Large-Scale Pattern Search Using Reduced-Space On-Disk Suffix Arrays

Simon Gog, Alistair Moffat, J. Shane Culpepper, Andrew Turpin, and Anthony Wirth

***Abstract*—The suffix array is an efficient data structure for in-memory pattern search. Suffix arrays can also be used for external-memory pattern search, via two-level structures that use an internal index to identify the correct block of suffix pointers. In this paper we describe a new two-level suffix array-based index structure that requires significantly less disk space than previous approaches. Key to the saving is the use of disk blocks that are based on prefixes rather than the more usual uniform-sampling approach, allowing reductions between blocks and subparts of other blocks. We also describe a new in-memory structure – the condensed BWT – and show that it allows common patterns to be resolved without access to the text. Experiments using 64 GB of English web text on a computer with 4 GB of main memory demonstrate the speed and versatility of the new approach. For this data the index is around one-third the size of previous two-level mechanisms; and the memory footprint of as little as 1% of the text size means that queries can be processed more quickly than is possible with a compact FM-INDEX.**

***Index Terms*—String search, pattern matching, suffix array, Burrows-Wheeler transform, succinct data structure, disk-based algorithm, experimental evaluation.**

I. INTRODUCTION

STRING search is a well known problem: given a text $\mathsf{T}[0 \ldots n-1]$ over some alphabet Σ of size $\sigma = |\Sigma|$, and a pattern $\mathsf{P}[0 \ldots m-1]$, locate the occurrences of P in T. Several different query modes are possible: asking whether or not P occurs (*existence* queries); asking how many times P occurs (*count* queries); asking for the byte locations in T at which P occurs (*locate* queries); and asking for a set of extracted contexts of T that includes each occurrence of P (*context* queries).

When T and P are provided on a one-off basis, sequential pattern search methods take $O(n+m)$ time. When T is fixed, and many patterns are to be processed, it is likely to be more efficient to pre-process T and construct an *index*. The *suffix array* [1] is one such index, allowing *locate* queries to be answered in $O(m + \log n + k)$ time when there are k occurrences of P in T, using $O(n \log n)$ bits of space in addition to T. But suffix arrays only provide efficient querying if T plus the index require less main memory than is available on the host computer, because multiple accesses are required to both. For large texts, two-tier structures are needed, with an in-memory component consulted first in order to identify the data that must be retrieved from an on-disk index.

S. Gog, A. Moffat, A. Turpin, and A. Wirth are with the Department of Computing and Information Systems, The University of Melbourne, Australia, e-mail: {simon.gog,ammoffat,aturpin,awirth}@unimelb.edu.au.

J. S. Culpepper is with the School of Computer Science and Information Technology, RMIT University, Australia, e-mail shane.culpepper@rmit.edu.au

Manuscript received March 2013, revised June 2013.

A. Our Contributions

We show that if the in-memory index of a two-level suffix array is sampled via a method that respects common prefixes, the space required by the suffix array blocks on disk can be reduced by as much as 50%. This gain is achieved by identifying *reducible* blocks that can be replaced by references to subintervals within other blocks on disk.

We also describe a new in-memory structure for indexing variable-length common-prefix blocks that is comparable in size to the *bit-blind tree*. In terms of operational functionality, the new *condensed BWT* approach has the benefit of being comprehensive, meaning that *existence* and *count* searches for frequently occurring patterns can be resolved without disk accesses. The new approach employs backward searching and the Burrows-Wheeler Transform.

Combining these two mechanisms yields the RoSA, a new approach to two-level suffix array searching. Experiments using 64 GB of English web text and a laptop computer with just 4 GB of main memory demonstrate the RoSA's speed and versatility. For this data the disk index is around one third of the size of the previous LOF-SA two-level mechanism [2], [3], and the in-memory part of the index has a very small requirement – as little as 1% of the size of the input text.

B. Definitions

Text $\mathsf{T}[0 \ldots n-1]$ is assumed to consist of n symbols, each of which is drawn from an alphabet $\Sigma = \{a_0, a_1, a_2, \ldots, a_{\sigma-1}\}$ of size $\sigma = |\Sigma|$, augmented by a sentinel in $\mathsf{T}[n]$ that is smaller than every element in Σ. The ith suffix of T is the sequence $\mathsf{T}[i \ldots n]$, including the sentinel, and is denoted by T_i. The longest common prefix $LCP(\mathsf{T}_i, \mathsf{T}_j)$ of two suffixes of T is the maximal value k such that $\mathsf{T}[i+\ell] = \mathsf{T}[j+\ell]$ for all $0 \le \ell < k$. If T_i and T_j are suffixes of T, then $\mathsf{T}_i < \mathsf{T}_j$ if and only if $\mathsf{T}[i+k] < \mathsf{T}[j+k]$, where $k = LCP(\mathsf{T}_i, \mathsf{T}_j)$. A pattern $\mathsf{P}[0 \ldots m-1]$ matches T at i if $\mathsf{P}[0 \ldots m-1]$ is identical to $\mathsf{T}[i \ldots i+m-1]$, that is, if P is a prefix of the ith suffix of T.

Array $\mathsf{SA}[0 \ldots n]$ is a suffix array for text T if $\mathsf{T}_{\mathsf{SA}[i]} < \mathsf{T}_{\mathsf{SA}[j]}$ whenever $i < j$. In the context of a suffix array it is then useful to define $\mathsf{LCP}[i] = LCP(\mathsf{T}_{\mathsf{SA}[i-1]}, \mathsf{T}_{\mathsf{SA}[i]})$, with $\mathsf{LCP}[0] = 0$. The Burrows-Wheeler transform (BWT), denoted by L, is also required in our development: $\mathsf{L}[i]$ contains the preceding character of the ith sorted suffix, $\mathsf{L}[i] = \mathsf{T}[(\mathsf{SA}[i]-1) \bmod n]$. Figure 1 shows an example string of $n = 16$ characters that is used throughout the discussion, plus its sorted suffixes. The column headed $\mathsf{SA}[i]$ is the value stored in the ith entry in the suffix array for the string; and the

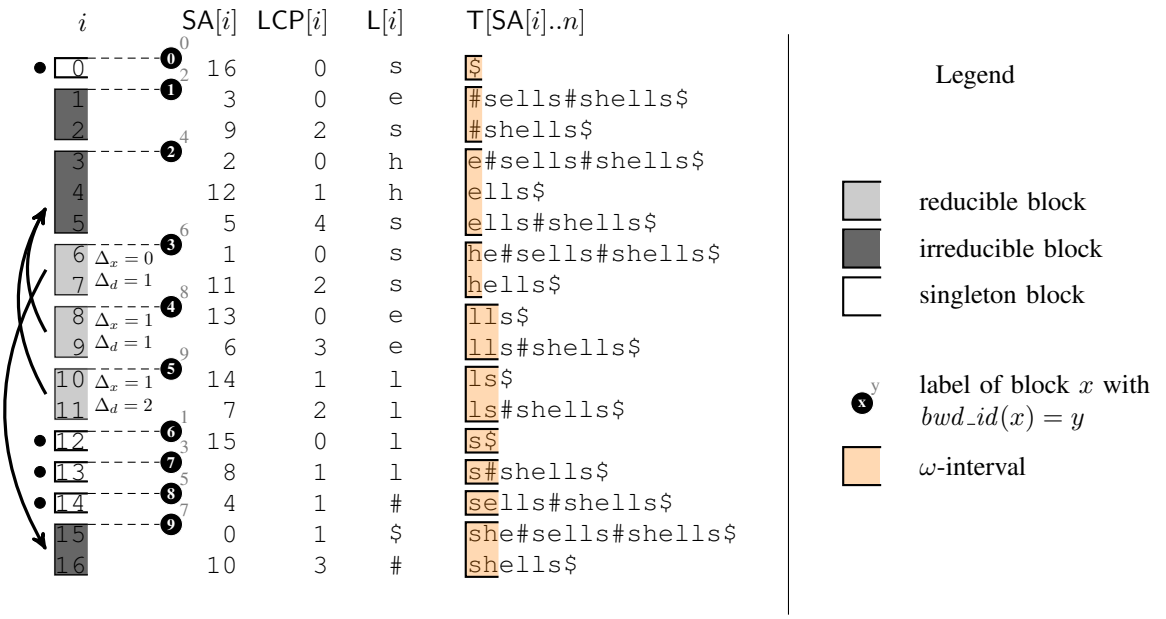

Fig. 1. External common-prefix suffix blocks formed for T = "she#sells#shells$" with blocksize $b = 3$.

column headed $\mathsf{L}[i]$ is the corresponding BWT symbol, being the character immediately prior to the ith sorted suffix. The other parts of Figure 1 are described shortly.

We also employ *rank* and *select* operations: for sequence X operation $rank(X, i, c)$ returns the number of occurrences of symbol or sequence c in $X[0..i-1]$; and $select(X, i, c)$ returns the position of the ith occurrence of c, counting from zero. For example, if $X[0..15]$ = "she#sells#shells", then $rank(X, 8, \text{"s"})$ is 2, and $select(X, 2, \text{"e"})$ is 12. Although sophisticated mechanisms exist for implementing *rank* and *select* that have good asymptotic properties, one of the most useful practical approaches simply adds regular cumulative sums to a standard bitvector representation, expanding it by 25% or by 6.25%, depending on the sampling interval [4], [5].

II. Background: Suffix Trees and Arrays

A number of index structures can be used for string search over a static text T.

A. Suffix Tree

A suffix tree for text T is a modified suffix trie in which the parent-child edges represent sequences of symbols from Σ rather than single symbols; and in which internal nodes that only have a single child are eliminated. The edge labels are stored as references to T rather than as explicit sequences of symbols, and the per-edge space requirement is thus $O(\log n)$ bits. A suffix tree has n leaves and at most n internal nodes, and occupies at most $O(n \log n)$ bits in total, with typical implementations requiring $3n$ or more $\log n$-bit pointers. Searching involves an access to T as each edge is traversed, in order to match symbols in the pattern.

B. Blind Tree

The suffix tree's accesses to the text T are not localized, and are relatively costly. In a *blind tree* [6], [7], [8] the outgoing edges at each node are represented by the first symbol of the corresponding sequence, rather than by pointers to T. The remaining (if any) symbols that label that edge in the corresponding suffix tree are not stored. Instead, internal nodes store the LCP of the set of strings represented at that node, and during querying, when a node is reached, the search steps forward to the indicated symbol, bypassing any omitted symbols. Having edges labeled by just a single symbol means that the symbols that are bypassed may not match between P and T. To address that risk, once either the pattern has been exhausted, or a leaf has been reached, the full pattern is checked against the indicated location in T. Proceeding with the search based on only partial matches means that the majority of the accesses to T are eliminated.

C. Bit-Blind Tree

A concise form of blind tree has been developed [6] which, for clarity, we refer to here as a *bit-blind tree*. Rather than character LCP values and character edge labels, bit-based LCP values are employed. Moreover, because internal nodes have exactly two children, the edge labels do not need to be stored. The tree becomes deeper by a factor of as much as $\log \sigma$; but takes less space per node. In total, once the tree structure has been provided, the cost of a bit-blind tree storing the n suffixes of a text T is $n-1$ internal nodes, each containing a bit-LCP value; and n leaves, each containing a $\log n$-bit suffix pointer.

Figure 2 shows the bit-blind tree for the set of blocks identified in the right-hand side of Figure 1. The reason that these particular strings are of interest, and only a partial tree is stored, is discussed shortly. The ten strings are each represented by one of the leaves of the tree; the categorization of those leaves into three types is also described below.

The bitvector bv at the bottom of Figure 2 describes the structure of the bit-blind tree, and eliminates the need for explicit pointers at the internal nodes. To create bv the nodes

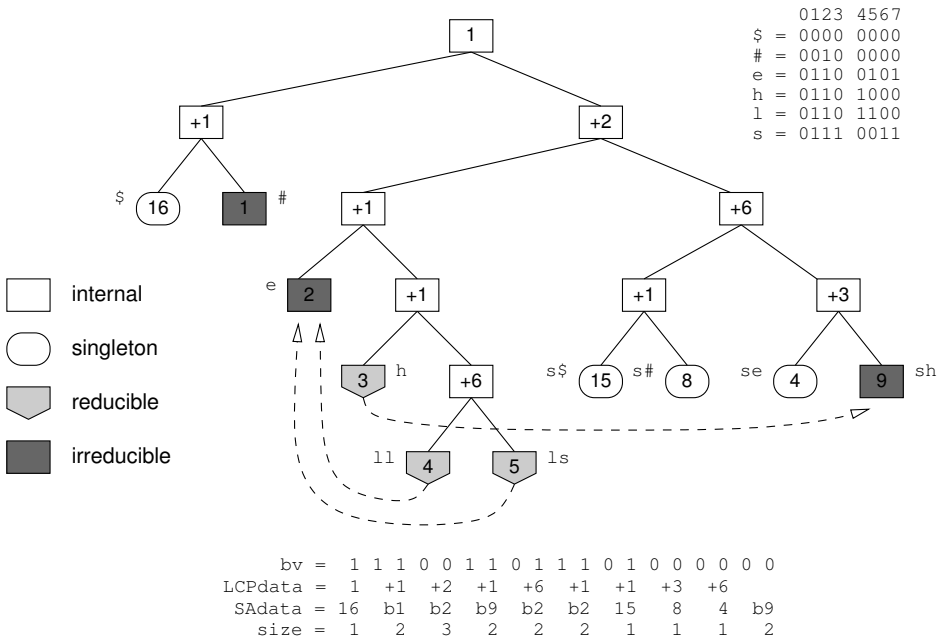

Fig. 2. Bit-blind tree for the ASCII strings "$", "#", "e", "h", "ll", "ls", "s$", "s#", "se", and "sh", being the identifying block prefixes of the ten suffix array blocks identified in Figure 1 when the example string is processed with $b = 3$. The three different types of leaf nodes, and the meaning of the dotted lines, are discussed in Section IV. The ASCII codes for the characters in question are shown at the top-right.

of the tree are labeled in row-level order, and a "1" bit is stored for nodes with (a pair of) children, and a "0" bit is stored if not. The "1" bits exactly correspond to the locations at which relative LCP values are required; conversely, the "0" bits exactly correspond to the locations at which block pointers are required. The required tree navigation operations on internal nodes (that is, node identifiers x such that $\mathsf{bv}[x] = 1$) are then provided via *rank* and *select* operations, as follows:

- $lchild(x) \leftarrow 2 \times rank(\mathsf{bv}, x, 1) + 1$
- $rchild(x) \leftarrow 2 \times rank(\mathsf{bv}, x, 1) + 2$
- $\mathsf{LCP}[x] \leftarrow \mathsf{LCP}[parent(x)] + LCPdata[rank(\mathsf{bv}, x, 1)]$

where $LCPdata$ is an array of LCP differentials, as shown at the bottom of the diagram, and $\mathsf{LCP}[parent(x)]$ will have been computed during the previous iteration of the tree traversal loop. Details of the three types of leaf node, and of the meaning of the $SAdata$ and $size$ fields, are given in Section IV.

D. Suffix Array

The suffix array is more compact than the suffix tree-based alternatives, including the bit-blind tree, and is typically represented as a single $\log n$-bit value for each suffix of T. In addition, if an LCP array is provided, the set of all matching locations of P in T can be identified in $O(1)$ time each once the first one has been identified.

Mäkinen and Navarro [9] note that runs in the BWT string L can be used to identify suffix pointer indirections that allow space to be saved. González and Navarro [10] extended this work, recognizing repeated patterns of suffix pointer differences using the RE-PAIR compression technique. But when T is small enough that it fits into available memory, the FM-INDEX, described next, is the most attractive option. In Section IV we apply similar techniques to disk-based suffix arrays, where the space reduction does result in practical benefit.

E. FM-Index

The last decade has seen considerable development in the area of *compressed self indexing*. Hon et al. [11] survey much of this work; perhaps the best exemplar of the category is the FM-INDEX of Ferragina and Manzini [12], [13]. Based on the Burrows-Wheeler transform, the FM-INDEX has a highly desirable blend of properties – it allows pattern search in $O(m \log \sigma)$ time; it requires space proportional to $nH_k(\mathsf{T}) + \sigma^k$, the information content of the original text[1]; and it allows reconstruction of T from the beginning, and from (with additional storage cost) sampled re-entry points.

For texts for which the FM-INDEX fits into random access memory, *existence* and *count* queries are fast; while the speed of *locate* queries depends on the sampling rate for decoding. We include experimental results for the FM-INDEX in Section VI, using a recent implementation [5].

The FM-INDEX is not efficient when the compressed representation of T is too large for main memory, because random accesses are required across the data structure. This means that even the best external variants potentially make m disk accesses [14], and are impractical for long patterns.

III. ON-DISK SUFFIX ARRAYS

Two approaches have emerged for storing suffix array structures on secondary storage: methods that make use of uniform-size blocks, so that every block except the last contains exactly

[1]That is, the number of bits required to store the text using an order-k statistical context-based compression model, including an allowance for storing the model parameters.

b pointers, where the blocksize b is chosen at the time the index is constructed; and methods that make use of variable-sized blocks, in which b is an upper bound on the blocksize.

A. Uniform Blocks and the String B-Tree

Baeza-Yates et al. [15] describe the SPAT, a structure in which the suffix array is formed into uniform blocks each containing b pointers, and the in-memory index is an array of n/b fixed-length strings, being the first ℓ_s symbols of the last suffix in each block. The AUGMENTED-SA proposal of Colussi and De Col [16] also partitions the on-disk suffix array into uniform blocks (each of $b = \log n$ suffix pointers) but with the in-memory index constructed as a suffix tree to the (full) first suffix string of the block. González and Navarro [14] provide a summary of these early techniques.

Ferragina and Grossi [6] describe a dynamic string search structure they call the String B-tree, or SB-TREE. For static data the SB-TREE can be implemented as a uniform partitioning of a suffix array, with an in-memory suffix tree index implemented as a blind tree or bit-blind tree. More than one level of indexing can be used if necessary, with all blocks having the same structure. Each node of the SB-TREE indexes b strings via $2b$ bits describing the shape of a binary tree with b leaves; plus $b-1$ internal node depths, expressed in bit offsets, each taking at most $\log(\hat{n}\log\sigma)$ bits, where $\hat{n}$ is the longest character LCP value; plus b suffix pointers each of $\log n$ bits. No pointers are required in internal nodes, because all blocks are the same size, and addresses can be calculated rather than stored. The only pointers stored in the SB-TREE are to the text T rather than to disk blocks.

In total, a static SB-TREE for the n suffixes of a text T using a blocksize of b pointers requires

$$n\,(2 + \log(\hat{n}\log\sigma) + \log n) \tag{1}$$

bits, where $\hat{n} < n$. An SB-TREE index adds as much as 100% to the $n\log n$ bits required by a suffix array.

B. Variable Blocks and the LOF-SA

Sinha et al. [2] (including two of the current authors) describe the LOF-SA, a two-level index structure in which the block control parameter b is an upper bound, and suffix array blocks correspond to subtrees in the suffix tree. If v is a node in the suffix tree for text T and there are $size(v)$ leaves in the corresponding subtree, then a suffix array block is formed for node v if and only if $size(parent(v)) > b$ and $size(v) \le b$. All elements in the block share the prefix associated with v. The horizontal divisions in Figure 1 show the ten blocks that result when the example string is partitioned using $b = 3$; and Figure 2 shows how those ten block prefixes are stored in a bit-blind tree.

Sinha et al. use a trie for the in-memory component of the LOF-SA, but a trie has the disadvantage of a quadratic worst-case space requirement. A bit-blind trie, and the condensed BWT structure we present in Section V, both require less space in both the average case and the worst case.

Pattern search using the LOF-SA steps through the symbols in P, navigating the in-memory search structure, either until the pattern is exhausted, in which case all children of the node that was reached are answers to the query; or until a leaf in the trie is reached, in which case the answers, if any exist, are confined to a single block of the on-disk suffix array. In the latter case that block is fetched and searched.

Regardless of how the internal structure is organized, the variable sized disk blocks mean that a disk address of $\log n$ bits must be stored at each in-memory leaf. In the on-disk blocks, Sinha et al. also store an LCP value for each suffix; plus, as was previously sketched by Colussi and De Col [16], a small number f of extension symbols (the *fringe*) to help minimize search ambiguities. Search within a LOF-SA suffix block is sequential, capitalizing on the LCP and fringe values. Accesses are made to T only if there are gaps in the fringe that result in pattern uncertainty. Inclusion of the fringe for each suffix increases the size of disk blocks, and each entry in each on-disk suffix block contains an LCP value, a pointer into T, and a set of fringe symbols.

Sinha et al. undertook a range of experiments with 2 GB of DNA and 471 MB of English text, and patterns of length 6 to 1,000. With a blocksize bound of $b = 4{,}096$ and a fringe length of $f = 4$ characters, the in-memory component and on-disk component for the 471 MB English text file required 21 MB and 5.5 GB respectively, and yielded searching times around half or less of the SPAT, and around 8 times faster than a pure suffix array. Moffat et al. [3] considered compression of the on-disk components, and showed that the space required by the on-disk data can be reduced by approximately 40%, from $12n$ bytes down to around $7.1n$ bytes.

IV. REDUCIBLE BLOCKS

The next two sections describe our enhancements to the LOF-SA. First, in this section, we show that as many as half of the suffix pointers can be eliminated, via a process we call *block reduction*; we also remove the need for the LOF-SA's fringe characters, through the use of a bit-blind tree to store each of the suffix blocks. Then, in Section V we introduce a *condensed BWT* in-memory index structure that provides a unique mix of attributes and allows fast searching over a set of strings.

Figure 3 shows the overall structure of the new combination, which we call the ROSA, for *reduced*-space *on*-disk *suffix array*. A small search structure is maintained in memory and indexes variable-sized suffix array blocks on disk; the text T is also stored on disk. Each suffix array block stores suffix pointers to T, plus some navigational information, plus a bit-blind tree. A key innovation is that there is now a many-to-one relationship between leaves in the in-memory block index and suffix array blocks on disk.

A. Identifying Reductions

Given the LOF-SA's approach to forming suffix array blocks, a whole-block reduction is possible exactly when all of the BWT symbols corresponding to the suffixes contained in a block are the same. For example, in Figure 1, the suffixes corresponding to the prefix "$\mathtt{h}$", with pointers $\mathsf{SA}[6] = 1$ and $\mathsf{SA}[7] = 11$, form a block when $b = 3$; and both have an "$\mathtt{s}$" in

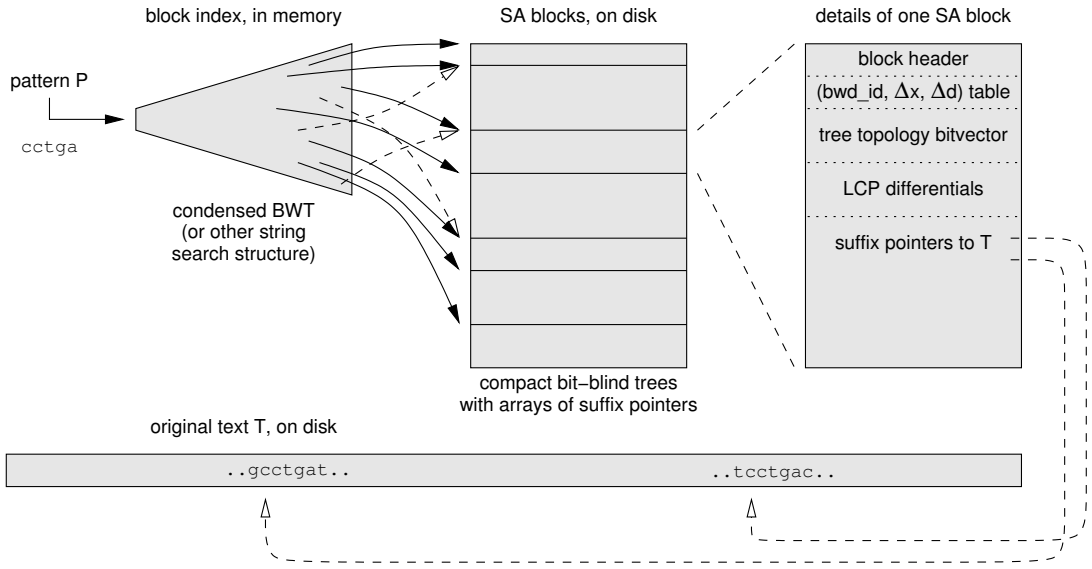

Fig. 3. Overall schematic of RoSA structure. Only the block index is in memory during querying. Solid arrows from the in-memory index represent irreducible blocks; dashed arrows represent reducible blocks. Block reductions mean that the relationship is many-to-one.

the column headed $\mathsf{L}[i]$. Hence, a reduction to the suffix "sh" is possible. Examination of the set of $b = 3$ blocks shown in Figure 1 reveals that the suffixes at offsets 8–9 for "ll" can be reduced to (a subset of) the block at suffixes at offset 3–5 for "e"; and that, via two such steps, the suffixes at offsets 10–11 for "ls" can be reduced to the same underlying block. The three arrows at the left of Figure 1 show the full set of block relationships that exist in the example string, with the three reducible blocks lightly shaded. Mäkinen and Navarro [9] also note that suffix array reductions can be used to save space.

B. Singleton Blocks

The LOF-SA variable block approach also sometimes generates blocks with just one pointer in them; we call these *singleton blocks*. They are unshaded in Figures 1 and 2, and represent another opportunity for space savings, since the corresponding suffix pointers to T can be stored directly in the in-memory index, rather than placed in a suffix block on disk. In the example string there are four singleton blocks. Only non-singleton *irreducible blocks* need to be placed onto disk; as can be seen in the example, there are three such blocks, and they contain a total of only seven suffix pointers.

C. Storing Information About Reductions

The details of the block reductions are held in a table of $(bwd\_id, \Delta_x, \Delta_d)$ triples in each irreducible block, where Δ_x is the offset from the start of the irreducible block at which the reduced block commences, Δ_d is the offset to be applied to each suffix pointer, and bwd_id is the backwards block number, defined in Section V. The three reducible blocks in Figure 1 are annotated with their Δ_x, Δ_d offsets. The leaves of the in-memory index only store a block number.

The table of triples contains one entry per suffix block (reducible or irreducible) that is hosted within that block's suffix pointers. Each of those entries is identified by the block's bwd_id. Information accumulated during the in-memory search (position reached in the pattern, and current suffix interval width) are combined with the corresponding Δ_x, Δ_d values and used to continue the search within the bit-blind tree used to represent the block. Note that the in-memory part does not differentiate between reducible and irreducible blocks at all – the latter correspond to $\Delta_x = 0$ and $\Delta_d = 0$. There is no "indirection penalty" caused by block reductions.

The in-memory structure identifies singletons by virtue of the fact that the search interval is one. Singletons are also reducible, by definition, but a search-time disk access can be saved if they point directly to T rather than via a suffix block. In Figure 2 non-singleton pointers are marked with a "b", but no such differentiation is required in practice, since singleton-block suffix pointers exactly correspond to situations where the $size$ field (shown at the bottom of Figure 2) is 1.

D. Storing the On-Disk Suffix Array

Each suffix block also stores a set of suffix pointers, plus a set of differential (relative to the parent) LCP values, plus two bits per leaf to indicate the tree structure, plus a small fixed overhead on the latter to allow $rank$ operations. Figure 3 shows this arrangement. One key advantage of the LOF-SA variable-block arrangement is that each block can store the LCP values (shown as $LCPdata$ in Figure 2) in compressed form, since there is no requirement that all disk blocks be the same size. This difference is significant in terms of space utilization.

In our implementation the LCPs are stored as differences relative to their parent in the suffix tree, and coded using the Elias δ code [17] with cumulative-sum samples inserted every 64 values to allow pseudo-random access to be carried out. The tree topology is stored as a balanced parentheses string for the Cartesian tree of the LCP values, as described by Gog and

Fischer [18]. Like the bv approach illustrated in Figure 2, the balanced parentheses string also consumes two bits per suffix. Rank and select structures to navigate the tree are calculated on-the-fly as the block is loaded from disk. The sizes of nodes can be computed in constant time during the traversal of the tree.

We also experimented with an alternative approach, in which absolute LCP values were stored and the tree structure and its rank and select structures were generated on-the-fly from the LCP values. This option turned out to be both larger in size and slower in operation, and was not pursued beyond preliminary experimentation.

V. Indexing Using a Condensed BWT

We now turn to the second major component of the RoSA – an efficient in-memory string index (the leftmost component in Figure 3). This section introduces a *condensed BWT* index that resolves *existence* and *count* queries for frequently appearing patterns (patterns that occur more than b times in T) without any suffix blocks needing to be retrieved, and without reference to T. The critical observation is that reversing each of the strings stored in the index allows backward search within them to match a forwards-prefix of P, which is what is required to identify a single block in the on-disk part of the RoSA.

A. Indexing the Blocks using a Bit-Blind Tree

The LOF-SA employs a suffix trie to store the set of block prefix strings, and hence requires quadratic space in the worst-case. A second option is to use a bit-blind tree (Section II-B). Figure 2 shows a bit-blind tree storing the block prefix strings for the example text. Each of the ten leaves corresponds to one of the blocks shown in Figure 1; only the irreducible blocks, shown with dark shading, need to be stored on disk.

When $\mathsf{bv}[x] = 0$ and x is the identifier of a leaf, the quantity $SAdata[x - rank(\mathsf{bv}, x, 1)]$ indicates where corresponding suffix pointer(s) are located, with $SAdata$ another dense array, containing either suffix array pointers, or suffix block disk addresses (indicated in the example by a "b" prefix). The *size* array also allows *count* queries to be handled efficiently.

In total, if there are K suffix array blocks, the structure shown in Figure 2 requires storage of: $2K$ bits for the tree structure; $K-1$ bit-LCP differentials, each of which is less than $n \log \sigma$; K suffix or disk pointers, each of which is less than n; and K block sizes, each of which is less than b. In the worst case, processing of a pattern P of length m requires navigation of the tree from the root to a leaf, involves $m \log \sigma$ bit-extraction operations and the same number of rank operations, and takes $O(m \log \sigma)$ time.

B. Backward Search in a Forward BWT

Ferragina and Manzini [12] show that pattern matching can be realized via the BWT string L. Suppose that a suffix $\omega = \mathsf{P}[m-i..m-1]$ of length i has been matched, and that the corresponding SA-interval is $[lb_i..rb_i]$. We denote this configuration with the notation $(\omega, i)[lb_i..rb_i]$. At the beginning of the search, $(\epsilon, 0)[0..n-1]$ is established. The new SA-interval $[lb_{i+1}..rb_{i+1}]$ for $\omega' = c\omega$ with $c = P[m-i-1]$ is contained within the section of SA corresponding to strings that commence with c. The offset from the start of that range is computed by counting the number of length-i substrings which are both lexicographically smaller than ω and preceded by c. Hence, $(c\omega, i+1)[\mathsf{C}[c] + rank(\mathsf{L}, lb_i, c)..\mathsf{C}[c] + rank(\mathsf{L}, rb_i + 1, c) - 1]$ is the next configuration of the backward search, where C is a σ-element array that stores in $\mathsf{C}[c]$ the location in SA of the first suffix commencing with symbol c, and can be computed when L is constructed.

The best approach for $rank$ on general sequences over a non-binary alphabet is to use a wavelet tree [19] or variant thereof, which reduces each operation to at most $\log \sigma$ operations over binary sequences. Here we use a Huffman-shaped tree using compressed bitvectors [20], which represents a sequence of symbols in its H_0 self-entropy. As already noted, on a binary alphabet, $rank$ and $select$ can be carried out in constant time by adding a fixed overhead on top of the original bitvector [4], [5].

C. Backward Search in a Condensed Backward BWT

A backward search in a reversed text is equivalent to a forward search in a forward text. Figure 4 shows the reversed example text in sorted suffix order, with a number of divisions marked on the right-hand side. The column headed $\mathsf{L}^{\mathsf{T}^r}[i]$ shows the full BWT of the reversed text; but for our purposes only a subset of the BWT is required, shown in the example as CL = "`s#lelshe$#`". To allow positions in the condensed BWT to be mapped to their positions in $\mathsf{L}^{\mathsf{T}^r}$, the bitvector bf is used, with $\mathsf{bf}[i] = 1$ when the predecessor symbol of the ith suffix is in CL. Similarly, bitvector $\mathsf{bl}[i] = 1$ if the ith entry of $\mathsf{L}^{\mathsf{T}^r}$ appears in CL. The run-length compressed FM-Index of Mäkinen and Navarro [21] makes use of auxiliary bitvectors in a similar manner to what we are about to describe.

Consider the suffix strings on the right-hand side of Figure 1. The block-prefixes (shown by the shading) that need to be reversed and indexed are "`$`", "`#`", "`e`", "`h`", "`ll`", "`ls`", "`s$`", "`s#`", "`se`", and "`sh`". When reversed, they become "`$`", "`#`", "`e`", "`h`", "`ll`", "`sl`", "`$s`", "`#s`", "`es`", and "`hs`". To create the bitvector bf that indicates which of the BWT characters are needed in the condensed BWT, the interval $[lb, rb]$ associated with each of these reversed strings, and each prefix of them, is located in the reversed BWT, and the bits $\mathsf{bf}[lb]$ and $\mathsf{bf}[rb+1]$ are set to 1, to mark the beginning and end of each search interval that might be required. Any locations in bf with 1-bits at the end of this stage have their corresponding first suffix character located in $\mathsf{L}^{\mathsf{T}^r}$ and copied in to CL; and an inverse mapping bl is computed that stores the locations extracted. For example, in Figure 4 the first and fourth suffixes commencing with "`s`" are tagged in bf. Those "`s`" symbols occur in positions $\mathsf{L}^{\mathsf{T}^r}[0]$ and $\mathsf{L}^{\mathsf{T}^r}[10]$, so both $\mathsf{bl}[0]$ and $\mathsf{bl}[10]$ are set to 1, and two "`s`" symbols appear in CL. Finally, a set of condensed symbol counts CC is formed from the condensed BWT string CL.

Figure 5 details the backward search for a pattern P using the condensed BWT CL and corresponding counts CC. As for regular backward search, an interval is maintained, initially

Fig. 4. Full BWT text $\mathsf{L}^{\mathsf{T}^r}$, condensed BWT text CL, and indexing bitvectors bf and bl for the reversed text $\mathsf{T}^r =$ "sllehs#slles#ehs$".

```
00  get_interval(P, m)
01    d ← 0; lb ← 0; rb ← n − 1
02    while d < m and rb − lb + 1 > b do
03      c ← P[d]
04      (lb′, rb′) ← (rank(bl, lb, "1"), rank(bl, rb + 1, "1"))
05      (lb″, rb″) ← (rank(CL, lb′, c), rank(CL, rb′, c))
06      if lb″ = rb″ then
07        return not_found
08      lb ← select(bf, CC[c] + lb″, "1")
09      rb ← select(bf, CC[c] + rb″, "1") − 1
10      d ← d + 1
11    return (P[0..d − 1], d)[lb..rb]
```

Fig. 5. Backward search using a condensed BWT text CL and a condensed count array CC.

$(\epsilon, 0)[0..n-1]$. That interval is then narrowed using the condensed arrays, adding one more character into the matched string at each iteration of the loop. The search commences with the rightmost symbol in the reverse of P, which is the leftmost symbol in P; and (in the frame of reference established in Figure 4) prepends subsequent matched characters to the left. In particular, the search process maintains

$$lb = \min\{k \mid \mathsf{T}[\mathsf{SA}[k]..\mathsf{SA}[k]+d-1] = \mathsf{P}[0..d-1]\}$$

as the first suffix in SA that matches P to depth d, and

$$rb = \max\{k \mid \mathsf{T}[\mathsf{SA}[k]..\mathsf{SA}[k]+d-1] = \mathsf{P}[0..d-1]\}$$

as the last such suffix.

To step from one configuration to the next, symbol $\mathsf{P}[d]$ must be processed, with lb and rb updated so that the assignment $d \leftarrow d+1$ then restores the invariant. To narrow the (lb, rb) interval the process described by Ferragina and Manzini [13] is used, but with an added level of complexity:

```
00  get_bwd_id(lb, d)
01    run_nr ← rank(bf, lb, "1")
02    if run_nr = 0 then
03      return 0
04    run_pos ← select(bm, run_nr − 1, "1") + 1
05    x ← min_depth[rank(bm, run_pos, "10")]
06    return run_pos − run_nr + (d − x)
```

Fig. 6. Determining the block identifier matching a reverse search configuration $(\omega, d)[lb..rb]$.

lb and rb are first translated into the condensed domain, then processed against the condensed BWT CL in that domain, and finally translated back to the full domain, ready for the next iteration. Those transformations are specified by the bitvectors bl and bf.

For example, to match P = "she", the first iteration processes the "s", and the configuration becomes ("s", 1)[12..16]. Then a second iteration in which the "h" is processed results in the configuration ("hs", 2)[6..7]. Now the interval is smaller than $b = 3$, so the in-memory search is ended, and the indicated suffix block (backward identifier 7, forward identifier 9) is fetched. A search for "shy" would also require that block 9 be accessed before the search could be declared a failure. On the other hand, the pattern "say" generates the (condensed domain equivalent of the) empty configuration ("as", 2)[3..2] at Step 05 after two iterations, and reports failure at Step 07 without any access being needed to a suffix block.

D. Computing Block Numbers

Once a configuration $(\omega, d)[lb..rb]$ has been established by *get_interval*(), the next step is to map it to a *bwd_id* block number; that is, identify the correct gray superscript value associated with the black block identification circles

TABLE I
STRUCTURES REQUIRED IN MEMORY DURING ROSA PATTERN MATCHING. THE VALUE z IS THE NUMBER OF ENTRIES IN EACH OF bf AND bl. IF THERE ARE B SUFFIX BLOCKS, THEN $z \leq \min\{4B, n\}$. THE FINAL TWO COLUMNS SHOW THE ACTUAL COST FOR TEST FILE WEB-64000, DESCRIBED IN TABLE II, PLUS THAT SIZE EXPRESSED AS A MULTIPLE OF $B \log n$ BITS, WITH $b = 4{,}096$, AND $B = 219{,}319{,}568$ BLOCKS GENERATED.

Structure	Type	Operations	Parameters	Space (upper-bound, bits)	Space (actual, MB)	$\times B \log n$
bf	bitvector	*select*	z elements, each $0 \leq x \leq n$	$z(2 + \log(n/z)) + o(z)$	135.3	0.144
bl	bitvector	*rank*	z elements, each $0 \leq x \leq n$	$z(2 + \log(n/z)) + o(z)$	135.3	0.144
bm	bitvector	*rank*/*select*	$2B$ elements, each $0 \leq x \leq n$	$2B(1 + \log(n/B)) + o(B)$	37.4	0.040
min_depth	array	*access*	B elements, each $0 \leq x \leq n - B$	$B \log n$	72.3	0.077
CC	array	*access*	σ integers, each $0 \leq x < z$	$\sigma \log n$	<0.1	<0.001
CL	array	*rank*	z symbols, each $0 \leq x < \sigma$	$O(zH_0(\mathsf{CL})) = O(z \log \sigma)$	74.1	0.079
pointers	array	*access*	B elements, each $0 \leq x \leq n$	$B \log n$	967.4	1.023

TABLE II
DETAILS OF DATA FILES. THE VALUE OF H_k IS EMPIRICAL, GENERATED BY EXECUTING `xz --best`.

Name	Type	Size (MB)	σ	H_k (bits/char)	LCP		
					Median	Average	Maximum, $\hat{n}$
WEB-256	HTML/Web	256	129	0.45	141	5,937	556,673
WEB-4000	HTML/Web	4,000	129	0.57	281	11,506	692,160
WEB-64000	HTML/Web	64,002	129	0.61	1,896	20,500	1,204,953
DNA-3000	Text/Genomic	2,985	9	1.65	16	554,171	29,999,999
DBLP-1000	XML/Bibliographic	1,032	99	0.90	36	45	1,353

in Figures 1 and 4. Because multiple blocks might map to the same lb value but with different depths d, a further bitvector bm is required, containing a 0-bit for each block in the forward suffix array, plus a 1-bit for each 1-bit in bf, corresponding to blocks in the reversed suffix array. The bits are interleaved so that each entry point in bm is preceded by a string of 0-bits that indicates the number of disk blocks converging at that entry point. The process of mapping via that structure, plus another array of integers that records the minimum configuration depths at each valid entry point, is described in Figure 6.

The block number is next converted to an on-disk byte address via an array storing a mapping that is many-to-one because of the reducible blocks (not shown in Figure 3). That block is fetched, and the required bwd_id located in the block's header, to identify the matching (Δ_x, Δ_d) region or subregion of the block at which the search should be continued.

E. Space Requirement

The bitvectors and arrays required in memory during querying are summarized in Table I. The symbols extracted into the condensed BWT are exactly those required during searching for any of the block prefix strings. No BWT symbols that would only be accessed if $rb - lb$ was permitted to become smaller than b are needed. At most two bits are required for each node in the corresponding frequency-pruned suffix tree, and that tree contains at most $2B$ nodes if the ROSA contains B disk blocks. The maximum number of bits that can be set is n, meaning that the actual number of bits set, z, is bounded by $z \leq \min\{4B, n\}$. When b is large, B can be expected (but not guaranteed) to be small, making the bitvectors bf and bl sparse and highly compressible; and making the CL and CC arrays that represent the condensed BWT small too.

F. Execution Time

Function $get\_interval()$ in Figure 5 iterates at most once for each character in the pattern. A total of two bitvector $rank$ operations and two bitvector $select$ operations are required per iteration; each of these take $O(1)$ time. Step 05 involves $rank$ operations on an array, CL. That array is implemented as a Huffman-shaped wavelet tree, based on underlying bitvectors, meaning that symbol-based $rank$ queries can be carried out via not more than $\log \sigma$ bitvector-based $rank$ queries, or in $O(\log \sigma)$ time. The process of finding the matching block identifier (function $get\_bwd\_id()$ in Figure 6) involves only rank and select operations on bitvectors, and takes $O(1)$ time per pattern.

We now combine these arguments, and state the main result of this section.

THEOREM 1: *Given a set of B strings corresponding to the leaves of a pruned suffix tree for a text of n symbols, the condensed BWT structure requires $O((B + \sigma) \log n)$ bits of storage and identifies the leaf corresponding to an m-symbol pattern in $O(m \log \sigma)$ time.*

VI. EXPERIMENTS

We have implemented the ROSA so that it is 64-bit compliant, and compared it against a range of alternatives.

A. Experimental Hardware and Methodology

Experiments were run on two different hardware platforms: a MacBook Pro with a 2.4 GHz Intel Core i5 processor, 4 GB RAM, and 500 GB hard disk; and a MacBook Air with 1.8 GHz Intel Core i7 processor, 4 GB RAM, and a 250 GB solid-state disk. The suffix array itself was prepared on a separate server with considerably more memory than the test machine.

The methodology used in each experimental run was to start the program; read the in-memory index; start the timing clock; execute 1,000 queries; stop the timing clock; and report "elapsed (wall clock) time divided by 1,000" as being the average query time. That is, each run started from a "cold" configuration in terms of the processor cache and buffering of disk blocks, but times were measured over a long sequence of patterns.

B. Test Data

Data was obtained from a range of sources, with an emphasis on large files. The first suite of test files were drawn from the 2009 CLUEWEB collection, a large-scale web crawl[2]. Three files were extracted as prefixes of the concatenation of the first 64 files in the directory ClueWeb09/disk1/ClueWeb09_English_1/enwp00/, with null bytes in the text replaced by 0xFF-bytes. (Null byte is the "$" symbol reserved in all our implementations to mark the end of the input string.) In Table II these three files are denoted as WEB-256, WEB-4000, and WEB-64000. Two other types of data were also used: file DNA-3000 is a text file representing the human genome stored as a sequence of ASCII letters (primarily "A", "C", "G", and "T"); and file DBLP-1000 is an XML repository containing 844,702 bibliographic references to computing research papers[3].

The three different types of data differ markedly in the extent to which they contain sequence repetitions. In the web data the LCP values are particularly high, caused by reuse of formatting text, and by duplicate documents. The median LCP is much lower for the XML and DNA data; but note that the file DNA-3000 contained a repeated subsequence of thirty million characters. The three data types also differ in the size of the alphabet used, and in compressibility. To estimate the latter quantity, the column marked H_k shows the compression achieved by a high-quality mechanism, expressed in terms of bits per character relative to the original. The web and XML data are highly compressible; the DNA file somewhat less so.

C. Test Patterns

To generate test queries, a suffix tree representation of each file was processed sequentially, and a large set of ⟨pattern, frequency⟩ pairs identified. These were then quantized by both pattern length and by pattern frequency, with agreement assumed in the second dimension if the actual frequency was within 25% of one of a set of target frequencies. This approach allowed a total of 25 different query sets to be formed for each file, representing all combinations of $|\mathsf{P}| \in \{4, 10, 20, 40, 100\}$ and pattern frequency $k \in \{10^0, 10^1, 10^2, 10^3, 10^4\}$. On the web data, all combinations occurred more than 1,000 times, and experiments were run on random subsets of size 1,000 drawn from the corresponding category. Selected combinations of $|\mathsf{P}|$ and k were used for the other data files, and results are similarly the average over 1,000 patterns. It was not possible to identify any patterns with $|\mathsf{P}| = 4$ and $k = 10{,}000$ on DNA-3000, and as a result one entry is omitted in the tables below.

[2]http://lemurproject.org/clueweb09.php/

[3]http://dblp.uni-trier.de/xml/

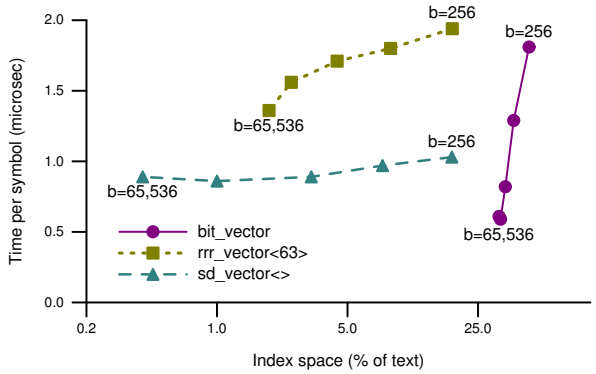

Fig. 7. Space and processing time of in-memory search using condensed BWT approach as a function of blocksize, for three different bitvector representations. Data is for WEB-256, averaged over 1,000 patterns with $|\mathsf{P}| = 40$ and $k = 10{,}000$ matches per pattern, and with the blocksize varying between $b = 2^8$ and $b = 2^{16}$.

TABLE III
PERCENTAGE OF INDEX SPACE REQUIRED BY COMPONENTS OF CONDENSED BWT INDEX FOR WEB-4000.

Component	$b = 2^8$	$b = 2^{12}$	$b = 2^{16}$
Bitvectors (bf, bl; SD-array)	14.3	22.4	31.7
Condensed BWT (CL; wavelet tree)	5.2	6.5	7.8
Auxiliary information	7.7	8.9	9.7
Pointers (binary)	72.8	62.2	50.7

D. Compressed Bit Vectors

A key decision is how to represent the two large bitvectors bf and bl. Conceptually each of them contains n bits, but, by construction, the number of 1 bits is close to the number of suffix array disk blocks, and so they are sparse and amenable to compression. The drawback of compression is that *rank* and *select* operations become slower. Figure 7 compares the space and access cost of three different representations for the two bitvectors, with space plotted on the horizontal axis, measured as the ratio of the complete condensed BWT data structure to the text size; and processing time per matched character plotted vertically. The alternatives are denoted by their `sdsl` class identifiers[4]: uncompressed bitvectors (class `bit_vector`); the well-known RRR structure [20] (`rrr_vector<63>`); and the SD-array (`sd_vector<>`) of Okanohara and Sadakane [22]. The SD-array offers the best balance, and while it is not always faster than the uncompressed bitvector alternative, it occupies much less space.

Once the bitvectors are compressed, the disk block pointers are the most costly component of the condensed BWT index. These are addresses into the index (for irreducible and reducible blocks) or into the text (for singletons), and are represented as minimal-width binary numbers. Table III shows the percentage of the total memory space required by

[4]https://github.com/simongog/sdsl

TABLE V
IN-MEMORY SEARCH STRUCTURES FOR VARIABLE SUFFIX ARRAY BLOCKS: SPACE AND SPEED AS A FUNCTION OF BLOCKSIZE b.

Data	b	Memory (MB)		Query speed (microseconds/query)	
		Condensed BWT	Bit-blind tree	Condensed BWT	Bit-blind tree
WEB-4000	2^{10}	269.8	329.1	33.7	36.3
WEB-4000	2^{12}	98.6	112.2	24.3	31.5
WEB-4000	2^{14}	15.8	15.1	19.7	28.6
DBLP-1000	2^{10}	58.3	58.4	26.8	29.4
DBLP-1000	2^{12}	21.1	18.9	19.2	24.2
DBLP-1000	2^{14}	7.6	6.2	15.6	19.9
DNA-3000	2^{10}	410.2	382.8	29.7	24.3
DNA-3000	2^{12}	342.9	319.3	21.1	21.0
DNA-3000	2^{14}	326.6	307.8	17.8	17.3

TABLE IV
TOTAL MEMORY AND DISK SPACE REQUIRED FOR TWO-LEVEL SUFFIX ARRAY STRUCTURES AND THE FM-INDEX, FOR WEB-4000. THE VALUES MARKED * ARE COMPUTED USING TEXT STATISTICS. THE OTHER VALUES ARE MEASURED USING AN IMPLEMENTATION.

Structure		Ref.	Size (GB)
Suffix array		[1]	15.6
LOF-SA	$b = 4{,}096$	[2]	46.9*
LOF-SA	$b = 4{,}096$	[3]	27.3*
SB-TREE	$b = 4{,}096$	[6]	24.5*
ROSA	$b = 4{,}096$	*this paper*	7.8
FM-INDEX		[13]	0.6

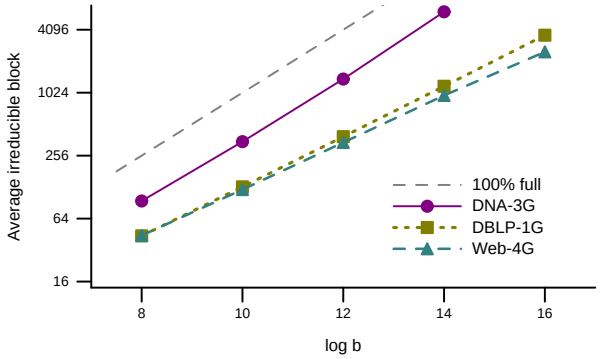

Fig. 8. Average size of irreducible blocks (in pointers).

each of the four main components of the condensed BWT search structure, for the file WEB-4000 and three different blocksizes. The dominance of the pointers is clear.

E. Baseline Methods and Total Disk Space

The ROSA structure – consisting of condensed in-memory BWT array index, and a set of suffix array blocks stored on disk – can be compared with the LOF-SA (which in turn is compared by Sinha et al. [2] against previous data structures); with the SB-TREE; and with the FM-INDEX. The FM-INDEX is not a two-level disk-based mechanism, and can only be used if the complete structure fits main memory. Nevertheless, it is substantially smaller than the other structures, meaning that its zone of applicability overlaps the size range for which two-level structures are appropriate.

Table IV compares index sizes for these various approaches, including both components for the two-level ones. The values for the ROSA and FM-INDEX are measured based on our experimental implementations. There is no software for the SB-TREE or LOF-SA capable of handling the data sizes used in our experiments, and the values shown in the table marked with "*" are computed using Equation 1 (in Section III) for the SB-TREE, and estimated from the results given by Sinha et al. [2], [3] for the LOF-SA. With the exception of the FM-INDEX, all of these structures require that the text T also be stored, adding a further 3.9 GB.

The block reductions achieved in the ROSA mean that it is the smallest of the two-level approaches. Indeed, the ROSA index requires just half the space of a plain suffix array. On the other hand, the SB-TREE and the LOF-SA are expensive to store; neither of these structures support block reductions, and in the case of the SB-TREE, the LCP values are also a costly component because the fixed block structure means that they cannot be stored compressed. Because of their clear space superiority, the remainder of the experimentation focuses on the ROSA and the FM-INDEX alone.

F. Choice of In-Memory Structure

The second step of the experimental evaluation was to compare the condensed BWT method with the bit-blind tree, in terms of memory space required and search time to identify suffix blocks (Table V). Search times are measured over frequently-occurring long queries ($|\mathsf{P}| = 40$ and $k = 10{,}000$, so that the search is driven towards the extremities of the in-memory structure); and include only the cost of processing the in-memory data structure. As is shown in the table, the two methods are broadly comparable in terms of space and speed for the in-memory computation. But note that Table V does not include the cost of the disk accesses to T needed to resolve the uncertainty inherent in the bit-blind search process. Details of disk access costs are presented shortly; the condensed BWT arrangement has a clear advantage when that cost is included.

G. Blocksizes and Non-Uniform Sampling

Figure 8 depicts the average number of pointers stored in each irreducible block for three of the test files. The growth in average block size is linear in the size of the block, but for the non-genomic data the average is well below the limit b. This

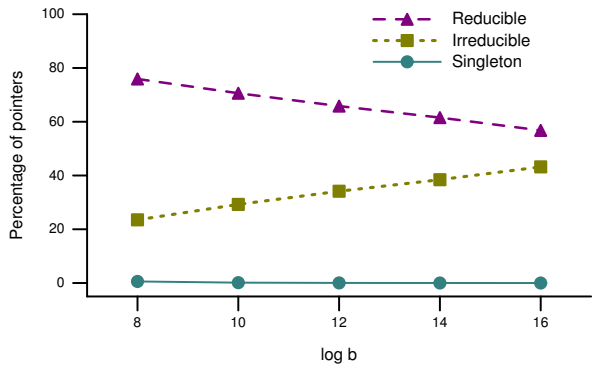

Fig. 9. Fraction of pointers in reducible, irreducible, and singleton blocks for WEB-4000 and different values of b.

TABLE VI
SPACE REQUIRED BY ROSA QUERY-TIME INDEX COMPONENTS WITH $b = 4{,}096$, EXPRESSED AS MULTIPLES OF THE SOURCE TEXT SIZE.

File	Memory	Disk	Total, inc. T
WEB-256	0.033	1.943	2.976
WEB-4000	0.025	1.961	2.986
WEB-64000	0.022	1.900	2.922
DBLP-1000	0.020	2.126	3.146
DNA-3000	0.116	4.704	5.820

relationship is not unexpected – blocks are formed at nodes of the suffix tree whenever the parent has a count of more than b, but the node in question does not. At that boundary node, the available symbol count is split across all of the children. Hence when the alphabet size σ is large, those child counts will, on average, be relatively small.

Figure 9 shows the fraction of the suffix pointers located in reducible blocks, irreducible blocks, and singleton blocks for WEB-4000. When b is small, more than two thirds of the suffix pointers are in reducible blocks. That fraction decreases as b increases, not because the reductions are no longer present, but because the similar sections no longer span whole blocks. But even when $b = 65{,}536$, around half of the suffix pointers can be eliminated. Similar behavior was observed for DBLP-1000. On the other hand, the DNA data has markedly different characteristics, and while it generates many more singleton blocks, the number of block reductions is very small.

Table VI shows the balance between in-memory space and on-disk space required by the ROSA for the full set of data files. For the web and XML data, the total space required is much less than would be required by a plain suffix array (a factor of 4.75 for DBLP-1000, and of 5.0 for WEB-4000). On the other hand, the ROSA handles the DNA data relatively poorly, and both the in-memory index and the on-disk component are large. Indeed, on the DNA data the ROSA takes more space than a plain suffix array, a consequence of the relative absence of repetitions. It is, of course, still faster than a plain suffix array.

TABLE VII
DISK ACCESSES PER COUNT QUERY FOR FILE WEB-4000, WITH $b = 4{,}096$.

|P|	Number of answers				
	1	10	100	1,000	10,000
4	1.79	1.52	1.12	0.35	0.00
10	1.99	1.99	1.94	1.70	0.00
20	2.00	1.99	1.98	1.83	0.00
40	2.00	2.00	1.99	1.90	0.00
100	2.00	2.00	2.00	1.95	0.00

(a) Condensed BWT

|P|	Number of answers				
	1	10	100	1,000	10,000
4	1.86	2.00	2.00	2.00	1.84
10	1.99	2.00	2.00	2.00	1.87
20	2.00	2.00	2.00	2.00	1.90
40	2.00	2.00	2.00	2.00	1.87
100	2.00	2.00	2.00	2.00	1.94

(b) Bit-blind tree

H. Disk Accesses and Execution Cost For Count Queries

Table VII shows the number of disk accesses required by the two options for the in-memory structure. The benefit of the condensed BWT arrangement is clear – because it admits no ambiguity, fewer disk accesses are required for *count* queries when the pattern is common in the text and can be resolved entirely within the in-memory index. When the pattern is frequent, the discrepancy is even greater – the condensed BWT allows *count* queries to be processed without recourse to disk, whereas the bit-blind tree still requires an average of more than 1.8 disk accesses per query.

Table VIII shows overall elapsed times for a range of query lengths and frequencies across the set of data files (including the 64 GB file), for two hardware platforms. The in-memory condensed BWT index for WEB-64000 requires 1.39 GB (around two-thirds of which is pointers, as shown by the last two columns of Table I), and the on-disk part a total of 119 GB, with the latter composed of 1.4 GB for block headers and other auxiliary data; 29.5 GB for compressed LCP differentials and for tree structure bits; and 82.7 GB for suffix pointers. Including the text T, the entire search system requires 183 GB, a factor of 2.9 relative to the text, a little over half of the 5.5-factor that would be required by a plain suffix array.

As can be seen, access via SSD memory is much faster than access via mechanical disk. But even with the mechanical disk, pattern queries on WEB-64000 can be answered by the ROSA in under 50 milliseconds. Moreover, search times are largely unaffected by pattern length, except that queries on frequently-occurring strings are always handled within a small number of microseconds.

I. Compared to the FM-INDEX

The last three rows of Table VIII show the query cost of a highly-tuned (for both space and speed) FM-INDEX implementation that has been demonstrated to outperform all alternatives [5, Section 6.6]. For WEB-4000, a run-length compressed wavelet tree and SD-array implementations for the two FM-INDEX bitvectors was used, the fastest configuration. During querying, this FM-INDEX version requires 659.4 MB

TABLE VIII
EXECUTION TIMES IN MILLISECONDS PER QUERY, USING TWO DIFFERENT HARDWARE PLATFORMS, WITH $b = 4{,}096$.

Text	Platform	$\|\mathsf{P}\| = 4$ $k = 10{,}000$	$\|\mathsf{P}\| = 10$ $k = 1{,}000$	$\|\mathsf{P}\| = 20$ $k = 100$	$\|\mathsf{P}\| = 40$ $k = 10$	$\|\mathsf{P}\| = 100$ $k = 1$
Using the ROSA						
DBLP-1000	MacBook Air, SSD	0.004	1.02	1.10	1.09	1.13
DNA-3000	MacBook Air, SSD	—	0.72	1.10	1.15	1.23
WEB-4000	MacBook Air, SSD	0.006	1.00	1.06	1.06	1.05
WEB-64000	MacBook Air, SSD	0.009	0.98	1.04	1.09	1.13
DBLP-1000	MacBook Pro, mechanical disk	0.005	21.1	25.5	24.8	26.5
DNA-3000	MacBook Pro, mechanical disk	—	14.9	25.3	25.8	26.7
WEB-64000	MacBook Pro, mechanical disk	0.009	33.9	40.3	40.7	44.6
Using an efficient FM-INDEX						
WEB-4000	MacBook Air, SSD	0.011	0.03	0.07	0.14	0.36
WEB-64000	MacBook Air, SSD	44.6	85.6	88.9	118.9	70.0
WEB-64000	MacBook Pro, mechanical disk	630	1450	2040	2500	980

of memory space. For short *count* queries it is much faster than the ROSA. With a different bitvector representation (using the RRR variant), space can be reduced to 404.6 MB, but querying time increases by a factor of around three.

For WEB-64000 (the last two lines of Table VIII), the more compact RRR bitvector option was used, requiring 8.3 GB for the index. As can be seen, when only a subset of a large index can be maintained permanently in memory, the non-sequential access pattern means that retrieval times increase dramatically. When SSD disk is used the times are still somewhat plausible, but the two-second response times that arise when a mechanical disk is used are anything but plausible. The sequence of results in Table VIII clearly highlights the situations for which the ROSA is the fastest search mechanism.

J. Construction and Applicability

Despite recently developed techniques [23], a drawback of all suffix array-based pattern search methods is the cost of building the suffix array. The structures used in our experiments were generated on a server with considerably more memory than the laptops that were used for the search experiments, and reflect the situation for which we believe static two-level structures are best suited – namely, when large fixed texts are to be pre-processed by a central service to make "searchable packages" that can be distributed onto low-cost devices for querying purposes.

The FM-INDEX is a strong competitor for the same type of applications. It has approximately the same construction cost, but a much smaller query-time disk storage footprint. The disadvantage of using an FM-INDEX is that for any given text T, its memory requirement is likely to be greater than that of the ROSA, because the entire structure must be present in memory. That is, there is a size of text for which an FM-INDEX cannot be supported by the available hardware, but a ROSA can, albeit with significantly greater disk storage consumption. Depending on the exact configuration used, *locate* and *context* queries might also be slower in an FM-INDEX than using the ROSA.

It is also interesting to calculate the break-even point at which a pre-computed data structure becomes more economical than sequential search. Construction of the ROSA for WEB-4000 requires around 100 minutes, and the current implementation involves a peak memory requirement of $9n$ bytes during the two suffix sorting steps (external methods for suffix sorting are available that reduce the memory cost, but increase the construction time). Using the MacBook Pro to search the same 4 GB file for patterns using `agrep`[5] requires about three seconds, once the file containing T has been brought in to memory. Hence, construction of a ROSA index is warranted if more than around $2{,}000$ queries are to be processed against the same text T.

VII. OTHER RECENT WORK

Ferragina et al. [24] describe *xbw*, a searchable succinct tree representation over sets of strings which takes $2t+t\lceil\log\sigma\rceil$ bits of space, where t is the number of nodes in the tree. Although it is an optimal-space tree representation, *xbw* cannot be used in our scenario, since in the worst-case the in-memory trie consists of a quadratic number of labeled nodes. This worst case disqualifies the use of any tree structure in which size is dependent on the number of nodes, including the trie approach suggested for use in the original LOF-SA [2].

Phoophakdee and Zaki [25] describe a partition/merge approach to suffix tree construction that allows them to undertake pattern search on a human genome. They compare their TRELLIS approach to other options on files of up to three billion DNA base pairs, with a build time of under six hours, and a final size of 71.6 GB, or 27 times larger than the input text. Using their suffix tree, they are able to undertake queries of $100+$ base pairs in approximately 60 milliseconds.

Wong et al. [26] describe a partitioned suffix tree they call a CPS-TREE. They experiment with files of 118 million base pairs and 4.6 million base pairs, and obtain suffix trees that require between $7n$ and $9n$ bytes. With these small test files, querying is fast – of the order of 20 microseconds per query – because it still takes place in main memory.

Orlandi and Venturini [27] have also described a structure for storing a pruned suffix tree. Their pruning definition differs from the one used in the ROSA, and they retain a node if its size is greater than b, whereas in the ROSA a node appears in the condensed BWT structure if its *parent* is of size greater

[5] ftp://ftp.cs.arizona.edu/agrep/.

than b. The difference means that care must be taken when comparing sizes for a given parameter value, since the RoSA retains as many as σ times more tree nodes than does the CPST, including, for example, singleton blocks.

For a CPST over n symbols in which there are K suffix tree nodes retained each of size b or more, the space required by Orlandi and Venturini's structure is $O(K \log(\sigma b) + \sigma \log n)$ bits. Direct comparison with the costs shown in Table I is not possible, because for any given value of b the number of nodes K in the CPST is much less than the number of leaves B in the RoSA index structure. The RoSA's condensed BWT index provides greater functionality, since it retains frequency counts for (lb, rb) intervals narrower than b, whereas the CPST replies to *locate* and *count* queries on rare and non-existent patterns with a uniform answer of "don't know; if P does exist, it appears fewer than b times". The RoSA also stores disk block pointers, a component that is not required in the CPST. Orlandi and Venturini [27] also describe a uniform-sampling index in order to undertake approximate *count* queries, where the returned pattern frequency in *count* queries is correct to within an additive fidelity constraint determined at the time the index is constructed. Building a CPST requires initial construction of a suffix tree, and needs more resources than creation of the BWT string, the basis of the RoSA's construction process.

Other recent work is by Ferguson [28], who describes a search structure called FEMTO, and provides experiments on 43 GB of English text (Project Gutenberg files), and on 182 GB of genomic data. The FEMTO system uses a partitioned FM-INDEX, with the search for each pattern proceeding through (at least) one disk block per symbol. Ferguson gives experimental results showing that the constructed index requires as little as half of the space of the original file, but with query response times of 1–3 seconds for *count* queries against selected patterns of 12–28 symbols (two to three word phrases, with tests carried out on an individual basis on hand-selected strings, rather than as part of a regime of extensive measurement) against the English text when using a conventional disk drive; and of 10 or more seconds when searching the Genomic data for patterns of length 128. The high search times arise because of the disk accesses. When multiple queries are simultaneously active, and duplicate requests for disk blocks can be batched and processed all at once, throughput improves dramatically, but with a corresponding increase in individual response times. Compared to the FEMTO, the methods presented here require more disk space for the suffix array data, but operate an order of magnitude more quickly.

Another approach to large-scale pattern search is to index overlapping t-grams from T, each containing t consecutive symbols. In total, $n - t + 1$ locations in T are indexed via a vocabulary containing at most $O(\sigma^t)$ entries. An inverted index is built, storing a variable-length postings list for each unique t-gram, and recording the locations in T at which that particular combination of t symbols appears [29]. Queries of length $m > t$ are resolved by intersecting the relevant postings lists, identifying locations at which fragments overlap in the desired manner; queries of length $m \leq t$ are resolved by taking the union of the postings lists of the vocabulary entries that contain P within the t-symbol identifier.

Inverted indexes allow queries to be resolved in (at most) two disk accesses per query term, one to retrieve a block of the vocabulary (if it is not stored in memory) and one to retrieve a postings list [17]. If t is chosen so that the t-gram vocabulary for T can be held in main memory, the number of disk accesses required to match a pattern P and resolve *locate* queries is $\lceil m/t \rceil$. In terms of space, a t-gram index with $t \approx 5$ to 10 can be expected to consume around 150–200% of the space required by T, and to grow larger as t increases. Note that in the t-gram approach to pattern search T is not required in memory. Tang et al. [30] give details of the construction and use of n-gram indexes for pattern matching. Puglisi et al. [31] have also examined this problem.

VIII. SUMMARY

We have carried out a detailed investigation of two-level suffix-array based pattern search mechanisms, and: (1) described an efficient mechanism for exploiting whole block reductions, to approximately halve the space required by the suffix array pointers; and (2) described and analyzed a condensed BWT mechanism for storing and searching the string labels of a pruned suffix tree. We have demonstrated that in combination these techniques provide efficient large-scale pattern search, requiring around half the disk space of previous two-level techniques, and providing faster search than an FM-INDEX when the data is such that the FM-INDEX cannot be accommodated in main memory. While we have focused on the memory-disk interface, we note that structures with the properties exhibited by the RoSA are effective across all interface levels in the memory hierarchy. In future work we plan to make use of the suffix block prefix strings as a dictionary of phrases with which to compress the space required by the text.

ACKNOWLEDGMENT

This work was funded by the Australian Research Council. The RoSA software is at https://github.com/simongog/RoSA.

REFERENCES

[1] U. Manber and G. W. Myers, "Suffix arrays: a new method for on-line string searches," *SIAM J. Computing*, vol. 22, no. 5, pp. 935–948, 1993.
[2] R. Sinha, S. J. Puglisi, A. Moffat, and A. Turpin, "Improving suffix array locality for fast pattern matching on disk," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2008, pp. 661–672.
[3] A. Moffat, S. J. Puglisi, and R. Sinha, "Reducing space requirements for disk resident suffix arrays," in *Proc. Conf. Database Systems for Advanced Applications*, 2009, pp. 730–744.
[4] S. Vigna, "Broadword implementation of rank/select queries," in *Proc. Int. Wkshp. Experimental Algorithmics*, 2008, pp. 154–168.
[5] S. Gog and M. Petri, "Optimized succinct data structures for massive data," *Software Practice & Experience*, 2013.
[6] P. Ferragina and R. Grossi, "The string B-tree: A new data structure for search in external memory and its applications," *J. ACM*, vol. 46, no. 2, pp. 236–280, 1999.
[7] J. Kärkkäinen and S. S. Rao, "Full-text indexes in external memory," in *Algorithms for Memory Hierarchies*, 2002, pp. 149–170.
[8] G. Manzini and P. Ferragina, "Engineering a lightweight suffix array construction algorithm," *Algorithmica*, vol. 40, no. 1, pp. 33–50, 2004.
[9] V. Mäkinen and G. Navarro, "Compressed compact suffix arrays," in *Proc. Symp. Combinatorial Pattern Matching*, 2004, pp. 420–433.

[10] R. González and G. Navarro, "Compressed text indexes with fast locate," in *Proc. Symp. Combinatorial Pattern Matching*, 2007, pp. 216–227.
[11] W.-K. Hon, R. Shah, and J. S. Vitter, "Compression, indexing, and retrieval for massive string data," in *Proc. Symp. Combinatorial Pattern Matching*, 2010, pp. 260–274.
[12] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proc. IEEE Symp. Foundations of Computer Science*, 2000, pp. 390–398.
[13] ——, "Indexing compressed text," *J. ACM*, vol. 52, no. 4, pp. 552–581, 2005.
[14] R. González and G. Navarro, "A compressed text index on secondary memory," *J. Combinatorial Mathematics and Combinatorial Computing*, vol. 71, pp. 127–154, 2009.
[15] R. A. Baeza-Yates, E. F. Barbosa, and N. Ziviani, "Hierarchies of indices for text searching," *Information Systems*, vol. 21, no. 6, pp. 497–514, 1996.
[16] L. Colussi and A. De Col, "A time and space efficient data structure for string searching on large texts," *Information Processing Letters*, vol. 58, no. 5, pp. 217–222, 1996.
[17] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, 2nd ed. San Francisco: Morgan Kaufmann, 1999.
[18] S. Gog and J. Fischer, "Advantages of shared data structures for sequences of balanced parentheses," in *Proc. IEEE Data Compression Conf.*, Snowbird, UT, 2010, pp. 406–415.
[19] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proc. ACM-SIAM Symp. Discrete Algorithms*, 2003, pp. 841–850.
[20] R. Raman, V. Raman, and S. S. Rao, "Succinct indexable dictionaries with applications to encoding k-ary trees and multisets," in *Proc. ACM-SIAM Symp. Discrete Algorithms*, 2002, pp. 233–242.
[21] V. Mäkinen and G. Navarro, "Succinct suffix arrays based on run-length encoding," in *Proc. Symp. Combinatorial Pattern Matching*, 2005, pp. 45–56.
[22] D. Okanohara and K. Sadakane, "Practical entropy-compressed rank/select dictionary," in *Proc. Wkshp. Algorithm Engineering and Experiments*, 2007.
[23] T. Bingmann, J. Fischer, and V. Osipov, "Inducing suffix and lcp arrays in external memory," in *Proc. Wkshp. Algorithm Engineering and Experiments*, 2013.
[24] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan, "Structuring labeled trees for optimal succinctness, and beyond," in *Proc. IEEE Symp. Foundations of Computer Science*, 2005, pp. 184–196.
[25] B. Phoophakdee and M. J. Zaki, "Genome-scale disk-based suffix tree indexing," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2007, pp. 833–844.
[26] S.-S. Wong, W.-K. Sung, and L. Wong, "CPS-tree: A compact partitioned suffix tree for disk-based indexing on large genome sequences," in *Proc. Int. Conf. Data Engineering*, 2007, pp. 1350–1354.
[27] A. Orlandi and R. Venturini, "Space-efficient substring occurrence estimation," in *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, 2011, pp. 95–106.
[28] M. P. Ferguson, "FEMTO: Fast search of large sequence collections," in *Proc. Symp. Combinatorial Pattern Matching*, 2012, pp. 208–219.
[29] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Computing Surveys*, vol. 38, no. 2, Jul. 2006.
[30] N. Tang, L. Sidirourgos, and P. A. Boncz, "Space-economical partial gram indices for exact substring matching," in *Proc. Conf. Information and Knowledge Management*, 2009, pp. 285–294.
[31] S. J. Puglisi, W. F. Smyth, and A. Turpin, "Inverted files versus suffix arrays for locating patterns in primary memory," in *Proc. Symp. String Processing and Information Retrieval*, 2006, pp. 122–133.

Simon Gog completed a PhD in Computer Science at Ulm University in 2011 in the area of practical compressed index data structures. He is now a Research Fellow at The University of Melbourne, with interests in engineering space-efficient data structures and algorithms.

Alistair Moffat completed a PhD at the University of Canterbury in 1986. Since then he has been a faculty member at The University of Melbourne, with interests in text and index compression, and algorithms for string search and information retrieval.

J. Shane Culpepper completed a PhD at The University of Melbourne in 2008. Since then he has been a faculty member at RMIT University, with research interests in space-efficient data structures, data compression and information retrieval.

Andrew Turpin completed a PhD at the University of Melbourne in 1999. Since then he has been a faculty member at Curtin University, RMIT University, and The University of Melbourne. In addition to his interests in compression and text search, he also actively researches computational problems in human vision testing.

Anthony Wirth completed a PhD at Princeton University in 2005. Since then, he has been a faculty member at The University of Melbourne, with interests in graph, string and approximation algorithms, clustering, algorithm engineering, adaptive sampling, and biological sequence analysis.