Indexing Mixed Types for Approximate Retrieval

Liang Jin
University of California, Irvine, USA
Chen Li
University of California, Irvine, USA
Nick Koudas
University of Toronto, Canada
Anthony K. H. Tung
National University of Singapore, Singapore

Abstract

In various applications such as data cleansing, being able to retrieve categorical or numerical attributes based on notions of approximate match (e.g., edit distance, numerical distance) is of profound importance. Commonly, approximate match predicates are specified on combinations of attributes in conjunction. Existing database techniques for approximate retrieval, however, limit their applicability to single attribute retrieval through B-trees and their variants. In this paper, we propose a methodology that utilizes known multidimensional indexing structures for the problem of approximate multi-attribute retrieval. Our method enables indexing of a collection of string and/or numeric attributes to facilitate approximate retrieval using edit distance as an approximate match predicate for strings and numeric distance for numeric attributes. The approach presented is based on representing sets of strings at higher levels of the index structure as tries suitably compressed in a way that reasoning about edit distance between a query string and a compressed trie at index nodes is still feasible. We propose and evaluate various techniques to generate the compressed trie representation and fully specify our indexing methodology. Our experimental results show the benefits of our proposal when compared with various alternate strategies for the same problem.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

1 Introduction

Data cleansing, namely the process of removing errors and inconsistencies from data in databases, has traditionally been an area of high research interest due to its vast practical significance. A multitude of problems (e.g., typing/insertion mistakes, lack of standardized ways for recording data fields such as addresses) can degrade data quality and possibly impact common business practices. Traditionally, data cleansing methodologies have been used on data in an off-line fashion. An algorithm is executed on large data sets or directly on relational tables aiming to identify and report all candidate inconsistencies [1, 9, 10, 17, 18]. Inconsistencies are quantified in terms of proximity expressed via similarity measures defined on pairs of attribute values. Such measures include edit distance [11], cosine similarity [9] or variants thereof [5, 21]. As an example, given a large relation of customer names, techniques in the spirit of [1, 9, 10] will identify all pairs of customers that according to some particular similarity measure (e.g., edit distance) applied on each pair of names, are within a pre-specified threshold value (say within an edit distance of 3). Algorithms operating in this fashion could be highly time consuming, as they commonly involve joins between large data collections, possibly complemented by additional post processing of the results.

A variety of applications with data cleansing requirements, however, list strong demand for interactive response times. Consider, for example an operator during a service call. The customer supplies personal information such as name, year of birth, address etc, in order for the operator to retrieve the relevant profile from the database. Such examples are prevalent in common business practices. It is desirable to support searches while being able to cope with data-quality problems existing either in the underlying database or the query itself. In such cases, for example, it is desirable to retrieve all relevant records from the database that are "close" (with respect to some similarity measure) to the query values, in case an exact match is not identified. This requirement is imperative in order to aid operators in this example to correctly

decide the customer identity. It is also desirable to return such close matches in an online fashion providing interactive response times. As another example, consider a data-cleansing scenario with a clean, reference table [7]. Given a new input record, we want to quickly find records in the reference table that potentially match this record. In these applications, there is a clear requirement to relax query semantics and return fuzzy (approximate) matches quickly.

In this paper, we propose a methodology to efficiently support such fuzzy queries in an interactive fashion, when the specified attributes contain various data types including numerical and character strings and combinations thereof. Our methodology enables adoption of various similarity functions, one per attribute type in the query. Queries specify values and bounds on multiple attributes (e.g., name, address, age), effectively enabling conjunctive queries constraining the similarity function on each specified attribute. We refer to such queries as *fuzzy range queries*. Consider a relation R with attributes *name* and *YOB*. Assume we adopt edit distance, *ed*, to quantify proximity of string values in the name attribute and absolute numerical distance for proximity in the YOB attribute. A query q specifying values Q_{name} and Q_{YOB} will return all tuples t from R such that

$$ed(Q_{name}, t_{name}) \leq \delta_{name} \ \& \ |Q_{YOB} - t_{YOB}| \leq \delta_{YOB}$$

for constants δ_{name} and δ_{YOB} supplied at query time.

We realize our methodology for answering fuzzy range queries by proposing indices that utilize prevalent concepts from common indexing techniques (such as R-trees). Designing practical indices for such problems faces a multitude of challenges. Firstly, when the query attributes have mixed types, different similarity functions (one per type) have to be supported by the index (e.g., edit distance for strings, absolute numeric difference for numbers). Although multi-attribute indices have been studied for the case of exact string retrieval [14], we are not aware of work on multi-attribute indices supporting mixed types for fuzzy retrieval. Secondly, to assure practical deployment of such indices, they should be easy to realize drawing as many concepts as possible from common indexing practise.

We propose the *Mixed-Attribute-Type Tree* ("MAT-tree"), an R-tree-based indexing structure that aids fuzzy retrieval for range queries specifying combinations of numbers and strings on input. Edit distance has been previously utilized in various contexts, including data cleansing to quantify similarity of strings. We adopt it to quantify similarity between the query and database strings. We use the absolute numerical distance to quantify similarity between query values and database numeric values. We fully specify and describe our index restricting ourselves to a single (string, number) query pair to ease presentation. Following a very similar approach it is possible to generalize our framework for larger collections of strings and numbers. In this work, we make the following contributions:

(1) We propose and fully specify the MAT-tree as a solution to the problem of fuzzy range query answering, for queries involving mixed types. Our work is the first to address this important problem.

(2) Although supporting fuzzy numerical retrieval in the index follows from applications of known concepts from B-trees, handling fuzzy retrieval for strings in conjunction presents many challenges. In order to support the edit distance as a similarity predicate for strings in the index, we show it is possible to represent information about a collection of strings on internal index nodes using a fixed amount of space, while still being able to reason about edit distance in the string collection. We propose and evaluate various algorithms for reducing (compressing) the string volume to a fixed amount of space. We also derive an algorithm to reason about edit distance between a query string and the compressed string collection.

(3) In addition to fully specifying the operations in our index (including dynamic maintenance under insertions and deletions), we conduct a thorough experimental evaluation of our index comparing it with a multitude of alternate applicable approaches based on M-trees or q-gram structures. Our results indicate that our methodology offers large performance advantages for this problem as parameters of interest vary. The results validate that indexing strings in a tree structure can support effective pruning during a top-down traversal of the tree using the string condition, in addition to the pruning by the numerical attribute. Such pruning effectively improves the search performance.

We emphasize the following about the MAT-tree indexing structure. First, our technique can be adopted to solve the problem of indexing strings as a tree to support approximate predicates, which by itself is of independent research interest. Such a tree can be easily integrated with another indexing structure on numeric data. Second, the intuition behind using a tree on multiple attributes is that, doing a search using such an integrated tree tends to be more efficient than using two separate trees and intersecting their answers, similarly to the observation that the pruning power of an R-tree is generally better than two B-trees together. By keeping "ranges" of both the numeric attribute and the string attribute in an internal node in the tree, we can support more effective pruning during a tree traversal. This advantage is verified by our experiments.

This paper is organized as follows. Section 2 formulates the studied problem. Section 3 summarizes approaches that naturally extend existing structures. Section 4 gives an overview of our proposed MAT-tree structure, and discusses how to represent strings in an index entry. Section 5 presents how to construct such a tree structure, how to compress strings to fit into an indexing entry, and how to maintain the structure

incrementally. Section 6 reports our experimental results. We conclude the work in Section 7.

1.1 Related Work

A lot of work on data cleansing has focused on the off-line case, in which algorithms are applied to base tables to identify inconsistencies, commonly identify pairs of tuples which are approximate duplicates. Gravano et al. [9, 10] discuss how to realize popular similarity functions between strings in a declarative framework. Anathakrishana et al. [1] discuss a related similarity function for the same problem. Hernandez and Stolfo [13] examine the same problem by weighting and combining multiple attributes. Garhaldas et al. [8] examine declarative specifications of cleaning operations. Sarawagi et al. [23] present a learning framework for identifying approximate duplicates, and Raman et al. [20] develop an interactive system for data cleansing. A survey of some data cleaning system prototypes is available in [22].

An indexing structure supporting fuzzy retrieval using a similarity function called *fms* (a variant of edit distance between entire tuples) is proposed [7]. Such an approach is probabilistic (returns exact tuples with high probability), and it may miss relevant results. Moreover, the entire approach suggested in [7] is not capable of taking individual type information as well as constraints associated with individual types into account. Jagadish et al. [14] proposed a family of indexing structures supporting prefix and exact match queries on multi-attribute strings. Such structures cannot support fuzzy retrieval using notions of approximate string match. Jin et al. [16] study how to estimate selectivity of fuzzy string predicates.

A large body of work in combinatorial pattern matching deals with problems of approximate retrieval of strings [2, 11]. Leading data structures utilized for this purpose are suffix trees [11] and suffix arrays [2]. Such structures however, are highly specialized and commonly non-balanced, and they are static in their majority, i.e., they do not provide efficient support for incremental changes. Moreover, the bulk of these works assume the index fits in memory. Extensions to secondary storage exist, but are primarily of theoretical interest.

2 Problem Formulation

Consider a relation whose schema includes a string attribute A_S, a numeric attribute A_N, and possibly other attributes. Table 1 shows such a relation, which stores information about movie stars and directors. "Name" is a string attribute (A_S) and "YOB" ("Year of Birth") is a numeric attribute (A_N). The relation also has other information such as the movies they starred in or directed. Although our framework is general enough to encompass a variety of attributes with mixed (string, numeric) types, we realize things concrete by confining our discussion on two attributes. In the full version of this paper [15] we discuss how to extend our technique to more general distance functions and queries with conditions on more than two attributes.

Name	YOB	Movies	...
Hanks	1956	Forrest Gump	...
Robert	1968	Erin Brockovich	...
Roberrts	1977	Notting Hill	...
Crowe	1964	Gladiator	...
...	...	...	...

Table 1: A movie relation.

We consider *mixed-type fuzzy queries* with conditions on both attributes. Each query consists of a tuple $(Q_S, \delta_S, Q_N, \delta_N)$, where Q_S is a string for attribute A_S, δ_S is a threshold for this attribute, Q_N is a value for attribute A_N, and δ_N is a corresponding threshold. The query is requesting all records r in the relation, such that:

$$ed(r.A_S, Q_S) \leq \delta_S \ \& \ |r.A_N - Q_N| \leq \delta_N,$$

where "*ed*" stands for "edit distance." Formally, given two strings s_1 and s_2, their edit distance, denoted $ed(s_1, s_2)$, is the *minimum* number of edit operations (insertions, deletions, and substitutions) of single characters that are needed to transform s_1 to s_2. For instance, ed("Robert", "Roberrts") = 2. Consider a user query ("Roberts", 2, 1967, 10) on the movie relation. The answers are ("Robert", 1968) and ("Roberrts", 1977). Tuple ("Robert", 1968) is an answer because ed("Roberts", "Robert") = 1 and $|1967 - 1968| = 1$. Tuple ("Roberrts", 1977) is also an answer since ed("Roberts", "Roberrts") = 1 and $|1977 - 1967| = 10$.

3 Existing Approaches

We present existing indexing structures to support mixed-type fuzzy queries, with possible natural extensions. We first focus on how to build an indexing structure for the string attribute, discuss how to combine such an indexing structure with the structure on the numeric attribute to answer mixed-type queries.

M-Trees: The M-tree [4] is an indexing structure for supporting queries on objects in a metric space. Since edit distance forms a metric space, we can utilize an M-tree to index strings to support fuzzy queries on strings. We build an M-tree using string attribute A_S. Each routing object (string) p in the tree has a radius r that is an upper bound for the edit distance between this string and any string in its subtree. (In case routing strings are too large to fit in a node, they can be suitably truncated to desired space. The corresponding radius has to be adjusted in response to such a truncation.) Given a threshold δ_S and a query string Q_S, we

access the subtree of a routing object (string) p with a covering radius r in a node only if $ed(p, Q_S) \leq r + \delta_S$.

Q-Trees: Another approach to building an index tree for the string collection is to use their *q-grams*. Given a string s and an integer q, the set of *q-grams* of s, denoted $G_q(s)$, is obtained by sliding a window of length q from left to right over the characters of string s. If the number of remaining characters in the window is smaller than q, we use a special symbol not in the alphabet, e.g., "#", to fill in the empty slots. For instance, the set of 3-grams of the string "Roberts" is: $G_3(\texttt{Roberts}) = \{\texttt{Rob}, \texttt{obe}, \texttt{ber}, \texttt{ert}, \texttt{rts}, \texttt{ts\#}, \texttt{s\#\#}\}$.

Consider a query string s, a record string t, and a threshold k. We split s into $k+1$ pieces (with similar sizes), and compute the q-gram set for string t. Navarro et al. [3] show that, if $ed(s,t) \leq k$, then at least one of the $k+1$ pieces "matches" with one of the q-grams. We say two strings *match* if they are exactly the same after the longer one being truncated to the shorter length. The special character we introduced ("#") does not match with any character in the alphabet. For example, suppose $s =$ `Roberrts`, $t =$ `Roberts`, and $k = 1$. We split s into 2 pieces (`Robe` and `rrts`) and use the 3-gram set above for t. Because $ed(s,t) \leq 1$, there must be a match between a piece and a q-gram. In particular, the piece `Robe` and the q-gram `Rob` match since they are exactly the same if we truncate them to the shorter length. Therefore, if none of the $k+1$ pieces can find a match in the q-grams, there are at least $k+1$ edit distance errors between s and t, and the condition $ed(s,t) \leq k$ cannot be true.

If there are many strings in the dataset, we may not want to store all their q-grams. Instead, we just store the q-gram range for the strings. For the above string t, we can just store a lexicographical range (`ber`, `ts#`) that covers all the q-grams of t. (When comparing two strings, if they have different lengths, we first truncate the longer one to the shorter length, then do the comparison. The special symbol "#" is lexicographically larger than any other character.) If none of the pieces falls into the range, we can be sure that $ed(s,t) \leq k$ cannot be true.

Based on this observation, we can build a tree, called "Q-tree," that indexes the string attribute A_S. Each node in the tree has a list of string ranges $[min_S, max_S]$. The min_S and max_S are the minimal and maximal lexicographical values of all the q-grams extracted from all the A_S strings of the records in the corresponding subtree. To find all strings within distance δ_S of a string Q_S, we chop Q_S into $\delta_S + 1$ pieces. We traverse the Q-tree top down. For each range in a node, we compare the pieces against the range. We visit the subtree of the range only if one of the pieces is within the q-gram range. For the candidate strings in the leaf nodes, we compare their strings against the query condition to compute the final answers.

Indexing Structures for Both Attributes: We can build an indexing structure to support mixed-type queries by integrating indexing structures for both attributes. Such a structure, called "BQ-tree," indexes on both attributes A_S and A_N by combining a B-tree and a Q-tree. Figure 1 shows the indexing structure for our sample dataset. Each node in the tree has a list of MBR's ("minimal bounding rectangles"), each of which stores the range information of the descendent records. An MBR is represented as: $[min_S, max_S]$, $[min_N, max_N]$. The min_N and max_N are the minimal and maximal A_N values for all the records covered by this MBR. The min_S and max_S are the minimal and maximal lexicographical values of all the q-grams extracted from all the A_S strings of the records covered by the MBR. During a search using the tree, at each entry we use both ranges to prune the branches that do not satisfy any of the two query conditions. We can build a similar indexing structure, called "BM-tree," by integrating a B-tree on the numeric attribute and an M-tree on the string attribute.

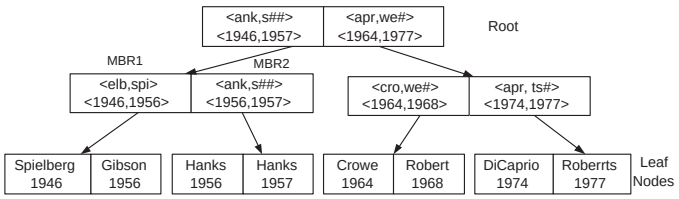

Figure 1: A BQ-tree for the movie dataset.

4 The MAT-Tree

In this section we realize our methodology for efficiently answering mixed-type fuzzy queries by developing an indexing structure called "MAT-tree." It is instantiated by adapting concepts from R-trees [12]. Each index entry in the tree (with a pointer to a child node) has a numeric range for the numeric attribute A_N, such that all the descendant records have an A_N value within this range. In addition, their string values of the A_S attribute are represented as a *trie*, which is compressed to fit into the index entry with a fixed size. Figure 2 shows an example MAT-tree for the movie relation. The details of the tree are explained shortly.

Given a mixed-type query $\{Q_S, \delta_S, Q_N, \delta_N\}$, we traverse the tree from the root to the leaf nodes. For each index entry in a node n, we check if the range $[Q_N - \delta_N, Q_N + \delta_N]$ overlaps with the numeric range. If so, we then check if the strings represented by the compressed trie could satisfy

$$minEditDist(Q_S, T_n) \leq \delta_S,$$

where T_n is the compressed trie in the entry, and we use $minEditDist(Q_S, T_n)$ to represent the minimum edit distance between query string Q_S and any string represented by the compressed trie. The child of this entry is visited only if both conditions are satisfied.

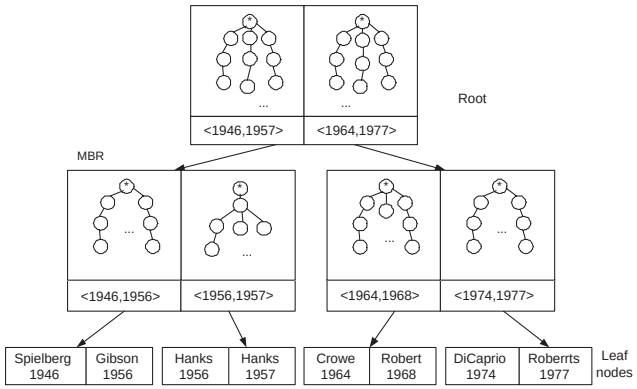

Figure 2: A MAT-tree for the movie relation.

4.1 Representing Strings as a Compressed Trie

A *trie* for a set of strings, is a tree such that every node excluding the root is labeled with a character. Each path from the root to a leaf represents a string from the set of strings; all the paths in the trie represent all the strings in the set [24]. Figure 3 shows such a trie, in which the path $n1 \rightarrow n3 \rightarrow n7 \rightarrow n13$ represents the string `beh`.

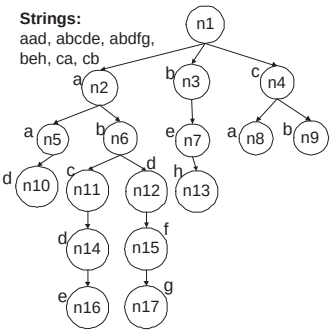

Figure 3: A trie.

When constructing a MAT-tree, we need to store a trie in a node entry. (A trie can be easily linearized and stored as a string.) Because the entry has limited space, we choose to compress the trie to satisfy the space constraint. Our compression approach is to select a set of k representative nodes $C = \{c_1, \ldots, c_k\}$ in the trie to form the "backbone" of an automaton, which accepts all the strings in the original trie (and possibly more). For simplicity of reference, we call these representative nodes *centers*.

After selecting nodes as centers, we convert the original trie T to a compressed trie T_c as follows. In T_c there is a node for each center. There is an edge in T_c from a node c_i to a node c_j if node c_i is an ancestor of node c_j in T, and there is no other center on the path from c_i to c_j in T. Figure 4 is a compressed trie for the trie in Figure 3, where nodes $n1$, $n2$, $n4$, $n5$, $n6$, $n7$, $n11$, $n13$, and $n15$ are selected as centers. Since $n1$ is an ancestor of $n2$, $n4$, and $n7$ in T, and there is no other center between $n1$ and the other three nodes, we have an edge from $n1$ to each of the three nodes in the compressed trie.

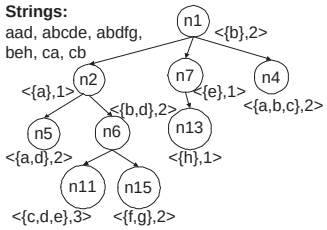

Figure 4: A compressed trie.

We next look at how these centers are used to represent other nodes in the original trie T. Denoting the set of nodes in the original trie as $N = \{n_1, \ldots, n_m\}$, we use $C' = N - C$ to refer to the set of non-center nodes. We say a non-center node $n_i \in C'$ is "covered" by a center c_j if c_j is an ancestor of n_i in T, and the path from c_j down to n_i does not contain any other center. Going back to the earlier example, nodes $n8$ and $n9$ are covered by center $n4$, while node $n3$ is covered by center $n1$.

It is easy to see that each non-center node will be covered by only one center. For the rest of this paper, we will refer to the set of nodes covered by a center c_j as its *covered-node set*, denoted $cns(c_j)$. Each center c_j represents the nodes it covers as a tuple $\langle \Sigma_j, L_j \rangle$. Here Σ_j refers to the set of characters that are represented by c_j or any node in $cns(c_j)$. Symbol L_j represents the longest-path length between c_j and any nodes in $cns(c_j)$. In Figure 4, since $n4$ covers nodes $n8$ and $n9$, we have $\Sigma_4 = \{a, b, c\}$ and $L_4 = 2$, where "2" is the length of the path from $n4$ to $n8$. Obviously, the amount of information loss in adopting such a compressed trie highly depends on the selection of the k centers. In Section 5.1, we look at various methods for selecting k centers to minimize the information loss.

4.2 Minimum Edit Distance Between a String and a Compressed Trie

Given a query string q and a compressed trie T_c, we want to compute the minimum edit distance between q and all the strings represented by T_c, denoted as $minEditDist(q, T_c)$. The main idea of computing the distance is to convert the compressed trie to an automaton, and then compute the minimum edit distance between the string and all the strings acceptable by the automaton. We adapt an algorithm for computing such a distance [19], and show how a compressed trie can be converted to an automaton.

4.2.1 Distance Between a String and an Automaton

Myers and Miller [19] proposed an algorithm for computing the minimum edit distance between a string s and all the strings accepted by an automaton M. We use a simple example to briefly illustrate our adaptation of this algorithm. (See [19] for details.) Given an automaton M and a string $s = s_1 \dots s_n$, we construct an edge-labeled directed *edit graph*, which is designed so that paths between two vertices correspond to alignments between string s and strings generated by M. For example, Figure 5 shows a string $s = \texttt{ac}$, an automaton, and the corresponding edit graph.

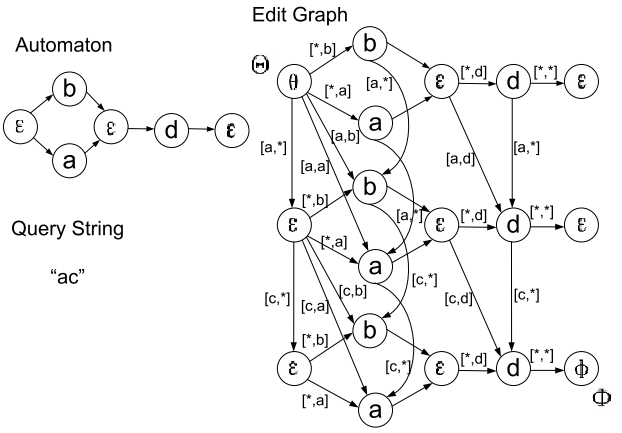

Figure 5: String, automaton, and the edit graph.

In general, the automaton edit graph consists of $n+1$ copies of M in $n+1$ rows. For each character s_i in the string ($i \in [0, n]$) and each state j in the automaton, there is a vertex in the edit graph represented as a pair (i, j). Some nodes are labeled with the empty symbol ϵ to connect two nodes. A special vertex Θ represents the starting node and another special vertex Φ represents the final ending node. We add three types of edges to the edit graph corresponding to the three types of edit operations.

1. If $i \in [1, n]$ and either j is the starting state in M or $j \neq \epsilon$, there is a *deletion* edge: $(i-1, j) \rightarrow (i, j)$ labeled $[s_i, \epsilon]$. For instance, edge $(0, b) \rightarrow (1, b)$ with label $[a, \epsilon]$ in Figure 5 is a deletion edge.

2. If $i \in [0, n]$ and there is a transition edge $j \rightarrow k$ in M, then there is an *insertion* edge $(i, j) \rightarrow (i, k)$ labeled $[\epsilon, k]$. The edge $(0, \theta) \rightarrow (0, b)$ with label $[\epsilon, b]$ is an insertion edge.

3. If $i \in [1, n]$, there is a transition edge $j \rightarrow k$ in M, and $k \neq \epsilon$, then there is a *substitution* edge $(i-1, j) \rightarrow (i, k)$ labeled $[s_i, k]$. The edge $(1, \theta) \rightarrow (2, b)$ with label $[c, b]$ is a substitution edge.

After constructing the edit graph, we can assign a weight to each edge, which is the cost of the corresponding edit operation. That is, the weight of an edge $[x, y]$ is 1 if $x \neq y$; and 0 otherwise. It can be shown that the minimum edit distance between s and M is the length of a shortest path from the starting state Θ to the ending state Φ, which can be computed efficiently using a shortest-path algorithm [6].

4.2.2 Converting a Trie to an Automaton

Given an uncompressed trie, we can convert it to an automaton as follows: each node of the trie becomes a state, and each edge becomes a transition edge between the two corresponding states in the automaton. Given a compressed trie, we consider each compressed node $\langle \Sigma, L \rangle$ and expand it to a substructure of L levels, which creates the enumeration of strings with every possible length from 1 up to L. Each level contains all the single nodes of characters in Σ, and every node is connected to a common tailing ϵ. This ϵ state has a direct link to the final "tail" ϵ. We call this kind of link a "bypass link." For instance, the substructure of a compressed node $[\{\texttt{a}, \texttt{b}, \texttt{c}\}, 2]$ is shown in Figure 6. We convert the final expanded trie to an automaton. In [15] we discuss how to do early termination during the computation of the minimum edit distance.

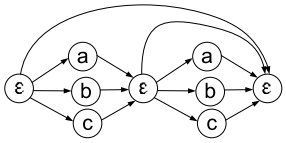

Figure 6: Expanding a compressed node $[\{\texttt{a}, \texttt{b}, \texttt{c}\}, 2]$ to automaton nodes.

5 Construction and Maintenance

There are two ways to construct a MAT-tree. The first way is to perform repetitive insertions into an empty structure. Handling insertions is described in Section 5.2 where we deal with dynamic maintenance issues. Here, we describe how the index can be constructed in a bottom-up fashion.

Initially we assume that records are materialized on disk with each tuple consisting of a string-and-numeric value pairs (and possibly other values). The bottom-up construction follows common bulk-loading practise. The records are sorted according to an attribute, either string in alphanumeric order or the numeric, depending on which attribute one wishes to maintain proximity (this is an analog of the *low-x* bulk-loading algorithm for R-trees [12]). As pages are filled with tuples, all the strings belonging to a page are grouped into a trie, which is subsequently pruned down to fixed space, via algorithms detailed in the next subsection. In addition, the minimum and maximum values of the range corresponding to the numeric values associated with all the strings in the page are computed and stored along with the resulting trie (after compression) as an index node entry for that page. After the first (leaf)

level of the index is constructed, the remaining levels of the index are constructed in a recursive fashion. Compressed tries at level i are merged at their roots and subsequently pruned (Section 5.1) down to a fixed size to form tries at level $i+1$ until the root is formed.

5.1 Effective Trie Compression

One critical requirement is to accurately represent a set of strings using a compressed trie that can fit into an indexing entry in the tree. The amount of information loss in compression greatly affects the pruning power during a traversal of the tree to answer a query. We now discuss three different methods for compressing the trie with each method trying to minimize a different measurement of information loss.

Method 1: Reducing Number of Accepted Strings. One way to measure the accuracy of a compressed trie is to use the number of its strings represented. Intuitively, we want to minimize this number since the more additional strings the compressed trie (suitably transformed into an automaton) can accept, the more inaccurate the compression is, compared to the original one. As a result the trie will provide more inaccurate edit distance estimates against a query string, and the pruning power during the search will be weakened. We use a goodness function referred to as "StrNum()" to compute such a number. Formally, let T be a trie and $C = \{c_1, \ldots, c_k\}$ be a set of k centers. Each center is represented as $\langle \Sigma_i, L_i \rangle$. Consider the corresponding compressed trie T_c with these k centers. The function "StrNum" computes the total number of strings accepted by T_c. $StrNum(T, C) = f(c_1)$ where c_1 is the center at the root of T_c, and $f(c_i)$ is a recursive function defined as follows:

- If all descendants of c_i are not in C, then $f(c_i) = |\Sigma_i|^{L_i}$.
- Otherwise, let $c_{m_1}, \ldots, c_{m_p}$ be the set of the closest centers that are descendants of c_i. Then $f(c_i) = |\Sigma_i|^{L_i} * \sum_{j=1}^{m_p} f(c_j)$.

Take the compressed trie in Figure 4 as an example. Given the nine centers placed at the indicated nodes, each path from the root to a leaf node represents a finite set of strings. For example, the path from the root $n1$ to $n5$ will result in at most $1^2{*}1^1{*}2^2{=}4$ strings. By summing up the number of strings that could potentially be represented by each path, we will be able to compute the function $StrNum(T, C)$.

Given a trie T and a number k, we want to select k nodes to be centers, so that the corresponding compressed trie yields a small value for *StrNum*. Figure 7 shows a randomization algorithm for solving the problem. The main idea is the following. We always choose the root as a center. Initially we randomly pick $k-1$ nodes as centers. Then we attempt to replace some centers in the current selection so that the goodness function is improved (i.e., the function value is decreased). We randomly select a center c_i from C, and then randomly pick another non-center node n_j as a candidate to replace c_i. A new selection of centers C_{new} is formed by removing c_i from C and adding n_j into C. If the *StrNum* value of C_{new} is lower than that of the original C, then C_{new} is considered to be a better solution, and it will be used to replace C. The algorithm terminates after the attempt to find a better solution is unsuccessful for M times.

Algorithm: **Randomization**
Input: T, original trie; k, number of centers;
M: number of iterations;
Output: a set of k centers according to Method 1.
Method:
select root of T as the center c_1;
randomly pick $k-1$ nodes as remaining centers;
C = {these k centers}; i=0;
while ($i \leq M$) {
randomly select a center c_i in C ($i \neq 1$);
randomly select a non-center node n_j;
let $C_{new} = C - \{c_i\} + \{n_j\}$;
If $StrNum(T, C_{new}) < StrNum(T, C)$ then
{ $C = C_{new}$; i=0; }
else i++;
}
Output C;

Figure 7: Randomization for selecting centers.

Method 2: Keeping Accepted Strings Clustered. Another way to measure the goodness of a compressed trie is to look at the distribution of its accepted strings, and check how similar they are to the original ones. Ideally we want to keep these strings as similar to the original ones as possible. To maximize this possibility, given k centers, our strategy is to place these centers as close to the root as possible, so that the strings represented can share common prefixes. That is, given k as the number of centers, we can do a breadth-first traversal of the trie, and choose the first k as centers. Figure 8 shows how the earlier trie in Figure 3 will be compressed using this algorithm with k=9 centers being placed at $n1, n2, \ldots, n9$.

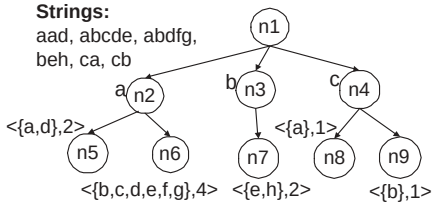

Figure 8: Compressed trie (BreadthFirst).

Method 3: Combining Methods 1 and 2. We can combine the two previous methods by considering both the number of accepted strings and their similarity to the original strings. For instance, when adopting method 2, we could have chosen a node as a center, while it is the starting node of a long path. Choos-

ing this node to compress could introduce a lot of new strings. (Recall that the number of strings accepted by a compressed node $\langle \Sigma, L \rangle$ is $|\Sigma|^L$.)

In combining the two methods, we still try to keep the prefixes of the original strings by placing the centers close to the root of the trie. In addition, we also consider the number of strings accepted by each compressed node in making the selection. We present a simple, yet intuitive compression technique called "BottomUp." The intuition is that for the edit distance calculation, especially with the early termination ideas, the nodes close to the root are relatively more important. Therefore, we want to keep them as precise as possible. Figure 9 shows the algorithm, in which "$cost(x)$" for a node x is the number of additional accepted strings if node x is compressed. Figure 10 shows how the algorithm selects centers for the trie in Figure 3. The result has $k = 9$ centers located at $n1$, $n2$, $n3$, $n4$, $n5$, $n6$, $n7$, $n11$, and $n12$.

Algorithm: **BottomUp**
Input: T, original trie; k, number of centers;
Output: a set of k centers according to Method 3.
Method:
set C = empty; priority query Q = empty;
for the parent node p of each leaf node
  insert pair $(p, cost(p))$ into queue Q;
while (T has more than k nodes) {
  pick the entry (n, x) from Q with the smallest x;
  remove the entries for n's descendants from Q;
  remove from T the subtree rooted at n;
  add n to C;
  remove n's descendant centers from C;
  p = parent node of n;
  insert pair $(p, cost(p))$ into queue Q;
}
Output C;

Figure 9: BottomUp for selecting centers.

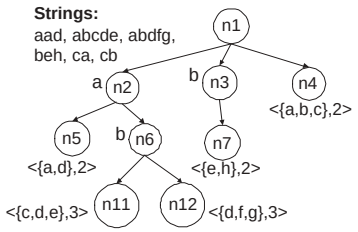

Figure 10: Compressed trie (BottomUp).

5.2 Dynamic Maintenance

The MAT-tree is a fully dynamic index. As such it provides support for common index-management operations such as insertion, deletion, and update of tuples.

Insert record (s, n): where (s, n) is a string-number pair. A search operation is first issued in the index. If the pair is not present, its position in a page p is identified. If the insertion does not cause page overflow, the compressed trie T_c corresponding to this page needs to be updated. This can be accomplished by going through each character of s in conjunction to T_c and walking down the trie, skipping characters of s as indicated by the branches of T_c. If a node in T_c is reached which does not match the corresponding character of s, then the character is added into that node in T_c. Since center nodes are of $O(\Sigma)$ in length, there is always space to handle this insertion. For improved space efficiency, center nodes are represented as bit vectors (a bit for each character in Σ), thus inserting a character is a simple bitwise operation. This process is repeated until the end of s or the end of the corresponding branch in T_c whichever occurs first. If T_c is modified, then a similar operation has to be issued recursively to s and the parent of T_c, until the root of the index or until a compressed trie is reached in which there is no modification. If a page split occurs, two new pages are formed containing approximately equal number of string-number pairs. A compressed trie is constructed for each of them using the method above, and the corresponding numeric ranges for each trie are computed. The resulting entries are inserted into the parent node of the split page. This process is repeated recursively upwards as long as page overflow occurs. Splitting a page into two containing approximately equal number of compressed trie-numeric-range pairs can be performed using algorithms similar to those used in R-trees (e.g., quadratic split [12]). However, the objective function needs to be adjusted to the particular domain. Any function on $StrNum$ of the compressed trie and the corresponding numeric range can be used as an objective for optimization while splitting a page into two.

Delete record (s, n): A delete operation is handled in a similar way. First the (s, n) pair is located in a page and is removed. If the page does not underflow, then the operation halts. Otherwise, the page is merged with a neighboring leaf page if possible, in a way similar to R-trees. After the merge, a trie is constructed for all entries of this newly formed page, and is pruned to the fixed space. Subsequently the compressed tries in the leaf page are merged at their roots and pruned to the fixed space, recursively up until the root in the worst case. Merging two tries is easy. Depending on the structure of the compressed trie, there are several way to do the merging. An example is the following: in the bit vector representation, we just perform a logical OR operation between the center representation. If a page underflow occurs, this process is also repeated in the index pages as well.

Update record (s, n): It can be handled by locating the pair in the index and performing the modification in that page. The process proceed similarly to the bottom-up index adjustment in the case of an index insert operation.

6 Experiments

In this section, we present the results of our intensive experiments to evaluate the proposed approaches. We used two real datasets. Data set 1 consists of 100,000 movie star records extracted from the Internet Movie Database.[1] Each record has a name and year of birth (YOB) of a movie star. The length of each name varies from 5 to 20, and the average length is around 12. We generated the YOB values using a normal-distribution data generator. The YOB values vary from 1900 to 2000 with a mean of 1950. Data set 2 is a customer relation database with 50,000 records. Each record also has a name and a YOB value. For each dataset, we randomly chose records, and introduced random errors to the records to form a query set. We ran 100 queries for each experiment and measured the average performance. All our experiments were run on a PC, with a 2.4GHz P4 CPU and 1.2GB memory, running Windows XP. The compiler was Visual C++. In addition, we used 8,192 bytes as the page size of the tree structures. Since the experimental results for both data sets had similar trends, due to space limitation we mainly report the results for data set 1. More results are in [15].

6.1 Advantages of Indexing Both Attributes

We evaluated the performance gains by building an indexing structure on both attributes, compared to two separate indexing structures. We implemented the following approaches. (1) "B-tree": We built a B-tree for the numeric attribute A_N (YOB), and postprocessed the string condition for those records that passed the B-tree check. (2) "Q-tree": We built a Q-tree for the the string attribute A_S (name), and postprocessed the numeric condition for those records that passed the Q-tree check. (3) "B-tree & Q-tree": We built a B-tree on A_N and a Q-tree on A_S, and intersected the records that passed the check for either structure. (4) "BQ-tree": We built a BQ-tree on both attributes.

B-tree	Q-tree	B-tree & Q-tree	BQ-tree
152	102	212	81

Table 2: Times of different approaches (seconds).

Table 2 shows the running times of these approaches for a dataset with 80,000 records, when $\delta_N = 4$ and $\delta_S = 3$. The BQ-tree performed the best, validating the intuition that integrating two attributes into one indexing structure performs more powerful pruning in the tree traversal to answer a query. This result is in agreement with the intuition that an R-tree is better than one B-tree or two B-trees with intersection. We varied the values of δ_N and δ_S, and observed similar results. We also had similar results for other indexing structures such as M-trees. In general, the savings depend on the correlation between the two attributes.

In the rest of this section, we mainly focus on the results of indexing structures on both attributes. We evaluated the following approaches: (1) Sequential scan ("SEQ"); (2) BM-trees ("BM-Tree"); (3) BQ-trees ("BQ-Tree"); (4) MAT-trees with Randomization compression ("MAT-RANDOMIZED"); (5) MAT-trees with BreadthFirst trie compression ("MAT-BF"); and (6) MAT-trees with BottomUp trie compression ("MAT-BOTTOMUP"). For sequential scan, to reduce the running time, we first checked if an entry in a node satisfied the numerical query condition. If so, we further checked the condition on the string attribute. In the evaluation we measured both the running time and the number of IO's for each approach.

6.2 Scalability

We evaluated the scalability of these approaches using subsets with different numbers of records from data set 1. We set the numeric range threshold $\delta_N = 4$ and edit-distance threshold $\delta_S = 3$. In addition, we set the number of centers in a trie $k = 400$. We report the running time in Figure 11 and the number of disk IO's in Figure 12.

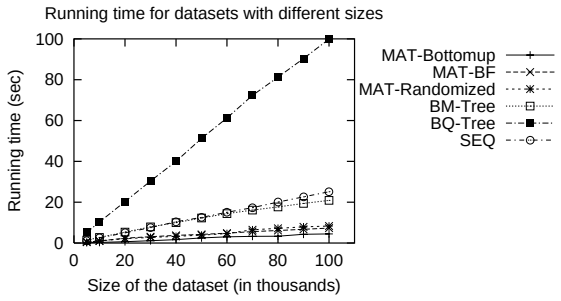

Figure 11: Time for datasets with different sizes.

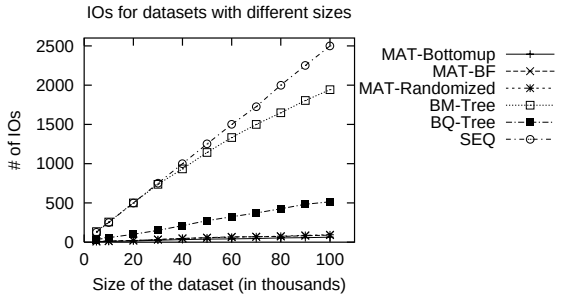

Figure 12: IO's for datasets with different sizes.

The figures show that MAT-tree took the minimum running time and the least number of disk IO's. Among the three trie-compression approaches of MAT-trees, BottomUp was the best for both measurements, while Randomization and BreadthFirst were close to each

[1] http://www.imdb.com

other. BreadthFirst compression was slightly better than Randomization in running time due to more efficient optimization for the edit-distance calculation. For example, when the database had 80,000 records, the BottomUp, BreadthFirst, and Randomization approaches took 3.3, 7.2, and 7.2 seconds, respectively, while their numbers of disk accesses were 54, 73, and 76, respectively.

For the other approaches, although BM-tree incurred many disk IO's, it took less time than Sequential Scan and BQ-tree. The reason is that at each node of the BM-tree, we only need to calculate the edit distance between the query string and the routing string, and the pruning can save computation of some edit distances.

6.3 Changing Thresholds

We next varied the numeric and string thresholds in queries to study their impact on the performance of different approaches. We ran the experiments on 80,000 records, and the number of centers in the MAT-tree was 400. We first fixed δ_S as 3 and let δ_N vary from 1 to 8. The time for the approaches is shown in Figures 13. Similarly, to study the effect of the string threshold, we fixed δ_N to 4, and varied δ_S from 1 to 6. The time and IO count are depicted in Figures 14 and 15, respectively.

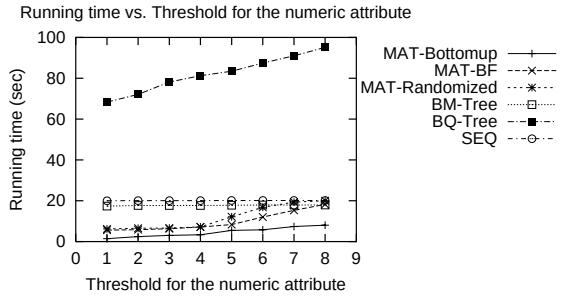

Figure 13: Effect of numeric threshold: time.

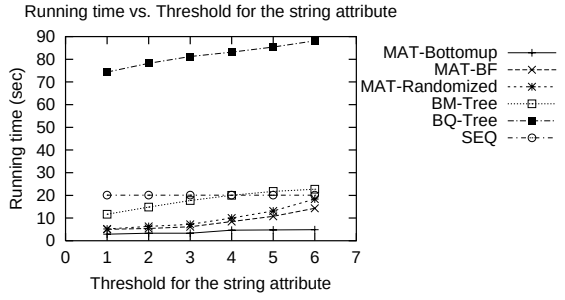

Figure 14: Effect of string threshold: time.

These figures show that, with the increase of the numeric threshold and/or the string threshold, both the running time and the IO number increased for all the

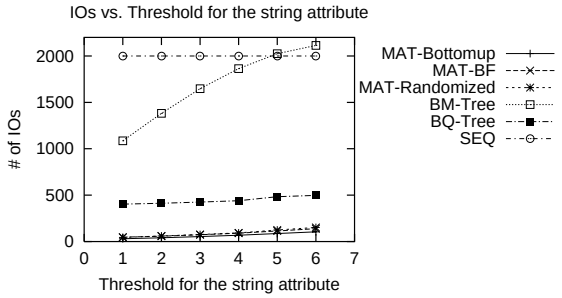

Figure 15: Effect of string threshold: IO's.

approaches. However, different approaches showed different trends. The performance of the MAT-tree with the BottomUp compression was very stable for different numeric thresholds and string thresholds, e.g., it took less than 4 seconds to find the answers when δ_N was 4 and δ_S was 4. The performance of the BM-tree approach deteriorated significantly as the string threshold increased. Still, our proposed MAT-tree had the best performance and efficiency among these approaches. The main reason is that, in each node in the MAT-tree, the compressed trie can effectively represent the information about the strings in the subtree of the node. We can use both attributes to do effective pruning during the traversal of the tree.

6.4 Number of Centers in MAT-trees

As discussed in Section 5.1, the number of centers in each index entry of a MAT-tree can greatly affect the accuracy of the compressed trie and thus the pruning power during the tree traversal. This number determines how much information we can retain from the original trie. We ran experiments on the subset of 80,000 records. The numeric threshold δ_N was 4 and the string threshold δ_S was 3. We varied the number of centers from 50 to 500.

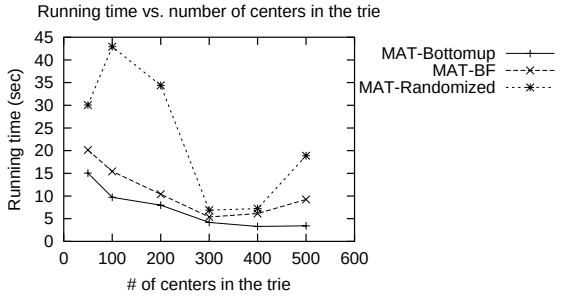

Figure 16: Time versus number of centers.

Figure 17 shows that the number of IO's decreased as the number of centers increased. This result is consistent with our analysis. When we increase the number of centers, the compressed trie is more accurate, and is more likely to produce a good estimation of the minimum edit distance to a query string, thus giv-

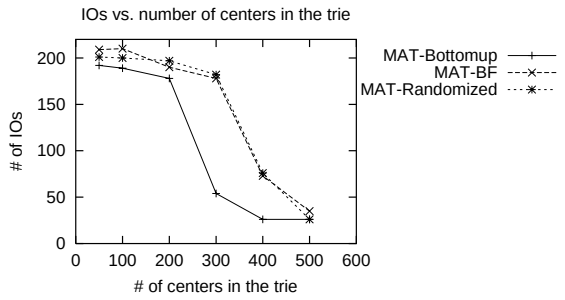

Figure 17: IO's versus number of centers.

ing better pruning power which reduces the number of IO's. For example, a MAT-tree with the BottomUp compression required 192 IO's with 50 centers, while only 26 IO's with 400 centers. Similarly, the numbers of IO's for the Randomization and BreadthFirst approaches dropped from 209 and 201 at 50 centers to 73 and 76 with 400 centers.

However, Figure 16 shows that reducing the number of IO's does not necessarily reduce the running time. The reason is that, as the number of centers grows, the computational complexity for comparing the query against the automaton increases. For the BottomUp and BreadthFirst approaches, this effect is offset by the fact that the centers are placed near the root, which provide more opportunities for making use of an early termination strategy [15]. Thus we only saw a slight running-time increase for the BreadthFirst approach when the number of centers was 500. The analysis cannot be used for the Randomization approach, which does not have centers concentrated at the root. The final trie gives fewer opportunities for early termination. The increase in the IO pruning due to the larger number of centers is not able to compensate for the increase in the computational complexity. We see this taking effect in two instances in Figure 16 when the number of centers is increased from 50 to 100 and from 300 through to 500.

6.5 Dynamic Maintenance

We conducted experiments on dynamic maintenance for a MAT-tree, as described in Section 5.2. We evaluated the efficiency of each change operation for a record (insertion, deletion, and modification), and the effect on the indexing structure after such operations.

For these experiments we used data set 1 that consists of 100,000 records. We created a collection C of N records that were randomly selected from the data set, and we let N vary from 60,000 to 80,000. We created a MAT-tree for the collection of the records using the Randomization approach, and the number of centers was 300. We generated a workload of 1,000 operations, including insertion of a new record, and deletion or update of an existing record in the collection C. For an insertion, we chose one of the remaining records not in the collection C to insert (into C). For a deletion, we randomly chose one record in the collection to delete. For an update, we randomly chose one record in the collection, and performed the update by adding or subtracting a random number between 1 and 20 to the numeric value. We also introduced a random number (between 1 and 5) of edit changes to the string value. We measured the running time for each operation. The results are shown in Figure 18.

The figure shows that the MAT-tree can be dynamically maintained very efficiently for each operation. For instance, when the collection had 80,000 records, the maintenance of the MAT-tree for an insertion, deletion, and update only took $28ms$, $29ms$, and $20ms$, respectively. The update maintenance was faster than other two operations because in our implementation, an update does not cause page splitting and merging, since we utilized the space in the MAT-tree for the original record to store the information for the modified record. The figure also shows the MAT-tree maintenance performance scales very well for different database sizes. The main reason is that most of the maintenance time was spent on the disk IO's during the top-down traversal and the possible bottom-up adjustment of the tree. Since the height of the tree does not increase much as the database size increases, the maintenance time does not increase much.

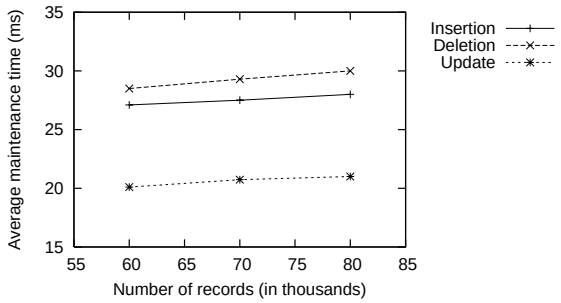

Figure 18: Dynamic-maintenance time for a MAT-tree.

Next, we conducted experiments to evaluate how the search performance of a MAT-tree degrades as we increase the number of change operations to the tree. For a collection of records, after a number of change operations on the MAT-tree, we ran 100 queries on the collection, and each query has a numeric threshold $\delta_N = 4$ and a string threshold $\delta_S = 3$. We computed the average search time using the MAT-tree.

Figure 19 shows the average search time after varying number of change operations (for varying number of records in the collection). It shows that the quality of the MAT-tree remained high as the number of operations increased. For example, for the 70,000-record collection, the original MAT-tree could answer a search query in 8.2 seconds. After 2,000 operations, the response time increased to 10.1 seconds only. In other words, the experiments show that, if there are not

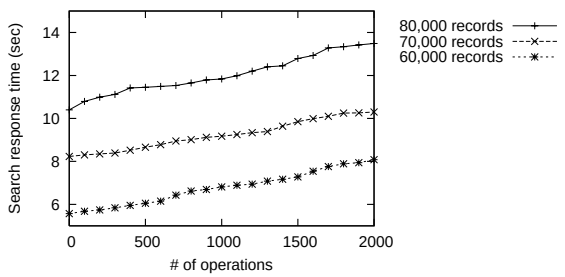

Figure 19: MAT-tree research response time after dynamic maintenance.

many changes to the data records, the MAT-tree can still achieve a high search performance. If the number of operations on the records becomes too large and the search performance of the MAT-tree becomes too low, it becomes necessary to rebuild the MAT-tree to ensure its high quality.

Sizes of Indexing Structures: We collected the sizes of indexing structures for a dataset with $80,000$ records. The sizes (in KB) of B-tree, Q-tree, M-tree, BM-tree, BQ-tree, and MAT-tree are 1799, 1854, 529, 531, 2124, and 2034, respectively. The results show that the size of a MAT-tree has the same scale as other structures. Specifically, the MAT-tree provided up to 14 seconds of saving in access time, while imposing about 1.6MB space overhead. This overhead is worthwhile as disks are getting cheaper while their capacities are getting larger.

7 Conclusions

We proposed a methodology that allows approximate retrieval on combinations of string and/or numeric attributes using edit distance as an approximate match predicate for strings and numeric distance for numeric attributes. We instantiated our methodology using two attributes. Generalizing it to more than two attributes with mixed (string, numeric) types is also possible. The approach is based on representing sets of strings at higher levels of the index structure as tries suitably compressed in a way that reasoning about edit distance between a query string and a compressed trie at index nodes is still feasible. We proposed various techniques to generate the compressed trie and fully specified the indexing methodology. Our intensive experiments showed the benefits of our proposal when compared with various alternate strategies.

References

[1] R. Anathakrishna, S. Chaudhuri, and V. Ganti. Eliminating Fuzzy Duplicates in Data Warehouses. *VLDB*, Aug. 2002.

[2] A. Apostolico and Z. Galil. *Pattern Matching Algorithms*. Oxford University Press, 1992.

[3] R. Baeza-Yates and G. Navarro. A practical index for text retrieval allowing errors, 1997.

[4] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *The VLDB Journal*, pages 426–435, 1997.

[5] W. Cohen, P. Ravikumar, and S. Fienberg. A Comparison of String Metrics for Matching Names and Records. *Data Cleaning Workshop in Conjunction with KDD*, Aug. 2003.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Presss, 1990.

[7] V. Ganti, S. Chaudhuri, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. *SIGMOD*, June 2003.

[8] H. Garhaldas, D. Florescu, D. Shasha, E. Simon, and E. Saita. Declarative Data Cleaning. *Proceedings of VLDB*, 2001.

[9] L. Gravani, P. Ipeirotis, N. Koudas, and D. Srivastava. Approximate Text Joins and Their Integration into an RDBMS. *Proceedings of WWW*, 2002.

[10] 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.

[11] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1998.

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

[13] M. Hernandez and S. Stolfo. The Merge Purge Problem for Large Databases. *SIGMOD*, pages 127–138, June 1995.

[14] H. V. Jagadish, N. Koudas, and D. Srivastava. On Effective Multidimensional Indexing for Strings. *SIGMOD*, June 2001.

[15] L. Jin, N. Koudas, C. Li, and A. K. Tung. Indexing mixed types for approximate retrieval (full version). Technical report, UC Irvine, Department of Computer Science, 2005.

[16] L. Jin and C. Li. Selectivity estimation for fuzzy string predicates in large data sets. In *Proc. of VLDB*, 2005.

[17] L. Jin, C. Li, and S. Mehrotra. Efficient record linkage in large data sets. In *Eighth International Conference on Database Systems for Advanced Applications*, 2003.

[18] M.-L. Lee, T. W. Ling, and W. L. Low. Intelliclean: a knowledge-based intelligent data cleaner. In *Knowledge Discovery and Data Mining*, pages 290–294, 2000.

[19] E. W. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, pages 7–37, 1989.

[20] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *The VLDB Journal*, pages 381–390, 2001.

[21] S. C. Sahinalp, M. Tasan, J. Macker, and Z. M. Ozsoyoglu. Distance based indexing for string proximity search. In *International Conference on Data Engineering*, 2003.

[22] S. Sarawagi. Special Issue on Data Cleaning. *IEEE Data Engineering Bulletin, 23 (4)*, 2000.

[23] S. Sarawagi and A. Bhamidipaty. Interactive Deduplication Using Active Learning. *Proceedings of VLDB*, 2002.

[24] H. Shang and T. H. Merrett. Tries for Approximate String Matching. *IEEE Trans. on Knowledge and Data Engineering*, pages 540–7, 1996.