Approximate String Search in Spatial Databases

Bin Yao[1], Feifei Li,[1] Marios Hadjieleftheriou[2], Kun Hou[1]
[1]*Computer Science Department, Florida State University, Tallahassee, FL, USA*
[2]*AT&T Labs Research, Florham Park, NJ, USA*
[1]{yao,lifeifei,hou}@cs.fsu.edu, [2]marioh@research.att.com

***Abstract*— This work presents a novel index structure, MHR-tree, for efficiently answering approximate string match queries in large spatial databases. The MHR-tree is based on the R-tree augmented with the min-wise signature and the linear hashing technique. 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 set resemblance between the query string and the q-grams from the sub-trees of index nodes. MHR-tree supports a wide range of query predicates efficiently, including range and nearest neighbor queries. We also discuss how to estimate range query selectivity accurately. We present a novel adaptive algorithm for finding balanced partitions using both the spatial and string information stored in the tree. Extensive experiments on large real data sets demonstrate the efficiency and effectiveness of our approach.**

I. Introduction

Keyword search over a large amount of data is an important operation in a wide range of domains [22]. 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 practical, real-world applications, and proposed the IR^2-Tree. A major limitation of the IR^2-Tree is that it only supports efficient keyword search with exact matches. In reality, for many scenarios, keyword search for retrieving approximate string matches is required [5], [10], [12], [28], [29], [31], [34], [37]. Since exact string 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 could be necessary when users have a fuzzy search condition or simply 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, including range and nearest neighbor queries. An example for the approximate string match range query is shown in Figure 1, depicting a common scenario in location-based services: find all objects within a spatial range r that have a description that is similar to "theatre". Similar examples could be constructed for k nearest neighbor (kNN) queries. We refer to these queries as *Spatial Approximate String* (SAS) queries.

A key issue in understanding the semantics of these queries is to define the similarity between two strings. The *edit distance* metric is often adopted for such approximate string queries [5], [10], [12], [28], [29], [31], [34], [37]. 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 σ_1 ='theatre' and σ_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 $O(n_1 n_2)$.

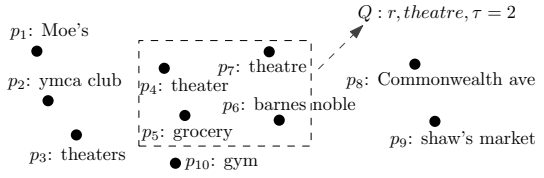


Fig. 1. Approximate string search with a range query.

A straightforward solution to any SAS query is to simply use any existing techniques for answering the spatial component of a SAS query and verify the approximate string match predicate either in a post-processing step or on the intermediate results of the spatial search. This means that for a SAS range query with a query range *r* and a query string σ over a spatial data set *P*, one could use an R-tree to index points in *P* and find all points $\mathcal{A}_c$ that fall into range *r*. Finally, for each point from $\mathcal{A}_c$ the similarity of its associated string will be compared against σ. Similarly, in a SAS kNN query with a query point t and a query string σ, one could use the same R-tree index and apply the normal kNN search algorithm w.r.t. t, then evaluate the similarity between the query string σ and the candidate nearest neighbors encountered during the search. The search terminates when k points with strings that satisfy the string similarity requirement have been retrieved. We generally refer to these approaches as the *R-tree solution*. Another straightforward solution is to build a string matching index and evaluate the string predicate first, 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 refer to this solution as the string index approach.

While being simple, the 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 a SAS 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 a SAS range query with a query string that does not have any similar strings within its query range from set P, $\mathcal{A} = \emptyset$. 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 that indexes P. 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. The fundamental issue here is that the possible pruning power from the string match predicate has been completely ignored by the R-tree solution. Similar arguments hold for the string index approach, where a query might retrieve a very large number of similar strings only to prune everything based on the spatial predicate at post-processing. Clearly, in practice none of these solutions will work better than a combined approach that prunes simultaneously based on the string match predicate and the spatial predicate.

Another interesting problem in this context is the *selectivity estimation* for SAS queries. The goal is to accurately estimate $|\mathcal{A}|$ 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 queries [3], [32].

Motivated by these observations, this work proposes a novel index structure that takes into account the potential pruning capability provided by the string match predicate and the spatial predicate simultaneously. The main challenge for this problem is that the basic solution of simply integrating approximate edit distance evaluation techniques (e.g., based on q-grams [8], [18] or tries [9], [24]) into a normal R-tree is expensive and impractical.

Our main contributions are summarized as follows:

- We formalize the notion of *spatial approximate string queries* and *selectivity estimation for spatial approximate string range queries* in Section II.
- We introduce a new index for answering SAS queries efficiently in Section IV. We first present the problems associated with storing q-grams for approximate edit distance evaluation directly into the R-tree. Then, we propose the solution of embedding min-wise signatures of q-grams into the R-tree and convert the problem into that of evaluating set resemblance.
- We present a novel algorithm that builds a robust selectivity estimator for SAS range queries in Section V. Our idea is to leverage an adaptive algorithm that finds balanced partitions of nodes from any R-tree based index, including the MHR-tree, 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 discuss issues related with associating multiple strings with query and data points, other spatial query types, and dynamic updates in Section VI.
- We demonstrate the efficiency, practicality and effectiveness of the MHR-tree for answering SAS queries, as well as the selectivity estimation algorithms for SAS range queries using a comprehensive experimental evaluation in Section VII. Our experimental evaluation covers both synthetic and real data sets of up to 10 millions points and 6 dimensions.

We introduce the basic tools used in our construction in Section III and survey the related work in Section VIII. The paper concludes with Section IX.

II. 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, we simply assume that each point in P has ω number of strings. Hence, a data set P with N points is the following set: $\{(p_1, \sigma_{1,1}, \ldots, \sigma_{1,\omega}), \ldots, (p_N, \sigma_{N,1}, \ldots, \sigma_{N,\omega})\}$. 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 strings.

A *spatial approximate string* (SAS) query Q consists of two parts: the spatial query Q_r and the string query Q_s. The spatial query specifies a spatial predicate and predicate parameters. In this paper we concentrate on the classical *range* and *nearest neighbor* predicates. A range query Q_r is simply defined by a query rectangle r; a kNN query Q_r is defined by a pair (t, k) where t is a point and k is an integer. The string query is defined by one or more query strings and their associated edit distance thresholds, i.e., $Q_s = \{(\sigma_1, \tau_1), \ldots, (\sigma_\beta, \tau_\beta)\}$, where $\forall i \in [1, \beta]$, σ_i is a string and $\tau_i \in \mathbb{N}$.

Let the set $\mathcal{A}_s = \{p_x | p_x \in P \wedge \forall i \in [1, \beta], \exists j_i \in [1, \omega], \varepsilon(\sigma_i, \sigma_{x,j_i}) \leq \tau_i\}$, i.e., $\mathcal{A}_s$ is the subset of points in P such that each point in $\mathcal{A}_s$ has one or more similar matching strings *for every query string*, with respect to the corresponding edit distance threshold. We define the SAS range and kNN queries as follows:

Definition 1 (SAS range query) A SAS range query $Q : (Q_r = r, Q_s)$ retrieves all points (denoted as $\mathcal{A}$) in P such that, $\mathcal{A} \subseteq P$ and,

$$\forall p \in \mathcal{A} \begin{cases} p \in r & p \text{ is contained in } r; \\ p \in \mathcal{A}_s & p \text{ has similar strings to all query strings.} \end{cases}$$

Definition 2 (SAS kNN query) Let $\|p, t\|$ be the Euclidean distance between points p and t. A SAS kNN query $Q : (Q_r = t, Q_s)$ retrieves a subset of points $\mathcal{A}$ from $\mathcal{A}_s$ such that,

$$\begin{cases} \mathcal{A} = \mathcal{A}_s & \text{if } |\mathcal{A}_s| \leq k; \\ |\mathcal{A}| = k \text{ and} \\ \forall p' \in \mathcal{A}, \forall p'' \in \mathcal{A}_s - \mathcal{A}, \|p', t\| < \|p'', t\| & \text{if } |\mathcal{A}_s| > k. \end{cases}$$

A SAS kNN query finds the top-k closest points to the query point t that have one or more similar matching strings *for every*

query string, with respect to the corresponding edit distance threshold. When there are less than k such points in P, the SAS kNN query simply finds all of them.

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

In our definition, we require that the returned points have one or more matching strings for every query string. We could certainly generalize this to any logical expression, for example, define $\mathcal{A}_s$ to be those points that have *at least* κ matching string(s) to *some of the query strings*. Our techniques could be generalized to handle any logical expression on the "matching" definition to the set of query strings. We discuss such generalization in Section VI.

To simplify our discussion in the rest we assume that $\omega = 1$ and $\beta = 1$. In this case, for every point $p \in P$, we simply use σ_p to denote its associated string; and for the string query component of a SAS query, we simply let σ and τ to denote the query string and the edit distance threshold respectively. We also assume that the data set P is static. Extending our techniques to the general case and dealing with dynamic updates will be discussed in Section VI.

III. PRELIMINARIES

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

A. 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], [12], [32]. 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 $\{\#t, th, he, ea, at, te, er, r\$\}$.

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 [18], [36] 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 [18]] For strings σ_1 and σ_2 of length $|\sigma_1|$ and $|\sigma_2|$, if $\varepsilon(\sigma_1, \sigma_2) = \tau$, then $|G_{\sigma_1} \cap G_{\sigma_2}| \geq \max(|\sigma_1|, |\sigma_2|) - 1 - (\tau - 1) * q$.*

B. The min-wise signature

The min-wise independent families of permutations were first introduced in [7], [13]. A family of *min-wise independent permutations* $\mathcal{F}$ must satisfy the following. 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:

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

In the above formula, $\min\{\pi(A)\} = \min\{\pi(x)|x \in A\}$. 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 [7] 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:

$$s(A) = \{\min\{\pi_1(A)\}, \min\{\pi_2(A)\}, \ldots, \min\{\pi_\ell(A)\}\}, \quad (1)$$

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, $A_1, \ldots, A_k$:

$$\widehat{\rho}(A_1, \ldots, A_k) = \frac{|\{i| \min\{\pi_i(A_1)\} = \cdots \min\{\pi_i(A_k)\}\}|}{\ell}. \quad (2)$$

Implementation of min-wise independent permutations requires generating random permutations of a universe and Broder et al. [7] 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 [7]. Let

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

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

IV. THE MHR-TREE

Suppose the disk block size is B. The R-tree [20] and its variants (R*-tree in particular [6]) share a similar principle. They first group $\leq B$ points that are in spatial proximity with each other into a minimum bounding rectangle (MBR); these points will be stored in a leaf node. The process is repeated until all points in P are assigned into MBRs and the leaf level of the tree is completed. 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. An example of an R-tree is illustrated in Figure 2.

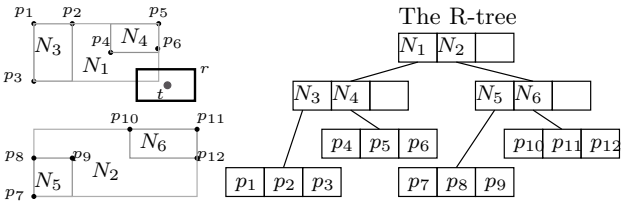


Fig. 2. The R-tree.

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. A kNN query t is answered by either the depth-first [33] or the best-first [21] approach. These algorithms follow the branch and bound principle [6], [20]. The MinDist metric between MBR(u) and t (the minimum possible distance from any objects inside u to t) is used to maintain a min priority queue to organize the search order among R-tree nodes [21].

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 or it needs to be explored based on the MinDist metric for the kNN query. 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 pruning based on string edit distance into the R-tree by storing sets G_u for all R-tree nodes u. However, the problem with this approach is that G_u becomes extremely large for nodes located in higher levels of the R-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. For large data sets, the overhead of storing large number of q-grams in an R-tree node significantly out-weights the potential savings offered by the pruning based on these q-grams.

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 1; 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.

A. 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 $s(G_p)$. In order to compute the min-wise signature for the node, a straightforward solution is to first compute $G_u = \bigcup_{p \in \mathbf{u}_p} G_p$ and then obtain $s(G_u)$ based on the q-gram set G_u. However, this approach requires storing all G_ps as intermediate results and computes the set union of a potentially large number of sets. Based on the definition of the min-wise signature, we can do this much more efficiently. Specifically, let $s(A)[i]$ be the ith element for the min-wise signature of 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 1, one can easily derive that:

$$s(A_1 \cup \cdots \cup A_k)[i] = \min\{s(A_1)[i], \ldots, s(A_k)[i]\}, \quad (3)$$

for $i = 1, \ldots, \ell$, since each element in a min-wise signature always takes the smallest image for a set. This simple property has been leveraged by many works using min-wise independent permutations.

We can obtain $s(G_u)$ using Equation 3 and $s(G_p)$s for every point $p \in \mathbf{u}_p$, directly. Finally, 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, $G_u = \bigcup_{i=1,\ldots,f} G_{w_i}$. Hence, $s(G_u) = s(G_{w_1} \cup \cdots \cup G_{w_f})$; based on Equation 3, 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 $s(G_u)$, which requires computing and storing G_{w_i}s as intermediate results.

This procedure is recursively applied in a bottom-up fashion until the root node of the R-tree has been processed. The complete construction algorithm is presented in Algorithm 1.

B. 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.

Algorithm 1: CONSTRUCT-MHR(Data Set P, Hash Functions $\{h_1, \ldots, h_\ell\}$)

```
1  Use any existing bulk-loading algorithm A for R-tree;
2  Let u be an R-tree node produced by A over P;
3  if u is a leaf node then
4      Compute G_p and s(G_p) for every point p ∈ u_p;
5      Store s(G_p) together with p in u;
6  else
7      for every child entry c_i with child node w_i do
8          Store MBR(w_i), s(G_{w_i}), and pointer to w_i in c_i;
9      for i = 1, ..., ℓ do
10         Let s(G_u)[i] = min(s(G_{w_1})[i], ..., s(G_{w_f}[i]));
11         Store s(G_u) in parent of u;
```

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 in Algorithm 1. 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 3. Next, we estimate the set resemblance between G and G_σ, $\rho(G, G_\sigma)$ as follows:

$$\widehat{\rho}(G, G_\sigma) = \frac{|\{i | \min\{h_i(G)\} = \min\{h_i(G_\sigma)\}\}|}{\ell}. \quad (4)$$

Equation 4 is a direct application of Equation 2. Note that:

$$\rho(G, G_\sigma) = \frac{|G \cap G_\sigma|}{|G \cup G_\sigma|} = \frac{|(G_u \cup G_\sigma) \cap G_\sigma|}{|(G_u \cup G_\sigma) \cup G_\sigma|} = \frac{|G_\sigma|}{|G_u \cup G_\sigma|}. \quad (5)$$

Based on Equations 4 and 5 we can estimate $|G_u \cup G_\sigma|$ as:

$$\widehat{|G_u \cup G_\sigma|} = \frac{|G_\sigma|}{\widehat{\rho}(G, G_\sigma)}. \quad (6)$$

Finally, we can estimate $\rho(G_u, G_\sigma)$ by:

$$\widehat{\rho}(G_u, G_\sigma) = \frac{|\{i | \min\{h_i(G_u)\} = \min\{h_i(G_\sigma)\}\}|}{\ell}. \quad (7)$$

Note that $\rho(G_u, G_\sigma) = |G_u \cap G_\sigma| / |G_u \cup G_\sigma|$. Hence, based on Equations 6 and 7 we can now estimate $|G_u \cap G_\sigma|$ as:

$$\widehat{|G_u \cap G_\sigma|} = \widehat{\rho}(G_u, G_\sigma) * \widehat{|G_u \cup G_\sigma|}. \quad (8)$$

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 $\mathcal{A}_s$, since they cannot produce any points from $\mathcal{A}$ (recall that $\mathcal{A} \subseteq \mathcal{A}_s$). 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 VII 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 SAS range query algorithm is presented in Algorithm 2. When the object is a data point (line 6), we can obtain

Algorithm 2: RANGE-MHR(MHR-tree R, Range r, String σ, int τ)

```
1  Let B be a FIFO queue initialized to ∅, let A = ∅;
2  Let u be the root node of R; insert u into B;
3  while B ≠ ∅ do
4      Let u be the head element of B; pop out u;
5      if u is a leaf node then
6          for every point p ∈ u_p do
7              if p is contained in r then
8                  if |G_p ∩ G_σ| ≥ max(|σ_p|, |σ|) − 1 − (τ − 1) * q then
9                      if ε(σ_p, σ) < τ then
10                         Insert p in A;
11     else
12         for every child entry c_i of u do
13             if r and MBR(w_i) intersect then
14                 Calculate s(G = G_{w_i} ∪ G_σ) based on
                   s(G_{w_i}), s(G_σ) and Equation 3;
15                 Calculate |G_{w_i} ∩ G_σ| using Equation 8;
16                 if |G_{w_i} ∩ G_σ| ≥ |σ| − 1 − (τ − 1) * q then
17                     Read node w_i and insert w_i into B;
18 Return A.
```

$|G_p \cap G_\sigma|$ 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 8. When the object is an R-tree node, we apply Equation 8 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).

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 2) 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.

V. SELECTIVITY ESTIMATION FOR SAS RANGE 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 [26], [29], [32]. A state of the art technique based

on q-grams and min-wise signatures is *VSol* [32]. 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 exactly L lists, for all M choose L combinations of lists. Denote the set of string ids that appear in all lists in the i-th combination with L_i, $1 \leq i \leq \binom{M}{L}$. The *L-M* similarity is defined as:

$$\rho_{LM} = |\cup L_i|. \tag{9}$$

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 L choose M 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 [32] and are beyond the scope of this paper.

A key observation in [32] 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 existing works on spatial databases [3], [19], which mostly differ on how buckets are formed.

A straightforward solution for our problem is to simply build a set of buckets $\{b_1, \ldots, b_k\}$ for some budget k based on an existing spatial range query selectivity estimator. 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 based on the min-wise signatures of the q-gram inverted lists of the strings contained in the bucket. Finally, the selectivity estimation for a SAS range 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 with σ, ρ^i_{LM}, using *VSol*. Let $\mathcal{A}_{b_i}$ denote the set of points from b_i that satisfy Q, then $|\mathcal{A}_{b_i}|$ is estimated as:

$$\widehat{|\mathcal{A}_{b_i}|} = n_i \frac{\Theta(b_i, r)}{\Theta(b_i)} \frac{\rho^i_{LM}}{n_i} = \frac{\Theta(b_i, r)}{\Theta(b_i)} \rho^i_{LM}. \tag{10}$$

The above basic solution assumes independence between the spatial locations of points and the strings associated with those points. Naturally, a better approach is to minimize the reliance on independence as much as possible to improve accuracy. Our challenge thus becomes how to integrate the minimum number of neighborhoods principle from *VSol* into the spatial uniformity principle effectively when building buckets.

A. 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 [15]) 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:

$$\Delta(b) = \eta_b n_b \sum_{1,\ldots,d} X_i \tag{11}$$

Intuitively, $\Delta(b)$ measures the total "uncertainty" of all points p 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 10. Intuitively, the larger the perimeter of a bucket, the more error the spatial estimation for the point's location introduces (the fact that it uses the intersection of the area of b and r); 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. For uniform (in terms of both the coordinates and strings) data, n_b and η_b are constant. Hence, the problem becomes grouping the points in k buckets such that the sum of perimeters of the buckets is minimized (which is exactly the same heuristic as the one used by the R*-tree partitioning algorithm). For non-uniform data this problem is non-trivial.

Once such k buckets are found, we build and maintain their *VSol* estimators and use the method illustrated in Equation 10 to estimate the selectivity. Unfortunately, we can show that for $d > 1$, this problem is NP-hard. Specifically (the proof will appear in the full version of the paper due to space constraints):

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.

B. The Greedy Algorithm

The first heuristic is based on a simple, top-down, greedy principle. We term this algorithm *Top-Down Greedy*. 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 $\overline{P}$. More importantly, we 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:

$$U(b_i) = \eta_i \cdot n_i \cdot \Pi(\mathbf{b}_{i,p}) + \frac{\overline{\eta}}{k-i} \cdot |\overline{P}| \cdot \left(\frac{\Theta(\overline{P})}{k-i}\right)^{1/d} \cdot d \quad (12)$$

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 $\left(\frac{\Theta(\overline{P})}{k-i}\right)^{1/d} d$, 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 summing over all of them yields the second term in Equation 12.

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 12. 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 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 $O(kN^2)$.

Another greedy strategy, that we mention here briefly, is to start with N buckets, one point per bucket and start merging buckets in a bottom-up fashion. We term this algorithm *Bottom-Up Greedy*. The objective function is to minimize $U(b_i) = \sum_{j=1}^{N-i} \Delta(b_j)$. Notice that initially, $U(b_0)$ is as small as possible (one point per bucket, one neighborhood per bucket, and zero perimeter), and $U(b_i)$ is a monotone increasing function for increasing i. During the i-th iteration we merge the two buckets that lead to the smallest $U(b_{i+1})$, until we reach a total of k buckets. We have a total of $N-k$ iterations and at each iteration we evaluate at most N^2 merges. Hence, the run-time complexity of the algorithm is $O(N^3)$.

C. The Adaptive R-tree Algorithm

Directly applying the greedy algorithm on a large spatial database can be 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 R and a budget k, descending from the root node of the R-tree, we find the first level in the R-tree that

has more than k nodes, say it has $\kappa > k$ nodes $\{u_1, \ldots, u_\kappa\}$. Then, 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 [16]. For remaining nodes, we assume that they are grouped into $k - i$ buckets in a uniform and independent fashion. Let remaining nodes be $\{u_{x_1}, \ldots, u_{x_\ell}\}$ 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 bucket as follows:

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

In Equation 13, $\Theta(\bigcup_{i=1}^{\ell} \mathbf{u}_{x_i,p})$ is simply the area for the MBR of the remaining points covered by the remaining nodes $\{u_{x_1}, \ldots, u_{x_\ell}\}$. Note that 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 a SAS range query, we simply find the set of buckets that intersects with the query range r and estimate the selectivity using Equation 10.

VI. Other Issues

Multiple strings. In the general case, points in the data set P and the query may contain multiple strings. Extending our techniques to handle this case is straightforward. For a data point with multiple strings, we simply 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, we simply apply the pruning discussed in Algorithm 2 for each query string on every index node. As soon as there is one string that does not satisfy the pruning test, the corresponding node can be pruned.

Another interesting problem is to define the string query component for a SAS query using more general conjunction/disjunction semantics. A simple solution is to check each query string against the pruning condition as specified in Algorithm 2 and combine the results of these individual tests depending on the logical expression specified by the query.

Other spatial query types. Our query processing technique is flexible enough to be adapted to work with other spatial query types, for example, reverse nearest neighbor queries and skyline queries. On the other hand, our query selectivity techniques are designed specifically for range queries. Generalizing our partitioning technique to other spatial query types with approximate string matches may require leveraging different insights, specific to those query types.

Updates. Coupling the R-tree nodes with the min-wise signatures in the MHR-tree complicates dynamic updates. To support dynamic updates one needs to do the following. 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.

Lastly, maintaining good selectivity estimators under dynamic updates is a challenging problem in general. Most existing work (see Section VIII) concentrates on static data sets. 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. For this work, we simply propose to rebuild the buckets periodically, e.g., after a certain number of updates, and we will investigate this issue further in future work.

VII. Experimental Evaluation

We implemented the *R-tree solution*, the string index solution and the MHR-tree, using the widely adopted spatial index library [2]. We do not report any results for the string index since, first, it requires linear space with respect to the number of data q-grams and hence space-wise it is not competitive, and, its query performance was not up to par across the board. The adaptive R-tree algorithm for the selectivity estimator seamlessly works for both the R-tree and the MHR-tree. 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.

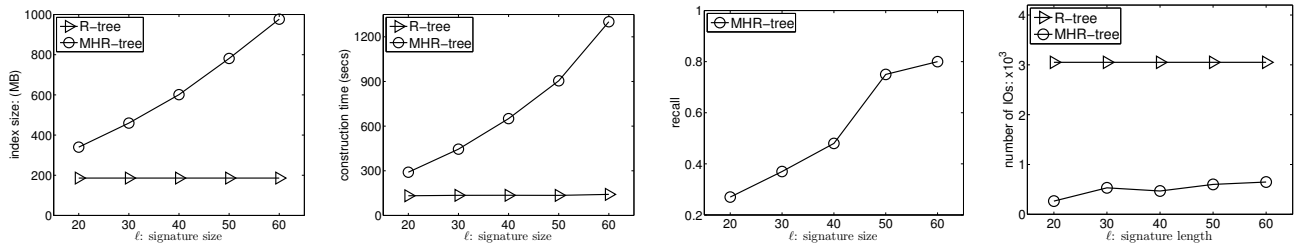
Data sets. The real data sets were obtained from the open street map project [1]. Each data set contains the road network and streets for a state in the USA. Each point has its longitude and latitude coordinates and several string attributes. We combine the state, county and town names for a point as its associated string. For our experiments, we have used the Texas (*TX*) and California (*CA*) data sets, since they are the largest few in size among different states. The *TX* data set has 14 million points and the *CA* data set has 12 million points. The real data sets are in two dimension. To test the performance of our algorithms on different dimensions we use two synthetic data sets. In the *UN* data set, points are distributed uniformly in the space and in the *RC* data set, points are generated with random clusters in the space. For both the *UN* and the *RC* data sets, we assign strings from our real data sets randomly to the spatial points generated. For all experiments the default size N of the data set P is 2 million. For the *TX* and *CA* data sets, we randomly sample 2 million points to create the default data sets. For all data sets the average length of the strings is approximately 14.

Setup. We concentrate on the SAS range queries. A range query r is generated by randomly selecting a center point t_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, for a range query, we select the query string as the associated string of the nearest neighbor of t_r from P. For both the MHR-tree and the *VSol*s used in our estimator our experiments indicate that two-grams work the best. Hence, the default q-gram length is 2. The default value for θ is 3%. The default size of the signature is $\ell = 50$ hash values (200 bytes), and the default edit distance threshold is $\tau = 2$. For all query-related experiments we report averages over 100 randomly generated queries.

A. The SAS Range Queries

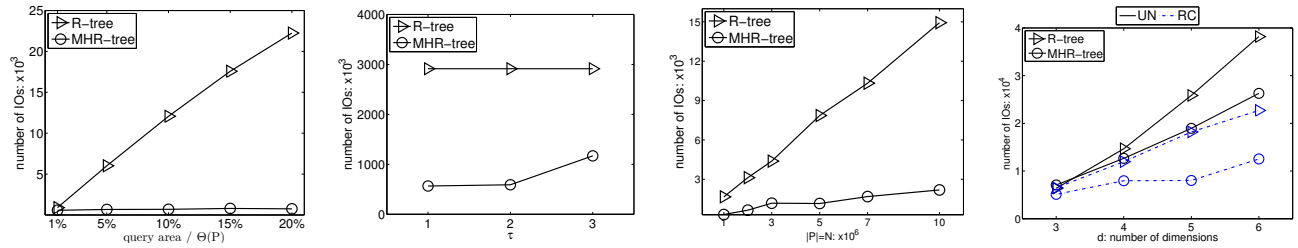
This section studies the effectiveness of the MHR-tree for the SAS range queries. We first study the impact of the signature size on the performance of the MHR-tree. Figure 3 summarizes the results using the *TX* data set. The results from the *CA* data set are similar. The first set of experiments investigates 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 3(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 3(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 a SAS range 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 3(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 3(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, we use the default signature size of $\ell = 50$. Lastly, we would like to highlight that storage is cheap in modern computing environments and the construction cost is a one-time expense.

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 4. Figure 4(a) shows the average number of IOs for various query area sizes, θ, from 1% to 20%, using the *TX* data set. 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.



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

Fig. 3. Impact of the signature size: TX data set, $d = 2, N = 2 \times 10^6, \tau = 2, \theta = 3\%$.



(a) vary θ, TX data set, $d = 2, N = 2 \times 10^6, \tau = 2$. (b) vary τ, TX data set, $d = 2, N = 2 \times 10^6, \theta = 3\%$. (c) vary N, TX data set, $d = 2, \theta = 3\%, \tau = 2$. (d) vary d, UN and RC data sets, $N = 2 \times 10^6, \theta = 3\%, \tau = 2$.

Fig. 4. Query performance, $\ell = 50$.

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 4(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 (relative to the average string length).

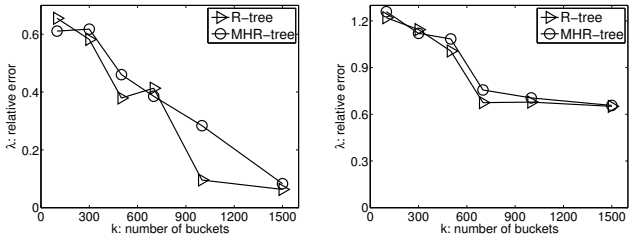
The next experiment, Figure 4(c), studies the scalability of the MHR-tree by varying the data size N. Using the *TX* data set, N ranges from 1 to 10 million. The MHR-tree scales much better w.r.t. N. The IO difference between the MHR-tree and R-tree enlarges quickly for larger data sets. For example, Figure 4(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* data set when we vary θ, τ and N.

We study the scalability of the two indexes for higher dimensions. Using the default *UN* and *RC* data sets, for $d = 3, 4, 5, 6$. Figure 4(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 data set, 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 very close 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. Given that storage is inexpensive the substantial query performance improvement of MHR-tree is significant.

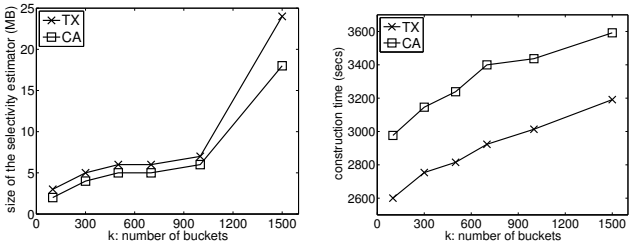
B. Selectivity Estimation for the SAS Range Queries

This section presents the experimental evaluation of our selectivity estimator for the SAS range 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 a SAS range 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.



(a) CA data set. (b) TX data set.

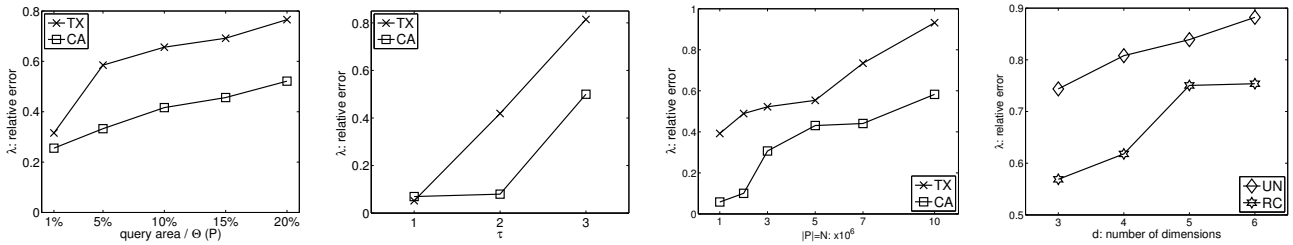
Fig. 5. Relative errors for the adaptive estimator for SAS range queries, $d = 2, N = 2 \times 10^6, \ell = 50, \tau = 2$, vary k.



(a) size. (b) construction time.

Fig. 6. Size and construction cost of the adaptive estimator, CA and TX data sets, $d = 2, N = 2 \times 10^6, \ell = 50, \tau = 2$, vary k.

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 5 when we vary the number of buckets from 100 to 1,500, on both the *CA* and *TX* data sets. Note that our adaptive algorithm works



(a) vary θ, CA and TX data sets, $d = 2, N = 2 \times 10^6, \tau = 2$.

(b) vary τ, CA and TX data sets, $d = 2, N = 2 \times 10^6, \theta = 3\%$.

(c) vary N, CA and TX data sets, $d = 2, \theta = 3\%, \tau = 2$.

(d) vary d, UN and RC data sets, $N = 2 \times 10^6, \theta = 3\%, \tau = 2$.

Fig. 7. Relative errors of the adaptive estimator, $k = 1,000$, vary θ, τ, N and d.

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* data set, λ improves from above 0.5 to below 0.1 when k changes from 100 to $1,500$ (Figure 5(a)). On the *TX* data set, λ improves from above 1.2 to 0.6 for the same setting (Figure 5(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 data sets when a small number of buckets is used, say $k = 1,000$ on 2 million points. Specifically, when $k = 1,000$, on the R-tree index, the adaptive estimator has approximately only 0.1 relative error on the *CA* data set (Figure 5(a)) and approximately 0.6 relative error on the *TX* data set. The higher λ value from the *TX* data set is mainly due to the larger errors produced by *VSol* on strings in the *TX* data set. 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 data set used, as has been shown in [32]. 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 only. But we emphasize that the results from the MHR-tree are very similar. We would also like to highlight that on each set of experiments a different set of 100 random queries is generated. Combined with the approximate nature of the min-wise signatures the cross-points of the same parameters in different figures do not match exactly.

The higher accuracy delivered by using a larger number of buckets comes at the expense of space overhead and higher construction costs. Figure 6 investigates these issues in detail. Not surprisingly, the size of the selectivity estimator increases with more buckets, as shown in Figure 6(a). Since each bucket b has to maintain the min-wise signatures of all distinct q-grams of the strings contained in it 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 $k = 1,500$ in Figure 6(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 data sets for 2 million points. When $k = 1,000$, the size of the adaptive estimator is about 6 MB for both data sets. On the other hand, the construction cost of the adaptive estimator is almost linear to the number of buckets k as shown in Figure 6(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 data sets. We omitted these comparisons for brevity.

We further investigate the accuracy of the adaptive estimator when varying other parameters, such as θ, τ, N and d, using CA, TX, UN and RC data sets. Given the results from Figures 5 and 6, we set the number of buckets to $1,000$. The results are reported in Figure 7. 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* data set compared to the *TX* data set. The estimation, with a fixed number of buckets, is more accurate for smaller query ranges (Figure 7(a)), smaller edit distance thresholds (Figure 7(b)), smaller data sets (Figure 7(c)) and lower dimensions (Figure 7(d)).

C. Other Results

We also tested the performance of MHR-tree on the SAS kNN queries, compared to the R-tree. The results are similar to the performance comparison for range queries in Section VII-A, i.e., the MHR-tree significantly outperforms the R-tree. Another set of experiments was to test the impact of the q-gram length. In general, small q-grams tend to work better in practice. From our experiments we found that $q = 2$ or 3 achieves similar performance, and $q = 2$ is slightly better. Hence, our default value for the q-gram length is 2.

VIII. Related Work

The IR^2-tree was proposed in [17] where the focus is to perform exact keyword search with kNN queries in spatial databases. The IR^2-tree cannot support approximate string searches, neither range queries and their selectivity estimation was addressed therein. Another relevant study appears in [14]

where ranking queries that combine both the spatial and text relevance to the query object was investigated.

Approximate string search has been extensively studied in the literature [5], [8], [10], [12], [18], [28], [31], [34]–[37]. 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 [18], [35], [36]. Improvements to the q-grams based pruning has also been proposed, such as v-grams [37], where instead of having a fixed length for all grams variable length grams were introduced, or the two-level q-gram inverted index [27].

Another well-explored topic is the selectivity estimation of approximate string queries [11], [23], [25], [29], [30], [32]. Most of these works use the edit distance metric and q-grams to estimate selectivity. In particular, our selectivity estimation builds on the *VSol* estimator proposed in [32]. Other work uses clustering [26]. Finally, special treatment was provided for selectivity of approximate string queries with small edit distance [29] and substring selectivity estimation was examined in [23], [30].

Our effort in dealing with selectivity estimation for the SAS range query is also related to the problem of selectivity estimation for spatial range queries [3], [19]. 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.

IX. Conclusion

This paper presents a comprehensive study for spatial approximate string queries. Using edit distance as the similarity measurement, we design the MHR-tree that embeds the min-wise signatures for the q-grams of the subtrees into the index nodes of the R-tree. The MHR-tree supports both range and NN queries effectively. We also address the problem of query selectivity estimation for SAS range queries. Interesting future work includes examining spatial approximate sub-string queries, and designing methods that are more update-friendly.

X. Acknowledgment

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

References

[1] "Open street map," http://www.openstreetmap.org.
[2] "The spatialindex library," www.research.att.com/∼marioh/spatialindex.
[3] S. Acharya, V. Poosala, and S. Ramaswamy, "Selectivity estimation in spatial databases," in *SIGMOD*, 1999.
[4] A. Arasu, V. Ganti, and R. Kaushik, "Efficient exact set-similarity joins," in *VLDB*, 2006.
[5] A. Arasu, S. Chaudhuri, K. Ganjam, and R. Kaushik, "Incorporating string transformations in record matching," in *SIGMOD*, 2008.
[6] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," in *SIGMOD*, 1990.
[7] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations (extended abstract)," in *STOC*, 1998.
[8] K. Chakrabarti, S. Chaudhuri, V. Ganti, and D. Xin, "An efficient filter for approximate membership checking," in *SIGMOD*, 2008.
[9] S. Chaudhuri and R. Kayshik, "Extending autocompletion to tolerate errors," in *SIGMOD*, 2009.
[10] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani, "Robust and efficient fuzzy match for online data cleaning," in *SIGMOD*, 2003.
[11] S. Chaudhuri, V. Ganti, and L. Gravano, "Selectivity estimation for string predicates: Overcoming the underestimation problem," in *ICDE*, 2004.
[12] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *ICDE*, 2006.
[13] E. Cohen, "Size-estimation framework with applications to transitive closure and reachability," *Journal of Computer and System Sciences*, vol. 55, no. 3, pp. 441–453, 1997.
[14] G. Cong, C. S. Jensen, and D. Wu, "Efficient retrieval of the top-k most relevant spatial web objects," *PVLDB*, vol. 2, no. 1, pp. 337–348, 2009.
[15] E. D. Demaine, D. Emanuel, A. Fiat, and N. Immorlica, "Correlation clustering in general weighted graphs," *Theoretical Computer Science*, vol. 361, no. 2, pp. 172–187, 2006.
[16] C. Ding and X. He, "Cluster merging and splitting in hierarchical clustering algorithms," in *ICDM*, 2002.
[17] I. D. Felipe, V. Hristidis, and N. Rishe, "Keyword search on spatial databases," in *ICDE*, 2008.
[18] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava, "Approximate string joins in a database (almost) for free," in *VLDB*, 2001.
[19] D. Gunopulos, G. Kollios, J. Tsotras, and C. Domeniconi, "Selectivity estimators for multidimensional range queries over real attributes," *The VLDB Journal*, vol. 14, no. 2, pp. 137–154, 2005.
[20] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *SIGMOD*, 1984.
[21] G. R. Hjaltason and H. Samet, "Distance browsing in spatial databases," *ACM Trans. Database Syst.*, vol. 24, no. 2, 1999.
[22] V. Hristidis and Y. Papakonstantinou, "Discover: keyword search in relational databases," in *VLDB*, 2002.
[23] H. V. Jagadish, R. T. Ng, and D. Srivastava, "Substring selectivity estimation," in *PODS*, 1999.
[24] S. Ji, G. Li, C. Li, and J. Feng, "Efficient interactive fuzzy keyword search," in *WWW*, 2009.
[25] L. Jin and C. Li, "Selectivity estimation for fuzzy string predicates in large data sets," in *VLDB*, 2005.
[26] L. Jin, C. Li, and R. Vernica, "Sepia: estimating selectivities of approximate string predicates in large databases," *The VLDB Journal*, vol. 17, no. 5, pp. 1213–1229, 2008.
[27] 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*, 2005.
[28] N. Koudas, A. Marathe, and D. Srivastava, "Flexible string matching against large databases in practice," in *VLDB*, 2004.
[29] H. Lee, R. T. Ng, and K. Shim, "Extending q-grams to estimate selectivity of string matching with low edit distance," in *VLDB*, 2007.
[30] H. Lee, R. T. Ng, and K. Shim, "Approximate substring selectivity estimation," in *EDBT*, 2009.
[31] C. Li, J. Lu, and Y. Lu, "Efficient merging and filtering algorithms for approximate string searches," in *ICDE*, 2008.
[32] A. Mazeika, M. H. Böhlen, N. Koudas, and D. Srivastava, "Estimating the selectivity of approximate string queries," *ACM TODS*, vol. 32, no. 2, pp. 12–52, 2007.
[33] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," in *SIGMOD*, 1995.
[34] S. Sahinalp, M. Tasan, J. Macker, and Z. Ozsoyoglu, "Distance based indexing for string proximity search," in *ICDE*, 2003.
[35] E. Sutinen and J. Tarhio, "On using q-gram locations in approximate string matching," in *ESA*, 1995.
[36] E. Ukkonen, "Approximate string-matching with q-grams and maximal matches," *Theor. Comput. Sci.*, vol. 92, no. 1, pp. 191–211, 1992.
[37] X. Yang, B. Wang, and C. Li, "Cost-based variable-length-gram selection for string collections to support approximate queries efficiently," in *SIGMOD*, 2008.