Practical Suffix Tree Construction

Sandeep Tata Richard A. Hankins Jignesh M. Patel

University of Michigan
1301 Beal Avenue; Ann Arbor, MI 48109-2122; USA
{tatas,hankinsr,jignesh}@eecs.umich.edu

Abstract

Large string datasets are common in a number of emerging text and biological database applications. Common queries over such datasets include both exact and approximate string matches. These queries can be evaluated very efficiently by using a suffix tree index on the string dataset. Although suffix trees can be constructed quickly in memory for small input datasets, constructing persistent trees for large datasets has been challenging. In this paper, we explore suffix tree construction algorithms over a wide spectrum of data sources and sizes. First, we show that on modern processors, a cache-efficient algorithm with $O(n^2)$ complexity outperforms the popular $O(n)$ Ukkonen algorithm, even for in-memory construction. For larger datasets, the disk I/O requirement quickly becomes the bottleneck in each algorithm's performance. To address this problem, we present a buffer management strategy for the $O(n^2)$ algorithm, creating a new disk-based construction algorithm that scales to sizes much larger than have been previously described in the literature. Our approach far outperforms the best known disk-based construction algorithms.

1 Introduction

Querying large string datasets is becoming increasingly important in a number of emerging text and life sciences applications. Life science researchers are often interested in explorative querying of large biological sequence databases, such as genomes and large sets of protein sequences. Many of these biological datasets are growing at exponential rates — for example, the sizes of the sequence datasets in GenBank have been doubling every sixteen months [31]. Consequently, methods for *efficiently* querying large string datasets are critical to the success of these emerging database applications.

Suffix trees are versatile data structures that can help execute such queries very efficiently. In fact, suffix trees are useful for solving a wide variety of string based problems [17]. For instance, the exact substring matching problem can be solved in time proportional to the length of the query, once the suffix tree is built on the database string. Suffix trees can also be used to solve approximate string matching problems efficiently. Some bioinformatics applications such as MUMmer [10, 11, 22], REPuter [23], and OASIS [25] exploit suffix trees to efficiently evaluate queries on biological sequence datasets. However, suffix trees are not widely used because of their high cost of construction. As we show in this paper, building a suffix tree on moderately sized datasets, such as a single chromosome of the human genome, takes over 1.5 hours with the best known existing disk-based construction technique [18]. In contrast, the techniques that we develop in this paper reduce the construction time by a factor of 5 on inputs of the same size.

Even though suffix trees are currently not in widespread use, there is a rich history of algorithms for constructing suffix trees. A large focus of previous research has been on linear-time suffix tree construction algorithms [24, 32, 33]. These algorithms are well suited for small input strings where the tree can be constructed entirely in main memory. The growing size of input datasets, however, requires that we construct suffix trees efficiently on disk. The algorithms proposed in [24, 32, 33] cannot be used for disk-based construction as they have poor locality of reference. This poor locality causes a large amount of random disk I/O once the data structures no longer fit in main memory. If we naively use these main-memory algorithms for on-disk suffix tree construction, the process may take well over a day for a single human chromosome.

Large (and rapidly growing) size of many string datasets underscores the need for fast *disk-based* suffix tree construction algorithms. A few recent research efforts have also considered this problem [4, 18], though neither of these approaches scales well for large datasets (such as a large chromosome, or an entire eukaryotic genome).

In this paper, we present a new approach to *efficiently*

Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004

construct suffix trees on disk. We use a philosophy similar to the one in [18]. We forgo the use of suffix links in return for a much better memory reference pattern, which translates to better scalability and performance for large trees.

The main contributions of this paper are as follows:

1. We introduce the "Top Down Disk-based" (TDD) approach to building suffix trees efficiently for a wide range of sizes and input types. This technique, includes a suffix tree construction algorithm called PWOTD, and a sophisticated buffer management strategy.

2. We compare the performance of TDD with the popular Ukkonen's algorithm [32] for the in-memory case, where all the data structures needed for building the suffix trees are memory resident (i.e. the datasets are "small"). Interestingly, we show that even though Ukkonen has a better worst case *theoretical* complexity, TDD outperforms Ukkonen on modern cached processors, since TDD incurs significantly fewer processor cache misses.

3. We systematically explore the space of data sizes and types, and highlight the advantages and disadvantages of TDD with respect to other construction algorithms.

4. We experimentally demonstrate that TDD scales gracefully with increasing input size. Using the TDD process, we are able to construct a suffix tree on the *entire human genome in 30 hours* (on a single processor machine)! To our knowledge, suffix tree construction on an input string of this size (3 billion symbols approx.) has yet to be reported in literature.

The remainder of this paper is organized as follows: Section 2 discusses related work. The TDD technique is described in Section 3, and we analyze the behavior of this algorithm in Section 4 . Section 5, presents the experimental results, and Section 6 presents our conclusions.

2 Related Work

Linear time algorithms for constructing suffix trees have been described by Weiner [33], McCreight [24], and Ukkonen [32]. Ukkonen's is a popular algorithm because it is easier to implement than the other algorithms. It is an $O(n)$, in-memory construction algorithm based on the clever observation that constructing the suffix tree can be performed by iteratively expanding the leaves of a partially constructed suffix tree. Through the use of *suffix links*, which provide a mechanism for quickly traversing across sub-trees, the suffix tree can be expanded by simply adding the $i+1$ character to the leaves of the suffix tree built on the previous i characters. The algorithm thus relies on suffix links to traverse through all of the sub-trees in the main tree, expanding the outer edges for each input character. However, they have poor locality of reference since they traverse the suffix tree nodes in a random fashion. This leads to poor performance on cached architectures and when used to construct on-disk suffix trees.

Recently, Bedathur et al. developed a buffering strategy, called TOP-Q, which improves the performance of the Ukkonen's algorithm (which uses suffix links) when constructing on-disk suffix trees [4]. A different approach was suggested by Hunt et al. [18] where the authors drop the use of suffix links and use an $O(n^2)$ algorithm with a better locality of reference. In one pass over the string, they index all suffixes with the same prefix by inserting them into an on-disk subtree managed by PJama [3], a Java based object store. Construction of each independent subtree requires a full pass over the string.

Several $O(n^2)$ and $O(n \log n)$ algorithms for constructing suffix trees are described in [17]. A top-down approach has been suggested in [1, 14, 16]. In [15], the authors explore the benefits of using a lazy implementation of suffix trees. In this approach, the authors argue that one can avoid paying the full construction cost by constructing the subtree only when it is accessed for the first time. This approach is useful only when a small number of queries are posed against a string dataset. When executing a large number of queries, most of the tree must be materialized, and in this case, this approach will perform poorly.

Previous research has also produced theoretical results on understanding the average sizes of suffix trees [5, 30], and theoretical complexity of using sorting to build suffix trees for different computational models such as RAM, PRAM, and various other external memory models [12].

Suffix arrays have also been used as an alternative to suffix trees for specific string matching tasks [8, 9, 26]. However, in general, suffix trees are more versatile data structures. The focus of this paper is only on suffix trees.

Our solution uses a simple partitioning strategy. However, a more sophisticated partitioning method has been proposed recently [6], which can complement our existing partitioning method.

3 The TDD Technique

Most suffix tree construction algorithms do not scale due to the prohibitive disk I/O requirements. The high per-character overhead quickly causes the data structures to outgrow main memory and the poor locality of reference makes efficient buffer management difficult.

We now present a new disk-based construction technique called the "Top-Down Disk-based" technique, hereafter referred to simply as TDD. TDD scales much more gracefully than existing techniques by reducing the main-memory requirements through strategic buffering of the largest data structures. The TDD technique consists of a suffix tree construction algorithm, called PWOTD, and the related buffer management strategy described in the following sections.

3.1 PWOTD Algorithm

The first component of the TDD technique is our suffix tree construction algorithm, called PWOTD (Partition and Write Only Top Down). This algorithm is based on the *wotdeager* algorithm suggested by Kurtz [15]. We improve on

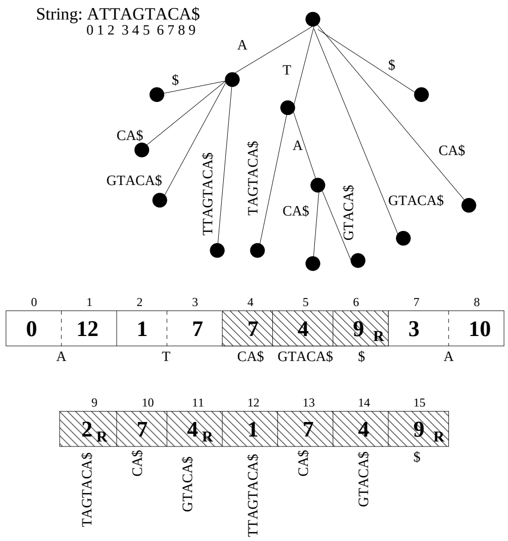

Figure 1: Suffix Tree Representation

this algorithm by using a partitioning phase which allows one to immediately build larger, independent sub-trees in memory. Before we explain the details of the algorithm, we briefly discuss the representation of the suffix tree.

The suffix tree is represented by a linear array, as in *wotdeager*. This is a compact representation using an average of 8.5 bytes per symbol indexed. Figure 1 illustrates a suffix tree on the string ATTAGTACA$ and the tree's corresponding array representation in memory. Shaded entries in the array represent leaf nodes, with all other entries representing non-leaf nodes. An *R* in the lower right-hand corner of an entry denotes a rightmost child. A branching node is represented by two integers. The first is an index into the input string; the character at that index is the starting character of the incoming edge's label. The length of the label can be deduced by examining the children of the current node. The second entry points to the first child. Note that the leaf nodes do not have a second entry. The leaf node requires only the starting index of the label; the end of the label is the string's terminating character. See [15] for a more detailed explanation.

The PWOTD algorithm consists of two phases. In phase one, we partition the suffixes of the input string into $|A|^{prefixlen}$ partitions, where $|A|$ is the alphabet size of the string and *prefixlen* is the depth of the partitioning. The partitioning step is executed as follows. The input string is scanned from left to right. At each index position i the *prefixlen* subsequent characters are used to determine one of the $|A|^{prefixlen}$ partitions. This index i is then written to the calculated partition's buffer. At the end of the scan, each partition will contain the suffix pointers for suffixes that all have the same prefix of size *prefixlen*.

To further illustrate the partition step, consider the following example. Partitioning the string ATTAGTACA$ using a *prefixlen* of 1 would create four partitions of suffixes, one for each symbol in the alphabet. (We ignore the final partition consisting of just the string terminator symbol $.) The suffix partition for the character A would be {0,3,6,8}, representing the suffixes {ATTAGTACA$, AGTACA$, ACA$, A$}. The suffix partition for the character T would be {1,2,5} representing the suffixes {TTAGTACA$, TAGTACA$, TACA$}. In phase two, we use the *wotdeager* algorithm to build the suffix tree on each partition using a top down construction.

Algorithm PWOTD(*String,prefixlen*)
Phase1:
Scan the *String* and partition *Suffixes* based on the first *prefixlen* symbols of each suffix
Phase2: Do for each partition:
1. START BuildSuffixTree
2. Populate *Suffixes* from current partition
3. Sort *Suffixes* on first symbol using *Temp*
4. Output branching and leaf nodes to the *Tree*
5. Push the nodes pointing to an unevaluated range onto the *Stack*

While *Stack* is not empty
6. Pop a node
7. Find the Longest Common Prefix (LCP) of all the suffixes in this range by checking the *String*
8. Sort the range in *Suffixes* on the first symbol using *Temp*
9. Write out branching nodes or leaf nodes to *Tree*
10. Push the nodes pointing to an unevaluated range onto the *Stack*
11. END

Figure 2: The TDD Algorithm

The pseudo-code for the PWOTD algorithm is shown in Figure 2. While the partitioning in phase one of PWOTD is simple enough, the algorithm for *wotdeager* in phase two warrants further discussion. We now illustrate the *wotdeager* algorithm using an example.

3.1.1 Example Illustrating the *wotdeager* Algorithm

The PWOTD algorithm requires four data structures for constructing suffix trees: an input string array, a suffix array, a temporary array, and the suffix tree. For the discussion that follows, we name each of these structures *String*, *Suffixes*, *Temp*, and *Tree*, respectively.

The Suffixes array is first populated with suffixes from a partition after discarding the first *prefixlen* characters. Using the same example string as before, ATTAGTACA$, consider the construction of the Suffixes array for the T-partition. The suffixes in this partition are at positions 1, 2, and 5. Since all these suffixes share the same prefix, T, we add one to each offset to produce the new Suffix array {2,3,6}. The next step involves sorting this array of suffixes based on the first character. The first characters of each suffix are {T, A, A}. The sorting is done using an efficient algorithm called *count-sort* in linear time (for a constant alphabet size). In a single pass, for each character

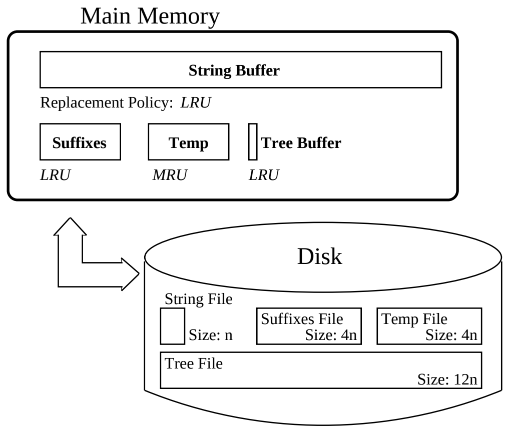

Figure 3: Buffer Management Schema

in the alphabet, we count the number of occurrences of that character in the first character of each suffix, and copy the suffix pointers into the Temp array. We see that the count for A is 2 and the count for T is 1; the counts for G, C, and $ are 0. We can use these counts to determine the character group boundaries: group A will start at position 0 with two entries, and group T will start at position 2 with 1 entry. We make a single pass through the Temp array and produce the Suffixes array sorted on the first character. The Suffixes array is now $\{3, 6, 2\}$. The A-group has two members and is therefore a branching node. These two suffixes completely determine the sub-tree below this node. Space is reserved in the Tree to write this non-leaf node once it is expanded, then the node is pushed onto the stack. Since the T-group has only one member, it is a leaf node and will be immediately written to the Tree. Since no other children need to be processed, no additional entries are added to the stack, and this node will be popped off first.

Once the node is popped off the stack, we find the longest common prefix (LCP) of all the nodes in the group $\{3, 6\}$. We examine position 4 (G) and position 7 (C) to determine that the LCP is 1. Each suffix pointer is incremented by the LCP, and the result is processed as before. The computation proceeds until all nodes have been expanded and the stack is empty. Figure 1 shows the complete suffix tree and its array representation.

3.1.2 Discussion of the PWOTD Algorithm

Observe that phase 2 of PWOTD operates on subsets of the suffixes of the string. In *wotdeager*, for a string of n symbols, the size of the Suffixes array and the Temp array needed to be $4 \times n$ bytes (assuming 4 byte integers are used as pointers). By partitioning in Phase 1, the amount of memory needed by the suffix arrays in each run is just $\frac{4 \times n}{|A|^{prefixlen}}$. This is an important point: partitioning decreases the main-memory requirements for suffix tree construction, allowing independent sub-tree to be built entirely in main memory. Suppose we are partitioning a 100 million symbol string over an alphabet of size 4. Using a $prefixlen = 2$ will decrease the space requirement of the Suffixes and Temp arrays from 400 MB to 25 MB each, and the Tree array from 1200 MB to 75 MB. Unfortunately, this savings is not entirely free. The cost to partition increases linearly with *prefixlen*. For small input strings where we have sufficient main memory for all the structures, we can skip the partitioning phase entirely. It is not necessary to continue partitioning once the Suffixes and Temp arrays fit into memory. For even very large datasets, such as the human genome, partitioning beyond 7 levels is not beneficial.

3.2 Buffer Management

Since suffix trees are an order of magnitude larger in size than the input data string, suffix tree construction algorithms require large amounts of memory, which may exceed the amount of main memory that is available. For such large data sets, efficient disk-based construction methods are needed that can scale well for large input sizes. One strength of TDD is that it transitions the data structures gracefully to disk as necessary, and uses individual buffer management polices for each structure. As a result, TDD can scale gracefully to handle large input sizes.

Recall that the PWOTD algorithm requires four data structures for constructing suffix trees: *String*, *Suffixes*, *Temp*, and *Tree*. Figure 3 shows each of these structures as separate, in-memory buffer caches. By appropriately allocating memory and by using the right buffer replacement policy for each structure, the TDD approach is able to build suffix trees on extremely large inputs. The buffer management policies are summarized in Figure 3 and are discussed in detail below.

The largest data structure is the Tree buffer. This array stores the suffix tree during its intermediate stages as well as the final computed result. The Tree data structure is typically 8-12 times the size of the input string. The reference pattern to Tree consists mainly of sequential writes when the children of a node are being recorded. Occasionally, pages are revisited when an unexpanded node is popped off the stack. This access pattern displays very good temporal and spatial locality. Clearly, the majority of this structure can be placed on disk and managed efficiently with a simple LRU (*L*east *R*ecently *U*sed) replacement policy.

The next largest data structures are the Suffixes and the Temp arrays. The Suffixes array is accessed as follows: first a sequential scan is used to copy the values into the Temp array. The sort operation following the scan causes random writes from the Temp array back into the Suffixes array. However, there is some locality in the pattern of writes, since the writes start at each character-group boundary and proceed sequentially to the right. Based on the (limited) locality of reference, one expects LRU to perform reasonably well.

During the sort, the Temp array is referenced in two linear scans: the first to copy all of the suffixes in the Suffixes array, and the second to copy all of them back into the Suffixes array in sorted order. For this reference pattern, replacing the most recently used page (MRU) works best.

The String array has the smallest main-memory requirement of all the data structures, but the worst locality of ac-

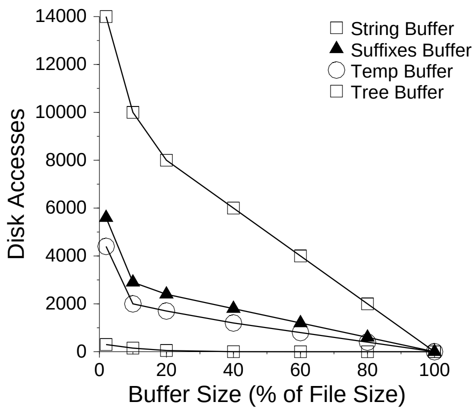

Figure 4: Sample Page Miss Curves

cess. The String array is referenced when performing the count-sort and to find the longest common prefix in each sorted group. During the count-sort all of the portions of the string referenced by the suffix pointers are accessed. Though these positions could be anywhere in the string, they are always accessed in left to right order. In the function to find the longest common prefix of a group, a similar pattern of reference is observed. In the case of the find-LCP function, each iteration will access the characters in the string, one symbol to the right of those previously referenced. In the case of the count-sort operation, the next set of suffixes to be sorted will be a subset of the current set. Based on these observations, one can conclude that the LRU policy would be the best management policy.

We summarize the choice of buffer management policies for each of the structures in Figure 3. As shown in the figure, the String, Suffixes, and Tree arrays should use the LRU replacement policy; the Temp array should use an MRU replacement policy. Based on experiments in Section 5.3, we confirm that these are indeed good choices.

3.3 Buffer Size Determination

To obtain the maximum benefit from buffer management policy, it is important to divide the available memory between the data structures appropriately. A careful apportioning of the available memory between these data structures can affect the overall execution time dramatically. In the rest of this section, we describe a technique to divide the available memory among the buffers.

If we know the access pattern for each of the data structures, we can devise an algorithm to partition the memory to minimize the overall number of buffer cache misses. Note that we only need an access pattern on a string representative of each class, such as DNA sequences, protein sequences, etc. In fact, we have found experimentally that these access patterns are similar across a wide-range of datasets (we discuss these results in detail in Section 5.3.) An illustrative graph of the buffer cache miss pattern for each data structure is shown in Figure 4. In this figure, the X-axis represents the number of pages allocated to the buffer as a percentage of the total size of the data structure. The Y-axis shows the number of cache misses. This figure is representative of biological sequences derived from actual experiments in Section 5.3.

As we will see at the end of section 3.3.1, our buffer allocation strategy only needs to estimate the relative magnitudes of the slopes of each curve, and the position of the "knee" towards the start of the curve. The full curve as shown in Figure 4 is not needed for the algorithm. However, it is useful to facilitate the following discussion.

3.3.1 TDD Heuristic for Allocating Buffers

We know from Figure 4 that the cache miss behavior for each buffer is approximately linear once the memory is allocated beyond a minimum point. Once we identify these points, we can allocate the minimum buffer size necessary for each structure. The remaining memory is then allocated in order of decreasing slopes of the buffer miss curves.

We know from arguments in Section 3.2 that references to the String have poor locality. One can infer that the String data structure is likely to require the most buffer space. We also know that the references to the Tree array have very good locality, so the buffer space it needs is likely to be a very small fraction of its full size. Between Suffixes and Temp, we know that the Temp array has more locality than the Suffixes array, and will therefore require less memory. Both Suffixes and Temp require a smaller fraction of their pages to be resident in the buffer cache when compared to the String. We exploit this behavior to design a heuristic for memory allotment.

We suggest the minimum number of pages allocated to the Temp and Suffixes arrays to be $|A|$. During the sort phase, we know that the Suffixes array will be accessed at $|A|$ different positions which correspond to the character group boundaries. The incremental benefit of adding a page will be very high until $|A|$ pages, and then one can expect to see a change in the slope at this point. By allocating at least $|A|$ pages, we avoid the penalty of operating in the initial high miss-rate region. The TDD heuristic chooses to allocate a minimum of $|A|$ pages to Suffixes and Temp first.

We suggest allocating two pages to the Tree array. Two pages allow a parent node, possibly written to a previous page and then pushed onto the stack for later processing, to be accessed without replacing the current active page. This saves a large amount of I/O over choosing a buffer size of only one page.

The remaining pages are allocated to the String array. If any pages are left over, they are allocated to Suffixes, Temp, and Tree, in that order.

The reasoning behind this heuristic is borne out by the graphs in Figure 4. The String, which has the least locality of reference, has the highest slope and the largest magnitude. Suffixes and Temp have a lower magnitude and a more gradual slope, indicating that the improvement with each additional page allocated is smaller. Finally, the Tree, which has excellent locality of reference, is nearly zero. All curves have a knee at the initial point which we estimate by choosing minimum allocations.

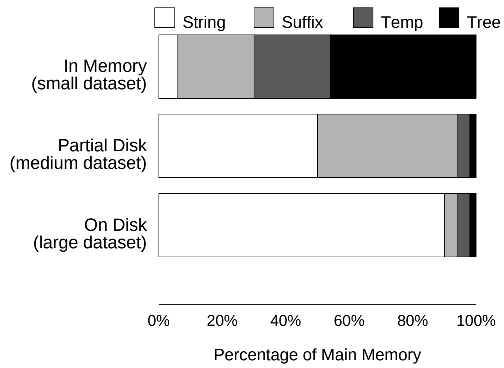

Figure 5: Scaling Buffer Allocation

3.3.2 An Example Allocation

The following example demonstrates how to allocate the main memory to the buffer caches. Assume that your system has 100 buffer pages available for use and that you are building a suffix tree on a small string that requires 6 pages. Further assume that the alphabet size is 4 and that 4 byte integers are used. Assuming that no partitioning is done, the Suffixes array will need 24 pages (one integer for each character in the String), the Temp array will need 24 pages, and the Tree will need at most 72 pages. First we allocate 4 pages each to Suffixes and Temp. We allocate 2 pages to Tree. We are now left with 90 pages. Of these, we allocate 6 pages to the String, thereby fitting it entirely in memory. From the remaining 84 pages, Suffixes and Temp are allocated 20 and fit into memory, and the final 44 pages are all given to Tree. This allocation is shown pictorially in the first row of Figure 5.

Similarly, the second row in Figure 5 is an allocation for a medium sized input of 50 pages. First, the heuristic allocates 4 pages each to Suffixes and Temp, and 2 pages to Tree. The String is given 50 pages. The remaining 40 pages are given to Suffixes, producing the second allocation in Figure 5. The third allocation corresponds to a large string of 120 pages. Here, Suffixes, Temp, and Tree are allocated their minimums of 4, 4, and 2 respectively, and the rest of the memory (90 pages) is given to String. Note that the entire string does not fit in memory now, and portions will be swapped into memory from disk when they are needed.

It is interesting to observe how the above heuristic allocates the memory as the size of the input string increases. This trend is indicated in Figure 5. When the input is small and all the structures fit into memory, most of the space is occupied by the largest data structure: the Tree. As the input size increases , the Tree is pushed out to disk. For very large strings that do not fit into memory, everything but the String is pushed out to disk, and the String is given nearly all of the memory. By first pushing the structures with better locality of reference onto disk, TDD is able to scale gracefully to very large input sizes.

Note that our heuristic does not need the actual utility curves to calculate the allotments. It estimates the “knee” of each curve using the algorithm, and assumes that the curve is linear for the rest of the region.

4 Analysis

In this section, we analyze the advantages and the disadvantages of using the TDD technique for various types and sizes of string data. We also describe how the design choices we have made in TDD overcome the performance bottlenecks present in other proposed techniques.

4.1 I/O Benefits

Unlike the approach of [4] where the authors use the best in-memory $O(n)$ algorithm (Ukkonen) as the basis for their disk-based algorithm, we use the theoretically less efficient $O(n^2)$ *wotdeager* algorithm [15]. A major difference between the two algorithms is that the Ukkonen algorithm sequentially accesses the string data and then updates the suffix tree through random traversals, while our TDD approach accesses the input string randomly and then writes the tree sequentially. For disk based construction algorithms, random access is the performance bottleneck as on each access an entire page will potentially have to be read from disk; therefore, efficient caching of the randomly accessed disk pages is critical.

On first appearance, it may seem that we are simply trading random disk I/Os for more random disk I/Os, but the input string is the smallest structure in the construction algorithm, while the suffix tree is the largest structure. TDD can place the suffix tree in very small buffer cache as the writes are almost entirely sequential, which leaves the remaining memory free to buffer the randomly accessed, but much smaller, input string. Therefore, our algorithm requires a much smaller buffer cache to contain the randomly accessed data. Conversely, for the same amount of buffer cache, we can cache much more of the randomly accessed pages, allowing us to construct suffix trees on much larger input strings.

4.2 Main-Memory Analysis

When we build suffix trees on small strings, where data structures fit in memory, no disk I/O is incurred. For the case of in-memory construction, one would expect that a linear time algorithm such as Ukkonen would perform better than the TDD approach which has an average case complexity of $O(nlog_{|A|}n)$. However, one must consider more than just the computational complexity to understand the execution time of the algorithms.

Traditionally, all accesses to main memory were considered equally good, as the disk I/O was the performance bottleneck. But, for programs that require little disk I/O, the performance bottleneck shifts into the main-memory hierarchy. Modern processors typically employ one or more data caches for improving access times to memory when there is a lot of spatial and/or temporal locality in the access patterns. The processor cache is analogous to a database's buffer cache, the primary difference being that the user does not have control over the replacement policy. Reading data from the processor's data cache is an order of magnitude faster than reading data from the main memory. And as the speed of the processor increases, so does the main-memory latency; as a result, the latency of random memory accesses will only grow with future processors.

Linear time algorithms such as Ukkonen require a large number of random memory accesses due to linked list traversals through the tree structure. A majority of cache misses occur after traversing a suffix link to a new sub-tree and then examining each child of the new parent. The traversal of the suffix link to the sibling sub-tree and the subsequent search of the destination node's children require random accesses to memory over a large address space. Because this span of memory is too large to fit in the processor cache, each access has a very high probability of incurring the full main-memory latency. Using an array based representation [21], where the pointers to the children are stored in an array with an element for each symbol in the alphabet, can reduce the number of cache misses. However, this representation uses up a lot more space and could therefore lead to a higher run time anyway.

Observe that as the alphabet size of the input string grows, the number of children for each non-leaf node will increase proportionately. As more children are examined to find the right position to insert the next character, the more cache misses will be incurred. Therefore, the Ukkonen method will incur an increasing number of processor cache misses with an increase in alphabet size.

For TDD, the alphabet size has the opposite effect. As the branching factor increases, the working set of the Suffixes and Temp arrays quickly decreases, and can fit into the processor cache sooner. The majority of read misses in the TDD algorithm occur when calculating the size of each character group (in Line 8 of Figure 2). This is because the beginning character of each suffix must be read, and there is little spatial locality in the reads. While both algorithms must perform random accesses to main memory which incur very expensive cache misses, there are three properties about the TDD algorithm that make it more suited for in-memory performance: (a) the access pattern is sequential through memory, (b) each random memory access is independent of the others accesses, and (c) the accesses are known a priori. Because the accesses to the input data string are sequential through the memory address space, hardware-based data prefetchers may be able to identify opportunities for prefetching the cache lines [19]. In addition, recently proposed techniques for overlapping execution with main-memory latency, such as software pipelining [7], can easily be incorporated in TDD.

4.3 Effect of Alphabet Size and Data Skew

There are two properties of the input string that can affect the execution time of suffix tree construction techniques: the size of the alphabet and the skew in the string. The average case running time for constructing a suffix tree on uniformly random input strings is $O(n \log_{|A|} n)$, where $|A|$ is the size of the input alphabet and n is the length of the input string. The intuition behind this average case time is as follows. There are $\log_{|A|} n$ levels in the tree, and at each level i the suffixes array is divided into $i^{|A|}$ parts. On each part, the count-sort and the find-LCP functions have to be run. The running time of count-sort is linear. To find the longest common prefix for a set of suffixes from a uniformly distributed string, the expected number of suffixes compared before a mismatch is slightly over 1. Therefore, the find-LCP function would return after just one or two comparisons most of the time. In some cases, the actual LCP is more than 1 and a scan of all suffixes is required. Therefore, in the case of uniformly distributed data, the find-LCP function is expected to run in constant time. This gives rise to the overall running time of $O(n \log_{|A|} n)$.

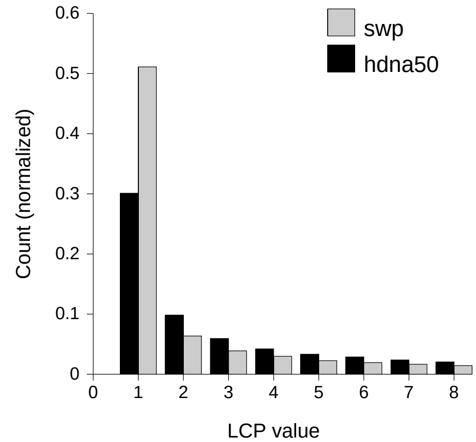

Figure 6: LCP Histogram

Interestingly, the longest common prefix is actually the label on the incoming edge for the node that corresponds to this range of suffixes. The average of all the LCPs computed while building a tree is equal to the average length of the labels on each edge ending in a non-leaf node.

Real datasets, such as DNA strings, have a skew that is particular to them. By nature, DNA often consists of large repeating sequences; different symbols occur with more or less the same frequency and certain patterns occur more frequently than others. As a result, the average LCP is higher than that for uniformly distributed data. Figure 6 shows a histogram for the longest common prefixes generated while constructing suffix trees on the SwissProt [2] and a 50 MB Human DNA sequence [13]. Notice that both sequences have a high probability that the LCP will be greater than 1. Even among biological datasets, the differences can be quite dramatic. From the figure, the DNA sequence is much more likely to have LCPs greater than 1 compared with the protein sequence (70% versus 50%). It is important to note that the LCP histograms for the DNA and protein sequences shown in the figure do not represent all strings, but these particular results do highlight the differences one can expect between input sets.

For data with a lot of repeating sequences, the find-LCP function will not be able to complete in a constant amount of time. It will have to scan at least the first l characters of all the suffixes in the range, where l is the actual LCP. In this case, the cost of find-LCP becomes $O(l \times r)$ where l is the actual LCP, and r is the number of suffixes in the range that the function is examining. As a result, the PWOTD algorithm will take longer to complete. However, note that the average case complexity remains $O(nlog_{|A|}n)$.

Inputs with a lot of repeated sequences, such as DNA, decrease the performance of TDD but may perform well for algorithms similar to Ukkonen's. Ukkonen's algorithm can exploit the repeated subsequences by terminating an insert phase when the duplicate suffix is already in the tree. This will happen more frequently in the case of input string like DNA which often have long repeating sequences, thereby providing a computational savings to the Ukkonen algorithm. Unfortunately, this advantage is offset by the random reference pattern which makes it a poor choice for larger input string on cached architectures.

The size of the input alphabet also has an important effect. Larger input alphabets are an advantage for TDD because the running time is $O(n \log_{|A|} n)$, where $|A|$ is the size of the alphabet. A larger input alphabet implies a larger branching factor for the suffix tree. This in turn implies that the working size of the Suffixes and Temp arrays shrinks more rapidly - and could fit into the cache entirely at a lower depth. For Ukkonen, a larger branching factor would imply that on an average, more siblings will have to be examined while searching for the right place to insert. This leads to a longer running time for Ukkonen. There are hash-based and array based approaches that alleviate this problem [21], but at the cost of consuming much more space for the tree. A larger representation naturally implies that we are limited to building trees on smaller strings. We experimentally demonstrate these effects in Section 5.

Note that the case where Ukkonen will have an advantage over TDD is for short input strings over a small alphabet with high skew (repeat sequences). TDD is a better bet in all other cases.

4.4 Summary of the Analysis

In this section, we discussed why the $O(n^2)$ construction algorithm used in the TDD technique is more amenable to disk-based suffix tree construction than the $O(n)$ algorithm of Ukkonen. Because the PWOTD algorithm trades random accesses into the input string (of size n) for sequential accesses into the Tree data structure (of size $12n$), we can manage the Tree structure with only a fraction of the main memory required by other techniques. This property provides a fundamental advantage over other disk-based approaches since our disk I/O performance is primarily dependent on the smallest data structure, instead of being dependent on the largest data structure as is the case with other techniques.

We also argued that even for small strings where all the structures fit into main memory, using an $O(n)$ algorithm like Ukkonen might not be the best choice. The behavior of the algorithm with respect to the processor caches is also important, and as we show later in Section 5, TDD outperforms existing methods even for the in-memory case.

Finally, we explored the effects of alphabet size and the skew in the input string on TDD. We argue that TDD performs better on larger alphabet sizes, and is disadvantaged by skew in the string. Algorithms like Ukkonen on the other hand are poor for larger alphabet sizes and have an advantage for skewed data. Again, we point to Section 5 for an experimental verification of these claims.

5 Experimental Evaluation

In this section, we present the results of an extensive experimental evaluation of the different suffix tree construction techniques. In addition to TDD, we compare Ukkonen's algorithm [32] for in-memory construction performance, and Hunt's algorithm [18] for disk-based construction performance. Ukkonen's and Hunt's algorithms are considered the best known suffix tree construction algorithms for the in-memory case and the disk based case respectively.

5.1 Experimental Implementation

Our TDD algorithm uses separate buffer caches for the four main structures: the string, the suffixes array, the temporary working space for the count sort, and the suffix tree. We use fixed-size pages of 8K for reading and writing to disk. Buffer allocation for TDD is done using the method described in Section 3.3. If the amount of memory required is less than the size of the buffer cache, then that structure is loaded into the cache, with accesses to the data bypassing the buffer cache logic. TDD was written in C++ and compiled with GNU's g++ compiler version 3.2.2 with full optimizations activated.

For an implementation of the Ukkonen's algorithm, we use the version from [34]. It is a textbook implementation of Ukkonen's algorithm based on Gusfield's description [17] and written in C. The algorithm operates entirely in main memory, and there is no persistence. The representation uses 32 bytes per node.

Our implementation of Hunt's algorithm is from the OASIS search tool [25], which is part of the Periscope project [27]. The OASIS implementation uses a shared buffer cache instead of the persistent Java object store, PJama [3], described in the original proposal [18]. The buffer manager employs the CLOCK replacement policy. The OASIS implementation performed better than the implementation described in [18]. This is not surprising since PJama incurs the overhead of running through the Java Virtual Machine.

For the disk based experiments that follow, unless stated otherwise, all I/O is to raw devices; i.e., there is no buffering of I/O by the operating system and all reads and writes to disk are synchronous (blocking). This provides an unbiased accounting of the performance for disk based construction as operating system buffering will not (positively) affect the performance. Therefore, our results present the worst case performance of disk based construction. Using asynchronous writes is expected to improve the performance of our algorithm over the results presented. Each raw device accesses a single partition on one Maxtor Atlas

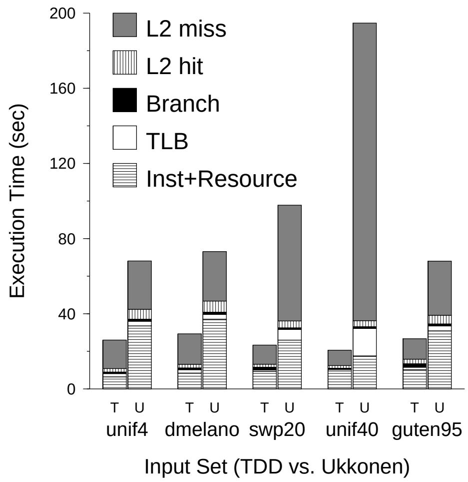

Figure 7: Execution Time Breakdown

10K IV drive. The disk drive controller is an LSI 53C1030, Ultra 320 SCSI controller.

The experiments were performed on an Intel Pentium 4 processor with 2.8 GHz clock speed and 2 GB of main memory. This processor includes a two level cache hierarchy. There are two first level caches, named L1-I and L1-D, that cache instructions and data respectively. There is also a single L2 cache that stores both instructions and data. The L1 data cache is an 8 KB, 4-way set-associative cache with a 64 byte line size. The L1 instruction cache is a 12 K trace cache, 4-way set associative. The L2 cache is a 512 KB, 8-way, set-associative cache, also with a 128 byte line size. The operating system was Linux, kernel version 2.4.20.

The Pentium 4 processor includes 18 event counters that are available for recording micro-architectural events, such as the number of instructions executed [20]. To access the event counters, the *perfctr* library was used [28]. The events measured include: clock cycles executed, instructions and micro-operations executed, L2 cache accesses and misses, TLB misses, and branch mispredictions.

5.2 Comparison of In-Memory Algorithms

To evaluate the performance of the TDD technique for in-memory construction, we chose to compare with the performance of the $O(n)$ time Ukkonen's algorithm. We do not evaluate Hunt's algorithm in this section as it was not designed as an in-memory technique. For this experiment, we used five different data sources : chromosome 2 of Drosophila Melanogaster from GenBank [13], a slice of the SwissProt dataset [2] having 20 million symbols, and the text from the 1995 collection from project Gutenberg [29]. We also chose two strings that contain uniformly distributed symbols from an alphabet of size four and forty. This data is summarized in Table 1.

Figure 7 shows the execution time breakdown for both algorithms, grouped by data source with TDD performance on the left and Ukkonen performance on the right. Note that since this is the in-memory case, TDD reduces to

Data Source	**Description**	**Symbols** (10^6)
dmelano	D.Melanogaster Chr. 2 (DNA)	20
guten95	Gutenberg Project, Year 1995 (English Text)	20
swp20	Slice of SwissProt (Protein)	20
unif4	4-char alphabet, uniform distrib.	20
unif40	40-char alphabet, uniform distrib.	20

Table 1: Main Memory Data Sources

just the PWOTD algorithm. In these experiments, all data structures fit into memory. The total execution time is decomposed into the time executing the following micro-architectural events (from bottom to top): instructions executed plus resource related stalls, TLB misses, branch mispredictions, L2 cache hits, and L2 cache misses (or main-memory reads).

From Figure 7, the L2 cache miss component is a large contributor to the execution time for both algorithms. Both algorithms show a similar breakdown for the small alphabet sizes of DNA data (unif4 and dmelano). When the alphabet size increases from 4 symbols to 20 symbols for SwissProt and to 40 symbols for unif40, the cache miss component of Ukkonen's algorithm increases dramatically while the cache miss component for the TDD algorithm remains low. The reason for this, as discussed in Section 4.2, is that Ukkonen's algorithm incurs a lot of cache misses while following the suffix link to a new portion of the tree, and traversing all the children when trying to find the right position to insert the new entry.

We observe that for each dataset, TDD outperforms Ukkonen's algorithm and the performance difference increases with alphabet size. This was expected based on discussions in Section 4.3. For instance, on the DNA dataset of *dmelano* ($|A| = 4$) , TDD is faster than Ukkonen by a factor of 2.5. For the *swp20* protein dataset ($|A| = 20$), TDD is faster by a factor of 4.5. Finally, for the *unif40* ($|A| = 40$), TDD is faster by a factor of 10! These results demonstrate that, despite having a $O(n^2)$ time complexity, the TDD technique significantly outperforms Ukkonen's algorithm on cached architectures.

5.3 Buffer Management with TDD

In this section we evaluate the effectiveness of various buffer management policies . For each data structure used in the TDD algorithm, we analyze the performance of the LRU, MRU, RANDOM, and CLOCK page replacement polices over a wide range of buffer cache sizes. To facilitate this analysis over the wide range of variables, we employed a buffer cache simulator. The simulator takes as input a trace of the address requests into the buffer cache and the page size. The simulator outputs the disk I/O statistics for the desired replacement policy. For all data shown here except the Temp array, MRU performs the worst by far and is not shown in the figures that we present in this section.

To generate the traces of address requests, we built suffix

Data Structure	SwissProt (size in pages)	Human DNA (size in pages)
String	6,250	6,250
Suffixes	1,250	6,250
Temp	1,250	6,250
Tree	4,100	16,200

Table 2: Array Sizes

trees on the *SwissProt* database [2] and a 50 Mbps slice of the *Human Chromosome*-1 database [13]. A *prefixlen* of 1 was used for partitioning in the first phase. The size of each of the arrays for these datasets is summarized in Table 2.

5.3.1 Page Size

In order to determine the page size to use for the buffers, we conducted several experiments. We observed that larger page sizes produced fewer page misses when the alphabet size was large (protein datasets, for instance). Smaller page sizes seemed to have a slight advantage in the case of input sets with smaller alphabets (like DNA sequences). We observed that a page size of 8192 bytes performed well for a wide range of alphabet sizes. In the interest of space, we omit the details of our page-size study. For all the experiments described in this section we use a page size of 8KB.

5.3.2 Buffer Replacement Policy

The results showing the effect of the various buffer replacement policies for the four data structures are shown in Figures 8 to 11. In these figures, the x-axis is the buffer size (shown as a percentage of the original input string size), and the y-axis is the number of buffer misses that are incurred by various replacement policies.

From Figure 8, we observe that for the String buffer LRU, RANDOM, and CLOCK all perform similarly. In fact, RANDOM has a very small advantage over the other two because there is very limited locality in the reference pattern in the string. Of all the arrays, when the buffer size is a fixed fraction of the total size of the structure, the String incurs the largest number of page misses. This is not surprising since this structure is accessed the most and in a random fashion.

In the case of the Suffixes buffer (shown in Figure 9), all three policies perform similarly for small buffer sizes. In the case of the Temp buffer, the reference pattern consists of one linear scan from left to right to copy the suffixes from the Suffixes array, and then another scan from left to right to copy the suffixes back into the Suffixes array in the sorted order. Clearly, MRU is the best policy in this case as shown by the results in Figure 10. It is interesting to observe that the space required by the Temp buffer is much smaller than the space required by the Suffixes buffer to keep the number of misses down to the same level, though the array sizes are the same.

For the Tree buffer (see Figure 11), with very small buffer sizes, LRU and CLOCK outperform RANDOM. However, this advantage is lost for even moderate buffer sizes. The most important fact here is that despite being the largest data structure, it requires the smallest amount of buffer space, and takes a relatively insignificant number of misses for any policy. Therefore for the Tree buffer, we can choose to implement the cheapest policy - the random replacement policy.

Data Source	Description	Symbols (10^6)
swp	Entire UniProt/SwissProt (Protein)	53
H.Chr1-50	50 Mbps slice of Human Chromosome-1 (DNA)	50
guten03	2003 Directory of Gutenberg Project (English Text)	58
trembl	TrEMBL (Protein)	338
H.Chr1	Entire Human Chromosome-1 (DNA)	227
guten	Entire Gutenberg Collection (English Text)	407
HG	Entire Human Genome (DNA)	$3,000$

Table 3: On-Disk Data Sources

Data Source	Symbols (10^6)	Hunt (min)	TDD (min)	Speedup
swp	53	13.95	2.78	5.0
H.Chr1-50	50	11.47	2.02	5.7
guten03	58	22.5	6.03	3.7
trembl	338	236.7	32.00	7.4
H.Chr1	227	97.50	17.83	5.5
guten	407	463.3	46.67	9.9
HG	$3,000$	—	30hrs	—

Table 4: Performance Comparison

5.4 Comparison of Disk-based Algorithms

In this section we first compare the performance of our technique with the technique proposed by Hunt et al. [18], which is currently considered the best disk-based suffix tree construction algorithm. For this experiment, we used seven datasets which are described in Table 3. The suffix tree construction times for the two algorithms are shown in Table 4.

From this table, we see that in each case TDD performs significantly better than Hunt's algorithm. For example, on the TrEMBL database, ($|A| = 20$), TDD is faster by a factor of 7.4. For Human Chromosome-1 ($|A| = 4$), TDD is faster by a factor of 5.5. For a large text dataset like the Gutenberg Collection ($|A| = 60$), TDD is nearly ten times faster! For the largest dataset, the human genome, Hunt's algorithm did not complete in a reasonable amount of time. The reason why TDD performs better is that Hunt's algorithm traverses the on-disk tree during construction, while TDD does not. During construction, a given node in the tree is written over at most once. By careful management of the buffer sizes, and the buffer replacement policies, the disk I/O in TDD is brought down further.

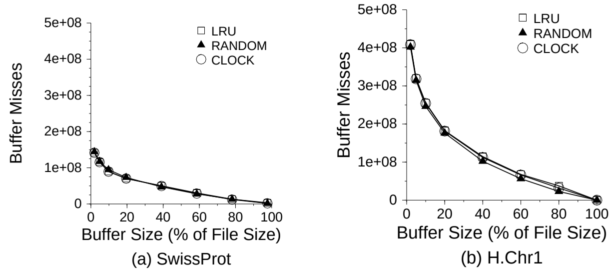

Figure 8: String Buffer

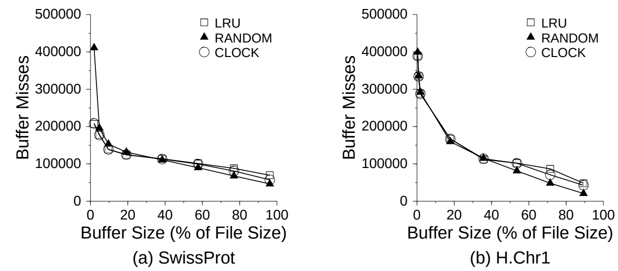

Figure 9: Suffix Buffer

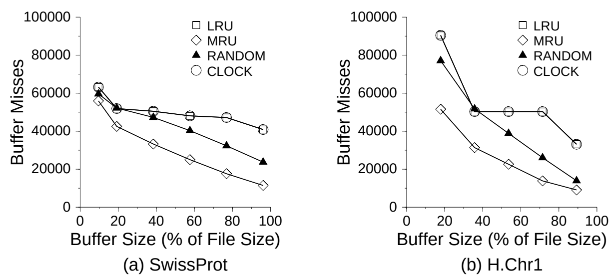

Figure 10: Temp Buffer

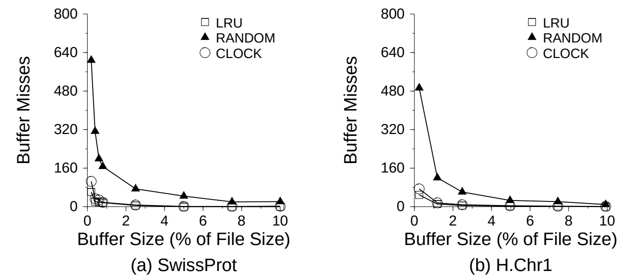

Figure 11: Tree Buffer

Comparison of TDD with TOP-Q

Very recently, Bedathur and Haritsa have proposed the TOP-Q technique for constructing suffix trees [4]. TOP-Q is a new low overhead buffer management method which can be used with Ukkonen's construction algorithm. The goal of these researchers is to invent a buffer management technique that does not require modifying an existing in-memory construction algorithm. In contrast, TDD and Hunt's algorithm [18] take the approach of modifying existing suffix tree construction algorithms to produce a new disk-based suffix tree construction algorithm. Even though the research focus of TOP-Q is different from TDD and Hunt's algorithm, it is natural to ask how the TOP-Q method compares to these other approaches.

To compare TDD with TOP-Q, we obtained a copy of the TOP-Q code from the authors. This version of the code only supports building suffix tree indices on DNA sequences. As per the recommendation in [4], we used a buffer pool of 880M for the internal nodes and 800M for the leaf nodes (this was the maximum memory allocation possible with the TOP-Q code). On 50Mbp of Human Chromosome-1, TOP-Q took about 78 minutes. By contrast, under the same conditions, TDD took about 2.1 minutes: faster by a factor of 37. On the entire Human Chromosome-1, TOP-Q took 5800 minutes, while our approach takes around 18 minutes. In this case, TDD is faster by two orders of magnitude!

6 Conclusions and Future Work

Suffix tree construction on large character sequences has been virtually intractable. Existing approaches have excessive memory requirements and poor locality of reference and therefore do not scale well for even moderately sized datasets.

To address these problems and unlock the potential of this powerful indexing structure, we have introduced the "Top Down Disk-based" (TDD) technique for disk-based suffix tree construction. The TDD technique includes a suffix tree construction algorithm (PWOTD), and an accompanying buffer cache management strategy. We demonstrate that PWOTD has an advantage over Ukkonen's algorithm by a factor of 2.5 to 10 for in-memory datasets.

Extensive experimental evaluations show that TDD scales gracefully as the dataset size increases. The TDD approach lets us build suffix trees on large frequently used sequence datasets such as UniProt/TrEMBL [2] in a few minutes. Algorithms to construct suffix trees on this scale (to our knowledge) have not been mentioned in literature before. The TDD approach outperforms a popular disk-based suffix tree construction method (the Hunt's algorithm) by a factor of 5 to 10. In fact, to demonstrate the strength of TDD, we show that using slightly more main-memory than the input string, a suffix tree can be constructed on the *entire Human Genome in 30 hours on a single processor machine!* These input sizes are one or two orders of magnitude larger than the datasets that have been used in previously published approaches.

Others researchers have proposed buffer management strategies for on-disk suffix tree construction, but our method is unique in that the larger data structures that are required during the suffix tree construction can be accessed efficiently even with a small number of buffer pages. This behavior leads to the highly scalable aspect of TDD.

As part of our future work, we plan on making TDD more amenable to parallel execution. We believe that the

TDD technique is extremely parallelizable due to the partitioning phase that it employs. Each partition is the source for an independent subtree of the complete suffix tree. Since the partitions are independent, multiple processors can simultaneously construct the sub-trees.

7 Acknowledgements

This research was supported by the National Science Foundation under grant IIS-0093059, and by research gift donations from IBM and Microsoft. We would like to thank the reviewers of VLDB and Ela Hunt for their valuable comments on earlier drafts of this paper. We would also like to thank Srikanta J. Bedathur and Jayant Haritsa for providing us a copy of their TOP-Q code.

References

[1] A. Andersson and S. Nilsson. Efficient Implementation of Suffix Trees. *Software–Practice and Experience (SPE)*, 25(2):129–141, 1995.

[2] R. Apweiler, A. Bairoch, C. H. Wu, W. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, R. Lopez, M. Magrane, M. J. Martin, D. Natale, A. C. O'Donovan, N. Redaschi, and L. L. Yeh. UniProt: the Universal Protein Knowledgebase. *Nucleic Acids Research*, 32(D):115–119, 2004.

[3] M. Atkinson and M. Jordan. Providing Orthogonal Persistence for Java. In *European Conference on Object-Oriented Programming (ECOOP)*, 1998.

[4] S. J. Bedathur and J. R. Haritsa. Engineering a Fast Online Persistent Suffix Tree Construction. In *ICDE*, 2004.

[5] A. Blumer, A. Ehrenfeucht, and D. Haussler. Average Sizes of Suffix Trees and DAWGs. *Discrete Applied Mathematics*, 24(1):37–45, 1989.

[6] A. Carvalho, A. Freitas, A. Oliveira, and M.-F. Sagot. A Parallel Algorithm for the Extraction of Structured Motifs. In *ACM Symposium on Applied Computing*, 2004.

[7] S. Chen, A. Ailamaki, P. Gibbons, and T. Mowry. Improving Hash Join Performance through Prefetching. In *ICDE*, 2004.

[8] L.-L. Cheng, D. Cheung, and S.-M. Yiu. Approximate String Matching in DNA Sequences. In *Proceeings of the Eighth International Conference on Database Systems for Advanced Applications*, pages 303–310, 2003.

[9] A. Crauser and P. Ferragina. A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory and its Applications. *Algorithmica*, 32(1):1–35, 2002.

[10] A. Delcher, S. Kasif, R. Fleischmann, J. Peterson, O. White, and S. Salzberg. Alignment of Whole Genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[11] A. Delcher, A. Phillippy, J. Carlton, and S. Salzberg. Fast Algorithms for Large-scale Genome Alignment and Comparision. *Nucleic Acids Research*, 30(11):2478–2483, 2002.

[12] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the Sorting-complexity of Suffix tree Construction. *J. ACM*, 47(6):987–1011, 2000.

[13] GenBank, NCBI, 2004. `www.ncbi.nlm.nih.gov/GenBank`.

[14] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-time Suffix Tree Construction. *Algorithmica*, 19(3):331–353, 1997.

[15] R. Giegerich, S. Kurtz, and J. Stoye. Efficient Implementation of Lazy Suffix Trees. In *Proceedings of the Third Workshop on Algorithm Engineering (WAE'99)*, 1999.

[16] D. Gusfield. An "Increment-by-one" Approach to Suffix Arrays and Trees. *Technical Report CSE-90-39, Computer Science Division, University of California, Davis*, 1990.

[17] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[18] E. Hunt, M. P. Atkinson, and R. W. Irving. A Database Index to Large Biological Sequences. *The VLDB J.*, 7(3):139–148, 2001.

[19] Intel Corporation. *The IA-32 Intel Architecture Optimization Reference Manual*. Intel (Order Number 248966), 2004.

[20] Intel Corporation. *The IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*. Intel (Order Number 253668), 2004.

[21] S. Kurtz. Reducing Space Requirement of Suffix Trees. *Software Practice and Experience*, 29(13):1149–1171, 1999.

[22] S. Kurtz, A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. Salzberg. Versatile and Open Software for Comparing Large Genomes. *Genome Biology*, 5(R12), 2004.

[23] S. Kurtz and C. Schleiermacher. REPuter: Fast Computation of Maximal Repeats in Complete Genomes. *Bioinformatics*, 15(5):426–427, 1999.

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

[25] C. Meek, J. M. Patel, and S. Kasetty. OASIS: An Online and Accurate Technique for Local-alignment Searches on Biological Sequences. In *VLDB*, 2003.

[26] G. Navarro, R. Baeza-Yates, and J. Tariho. Indexing Methods for Approximate String Matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.

[27] J. M. Patel. The Role of Declarative Querying in Bioinformatics. *OMICS*, 7(1):89–92, 2003.

[28] M. Pettersson. Perfctr: Linux Performance Montioring Counters Driver, `user.it.uu.se/~mikpe/linux/perfctr`.

[29] Project Gutenberg, `www.gutenberg.net`.

[30] W. Szpankowski. *Average-Case Analysis of Algorithms on Sequences*. John Wiley and Sons, 2001.

[31] The Growth of GenBank, NCBI, 2004. `www.ncbi.nlm.nih.gov/genbank/genbankstats.html`.

[32] E. Ukkonen. Constructing Suffix-trees On-Line in Linear Time. *Algorithms, Software, Architecture: Information Processing*, 1(92):484–92, 1992.

[33] P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, 1973.

[34] S. Yona and D. Tsadok. ANSI C Implementation of a Suffix Tree, `cs.haifa.ac.il/~shlomo/suffix_tree`.