Engineering a Fast Online Persistent Suffix Tree Construction

Srikanta J. Bedathur
Database Systems Lab, SERC
Indian Institute of Science
Bangalore 560012, India
srikanta@dsl.serc.iisc.ernet.in

Jayant R. Haritsa
Database Systems Lab, SERC
Indian Institute of Science
Bangalore 560012, India
haritsa@dsl.serc.iisc.ernet.in

Abstract

Online persistent suffix tree construction has been considered impractical due to its excessive I/O costs. However, these prior studies have not taken into account the effects of the buffer management policy and the internal node structure of the suffix tree on I/O behavior of construction and subsequent retrievals over the tree. In this paper, we study these two issues in detail in the context of large genomic DNA and Protein sequences. In particular, we make the following contributions: (i) a novel, low-overhead buffering policy called TOP-Q which improves the on-disk behavior of suffix tree construction and subsequent retrievals, and (ii) empirical evidence that the space efficient linked-list representation of suffix tree nodes provides significantly inferior performance when compared to the array representation. These results demonstrate that a careful choice of implementation strategies can make online persistent suffix tree construction considerably more scalable – in terms of length of sequences indexed with a fixed memory budget, than currently perceived.

1. Introduction

The suffix tree is a versatile datastructure, that is used in numerous bioinformatics applications (see [10, 11] for a comprehensive list of these applications). For example, they are extensively used in many tasks requiring similarity searching over genetic sequences such as DNA and Proteins.

Although the utility of suffix trees is well known, their usage is limited to small length datasets due to their space requirements – the best in-memory implementations so far take upto 12.5 times the database size [19]. This is in marked contrast to traditional index structures (e.g., B^+-trees), where the size of the index is usually much smaller than the indexed database.

Thus, it becomes untenable to consider a suffix tree residing fully in memory, indexing an ever growing sequence corpus such as the GenBank maintained by NCBI[1]. An obvious solution to handle this space problem is to maintain the suffix tree index on disk. Unfortunately, due to seemingly random traversals induced by the linear-time construction algorithms, resulting in unacceptably high I/O costs, the folk wisdom is that disk based implementations of suffix trees are unviable [8].

In order to overcome this infamous "memory bottleneck" [13] of persistent suffix tree construction, there are two possible approaches: (a) The suffix tree construction algorithm and its structure could be modified to make it more suitable for on-disk implementation, or (b) Tune the parameters of the environment in which suffix tree is implemented, *without modifying either the structure or the construction algorithm*.

In a recent work, Hunt et al. [4], took the former approach wherein they completely abandoned the use of *suffix links* – additional edges over the internal nodes of the suffix tree that are crucial in obtaining linear-time construction of suffix tree. Using the resulting batch-wise construction with quadratic worst case time complexity, they showed that suffix trees can be built efficiently on-disk. However, due to the resulting structure without suffix links, some of the fast approximate string processing algorithms that make use of suffix links, such as computing *matching statistics* [22], are rendered unusable.

In this paper, we take the second approach, and identify the parameters that affect online persistent suffix tree construction and quantify their impact. Specifically, the contributions of this paper are threefold:

1. We propose a novel paging strategy, TOP-Q, that takes into account the probabilistic behavior of traversals during suffix tree construction. This strategy uses only the path length invariant (defined later) of suffix tree

1 As on January 2003, NCBI-GenBank houses about 28.5Gbp of DNA sequence data [15]

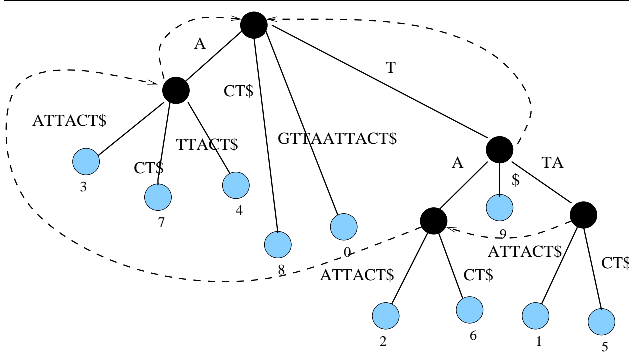

Figure 1. Suffix tree for the DNA fragment "GTTAATTACT$"

nodes and results in a *static* policy, with extremely low computational overhead.

2. We study a variety of paging policies in terms of their utilization of buffer space during the persistent suffix tree construction and compare their performance with that of TOP-Q.
3. We show that the much preferred suffix tree implementation using linked-list representation of nodes [19, 3], is extremely expensive in terms of disk I/O in spite of its space economy, in comparison to a simple and often neglected array representation of edges.

In our experiments, we use the classical online suffix tree construction algorithm of Ukkonen [7]. Our evaluation testbed consists of a variety of real DNA sequences, a synthetic symmetric Bernoulli sequence over a 4 character alphabet, as well as a subset of the SWISS-PROT protein database [20].

2. Persistent Suffix Tree Construction

In this section, we provide an overview of the online suffix construction algorithm of Ukkonen, detailed in [10], and consider the pattern of node accesses made during the algorithm's execution.

To aid the presentation, we first present some terminology related to suffix trees. Let $S[0..N]$ be the string indexed by the tree T_S. The leaf node corresponding to the i-th suffix, $S[i..N]$, is represented as l_i. An internal node, v, has an associated length $L(v)$, which is the sum of edge lengths on the path from root to v. We represent by $\sigma(v)$, the string at v to represent the substring $S[i..i + L(v)]$ where l_i is any leaf under v. A suffix link $sl(v) = w$ exists for every node v in the suffix tree such that if $\sigma(v) = a\alpha$, then $\sigma(w) = \alpha$, where a is a single character of the alphabet and α is a substring (possibly null) of the string. Note that $sl(v)$ is defined for *every* node in the suffix tree. And, more importantly, $sl(.)$ – the entire set of suffix links, forms a tree rooted at the root of T_S, with the depth of any node v in this $sl(.)$ tree being $L(v)$. The suffix tree for an example 10-letter DNA subsequence, S = "GTTAATTACT$" is shown in Figure 1. The numbers at the bottom of leaf nodes represent the start of the suffix $S[i..N]$ that they represent. The dashed edges between internal nodes represent the suffix links.

The suffix tree is constructed incrementally by scanning the string from left to right, one character at a time. A high level description of the construction process is given in Algorithm 1. The algorithm can be viewed to consist of two phases, *Locate phase* and *Insert phase*, for each character in the sequence. If implemented naively, the locate-phase would have quadratic time complexity, resulting in a overall $O(n^3)$ time complexity. Therefore, the tree is augmented with additional edges, called *suffix links*, that provide shortcuts to move *across* the tree quickly. These suffix links play a crucial role in reducing the running time of the algorithm to linear in the length of the indexed string.

Algorithm 1 Online Algorithm of Ukkonen

```
1:  procedure Ukkonen {Outputs an implicit suffix tree}
2:  input S[0..m] : string to be indexed
3:  I_0 ← Implicit suffix tree for S[0...0]
4:  for i = 0 to m do
5:    for j = 0 to i + 1 do
6:      {LOCATE PHASE}
7:      Locate β = S[j...i] in I_i
8:      {INSERT PHASE}
9:      if β ends at a leaf then
10:       I_{i+1} ← add S[i + 1] to I_i
11:     else {β ends at an internal node, or the middle of
        the edge}
12:       if from the end of β there is no path labeled
          S[i + 1] then
13:         I_{i+1} ← split edge in I_i and add a new leaf
14:       else
15:         I_{i+1} ← I_i {β already exists in I_i}
16:       end if
17:     end if
18:   end for
19: end for
```

However, when we move the suffix tree construction from memory to disk, these linear bounds no longer reflect reality, since they were obtained with a RAM machine model, where every memory access has the same cost, *irrespective* of its address. On the other hand, access-cost in secondary memory is dependent on the address to which the previous access was made. For example, a long chain of accesses to spatially contiguous addresses (block accesses) could cost much less than fewer but random accesses.

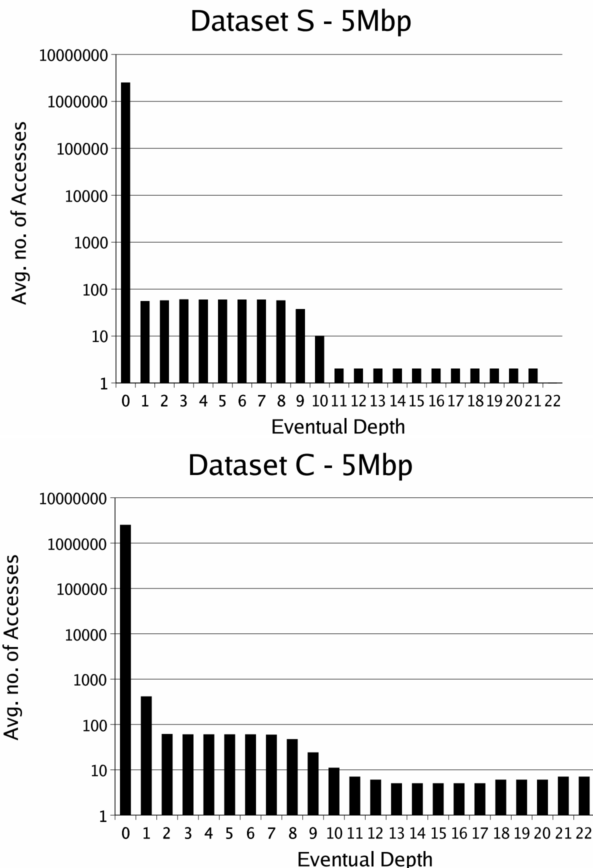

Figure 2. Node Access Frequency

2.1. Node Access Patterns

During the construction of a suffix tree, accesses to nodes are spatially non-contiguous. Specifically, in the locate-phase, already constructed parts of the tree are re-accessed many times via suffix links. These traversals are not necessarily spatially local, leading to seemingly random traversals over the tree. The following words of Giegerich and Kurtz [17], typifies the behavior of these algorithms: *"The active suffix creeps through the text like a caterpillar. At the same time, the corresponding active node swings through the tree like a butterfly"*.

Thus, it is strongly believed that the accesses are random in nature – with no obvious useful patterns discernible from the access traces.

3. Locating Preferred Nodes

In this section, we closely analyse the traces of accesses to nodes during online suffix tree construction, show that some of the nodes are indeed accessed far more frequently than others, and provide a simple observation that helps to identify such nodes during construction of the tree.

Before we proceed to analyse the traces, we note that the nature of accesses that are expected during the construction of the suffix tree is intricately linked to the stochastic properties of the specific sequence at hand. There have been many efforts to classify the sequences based on their stochastic properties [23]. One of the simplest sequence models that is proposed as an approximation to genome sequences is that of *Bernoulli generators*. In this model, symbols of the alphabet are drawn independently of one another; thus a string can be described as the outcome of a sequence of Bernoulli trials. In addition, if all symbols are drawn with equal probability, then the sequence is called *symmetric*, otherwise, it is *asymmetric*.

Now, consider the internal node access statistics during suffix tree construction for a symmetric Bernoulli sequence (dataset S) derived from the access traces, shown in Figure 2. These provide the correlation between the average number of accesses made to a node and the eventual depth of the node in the tree, illustrating that, during the suffix tree construction, *nodes higher up in the tree are accessed more number of times than nodes lower in the tree*. This correlation is also evident for suffix tree construction over real chromosomal sequences (dataset C) as shown in Figure 2.

Thus, it seems reasonable to cache the nodes that end up higher in the tree, in order to serve these accesses faster. However, due to the nature of the edge splits during construction, the depth of a node cannot be maintained without propagating the update throughout the subtree under the node.

3.1. Estimating the Depth of Internal Nodes

Although the depth of internal nodes cannot be maintained accurately, a simple observation on the structure of the resulting suffix tree provides us with a means to *estimate* this value efficiently. Considering sequences drawn from a symmetric Bernoulli stochastic models, it is straightforward to see that:

- Substrings of equal length are equally likely.
- If s is a substring of the sequence S, with $|s| = l$, then the number of substrings occurring elsewhere in S which have a common prefix with s is directly proportional to l.

Applying these to the behavior of the suffix tree during its construction, we get:

Observation 1 *The longer the edge in suffix tree of a symmetric Bernoulli sequence, more the likelihood of its being split in the limiting sense.*

And obviously, no edge can be split once it has reached the limiting minimum length of 1.

From the above observation, we can infer that a node, in the limiting sense, can move further down in the suffix tree until its incoming edge has only one symbol as its label. Thus $L(.)$ of any internal node forms an *upper bound* on its eventual depth. In addition, this measure of path length is an *invariant* for the node, which results in easy maintenance of this information during the construction of the suffix tree. Hence, for suffix tree nodes over large sequences, *we can approximate their eventual depth in the tree by their path length.*

3.2. Impact of Asymmetric Distribution

In deriving the above approximation to the eventual depth of a node, we made the assumption that the sequence is drawn from a symmetric Bernoulli model. However, it is unlikely that any real-world DNA sequence would conform to this restrictive model. The asymmetry of real-life distribution results in substrings containing a larger proportion of frequent symbols, having higher probability of occurrence than other substrings of the same length. This implies that if a node v has its label $\sigma(v)$ containing smaller number of frequent symbols, it has lower probability of being split – resulting in a possible over-estimation of its eventual depth by its path length $L(v)$.

The graphs in Figure 3 show, for four DNA datasets used in our experiments, the error in estimation of eventual depth of a node vis-a-vis the actual depth of the node, as well as the corresponding number of nodes at each depth of the tree. These graphs were obtained after processing 5Mbp of each of the datasets. The figure shows the statistics for nodes only upto a depth of 20, since the number of the internal nodes at depths greater than 20 is too small to make any impact although the error in estimation of eventual depth is quite large for such nodes.

The average error at each depth was computed as the arithmetic mean of $(L(.) - Depth(.))$ for all nodes at that depth. Note that since $L(.)$ forms an upperbound on the depth of the node, this value is always positive. The following points have to be noted with respect to these graphs:

- For all the datasets, path length corresponds *exactly* to the depth of the node up to a certain value of actual depth, which is dependent on the length of the sequence processed. As shown in the graphs, this value is 10, after processing 5Mbp of each of the datasets.
- The number of nodes peaks at a depth of 11, at which point the error in the estimation of depth is small for all the datasets.
- For the dataset S, estimates are accurate throughout providing empirical evidence for the soundness of Observation 1.
- The worst estimation error are with dataset C, which has a highly skewed distribution of basepairs. However, a majority of nodes in the tree occur within depth 14, where errors are not too large.
- The depth of a node approaches its path length with the increase in length of the sequence indexed. Therefore, the estimation errors continue to decrease as we process sequences of greater length.

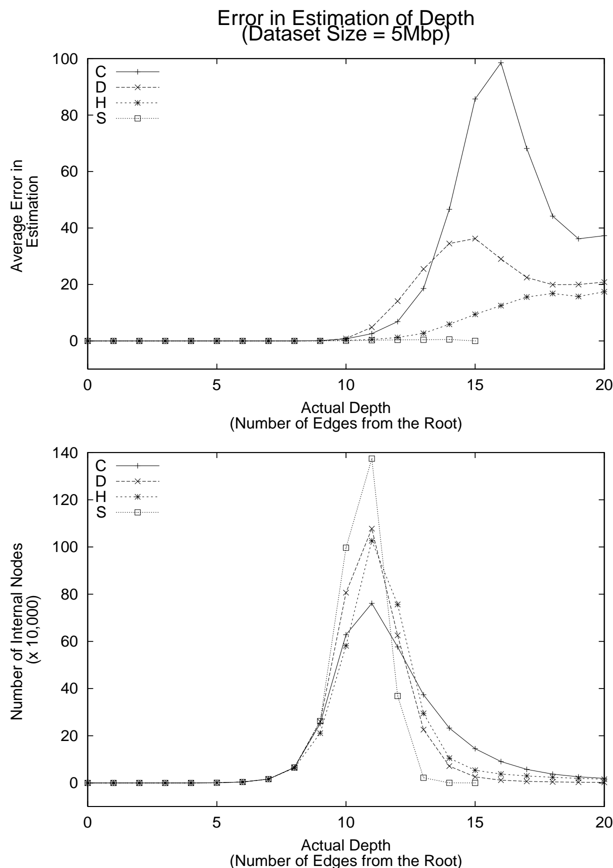

Figure 3. Depth Estimation Error

In summary, these graphs show that for even for datasets that deviate from the symmetric Bernoulli model, the impact of errors incurred by the proposed approximation to eventual depth is not very significant, and the quality of approximation improves with the increase in the length of the sequence.

4. Design of TOP-Q

Before describing the TOP-Q buffering strategy, we present the design of TOP, a simpler version of TOP-Q, which exploits the observation that higher number of accesses are to the nodes that are eventually higher in the tree, as well as the approximation to the eventual depth presented above.

We consider the situation where each disk-page contains a collection of nodes of the suffix tree – either internal or leaf, but not a mix of both. In order to minimize the storage cost of maintaining the path length, each disk page contains an associated path length, which is the average of the path lengths of all nodes packed in it. Each disk-page is completely packed with nodes as they created, and since the path length for each node is an invariant, the path length of the page can also be computed at the time it is committed to the disk.

Using this depth estimation for each page, the ranking policy employed for paging is to *rank the pages with smaller path lengths for retention in memory*. We call this buffering policy as *TOP*, to indicate that it tries to retain those pages of the suffix tree that are estimated to be top pages i.e., pages containing the top nodes of the tree.

4.1. Accomodating Correlated Accesses

Although the TOP buffering policy exploits the preferential access to the nodes with lower path lengths, it ignores the presence of correlated access patterns exhibited by the suffix tree construction. We provide below an example situation in the construction process, where this makes a impact on the performance of TOP.

The construction proceeds by splitting an edge, introducing a new branching node and a leaf node at that location, and filling in suffix-link pointers if needed. Every node stores the details of its incoming edge, i.e., $(start, end)$ indexes into the sequence and the length and label of the edge. When an edge $p(v) \rightarrow v$, is about to be split, the following actions are performed:

1. create a new branching node, v', and a leaf node l,
2. set the incoming edge details for both v' and l,
3. update the incoming edge details for v (the edge length is shortened, and the $start$ value and corresponding edge label are changed), and finally,
4. v' is set as the child of $p(v)$ in place of v, and v is now located under v'.

In addition, by the nature of the algorithm, only v has an active reference to it, and $p(v)$ is to be accessed through the parent pointer available with v. Therefore, $p(v)$ is not guaranteed to be pinned in memory during this process.

As the construction progresses, the internal nodes have ever-increasing path lengths associated with them. Therefore, the TOP policy evicts the pages as soon as they are filled and are not pinned through any active reference, since the internal pages also have larger path length values as the construction proceeds. Therefore, there is a possibility that $p(v)$, more precisely, the page containing $p(v)$, is not retained in the buffer for long.

We evaluated the impact of such correlated accesses to nodes, by measuring the the number of characters processed by the algorithm between the instant a page is evicted and the instant it is next requested, generating a fault on the buffer pool. The initial interesting portion of the results is shown in Figure 4, which plots the number of faulted pages on a logscale and the number of characters processed since they were last evicted from the buffer pool. As shown in these graphs, the number of evictions that have pages with immediate reference – i.e., before the complete addition of the next character into the suffix tree – is *orders of magnitude larger* than the evictions of pages that are accessed after many more characters are added.

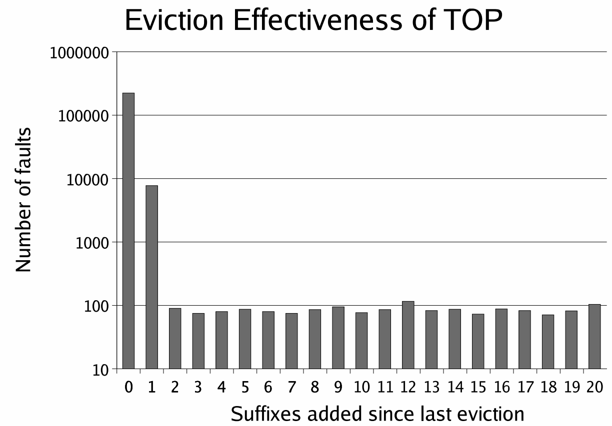

Figure 4. Correlated Accesses in TOP

The *TOP-Q* strategy compensates for this unresponsiveness of TOP to such accesses, by splitting the buffer pool into a collection of pages maintained in the order of their path lengths – implemented as a *Heap* structure, and a short fixed-length queue of pages to hold the pages evicted from the heap. The buffered pages in the heap are chosen for eviction just like in the TOP policy. However, unlike TOP, these pages are moved to the short, fixed-length queue part of the buffer pool managed in a FIFO fashion. The presence of the queue of pages effectively introduces a *delay* in the eviction of pages, satisfying almost all the immediate references to the page. We have used a queue of 10 pages with buffer-pool sizes ranging from thousands to hundreds of thousands of pages in our experiments and found it to perform well in practice.

5. Suffix Tree Representation

We now turn our focus to the physical representation of the nodes of the suffix tree. Much attention has been paid to reducing the size of these nodes [19], the goal being to maintain the tree entirely in memory, so that nonlocal accesses over the tree induced by linear-time construction algorithms are not affected by the virtual memory pag-

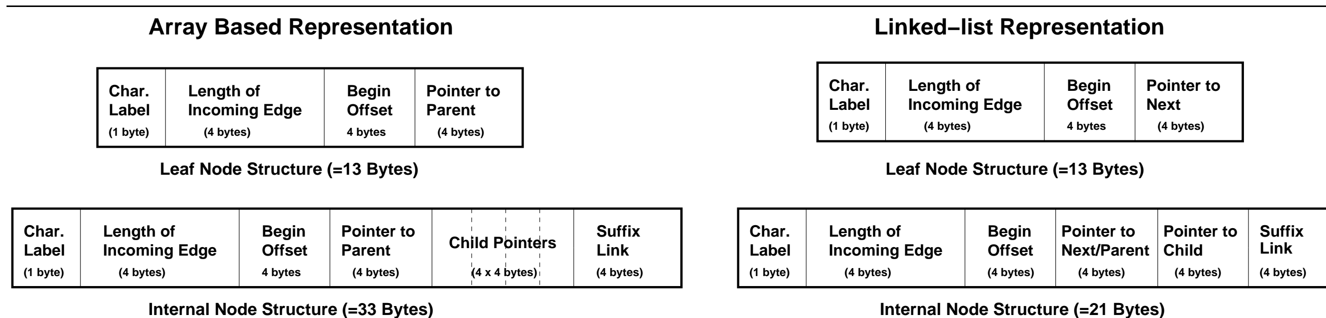

Figure 5. Structure of Suffix tree Nodes

ing [10]. However, it is not known which of these node representations is more appropriate when the suffix tree has to be constructed and maintained completely on disk.

The simplest but most space consuming strategy is to use an array of size $|\Sigma|$ (where Σ is the size of the alphabet) at each internal node of the tree. Each array entry corresponds to an edge, whose edge-label begins with the character associated with the array entry. These edges are implemented as pointers to corresponding child nodes in the suffix tree. We term this representation as *array implementation* of suffix tree nodes. This representation, although simple to program, is not preferred in most practical implementations since this results in a lot of wasted space, with many pointers containing $null$ values. This overhead is especially severe in nodes lower in the tree since the tree edges become sparser at lower portions of the tree.

As suggested by McCreight [6], a space efficient alternative to the array is to use a linked list of siblings and at every internal node maintain a single pointer to the head of the linked list containing its child nodes. Traversals from the internal node are implemented by sequentially searching this list for the appropriate child node. We refer to the resulting representation as the *linked-list representation* of the suffix tree node. This structure is a popular choice, due to its simplicity in implementation as well as superior space economy.

The structure of internal nodes and leaf nodes, for the array and linked-list implementation is given in Figure 5. It is clear that the linked-list representation achieves its superior space economy by significantly reducing the size of every internal node – in our implementation, from 33 bytes per internal node in array representation to 21 bytes per internal node.

Although it is possible to maintain the linked-list of siblings in a sorted fashion to reduce the searching time, this provides no significant advantage with small alphabet size sequences (such as DNA) and only complicates the implementation. Moreover, maintaining the sorted order of the list on disk involves update of many nodes for every insertion leading to a large number of accesses. Hence, we do not maintain the sibling linked-list in sorted fashion.

Name	Description	Length (in Mbp)	%-age Distribution of nucleotides			
			A	T	C	G
S	Sym. Bernoulli	25	25	25	25	25
D	Drosoph. genome	25	29	29	21	21
H	Human Chr. II	25	30	30	20	20
C	C.elegans Chr. I	15	32	32	18	18

Table 1. Characteristics of the Datasets

6. Evaluation Framework

In this section, we describe the framework used for evaluation of various buffering strategies, and node implementation choices, during the online construction of persistent suffix trees.

In our evaluation, we use a total of four DNA datasets, three of which are drawn from C.elegans Chromosome I (dataset C), Human Chromosome II (dataset H) and complete genome of Drosophila Melanogaster (dataset D). The remaining dataset is a synthetic symmetric Bernoulli sequence over DNA alphabet (dataset S). The details of these datasets are summarized in Table 1. From these statistics we see that these datasets comprise of sequences ranging from symmetric distribution of alphabets (dataset S) to highly skewed distribution (dataset C). Additionally, we also use an amino-acid sequence, SPROT, derived from SWISS-PROT collection of proteins [20]. Since most of the individual protein sequences are short, a 25Mb dataset was generated by concatenating the sequences together.

6.1. Implementation Details

As already mentioned in Section 4, suffix tree nodes are packed into fixed size pages before they are committed to the disk. The pages on disk are either internal pages or leaf pages, depending on whether they store internal nodes or leaf nodes of the tree. The variation in the internal and leaf node sizes leads to the packing density (i.e., the number of

nodes in a disk page) of leaf pages being greater than that of internal pages.

The storage of both internal nodes and leaf nodes is in their order of creation. Each page is committed immediately to the disk, as soon as all the space in the page is utilized. Note that, at the time of committing to the disk, all the entries in nodes of the page may not be filled – some of them may be defined and updated at later times in the construction process. Also, a page is pinned in memory if there are any active references pointing to nodes in the page.

The internal and leaf pages are distinguished also in terms of their buffer pools. This is due to the distinctly different access patterns made during the construction of the tree. Internal nodes of the tree are used repeatedly (with or without suffix links) for reaching the location of next suffix. On the other hand, leaf nodes are re-visited only when the suffix being located ends in a leaf node. In fact, our buffer management system is designed such that separate policies can be applied to internal and leaf page buffer pools.

6.2. Buffer Management Policies

The design of buffer management policies has been an active area of research for many years, and a host of policies that show improved hitrates over various database workloads have been proposed [21, 2, 9, 5].

We compare the static policy of TOP-Q against the following popular policies that are based on page access statistics, commonly used in database management systems:

LRU (*Least Recently Used*) In case of a page fault, it replaces the least recently used page from the buffer pool to accommodate the new page. It incurs a constant time computational overhead for every access, in order to manipulate the list of pageframes maintained in the order of recency of access.

2Q The 2Q algorithm [12] is a constant time overhead approximation of LRU-2 [5] that is found to perform as well as LRU-2 for a variety of reference patterns. The 2Q algorithm covers the most important drawback of LRU-2, by reducing the computational overhead from $log(N)$ work for *every access* in LRU-2 to a constant time overhead.

The metric of comparison between these popular buffering policies and our TOP-Q strategy is the *overall* buffer hit-rates observed during the construction of the suffix tree, with increasing length of sequence indexed, and for a fixed amount of buffer space.

6.3. Buffer Pool Allocation

In our evaluation of buffering policies, we maintain separate buffer pools for leaf and internal pages, with the buffering policy applied within each of the buffer pools. With a fixed amount of memory space at our disposal as the buffer space, it is interesting to see if there is an effective way to *partition* this space between the two classes of pages of the suffix tree.

The simplest partitioning is to distribute the available buffer pages equally between both leaf and internal nodes. It results in more number of leaf nodes being buffered than the internal nodes due to the better packing density of the leaf pages. Therefore, the effectiveness of buffer pool can be improved by partitioning it to hold equal number of internal nodes and leaf nodes. Although the number of internal nodes is 0.6 - 0.8 times the number of leaf nodes (for typical DNA sequences), the level of activity over internal nodes, in terms of their accesses and updates, is much higher than over leaf nodes. Hence, partitioning schemes that are skewed to hold more number of internal pages than leaf pages can be expected to perform better in practice.

Additionally, it should be noted that, as the construction of the suffix tree progresses, the overall size of the tree increases, leading to traversals over the tree covering a larger number of pages. In fact, a point may arrive when the available fixed size buffer may not be sufficient to efficiently handle the requests over an extremely large suffix tree. In order to compensate for this growing size of the data-structure and provide a normalized performance measure for all the policies, we consider the steady hitrates obtained, when a fraction of the suffix tree size is provided for buffering. In other words, as the suffix tree construction progresses, more pages are introduced into the buffer pool such that the ratio of buffer pool size to the total size of the suffix tree (measured in number of pages) is held constant. The distribution of steady hitrates obtained through these experiments for various settings of fraction buffered, provide an *upper bound* on the performance of each of the buffering policies, independent of the size of the tree.

7. Experimental Results

In this section, we present the results of our empirical evaluation of the buffering policies and the node implementation choices outlined in previous sections. The parameters applicable for all our experiments are summarized in Table 2. Due to space limitations, we provide results for only two DNA datasets, dataset S and dataset H, and for the SPROT dataset. The results for other two DNA sequences, dataset C and D, show behavior similar to that of dataset H and S, respectively.

7.1. Construction with Fixed-size Buffer

The fixed buffer size experiments were conducted with total memory space allocated for the buffer pool restricted to just 32MB, a total of 8000 pages. This enabled us to work

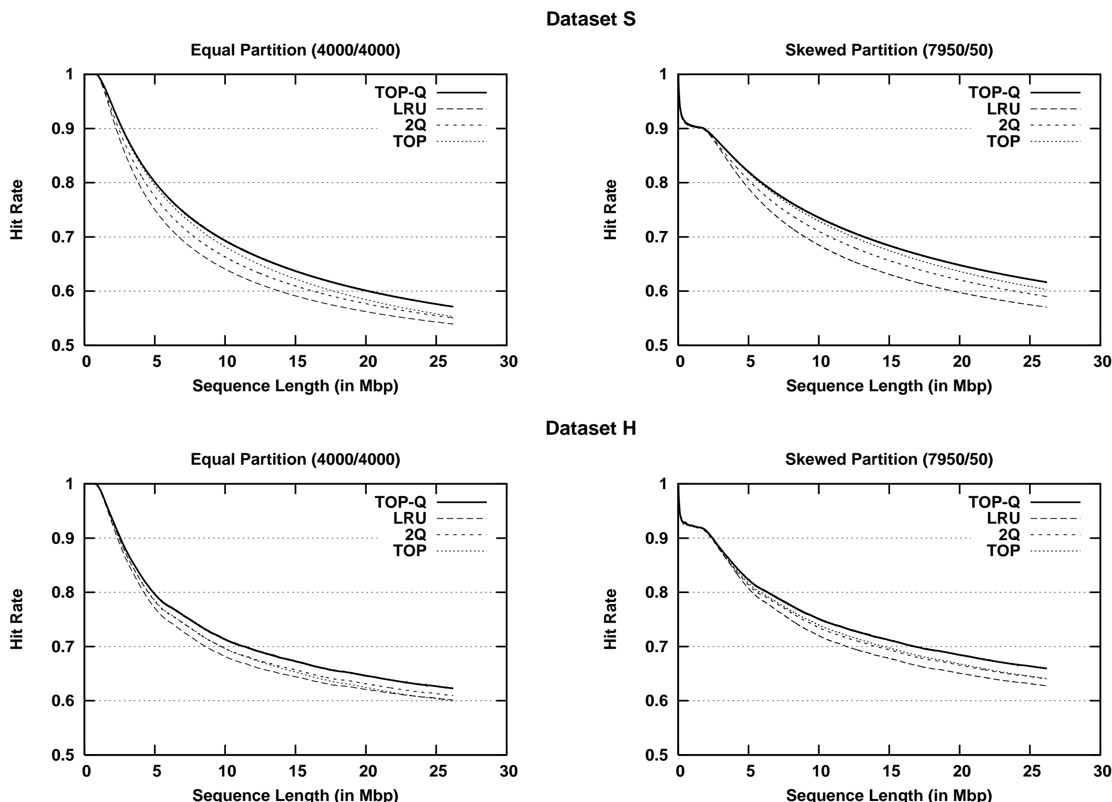

Figure 6. Combined Hit-rates for Construction with Array-based nodes

Parameter		Value
Node size	**Leaf**	13 Bytes
	Internal (Array)	33 Bytes
	Internal (List)	21 Bytes
Page size		4096 Bytes

Table 2. Constant Experimental Parameters

with smaller length sequences, and perform experiments for collecting buffer pool statistics using a simulated memory hierarchy. However, in practice, much larger datasets will be indexed and therefore proportionately larger buffer pool sizes should be used.

As described earlier in Section 6.3, the available buffer space can be partitioned between internal and leaf buffer pools, ranging from equal partitioning to skewed partitioning in favor of the internal buffer pool. We experimented with many buffer partitioning schemes, and found that the overall hitrate is dominated completely by the internal node accesses alone. Thus, the performance improves with increased skew in partitioning favoring the internal pages. This observation holds for both the array as well as linked-list representation of the suffix tree. In this paper, we present results for equi-partitioning of the buffer pool with 4000 pages each for managing internal and leaf pages and a highly skewed partitioning with 7950 pages to internal buffer pool and remaining 50 pages as leaf buffers.

Array-based Suffi x Tree Construction The buffer hitrate obtained for page accesses, of both internal and leaf pages, using array representation of nodes is shown in Figure 6. The graphs also show the hitrate for simple TOP, in order to provide a measure of gains obtained by the TOP-Q extension.

Figure 6 shows that TOP-Q provides consistently higher hitrate than LRU and 2Q. In addition, the following observations can be made about these results:

- The hitrates with skewed partitioning of buffer pool are higher than with equi-partitioning. This clearly shows that the overall performance is dominated by the effectiveness of the buffering over internal pages.
- TOP-Q, as expected, performs better than the plain TOP strategy and provides hitrates that degrade slower with increasing suffix tree size, as compared to the other policies.
- LRU exhibits the lowest hitrate, and upon further investigation was found to have performance almost

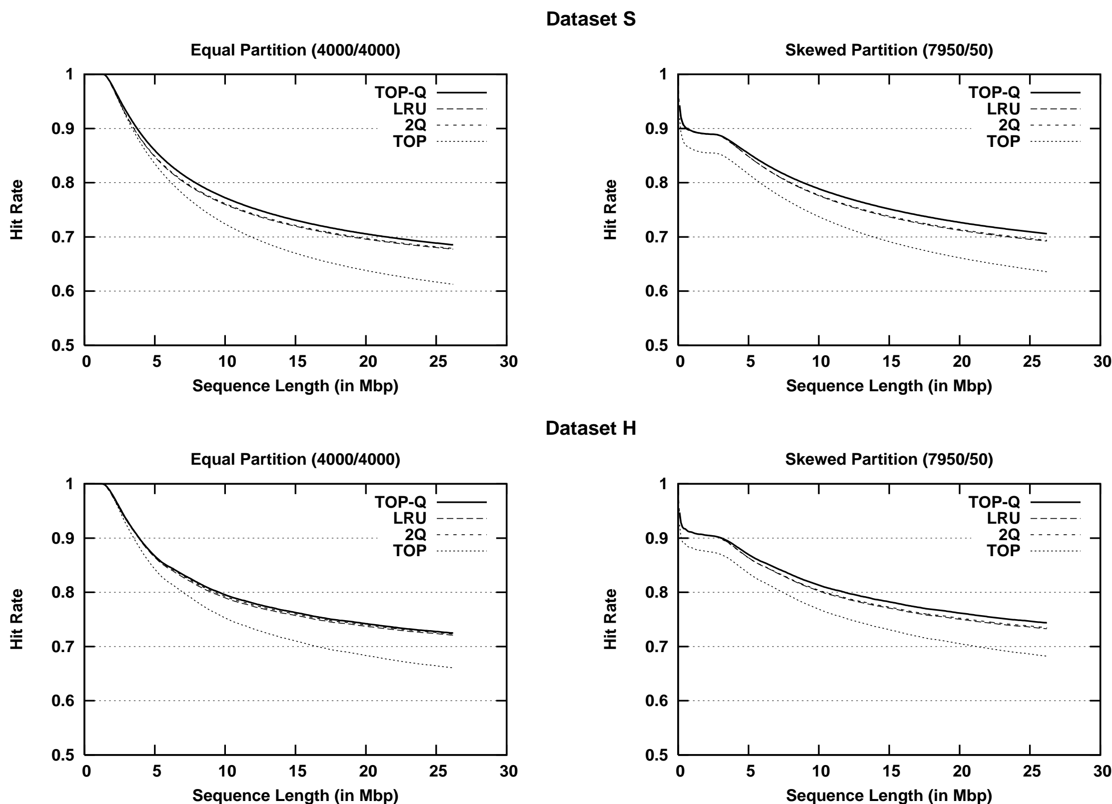

Figure 7. Combined Hit-rates for Construction with Linked-list Representation

equal to the policy of evicting a randomly selected page (Random Replacement strategy).

- The performance of TOP is better than LRU in the initial stages of the construction, and with increased skew in the buffer allocation, TOP improves to match the performance of 2Q.
- The ideal sequence for TOP is the dataset S, which is a symmetric Bernoulli sequence. With this dataset, it provides improved hitrates over even the highly sophisticated 2Q algorithm.

In summary, TOP-Q emerges as the best buffer management policy with any buffer partitioning strategy over all the datasets. Moreover, it should be noted that TOP-Q has an extremely low computational overhead as compared to LRU and 2Q, since its control data-structures are not updated at every access to a page.

Linked-list based Suffix Tree Construction The behavior of all the buffering policies for linked-list representation of nodes is shown in Figure 7. First of all, it is worth noting that all the algorithms provide better hitrates than in the case of array representation, with the sole exception of TOP. This is due to greater presence of correlated accesses, similar to those discussed in Section 4.1, arising out of traversals over the linked list of siblings. But the TOP-Q algorithm still maintains an edge over the LRU and 2Q algorithms, although the improvements are not as significant as in the case of the array representation. Another interesting point is that both LRU and 2Q show almost the same hitrates, with both equal and skewed partitioning of the buffer pool.

Choice of Implementation The comparison of graphs in Figure 6 and Figure 7 indicates that the linked-list representation provides better hitrates than array representation. This is partly due to the fact that for the same amount of buffer space at our disposal, the linked-list representation buffers more number of internal nodes due to its improved space economy. This seems to suggest that the linked-list representation is better suited for persistent construction with buffering.

However, this conclusion is misleading since the sequence of page references in both cases are very different and the hitrates are normalized within each reference sequence. The absolute number of disk accesses made during the construction provides a metric that is independent of the reference sequence. Figure 8 shows the absolute number of read and write disk accesses, using the TOP-Q buffering policy, for the dataset H. As these numbers indicate, the

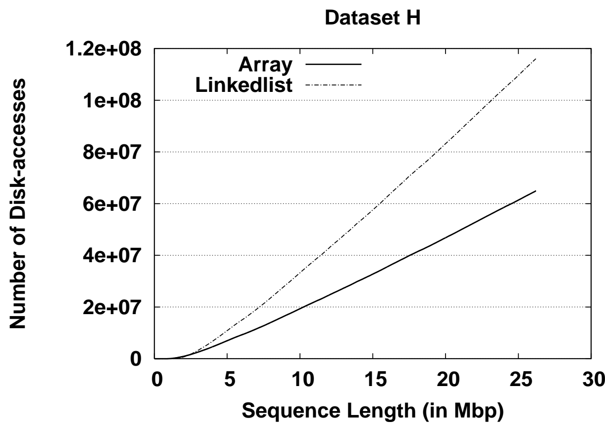

Figure 8. Disk-accesses during construction

linked list representation has a significantly higher I/O overhead than the array representation. This overhead is primarily due to traversals over siblings in the linked-list to locate the appropriate child to follow. Each of these siblings could have been created at different points during the construction, resulting in their non-contiguous storage on disk.

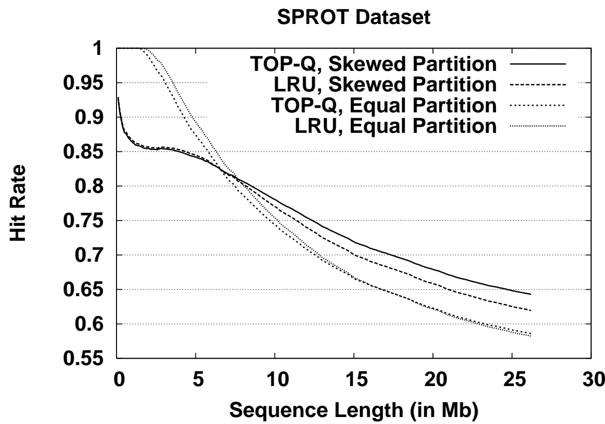

Figure 9. Hitrates over Protein Data

Suffix Tree over Protein Data The results of our experiments with the SPROT dataset are summarized in the Figure 9. The graph does not include hitrates for 2Q, since they were very similar to LRU. As the numbers in this graph demonstrate, TOP-Q performs better than LRU with increasing skew in the buffer pool allocation. Further, with increasing length of the indexed sequence, the hitrate of TOP-Q degrades much slower than LRU, in both the partitioning schemes. Thus, the benefits of TOP-Q are applicable not only with small-alphabet sequences such as DNA, but also with sequences with larger alphabet sizes.

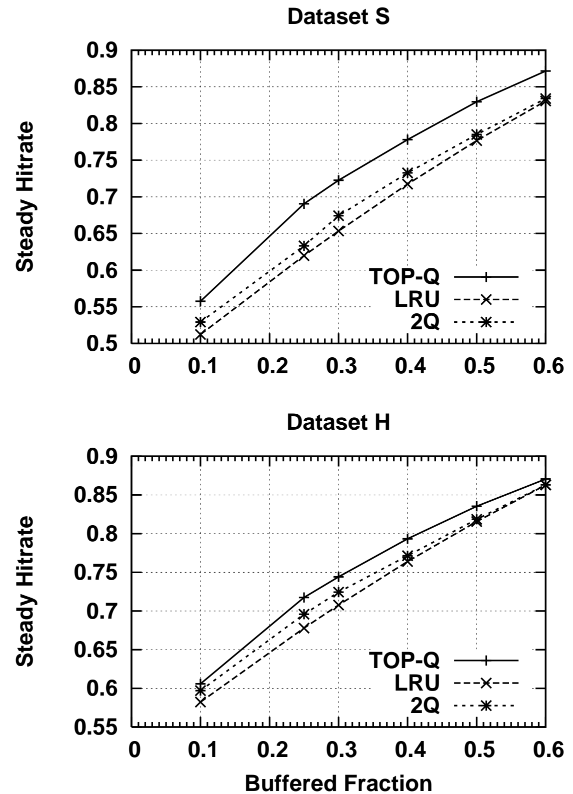

Figure 10. Behavior with Proportional Buffers

7.2. Construction with Proportional Buffering

We now move on to proportional buffering, discussed in Section 6.3. The resulting steady hitrate values are plotted in Figure 10.

As shown in these graphs, the performance of LRU improves almost linearly with the buffered fraction of the datastructure and 2Q is only marginally better than LRU. On the otherhand, TOP-Q provides super-linear improvements with diminishing returns, with increasing fraction of buffering. The performance of TOP-Q peaks for about 25% of the tree in the buffer providing close to 72.5% hitrate for construction over real-life DNA sequences. When more than 60% of the suffix tree is buffered, then all the three buffering strategies perform equally well with upto 85% steady hitrate.

7.3. On-disk Construction

In order to consider the practical impact of the improved performance statistics for TOP-Q buffering and array representation of nodes, we built persistent suffix trees for large DNA sequences on two different classes of machines. One was a PC class machine running Linux Redhat 8.0

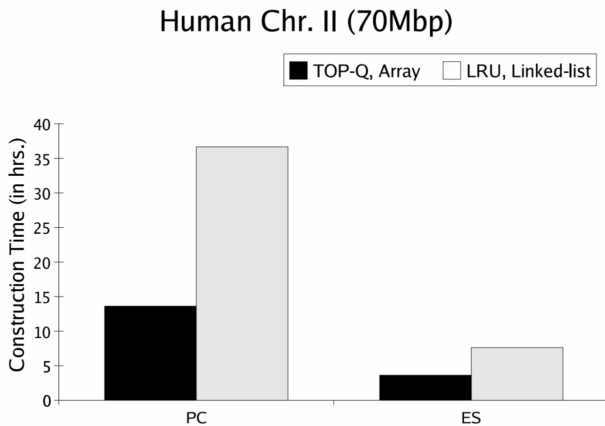

Figure 11. Persistent Construction Times

and having an 18GB 10,000-RPM SCSI hard disk (IBM DDYS-T18350M model). The other machine represents the server class hardware used in current day large bioinformatics projects – a HP-Compaq ES45 server running Tru64 Unix 5.1 and 432GB (6*72GB) storage with single channel RAID controller at RAID-0 configuration. We refer to these two platforms as *PC* and *ES* ,respectively, in the rest of this discussion.

We compared the total execution time for constructing a persistent suffix tree using 1GB of buffer space, split in a moderately skewed fashion with 68% of pages to internal buffer pool and the remaining 32% of pages as leaf buffer pool. The strategies we used are: Linked-list representation with LRU buffering, and array representation with TOP-Q buffering scheme.

Figure 11 provides the performance of these two strategies on both the platforms. These results show that TOP-Q with array representation provides 50% to 65% improved construction time over both platforms considered.

Impact of OS Buffering Due to the O_SYNC option for file operations used during the persistent suffix construction, *file writes* are immediately synchronized with the disk image. On the otherhand, the file reads might be served from the I/O buffers maintained by the OS. Hence, the number of disk page writes during construction makes a significant impact on the overall construction time. Note that when a page chosen as the victim by the buffering policy is *dirty*, all the policies immediately flush it out to the disk. Upon investigation we found that TOP-Q results in fewer disk-writes than the other buffering strategies, which adds to the reduction in seek-times achieved through improved hitrates, translating to significant improvements in on-disk construction times.

8. Related Work

Since the time Weiner [16] introduced the suffix tree data structure and a linear time algorithm for its construction, there has been growing interest in more space- and time-efficient algorithms for construction of suffix trees. A conceptually different and space efficient algorithm to build suffix trees in linear time have been given by McCreight [6], and later, by Ukkonen [7]. Further, McCreight also suggested the use of linked-list implementation of nodes for reducing the space overhead of the suffix trees. All these algorithms make use of suffix links to achieve linear-time construction and are implemented for various constant-sized alphabet datasets.

However, all of these algorithms show very bad performance when applied to suffix tree construction in secondary memory. The main bottleneck in the direct application of these algorithms over suffix-trees on disk is considered to be the random seeks induced during construction [13]. It has also been noted in [4] that suffix links utilized by all these algorithms traverse the suffix tree "horizontally", while edges span the tree "vertically". Thus, at-least one of them is expected to result in random access of memory. However, this is true only if the tree is stored on disk using depth-wise traversal of either edges of the tree or links of the tree. But this storage pattern is not feasible during the on-line construction of suffix tree on disk. Therefore, in practice, *both* edges and suffix links show non-local access patterns when the suffix tree is constructed on disk. Thus, we need to carefully consider tuning of paging policies to reduce the impact of such traversals.

Suffix trees provide an *accurate* indexing solution for searching over large corpus of DNA or Protein sequences. Initial use of suffix trees in genomic indexing was restricted over small length DNA sequences [1], where suffix tree could fit completely into main memory. However, until recently, they were not considered for construction and maintenance in secondary-memory. In fact, even in a recent work [8], it was reported that whenever the dataset is large enough suffix trees are not a viable option of indexing, since the memory is too small to hold the index completely.

Initial theoretical breakthrough for suffix tree construction on secondary memory was given by Farach et al. [13, 14]. They introduced a novel way to construct the suffix tree over a large sequence by following a divide-conquer approach (as opposed to the traditional suffix-at-a-time approach), and used that to show that persistent suffix trees could be built with the same I/O complexity as that of external sorting. They also pointed out that "traditional" algorithms (such as those of Weiner, Ukkonen, and McCreight etc.), which follow incremental extension of suffix trees, will be forced to make random I/Os resulting in bad on-disk performance. Their observations on traditional algo-

rithms were made without considering the effects of paging/caching policies that could be employed during the construction process. In fact, they state at the end of their paper [13], that it would be worthwhile considering the behavior of the construction algorithms in presence of paging, which is the topic we address in this paper.

Recently, Hunt et al. [4] proposed a phased construction approach to building suffix trees, where they use an asymptotically quadratic algorithm for construction of suffix trees without suffix links. They report empirical evaluation of both linear-time suffix link based algorithm and their phased approach for persistent suffix trees to show the superiority of their approach. Their results do not consider the effects of paging policies and the storage management issues. In fact, in [18], it has been reported that the bottleneck is in the choice of checkpointing scheme of PJama – the underlying persistent mechanism used in [4].

9. Conclusions

In this paper, we have evaluated the impact of buffering and internal node implementation choices on the construction of a suffix tree in secondary memory. We also proposed a novel low overhead buffer management policy called TOP-Q, which exploits the pattern of accesses over the suffix tree during its construction. Through an extensive empirical study involving both DNA and Protein sequences, we showed that TOP-Q performs better than other popular buffer algorithms such as LRU and LRU-2. The TOP-Q algorithm saves more than 75% of disk I/O by buffering merely 25% of the tree.

In addition, it was shown that that the commonly used, space-economical linked-list representation of the suffix tree is extremely expensive for construction on secondary memory. Instead, a simple implementation using arrays at each internal node is shown to be far better suited for persistent suffix tree representation.

A performance evaluation of TOP-Q with array representation of nodes against a popularly reported linked-list representation with LRU buffering policy showed that significant speedups to the tune of 50% to 70% were obtained, over different platforms.

Acknowledgements This work was supported in part by a Swarnajayanti Fellowship from the Dept. of Science & Technology, Govt. of India, and a research grant from the Dept. of Bio-technology, Govt. of India.

References

[1] P. Bieganski. *Genetic sequence data retrieval and manipulation based on generalized suffi x trees*. Ph. D. Thesis, University of Minnesota, 1995.

[2] H. Chou and D. Dewitt. An evaluation of buffer management strategies for relational database systems. In *Proc. of 11th VLDB Conf.*, 1985.

[3] E. Hunt. PJama Stores and Suffi x Tree Indexing for Bioinformatics Applications. In *Proc. of ECOOP*, 2000.

[4] E. Hunt, M. P. Atkinson, and R. W. Irving. A Database Index to Large Biological Sequences. In *Proc. of the 27th VLDB Conference*, 2001.

[5] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proc. of the ACM SIGMOD Intl. Conf. on Mgmt. of Data*, 1993.

[6] E. M. McCreight. A Space-Effi cient Suffi x Tree Construction Algorithm. *JACM*, 23(2), 1976.

[7] E. Ukkonen. Online Construction of Suffi x-trees. *Algorithmica*, 14, 1995.

[8] G. Navarro and R. Baeza-Yates. A Hybrid Indexing Method for Approximate String Matching. *Jl. of Discrete Algorithms*, 2000.

[9] Giovanni Maria Sacco. Index Access with Finite Buffer. In *Proc. of 13th VLDB Conf.*, 1987.

[10] D. Gusfi eld. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, Cambridge, 1997.

[11] D. Gusfi eld. Suffi x trees come of age in bioinformatics (invited talk). In *IEEE Bioinformatics Conference (CSB)*, 2002.

[12] T. Johnson and D. Shasha. 2Q : A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proc. of the 20th VLDB Conf.*, 1994.

[13] M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the Memory Bottleneck in Suffi x Tree Construction. *Proc. of IEEE Annual Symposium on Foundations of Computer Science*, 1998.

[14] M. Farach-Colton. Optimal Suffi x Tree Construction with Large Alphabets. In *Proc. of IEEE Annual Symposium on Foundations of Computer Science*, 1997.

[15] GenBank Overview. http://www.ncbi.nlm.nih.gov/Genbank/GenbankOverview.html, 2002.

[16] P. Weiner. Linear Pattern Matching algorithms. In *Proc. of the 14th IEEE Symp. on Switching and Automata Theory*, 1973.

[17] R. Giegerich and S. Kurtz. A Comparison of Imperative and purely Functional Suffi x tree constructions. *Science of Programming*, 1995.

[18] R. Japp. First Year Report. Master's thesis, University of Glasgow, July 2001.

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

[20] Swiss-prot protein knowledgebase. http://www.expasy.org/sprot/, 2002.

[21] W. Effelsberg and T. Haerder. Principles of Database Buffer Management. *ACM Transactions on Database Systems*, 9(4), 1984.

[22] W. I. Chang and E. L. Lawler. Approximate string matching in sublinear expected time. In *Proc. of IEEE Annual Symposium on Foundations of Computer Science*, 1990.

[23] W. Szpankowski. *Average Case Analysis of Algorithms on Sequences*. Wiley-Interscience, 2001.