Title	Cache-oblivious index for approximate string matching
Author(s)	Hon, WK; Lam, TW; Shah, R; Tam, SL; Vitter, JS
Citation	Theoretical Computer Science, 2011, v. 412 n. 29, p. 3579-3588
Issued Date	2011
URL	http://hdl.handle.net/10722/140789
Rights	Creative Commons: Attribution 3.0 Hong Kong License

Cache-Oblivious Index for Approximate String Matching*

Wing-Kai Hon† Tak-Wah Lam‡ Rahul Shah§
Siu-Lung Tam‡ Jeffrey Scott Vitter¶

Abstract

This paper revisits the problem of indexing a text for approximate string matching. Specifically, given a text T of length n and a positive integer k, we want to construct an index of T such that for any input pattern P, we can find all its k-error matches in T efficiently. This problem is well-studied in the internal-memory setting. Here, we extend some of these recent results to external-memory solutions, which are also cache-oblivious. Our first index occupies $O((n \log^k n)/B)$ disk pages and finds all k-error matches with $O((|P|+occ)/B+\log^k n \log\log_B n)$ I/Os, where B denotes the number of words in a disk page. To the best of our knowledge, this index is the first external-memory data structure that does not require $\Omega(|P|+occ+\text{poly}(\log n))$ I/Os. The second index reduces the space to $O((n \log n)/B)$ disk pages, and the I/O complexity is $O((|P|+occ)/B+\log^{k(k+1)} n \log\log n)$.

1 Introduction

Recent years have witnessed a huge growth in the amount of data produced in various disciplines. Well-known examples include DNA sequences, financial time-series, sensor data, and web files. Due to the limited capacity of the main memory, traditional data structures and algorithms that perform optimally in main memory become inadequate in many applications. For example, suffix tree [19, 25] is an efficient data structure for indexing a text T for exact pattern matching; given a pattern P, it takes $O(|P|+occ)$ time to report all occurrences of P in T, where occ denotes the number of occurrences. However, if we apply a suffix tree to index the human genome, which has 3 billion characters, at least 64G bytes of main memory would be needed.

To deal with these massive data sets, a natural way is to exploit the external memory as an extension of main memory. In this paradigm of computation, data can be transferred in and out of main memory through an I/O operation. In practice, an I/O operation takes much more time than an operation in main memory. Therefore, it is important to minimize the number of I/Os.

Aggarwal and Vitter [2] proposed a widely accepted *two-level I/O-model* for analyzing the I/O complexity. In their model, the memory hierarchy consists of a main memory of M words and an external memory of unlimited space. Data reside in the external memory initially (as they exceed the capacity of the main memory), and computations can be performed only when the required data are present in the main memory. With one I/O operation, a disk page with B contiguous words can be read from the external memory to the main memory, or B words from the main memory can be written to a disk page in the external memory; the I/O complexity of an algorithm counts only the number of I/O operations involved. To reduce the I/O complexity, an algorithm must be able to exploit the locality

*A preliminary version appears in *Proceedings of 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 40–51, 2007. This research is supported in part by Taiwan NSC Grant 96-2221-E-007-082-MY3 (W.K. Hon), Hong Kong RGC Grant 7140/06E (T.W. Lam), and US NSF Grant CCF–0621457 (R. Shah and J. S. Vitter).

†Department of Computer Science, National Tsing Hua University, Taiwan. Email: `wkhon@cs.nthu.edu.tw`

‡Department of Computer Science, The University of Hong Kong, Hong Kong. Email: `{twlam,sltam}@cs.hku.hk`

§Department of Computer Science, Louisiana State University, Louisiana, US. Email: `rahul@csc.lsu.edu`

¶Department of Electrical Engineering and Computer Science, Kansas University, Kansas, US. Email: `jsv@ku.edu`

of data in external memory. For instance, under this model, sorting a set of n numbers can be done in $O\left((\frac{n}{B}\log\frac{n}{B})/\log(\frac{M}{B})\right)$ I/Os, and this bound is proven to be optimal. (See [24] for more algorithms and data structures in the two-level I/O model.)

Later, Frigo et al. [15] introduced the notion of *cache-obliviousness*, in which we do not know the values of M or B when we design data structures and algorithms for the external-memory setting; instead, we require the data structures and the algorithms to work for any given M and B. Furthermore, we would like to match the I/O complexity when M and B are known in advance. Thus, cache-obliviousness implies that the data structures and the algorithms will readily work well under different machines, without the need of fine tuning the algorithms (or recompilation) or rebuilding the data structures. Many optimal cache-oblivious algorithms and data structures are proposed over the recent years, including algorithms for sorting [20] and matrix transposition [20], and data structures like priority queues [8] and B-trees [7].

For string matching, the recent data structure proposed by Brodal and Fagerberg [9] can index a text T in $O(n/B)$ disk pages[1] and find all occurrences of a given pattern P in T in $O((|P|+occ)/B+\log_B n)$ I/Os. This index works in a cache-oblivious manner, improving on the String-B tree, which is an earlier work by Ferragina and Grossi [14] that achieves the same space and I/O bounds but requires the knowledge of B to operate.[2] In this paper, we consider the *approximate string matching* problem defined as follows:

> Given a text T of length n and a fixed positive integer k, construct an index on T such that for any input pattern P, we can find all *k-error matches* of P in T, where a k-error match of P is a string that can be transformed to P using at most k character insertions, deletions, or replacements.[3]

The above problem is well-studied in the internal-memory setting [21, 12, 3, 10, 17, 11]. Recently, Cole et al. [13] proposed an index that occupies $O(n\log^k n)$ words of space, and can find all k-error matches of a pattern P in $O(|P|+\log^k n\log\log n+occ)$ time. This is the first solution with time complexity linear to $|P|$; in contrast, the time complexity of other existing solutions depends on $|P|^k$. Chan et al. [11] later gave another index that requires only $O(n)$ space, and the time complexity increases to $O(|P|+\log^{k(k+1)} n\log\log n+occ)$. In this paper, we extend these two results to the external-memory setting. In addition, our solution is cache-oblivious.

The main difficulty in extending Cole et al.'s index to the external-memory setting lies in answering the longest common prefix (LCP) query for an arbitrary suffix of a pattern P using a few number of I/Os. More specifically, given a suffix P_i, we want to find the longest substring of T that is a prefix of P_i. In the internal-memory setting, we take advantage of the suffix links in the suffix tree of T to compute the answers of all possible LCP queries in $O(|P|)$ time (there are $|P|$ such queries). In the external-memory setting, a naive implementation would require $\Omega(\min\{|P|^2/B,|P|\})$ I/Os to compute the answers of all LCP queries. To circumvent this bottleneck, we create a new notion called k-partitionable patterns. If a given pattern P is not k-partitionable, we can show that T contains no k-error match of P. If P is k-partitionable, we can process all its LCP queries efficiently in two phases: We first compute the answers of some "useful" LCP queries (using $O(|P|/B+k\log_B n)$ I/Os), which would then enable us to answer each other LCP query efficiently (using $O(\log\log_B n)$ I/Os). To support this idea, we devise an I/O-efficient *screening test* that checks whether P is k-partitionable; if P is k-partitionable, the screening test would also compute some useful LCP values as a by-product, which can then be utilized to answer the LCP query for an arbitrary P_i in $O(\log\log_B n)$ I/Os.

[1] Under the cache-oblivious model, the index occupies $O(n)$ contiguous words in the external memory. The value of B is arbitrary, which is considered only in the analysis.

[2] Recently Bender et al. [6] have devised the cache-oblivious string B-tree, which is for other pattern matching queries such as prefix matching and range searching.

[3] A k-error match should more precisely be termed as a k-edit-error match. We use the former one for the sake of brevity.

Together with other cache oblivious data structures (for supporting LCA, Y-Fast Trie and WLA), we are able to construct an index that finds all k-error matches using $O((|P|+occ)/B+\log^k n \log\log_B n)$ I/Os. The space of the index is $O((n\log^k n)/B)$ disk pages. To the best of our knowledge, this is the first external-memory data structure that does not require $\Omega(|P| + occ + \text{poly}(\log n))$ I/Os. Note that both Cole et al.'s index and our index can work even if the alphabet size is unbounded.

Recall that the internal-memory index by Chan et al. [11] occupies only $O(n)$ space. The reduction of space demands a more involved searching algorithm. In particular, they need the data structure of [10] to support a special query called Tree-Cross-Product. Again, we can 'externalize' this index. Here, the difficulties come in two parts: (i) computing the LCP values, and (ii) answering the Tree-Cross-Product queries. For (i), we will use the same approach as we externalize Cole et al.'s index. For (ii), there is no external memory counter-part for the data structure of [10]; instead, we reduce the Tree-Cross-Product query to a two-dimensional orthogonal range search query, the latter can be answered efficiently using an external-memory index based on the work in [1]. In this way, for any fixed $k \geq 2$, we can construct an index using $O((n\log n)/B)$ disk pages, which can find all k-error matches of P in $O((|P| + occ)/B + \log^{k(k+1)} n \log\log n)$ I/Os. Following [11], our second result assumes the alphabet size to be constant.

In Section 2, we give a survey of a few interesting queries that have efficient cache-oblivious solutions. We also introduce a novel cache-oblivious index for the weighted level ancestor (WLA) queries. Section 3 reviews Cole et al.'s internal memory index for k-error matching and discusses how to turn it into an external memory index. Section 4 defines the k-partitionable property, describes the screening test, and shows how to compute the answers of the LCP queries efficiently. Section 5 shows how to externalize Chan et al.'s index. We conclude the paper in Section 6.

2 Preliminaries

2.1 Suffix Tree, Suffix Array, and Inverse Suffix Array

Given a text $T[1..n]$, the substring $T[i..n]$ for any $i \in [1, n]$ is called a suffix of T. We assume that characters in T are drawn from an ordered alphabet which is of constant size, and $T[n] = \$$ is a distinct character that does not appear elsewhere in T. The *suffix tree* of T [19, 25] is a compact trie storing all suffixes of T. Each edge corresponds to a substring of T, which is called the *edge label*. For any node u, the concatenation of the edge labels along the path from the root to u is called the *path label* of u. There are n leaves in the suffix tree, with each leaf corresponding to a suffix of T. Each leaf stores the starting position in T of its corresponding suffix, which is called the *leaf label*. The children of an internal node are ordered by the lexicographical order of their edge labels.

The *suffix array* of T [18], denoted by SA, is an array of integers such that $SA[i]$ stores the starting position in T of the ith lexicographically smallest suffix. It is worth-mentioning that SA can be obtained by traversing the suffix tree in a left-to-right order and recording the leaf labels. Furthermore, the descendant leaves of each internal node u in the suffix tree correspond to a contiguous range in the suffix array, and we call this the SA range of u.

The *inverse suffix array*, denoted by SA^{-1}, is defined such that $SA^{-1}[i] = j$ if and only if $i = SA[j]$. When stored in the external memory, the space of both SA and SA^{-1} arrays take $O(n/B)$ disk pages, and each entry in the arrays can be reported in one I/O.

Suppose that we are given a pattern P, which appears at position i of T. That is, $T[i..i+|P|-1] = P$. Then, P must be a prefix of the suffix $T[i..n]$. Furthermore, each distinct occurrence of P in T corresponds to a distinct suffix of T sharing P as a prefix. Based on this observation, the following lemma from [18] shows a nice property about the suffix array.

Lemma 1. *Suppose that P is a pattern appearing in T. Then there exists $i \leq j$ such that $SA[i]$, $SA[i+1], \ldots, SA[j]$ are the starting positions of all suffixes sharing P as a prefix. In other words,*

$SA[i..j]$ lists all occurrences of P in T.

2.2 Cache-Oblivious String Dictionaries

Brodal and Fagerberg proposed an external-memory index for a text T of length n that supports efficient pattern matching query [9]. Their index takes $O(n/B)$ disk pages and it is cache-oblivious. For the pattern matching query, given any input pattern P, we can find all occurrences of P in T using $O((|P| + occ)/B + \log_B n)$ I/O operations.

In this paper, we are interested in answering a slightly more general query. Given a pattern P, let ℓ be the length of the longest prefix of P that appears in T. We want to find all suffixes of T that has $P[1..\ell]$ as a prefix (that is, all suffixes of T whose common prefix with P is the longest among the others). We denote Q to be the set of starting positions in T of all such suffixes. Note that Q occupies a contiguous region in SA, say $SA[i..j]$. We define the *LCP query* of P with respect to T, denoted by $LCP(P, T)$, to be a query which reports (i) the *SA range*, $[i, j]$, that corresponds to the SA region occupied by Q, and (ii) the *LCP length*, ℓ.

With very minor adaptation, Brodal and Fagerberg's index can be used to answer the LCP query efficiently, as stated in the following lemma.

Lemma 2. *We can construct a cache-oblivious index for a text T of length n, such that given any input pattern P, we can compute $LCP(P, T)$ in $O(|P|/B + \log_B n)$ I/O operations. The space of the index is $O(n/B)$ disk pages.*

2.3 LCA Index on Rooted Tree

For any two nodes u and v in a rooted tree, a *common ancestor* of u and v is a node that appears in both the path from u to the root and the path from v to the root; among all common ancestors of u and v, the one that is closest to u and v is called the *lowest common ancestor* of u and v, denoted by $LCA(u, v)$. The lemma below states the performance of an external-memory index for LCA queries, which follows directly from the results in [16, 5].

Lemma 3. *Given a rooted tree with n nodes, we can construct a cache-oblivious index of size $O(n/B)$ disk pages such that for any nodes u and v in the tree, $LCA(u, v)$ can be reported in $O(1)$ I/O operations.*

2.4 Cache-Oblivious Y-Fast Trie

Given a set X of x integers, the predecessor of r in X, denoted by $Pred(r, X)$, is the largest integer in X which is smaller than r. If the integers in X are chosen from $[1, n]$, the *Y-fast trie* on X [26] can find the predecessor of any input r in $O(\log \log n)$ time under the word RAM model;[4] the space occupancy is $O(x)$ words. In the external-memory setting, we can store the Y-fast trie easily using the van Emde Boas layout [7, 22, 23, 20], giving the following lemma.

Lemma 4. *Given a set X of x integers chosen from $[1, n]$, we can construct a cache-oblivious Y-fast trie such that $Pred(r, X)$ for any integer r can be answered using $O(\log \log_B n)$ I/O operations. The space of the Y-fast trie is $O(x/B)$ disk pages.*

2.5 Cache-Oblivious WLA Index

Let R be an edge-weighted rooted tree with n nodes, where the weight on each edge is an integer in $[1, W]$. We want to construct an index on R so that given any node u and any integer w, we can find the unique node v (if exists) with the following properties: (1) v is an ancestor u, (2) the sum of the

[4] A word RAM supports standard arithmetic and bitwise boolean operations on word-sized operands in $O(1)$ time.

weights on the edge from the root of R to v is at least w, and (3) no ancestor of v satisfies the above two properties. We call v the weighted level ancestor of u at depth w, and denote it by $WLA(u, w)$.

Assume that $\log W = O(\log n)$. In the internal-memory setting, we can construct an index that requires $O(n)$ words of space and finds $WLA(u, w)$ in $O(\log \log n)$ time [4]. In the following, we describe the result of a new WLA index that works cache-obliviously, which may be of independent interest. This result is based on a recursive structure with careful space management. The proof will be given in the Appendix.

Lemma 5. *We can construct a cache-oblivious index on R such that for any node u and any integer w, $WLA(u, w)$ can be reported in $O(\log \log_B n)$ I/O operations. The total space of the index is $O(n/B)$ disk pages.*

2.6 Cache-Oblivious Index for Join Operation

Let T be a text of length n. For any two strings Q_1 and Q_2, suppose that $LCP(Q_1, T)$ and $LCP(Q_2, T)$ are known. The *join* operation for Q_1 and Q_2 is to compute $LCP(Q_1Q_2, T)$, where Q_1Q_2 denotes the concatenation of Q_1 and Q_2.

Cole et al. (Section 5 of [13]) have developed an index of $O(n \log n)$ words that performs the *join* operation in $O(\log \log n)$ time in the internal-memory setting. Their index assumes the internal-memory results of LCA index, Y-fast trie, and WLA index. In the following lemma, we give an index that supports efficient *join* operations in the cache-oblivious setting.

Lemma 6. *We can construct a cache-oblivious index on T of $O((n \log n)/B)$ disk pages and supports the join operation in $O(\log \log_B n)$ I/O operations.*

Proof. Using Lemmas 3, 4, and 5, we can directly extend Cole et al.'s index into a cache-oblivious index. □

3 A Review of Cole et al.'s k-Error Matching

In this section, we review the internal-memory index for k-error matching proposed by Cole et al. [13], and explain the challenge in adapting it into a cache-oblivious index.

To index a text T of length n, Cole et al.'s index includes two data structures: (1) the suffix tree of T that occupies $O(n)$ words, and (2) a special tree structure, called k-error tree, that occupies a total of $O(n \log^k n)$ words of space. The k-error tree connects to a number of $(k-1)$-error trees, each of which connects to a number of $(k-2)$-error trees, and so on. The base of this recursive structure consists of 0-error trees.

Given a pattern P, Cole et al.'s matching algorithm considers different ways of making k edit operations on P in order to obtain an exact match in T. Intuitively, the matching algorithm first considers all possible locations of the leftmost error on P in which a match may exist; then for each location i that has an error, we can focus on searching the remaining suffix, $P[i+1..|P|]$, for subsequent errors. The searches are efficiently supported by the recursive tree structure. More precisely, at the top level, the k-error tree will immediately identify all matches of P in T with no errors; in addition, for those matches of P with at least one error, the k-error tree classifies the locations of the leftmost error on P into $O(\log n)$ groups, and then each group creates a search in a dedicated $(k-1)$-error tree. Subsequently, each $(k-1)$-error tree being searched will immediately identify all matches of P with one error, while for those matches of P with at least two errors, the $(k-1)$-error tree further classifies the locations of the second-leftmost error on P into $O(\log n)$ groups, and then each group creates a search in a dedicated $(k-2)$-error tree. The process continues until we get to the 0-error trees, where all matches of P with exactly k errors are reported.

The classification step in each k'-error tree is cleverly done to avoid repeatedly accessing characters in P. It does so by means of a constant number of LCA, LCP, Pred, and WLA queries; then, we are able to create enough search information (such as the starting position of the remaining suffix of P to be matched) to continue the subsequent $O(\log n)$ searches in the $(k'-1)$-error trees. Reporting matches in each error tree can also be done by a constant number of LCA, LCP, Pred, and WLA queries. In total, it can be done by $O(\log^k n)$ of these queries. See Figure 1 for the framework of Cole et al.'s algorithm. (In the subroutine SEARCH_ERROR_TREE, Line 2, Line 5 and Line 6 can be done by a constant number of LCA, LCP, Pred, and WLA queries; Line 8 requires $O(1)$ operations by following appropriate pointers.)

```
Algorithm APPROXIMATE_MATCH(P)
Input: A pattern P
Output: All occurrences of k-error match of P in T
1.  R ← k-error tree of T;
2.  SEARCH_ERROR_TREE(P, R, nil);
3.  return;

Subroutine SEARCH_ERROR_TREE(P, R, I)
Input: A pattern P, an error tree R, search information I
1.  if R is a 0-error tree
2.     then Output all matches of P with k errors based on R and I;
3.          return;
4.     else (* R is a k'-error tree for some k' > 0 *)
5.          Output all matches of P with k − k' errors based on R and I;
6.          Classify potential error positions into O(log n) groups based on P, R, and I;
7.          for each group i
8.                Identify the (k' − 1)-error tree R_i corresponding to group i;
9.                Compute search information I_i to continue the search in R_i;
10.               SEARCH_ERROR_TREE(P, R_i, I_i);
11. return;
```

Figure 1: Cole et al.'s algorithm for k-error matching.

Each LCA, Pred, or WLA query can be answered in $O(\log\log n)$ time. For the LCP queries, they are all in the form of $LCP(P_i, T)$, where P_i denotes the suffix $P[i..|P|]$. Instead of answering each query on demand, Cole et al. compute the answers of all these LCP queries at the beginning of the algorithm. There are $|P|$ such LCP queries, which can be computed in $O(|P|)$ time by exploiting the *suffix links* of the suffix tree of T (the algorithm is essentially McCreight's suffix tree construction algorithm [19]). Consequently, the answer of each LCP query is returned in $O(1)$ time when needed. Then, Cole et al.'s index supports k-error matching in $O(|P| + \log^k n \log\log n + occ)$ time, where occ denotes the number of occurrences.

3.1 Externalization of Cole et al.'s index

We are now ready to consider how to adapt Cole et al.'s index into a cache-oblivious index. Notice that each LCA, Pred, or WLA query can be answered in $O(\log\log_B n)$ I/Os by storing suitable data structures (see Lemma 3, 4, and 5). The only bottleneck lies in answering the $O(\log^k n)$ LCP queries. In the external-memory setting, though we can replace the suffix tree with Brodal and Fagerberg's cache-oblivious string dictionary (Lemma 2), if we compute $LCP(P_i, T)$ for all i in advance, we will need $\Omega(|P|^2/B)$ I/Os. Alternatively, if we compute each LCP query on demand without doing anything at the beginning, we will need a total of $\Omega((\log^k n)|P|/B)$ I/Os to answer all LCP queries during the search process. In summary, a direct adaptation of Cole et al.'s index into the external-memory setting will need $\Omega((\min\{|P|^2, |P|\log^k n\} + occ)/B + \log^k n \log\log_B n)$ I/Os for k-error matching.

In the next section, we propose another approach, where we compute the answers of *some* useful LCP queries using $O(|P|/B + k \log_B n)$ I/Os at the beginning, so that each subsequent query of $LCP(P_i, T)$ can be answered efficiently in $O(\log \log_B n)$ I/Os (see Lemma 9 in Section 4). This result leads to the following theorem.

Theorem 1. *For a fixed integer k, we can construct a cache-oblivious index on T of size $O((n \log^k n)/B)$ disk pages such that, given any pattern P, the k-error matches of P can be found in $O((|P| + occ)/B + \log^k n \log \log_B n)$ I/O operations.*

4 Cache-Oblivious k-Error Matching

Let P be a pattern, and let $P_i = P[i..|P|]$ be a suffix of P. In this section, we show how to preprocess P in $O(|P|/B + k \log_B n)$ I/Os. The aim is to obtain the answers of some useful $LCP(P_i, T)$ queries, such that a subsequent query of $LCP(P_j, T)$ for any j can be answered in $O(\log \log_B n)$ I/Os.

We observe that for a general pattern P, the above target may be difficult to achieve. Instead, we take advantage of concerning only those query patterns that potentially have a k-error match. We formulate a notion called *k-partitionable* and show that

- if P is k-partitionable, we can achieve the above target;
- if P is not k-partitionable, there must be no k-error match of P in T.

In Section 4.1, we first define the k-partitionable property, and describe an efficient *screening test* that checks whether P is k-partitionable; in case P is k-partitionable, the screening test would have computed the answers of $LCP(P_i, T)$ queries for some i as a by-product. In Section 4.2, we show how to utilize the answers precomputed in the screening test to answer $LCP(P_j, T)$ for any j in $O(\log \log_B n)$ I/Os.

In the following, we assume that we have maintained the suffix array and inverse suffix array of T. Each entry of these two arrays will be accessed one at a time, at the cost of one I/O per access.

4.1 k-Partitionable and Screening Test

Consider the following partitioning process on P. In Step 1, we delete the first ℓ characters of P where ℓ is the LCP length reported by $LCP(P, T)$. While P is not empty, Step 2 removes further the first character from P. Then, we repeatedly apply Step 1 and Step 2 until P is empty. In this way, P is partitioned into $\pi_1, c_1, \pi_2, c_2, \ldots, \pi_d, c_d, \pi_{d+1}$ such that π_i is a string that appears in T, and c_i is called a *cut-character* such that $\pi_i c_i$ is a string that does not appear in T. (Note that π_{d+1} is an empty string if P becomes empty after some Step 2.) Note that this partitioning is unique, and we call this the *greedy partitioning* of P.

Definition 1. *P is called k-partitionable if the greedy partitioning of P consists of at most k cut-characters.*

The following lemma states that the k-partitionable property is a necessary condition for the existence of a k-error match.

Lemma 7. *If P has a k-error match, P is k-partitionable.*

Proof. Suppose on contrary that P is not k-partitionable and the greedy partitioning of P has more than k cut-characters. Then, consider a particular k-error match M of P. Since $\pi_i c_i$ is not a string in T, there must be at least one edit operation in that region to turn P into M. This implies that at least $k + 1$ edit operations are needed to turn P into M, which is a contradiction. □

The *screening test* on P performs the greedy partitioning of P to check if P is k-partitionable. If P is not k-partitionable, we can immediately conclude that P does not have a k-error match in T. One way to perform the screening test is to apply Lemma 2 repeatedly, so that we discover π_1 and c_1 in $O(|P|/B + \log_B n)$ I/O operations, then discover π_2 and c_2 in $O((|P| - |\pi_1| - 1)/B + \log_B n)$ I/O operations, and so on. However, in the worst case, this procedure will require $O(k(|P|/B + \log_B n))$ I/O operations. In the following lemma, we make a better use of Lemma 2 with the standard doubling technique and show how to use $O(|P|/B + k \log_B n)$ I/O operations to determine whether P passes the screening test or not.

Lemma 8. *The screening test on P can be done cache-obliviously in $O(|P|/B+k \log_B n)$ I/O operations.*

Proof. Let $r = \lceil |P|/k \rceil$. In Round 1, we perform the following steps.

- We apply Lemma 2 on $P[1..r]$ to see if it appears in T. If so, we double the value of r and check if $P[1..r]$ appears in T. The doubling continues until we obtain some $P[1..r]$ which does not appear in T, and in which case, we have also obtained π_1 and $LCP(\pi_1, T)$.
- Next, we remove the prefix π_1 from P. The first character of P will then become the cut-character c_1, and we apply Lemma 2 to get $LCP(c_1, T)$. After that, remove c_1 from P.

In each subsequent round, say Round i, we reset the value of r to be $\lceil |P|/k \rceil$, and apply the same steps to find π_i and c_i (as well as $LCP(\pi_i, T)$ and $LCP(c_i, T)$). The algorithm stops when P is empty, or when we get c_{k+1}.

It is easy to check that the above process correctly outputs the greedy partitioning of P (or, up to the cut-character c_{k+1} if P does not become empty) and thus checks if P is k-partitionable. The number of I/O operations of the above process can be bounded as follows. Let a_i denote the number of times we apply Lemma 2 in Round i, and b_i denote the total number of characters compared in Round i. Then, the total I/O cost is at most $O((\sum_i b_i)/B + (\sum_i a_i) \log_B n)$ by Lemma 2. The term $\sum_i b_i$ is bounded by $O(|P| + k)$ because Round i compares $O(|\pi_i| + \lceil |P|/k \rceil)$ characters, and there are only $O(k)$ rounds. For a_i, it is bounded by $O(\log(k|\pi_i|/|P|) + 1)$, so that by Jensen's inequality, the term $\sum_i a_i$ is bounded by $O(k)$. □

4.2 Computing LCP for k-Partitionable Pattern

In case P is k-partitionable, the screening test in Section 4.1 would also have computed the answers for $LCP(\pi_i, T)$ and $LCP(c_i, T)$. To answer $LCP(P_j, T)$, we will make use of the *join* operation (Lemma 6) as follows. Firstly, we determine which π_i or c_i covers the jth position of P.[5] Then, there are two cases:

- **Case 1:** If the jth position of P is covered by π_i, we notice that the LCP length of $LCP(P_j, T)$ cannot be too long since $\pi_{i+1}c_{i+1}$ does not appear in T. Let $\pi_i(j)$ denote the suffix of π_i that overlaps with P_j. Indeed, we have:

  Fact 1. $LCP(P_j, T) = LCP(\pi_i(j)c_i\pi_{i+1}, T)$.

  This shows that $LCP(P_j, T)$ can be found by the *join* operations in Lemma 6 repeatedly on $\pi_i(j)$, c_i and π_{i+1}. The SA range of $\pi_i(j)$ can be found easily using SA, SA^{-1} and WLA as follows. Let $[p, q]$ be the SA range of π_i. The pth smallest suffix is the string $T[SA[p]..n]$, which has π_i as a prefix. We can compute $p' = SA^{-1}[SA[p] + j]$, such that the p'th smallest suffix has $\pi_i(j)$ as a prefix. Using the WLA index, we can locate the node (or edge) in the suffix tree of T corresponding to $\pi_i(j)$. Then, we can retrieve the required SA range from this node. The LCP query on P_j can be answered in $O(\log \log_B n)$ I/O operations.

[5] This is in fact a predecessor query and can be answered in $O(\log \log_B n)$ I/O operations by maintaining a Y-fast trie for the starting positions of each π_i and c_i.

- **Case 2:** If c_i is the jth character of P, the LCP query on P_j can be answered by the *join* operation on c_i and π_{i+1} in $O(\log\log_B n)$ I/O operations, using similar arguments as in Case 1.

Thus, we can conclude the section with the following lemma.

Lemma 9. *Let T be a text of length n, and k be a fixed integer. Given any pattern P, we can perform a screening test in $O(|P|/B + k\log_B n)$ I/Os such that if P does not pass the test, it implies that there is no k-error match of P in T. In case P passes the test, $LCP(P[j..|P|], T)$ for any j can be returned in $O(\log\log_B n)$ I/Os.*

5 An $O(n\log n)$-Space Cache-Oblivious Index

5.1 A Review of Chan et al.'s k-error Matching

Below we review the $O(n)$-space index for k-error pattern matching given by Chan et al. [11]. For simplicity, we only consider the case of Hamming distance.

Consider a text $T[1..n]$ over an alphabet Σ of constant size. Let $\beta = \Theta(\log^{k+1} n)$ be an integer. We first focus on handling "long" patterns, whose lengths are at least β. We call a character $T[a]$ a check-point if a is a multiple of β. Given a long pattern P, we cut P into $P[1..i-1]$ and $P[i..m]$ for all $i \in [1, \beta]$. For all k_1 and k_2 where $k_1 + k_2 \leq k$, we will search for all check-points a such that $P[i..m]$ has a k_2-error match starting from $T[a]$, and $P[1..i-1]$ has a k_1-error match ending at $T[a-1]$. Thus, P has a $(k_1 + k_2)$-error match starting at $T[a - i + 1]$.

The index is divided into two parts.

1. (Pattern Matching Index) T is indexed using some special search trees for the check-points, called $TAIL_\ell$ and $HEAD_\ell$, where $\ell \in [0, k]$. Each leaf in $TAIL_\ell$ (or $HEAD_\ell$) corresponds to a check-point. Given a pattern X, $TAIL_\ell$ supports finding all its ℓ-error matches in T that start at some check-points. Similarly, $HEAD_\ell$ finds all ℓ-error matches ending at the position in T just before some check-points. $HEAD_\ell$ and $TAIL_\ell$ do not report directly all the check-points; instead they report a set of $O(\log^\ell n)$ nodes such that the union of all their descendant leaves covers all the matches.

2. (Set Intersection Index) For any k_1, k_2 with $k_1 + k_2 \leq k$, if there exists a leaf x_1 in $TAIL_{k_1}$ and a leaf x_2 in $HEAD_{k_2}$ both corresponding to the same check-point a, then (x_1, x_2) is called a connecting pair. Given a node u_1 in $TAIL_{k_1}$ and a node u_2 in $TAIL_{k_2}$, the index for connecting pairs supports finding connecting pairs (x_1, x_2) where x_1 and x_2 are descendants of u_1 and u_2 respectively.

Chan et al. adapted the work of Cole et al. [13] to index the check-points in $O(n)$ space. Given a pattern P, after an $O(|P|)$-time preprocessing, the required nodes in the Pattern Matching Index can be found in $O(\beta\log^k n\log\log n) = O(\log^{2k+1} n\log\log n)$ time. In the Set Intersection Index, each pair of trees $(TAIL_{k_1}, HEAD_{k_2})$ is indexed using the technique of Buchsbaum et al. [10] for Tree-Cross-Product. The total space required is $O(n)$, and finding all connecting pairs takes $O(\log^{2k+1} n\log\log n + occ)$ time.

For short patterns with length less than β, we can use the data structure by Lam et al. [17], which supports k-error searching in $O(|\Sigma|^k|P|^k\log\log n + occ)$ time. If Σ is of constant size, then the time complexity becomes $O(\log^{k(k+1)} n\log\log n + occ)$.

5.2 Cache-Oblivious Data Structure

Now we show how to turn Chan et al.'s index into a cache-oblivious index by providing the corresponding cache-oblivious data structures.

Pattern Matching. Let T' (resp. P'_i) denote the string with all characters in string T (resp. P_i) reversed. Analogous to Section 3, Chan et al.'s data structure makes $O(\log^k n)$ LCA, Pred, WLA and LCP queries, where LCP queries are in the form $LCP(P_i, T)$ and $LCP(P'_i, T')$ where $1 \leq i \leq |P|$. We can build cache-oblivious data structures in Section 2 so that LCA, Pred and WLA queries can be answered in $O(\log \log_B n)$ I/O. Similar to the previous section, the major difficulty is to avoid using $\Omega(|P|^2/B)$-time to compute the answers of all LCP queries in advance for all i. By preprocessing P and P' using our screening test and greedy partition, LCP queries can be answered by Lemma 9 using $O(\log \log_B n)$ I/Os. This data structure requires $O(n \log n)$ space due to a data structure in Lemma 6 that was not needed in the index of Chan et al.

Set Intersection. There is no cache-oblivious data structure for Tree-Cross-Product. Here, we show a reduction of Tree-Cross-Product to two-dimensional orthogonal range queries. We first look at the Tree-Cross-Product indexing problem.

Let T_1 and T_2 be trees with at most n leaves. Let E be a set of pair of leaves (v_1, v_2) such that v_1 is a leaf of T_1 and v_2 is a leaf of T_2. We call the pairs in E connecting pairs. Given a node u_1 in T_1 and a node u_2 in T_2, we ask for all connecting pairs (v_1, v_2) such that v_1 and v_2 are descendants of u_1 and u_2, respectively.

The data structure of Buchsbaum et al. [10] takes $O(n + |E| \log n)$ space and answers each query in $O(\log \log n + occ)$ time where occ is the number of connecting pairs reported. We show the following cache-oblivious data structure to answer each Tree-Cross-Product query in $O(\log_B n + occ/B)$ I/O.

1. We label the leaves from left to right by their pre-order ranking. For each internal node, we store the smallest and the largest labels of all its descendant leaves. The data structure requires $O(n/B)$ pages, and on given a node u_1 in T_1, it takes $O(1)$ I/O to retrieve the corresponding stored labels. The tree T_2 is indexed similarly.

2. For a connecting pair (v_1, v_2), we store a point (x, y) in the two-dimensional plane such that x and y are labels of v_1 and v_2, respectively. Given an axis-parallel rectangle (x_1, y_1)–(x_2, y_2), we want to locate all points inside the rectangle. The data structure by Arge et al. [1] requires $O(|E| \log^2 n/B \log \log n)$ disk pages and takes $O(\log_B n + occ/B)$ I/Os to answer the query.

The above data structures handle "long" patterns in $O((|P| + occ)/B + \log^{2k+1} n \log_B n)$ I/Os. The Pattern Matching index requires $O(n \log n/B)$ disk pages and the Set Intersection index requires $O((n/\beta) \log^k n(\log^2 n/ \log \log n)) = O(n \log n/(B \log \log n))$ disk pages. The total space requirement is $O(n \log n/B)$ disk pages.

For short patterns, the index of Lam et al. [17] can be used directly as it stores the occurrences of each short pattern contiguously in the memory. Hence, searching takes only $O(\log^{k(k+1)} n \log \log n + occ/B)$ I/Os.

Theorem 2. *For a fixed integer $k \geq 2$, we can construct a cache-oblivious index on T of size $O(n \log n/B)$ disk pages such that on given any pattern P, the k-error matches of P can be found in $O((|P|+occ)/B+ \log^{k(k+1)} n \log \log n)$ I/O operations. For $k = 1$, searching takes $O((|P| + occ)/B + \log^3 n \log_B n)$ I/O operations.*

6 Conclusion

We adapted Cole et al.'s approximate matching index into a cache-oblivious index. We identified the answering of all LCP queries as the major bottleneck in a naive implementation. To circumvent this bottleneck, we compute only the answers of some "useful" LCP queries in advance, so that each subsequent LCP query can still be answered efficiently. We take advantage of a new notion called *k-partitionable* and devise an I/O-efficient *screening test* that checks whether P is k-partitionable. If

P is k-partitionable, the screening test would also compute the answers of some useful LCP queries as a by-product, which can then be utilized to answer LCP query for arbitrary P_i efficiently.

Together with other cache-oblivious data structures, we are able to construct an index to find all k-error matches using $O((|P| + occ)/B + \log^k n \log\log_B n)$ I/O operations. The space of the index is $O((n \log^k n)/B)$ disk pages. To the best of our knowledge, this is the first external-memory data structure that does not require $\Omega(|P| + occ + \text{poly}(\log n))$ I/O operations. Our index can work even if the alphabet size is unbounded.

When the alphabet size is a constant, we extended the internal-memory result of Chan et al. [11] to work in the external memory setting. The difficulties of the extension come in two parts: (i) computing the answers of the LCP queries, and (ii) answering the Tree-Cross-Product queries. For (i), we will use the same approach as we externalize Cole et al.'s index. For (ii), there is no external memory counter-part for the data structure of [10]; instead, we show how to reduce the Tree-Cross-Product query to a two-dimensional orthogonal range search query, so that we can store an appropriate data structure of [1] that can answer the latter query efficiently. In this way, for any fixed $k \geq 2$, we can construct an index using $O((n \log n)/B)$ disk pages, which can find all k-error matches of P in $O((|P| + occ)/B + \log^{k(k+1)} n \log\log n)$ I/Os.

It is open whether we can further reduce the number of searching I/Os to $O((|P| + occ)/B + \text{poly}(\log_B n))$. Another related open problem is on the construction of these indexes. It would be nice to know if one can obtain cache-oblivious construction algorithms for these indexes.

7 Acknowledgments

The authors would like to thank Gerth Stølting Brodal and Rolf Fagerberg for helpful discussion on their results in [9].

References

[1] L. Arge, G. S. Brodal, R. Fagerberg, and M. Laustsen. Cache-Oblivious Planar Orthogonal Range Searching and Counting. In *Proceedings of ACMl Symposium on Computational Geometry*, pages 160–169, 2005.

[2] A. Aggarwal and J. S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Text Indexing and Dictionary Matching with One Error. *Journal of Algorithms*, 37(2):309–325, 2000.

[4] A. Amir, G. M. Landau, M. Lewenstein and D. Sokol. Dynamic Text and Static Pattern Matching. *ACM Transactions on Algorithms*, 3(2), 2007.

[5] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest Common Ancestors in Trees and Directed Acyclic Graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

[6] M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. Cache-Oblivious String B-trees. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 233–242, 2006.

[7] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-Oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[8] G. S. Brodal and R. Fagerberg. Funnel Heap—A Cache Oblivious Priority Queue. In *Proceedings of International Symposium on Algorithms and Computation*, pages 219–228, 2002.

[9] G. S. Brodal and R. Fagerberg. Cache-Oblivious String Dictionaries. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, pages 581–590, 2006.

[10] A. L. Buchsbaum, M. T. Goodrich, and J. Westbrook. Range Searching Over Tree Cross Products. In *Proceedings of European Symposium on Algorithms*, pages 120–131, 2000.

[11] H. L. Chan, T. W. Lam, W. K. Sung, S. L. Tam, and S. S. Wong. A Compressed Indexes for Approximate Pattern Matching. *Algorithmica*, 58(2):263–281, 2010.

[12] A. L. Cobbs. Fast Approximate Matching using Suffix Trees. In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 41–54, 1995.

[13] R. Cole, L.A. Gottlieb, and M. Lewenstein. Dictionary Matching and Indexing with Errors and Don't Cares. In *Proceedings of Symposium on Theory of Computing*, pages 91–100, 2004.

[14] P. Ferragina and R. Grossi. The String B-tree: A New Data Structure for String Searching in External Memory and Its Application. *Journal of the ACM*, 46(2):236–280, 1999.

[15] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. In *Proceedings of IEEE Symposium on Foundations of Computer Science*, pages 285–298, 1999.

[16] D. Harel and R. Tarjan. Fast Algorithms for Finding Nearest Common Ancestor. *SIAM Journal on Computing*, 13:338–355, 1984.

[17] T. W. Lam, W. K. Sung, and S. S. Wong. Improved Approximate String Matching Using Compressed Suffix Data Structures. *Algorithmica*, 51(3):298–314, 2008.

[18] U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[19] E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[20] H. Prokop. *Cache-Oblivious Algorithms*, Master's thesis, Massachusetts Institute of Technology, 1999.

[21] E. Ukkonen. Approximate Matching Over Suffix Trees. In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 228–242, 1993.

[22] P. van Emde Boas. Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space. *Information Processing Letters*, 6(3):80–82, 1977.

[23] P. van Emde Boas, R. Kaas, and Zijlstra. Design and Implementation of an Efficient Priority Queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[24] J. S. Vitter. Algorithms and Data Structures for External Memory. *Foundations and Trends® in Theoretical Computer Science*, 2(4):305–474, 2008.

[25] P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[26] D. E. Willard. Log-Logarithmic Worst-Case Range Queries are Possible in Space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.

A Appendix

A.1 Cache-Oblivious WLA Index

Given a rooted tree R with n nodes, with integral weights on each edge drawn from $[1, W]$, we want to construct an index on R such that on given any node u and any integer w, we can find the ancestor node v of u such that the sum of weights on the edge from the root to v just greater than or equal to w. We call v the weighted level ancestor of u at depth w, and denote it by $WLA(u, w)$.

Assume that $\log W = O(\log n)$. In the following, we describe our cache-oblivious implementation of the WLA index.

Firstly, we focus on an index that supports WLA query for any node u being a leaf in R. Let us call that a WLA-for-leaf index on R. After that, with minor modification to the WLA-for-leaf index, we can easily extend it to support WLA query for the internal nodes in R as well.

For ease of discussion, we define some notation as follows. For each node v in R, the sum of the weights of all edges on the path from the root to v is called the *depth* of v. We call the subtree formed by those nodes having at least $\sqrt{n}$ descendant leaves the *top tree* of R. For each node x not in the top tree, but whose parent is in the top tree, we call the subtree rooted at x in R a *bottom tree* rooted at x. The edge that connects the bottom tree to a top tree is called a *middle edge*. That is, R can be decomposed into one top tree connected to multiple bottom trees through middle edges. Note that the

number of leaves in a top tree or in a bottom tree is at most $\sqrt{n}$. For any rooted tree S, a *compact tree* of S is defined by contracting every non-root degree-1 internal node of S.

A simple recursive structure: Our WLA-for-leaf index is a recursive structure based on the top-bottom tree decomposition, and it is stored using the van Emde Boas layout [7, 22, 23]. Precisely, the WLA-for-leaf index of a tree consists of the WLA-for-leaf index of the compact tree of its top tree first, the WLA index for each of the middle edges,[6] and the WLA-for-leaf index of the compact tree of each of its bottom tree. Our recursion stops when the number of leaves in the compact tree is one; in this case, the tree consists of a root node x and a leaf node ℓ. The corresponding WLA-for-leaf index will be a pointer to a cache-oblivious Y-fast trie structure (stored at some other memory locations) that is defined on the depth of all nodes on the path from x to ℓ in the original tree R. Similarly, the WLA-index for a middle edge (v, x) will be a pointer to a cache-oblivious Y-fast trie defined on the depth of all nodes on the path from v to x in R.

Before we give further details about WLA-for-leaf index when there is more than one leaf in the tree, let us see how one would find WLA of a leaf u at depth w based on the WLA-for-leaf indexes of (the compact tree of) the top tree and the bottom trees. Let x be the root of the bottom tree that contains u, and let (v, x) be the middle edge connecting the bottom tree to its top tree. (Recall that v is a node in the top tree.) There can be three cases:

Case 1: If w is between the depth of x and the depth of v, the desired WLA can be found by consulting the WLA index for the middle edge (v, x).

Case 2: Else if the depth of v is more than w, let ℓ' be the leftmost descendant leaf of v in the top tree. The desired WLA will be $WLA(\ell', w)$, which can be found by consulting the WLA-for-leaf index in the top tree.

Case 3: Else, we have the depth of x is less than w. The desired WLA can be found by consulting the WLA-for-leaf index in the bottom tree rooted at x.

In order to support the above procedure efficiently (when there is more than one leaf in the tree), we define the WLA-for-leaf index as follows: For each leaf u in the tree, we store the information `(i)` the depth of x (the root of the bottom tree where u belongs) and the depth of v, `(ii)` a pointer to the WLA index for the middle edge (v, x), `(iii)` a pointer to ℓ' (the leftmost descendant leaf of v in the top tree) in the WLA-for-leaf index for the top tree, and `(iv)` a pointer to u in the WLA-for-leaf index for the bottom tree containing u. With this information stored, we can find WLA of any leaf at any depth by recursion.

To count the number of I/O operations involved, we observe two things: (1) Though we do not know the actual value of B, once the size of the compact tree in the recursion contains at most B leaves (thus, $\Theta(B)$ nodes in total), the above procedure will access $O(1)$ number of disk pages in the remaining recursion; the reason is that by our memory layout, the WLA-for-leaf index of this tree fits into $O(1)$ disk pages. (2) If the number of leaves in a tree is n', the number of leaves in its top tree, or in each of its bottom tree, is at most $\sqrt{n'}$. Combining (1) and (2), we can conclude that there are at most $O(\log\log_B n)$ recursion steps to reduce the tree size from n to B (though we do not know B in advance), and for each of these recursion steps, it requires $O(1)$ number of I/O operations. Then, at the end of the recursion, we will follow a pointer to a Y-fast trie structure, and perform a predecessor search. By Lemma 4, this final step takes at most $O(\log\log_B(nW))$ I/O operations, which can be bounded by $O(\log\log_B n)$ as we assume $\log W = O(\log n)$. Thus, we can solve WLA for any leaf using $O(\log\log_B n)$ I/O operations.

A more compact structure: Unfortunately, the major drawback of the above recursive structure is its space usage. Consider any leaf u. It will be in a bottom tree of R, in a bottom tree of a bottom tree

[6] A middle edge in the current tree may refer to multiple edges in the original tree, since recursion is performed on compact trees.

of R, a bottom tree of a bottom tree of a bottom tree of R, and so on. With each leaf requiring $O(1)$ number of words of space (to store its information of `(i)` through `(iv)`), we can see that the above recursive structure takes at least $\Omega(n \log\log n)$ words of storage. One easy fix to reduce the space is to *remove the leaves of the bottom tree* in the recursion. Precisely, if S_x is a bottom tree of S rooted at x, we obtain a tree S'_x from S_x by removing all its leaves, and replace the original WLA-for-leaf index on the compact tree of S_x by the WLA-for-leaf index on the compact tree of S'_x. Then, for any leaf u in the bottom tree S_x, let (v', u) be the edge that connects u to S_x. In addition to keeping the previous information `(i)` to `(iii)`, we store `(iv)` the depth of u and the depth of v', `(v)` a pointer to ℓ'' in the WLA-for-leaf index of S'_x, where ℓ'' is the leftmost descendant leaf of v' in S'_x, and `(vi)` a pointer to a Y-fast trie structure (stored at some other memory locations) that is defined on the depth of all nodes on the path from v' to u in the original tree R. In this way, recursion in the top tree is the same as before, while recursion in the bottom tree can be done by the modified Case 3 as follows:

Modified Case 3: Else, we have the depth of x is less than w.

1. If the depth of u is less than w, report "not found" as $WLA(u, w)$ is undefined.
2. Else if the depth of v' is less than w, the desired WLA is on the path from v' to u in the original tree, which can be found by a predecessor search in the Y-fast trie pointed by u.
3. Else, the desired WLA is $WLA(\ell'', w)$, which can be found by consulting the WLA-for-leaf index on (the compact tree of) S'_x.

With the above modified implementation, we have the following lemma.

Lemma 10. *We can construct a cache-oblivious index on R such that for any leaf u and any integer w, $WLA(u, w)$ can be reported in $O(\log\log_B n)$ I/O operations. The total space of the index is $O(n/B)$ disk pages.*

Proof. In the modified implementation, the I/O operations can be bounded in the same way as in the original implementation. For the space, each node in R can become at most once as a leaf in some bottom tree, in which case $O(1)$ words are required to store the leaf information. And for each edge in R, it can be in a path corresponding to at most one Y-fast trie structure. It is because once a path is stored by a Y-fast trie, either the recursion stops, or the path contains a leaf in some bottom tree to be removed from recursion. This implies that the path (and the edges on it) will not appear in subsequent recursive structures. Thus, the number of words required for the Y-fast trie is linear to the number of edges in R. As R has $O(n)$ nodes and edges, the total space is $O(n)$ words, which is stored in $O(n/B)$ disk pages with our memory layout. □

To get the general WLA index, we observe that for any internal node v, suppose that ℓ_v is the leftmost descendant leaf of v in R, then $WLA(v, w)$ is undefined when depth of v is less than w, and otherwise, it will be $WLA(\ell_v, w)$. Thus, for each internal node v, if we store along `(i)` its depth and `(ii)` a pointer to ℓ_v, we can use the WLA-for-leaf index to solve the WLA of any internal node at any depth. The total space of this augmentation is $O(n)$ words, or $O(n/B)$ disk pages. This gives the following theorem.

Theorem 3. *Given a rooted tree R with n nodes, where edges in the tree have integral weights from $[1, W]$ with $\log W = O(\log n)$, we can construct a cache-oblivious index on R such that for any node v and any integer w, $WLA(v, w)$ can be reported in $O(\log\log_B n)$ I/O operations. The total space of the index is $O(n/B)$ disk pages.*