I/O-Efficient Dictionary Search with One Edit Error

Chin-Wan Chung[1], Yufei Tao[2], and Wei Wang[3]

[1]Korean Advanced Institute of Science and Technology, Daejeon, Korea
[2]Chinese University of Hong Kong, New Territories, Hong Kong
[3]University of New South Wales, Sydney, Australia
chungcw@kaist.edu,taoyf@cse.cuhk.edu.hk,weiw@cse.unsw.edu.au

Abstract. This paper studies the *1-error dictionary search* problem in external memory. The input is a set D of strings whose characters are drawn from a constant-size alphabet. Given a string q, a query reports the ids of all strings in D that are within 1 edit distance from q. We give a structure occupying $O(n/B)$ blocks that answers a query in $O(1 + \frac{m}{wB} + \frac{k}{B})$ I/Os, where n is the total length of all strings in D, m is the length of q, k is the number of ids reported, w is the size of a machine word, and B is the number of words in a block.

1 Introduction

In this paper, we consider the *1-error dictionary search* problem defined as follows. The input D—the *dictionary*—is a set of strings whose characters are drawn from a constant-size alphabet Σ. Each string is associated with a distinct integer as its id. Given a string q with characters in Σ, a query reports the ids of those strings $s \in D$ with $dist(s, q) \leq 1$, where $dist(s, q)$ is the *edit distance* between s and q (a.k.a. the *Levenshtein distance*).[1]

The problem is fundamental to search engines that aim to tolerate typos in the keywords entered by users. It is well known [12, 13] that between 80%-95% of the typos in practice are 1-edit-distance errors, thus providing a strong motivation to support 1-error dictionary search efficiently.

Computation Model. We study the problem in the standard *external memory* (EM) model [1]. Under this model, a machine has a disk that is an infinite sequence of *blocks*, each of which contains B words. Computation takes place only in (internal) memory, whose size M (in number of words) is at least $2B$. An I/O exchanges a block of data between the disk and memory. The *time* of an algorithm is the number of I/Os performed. The *space* of a structure is the number of blocks in the shortest prefix of the disk containing all the bits of the structure. Denote by w the number of bits in a machine word. We make the standard assumption that $w = \Omega(\lg B)$ (otherwise, the machine cannot even encode the address space of all the words in memory).

We will also make frequent use of the following notations:

- n: the total length of all the strings in D;

[1] Specifically, $dist(s, q)$ is the smallest number of the following edit operations needed to convert s to q: (i-ii) inserting or deleting a character, and (iii) replacing a character with another one.

space	query	update	source	remarks
$O(n/B)$	$O(m+k)$	$O(l)$ amortized (ins. only)	[5]	assume that data and query strings are equally long
$O(\frac{t}{B}+\frac{n}{wB})$	$O(m+k)$	-	[3]	
$O(\frac{n}{B}\lg n)$	$O(\frac{m}{B}+\frac{k}{B}+\lg n \lg\lg_B n)$	-	[11]	designed for the more general "full-text indexing" problem
$O(n/B)$	$O(1+\frac{m}{wB}+\frac{k}{B})$	$O(l)$ expected	**New**	

Table 1. Comparison of our and previous results in external memory

- t: the number of strings in D;
- m: the length of a query string;
- k: the number of qualifying strings for a query.

We define $\lg_b x = \max\{1, \log_b x\}$ with $b = 2$ if omitted.

Previous Results. The 1-error dictionary search problem was first studied in internal memory. Belazzougui [3] presented a static structure of $O(t+n/w)$ space that answers a query in $O(m+k)$ time. Belazzougui and Venturini [4] showed that the space can be reduced if the dictionary has small entropy, but their result does not improve that of [3] in general. Brodal and Gasieniec [5] considered a special instance of the problem, where all the data and query strings have the same length m (i.e., a query with string q essentially retrieves the strings whose *hamming distances* are within 1 from q). For this instance, they gave a structure of space $O(n)$ that answers a query in $O(m+k)$ time; in addition, their structure is semi-dynamic: it supports the insertion of a length-l data string in $O(l)$ amortized time.

The above structures, when applied in external memory, incur $O(m+k)$ I/Os answering a query. This, unfortunately, is at least a factor of B away from what we would like to achieve. Currently, the most I/O-efficient structure is due to Hon et al. [11]. Their structure, which is intended for a more general problem called *full-text indexing*, uses $O(\frac{n}{B}\lg n)$ blocks, and answers a query in $O(\frac{m}{B}+\frac{k}{B}+\lg n \cdot \lg\lg_B n)$ I/Os. It does not support updates efficiently.

It is worth mentioning that, while in this paper we focus on queries with 1-edit-distance errors, progress has also been made in the past decade towards tolerating a larger number of errors. Interested readers may refer to [6, 7, 15] for entry points into the literature.

Our Results. We give a new structure for 1-error dictionary search that uses $O(n/B)$ space, answers a query in $O(1+\frac{m}{wB}+\frac{k}{B})$ I/Os, and supports the insertion and deletion of a length-l string in $O(l)$ expected I/Os (see Table 1 for a comparison). With B set to 1, our structure also works in the RAM model directly, and provides a new tradeoff between space and query time there.

Remarks. Henceforth, we will focus on a binary alphabet $\Sigma = \{0, 1\}$. Our techniques can be extended to any constant-size alphabet, without affecting the claimed complexities. Clarification will be duly made when this is not straightforward.

2 Exact Matching

In this section, we revisit the (precise) *dictionary search* problem, whose solution will be useful later. Specifically, the input is the same dictionary D as in 1-error dictionary search. Given a string q, we want to report its id in D if $q \in D$, or declare its absence otherwise. This problem is also commonly known as *exact matching*.

Given a string s, we denote by $|s|$ the length of s. For an $i \in [1, |s|]$, let s_i be the i-th character of s. Given i, j with $1 \leq i \leq j \leq |s|$, let $s_{i..j}$ be the substring of s starting and ending at s_i and s_j, respectively. Specially, if $i > j$, $s_{i..j}$ is an empty string.

2.1 Preliminaries

Let us first review a result on perfect hashing. Given an integer $x > 0$, $[x]$ represents the set $\{0, 1, ..., x-1\}$. Consider a set $S \subseteq [U]$ for some positive integer U. Set $n = |S|$. Let f be a function from $[U]$ to $[2n]$. We say that f is *perfect* on S if it is injective with respect to S, namely, for any two different integers x_1, x_2 of S, $f(x_1) \neq f(x_2)$. Furthermore, f is *stable* if, for each $x \in S$, $f(x)$ remains the same until x is deleted from S.

Lemma 1 ([8]). *Let U be a positive integer at most 2^{wB-1}. For any $S \subseteq [U]$, we can store a stable perfect function f of S using $O(1 + \frac{n}{wB} \lg\lg \frac{U}{n})$ blocks, where $n = |S|$. For any $x \in [U]$, $f(x)$ can be computed in constant I/Os. If an integer is inserted or deleted in S, the representation of f can be updated in constant I/Os expected.*

We will also need the next fact about the string B-tree:

Lemma 2 ([10]). *Let D be a set of t strings (each with an integer id), and n be their total length. We can store D in a string B-tree of $O(1+\frac{t}{B}+\frac{n}{wB})$ space such that, given a string q of length m, using $O(\lg_B t + \frac{m}{wB})$ I/Os, we can report the id of q in D or declare the absence of q in D. A string of length l can be inserted and deleted in D with $O(\lg_B t + \frac{l}{wB})$ I/Os.*

2.2 A New Structure

We consider a slightly more general version of exact matching. Suppose that each string $s \in D$ carries an arbitrary *information field* that occupies $O(1)$ words. We want to support a *probe operation*: given a string q, decide whether $q \in D$ and if so, return the information field of q. The rest of the subsection serves as a proof for:

Theorem 1. *A set D of strings (each with an integer id) can be stored in a structure of $O(1 + \frac{t}{B} + \frac{n}{wB})$ blocks, where t is the number of strings in D and n is their total length, such that a probe operation can be performed in $O(1 + \frac{m}{wB})$ I/Os, where m is the length of the query string. To insert/delete a string with length l in D, the structure can be updated in $O(1 + \frac{l}{wB})$ expected I/Os.*

We say that a string is *short* if its length is at most $wB - 2$; otherwise, it is *long*. The two types of strings are processed separately.

Short Strings. Suppose that D has only short strings. We maintain a stable perfect function f on D by interpreting each string in D as an integer in $[2^{wB-2}]$. By Lemma 1, this demands $O(1 + \frac{t}{wB} \lg(wB))$ space, which is $O(1 + \frac{t}{B})$ because $w = \Omega(\lg B)$. Let I be an array of size $2t$ for storing information fields. For each string $s \in D$, its information field is stored in $I[f(s)]$.

To store the strings of D, we divide $[2t]$ into $\lceil 2t/B \rceil$ disjoint intervals of length B except possibly the last one. Refer to each interval as a *chunk*. All the at most B strings of D mapped to the same chunk by f are managed by a string B-tree, each of which occupies $O(1 + \frac{n'}{wB})$ space where n' is the total length of the strings it manages (Lemma 2). All the string B-trees use $O(1 + \frac{t}{B} + \frac{n}{wB})$ space in total.

To perform a probe with a short string q, we search the string B-tree of chunk $f(q)$ in $O(\lg_B B + \frac{|q|}{wB}) = O(1)$ I/Os (Lemma 2). If found, we return $I[f(q)]$ with another $O(1)$ I/Os; otherwise, q is not in D.

To insert a short string s in D, we first calculate $f(s)$ and update $I(f(s))$ in $O(1)$ expected I/Os (Lemma 1). Then, insert s in the string B-tree responsible for the chunk covering $I(f(s))$. By Lemma 2, the insertion cost is $O(\lg_B B + \frac{wB}{wB}) = O(1)$ I/Os. Using standard rebuilding techniques [14], we can resize I in $O(1)$ worst-case time per insertion. A deletion can be handled similarly.

Long Strings. Now, consider that D has only long strings. We sometimes regard a long string s of length l as a *blocked string* $\tilde{s}$ of length $\lceil l/(wB-2) \rceil$, where each character of $\tilde{s}$ comes from an alphabet $\tilde{\Sigma}$ of size 2^{wB-2}. Specifically, if we chop s into blocks of size $wB-2$ (possibly except the last block), character $\tilde{s}_j$ corresponds to the j-th block, where $1 \leq j \leq \lceil l/(wB-2) \rceil$. Denote by $\tilde{D}$ the set of blocked strings obtained from D.

Let T be a trie built on $\tilde{D}$. For an internal node u in T, let $child(u)$ be the set of its child nodes (note that $|child(u)|$ can be up to 2^{wB-2}). As each node in $child(u)$ is a character in $\tilde{\Sigma}$, it can be regarded as a short string. To allow efficient navigation, we create an aforementioned structure (for short strings) on $child(u)$ such that given any character $\tilde{\sigma} \in \tilde{\Sigma}$, we can tell in constant time whether $\tilde{\sigma} \in child(u)$ and if so, also the address of the child $\tilde{\sigma}$. For each string $s \in D$, its information field is stored at the node of T whose root-to-leaf path corresponds to $\tilde{s}$.

The short-string structure on an internal node u of T consumes $O(1 + \frac{|child(u)|}{B} + \frac{|child(u)|wB}{wB}) = O(|child(u)|)$ space, i.e., $O(1)$ blocks per child. Hence, the entire space usage of T is asymptotically the number of nodes in T. Since each $s \in D$ necessitates $O(|s|/(wB))$ nodes, T has $O(n/(wB))$ nodes. It thus follows that our structure occupies $O(n/(wB))$ space overall.

To perform a probe, we simply search T with $\tilde{q}$. The cost is clearly $O(|\tilde{q}|) = O(|q|/(wB))$. To insert a long string s of length l, we first obtain its blocked string $\tilde{s}$ in $O(l/(wB)) = O(|\tilde{s}|)$ I/Os. Then, insert $\tilde{s}$ in T using $O(|\tilde{s}|)$ I/Os. Specifically, at the node of $\tilde{s}_i$ ($i \geq 1$) in T, we can identify the node of $\tilde{s}_{i+1}$ in constant time (by exploiting the short-string structure on $\tilde{s}_i$), or create it in constant time if it does not exist. The deletion algorithm is analogous.

3 One-Error Dictionary Search

This section serves as a proof for our main result:

Theorem 2. *A set D of strings (each with an integer id) can be stored in a structure of $O(n/B)$ blocks, where n is the total length of all the strings in D, such that a 1-error dictionary search query can be answered in $O(1+\frac{m}{wB}+\frac{k}{B})$ I/Os, where m is the length of the query string, and k is the number of qualifying strings. To insert/delete a string with length l in D, the structure can be updated in $O(l)$ expected I/Os.*

We will focus on finding those strings $s \in d$ with 1 edit distance from a query string q (the string with 0 edit distance, if exists, can be found by exact matching). There are only 3 possibilities: an insertion, deletion, or a replacement of a character turns s to q—in these cases s is said to be an *insertion*, *deletion* and *replacement match*, respectively. We will concentrate on insertion matches in Section 3.1-3.3. The other types of matches can be reported using similar techniques, which are omitted from this extended abstract due to the space limit.

3.1 Signature Edits

We define an *insertion match* s to be an *appending match* if we can add a character at the end of s to turn it into q. Otherwise, s is a *non-appending match*. For instance, given $q = 11110$, $s = 1111$ is an appending match, while $s = 1110$ is a non-appending match. We will focus on reporting non-appending matches, because there is at most one appending match, as can be found by exact matching after trimming the last character of q.

Consider an insertion that turns a non-appending match s to q. Denote the insertion as (s, i, c) if it adds character c before s_i for some $i \in [1, |s|]$. Recall that there can be multiple such insertions. We define (s, i, c) to be a *signature insertion* if $c \neq s_i$. It turns out that only one insertion can be signature:

Lemma 3. *Every non-appending match s has a unique signature insertion that turns s into q.*

Proof. We first prove that s can be turned into q by a signature insertion. Since s is a non-appending match, there exists an $i \in [1, |s|]$ such that (s, i, c) turns s into q. If $c \neq s_i$, then this insertion is signature. Otherwise, let $j > i$ be the smallest integer such that $s_{j-1} \neq s_j$. Such j definitely exists; otherwise, we can append c to s to turn it into q, contradicting the fact that s is non-appending. Thus, (s, j, c) is a signature insertion.

We now prove that there is only one signature insertion. Assume that s has two: (s, i_1, c_1) and (s, i_2, c_2) with $i_1 < i_2$, both of which convert s to q. From (s, i_1, c_1), we know $q_{i_1} = c_1 \neq s_{i_1}$. However, from (s, i_2, c_2), we know that $q_{1..(i_2-1)} = s_{1..(i_2-1)}$, implying that $q_{i_1} = s_{i_1}$, giving a contradiction. □

3.2 Short Strings

In this subsection, we describe a structure for a dictionary D that has only short strings (i.e., length at most $wB - 2$). Consider a string $s \in D$. Obviously, s has $|s|$ signature insertions: for each $i \in [1, |s|]$, the i-th signature insertion adds the opposite of s_i before s_i. Let $\mathcal{N}^+(s)$ be the set of strings obtained by applying those signature insertions on s, respectively. We have:

Lemma 4. *$|\mathcal{N}^+(s)| = |s|$ and $\mathcal{N}^+(s)$ is exactly the set of strings for which s is a non-appending match.*

Proof. $|\mathcal{N}^+(s)| = |s|$ is because any two signature insertions turn s into different strings. It is obvious that s is a non-appending match of every string in $\mathcal{N}^+(s)$. Finally, by Lemma 3, every string of which s is non-appending match belongs to $\mathcal{N}^+(s)$. □

Structure. Let us first make a *disjoint-neighbor assumption*: for any two $s \neq s'$ in D, $\mathcal{N}^+(s)$ and $\mathcal{N}^+(s')$ are disjoint. Let D^+ be the union of $\mathcal{N}^+(s)$ of all $s \in D$. By Lemma 4, D^+ can have at most $\sum_{s \in D} |s| = n$ strings. Clearly, a query string q has a non-appending match in D if and only if $q \in D^+$. Next we utilize this fact to find the non-appending match of q efficiently (there is only one match under the disjoint-neighbor assumption).

We maintain a stable perfect function h^+ on D^+ using Lemma 1 (notice that all strings of D^+ have length at most $wB - 1$). The representation of h^+ occupies $O(1 + \frac{n}{wB} \lg(wB)) = O(n/B)$ space. Recall that h^+ maps each string $s^+ \in D^+$ to a distinct integer in $[2n]$. We create an array Δ^+ of size $2n$ to record signature insertions. Specifically, for each $s^+ \in D^+$, $\Delta^+[h^+(s^+)]$ stores a pair $(id(s), i)$ if (s, i, c) is the signature insertion generating s^+, where $id(s)$ gives the id of s. Each cell of Δ^+ can be stored in a word. Overall, Δ^+ uses $O(n/B)$ blocks. Finally, we build a structure of Theorem 1 on D so that exact matching in D can be done efficiently. The total space of our structure is therefore $O(n/B)$.

Given a string q, we search for its non-appending match as follows. First, locate cell $\Delta^+[h^+(q)]$ in constant I/Os. If nothing exists in the cell, $q \notin D^+$ and hence, has no non-appending match. Instead, suppose that $\Delta^+[h^+(q)]$ contains a pair $(id(s), i)$. We obtain a string q' from q by deleting q_i, and perform exact matching with q' in D using constant I/Os. If s is found, we return its id; otherwise, q has no non-appending match.[2] The total query time is $O(1)$.

Update. We discuss only insertions because a reverse procedure supports deletions. To insert a (short) string s, we first insert it in the exact matching structure (on D) using $O(1)$ expected I/Os (Theorem 1). Then, keeping s memory-resident, we can generate each string $s^+ \in \mathcal{N}^+(s)$ in memory. For each s^+ with signature insertion (s, i, c), calculate $h^+(s^+)$ and store $(id(s), i)$ at $\Delta^+[h^+(s^+)]$ in constant expected I/Os (Lemma 1). Therefore, the total insertion cost is $O(|\mathcal{N}^+(s)|) = O(l)$ I/Os expected.

Eliminating the Disjoint-Neighbor Assumption. We first need to solve a related problem we call *find-all-any*. Let r be an integer satisfying $0 \leq r \leq w$, and define

[2] The exact matching with q' is for detecting the scenario where $q \neq s^+$ but $h^+(q) = h^+(s^+)$.

$\phi = \lg(wB) + r$. Let $g \leq CwB/\phi$ for any constant $C > 0$, and $S_1, ..., S_g$ be g sets of integers in $[wB]$. Each integer in any S_i ($1 \leq i \leq g$) is associated with an arbitrary *information field* of r bits. We want to maintain a structure to support the following operations efficiently:

- given an $i \in [1, g]$, insert/delete an integer in S_i, as well as its information field.
- *find-all*(i): given an $i \in [1, g]$, return the information fields of all the integers in S_i.
- *find-any*(i): given an $i \in [1, g]$, return an arbitrary integer in S_i and its information field.

Lemma 5. *Let $L = \sum_{i=1}^{g} |S_i|$. There is a structure of $O(1 + \frac{\phi L}{wB})$ space supporting an insertion/deletion in $O(1)$ expected I/Os, find-any in $O(1)$ I/Os, and find-all in $O(1 + \frac{\phi |S_i|}{wB})$ I/Os.*

Proof. We say that S_i is *small* if $|S_i| \leq wB/(2\phi)$, and *big* if $|S_i| \geq wB/\phi$. S_i is neither small nor big when $wB/(2\phi) < |S_i| < wB/\phi$. We follow the invariant that all small sets are together managed by a B-tree U. In U, the integers (of the small sets) are first sorted by the sets they come from, and then by their values. In other words, each integer corresponds to a composite key (set-id, value), which fits in $O(\lg(wB))$ bits. The information field of an integer is stored in the same leaf node with the integer. Each leaf node contains $\Theta(wB/\phi)$ integers. It will be guaranteed that U manages $O((wB/\phi)^2)$ integers, and thus occupies $O(\frac{(wB/\phi)^2}{wB/\phi}) = O(wB)$ blocks. Hence, each block pointer within U can be stored in $O(\lg(wB))$ bits. We therefore can set the fanout of U to $\Theta(wB/\lg(wB))$ so that U has only constant levels.

Each big set S_i is stored in a *big structure*, which consists of a hash structure (e.g., [9]) and a linked list. The hash structure is created on the integers of S_i, whereas the linked list contains these integers (i.e., each integer has two copies: in the hash structure and linked list, respectively) as well as their information fields. The ordering in the linked list is not important, as long as each block accommodates $\Theta(wB/\phi)$ integers and their information fields. At all times, for each integer, we let its copies in the hash structure and the linked list keep pointers to each other. This allows us to reach the copy in the linked list in constant I/Os.

When S_i is neither big nor small, it can be stored either in U or a big structure. Hence, U can index $O(g(wB/\phi)) = O((wB/\phi)^2)$ integers.

Next, we explain how to insert/delete an integer x in S_i. First, if S_i is currently stored in U, we insert/delete x in U using $O(1)$ I/Os. Otherwise, update the hash structure and linked list on S_i in $O(1)$ expected I/Os. In the linked list, we sometimes need to split a block or merge two blocks to ensure $\Theta(wB/\phi)$ integers per block. Each split/merge incurs $O(wB/\phi)$ I/Os to correct the pointers between the hash structure and linked list. By standard techniques (e.g., as in updating a B-tree), this happens only after $\Omega(wB/\phi)$ updates, so that each update is charged only constant I/Os for the split/merge. S_i may become big when it is in U, or conversely, become small when it is in a big structure. In either case, we perform an *overhaul* to move the entire S_i into a big structure or U respectively using $O(wB/\phi)$ I/Os. As at least $wB/(2\phi)$ updates must have occurred between two overhauls, each update is charged only constant I/Os

for an overhaul. Therefore, overall an update requires constant expected I/Os amortized. The amortization can be easily removed using lazy rebuilding techniques of [14] (see also [2]).

The other two operations can be supported efficiently. To perform $\mathit{find}\text{-}\mathit{all}(i)$, we simply search U or scan the linked list on S_i, depending on where S_i is stored. In either case, the cost is $O(1 + \frac{\phi|S_i|}{wB})$, recalling that U has $O(1)$ levels. Finally, to perform $\mathit{find}\text{-}\mathit{any}(i)$, we simply return the first integer of S_i in U or its linked list. The cost is clearly $O(1)$. □

Next, we remove the disjoint-neighbor assumption. Define D^+ again as the union of $\mathcal{N}^+(s)$, i.e., having discarded all duplicates. Divide $[2n]$ into $\lceil 2n/B \rceil$ intervals (a.k.a. *chunks*) of size B. Consider a string $s^+ \in D^+$, which may now belong to the $\mathcal{N}^+(s)$ of several $s \in D$. In other words, multiple signature insertions may have generated s^+. Let $E^+(s^+)$ be the set of those signature insertions.

Recall that every signature insertion has the form (s, i, c). A crucial observation is that all signature insertions in $E^+(s^+)$ differ in their values of i. To prove this, first notice that no two signature insertions in $E^+(s^+)$ can have come from the same s, according to Lemma 3. Next, assume that $E^+(s^+)$ had signature insertions (s_1, i, c_1) and (s_2, i, c_2) with $s_1 \neq s_2$. Notice that c_1 must be identical to c_2, because both of them were equal to $(s^+)_i$. It thus follows that s_1 had to be the same as s_2, giving a contradiction.

Because of the previous observation, we will sometimes regard $E^+(s^+)$ as a set of integers: $\{i \mid (s, i, c) \in E^+(s^+)\}$. Further, we associate i with an information field $id(s)$ where s is the unique string such that $(s, i, c) \in E^+(s^+)$. Clearly, i and $id(s)$ can be stored in $\lg(wB)$ and w bits, respectively.

For each chunk, we store at most B sets of integers, namely, a set $E^+(s^+)$ for every s^+ whose $h^+(s^+)$ is covered by the chunk. We index these sets with a find-all-any structure of Lemma 5. To see that this is possible, set $r = w$ and hence $\phi = \lg(wB)+w$, in which case a find-all-any structure can manage up to $g = CwB/\phi = C\frac{wB}{\lg(wB)+w}$ sets where $C > 0$ can be any constant. In other words, it can indeed be used to manage B sets because $B = O(\frac{wB}{\lg(wB)+w})$ (applying $w = \Omega(\lg B)$).

Therefore, by Lemma 5, the find-all-any structure of a chunk uses $O(1 + \frac{L}{B})$ space where L is the total size of those sets. As there are $O(n/B)$ chunks, the structures of all chunks require $O(n/B)$ space altogether. We also build all the structures as were necessary when the disjoint-neighbor assumption was made. The only difference is that, at each cell $\Delta^+[h^+(s^+)]$, we store an arbitrary pair $(id(s), i)$ where $(s, i, c) \in E^+(s^+)$.

A query is answered as before, except that after finding the match of q' in D, we report all the string ids in $E^+(h^+(q))$ using a $\mathit{find}\text{-}\mathit{all}$ operation which performs $O(1+\frac{k}{B})$ I/Os by Lemma 5. This is correct because, by definition, all the signature insertions in $E^+(h^+(q))$ generate exactly q. To insert a length-l string s, we proceed as before, but also perform an insertion in the find-all-any structure of a relevant chunk for each signature insertion (s, i, c). The algorithm of deleting a string s is also the same as before, except that if s happens to be the string whose id is in cell $\Delta^+[h^+(s^+)]$, we perform a $\mathit{find}\text{-}\mathit{any}$ to extract another element from $E^+(s^+)$ to fill in the cell. By Lemma 5, the above changes incur only constant expected I/Os per signature insertion, and hence, $O(l)$ time in total.

Now we have arrived at:

Lemma 6. *We can store D in a structure of $O(n/B)$ blocks such that all insertion matches of a short query string with length at most wB can be performed in $O(1 + \frac{k}{B})$ I/Os, where k is the number of qualifying strings. To insert/delete a short string with length l in D, the structure can be updated in $O(l)$ expected I/Os.*

Remark. When Σ is not binary, the space consumption of our structure increases by a factor linear to $|\Sigma|$, and hence, remains $O(n/B)$ when $|\Sigma| = O(1)$. The major change is that, for each string s and each $i \in [1, |s|]$, there are $|\Sigma| - 1$ signature insertions, namely, (s, i, c) for every possible $c \neq s_i$ in Σ. Accordingly, $\mathcal{N}^+(s)$ has size $|s|(|\Sigma| - 1) = O(|s|)$. Furthermore, the length of a short string should be defined instead as $C \cdot wB$ for some appropriate $0 < C < 1$, making sure that a short string fits in one block (each character of Σ is now encoded by $\log_2 |\Sigma| = \Theta(1)$ bits). The other changes are obvious and therefore omitted.

3.3 Long Strings

This section explains the structure for *long* strings, each of which has length at least wB. We divide these strings by their lengths, and manage each group of strings with the same length separately. Our discussion below concentrates on a particular length l. For convenience, let $\rho = wB - 2$, and we will assume that $l = \lambda\rho - 1$ for some integer $\lambda \geq 2$. If not, we pad enough (at most $\rho - 1$) 0's at the end of each string to make the property hold. The padding can increase the space, query, and update costs of our structure by no more than a constant factor, as will be clear shortly.

Searching the Blocked Strings. Following the notation in Section 2.2, given a long string s, we denote its corresponding blocked string as $\tilde{s}$. Recall that $\tilde{s}$ is obtained by chopping s into blocks of size ρ. We may regard each character of $\tilde{s}$ as being in an alphabet $\tilde{\Sigma}$ of size 2^ρ, or equivalently, as a binary string of length at most ρ, whichever is more convenient. When the second interpretation is adopted, s_i is a bit in $\tilde{s}_j$ where $i \in [1, |s|]$ and $j = \lceil i/\rho \rceil$.

We denote by $\overline{s}$ as the reverse string of s (e.g., if $s = 10010$ then $\overline{s} = 01001$). Just like s, $\overline{s}$ also has its blocked string $\tilde{\overline{s}}$.

Consider a string q of length $l + 1$. Let s be a non-appending match of q. Note that both $\tilde{s}$ and $\tilde{q}$ have length λ (as long as $\rho \geq 2$). According to Lemma 3, there exists a unique signature insertion (s, i, c) that converts s to q. Let $j = \lceil i/\rho \rceil$. Depending on the value of j, we define a binary string $q^\star(j)$:

- If $j = \lambda$, then $q^\star(j) = \tilde{q}_\lambda$. In this case, $|q^\star(j)| = \rho$.
- Otherwise, $q^\star(j)$ is the string obtained by appending the first character of $\tilde{q}_{j+1}$ to $\tilde{q}_j$ (viewing both $\tilde{q}_j$ and $\tilde{q}_{j+1}$ in binary form). In this case, $|q^\star(j)| = \rho + 1$.

For example, suppose $\rho = 3$ and $q = 110100$; thus, if $j = 1$, then $q^\star(j) = 1101$, whereas if $j = 2$ then $q^\star(j) = 100$. The next lemma gives a crucial observation.

Lemma 7. *Let s be a non-appending match of q with signature insertion (s, i, c). Define $j = \lceil i/\rho \rceil$. All the following are true:*

(i) $\tilde{s}_{1..(j-1)} = \tilde{q}_{1..(j-1)}$
(ii) $\tilde{\overline{s}}_{1..(\lambda-j)} = \tilde{\overline{q}}_{1..(\lambda-j)}$
(iii) $\tilde{s}_j$ *is a non-appending match of* $q^\star(j)$.

Proof. By the definition of i, it holds that $s_{1..(i-1)} = q_{1..(i-1)}$, and $s_{i..l} = q_{(i+1)..(l+1)}$. The latter suggests $\overline{s}_{1..(l-i+1)} = \overline{q}_{1..(l-i+1)}$.

(i) Notice that $\rho(j-1) \leq i-1$. Thus, $\tilde{s}_{1..(j-1)} = \tilde{q}_{1..(j-1)}$ holds because they (in binary form) are prefixes of the same length of $s_{1..(i-1)}$ and $q_{1..(i-1)}$, respectively.

(ii) First observe that $\rho(\lambda - j) \leq l - i + 1$: since $\rho\lambda = l + 1$, this is equivalent to showing $i \leq j\rho$, which is true by the definition of j. Hence, $\tilde{\overline{s}}_{1..(\lambda-j)} = \tilde{\overline{q}}_{1..(\lambda-j)}$ holds because they are prefixes of the same length of $\overline{s}_{1..(l-i+1)}$ and $\overline{q}_{1..(l-i+1)}$, respectively.

(iii) We focus on $j < \lambda$ because the case $j = \lambda$ is obvious. We will abbreviate $q^\star(j)$ simply as $q^\star$. Since $\tilde{s}_j$ is clearly an insertion match of $q^\star$, what remains to prove is that it is *not* an appending match. The remainder of the proof will regard $\tilde{s}_j$ as a binary string. Let $i' = i - \rho(j-1)$, namely, $(\tilde{s}_j)_{i'}$ is the same character as s_i.

Now assume that $\tilde{s}_j$ was an appending match of $q^\star$. Let c' be the last character of $q^\star$. It thus follows that $\tilde{s}_j : c' = q^\star$ (where ":" means concatenation), implying $(\tilde{s}_j)_{i'} = q^\star_{i'}$. On the other hand, the fact that (s, i, c) turns s to q implies $c = q^\star_{i'}$. It thus follows that $c = (\tilde{s}_j)_{i'} = s_i$, violating the definition of (s, i, c). □

Structure. Let $\tilde{D}(l)$ be the set of blocked strings $\tilde{s}$ of all $s \in D$ with length l. Likewise, let $\tilde{\overline{D}}(l)$ be the set of blocked strings $\tilde{\overline{s}}$ of all such s. Regarding each blocked string as consisting of characters from $\tilde{\Sigma}$ (of size 2^ρ), we build a trie T on $\tilde{D}(l)$ and another trie $\overline{T}$ on $\tilde{\overline{D}}(l)$.

Consider a blocked string $\tilde{s} \in \tilde{D}(l)$, and the corresponding $\tilde{\overline{s}} \in \tilde{\overline{D}}(l)$. Recall that both $\tilde{s}$ and $\tilde{\overline{s}}$ have length λ. For each $j \in [1, \lambda]$, we do the following. First, identify the node u in T corresponding to $\tilde{s}_{1..(j-1)}$, and the node $\overline{u}$ in $\overline{T}$ corresponding to $\tilde{\overline{s}}_{1..(\lambda-j)}$. Insert $\tilde{s}_j$ to a set $S(u, \overline{u})$, which is therefore a set of short strings (when viewed in binary form). We build a structure of Lemma 6 on $S(u, \overline{u})$; let us represent this structure as $P^+(u, \overline{u})$, referred to as a P^+*-structure*.

Finally, we need a structure that, given any $(u, \overline{u})$, returns the beginning address of $P^+(u, \overline{u})$ in constant time. For this purpose, it suffices to maintain a standard dynamic hash structure (e.g., [9]) on the pair of addresses of u and $\overline{u}$; we will refer to it as the P^+*-lookup structure*.

Update. The algorithm for inserting a string of length l follows directly from the above description. The cost is $O(\frac{l}{wB} \cdot wB) = O(l)$ expected by Lemma 6 and our earlier discussion on maintaining a trie on blocked strings. Deletion is analogous.

Query. Given a string q of length $l + 1$, we find its non-appending matches as follows. For each $j \in [1, \lambda]$, we identify the node u in T corresponding to $\tilde{q}_{1..(j-1)}$, and the node $\overline{u}$ in $\overline{T}$ corresponding to $\tilde{\overline{q}}_{1..(\lambda-j)}$. Report all the non-appending matches of $q^\star(j)$ in $S(u, \overline{u})$. The algorithm's correctness is established by the next lemma.

Lemma 8. *Every non-appending match of* q *is reported exactly once.*

Proof. Lemma 7 shows that every non-appending match of q must be reported at least once. Next we prove that no non-appending match s can be reported twice. For this

purpose, let (s, i, c) be the signature insertion that converts s to q, and let $j^\star = \lceil i/\rho \rceil$. We will prove that, for any $j' \neq j^\star$, our algorithm does *not* report s when $j = j'$. In other words, s is reported only when $j = j^\star$.

Suppose on the contrary that s was reported at j', i.e., $\tilde{s}_{j'}$ is a non-appending match of $q^\star(j')$. We can easily rule out the possibility of $j' < j^\star$. This is because, by the definition of i and $j^\star$, $\tilde{s}_{j'}$ must be equivalent to $\tilde{q}_{j'}$ for every $j' < j^\star$, implying that $\tilde{s}_{j'}$ is an appending match of $q^\star(j')$.

Consider $j' > j^\star$. By the definition of (s, i, c), we know $q_i = c$. Since s is reported by our algorithm when $j = j'$, $\tilde{s}_{j'-1}$ and $\tilde{q}_{j'-1}$ must correspond to the same node in trie T, implying that $s_{1..(\rho(j'-1))} = q_{1..(\rho(j'-1))}$. As $i \leq \rho j^\star \leq \rho(j'-1)$, we have $s_i = q_i$, namely, $s_i = c$, thus violating the definition of (s, i, c). □

Analysis. To bound the space of our structure, let t' be the number of length-l strings in D, and $n' = t'l$ be the total length of these strings. The analysis in Section 2.2 shows that each of T and $\tilde{T}$ uses $O(\frac{t'}{B} + \frac{n'}{wB}) = O(\frac{n'}{wB})$ space.

Next we discuss the space of the P^+-structures. Consider a pair $(u, \overline{u})$ whose $S(u, \overline{u})$ is non-empty. Note that there are at most $O(n'/(wB))$ such $(u, \overline{u})$ because a string $s \in D$ of length l can necessitate $O(l/(wB))$ such pairs. Let $t(u, \overline{u})$ be the number of strings in $S(u, \overline{u})$, and $n(u, \overline{u})$ be their total length. By Lemma 6, $P^+(u, \overline{u})$ occupies $O(n(u, \overline{u})/B)$ blocks. Therefore, the space of all the P^+-structures is at most

$$\sum_{(u,\overline{u}) \text{ with non empty } S(u,\overline{u})} O\left(1 + \frac{n(u,\overline{u})}{B}\right)$$

$$= O\left(\frac{n'}{wB} + \frac{n'}{B}\right) = O(n'/B).$$

Finally, the space of the P^+-lookup structure is linear to the number of non-empty sets $S(u, \overline{u})$; as there are $O(n'/(wB))$ non-empty sets, the P^+-lookup structure occupies $O(n'/(wB^2))$ space. A query searches $O(l/(wB))$ P^+-structures. Combining Lemmas 6 and 8 proves that the query time is $O(\frac{l}{wB} + \frac{k}{B})$ overall.

We thus have established:

Lemma 9. *We can store D in a structure of $O(n/B)$ blocks such that all insertion matches of a length-m query string can be found in $O(1 + \frac{m}{wB} + \frac{k}{B})$ I/Os, where k is the number of qualifying strings. To insert/delete a long string with length l in D, the structure can be updated in $O(l)$ expected I/Os.*

Remark. The double-trie idea is due to [5], but the challenge in our contexts is to make the idea work on blocked strings. For this purpose, it is crucial to derive Lemmas 7 and 8, both of which rely on the notion of signature edits and the separation of non-appending matches. The two lemmas, notion, and separation are where our contributions lie.

Acknowledgements

Chin-Wan Chung was supported in part by Defense Acquisition Program Administration and Agency for Defense Development under the contract UD140022PD, Korea. Yufei Tao was supported in part by projects GRF 4165/11, 4164/12, and 4168/13 from HKRGC. Wei Wang was partly funded by ARC DP130103401 and DP130103405.

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *CACM*, 31(9):1116–1127, 1988.
2. L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM J. of Comp.*, 32(6):1488–1508, 2003.
3. D. Belazzougui. Faster and space-optimal edit distance "1" dictionary. In *Annual Symp. on Combinatorial Pattern Matching*, pages 154–167, 2009.
4. D. Belazzougui and R. Venturini. Compressed string dictionary look-up with edit distance one. In *Annual Symp. on Combinatorial Pattern Matching*, 2012.
5. G. S. Brodal and L. Gasieniec. Approximate dictionary queries. In *Annual Symp. on Combinatorial Pattern Matching*, pages 65–74, 1996.
6. H.-L. Chan, T. W. Lam, W.-K. Sung, S.-L. Tam, and S.-S. Wong. A linear size index for approximate pattern matching. In *Annual Symp. on Combinatorial Pattern Matching*, pages 49–59, 2006.
7. R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *STOC*, pages 91–100, 2004.
8. E. D. Demaine, F. Meyer auf der Heide, R. Pagh, and M. Patrascu. De dictionariis dynamicis pauco spatio utentibus (*lat.* on dynamic dictionaries using little space). In *LATIN*, pages 349–361, 2006.
9. M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. of Comp.*, 23(4):738–761, 1994.
10. P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *JACM*, 46(2):236–280, 1999.
11. W.-K. Hon, T. W. Lam, R. Shah, S.-L. Tam, and J. S. Vitter. Cache-oblivious index for approximate string matching. *Theoretical Computer Science*, 412(29):3579–3588, 2011.
12. K. Kukich. Techniques for automatically correcting words in text. *ACM Comp. Surv.*, 24(4):377–439, 1992.
13. G. Navarro. A guided tour to approximate string matching. *ACM Comp. Surv.*, 33(1):31–88, 2001.
14. M. H. Overmars. *The Design of Dynamic Data Structures*. Springer-Verlag, 1987.
15. D. Tsur. Fast index for approximate string matching. *Journal of Discrete Algorithms*, 8(4):339–345, 2010.