Large Scale Hamming Distance Query Processing

Alex X. Liu, Ke Shen, Eric Torng

Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, U.S.A.

{alexliu, shenke, torng}@cse.msu.edu

Abstract—**Hamming distance has been widely used in many application domains, such as near-duplicate detection and pattern recognition. We study Hamming distance range query problems, where the goal is to find all strings in a database that are within a Hamming distance bound k from a query string. If k is fixed, we have a static Hamming distance range query problem. If k is part of the input, we have a dynamic Hamming distance range query problem. For the static problem, the prior art uses lots of memory due to its aggressive replication of the database. For the dynamic range query problem, as far as we know, there is no space and time efficient solution for arbitrary databases. In this paper, we first propose a static Hamming distance range query algorithm called HEngine^s, which addresses the space issue in prior art by dynamically expanding the query on the fly. We then propose a dynamic Hamming distance range query algorithm called HEngine^d, which addresses the limitation in prior art using a divide-and-conquer strategy. We implemented our algorithms and conducted side-by-side comparisons on large real-world and synthetic datasets. In our experiments, HEngine^s uses 4.65 times less space and processes queries 16% faster than the prior art, and HEngine^d processes queries 46 times faster than linear scan while using only 1.7 times more space.**

I. INTRODUCTION

A. Background

Given two equal length strings α and β, the Hamming distance between them, denoted $HD(\alpha, \beta)$, is the number of positions where the two strings differ. For example, $HD(100, 111) = 2$. Several classic similarity search problems are defined for Hamming distance. In particular, there is the Hamming distance range query problem, which was first posed by Minsky and Papert in 1969 [22], where the goal is to find all strings in a database that are within a Hamming distance bound k from a query string. A related problem is the K nearest neighbor (KNN) problem, where the goal is to find the K nearest neighbors in a given database with respect to Hamming distance to a given query string. Although these problems are extremely simple and natural problems, finding fast and space efficient algorithms for solving them remains a fundamental open problem where new results are still emerging.

One reason these problems are widely studied is that they have meaning in a wide variety of domains such as near-duplicate detection, pattern recognition, virus detection, DNA sequence and protein analysis, error correction, and document classification. Hamming distance range query problems are especially useful in domains that deal with discrete data. For example, Manku *et al.* proposed a solution to the near-duplicate detection problem in Web crawling [19] that includes

a new solution to the Hamming distance range query problem. The basic idea is to hash each document into a 64-bit binary string, called a fingerprint or a sketch, using the simhash algorithm in [11]. Simhash has the following property: if two documents are near-duplicates in terms of content, then the Hamming distance between their fingerprints will be small. Similarly, Miller *et al.* proposed a scheme for finding a song that includes a given audio clip [20], [21]. They proposed converting the audio clip to a binary string (and doing the same for all the songs in the given database) and then finding the nearest neighbors to the given binary string in the database to generate candidate songs. Similar applications have been used for image retrieval and iris recognition [15], [26].

B. Problem Statement

In this paper, we focus on two variants of the Hamming distance range query problem. In a Hamming distance range query problem, we have a database T of binary strings, where the length of each string is m, and a query (α, k) where α is a binary string of length m and distance bound k is an integer that is at most m. The goal is to return the set of strings $HD(T, \alpha, k) = \{\beta \mid \beta \in T \text{ and } HD(\alpha, \beta) \leqslant k\}$. Let $n = |T|$, the number of strings in T. We assume that T is fixed which means that we will process many distinct queries (α, k) with the given database T. We assume a solution preprocesses the database T in order to minimize the query processing time; that is, the time required to identify $HD(T, \alpha, k)$ for any query (α, k). We evaluate a solution to this problem using four criteria: its query processing time, the space it needs to store the preprocessed T, the time it needs to preprocess T, and the time it needs to update T when small changes to T are made.

Our two variants of the problem differ in only one aspect. In the first variant, which we call the *static Hamming distance range query problem*, the Hamming distance bound k is fixed; that is, all queries will use the same value k and thus we can optimize the preprocessing of T for k. In the second variant, which we call the *dynamic Hamming distance range query problem*, the distance bound k is not fixed; that is, different queries may use different values of k and thus we cannot optimize the preprocessing of T for a particular k. Any solution to the dynamic bound problem is also a solution to the static bound problem, but the reverse is not true. To simplify notation in the rest of this paper, we use *static problem* and *dynamic problem* as shorthand for the two variants of the Hamming distance range query problem.

C. Limitations of Prior Art

For the static problem, most prior work focuses on the special case of $k = 1$. For arbitrary but small k, Manku's algorithm in [19] represents the state-of-the-art. However, it uses an extensive amount of memory due to its aggressive replication of the database. In many applications, solutions for small k are all that is needed. For example, in the near-duplicate Web page detection system in [19], solutions for $k = 3$ are sufficient. For arbitrary but large k, two solutions have been proposed in [12], [13]; however, they are inefficient for problems with small k.

For the dynamic problem, as far as we know, there is no space and time efficient solution for arbitrary databases. Existing solutions that use a trie [4], [5] are sensitive to the string length. However, in practice, long strings (*e.g.* 64-bits) are common. Other solutions assume that the strings in the database can be clustered and are therefore not suitable to databases with arbitrary strings [23].

D. Our Approach

We focus on both the static and dynamic problems with arbitrary but small k red(*i.e.*, $k \leqslant 10$). For the static problem, we address the space issue in prior art by dynamically expanding the query on the fly. This "query expansion" greatly reduces the number of database replicas needed. We use Bloom filters to quickly identify and eliminate most unsuccessful queries. The end result is that we not only save space but also achieve faster query processing than prior art. We use HEngine^s (s standing for "static") to denote our algorithm for the static problem.

For the dynamic problem, we address the string length issue in prior art using a divide-and-conquer strategy. For a given Hamming distance range query instance, if the distance bound k exceeds a given threshold, we divide the query string in half, all the strings in T in half, and k in half and then recursively solve the resulting smaller Hamming distance query instances. Thus, our algorithm's query processing time grows with the logarithm of string length. We use HEngine^d (d standing for "dynamic") to denote our dynamic Hamming distance range query algorithm.

Most modern microprocessors such as current 64-bit x86 processors have a popcount(s) instruction that returns the number of ones in a bit string s. Thus, the hamming distance between two bit strings α and β of length no more than 64 can be calculated by only two instructions: $s = \alpha \oplus \beta$ and $HD(\alpha, \beta) =$popcount(s). Thus, our query processing time roughly grows as a function of $m/64$. For moderate m, we can assume query processing time is independent of m.

E. Summary of Experimental Results

We implemented our HEngine algorithms in Java 1.6. To perform a side-by-side comparison, we also implemented Manku's algorithm in Java 1.6. We evaluated them using a large real-world dataset as our database T and randomly generated queries. In our experiments, HEngine^s uses 4.65 times less memory and is 16% faster in query processing, 4.15

times faster in preprocessing, and 4.53 times faster in handling data updates. We also perform an 'unfair' comparison of HEngine^d with Manku's algorithm; the comparison is unfair because Manku's algorithm cannot handle dynamic Hamming distance range queries. In our experiments, HEngine^d uses 6.12 times less memory and is 50.7% slower in query processing, 13.35 times slower in preprocessing, and 2.28 times faster in handling data updates. More preprocessing and query processing time is the price that HEngine^d pays for its flexibility in handling varying k. In our experiments, HEngine^d processes queries 46.42 times faster than linear scan using only 1.7 times more space.

F. Key Contributions

We make three key contributions. First, we propose an algorithm for the static problem that uses less memory but processes queries more quickly than prior art. redSecond, we propose an efficient algorithm for the dynamic problem that works with long strings (*e.g.* 64-bits). Last, we implemented our algorithms and conducted extensive experiments on both real and synthetic datasets.

The rest of this paper proceeds as follows. We first discuss related work in Section II. Then, we present HEngine^s and HEngine^d in Sections III and IV, respectively. We report our experimental results in Section V. We conclude the paper in Section VI.

II. RELATED WORK

In this section, we first discuss prior work on the static and dynamic problems. Then, we review prior work that converts similarity search problems in other distance metrics to similarity search problems in Hamming space.

A. Static Hamming Distance Range Query

We classify prior work on the static problem into three categories based on bound k: (a) $k = 1$, (b) k is small but $k > 1$, (c) k is large.

$k = 1$: Yao and Yao proposed an algorithm with $O(m \log \log n)$ query time and $O(nm \log m)$ space [27]. Yao's algorithm first cuts the query string and each string in the database in half, and then it finds exact matches in the database for the left and right halves of the query string; for each match, Yao's algorithm recursively applies this cutting strategy until the query string is just 1 bit. This recursive cutting is the fundamental strategy used in several subsequent algorithms for the static problem including ours, although Yao and Yao only focused on the $k = 1$ case. In [9], Brodal and Gąsieniec proposed a trie-based solution with $O(m)$ query time and $O(nm)$ space. Brodal and Venkatesh proposed an algorithm with $O(1)$ query time (assuming that m fits within one memory word) and $O(mn \log m)$ space that finds a string in the database that differs by at most 1 from the query string using perfect hash functions [10]. Unfortunately, we cannot leverage this algorithm in our work because we need to find all strings in the database that differ by at most 1 from the query string.

k **is small but** $k > 1$: This problem was mainly investigated in Manku *et al.*'s seminal work [19]. Manku *et al.* proposed an algorithm for small k (up to 7) using a cluster of machines. They generalized Yao's algorithm by cutting database strings and the query string into $k + 1$ segments ($k \geqslant 2$). This method is based on the observation that if two string are within Hamming distance k, then at least one of the $k + 1$ segments are exact matches for the two strings. This algorithm achieves high throughput for large databases. Its main drawback is high memory consumption due to extensive replication of the database. This algorithm needs to replicate the database $k+1$ times and for each copy, sort the strings based on one of the $k + 1$ segments. This limits the scalability of the approach for large databases or large k. For this reason, Manku *et al.* developed a distributed implementation where tables are stored on a farm of machines.

Our algorithm generalizes Yao's and Manku's algorithm in the following way. Their algorithms reduce to exact matching algorithms in the base case. Our algorithm reduces to an approximate matching algorithm in the base case (*i.e.*, two segments are within a certain Hamming distance z). This generalization gives us more flexibility to trade query time for storage space. As z increases, our algorithm uses less space but requires more query time. This adaptability allows our algorithm to be used for applications with varying space and query time requirements. In our experiments, our algorithm HEngine^s for the static problem with $z = 1$ uses 4.65 times less memory than Manku's algorithm while being 16% faster.

k **is large**: Two solutions have been proposed in [12], [13]; however, both are inefficient for problems where k is small.

B. Dynamic Hamming Distance Range Query

In the theory community, Arslan and Eğecioğlu proposed a simple solution using tries [5]. Its query processing complexity is $O(m^{k+2})$ and its space complexity is given by the space occupation of the trie plus additional space to store $O(m^{k+1})$ trie nodes. Later, Arslan proposed another trie-based algorithm with query processing complexity $O(m(\log_{4/3} n - 1)^k (\log_2 n)^{k+1})$ [4]. No experimental results were given in [5] and [4]. Because the trie processes the query string bit-by-bit, the query processing time of the trie approach is sensitive to string length m. In contrast, the query processing time of our dynamic Hamming distance range query algorithm HEngine^d grows with the logarithm of string length assuming that all strings fit within a memory word.

In the database community, Qian *et al.* proposed a solution using a dynamic indexing technique called ND-tree [23]. They cluster the database and build the tree from the clusters. A Hamming distance range query is processed from the root node pruning away any nodes that are out of the query bound until we find the leaf nodes containing the desired data. This algorithm's performance depends on whether the database strings can be well clustered. In contrast, our algorithm for the dynamic problem HEngine^d works reasonably well even on synthetic randomly generated data sets.

C. Set Similarity Join Problem

Similarity join is a common problem in near-duplicate detection and document clustering [3], [7], [16], [25]. It is a generalization of the well-known nearest neighbor problem. Two sets are joined to form all pairs of objects whose similarity score is above a threshold. When the two sets are identical, it is a self-join problem. The similarity join problem differs from Hamming distance range queries in that the focus in similarity joins is to find all the similar pairs from two given sets. In Hamming distance range queries, the focus is to preprocess the database that is repeatedly queried. Thus, for Hamming distance range queries, preprocessing time is relatively unimportant whereas query time is important. For similarity join, there is no separation of preprocessing time and query time. In [3], Arasu *et al.* proposed a similarity join algorithm called PartEnum, which is in essence a non-recursive version of Manku's algorithm. In our experiments, our static problem algorithm HEngine^s outperforms PartEnum.

III. STATIC HAMMING DISTANCE RANGE QUERIES

In this section, we present our static problem algorithm HEngine^s. Because k is fixed, we refer to the query as just the string α. We first present the theoretical foundation of HEngine^s. Second, we describe how HEngine^s preprocesses the database and processes queries. Third, we discuss a recursive implementation of HEngine^s as well as how to efficiently update the database when small changes to T are made. Then, we describe how we improve query processing time using Bloom filters. Further, we give the space and time analysis of HEngine^s. We show that theoretically HEngine^s requires significantly less memory space and less preprocessing time than the prior art.

A. Overview of HEngine^s Algorithm

There are three straightforward solutions for solving the static problem: linear scan, query expansion, and table expansion. Linear scan uses little space but query processing is slow. Table expansion uses a lot of space but query processing is fast. Query expansion uses the same amount of space as linear scan and can be much faster than linear scan depending on the values of k, m, and n, but it is much slower than table expansion. With linear scan, each string in T is compared to the query string α to find all strings that are within Hamming distance k of α. This requires a total of n string comparisons. Query expansion computes all strings that are within Hamming distance k of the query string α (including α itself). For any binary string α of length m, there are a total of $\sum_{b=0}^{k} \binom{m}{b}$ strings that are within Hamming distance k of α, denoted $\alpha_1, \cdots, \alpha_{\sum_{b=0}^{k} \binom{m}{b}}$. Query expansion generates all of these strings and does an exact match search within T for each of these strings. Since we are doing an exact match search, we can use binary search, assuming we sort T first. This requires a total of $\sum_{b=0}^{k} \binom{m}{b} \log n$ string comparisons, which can be much fewer than n. Table expansion requires a preprocessing step. For each binary string β in T, one first precomputes all the strings whose Hamming distance from β is less than or

equal to k (including β itself). For any binary string β of length m, there are a total of $\sum_{b=0}^{k} \binom{m}{b}$ strings that are within Hamming distance k, denoted $\delta_1, \cdots, \delta_{\sum_{b=0}^{k} \binom{m}{b}}$. Preprocessing concludes by storing the pairs $(\delta_1, \beta), \cdots, (\delta_{\sum_{b=0}^{k} \binom{m}{b}}, \beta)$ in a new database denoted as T^k in sorted order. Query processing involves finding all pairs of strings in T^k whose first string is α and returning the second string of each such pair. Because T^k is sorted, these strings will appear in consecutive locations and the entire range can be efficiently found in $O(\log n(\sum_{b=0}^{k} \binom{m}{b}))$ string comparisons using binary search.

HEngine^s balances storage space and processing time by carefully expanding the database so that a Hamming distance query can be processed by an efficient combination of selected query expansion and linear scans. The theoretical foundation of HEngine^s is captured in the following results. To formally state these results, we define the following notation.

Definition 3.1: For a length m string β, we define r-$cut(\beta)$ to be the process of splitting β into r substrings $(\beta_1, \ldots, \beta_r)$ where the first $r - (m \mod r)$ substrings have length $\lfloor m/r \rfloor$ and the last $m \mod r$ substrings have length $\lceil m/r \rceil$. We overload notation and use r-$cut(\beta)$ to also denote $(\beta_1, \ldots, \beta_r)$.

Theorem 3.1: For any two binary strings β and γ such that $HD(\beta, \gamma) \leqslant k$, consider r-$cut(\beta)$ and r-$cut(\gamma)$ where $r \geqslant \lfloor k/2 \rfloor + 1$. It must be the case that $HD(\beta_i, \gamma_i) \leqslant 1$ for at least $q = r - \lfloor k/2 \rfloor$ different values of i.

    Proof: We prove the theorem by showing that $HD(\beta_i, \gamma_i) \geqslant 2$ for at most $\lfloor k/2 \rfloor$ different values of i. Observe that $\lfloor k/2 \rfloor + 1 \geqslant k/2 + 1/2$; equality holds when k is odd. Suppose $HD(\beta_i, \gamma_i) \geqslant 2$ for $\lfloor k/2 \rfloor + 1$ values of i. This would imply $HD(\beta, \gamma) \geqslant 2(\frac{k}{2} + \frac{1}{2}) \geqslant k + 1 > k$ contradicting the fact that $HD(\beta, \gamma) \leqslant k$. Thus the result follows. ∎

Theorem 3.1 implies that all β with $HD(\alpha, \beta) \leqslant k$ must satisfy a signature match property that is formally expressed in Theorem 3.2.

Definition 3.2: For any length m string β and $r \geqslant \lfloor k/2 \rfloor + 1$, a signature of β, denoted β-$signature$, is any choice of $q = r - \lfloor k/2 \rfloor$ substrings from r-$cut(\beta)$. We define the choice vector of β-$signature$ as $(i_1, i_2, \ldots, i_q)$ where $1 \leqslant i_1 < i_2 \cdots < i_q \leqslant r$ represents the q substrings chosen from r-$cut(\beta)$.
Any string β has $\binom{r}{q}$ signatures.

Definition 3.3: For any two length m strings α and β and $r \geqslant \lfloor k/2 \rfloor + 1$, we say that two signatures α-$signature$ and β-$signature$ are compatible if their choice vectors are identical. We say that compatible signatures α-$signature$ and β-$signature$ match if $HD(\alpha_{i_j}, \beta_{i_j}) \leqslant 1$ for $1 \leqslant j \leqslant q$. That is, each of the q substrings from α-$signature$ is within Hamming distance 1 of the corresponding substring from β-$signature$.

Theorem 3.2: Consider any string $\beta \in T$ such that $HD(\alpha, \beta) \leqslant k$. Given any $r \geqslant \lfloor k/2 \rfloor + 1$, it follows that at least one signature β-$signature$ matches its compatible signature α-$signature$.

The basic idea of HEngine^s is to preprocess T to facilitate

finding all strings β in T that satisfy the signature match property of Theorem 3.2 and then verify which of these strings actually are within Hamming distance k of α.

B. HEngine^s Preprocessing

We preprocess a database T given Hamming distance bound k and segmentation factor $r \geqslant \lfloor k/2 \rfloor + 1$ by constructing $\binom{r}{q}$ duplicates of T, one for each signature. We call this set of tables the *signature set* of T for distance bound k and segmentation factor r; we call each table in the signature set a signature table. We begin by computing r-$cut(\beta)$ for all $\beta \in T$. In each signature table, each string β is permuted so that the q chosen substrings are moved to the front of the string. In each permuted string, the original order among the q chosen substrings is maintained. Likewise, the original order among the $r - q$ other substrings is also maintained. Finally, we sort the permuted strings in each table to enable efficient binary searches. The pseudocode of the above signature set construction procedure (the preprocessing phase of HEngine^s) is shown in Algorithm 1. We use $B(i)$ to denote the i-th element in set B and $a|b$ to denote the concatenation of substrings a and b.

Algorithm 1: HEngine^s preprocessing algorithm

Input: (1) Database T in which the length of each string is m.
      (2) Hamming distance bound k.
      (3) Segmentation factor r.
Output: The signature set of T based on distance bound k and segmentation factor r.

1 Function build (T, k, r)
2 begin
3     $\mathbb{RS} := \emptyset$;
4   for $i := 1$ to $\binom{r}{q}$ do
5         $T_i := \emptyset$;
6   foreach $\beta \in T$ do
7         $\mathbb{PS} = $ permute (β, k, r);
8   for $i := 1$ to $\binom{r}{q}$ do
9   insert $\mathbb{PS}(i)$ into T_i;
10   for $i := 1$ to $\binom{r}{r - \lfloor \frac{k}{2} \rfloor}$ do
11   insert T_i into $\mathbb{RS}$;
12   return $\mathbb{RS}$;

13 Function permute (β, k, r)
14 begin
15     $\mathbb{PS} := \emptyset$;
16   Compute r-$cut(\beta)$.
17   for $i := 1$ to $\binom{r}{q}$ do
18         $\mathbb{IS} := $ nextChoice (r, q);
19   front := ""; back := "";
20   foreach $x \in \mathbb{IS}$ do
21   front := front$|\beta_x$;
22   foreach $y \in [1, r] - \mathbb{IS}$ do
23   back := back$|B(y)$;
24   insert front$|$back into $\mathbb{PS}$;
25   return $\mathbb{PS}$;

We illustrate the HEngine^s preprocessing in Fig. 1. Here we have $m = 64$, $k = 4$, $r = 3$ and $q = 1$. The lengths of the three substrings will be 21, 21, and 22. We construct the $\binom{r}{q} = 3$ tables by moving $q = 1$ different substrings to the left in each table. These 3 tables constitute the signature set of T based on distance bound 4 and segmentation factor 3.
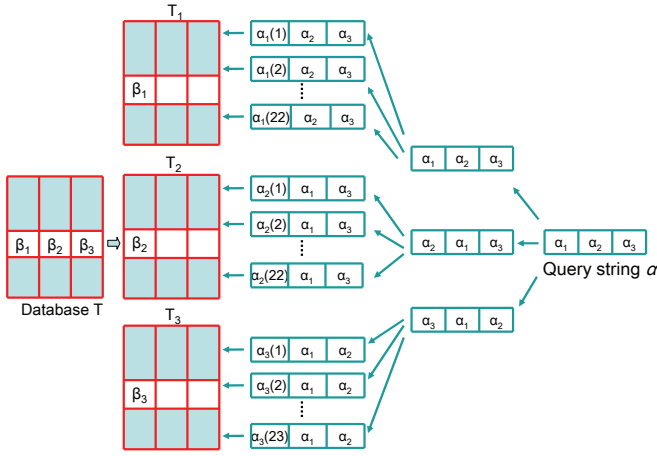
Fig. 1: Illustration of HEngine^s Storage and Query Schemes

C. HEngine^s Query Processing

HEngine^s processes query string α as follows. First, we compute $r\text{-}cut(\alpha)$ using the same r we used to construct the signature set for T. Second, for each of the $\binom{r}{q}$ signature tables $\mathbb{T}$, we compute the corresponding permuted string $\alpha_{\mathbb{T}}$ from the substrings of $r\text{-}cut(\alpha)$. Third, we search for the candidate strings $\beta \in \mathbb{T}$ such that $\beta\text{-}signature$ matches $\alpha\text{-}signature$; that is, those strings whose first q substrings are each within Hamming distance 1 from the corresponding substring in $\alpha_{\mathbb{T}}$. We find such candidate strings using a sequence of query expansion searches on each corresponding substring of $\alpha_{\mathbb{T}}$ with Hamming distance bound 1. That is, we generate all possible substrings s that are within Hamming distance 1 of the first substring of $\alpha_{\mathbb{T}}$ and then perform an exact match query on this substring in $\mathbb{T}$ using binary search. For all the strings in $\mathbb{T}$ whose first substring is equal to s, we repeat the above procedure for the second substring. This process continues until there are no matches or we have completed the signature matching process. Finally, we perform a linear scan on all strings in $\mathbb{T}$ that match the signature to exclude the ones that do not satisfy the query. redAlthough we perform linear scan here, the candidate set containing all strings in $\mathbb{T}$ that match the signature is generally very small. In case that the candidate sets are not small enough, we can recursively apply HEngine^s to further reduce their sizes. Recursive application of HEngine^s is discussed in Section III-D. The final query result is the union of the query results from each signature table. The pseudocode of the above query processing algorithm is shown in Algorithm 2.

We illustrate the query scheme of HEngine^s in Fig. 1. Given a query string α, we first compute $r\text{-}cut(\alpha)$. Second, for each signature table T_i ($1 \leqslant i \leqslant 3$), we generate the corresponding permutation of α. For example, for signature table T_1, the corresponding permutation of α is $\alpha_1|\alpha_2|\alpha_3$. Third, for each signature table and corresponding permuted α string, we perform query expansion on the first substring of the permuted α string to find the strings in T_i that satisfy the given signature. We then do a linear scan on these strings to

---

Algorithm 2: HEngine^s query processing algorithm

Input: (1) Query string α of m bits.
      (2) The signature set S of T based on distance bound k and segmentation factor r.
Output: All the binary strings in T that are within Hamming distance k from α.

1   Function query (α, S, r)
2    $\mathbb{RS} := \emptyset$;
3   Compute $r\text{-}cut(\alpha)$.
4   **for** $i := 1$ **to** $\binom{r}{q}$ **do**
5       $\mathbb{IS} := \text{nextChoice } (r, q)$;
6   **foreach** $x \in \mathbb{IS}$ **do**
7          $\text{front} := \text{front}|\alpha_x$;
8   **foreach** $y \in [1, r] - \mathbb{IS}$ **do**
9          $\text{back} := \text{back}|\alpha_y$;
10      $\alpha_i := \text{front}|\text{back}$;
11      $\mathbb{P} := \{\text{""}\}$;
12      $\mathbb{L} := T_i$;
13   **foreach** $x \in \mathbb{IS}$ **do**
14         $L' := \emptyset$;
15         $\mathbb{P}' := \emptyset$;
16         $\mathbb{R} :=$ the set of all substrings within Hamming distance 1 from α_x;
17   **foreach** $a \in \mathbb{P}$ **do**
18   **foreach** $b \in \mathbb{R}$ **do**
19              $p := a|b$;
20   Compute set Y of all strings in L that start with p by binary search;
21   **if** $Y \neq \emptyset$ **then**
22   insert p into $\mathbb{P}'$;
23                 $L' = L' \bigcup Y$;
24         $\mathbb{P} := \mathbb{P}'$;
25         $L := L'$;
26   **foreach** $\gamma \in L$ **do**
27   **if** γ is within Hamming distance k from $\alpha(i)$ **then**
28   insert γ into $\mathbb{RS}$
29   **return** $\mathbb{RS}$;

---

determine if these strings belong in the final query result. For example, on signature table T_1, we generate all 22 substrings (namely $\alpha_1(1), \alpha_1(2), \cdots, \alpha_1(22)$) that are within Hamming distance 1 from substring α_1 including α_1 itself. For each such substring $\alpha_1(j)$ ($1 \leqslant j \leqslant 22$), we perform binary search on T_1 to find all strings in T_1 that start with $\alpha_1(j)$. Then, we perform a linear scan on all such strings in T_1 that start with $\alpha_1(j)$ to only return those that are within Hamming distance 4 from α. The above process repeats for signature tables T_2 and T_3 (using α_2 and α_3, respectively). The final query result is the union of the query results from all 3 signature tables.

D. Recursive Application of HEngine^s

We can apply HEngine^s recursively. Recall that we construct each signature table by choosing q substrings and moving them to the left. Actually, we can apply the same table construction process to the substring formed by the remaining $\lfloor \frac{k}{2} \rfloor$ columns. That is, instead of performing a linear scan, we search for a new signature match on the remainder of the string. The benefit of this recursion is faster query processing because it reduces the number of linear scans that need to be performed. The cost of this recursion is the increase in storage space plus the time costs of query expansion with Hamming distance 1. Carefully choosing the depth of the recursion helps us to trade off between storage space and query efficiency. We use HEngine^s_n to denote the non-recursive version of HEngine^s

and HEngine$_r^s$ to denote the recursive version of HEngine^s.

We can implement this recursion as follows. For Algorithm 1, in line 25, instead of inserting `front|back` into $\mathbb{PS}$, we generate all permutations of `back` by calling `permutate(back, k, r)`, prepending each of them by `front`, and inserting them into $\mathbb{PS}$. The number of signature tables constructed is now $\binom{r}{q}^h$ where h is the depth of the recursion. For Algorithm 2, instead of a linear scan on all the identified candidate strings (lines 26–28), we process the remaining substrings of α by searching for those strings that match one of the $\binom{r}{q}$ signatures until we reach the base case, at which time we perform linear scan.

E. Handling Database Updates

Occasionally, database T may need to be updated. We consider three types of updates: insertion, deletion and modification. redAll the three update types can be handled efficiently without rerunning the preprocessing algorithm.

To insert a new string β into T, we generate all $\binom{r}{q}$ permutations of β and then insert each permutation into its corresponding signature table maintaining the sorted order of the signature table. The processing cost of a insertion is the sum of the cost of string permutation and $\binom{r}{q}$ times the cost of inserting one string into a sorted array. To delete a string β from T, we generate all the permutations of β, then delete them from the signature tables. The processing cost of deleting a string is the sum of the cost of string permutation and $\binom{r}{q}$ times the cost of deleting one string from a sorted array. redTo modify a string β from T to β_1, we generate all the permutations of β and β_1, then change all the permutations of β in the signature tables to the corresponding permutations of β_1. We also move the strings in each permutation table to keep it sorted. The processing cost of modifying a string to another is the sum of the cost of two string permutations and the cost of modifying one string from a sorted array.

F. Bloom Filter Optimization

We propose to use Bloom filters [8], an efficient data structure for set membership queries, to reduce the cost of query expansion. redA Bloom filter can quickly decide whether an item is in a set with no false negatives and a very small probability of false positive. This is much faster than a hash table. Specifically, we can reduce the number of binary searches performed, which we call the number of probes. For each signature table and for each of the first $r - q$ substrings, we build a Bloom filter. Before we perform equality queries by binary searches, we query the corresponding Bloom filters. If the Bloom filter query result is negative, we skip the binary search; otherwise, we perform the binary search. The storage overhead of Bloom filters is small (about 4 bits per string in T with a false positive rate of 14.6%).

This optimization technique dramatically reduces the cost of query expansion by significantly reducing the number of binary searches. Our experimental results show that it reduces the query processing time of HEngine^s by 57.8% on average.

G. Analysis of Storage Space of HEngine^s

The storage space of HEngine$_n^s$ is $nm\binom{r}{q}$ bits since each signature table requires nm bits, the size of the database T. Here $n = |T|$ and m is the string length. Since we typically choose $r = \lfloor \frac{k}{2} \rfloor + 1$, $nm\binom{r}{q} = (nm\lfloor \frac{k}{2} \rfloor + 1)$. This is essentially half the of the $nm(k+1)$ space used by the non-recursive version of *Manku's* algorithm [19]. The storage space of HEngine$_r^s$ is $(nm\lfloor \frac{k}{2} \rfloor + 1)^h$ bits where h is the recursion depth. This again is essentially half the $nm(k+1)^h$ space used by the recursive version of *Manku's* algorithm [19], assuming the same recursion depth.

H. Analysis of Query Processing Time of HEngine^s

Our time analysis focuses on the maximum number of required probes (*i.e.*, binary searches) in query expansions and the maximum number of required string comparisons in linear scans. Since we are dealing with binary strings, which can be represented by integers in memory, it is reasonable to use the string comparisons as the basic operation. For ease of analysis, we assume that n is a power of 2, $\log_2(n) = d$, and m is divisible by r. We also assume that the strings in T are truly random. Our experimental results imply that our algorithms work well even for strings that are not truly random.

1) Maximum Number of Probes: In HEngine$_n^s$, the maximum number of probes in one table is $(m/r + 1)^q$. The $m/r + 1$ term comes from the query expansion of a length m/r substring with Hamming distance bound 1. The q term arises because we may have to perform query expansion for q substrings. As we have $\binom{r}{q}$ signature tables, the maximum number of probes in HEngine$_n^s$ is $(\frac{m}{r} + 1)^q \cdot \binom{r}{q}$.

We now analyze the maximum number of probes in HEngine$_r^s$. The key observation is that if x is the total number of bits that need to be processed at level i, then $\frac{x}{r} \lfloor \frac{k}{2} \rfloor$ bits remain to be processed at level $i + 1$ (ignoring the fact that r may not divide x perfectly). For example, initially there are m bits to be processed. If we recurse one level, this means only $\lfloor k/2 \rfloor$ length m/r substrings remain to be processed at the next level for a total of $\frac{m}{r} \lfloor \frac{k}{2} \rfloor$ bits. In general, there are $\frac{m}{r^i} \lfloor \frac{k}{2} \rfloor^i$ bits remaining to be processed at recursion level i where $i = 0$ is the initial level. We can then replace m in the above formula $(\frac{m}{r} + 1)^q \cdot \binom{r}{q}$ by this value to determine the maximum number of probes that might occur at each level of recursion. Finally, assuming the worst case where each probe is successful, we simply multiply the terms from each level together to get a maximum number of probes of $\prod_{i=0}^{h-1}((m(\lfloor \frac{k}{2} \rfloor)^i \cdot \frac{1}{r^{i+1}} + 1)^q) \cdot \binom{r}{q}^h$.

2) Maximum Number of Hamming Distance Comparisons in Linear Scans: We first make a worst case assumption that every probe produces at least one match. Stated another way, each possible signature that can produce a match does produce a match. We then apply our assumption that the strings are random to compute the expected number of strings in the database that have each signature pattern. In HEngine$_n^s$, the signature will consist of qm/r bits. Given that there are a total of 2^d strings, we expect $2^{d-(qm)/r}$ strings to possess this

558

signature. This leads to $\left(\frac{m}{r}+1\right)^q \cdot \binom{r}{q} \cdot 2^{d-\frac{qm}{r}}$ as our bound. For HEngine$_r^s$, we computed above that the remaining bits to be processed after h iterations is $\frac{m}{r^h}\lfloor\frac{k}{2}\rfloor^h$. This means that the signature after h iterations contains $m(1-\lfloor\frac{k}{2}\rfloor^h\frac{1}{r^h})$ bits and the expected number of strings to match any signature is $2^{d-m(1-\lfloor\frac{k}{2}\rfloor^h\frac{1}{r^h})}$ giving a total number of linear scans $2^{d-m(1-\lfloor\frac{k}{2}\rfloor^h\frac{1}{r^h})} \cdot \prod_{i=0}^{h-1}((m(\lfloor\frac{k}{2}\rfloor)^i \cdot \frac{1}{r^{i+1}}+1)^q) \cdot \binom{r}{q}^h$.

I. Generalization to $z > 1$

In our definition of compatible signatures, we use $z = 1$. This can be generalized to any z such that $0 \leqslant z \leqslant k$. That is, for a query string α, we find those strings whose first q substrings are each within Hamming distance z from the corresponding substring in every permutated string of α. When $z = 0$, HEngine^s reduces to *Manku's* algorithm. When $z = k$, HEngine^s reduces to query expansion. The chose of z affects the number of probes and number of Hamming distance comparisons in linear scans. For $m = 64$, we observe in our experiments that $z = 1$ is the best chose. A larger z will result in too many probes, thus reduces query processing speed. We will perform a detailed analysis of the query processing time with respect to varying z in the future.

IV. DYNAMIC HAMMING DISTANCE RANGE QUERIES

In this section, we present our dynamic problem algorithm HEngine^d. Because k is no longer fixed, the query is now (α, k). We first present the basic idea of HEngine^d. Second, we describe how HEngine^d preprocesses the database and processes queries. Third, we discuss how we can efficiently update the database. Finally, we give the time and space analysis of HEngine^d.

A. Overview of HEngine^d

We wish to again use signatures to solve the dynamic Hamming distance range query problem. However, unlike the static problem where we can preprocess T knowing k, we must now preprocess T without knowing k. This means we cannot optimize our segmentation factor r for k, and we cannot build signature tables that depend on k. Thus, HEngine^d uses a simpler signature scheme than HEngine^s.

The intuition behind HEngine^d is that the dynamic problem is easier to solve when the bound k is smaller. HEngine^d uses a divide-and-conquer strategy where it recursively divides the problem into subproblems with smaller k and solves each subproblem separately. To create a generic divide-and-conquer solution that applies to any k, HEngine^d uses a segmentation factor of 2 and thus recursively applies a series of 2-*cut*s on α and all the strings in T. HEngine^d is based on Theorem 4.1, which can be proved by contradiction.

Theorem 4.1: For any two strings α and β such that $HD(\alpha, \beta) \leqslant k$, consider 2-$cut(\alpha)$ and 2-$cut(\beta)$. It must be the case that $HD(\alpha_1, \beta_1) \leqslant \lfloor k/2 \rfloor$ or $HD(\alpha_2, \beta_2) \leqslant \lfloor k/2 \rfloor$.

With recursion, HEngine^d essentially generates all segmentation factors that are powers of 2 up to some bound. The base case for a particular range query with bound k occurs for $t = 2^{\lfloor \log_2 k \rfloor + 1}$. Intuitively, HEngine^d uses a segmentation

factor of t (with $\log_2 t$ levels of recursion) to cut each string into t length m/t substrings. The base case signature is that one pair of the t resulting pairs of substrings from t-$cut(\alpha)$ and t-$cut(\beta)$ must be identical in order for $HD(\alpha, \beta) \leqslant k$.

B. HEngine^d Preprocessing

We first introduce some notations that facilitate the description of HEngine^d. Suppose we apply 2-cut to T. We use T_L to denote the set of left substrings of T (*i.e.*, $T_L = \{\beta_1 | \beta \in T\}$) and T_R to denote the set of right substrings of T (*i.e.*, $T_R = \{\beta_2 | \beta \in T\}$). For any string $u \in T_L$, let $A_{T,u} = \{v \mid uv \in T\}$. Similarly, for any string $u \in T_R$, let $B_{T,u} = \{v \mid vu \in T\}$.

HEngine^d preprocess T as follows. Given a database T, we first make a replica of T, denoted as T'. Second, we apply 2-cut to all the strings in both T and T' dividing all the strings in T and T' into two substrings and generating T_L and T_R. Third, we sort the strings in T using $\beta_1\beta_2$ while we sort the strings in T' using $\beta_2\beta_1$. The purpose of sorting T and T' is to group the strings in T into blocks based on the substrings in T_L and group the strings in T' into blocks based on the substrings in T_R. We further apply these three steps recursively to T_L and T_R, respectively. Because neither T_L nor T_R has repetitive strings, T_L and T_R potentially have fewer strings than T. To facilitate efficient query processing, we associate each string $u \in T_L$ with the index of the first string $\beta \in T$ such that $\beta_1 = u$. Similarly, we also associate each string $v \in T_R$ with the index of the first string $\beta \in T'$ such that $beta_2 = v$. We store these indexes in two separate arrays, which we call *index arrays*. The index arrays for T and T' are empty. The recursive splitting of tables continues until the string length in a table reaches a given threshold denoted λ. We call this set of signature tables the signature set of T based on threshold λ. Fig. 2 illustrates this process until the table width is reduced to 4.
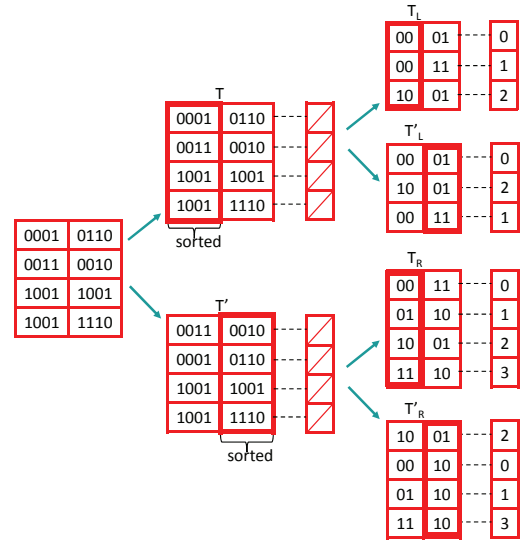

Fig. 2: HEngine^d Storage Scheme

The pseudocode of the HEngine^d preprocessing algorithm

is in Algorithm 3. This procedure runs offline and runs only once regardless of k.

linear scan of the current signature table. The pseudocode of the HEngine^d query algorithm is in Algorithm 4.

Algorithm 3: HEngine^d preprocessing algorithm

Input: (1) Database T where the length of each string is m.
      (2) Table splitting threshold λ.
Output: The signature set of T based on threshold λ.

1   Function build (T, λ)
2   **begin**
3      $\mathbb{RS} = \emptyset$;
4   **if** width$(T) = \lambda$ **then**
5   sort T;
6   insert T into $\mathbb{RS}$;
7   **else**
8          $T' := T$;
9   apply 2-cut to all strings in T and T';
10   sort T based on $\beta_1\beta_2$;
11   sort T' based on $\beta_2\beta_1$;
12       $T_L := \emptyset$; $T_R := \emptyset$;
13   **for** $i := 0$ **to** sizeof $(T)-1$ **do**
14   let $T[i]$ be uv;
15   **if** $u \notin T_L$ **then**
16   insert u into T_L;
17   associate u with the index i;
18   **for** $i := 0$ **to** sizeof $(T')-1$ **do**
19   let $T'[i]$ be uv;
20   **if** $v \notin T_R$ **then**
21   insert v into T_R;
22   associate v with the index i;
23   insert T into $\mathbb{RS}$;
24   insert T' into $\mathbb{RS}$;
25       $L=$build (T_L, λ);
26   **foreach** *table* $t \in L$ **do**
27   insert t into $\mathbb{RS}$;
28       $R=$build (T_R, λ);
29   **foreach** *table* $t \in R$ **do**
30   insert t into $\mathbb{RS}$;
31   **return** $\mathbb{RS}$;

Algorithm 4: HEngine^d query processing algorithm

Input: (1) Query string α of m bits.
      (2) The signature table set of T based on threshold λ.
      (3) Hamming distance bound k.
Output: All strings in T within Hamming distance k from α.

1   Function query (k, T, α)
2   **begin**
3   **if** $k = 0$ **then**
4   **if** $\alpha \in T$ **then**
5   **return** $\{\alpha\}$;
6   **else**
7   **return** $\emptyset$;
8   **if** length$(\alpha) = \lambda$ **then**
9   **return** linearscan(T, α, k);
10   **else**
11        $\mathbb{RS} = \emptyset$
12   Compute 2-$cut(\alpha) = (\alpha_1, \alpha_2)$;
13       $\mathbb{L} =$ query$(\lfloor k/2 \rfloor, T_L, \alpha_1)$;
14   **foreach** $\mu \in \mathbb{L}$ **do**
15          $\mathbb{RS} = \mathbb{RS} \bigcup$ linearscan$(A_{T,\mu}, \alpha, k)$;
16       $\mathbb{R} =$ query$(\lfloor k/2 \rfloor, T_R, \alpha_2)$;
17   **foreach** $\omega \in \mathbb{R}$ **do**
18          $\mathbb{RS} = \mathbb{RS} \bigcup$ linearscan$(B_{T,\omega}, \alpha, k)$;
19   **return** $\mathbb{RS}$;

C. HEngine^d Query Processing

Given a query (α, k), we process the query using a recursive divide-and-conquer strategy. First, we compute 2-$cut(\alpha) = (\alpha_1, \alpha_2)$. Second, we search for all strings in T_L within Hamming distance $\lfloor k/2 \rfloor$ of α_1, and we search for all strings in T_R within Hamming distance $\lfloor k/2 \rfloor$ of α_2. That is, we solve a subproblem in T_L and T_R with bound $\lfloor k/2 \rfloor$. Theorem 4.1 ensures that any string we are searching for must match have a substring in T_L or T_R within the required bound of α_1 or α_2. Consider any substring μ that is found within T_L (we handle substrings found in T_R analagously). We then locate the first string in T whose left substring is μ using the index associated with μ, and we linearly scan all strings whose left substring is μ to find the strings that satisfy the query. In other words, we linearly scan $A_{T,\mu}$ to find strings within Hamming distance k from α. Since T is sorted based on its left substring, all the strings in $A_{T,\mu}$ are located consecutively in T.

To search for the strings in T_L and T_R, we recursively use the same query process. We have two termination conditions. The first is when the Hamming distance bound is 0, in which case we simply perform a binary search in the current signature table to find the query string. The second is when the length of the query string reaches λ, in which case we perform a

D. Handling Database Updates

Similar to HEngine^s, we consider three types of database updates: insertion, deletion and modification. In HEngine^d, all the update operations can be handled efficiently without running preprocessing again.

To insert a new string β into T, we first compute 2-$cut(\beta) = (\beta_1, \beta_2)$. Second, we insert β into T and we insert β into T' based on $\beta_2\beta_1$. We update all appropriate index arrays. We then recursively apply the same process to insert β_1 into T_L and β_2 into T_R. This recursion continues until the string length reaches the threshold λ. The processing cost of inserting a binary string is the number of signature tables times the cost of inserting one string in a sorted table.

Deleting a string β from T is the reverse process of inserting a string into T. First, we compute 2-$cut(\beta) = (\beta_1, \beta_2)$. Second, we delete β from both T and T'. We update all appropriate index arrays. We then recursively apply the same process to delete β_1 into T_L and β_2 from T_R. This recursion continues until the string length reaches the threshold λ. The processing cost of deleting a string is the number of signature tables times the cost of deleting one string from a sorted table.

redTo update a string β to γ, we first compute 2-$cut(\beta) = (\beta_1, \beta_2)$ and 2-$cut(\gamma) = (\gamma_1, \gamma_2)$. Second, we change β to γ in both T and T'. We move the strings in T and T' to keep them sorted. We update all appropriate index arrays. We then recursively apply the same process to update β_1 in T_L and β_2 in T_R. If $\beta_1 = \gamma_1$, we do not need to update T_L. If $\beta_2 = \gamma_2$, we do not need to update $\mathbb{T}_R$. This recursion continues until the string length reaches the threshold λ. The upper bound of the processing cost of updating a string is the number of

signature tables times the cost of updating one string from a sorted table.

E. Analysis of Storage Space of HEngine^d

Theorem 4.2: The space complexity of the HEngine^d algorithm is $O(nm \log m)$.

Proof: We use s(n,m) to denote the storage space of HEngine^d for a database of size n and width m. For simplicity, we assume that m is a power of 2. Suppose the storage cost of storing n sorted strings of length m along with the indices associated with them is cnm where $c \geqslant 1$ is a positive constant. Clearly, s(n,m) follows the following recurrence relation: $s(n,m) \leqslant 2cnm + s(|T_L|, \frac{m}{2}) + s(|T_R|, \frac{m}{2})$.

Next, we prove the following inequality by induction, which is obviously true for the base case of $n = 1$ or $m = 1$.

$$s(n,m) \leqslant 2cnm(1 + \log_2 m) \tag{1}$$

The proof of the inductive step is:

$$
\begin{aligned}
s(n,m) &\leqslant 2cnm + s(|T_L|, \frac{m}{2}) + s(|T_R|, \frac{m}{2}) \\
&\leqslant 2cnm + 2c(|T_L| + |T_R|)\frac{m}{2}(1 + \log_2 \frac{m}{2}) \\
&\leqslant 2cnm + 2cnm(1 + \log_2 \frac{m}{2}) \\
&= 2cnm + 2cnm \log_2 m \\
&= 2cnm(1 + \log_2 m) \tag{2}
\end{aligned}
$$

This proves (1), which further proves Theorem 4.2. ∎

F. Analysis of Query Processing Time HEngine^d

Our time analysis focuses on the maximum number of (Hamming distance) comparisons in linear scans performed in HEngine^d. For simplicity, we assume that the strings in the database are uniformly distributed and m is a power of 2. We call a linear scan in one signature table a *probe*. Let $d = \log_2(n)$. Let $t(k,m,d)$ denote the maximum number of comparisons for a given table of size 2^d and width m under Hamming constraint k. In the root level signature table, the average number of strings that share the same left or right substring is $2^{d-\frac{m}{2}}$. Based on Algorithm 4, the recursion relation (3) holds.

$$t(k,m,d) = 2t(\frac{k}{2}, \frac{m}{2}, \frac{d}{2})2^{d-\frac{m}{2}} \tag{3}$$

Next, we analyze the base case for Hamming distance comparisons where $k = 1$ (we perform only binary search and no Hamming distance comparisons when $k = 0$). Because in each recursion level we reduce k by at least a factor of 2, the number of recursion levels for the bound to reach 1 is $\lfloor \log_2 k \rfloor + 1$. Furthermore, the number of signature tables when the bound reduces to 1 is $\zeta = 2^{\lfloor \log_2 k \rfloor + 1}$. Thus, when $k = 1$, the maximum number of comparisons is $2^{\frac{d}{\zeta} - \frac{m}{2\zeta}}$ for one probe. Since we have two probes (left and right substrings), the maximum number of comparisons in one signature table

is $2^{\frac{2d}{\zeta} - \frac{m}{\zeta}}$. Combined with equation (3), we can deduce the expression of $t(k,m,d)$ as follows.

$$
\begin{aligned}
t(k,m,d) &= 2t(\frac{k}{2}, \frac{m}{2}, \frac{d}{2})2^{d-\frac{m}{2}} \\
&= 2^2 t(\frac{k}{4}, \frac{m}{4}, \frac{d}{4})2^{\frac{d}{2}-\frac{m}{4}}2^{d-\frac{m}{2}} \\
&= \zeta 2^{2\frac{d}{\zeta} - \frac{m}{\zeta}} 2^{2\frac{d}{2} - \frac{m}{2}} \ldots 2^{d-\frac{m}{2}} \\
&= 2^{\lfloor \log_2 k \rfloor + 1} 2^{(2d-m)(1 - \frac{1}{2^{\lfloor \log_2 k \rfloor + 1}})} \tag{4}
\end{aligned}
$$

V. EXPERIMENTAL RESULTS

A. Results on Real-world Dataset

1) Dataset: The dataset used in our experiments consists of about 0.46 million Web pages and is drawn from the summary version of the WEBSPAM-UK2007 collection [1], which is based on a Web crawl of the .uk domain conducted in May 2007. We apply the simhash algorithm in [11] to each Web page to get a 64-bit fingerprint of the Web page. The fingerprints have the property that if two documents are near-duplicates, then the Hamming distance between their fingerprints is small [19].

The data set is not uniformly distributed. We tested uniformity by estimating the average Hamming distance distribution from a random query. We estimated this distribution by randomly selecting 30 query fingerprints and using their average distance distributions to approximate the Hamming distance distribution. The normalized histogram appears to be a normal distribution centered at Hamming distance 28 and 29 rather than 32 and 33 for uniformly distributed data set.

2) Performance Evaluation: We now report our experimental results on the preprocessing, storage space, query time, and database update time of our HEngine algorithms. Note that all the figures reporting performance results are in log scale. We implemented both HEngine^s and HEngine^d using Java 1.6. redFor HEngine^s, the choice of r is based on the theoretical analysis on the number of probes and number of number of Hamming distance comparisons in linear scans in Sections III. Since we observe that the number of probes affects query processing time more in our datasets, we chose the smallest value of $r = k/2 + 1$ for all experiments. As Manku's algorithm [19] represents the state-of-the-art static Hamming distance range query algorithm, we also implemented Manku's algorithm using Java 1.6 in order to conduct a side-by-side comparison of HEngine^s with Manku's algorithm. Given the size of the database, we did not use recursion for either HEngine^s or Manku's algorithm. We also implemented the PartEnum algorithm in [3] with $k_1 = k/2$ and $k_2 = 2$. With these settings, PartEnum is essentially a two-level non-recursive version of Manku's algorithm that uses less storage space but has slower query processing time than Manku's algorithm. Finally, we implemented the linear scan algorithm, denoted LS, as the baseline. Our experiments were carried out on a quad-core Windows Server running Microsoft Windows Server 2003 with 8GB memory and quad 2.6Ghz AMD Opteron processors.

Storage Space Cost: Fig. 3 shows the storage space cost of the five algorithms: HEngine^s, HEngine^d, Manku's algorithm, PartEnum and LS. Linear scan has the lowest storage space cost since it only holds the original database strings. In our experiments, HEngine^s uses on average 4.65 times less space than Manku's algorithm and 3.39 times less space than PartEnum. Furthermore, HEngine^s uses the same data structure and thus has the same cost for $k = 2i$ and $k = 2i+1$ for $i \geqslant 1$. HEngine^d uses on average 1.7 times more space than LS uses, 25.4% less space than HEngine^s uses when k exceeds 3, and 6.12 times less space than Manku's algorithm uses.
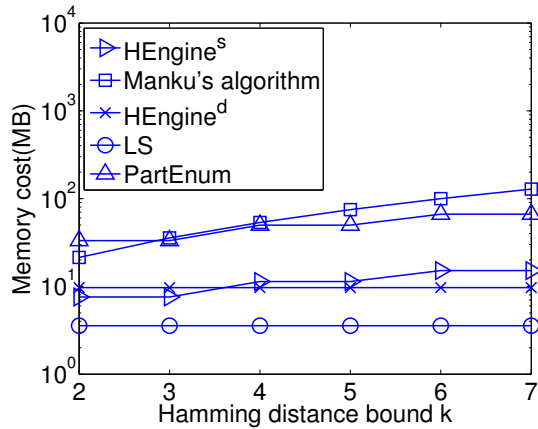
Fig. 3: Storage space cost

Query Processing Cost: Fig. 4 shows the average query processing time for the five algorithms. In our experiments, we randomly generate 1000 query strings and record the query time of each algorithm for each string. We varied k from 2 to 7. This figure shows that HEngine^s is faster than Manku's algorithm except for $k = 2$ and $k = 4$. We also observe that the query processing time of Manku's algorithm grows faster than that of HEngine^s. Our experiment results show that on average, HEngine^s is 16% faster than Manku's algorithm and 2.0 times faster than PartEnum. On average, as expected, HEngine^d is 62.2% slower than HEngine^s and 50.7% slower than Manku's algorithm, but it is 91.12 times faster than LS for $k \leqslant 3$ and 24.33 times faster than LS for $k > 3$.

Because we randomly generate query strings, the queries are likely to have no matches in the database, particularly when k is small. We show a histogram of the number of the number of query strings that match given number of database strings for $k = 2, 4,$ and 7 in Fig. 5. As the number of matched database strings increases, the average query processing time increases as would be expected. We show the average query processing time for specific values of matched database strings for $k = 7$ for our algorithms in Fig. 6.

Preprocessing Cost: Fig. 7 shows the preprocessing time of the two HEngine algorithms, Manku's algorithm and the PartEnum algorithm. In our experiments, HEngine^s's preprocessing algorithm is, on average, 4.15 times faster than Manku's preprocessing algorithm and 6.7 times faster than
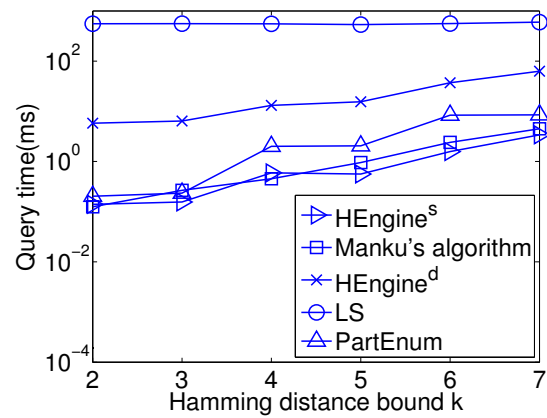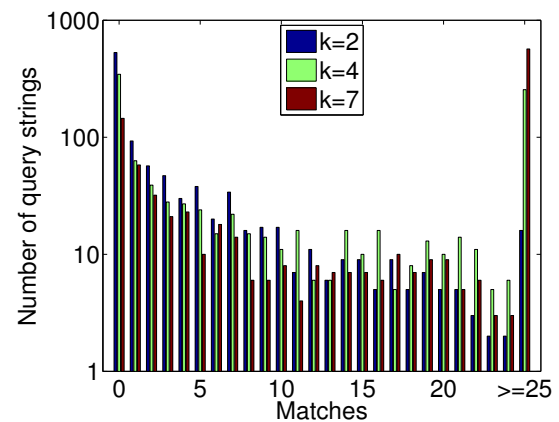

Fig. 4: Query processing time

Fig. 5: Histogram of number of query strings that match a given number of database strings

PartEnum's. HEngine^s is significantly faster than Manku's algorithm in the preprocessing phase because HEngine^s generates fewer duplicated tables. Although the preprocessing process can be done off-line, in some applications such as
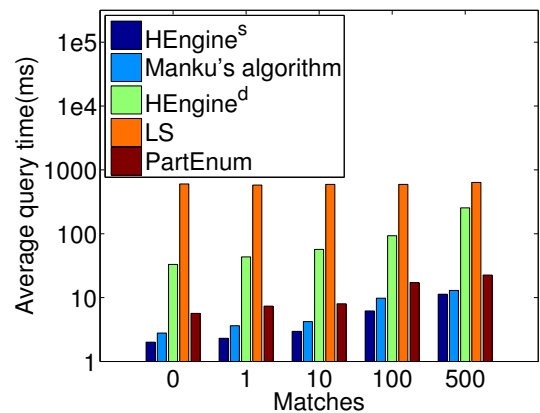

Fig. 6: Average query time versus number of matched database strings

the "batch query" application described in [19], it is a non-negligible factor to be considered. Efficient preprocessing is also important for applications that require a periodic flush of the database [19]. In our experiments, HEngine^d's preprocessing algorithm is, on average, 58.49 times slower than HEngine^s's preprocessing algorithm and 13.35 times slower than Manku's preprocessing algorithm. HEngine^d uses more preprocessing time because it has more replicas of the database and it needs to sort each of them. But once the sorting is done, database updates can be efficiently processed.
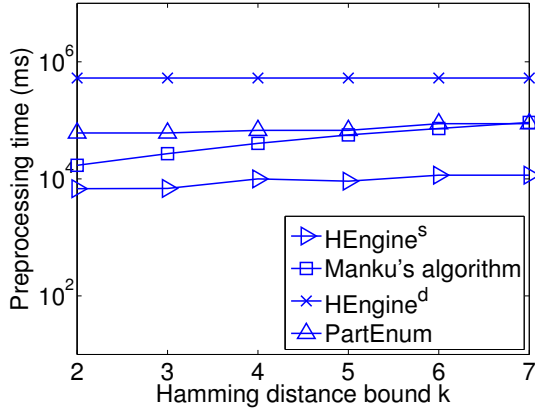
Fig. 7: Preprocessing time

Database Update Cost: Fig. 8 shows the database update time of the two HEngine algorithms, Manku's algorithm, and the PartEnum algorithm. In our experiments, we randomly generated 1000 strings to insert or delete and then updated the storage data structure. For database updates, our experiment results show that on average, HEngine^s is 4.53 times faster than Manku's algorithm and 17 times faster than PartEnum; HEngine^d is 2.28 times faster than Manku's algorithm and 9.64 times faster than PartEnum. The performance gain of our algorithms grows monotonically with k.
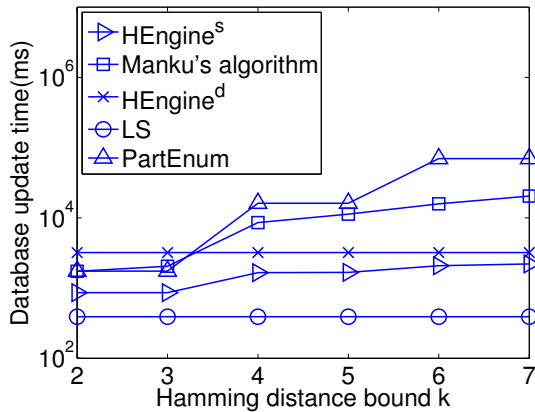

Fig. 8: Database update time

B. Results on Synthetic Dataset

We randomly generated 0.5 million 64-bit binary strings and evaluated the algorithms on this synthetic dataset. Since each string is equally likely to appear in the data set, the average Hamming distance distribution from a query is the normal distribution centered at Hamming distances 32 and 33. The experimental results are shown in Fig. 9– Fig. 12. These figures have the same trend as those on the real-world dataset, though the query processing time advantage of HEngine^s is reduced. In particular, we see that HEngine^s performs at least as well as Manku's algorithm for all but $k = 5$ and $k = 7$, and HEngine^s is 2.6% faster than Manku's algorithm and 1.79 times faster than PartEnum on average.
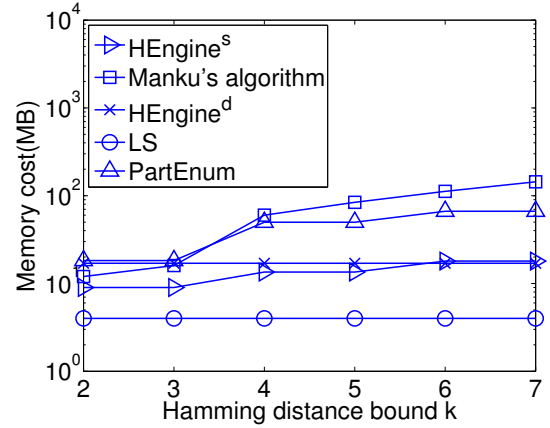

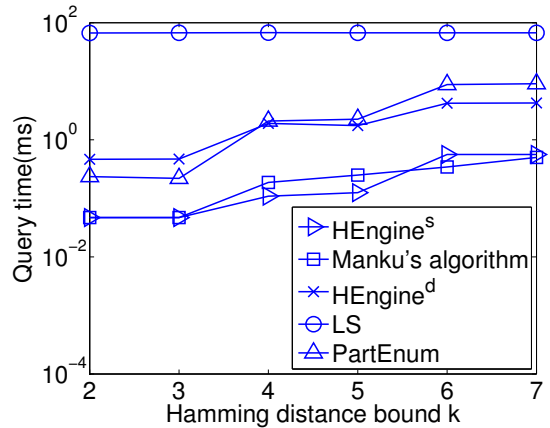Fig. 9: Synthetic storage space cost

Fig. 10: Synthetic query processing time

VI. CONCLUSION

We made three key contributions in this paper. First, we proposed a static Hamming distance range query algorithm called HEngine^s, which uses 4.65 times less memory and is 16% faster than the prior art Manku's algorithm in our experiments. Second, we proposed a dynamic Hamming distance range query algorithm called HEngine^d, which is 46.42 times faster than linear scan while using only 2 times more memory in our experiments. Last, we implemented both static and dynamic Hamming distance range query algorithms as well as Manku's algorithm using the same programming language. Side-by-side
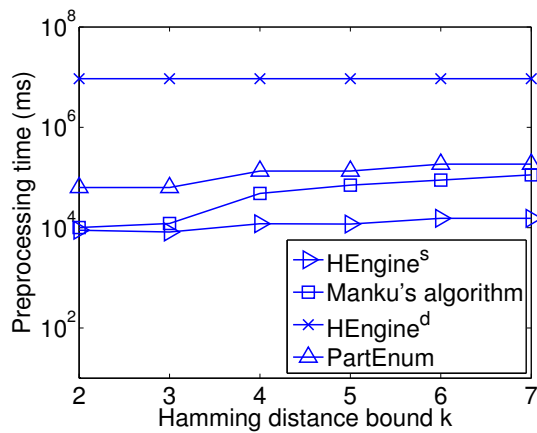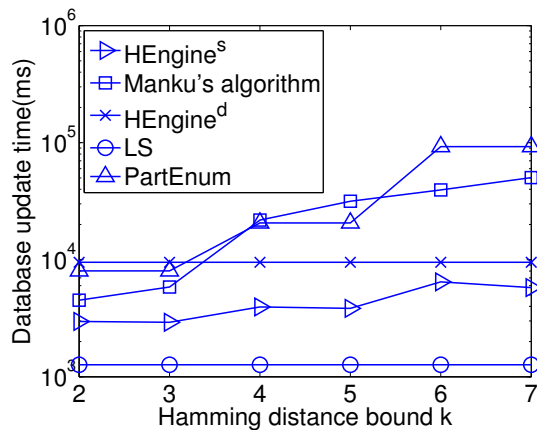
Fig. 11: Synthetic preprocessing time

Fig. 12: Synthetic database update time

comparison on both real-world and synthetic datasets showed clear advantages in terms of storage space, query processing, preprocessing, and database update costs.

REFERENCES

[1] Yahoo! research, web collection uk-2007. http://barcelona.research. yahoo.net/webspam/datasets/. Crawled by the Laboratory of Web Algorithmics, University of Milan. URL retrieved in April 2009.
[2] A. Andoni and R. Krauthgamer. The computational hardness of estimating edit distance. In *Proc. 48th FOCS*, 2007.
[3] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *Proc. 32nd VLDB*, 2006.
[4] A. N. Arslan. Efficient approximate dictionary look-up over small alphabets. Technical report, University of Vermont, 2005.
[5] A. N. Arslan and Ömer Eğecioğlu. Dictionary look-up within small edit distance. In *Proc. 8th COCOON*, pages 127–136, 2002.
[6] Z. Bar-Yossef, T. S. Jayram, R. Krauthgamer, and R. Kumar. Approximating edit distance efficiently. In *Proc. 45th FOCS*, 2004.
[7] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proc. 16th WWW*, 2007.
[8] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM*, 13(7):422–426, 1970.
[9] G. S. Brodal and L. Gąsieniec. Approximate dictionary queries. In *Proc. 7th CPM*, pages 65–74, 1996.
[10] G. S. Brodal and S. Venkatesh. Improved bounds for dictionary look-up with one error. *Information Processing Letters*, 75:57–59, 2000.
[11] M. Charikar. Similarity estimation techniques from rounding algorithms. In *Proc. 34th STOC*, 2002.
[12] D. Dolev, Y. Harari, N. Linial, N. Nisan, and M. Parnas. Neighborhood preserving hashing and approximate queries. *Journal on Discrete Mathematics*, 15:73–85, 2002.
[13] D. Greene, M. Parnas, and F. Yao. Multi-index hashing for information retrieval. In *Proc. 35th FOCS*, volume 0, pages 722–731, 1994.
[14] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. *SIAM Journal on Computing*, 30(2):457–474, 2000.
[15] J. Landré and F. Truchetet. Image retrieval with binary hamming distance. In *Proc. 2nd VISAPP*, 2007.
[16] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A fast similarity join algorithm using graphics processing units. In *Proc. 24th ICDE*, Cancun, Mexico, April 2008.
[17] Q. Lv, M. Charikar, and K. Li. Image similarity search with compact data structures. In *Proc. 13th CIKM*, pages 208–217, 2004.
[18] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Ferret: a toolkit for content-based similarity search of feature-rich data. In *Proc. 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2006.
[19] G. S. Manku, A. Jain, and A. D. Sarma. Detecting nearduplicates for web crawling. In *Proc. 16th WWW*, May 2007.
[20] M. L. Miller, M. A. Rodriguez, and I. J. Cox. Audio fingerprinting: Nearest neighbor search in high-dimensional binary space. In *MMSP*, 2002.
[21] M. L. Miller, M. A. Rodriguez, and I. J. Cox. Audio fingerprinting: nearest neighbor search in high dimensional binary spaces. *Journal of VLSI Signal Processing, Springer*, 41(3):285–291, 2005.
[22] M. Minsky and S. Papert. *Perceptrons*. MIT Press, Cambridge, MA, 1969.
[23] G. Qian, Q. Zhu, Q. Xue, and S. Pramanik. The nd-tree: A dynamic indexing technique for multidimensional non-ordered discrete data spaces. In *Proc. 29th VLDB*, 2003.
[24] Z. Wang, W. Dong, W. Josephson, Q. Lv, M. Charikar, and K. Li. Sizing sketches: A rank-based analysis for similarity search. In *Proc. 2007 SIGMETRICS*, San Diego, CA, USA, June 2007.
[25] C. Xiao, W. Wang, X. Li, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *Proc. 17th WWW*, 2008.
[26] H. Yang and Y. Wang. A LBP-based face recognition method with hamming distance constraint. In *Proc. Fourth ICIG*, 2007.
[27] A. C. Yao and F. F. Yao. Dictionary look-up with one error. *Journal of Algorithms*, 25:194–202, 1997.