number of points that fall into a bucket as well as adjust the shape of the bucket by the changes of the MBRs of the R-tree nodes it contains. However, the underlying neighborhoods in one bucket may shift over time. Hence, the initial bucketization may no longer reflect the actual data distribution. We can rebuild the buckets periodically, e.g., after a certain number of updates, and the in-depth examination of this problem is left as a future work.

Lastly, the updates of the disk-based storage model for the RSAS queries mostly just follow the update algorithms for an B+-tree and the FilterTree. We omit the details.

Appendix E: Acknowledgment

Mingwang Tang and Feifei Li were supported in part by NSF Grant IIS-1212310. Feifei Li was also supported in part by an 2011-2012 HP IRP award.

References

[1] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[2] D. Lichtenstein. Planar formulae and their uses. *SIAM J. Comput.*, pages 329–343, 1982.

[3] K. Yi, X. Lian, F. Li, and L. Chen. The world in a nutshell: Concise range queries. *TKDE*, 23:139–154, 2011.