Document Listing on Repetitive Collections with Guaranteed Performance∗

Gonzalo Navarro

Center for Biotechnology and Bioengineering, Dept. of Computer Science, University of Chile, Beauchef 851, Santiago, Chile. gnavarro@dcc.uchile.cl

Abstract

We consider document listing on string collections, that is, finding in which strings a given pattern appears. In particular, we focus on repetitive collections: a collection of size $N$ over alphabet $[1, \sigma]$ is composed of $D$ copies of a string of size $n$, and $s$ single-character or block edits are applied on ranges of copies. We introduce the first document listing index with size $\tilde{O}(n + s)$, precisely $O((n \log \sigma + s \log^2 N) \log D)$ bits, and with useful worst-case time guarantees: Given a pattern of length $m$, the index reports the $ndoc$ strings where it appears in time $O(m^2 + m \log^{1+\epsilon} N \cdot ndoc)$, for any constant $\epsilon > 0$. Our technique is to augment a range data structure that is commonly used on grammar-based indexes, so that instead of retrieving all the pattern occurrences, it computes useful summaries on them. We show that the idea has independent interest: we introduce the first grammar-based index that, on a text $T[1, N]$ with a grammar of size $r$, uses $O(r \log N)$ bits and counts the number of occurrences of a pattern $P[1, m]$ in time $O(m^2 + m \log^{2+\epsilon} r)$, for any constant $\epsilon > 0$.

1 Introduction

Document retrieval on general string collections is an area that has recently attracted attention [40]. On the one hand, it is a natural generalization of the basic Information Retrieval tasks carried out on search engines [3, 9], many of which are also useful on Far East languages, collections of genomes, code repositories, multimedia streams, etc. It also enables phrase queries on natural language texts. On the other hand, it raises a number of algorithmic challenges that are not easily addressed with classical pattern matching approaches.

In this paper we focus on one of the simplest document retrieval problems, document listing [36]. Let $D$ be a collection of $D$ documents of total length $N$. We want to build an index on $D$ such that, later, given a search pattern $P$ of length $m$, we report the identifiers of all the $ndoc$ documents where $P$ appears. Given that $P$ may occur $occ \gg ndoc$ times in $D$, resorting to pattern matching, that is, finding all the $occ$ occurrences and then listing the distinct documents where they appear, can be utterly inefficient. Optimal $O(m + ndoc)$ time document listing solutions appeared only in 2002 [36], although they use too much space. There are also more recent statistically compressed indices [40] [25], which are essentially space-optimal with respect to the statistical entropy and pose only a small time penalty.

∗ Supported in part by Fondecyt grant 1-170048 and Basal Funds FB0001, Conicyt, Chile. A preliminary version of this article appeared in Proc. CPM 2017.
We are, however, interested in highly repetitive string collections [39], which are formed by a few distinct documents and a number of near-copies of those. Such collections arise, for example, when sequencing the genomes of thousands of individuals of a few species, when managing versioned collections of documents like Wikipedia, and in versioned software repositories. Although many of the fastest-growing datasets are indeed repetitive, this is an underdeveloped area: most succinct indices for string collections are based on statistical compression, and these fail to exploit repetitiveness [30].

1.1 Modeling repetitiveness

There are few document listing indices that profit from repetitiveness. A simple model to analyze them is as follows [33, 20, 39]: Assume there is a single document of size \( n \) on alphabet \([1, \sigma]\), and \( D - 1 \) copies of it, on which \( s \) single-character edits (insertions, deletions, substitutions) are distributed arbitrarily, forming a collection of size \( N \approx nD \). This models, for example, collections of genomes and their single-point mutations. For versioned documents and software repositories, a better model is a generalization where each edit affects a range of copies, such as an interval of versions if the collection has a linear versioning structure, or a subtree of versions if the versioning structure is hierarchical. We also permit more general block edit operations, where a whole block of symbols can be deleted, moved, inserted, or replaced (in the last two cases, with a new content or with a content copied from elsewhere in the document). In this case, \( n \) accounts also for the amount of new content created.

The gold standard to measure space usage on repetitive collections is the size of the Lempel-Ziv parsing [31]. If we parse the concatenation of the strings in a repetitive collection under any of the models above, we obtain at most \( z = n/\lg \sigma n + O(s) \ll N \) phrases. Therefore, while a statistical compressor would require basically \( N \lg \sigma \text{ bits if the base document is incompressible } [30] \), we can aim to reach as little as \( O(n \lg \sigma + s \lg N) \) bits by exploiting repetitiveness via Lempel-Ziv compression (that is, we assume an arbitrary Lempel-Ziv pointer requires \( O(\lg N) \) bits, but those in the first document could use \( O(\lg n) \)).

This might be too optimistic for an index, however, as there is no known way to extract substrings efficiently from Lempel-Ziv compressed text. Instead, grammar compression allows extracting any text symbol in logarithmic time using \( O(r \lg N) \) bits, where \( r \) is the size of the grammar [8, 51]. It is possible to obtain a grammar of size \( r = O(z \lg(N/z)) \) [10, 27], which using standard methods [45] can be tweaked to \( r = n/\lg \sigma N + s \lg N \) under our repetitiveness model. Thus the space we might aim at for indexing is \( O(n \lg \sigma + s \lg^2 N) \) bits.

1.2 Our contributions

Although they perform reasonably well in practice, none of the existing structures for document listing on repetitive collections [13, 20] offer good worst-case time guarantees combined with worst-case space guarantees that are appropriate for repetitive collections, that is, growing with \( n + s \) rather than with \( N \). In this paper we present the first document listing index offering good guarantees in space and time for repetitive collections: our index

1. uses \( O((n \lg \sigma + s \lg^2 N) \lg D) \) bits of space, and
2. performs document listing in time \( O(m^2 + m \lg^{1+\epsilon} N \cdot ndoc) \), for any constant \( \epsilon > 0 \).

That is, at the price of being an \( O(\lg D) \) space factor away from what could be hoped from a grammar-based index, our index offers document listing with useful time bounds per listed document. The result is summarized in Theorem 2 for single-character edits and Theorem 3 for block edits.
We actually build on a grammar-based document listing index \cite{13} that stores lists of the documents where each nonterminal appears, and augment it by rearranging the nonterminals in different orders, following a wavelet tree \cite{24} deployment that guarantees that only $O(m \lg r)$ ranges of lists have to be merged at query time. We do not store the lists themselves in various orders, but just succinct range minimum query (RMQ) data structures \cite{17} that allow implementing document listing on ranges of lists \cite{46}. Even those RMQ structures are too large for our purposes, so they are further compressed exploiting the fact that their underlying data has long increasing runs, so the structures are reduced with techniques analogous to those developed for the ILCP data structure \cite{20}.

The space reduction brings new issues, however, because we cannot afford storing the underlying RMQ sequences. These problems are circumvented with a new, tailored, technique to extract the distinct elements in a range that might have independent interest (see Lemma 6 in Appendix A).

**Extensions**

The wavelet tree \cite{24} represents a two-dimensional grid with points. It is used in grammar-based indexes \cite{14,15} to enumerate all the occurrences of the pattern: a number of secondary occurrences are obtained from each point that qualifies for the query. At a high level, our idea above is to compute summaries of the qualifying points instead of enumerating them one by one. We show that this idea has independent interest by storing the number of secondary occurrences that can be obtained from each point. The result is an index of $O(r \lg N)$ bits, similar to the size of previous grammar-based indexes \cite{14,15}, and able to count the number of occurrences of the pattern in time $O(m^2 + m \lg^{2+\epsilon} r)$, for any constant $\epsilon > 0$, see Theorem 5. Current grammar-based indexes are unable to count the occurrences without generating them one by one, so for the first time a grammar-based index can offer efficient counting.

As a byproduct, we improve an existing result \cite{42} on computing summaries of two-dimensional points in ranges, when the points have associated values from a finite group. We show in Theorem 4 that, within linear space, the time to operate all the values of the points in a given range of an $r \times r$ grid can be reduced from $O((\lg^3 r)$ to $O((\lg^{2+\epsilon} r)$, for any constant $\epsilon > 0$.

**2 Related work**

The first optimal-time and linear-space solution to document listing is due to Muthukrishnan \cite{36}, who solves the problem in $O(m + ndoc)$ time using an index of $O(N \lg N)$ bits of space.

Later solutions \cite{16} improved the space to essentially the statistical entropy of $D$, at the price of multiplying the times by low-order polylogs of $N$ (e.g., $O(m + \lg N \cdot ndoc)$ time with $O(N)$ bits on top of the entropy \cite{16,7}). However, statistical entropy does not capture repetitiveness well \cite{30}, and thus these solutions are not satisfactory in repetitive collections.

There has been a good deal of work on pattern matching indices for repetitive string collections \cite[Sec 13.2]{43}: building on regularities of suffix-array-like structures \cite{33,37,38,5,21}, on grammar compression \cite{14,15}, on Lempel-Ziv compression and variants \cite{20,18,16}, and on combinations \cite{18,19,26,52,6}. However, there has been little work on document retrieval structures for repetitive string collections.

One precedent is Claude and Munro’s index based on grammar compression \cite{13}. It builds on a grammar-based pattern-matching index \cite{15} and adds an inverted index that explicitly indicates the documents where each nonterminal appears; this inverted index is
also grammar-compressed. To obtain the answer, an unbounded number of those lists of documents must be merged. No relevant worst-case time or space guarantees are offered.

Another precedent is ILCP [20], where it is shown that an array formed by interleaving the longest common prefix arrays of the documents in the order of the global suffix array, ILCP, has long increasing runs on repetitive collections. Then an index of size bounded by the runs in the suffix array [33] and in the ILCP array performs document listing in time \(O(\text{search}(m) + \text{lookup}(N) \cdot \text{ndoc})\), where \text{search} and \text{lookup} are the search and lookup time, respectively, of a run-length compressed suffix array [33] [21]. Yet, there are only average-case bounds for the size of the structure in terms of \(s\): If the base document is generated at random and the edits are spread at random, then the structure uses \(O(n \lg N + s \lg^2 N)\) bits.

The last previous work is PDL [20], which stores inverted lists at sampled nodes in the suffix tree of \(D\), and then grammar-compresses the set of inverted lists. For a sampling step \(b\), it requires \(O((N/b) \lg N)\) bits plus the (unbounded) space of the inverted lists. Searches that lead to the sampled nodes have their answers precomputed, whereas the others cover a suffix array range of size \(O(b)\) and are solved by brute force in time \(O(b \cdot \text{lookup}(N))\).

To be fair, those indexes perform well in many practical situations [11]. However, in this article we are interested in whether theoretical worst-case guarantees can be provided.

### 3 Basic Concepts

#### 3.1 Listing the different elements in a range

Let \(A[1, t]\) be an array of integers in \([1, D]\). Muthukrishnan [36] gives a structure that, given a range \([i, j]\), lists all the \(\text{ndoc}\) distinct elements in \(A[i, j]\) in time \(O(\text{ndoc})\). He defines an array \(C[1, t]\) storing in \(C[k]\) the largest position \(l < k\) where \(A[l] = A[k]\), or \(C[k] = 0\) if no such position exists. Note that the leftmost positions of the distinct elements in \(A[i, j]\) are exactly those \(k\) where \(C[k] < i\). He then stores a data structure supporting range-minimum queries (RMQs) on \(C\): \(\text{RMQ}(i, j) = \text{argmin}_{i \leq k \leq j} C[k]\) [17]. Given a range \([i, j]\), he computes \(k = \text{RMQ}(i, j)\). If \(C[k] < i\), then he reports \(A[k]\) and continues recursively on \(A[i, k - 1]\) and \(A[k + 1, j]\). Whenever it turns out that \(C[k] \geq i\) for an interval \([x, y]\), there are no leftmost occurrences of \(A[x, j]\) within \(A[x, y]\), so this interval can be abandoned. It is easy to see that the algorithm takes \(O(\text{ndoc})\) time and uses \(O(t \lg t)\) bits of space; the RMQ structure uses just \(2t + o(t)\) bits and answers queries in constant time [14].

Furthermore, the RMQ structure does not even access \(C\). Sadakane [10] replaces \(C\) by a bitvector \(V[1, D]\) to mark which elements have been reported. He sets \(V\) initially to all zeros and replaces the test \(C[k] < i\) by \(V[A[k]] = 0\), that is, the value \(A[k]\) has not yet been reported (these tests are equivalent only if we recurse left and then right in the interval [10]). If so, he reports \(A[k]\) and sets \(V[A[k]] \leftarrow 1\). Overall, he needs only \(O(t + D)\) bits of space on top of \(A\), and still runs in \(O(\text{ndoc})\) time (\(V\) can be reset to zeros by rerunning the query or through lazy initialization). Hon et al. [25] further reduce the extra space to \(o(t)\) bits, yet increasing the time, via sampling the array \(C\).

In this paper we introduce a variant of Sadakane’s document listing technique that might have independent interest, see Section 4.2 and Lemma 6 in Appendix A.

#### 3.2 Wavelet trees

A wavelet tree [21] is a sequence representation that supports, in particular, two-dimensional orthogonal range queries [11] [31]. Let \((1, y_1), (2, y_2), \ldots, (r, y_r)\) be a sequence of points with \(y_i \in [1, r]\), and let \(S = y_1 y_2 \ldots y_r\) be the \(y\) coordinates in order. The wavelet tree is a
perfectly balanced binary tree where each node handles a range of \( y \) values. The root handles
\([1, r]\). If a node handles \([a, b]\) then its left child handles \([a, \mu]\) and its right child handles
\([\mu + 1, b]\), with \( \mu = \lfloor (a + b) / 2 \rfloor \). The leaves handle individual \( y \) values. If a node handles
range \([a, b]\), then it represents the subsequence \( S_{a,b} \) of \( y \) coordinates that belong to \([a, b]\).
Thus at each level the strings \( S_{a,b} \) form a permutation of \( S \). What is stored for each such
node is a bitvector \( B_{a,b} \) so that \( B_{a,b}[i] = 0 \) iff \( S_{a,b} \leq \mu \), that is, if that value is handled in
the left child of the node. Those bitvectors are provided with support for rank and select
queries: \( \text{rank}_v(B, i) \) is the number of occurrences of bit \( v \) in \( B[1, i] \), whereas \( \text{select}_v(B, j) \) is
the position of the \( j \)th occurrence of bit \( v \) in \( B \). The wavelet tree has height \( \lg r \), and its
total space requirement for all the bitvectors \( B_{a,b} \) is \( r \lg r \) bits. The extra structures for rank
and select add \( o(r \lg r) \) further bits and support the queries in constant time [12, 35]. With
the wavelet tree one can recover any \( y \) value by tracking it down from the root to a leaf, but
let us describe a more general procedure.

**Range queries**

Let \([x_1, x_2] \times [y_1, y_2]\) be a query range. The number of points that fall in the range can
be counted in \( O(\lg r) \) time as follows. We start at the root with the range \( S[x_1, x_2] =
S_{1,r}[x_1, x_2] \). Then we project the range both left and right, towards \( S_{1,\mu}[\text{rank}_0(B_{1,r}, x_1 -
1) + 1, \text{rank}_0(B_{1,r}, x_2)] \) and \( S_{\mu+1,1}[\text{rank}_1(B_{1,r}, x_1 - 1) + 1, \text{rank}_1(B_{1,r}, x_2)] \), respectively, with
\( \mu = \lfloor (r + 1) / 2 \rfloor \). If some of the ranges is empty, we stop the recursion on that node. If the
interval \([a, b]\) handled by a node is disjoint with \([y_1, y_2]\), we also stop. If the interval \([a, b]\)
is contained in \([y_1, y_2]\), then all the points in the \( x \) range qualify, and we simply sum the
length of the range to the count. Otherwise, we keep splitting the ranges recursively. It
is well known that the range \([y_1, y_2]\) is covered by \( O(\lg r) \) wavelet tree nodes, and that we
trace \( O(\lg r) \) nodes to reach them (see Gagie et al. [22] for a review of this and more
refined properties). If we also want to report all the corresponding \( y \) values, then instead of
counting the points found, we track each one individually towards its leaf, in \( O(\lg r) \) time.
At the leaves, the \( y \) values are sorted. In particular, if they are a permutation of \([1, r]\), we
know that the \( i \)th left-to-right leaf is the value \( y = i \). Thus, extracting the \text{nocc} results takes
time \( O((1 + \text{nocc}) \lg r) \).

**Faster reporting**

By using \( O(r \lg r) \) bits, it is possible to track the positions faster in upward direction, and
associate the values to their root positions. By using \( O((1/\epsilon) r \lg r) \) bits, one can reach the
root position of a symbol in time \( O((1/\epsilon) \lg^r r) \), for any \( \epsilon > 0 \) [11, 11]. Therefore, the \text{nocc}
results can be extracted in time \( O(\lg r + \text{nocc} \lg^r r) \) for any constant \( \epsilon \).

**Summary queries**

Navarro et al. [42] showed how to perform \textit{summary} queries on wavelet trees, that is, instead
of listing all the points that belong to a query range, compute some summary on them
faster than listing the points one by one. For example, if the points are assigned values in
\([1, N]\), then one can use \( O(r \lg N) \) bits and compute the sum, average, or variance of the
values associated with points in a range in time \( O(\lg^3 r) \), or their minimum/maximum in
\( O(\lg^2 r) \) time. The idea is to associate further data to the sequences \( S_{a,b} \) and carry out range
queries on the \( O(\lg r) \) ranges into which two-dimensional queries are decomposed, in order
to compute the desired summarizations. To save space, one may sample the levels for which
the extra data is stored.
In this paper we show that the $O(\lg^3 r)$ time can be improved to $O(\lg^{2+\epsilon} r)$, for any $\epsilon > 0$, within the same asymptotic space. See Theorem 4 in Section 5.

3.3 Range minimum queries on arrays with runs

Let $A[1, t]$ be an array that can be cut into $\rho$ runs of nondecreasing values. Then it is possible to solve RMQs in $O(\lg \lg t)$ time plus $O(1)$ accesses to $A$ using $O(\rho \lg(t/\rho))$ bits. The idea is that the possible minima (breaking ties in favor of the leftmost) in $A[i, j]$ are either $A[i]$ or the positions where runs start in the range. Then, we can use a sparse bitvector $M[1, t]$ marking with $M[k] = 1$ the run heads. We also define an array $A'[1, \rho]$, so that if $M[k] = 1$ then $A'[\text{rank}_1(M, k)] = A[k]$. We do not store $A'$, but just an RMQ structure on it. Hence, the minimum of the run heads in $A[i, j]$ can be found by computing the range of run heads involved, $i' = \text{rank}_1(M, i - 1) + 1$ and $j' = \text{rank}_1(M, j)$, then finding the smallest value among them in $A'$ with $k' = \text{RMQ}_{A'}(i', j')$, and mapping it back to $A$ with $k = \text{select}_1(M, k')$. Finally, the RMQ answer is either $A[i]$ or $A[k]$, so we access $A$ twice to compare them.

This idea was used by Gagie et al. [20, Sec 3.2] for runs of equal values, but it works verbatim for runs of nondecreasing values. They show how to store $M$ in $\rho \lg(t/\rho) + O(\rho)$ bits so that it solves rank in $O(\lg \lg t)$ time and select in $O(1)$ time, by augmenting a sparse bitvector representation [44]. This dominates the space and time of the whole structure.

The idea was used even before by Barbay et al. [1, Thm. 2], for runs of nondecreasing values. They represented $M$ using $\rho \lg(t/\rho) + O(\rho) + o(t)$ bits so that the $O(\lg \lg t)$ time becomes $O(1)$, but we cannot afford the $o(t)$ extra bits in this paper.

3.4 Grammar compression

Let $T[1, N]$ be a sequence of symbols over alphabet $[1, \sigma]$. Grammar compressing $T$ means finding a context-free grammar that generates $T$ and only $T$. The grammar can then be used as a substitute for $T$, which provides good compression when $T$ is repetitive. We are interested, for simplicity, in grammars in Chomsky normal form, where the rules are of the form $A \rightarrow BC$ or $A \rightarrow a$, where $A$, $B$, and $C$ are nonterminals and $a \in [1, \sigma]$ is a terminal symbol. For every grammar, there is a proportionally sized grammar in this form.

A Lempel-Ziv parse [31] of $T$ cuts $T$ into $z$ phrases, so that each phrase $T[i, j]$ appears earlier in $T[i', j']$, with $i' < i$. It is known that the smallest grammar generating $T$ must have at least $z$ rules [45, 10], and that it is possible to convert a Lempel-Ziv parse into a grammar with $r = O(z \lg(N/z))$ rules [45, 10, 47, 28, 29]. Furthermore, such grammars can be balanced, that is, the parse tree is of height $O(\lg N)$. By storing the length of the string to which every nonterminal expands, it is easy to access any substring $T[i, j]$ from its compressed representation in time $O(j - i + \lg N)$ by tracking down the range in the parse tree. This can be done even on an unbalanced grammar [8]. The total space used by this representation, with a grammar of $r$ rules, is $O(r \lg N)$ bits.

3.5 Grammar-based indexing

The pattern-matching index of Claude and Navarro [14] builds on a grammar in Chomsky normal form that generates a text $T[1, N]$, with $r+1$ rules. Let $s(A)$ be the string generated by nonterminal $A$. Then they collect the strings $s(A)$ for all those nonterminals, except the initial symbol $S$. Let $C_1, \ldots, C_r$ be the nonterminals sorted lexicographically by $s(A)$ and let $B_1, \ldots, B_r$ be the nonterminals sorted lexicographically by the reverse strings, $s(A)^{rev}$. They create a set of points in $[1, r] \times [1, r]$ so that $(i, j)$ is a point (corresponding to nonterminal $A$) if the rule that defines $A$ is $A \rightarrow B_iC_j$. Those points are stored in a wavelet tree.
To search for a pattern $P[1,m]$, they first find the primary occurrences, that is, those that appear when $B$ is concatenated with $C$ in a rule $A \rightarrow BC$. The secondary occurrences, which appear when $A$ is used elsewhere, are found in a way that does not matter for this paper. To find the primary occurrences, they cut $P$ into two nonempty parts $P = P_1P_2$, in the $m - 1$ possible ways. For each cut, they binary search for $P_i^{rev}$ in the sorted set $s(B_i)^{rev}, \ldots, s(B_y)^{rev}$ and for $P_2$ in the sorted set $s(C_1), \ldots, s(C_y)$. Let $[x_1, x_2]$ be the interval obtained for $P_1$ and $[y_1, y_2]$ the one obtained for $P_2$. Then all the points in $[x_1, x_2] \times [y_1, y_2]$, for all the $m - 1$ partitions of $P$, are the primary occurrences.

To search for $P_i^{rev}$ or for $P_2$, the grammar is used to extract the required substrings of $T$ in time $O(m + \lg N)$, so the overall search time to find the nocc primary occurrences is $O(m \lg r(m + \lg N) + \lg r \cdot \text{nocc})$. Let us describe the fastest possible variant that uses $O(r \lg N)$ bits, disregarding constant factors in the space. Within $O(r \lg N)$ bits, one can store Patricia trees [34] on the strings $s(B_i^{rev})$ and $s(C_i)$, to speed up binary searches and reduce the time to $O(m(m + \lg N) + \lg r \cdot \text{nocc})$. Also, one can use the structure of Gasienc et al. [23] that, within $O(r \lg N)$ further bits, allows extracting any prefix/suffix of any nonterminal in constant time per symbol (see Claude and Navarro [15] for more details). Since in our search we only access prefixes/suffixes of whole nonterminals, this further reduces the time to $O(m^2 + \lg r \cdot \text{nocc})$. Finally, we can use the technique described at the end of Section 3.2 to obtain time $O(m^2 + \lg^\epsilon r \cdot \text{nocc})$, for any constant $\epsilon > 0$.

### Counting

This index locates the occurrences of $P$ one by one, but cannot count them without locating them all. This is a feature easily supported by suffix-array-based compressed indexes [33, 21] in $O(m \lg N)$ time or less, but so far unavailable in grammar-based or Lempel-Ziv-based compressed indexes. In Theorem 5 of Section 6, we offer for the first time efficient counting for grammar-based indexes. Within their same asymptotic space, we can count in time $O(m^2 + m \lg^{2+\epsilon} r)$ for any constant $\epsilon > 0$.

### Document listing

The original structure was also unable to perform document listing without locating all the occurrences and determining the document where each belongs. Claude and Munro [15] showed how to extend it in order to support document listing on a collection $D$ of $D$ string documents, which are concatenated into a text $T[1,N]$. To each nonterminal $A$ they associate the increasing list $\ell(A)$ of the identifiers of the documents (integers in $[1,D]$) where $A$ appears. To perform document listing, they find all the primary occurrences $A \rightarrow BC$ of all the partitions of $P$, and merge their lists. There is no useful worst-case time bound for this operation other than $O(\text{nocc} \cdot \text{nocc})$, where $\text{nocc}$ can be much larger than $\text{ndoc}$. To reduce space, they also grammar-compress the sequence of all the $r$ lists $\ell(A)$. They also give no worst-case space bound for the compressed lists (other than $O(rD \lg D)$ bits).

At the end of Section 5.1, we show that, under our repetitiveness model, this index can be made to occupy $O(n \lg \sigma + s \lg^2 N)$ bits, which is what can be expected from a grammar-based index according to our discussion. Still, it gives no worst-case guarantees for the document listing time. In Theorems 2 and 8, we show that, by multiplying the space by an $O(\lg D)$ factor, document listing is possible in time $O(m^2 + m \lg^{1+\epsilon} N \cdot \text{nocc})$ for any constant $\epsilon > 0$. 


4 Our Document Listing Index

We build on the basic structure of Claude and Munro [13]. Our main idea is to take advantage of the fact that the \textit{noc} primary occurrences to detect in Section 3.5 are found as points in the two-dimensional structure, along \(O(\log r)\) ranges within wavelet tree nodes (recall Section 3.2) for each partition of \(P\). Instead of retrieving the \(\text{noc}\) individual lists, decompressing and merging them \[13\], we will use the techniques to extract the distinct elements of a range seen in Section 3.1. This will drastically reduce the amount of merging necessary, and will provide useful upper bounds on the document listing time.

4.1 Structure

We store the grammar of \(T\) in a way that it allows direct access for pattern searches, as well as the wavelet tree for the points \((B_i, C_j)\), the Patricia trees, and extraction of prefixes/suffixes of nonterminals, all in \(O(r \log N)\) bits.

Consider any sequence \(S_{a,b}[1, q]\) at a wavelet tree node handling the range \([a, b]\) (recall that those sequences are not explicitly stored). Each element \(S_{a,b}[k] = A_k\) corresponds to a point \((i, j)\) associated with a nonterminal \(A_k \rightarrow B_i C_j\). Then let \(L_{a,b} = \ell(A_1) \cdot \ell(A_2) \cdots \ell(A_q)\) be the concatenation of the inverted lists associated with the nonterminals in \(S_{a,b}\), and let \(M_{a,b} = 10^{\ell(A_1) - 1} 10^{\ell(A_2) - 1} \cdots 10^{\ell(A_q) - 1}\) mark where each list begins in \(L_{a,b}\). Now let \(C_{a,b}\) be the \(C\)-array corresponding to \(L_{a,b}\), as described in Section 3.1. As in that section, we do not store \(L_{a,b}\) nor \(C_{a,b}\), but just the RMQ structure on \(C_{a,b}\), which together with \(M_{a,b}\) will be used to retrieve the unique documents in a range \(S_{a,b}[i, j]\).

Since \(M_{a,b}\) has only \(r\) 1s out of (at most) \(rD\) bits across all the wavelet tree nodes of the same level, it can be stored with \(O(r \log D)\) bits per level \[13\], and \(O(r \log r \log D)\) bits overall. On the other hand, as we will show, \(C_{a,b}\) is formed by a few increasing runs, say \(\rho\) across the wavelet tree nodes of the same level, and therefore we represent its RMQ structure using the technique of Section 3.3. The total space used by those RMQ structures is then \(O(\rho r \log (rD/\rho))\) bits.

Finally, we store the explicit lists \(\ell(B_i)\) aligned to the wavelet tree leaves, so that the list of any element in any sequence \(S_{a,b}\) is reached in \(O(\log r)\) time by tracking down the element. Those lists, of maximum total length \(rD\), are grammar-compressed as well, just as in the basic scheme \[13\]. If the grammar has \(r'\) rules, then the total compressed size is \(O(r' \log (rD))\) bits to allow for direct access in \(O(\log (rD))\) time, see Section 3.4.

In total, our structure uses \(O(r \log N + r \lg r \log D + \rho \log r \log (rD/\rho) + r' \log (rD))\) bits.

4.2 Document listing

A document listing query proceeds as follows. We cut \(P\) in the \(m - 1\) possible ways, and for each way identify the \(O(\log r)\) wavelet tree nodes (and ranges) where the desired points lie. Overall, we have \(O(m \log r)\) ranges and need to take the union of the inverted lists of all the points inside those ranges. We extract the distinct documents in each range and then compute their union. If a range has only one element, then we can track it to the leaves, where its list \(\ell(\cdot)\) is stored, and recover it by decompressing the whole list.

Otherwise, we use in principle the document listing technique of Section 3.1. Let \(S_{a,b}[i, j]\) be a range from where to obtain the distinct documents. We compute \(i' = \text{select}_1(M_{a,b}, i)\) and \(j' = \text{select}_1(M_{a,b}, j + 1) - 1\), and obtain the distinct elements in \(L_{a,b}[i', j']\), by using RMQs on \(C_{a,b}[i', j']\). Recall that, as in Section 3.3, we use a run-length compressed RMQ structure on \(C_{a,b}\). With this arrangement, every RMQ operation takes time \(O(\log \log (rD))\).
plus the time to accesses two cells in $C_{a,b}$. Those accesses are made to compare a run head with the leftmost element of the query interval, $C_{a,b}[i']$. The problem is that we have not represented the cells of $C_{a,b}$ and cannot easily compute them on the fly.

Barbay et al. [3, Thm. 3] give a representation that determines the position of the minimum in $C_{a,b}[i',j']$ without the need to perform the two accesses on $C_{a,b}$. They need $\rho \log(rD) + \rho \log(rD/\rho) + O(\rho) + o(rD)$ bits, which unfortunately is too high for us.

Instead, we modify the way the distinct elements are obtained, so that comparing the two cells of $C_{a,b}$ is unnecessary. In the same spirit of Sadakane’s solution (see Section 3.1) we use a bitvector $V[1,D]$ where we mark the documents already reported. Given a range $S_{a,b}[i,j] = A_1 \ldots A_j$, we first track $A_i$ down the wavelet tree, recover and decompress its list $\ell(A_i)$, and mark all of its documents in $V$. Note that all the documents in the list $\ell(\cdot)$ are different. Now we do the same with $A_{i+1}$, decompressing $\ell(A_{i+1})$ left to right and marking the documents in $V$, and so on, until we decompress a document $\ell(A_{i+d})[k]$ that is already marked in $V$. Only now we use the RMQ technique of Section 3.3 on the interval $C_{a,b}[i',j']$, where $i' = \text{select}_1(M_{a,b},i + d) − 1 + k$ and $j' = \text{select}_1(M_{a,b},j + 1) − 1$, to obtain the next document to report. This technique, as explained, yields two candidates: one is $L_{a,b}[i'] = \ell(A_{i+d})[k]$ itself, and the other is some run head $L_{a,b}[k']$ whose identity we can obtain from the wavelet tree leaf. But we know that $L_{a,b}[i']$ was already reported, so we act as if the RMQ was always $L_{a,b}[k']$: If the RMQ answer was $L_{a,b}[i']$ then, since it is already reported, we should stop. But in this case, $L_{a,b}[k']$ is also already reported and we do stop anyway. Hence, if $L_{a,b}[k']$ is already reported we stop, and otherwise we report it and continue recursively on the intervals $C_{a,b}[i', k' − 1]$ and $C_{a,b}[k' + 1,j']$. On the first, we can continue directly, as we still know that $L_{a,b}[i']$ is already reported. On the second interval, instead, we must restore the invariant that the leftmost element was already reported. So we find out with $M$ the list and position $\ell(A_i)[u]$ corresponding to $C_{a,b}[k' + 1]$ (i.e., $t = \text{rank}_1(M_{a,b},k' + 1)$ and $u = k' + 1 - \text{select}_1(M,t) + 1$), track $A_i$ down to its leaf in the wavelet tree, and traverse $\ell(A_i)$ from position $u$ onwards, reporting documents until finding one that has been reported.

The correctness of this document listing algorithm is proved in Appendix A

The $m − 1$ searches for partitions of $P$ take time $O(m^2)$. In the worst case, extracting each distinct document in the range requires an RMQ computation without access to $C_{a,b}$ ($O(\log \log (rD))$ time), tracking an element down the wavelet tree ($O(\log r)$ time), and extracting an element from its grammar-compressed list $\ell(\cdot)$ ($O(\log (rD))$ time). This adds up to $O(\log (rD))$ time per document extracted in a range. In the worst case, however, the same documents are extracted over and over in all the $O(m \log r)$ ranges, and therefore the final search time is $O(m^2 + m \log r \log (rD) \cdot ndoc)$.

### 5 Analysis in a Repetitive Scenario

Our structure uses $O(r \log N + r \log r \log D + \rho \log r \log (rD/\rho) + r' \log (rD))$ bits, and performs document listing in time $O(m^2 + m \log r \log (rD) \cdot ndoc)$. We now specialize those formulas under our repetitiveness model. Note that our index works on any string collection; we use the simplified model of the $D − 1$ copies of a single document of length $n$, plus the $s$ edits, to obtain analytical results that are easy to interpret in terms of repetitiveness.

We also assume a particular strategy to generate the grammars in order to show that it is possible to obtain the complexities we give. This involves determining the minimum
number of edits that distinguishes each document from the previous one. We first consider the model where each of the \( s \) edits is a single-character insertion, deletion, or substitution. If the \( s \) edit positions are not given explicitly, the optimal set of \( s \) edits can still be obtained at construction time, with cost \( O(Ns) \), using dynamic programming \[50\). Later, we consider a generalized case where each edit involves a block of characters.

### 5.1 Space

Consider the model where we have \( s \) single-character edits affecting a range of document identifiers. This includes the model where each edit affects a single document, as a special case. The model where the documents form a tree of versions, and each edit affects a whole subtree, also boils down to the model of ranges by numbering the documents according to their preorder position in the tree of versions.

An edit that affects a range of documents \( d_i, \ldots, d_j \) will be regarded as two edits: one that applies the change at \( d_i \) and one that undoes it at \( d_j \) (if needed, since the edit may be overridden by another later edit). Thus, we will assume that there are at most \( 2s \) edits, each of which affects all the documents starting from the one where it applies. We will then assume \( s \geq (D - 1)/2 \), since otherwise there will be identical documents, and this is easily reduced to a smaller collection with multiple identifiers per document.

#### Our grammar

The documents are concatenated into a single text \( T[1, N] \), where \( N \leq D(n + s) \). Let us make our grammar for \( T \) contain the \( O(N^{1/3}) \) nonterminals that generate all the strings of length up to \( \frac{1}{2} \lg_s N \), which we will call “metasymbols”. Then the grammar replaces the first document with \( \Theta(n/\lg_s N) \) such nonterminals, and builds a balanced binary parse tree of height \( h = \Theta(\lg n) \) on top of them. All the internal nodes of this tree are distinct nonterminal symbols (even if they generate the same text), and end up in a root symbol \( S_1 \).

Now we regard the subsequent documents one by one. For each new document \( d \), we start by copying the parse tree from the previous one, \( d - 1 \), including the start symbol \( S_d = S_{d-1} \). Then, we apply the edits that start at that document. Let \( h \) be the height of its parse tree. A character substitution requires replacing the metasymbol covering the position where the edit applies, and then renaming the nonterminals \( A_1, \ldots, A_n = S_d \) in the path from the parent of the metasymbol to the root. Each \( A_i \) in the path is replaced by a new nonterminal \( A_i' \). The nonterminals that do not belong to the path are not affected. A deletion proceeds similarly: we replace the metasymbol of length \( k \) by one of length \( k - 1 \) (we leave the metasymbol of length 0, the empty string, unchanged if it appears as a result of deletions). Finally, an insertion into a metasymbol of length \( k \) replaces it by one of length \( k + 1 \), unless \( k \) was already the maximum metasymbol length, \( \frac{1}{2} \lg_s N \). In this case we replace the metasymbol leaf by an internal node with two leaves, which are metasymbols of length around \( \frac{1}{2} \lg_s N \). To maintain a balanced tree, we use the AVL insertion mechanism, which may modify \( O(h) \) nodes toward the root. This ensures that, even in documents receiving \( s \) insertions, the height of the parse tree will be \( O(\lg(n + s)) \).

Since each edit creates \( O(\lg(n + s)) \) new nonterminals, the final grammar size is \( r = \Theta(N^{1/3} + n/\lg_s N + s \lg(n + s)) = \Theta(n/\lg_s N + s \lg N) \), where we used that either \( n \) or \( s \) is \( \Omega(\sqrt{N}) \) because \( N \leq D(n + s) \leq (2s + 1)(n + s) \). Once all the edits are applied, we add a balanced tree on top of those \( O(r) \) symbols, which asymptotically does not change \( r \) (we may also avoid this final tree and access the documents individually, since our accesses never cross document borders).
Inverted lists

Our model makes it particularly easy to bound \( r' \). Each nonterminal occurs at most once in a document, since we either create it when parsing the initial copy or when applying an edit. Thus it appears for the first time in a document \( d \) (which is 1 if it is in the initial copy). Later, when an edit replaces it by another nonterminal, it does never appear again: note that each new document starts by copying those appearing the previous copy, destroying some of them, and creating new ones. Even when an edit is “undone”, we do not restore the previous nonterminal, but rather create a new one that might happen to expand to the same string of the original one. Therefore, each nonterminal spans a range of documents \([d, d']\), which might be stored without resorting to grammars in \( O(r \lg D) \) bits. Any element of the list is obviously accessed in \( O(1) \) time, faster than on the general scheme we described.

Run-length compressed arrays \( C \)

Finally, let us now bound \( \rho \). When we have only the initial copy, all the existing nonterminals mention document 1, and thus \( C \) has a single nondecreasing run. Now consider the moment where we process document \( d \). We will insert the value \( d \) at the end of the lists of all the nonterminals \( A \) that appear in document \( d \). As long as document \( d \) uses the same parse tree of document \( d - 1 \), no new runs are created in \( C \).

\[ \text{Lemma 1. If document } d \text{ uses the same nonterminals as document } d - 1, \text{ inserting it in the inverted lists does not create any new run in the } C \text{ arrays.} \]

\textbf{Proof.} The positions \( p_1, \ldots, p_k \) where we insert the document \( d \) in the lists of the nonterminals that appear in it, will be chained in a list where \( C[p_{i+1}] = p_i \) and \( C[p_1] = 0 \). Since all the nonterminals \( A \) also appear in document \( d - 1 \), the lists will contain the value \( d - 1 \) at positions \( p_k, \ldots, p_1 \), and we will have \( C[p_{k+1}] = p_k - 1 \) and \( C[p_1] = 0 \). Therefore, the new values we insert for \( d \) will not create new runs: \( C[p_1] = C[p_1 - 1] = 0 \) does not create a run, and neither can \( C[p_{i+1}] = C[p_{i+1} - 1] + 1 \), because if \( C[p_{i+1} + 1] < C[p_{i+1}] = p_i \), then we are only creating a new run if \( C[p_{i+1} + 1] = p_i - 1 \), but this cannot be since \( C[p_{i+1} - 1] = p_i - 1 = C[p_{i+1} + 1] \) and in this case \( C[p_{i+1} + 1] \) should have pointed to \( p_i + 1 - 1 \).

Now, each edit we apply on \( d \) makes \( O(\lg N) \) nonterminals appear or disappear, and thus \( O(\lg N) \) values of \( d \) appear or disappear in \( C \). Each such change may break a run. Therefore, \( C \) may have at most \( \rho = \Theta(s \lg N) \) runs per wavelet tree level.

Total

Therefore, the total size of the index can be expressed as follows. The \( O(r \lg r \lg D) \) bits coming from the sparse bitvectors \( M \), is \( O(r \lg N \lg D) \) (since \( \lg r = \Theta(\lg(\lg n)) = \Theta(\lg N) \), and thus it is \( O(n \lg s \lg D + s \lg^2 N \lg D) \). This subsumes the \( O(\lg N) \) bits of the grammar and the wavelet tree. The \( O(\rho \lg r \lg(rD/\rho)) \) bits of the structures \( C \) are monotonically increasing with \( \rho \), so since \( \rho = s \lg N = O(r) \), we can upper bound it by replacing \( \rho \) with \( r \), obtaining \( O(r \lg r \lg D) \) as in the space for \( M \). Finally, the inverted lists can be represented with just \( O(\lg D) \) bits. Overall, the structures add up to \( O((n \lg \sigma + s \lg^2 N) \lg D) \) bits.

Note that we can also analyze the space required by Claude and Munro’s structure \cite{ClaudeMunro}, which is \( O(r \lg N) = O(n \lg s + s \lg^2 N) \) bits. Although smaller than ours by an \( O(\lg D) \) factor, their search time has no useful bounds.
5.2 Time

Our search time is $O(m^2 + m \log r \log (rD) \cdot ndoc) = O(m^2 + m \log^2 N \cdot ndoc)$. The $O(\log (rD))$ cost corresponds to accessing a list $\ell(A)$ from the wavelet tree, and includes the $O(\log r)$ time to reach the leaf and the $O(\log D)$ time to access a position in the grammar-compressed list. Since we have replaced the grammar-compressed lists by a simple range, this cost is now simply $O(\log r)$. As seen in Section 3.2 it is possible to reduce this $O(\log r)$ tracking time to $O((1/\epsilon) \log^\epsilon n)$ for any $\epsilon > 0$, within $O((1/\epsilon) r \log N)$ bits. In this case, the lists $\ell(A)$ are associated with the symbols at the root of the wavelet tree, not the leaves.

\textbf{Theorem 2.} Let collection $D$, of total size $N$, be formed by an initial document of length $n$ plus $D - 1$ copies of it, with $s$ single-character edit operations performed on ranges or subtrees of documents. Then $D$ can be represented within $O((n \log \sigma + s \log^2 N) \log D)$ bits, so that the ndoc documents where a pattern of length $m$ appears can be listed in time $O(m^2 + m \log^{1+\epsilon} N \cdot ndoc)$, for any constant $\epsilon > 0$.

We can also obtain other tradeoffs. For example, with $\epsilon = 1/\log \log r$ we obtain $O((n \log \sigma + s \log^2 N) (\log D + \log \log N))$ bits of space and $O(m^2 + m \log N \log \log N \cdot ndoc)$ search time.

5.3 Block edits

Let us now consider that each edit involves removing a block of characters, inserting a block that exists somewhere else, or inserting a completely new block. Other operations, like replacing a block by another existing or new block, or moving a block to another position, are simulated with a constant number of operations from the set we are considering.

Let us first consider the process of removing a block $S[i,j]$ from a document $S$, for which we have an AVL parse tree $T$ of height $h$. Let $v$ be the lowest common ancestor of the leaves $i$ and $j$. Then, we must remove from $T$ every right child in the path from $v$ to the leaf $i$ and every left child in the path from $v$ to the leaf $j$ ($v$ not included, in either case). The remaining tree is obtained by merging up to $2h$ subtrees of $T$: (1) from each node $u$ in the path from the root of $T$ to the leaf $i$, if the leaf descends from the right child of $u$, then the left child of $u$ is the root of a new subtree to merge; (2) for each node $u$ in the path from the leaf $j$ to the root of $T$, if the leaf descends from the left child of $u$, then the right child of $u$ is the root of a new subtree to merge. All the selected subtrees are AVL trees, and they form two series whose consecutive height differences add up to $h$. Rytter [45] shows how to merge those sets of subtrees into a new AVL using $O(h)$ new nodes. We can therefore accommodate the deletion of a block using $O(\log N)$ new nonterminal symbols.

Consider now an insertion between the leaves $i$ and $j = i + 1$. We cut $T$ into $2h$ subtrees as before. We also create a new AVL with the parse tree of the inserted block. If this is a new text of length $\ell$, then we create the $O(\ell / \log \sigma n)$ new nonterminals necessary to create a perfectly balanced tree on the new text. If, instead, it is a text copied from elsewhere in the previous document, then we obtain the $O(\log N)$ internal nodes in the parse tree that cover the source block. Rytter [45] also shows how to create an AVL from the nodes that cover an arbitrary substring of this type, using $O(\log \ell)$ new nodes. In any case, we have the first $h$ AVL subtrees of $T$, then the AVL tree of the block to be inserted, and then the second $h$ AVL subtrees of $T$. Those can be, again, merged into a single AVL using $O(\log N)$ new nonterminal symbols.

Summarizing, we can accommodate edits of whole blocks at the cost of $O(\log N)$ created or removed nonterminals per edit, plus $O(\ell / \log \sigma n)$ new nonterminals when a new block of length $\ell$ is created. As explained, we charge to $n$ not only the size of the initial document,
but also that of all the new blocks that are created upon insertions (i.e., not from copies of other parts of the text). If we call \( \ell \) the maximum size of a block edit and enforce \( \ell \leq n \), then \( N \leq D(n + s\ell) \leq D(n(s + 1)) \). We can therefore use \( \frac{1}{2} \log_2 N \) as our maximum metasymbol size, so that the grammar is of size at most \( O(N^{1/4} + n/\log_2 N + s \log N) = O(n/\log_2 N + s \log N) \), where we used that, since \( N \leq D(n + s\ell) \leq (2s + 1)n(s + 1) \), then either \( s \) or \( n \) must be \( \Omega(N^{1/3}) \). All the rest of our calculations stays unchanged.

\[ \textbf{Theorem 3.} \text{Let collection } \mathcal{D}, \text{ of total size } N, \text{ be formed by an initial document of length } n \text{ plus } D - 1 \text{ copies of it, with } s \text{ block edit operations performed on ranges or subtrees of documents. The blocks can be of length up to } n, \text{ and the sizes of the blocks created with new text are also counted in } n. \text{ Then } \mathcal{D} \text{ can be represented within } O(n(\log_2 \sigma + \log^2 N) \log D) \text{ bits, so that the } ndoc \text{ documents where a pattern of length } m \text{ appears can be listed in time } O(m^2 + m \log^{1+\epsilon} N \cdot ndoc), \text{ for any constant } \epsilon > 0. \]

The only word of caution about this model is that, if we are not given the edits explicitly, then obtaining the minimum number of block edits needed to convert the previous document to the current one may be costly. Depending on the edit operations permitted, finding the optimal set of block edits can be reasonably easy or NP-complete \[49, 32, 48, 2\].

### 6 Counting Pattern Occurrences

Our idea of associating augmented information with the wavelet tree of the grammar has independent interest. We illustrate this by developing a variant where we can count the number of times a pattern \( P \) occurs in the text without having to enumerate all the occurrences, as is the case with all the grammar-based indexes \[14, 15\]. In these structures, the primary occurrences are found as points in \( m - 1 \) ranges of a grid (recall Section 3.5). Each primary occurrence then triggers a number of secondary occurrences, distinct from those triggered by other primary occurrences. These secondary occurrences depend only on the point: if \( P \) occurs when \( B \) and \( C \) are concatenated in the rule \( A \rightarrow BC \), then every other occurrence of \( A \) or of its ancestors in the parse tree produces a distinct secondary occurrence.

We can therefore associate with each point the number of secondary occurrences it produces, and thus the total number of occurrences of \( P \) is the sum of the numbers associated with the points contained in the \( m - 1 \) ranges. By augmenting the wavelet tree (recall Section 3.2) of the grid, the sum in each range can be computed in time \( O(\log^3 r) \), using \( O(r \log N) \) further bits of space for the grid \[42, \text{ Thm. 6}\]. We now show how this result can be improved to time \( O(\log^{2+\epsilon} r) \) for any constant \( \epsilon > 0 \). Instead of only sums, we consider the more general case of a finite group \[42\], so our particular case is \((\{0, N\}, +, -, 0)\).

\[ \textbf{Theorem 4.} \text{Let a grid of size } r \times r \text{ store } r \text{ points with associated values in a group } (G, \oplus, -1, 0) \text{ of } N = |G| \text{ elements. For any } \epsilon > 0, \text{ a structure of } O((1/\epsilon) r \log N) \text{ bits can compute the sum } \oplus \text{ of the values in any rectangular range in time } O((1/\epsilon) \log^{2+\epsilon} r). \]

**Proof.** We modify the proof Navarro et al. \[42, \text{ Thm. 6}\]. They consider, for the sequence \( S_{a,b} \) of each wavelet tree node, the sequence of associated values \( V_{a,b} \). They store a cumulative array \( A_{a,b}[0] = 0 \) and \( A_{a,b}[i + 1] = A_{a,b}[i] \oplus V_{a,b}[i + 1] \), so that any range sum \( \oplus_{i \leq j \leq k} V_{a,b}[k] = A_{a,b}[j] \oplus A_{a,b}[i - 1]^{-1} \) is computed in constant time. The space to store \( A_{a,b} \) across all the levels is \( O(r \log r \log N) \) bits. To reduce it to \( O(r \log N) \), they store instead the cumulative sums

\[\text{Although the theorem states that it must be } t \geq 1, \text{ it turns out that one can use } t = \log r / \log N \text{ (i.e., } \tau = \log r \text{) to obtain this tradeoff (our } r \text{ is their } n \text{ and our } N \text{ is their } W).\]
of a sampled array \( V'_{a,b} \), where \( V'_{a,b}[i] = \oplus_{i-1 < k \leq i} V_{a,b}[k] \). They can then compute any range sum over \( V' \), with which they can compute any range sum over \( V \) except for up to \( \lg r \) elements in each extreme. Each of those extreme elements can be tracked up to the root in time \( O((1/\epsilon) \lg^r r) \), for any \( \epsilon > 0 \), using \( O((1/\epsilon) r \lg r) \) bits, as described at the end of Section 3.2. The root sequence \( V_{1,r} \) is stored explicitly, in \( r \lg N \) bits. Therefore, we can sum the values in any range of any wavelet tree node in time \( O((1/\epsilon) \lg^{1+\epsilon} r) \). Since any two-dimensional range is decomposed into \( O(\lg r) \) wavelet tree ranges, we can find the sum in time \( O((1/\epsilon) \lg^{2+\epsilon} r) \).

This immediately yields the first grammar-compressed index able to count pattern occurrences without locating them one by one.

\textbf{Theorem 5}. Let text \( T[1,N] \) be represented by a grammar of size \( r \). Then there exists an index of \( O(r \lg N) \) bits that can count the number of occurrences of a pattern \( P[1,m] \) in \( T \) in time \( O(m^2 + m \lg^{2+\epsilon} r) \), for any constant \( \epsilon > 0 \).

\section{Conclusions}

We have presented the first document listing index with worst-case space and time guarantees that are useful for repetitive collections. On a collection of size \( N \) formed by an initial document of length \( n \) and \( D-1 \) copies it, with \( s \) single-character or block edits applied on individual documents, or ranges of documents (when there is a linear structure of versions), or subtrees of documents (when there is a hierarchical structure of versions), our index uses \( O((n \lg s + s \lg^2 N) \lg D) \) bits and lists the \( ndoc \) documents where a pattern of length \( m \) appears in time \( O(m^2 + m \lg^{1+\epsilon} N \cdot ndoc) \), for any constant \( \epsilon > 0 \). We also prove that a previous index that had not been analyzed [13], but which has no useful worst-case time bounds for listing, uses \( O(n \lg s + s \lg^2 N) \) bits. As a byproduct, we offer a new variant of a structure that finds the distinct values in an array range [30, 46].

The general technique we use, of augmenting the range search data structure used by grammar-based indexes, can be used for other kind of summarization queries. We illustrate this by providing the first grammar-based index that uses \( O(r \lg N) \) bits, where \( r \) is the size of a grammar that generates the text, and counts the number of occurrences of a pattern in time \( O(m^2 + m \lg^{2+\epsilon} r) \), for any constant \( \epsilon > 0 \). As a byproduct, we improve a previous result [32] on summing values over two-dimensional point ranges.

\textbf{Future work}

The space of our document listing index is an \( O(\lg D) \) factor away from what can be expected from a grammar-based index. An important question is whether this space factor can be removed or reduced while retaining worst-case time guarantees for document listing.

Another interesting question is whether there exists an index (or a better analysis of this index) whose space and time can be bounded in terms of more general repetitiveness measures of the collection, for example in terms of the size \( r \) of a grammar that represents the text, as is the case of grammar-based pattern matching indexes that list all the occurrences of a pattern [14, 15], and of our own index that counts the number of occurrences of a pattern.

Yet another question is whether we can apply the idea of augmenting the two-dimensional data structure in order to handle other kinds of summarization queries that are of interest in pattern matching and document retrieval [40], for example counting the number of distinct documents where the pattern appears, or retrieving the \( k \) most important of those documents, or retrieving the occurrences that are in a range of documents.
A Proof of Correctness

We prove that our new document listing algorithm is correct. We remind that the algorithm proceeds as follows, to find the distinct elements in \( A[sp, ep] \). It starts recursively with \([i, j] = [sp, ep]\) and remembers the documents that have already been reported, globally. To process interval \([i, j]\), it considers \( A[i], A[i + 1], \ldots \) until finding an already reported element at \( A[d] \). Then it finds the minimum \( C[k] \) in \( C[d, j] \). If \( A[k] \) has been reported already, it stops; otherwise it reports \( A[k] \) and proceeds recursively in \( A[d, k - 1] \) and \( A[k + 1, j] \), in this order. (The actual algorithm is a slight variant of this procedure, and its correctness is established at the end.)

Lemma 6. The described algorithm reports the ndoc distinct elements in \( A[sp, ep] \) in \( O(ndoc) \) steps.

Proof. We prove that the algorithm reports the leftmost occurrence in \( A[sp, ep] \) of each distinct element. In particular, we prove by induction on \( j - i \) that, when run on any subrange \([i, j]\) of \([sp, ep]\), if (1) every leftmost occurrence in \( A[sp, i - 1] \) is already reported before processing \([i, j]\), then (2) every leftmost occurrence in \( A[sp, j] \) is reported after processing \([i, j]\). Condition (1) holds for \([i, j] = [sp, ep]\), and we need to establish that (2) holds after we process \([i, j] = [sp, ep]\). The base case \( i = j \) is trivial: the algorithm checks \( A[i] \) and reports it if it was not reported before.

On a larger interval \([i, j]\), the algorithm first reports \( d - i \) occurrences of distinct elements in \( A[i, d - 1] \). Since these were not reported before, by condition (1) they must be leftmost occurrences in \([sp, ep]\), and thus, after reporting all the leftmost occurrences of \( A[i, d - 1] \), condition (1) holds for any range starting at \( d \).

Now, we compute the position \( k \) with minimum \( C[k] \) in \( C[d, j] \). Note that \( A[k] \) is a leftmost occurrence iff \( C[k] < sp \), in which case it has not been reported before and thus it should be reported by the algorithm. The algorithm, indeed, detects that it has not been reported before and therefore recurses on \( A[d, k - 1] \), reports \( A[k] \), and finally recurses on \( A[k + 1, j] \). Since those subintervals are inside \([i, j]\), we can apply induction. In the call on \( A[d, k - 1] \), the invariant (1) holds and thus by induction we have that after the call the invariant (2) holds, so all the leftmost occurrences in \( A[sp, k - 1] = A[sp, d - 1] \cdot A[d, k - 1] \) have been reported. After we report \( A[k] \) too, the invariant (1) also holds for the call on \( A[k + 1, j] \), so by induction all the leftmost occurrences in \( A[sp, j] \) have been reported when the call returns.

In case \( C[k] \geq sp \), \( A[k] \) is not a leftmost occurrence in \( A[sp, ep] \), and moreover there are no leftmost occurrences in \( A[d, j] \), so we should stop since all the leftmost occurrences in \( A[sp, j] = A[sp, d - 1] \cdot A[d, j] \) are already reported. Indeed, it must hold \( sp \leq C[k] < d \), since otherwise \( C[C[k]] < C[k] \) and \( d \leq C[C[k]] \leq j \), contradicting the definition of \( k \). Therefore, by invariant (1), our algorithm already reported \( A[k] = A[C[k]] \), and hence it stops.

Then the algorithm is correct. As for the time, clearly the algorithm never reports the same element twice. The sequential part reports \( d - i \) documents in time \( O(d - i + 1) \). The extra \( O(1) \) can be charged to the caller, as well as the \( O(1) \) cost of the subranges that do not produce any result. Each calling procedure reports at least one element \( A[k] \), so it can absorb those \( O(1) \) costs, for a total cost of \( O(ndoc) \). ▶

3 Since \( A[k] \) does not appear in \( A[d, k - 1] \), the algorithm also works if \( A[k] \) is reported before the recursive calls, which makes it real-time.
The actual algorithm we use proceeds slightly differently, though. When it takes the minimum \( C[k] \) in \( C[d, j] \), if \( k = d \), it ignores that value and takes instead \( k = k' \), where \( k' \) is the second minimum in \( C[d, j] \). Note that, when processing \( C[d, j] \) in the algorithm of Lemma 5, \( A[d] \) is known to occur in \( A[sp, d-1] \). Therefore, \( C[d] \geq sp \), and if \( k = d \), the algorithm of Lemma 6 will stop. The actual algorithm chooses instead position \( k' \), but \( C[k'] \geq C[d] \geq sp \), and therefore, as seen in the proof, the algorithm has already reported \( A[k'] \), and thus the actual algorithm will also stop. Then the actual algorithm behaves identically to the algorithm of Lemma 6 and thus it is also correct.

References

1. Document retrieval on repetitive collections. *Information Retrieval*, 20:253–291, 2017.
2. H.-Y. Ann, C.-B. Yang, Y.-H. Peng, and B.-C. Liaw. Efficient algorithms for the block edit problems. *Information and Computation*, 208(3):221–229, 2010.
3. R. Baeza-Yates and B. Ribeiro. *Modern Information Retrieval*. Addison-Wesley, 1999.
4. J. Barbay, J. Fischer, and G. Navarro. LRM-trees: Compressed indices, adaptive sorting, and compressed permutations. *Theoretical Computer Science*, 459:26–41, 2012.
5. D. Belazzougui, F. Cunial, T. Gagie, N. Prezza, and M. Raffinot. Composite repetition-aware data structures. In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 9133, pages 26–39, 2015.
6. D. Belazzougui, T. Gagie, S. Gog, G. Manzini, and J. Sirén. Relative FM-indexes. In *Proc. 21st International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 8799, pages 52–64, 2014.
7. D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10(4):article 23, 2014.
8. P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. S. Rao, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015.
9. S. Bütüner, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.
10. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.
11. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.
12. D. R. Clark. *Compact PAT Trees*. PhD thesis, University of Waterloo, Canada, 1996.
13. F. Claude and J. I. Munro. Document listing on versioned documents. In *Proc. 20th SPIRE*, LNCS 8214, pages 72–83, 2013.
14. F. Claude and G. Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, 2010.
15. F. Claude and G. Navarro. Improved grammar-based compressed indexes. In *Proc. 19th Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 7608, pages 180–192, 2012.
16. H. H. Do, J. Jansson, K. Sadakane, and W.-K. Sung. Fast relative Lempel-Ziv self-index for similar sequences. *Theoretical Computer Science*, 532:14–30, 2014.
17. J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.
18. T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A faster grammar-based self-index. In *Proc. 6th International Conference on Language and Automata Theory and Applications (LATA)*, LNCS 7183, pages 240–251, 2012.
19 T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. LZ77-based self-indexing with faster pattern matching. In Proc. 11th Latin American Theoretical Informatics Symposium (LATIN), LNCS 8392, pages 731–742, 2014.

20 T. Gagie, K. Karhu, G. Navarro, S. J. Puglisi, and J. Sirén. Document listing on repetitive collections. In Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM), LNCS 7922, pages 107–119, 2013.

21 T. Gagie, G. Navarro, and N. Prezza. Optimal-time text indexing in BWT-runs bounded space. CoRR, abs/1705.10382, 2017.

22 T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. Theoretical Computer Science, 426-427:25–41, 2012.

23 L. Gasieniec, R. Kolpakov, I. Potapov, and P. Sant. Real-time traversal in grammar-based compressed files. In Proc. 15th Data Compression Conference (DCC), page 458, 2005.

24 R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 841–850, 2003.

25 W.-K. Hon, R. Shah, and J. Vitter. Space-efficient framework for top-k string retrieval problems. In Proc. 50th IEEE Annual Symposium on Foundations of Computer Science (FOCS), pages 713–722, 2009.

26 S. Huang, T. W. Lam, W.-K. Sung, S.-L. Tam, and S.-M. Yiu. Indexing similar DNA sequences. In Proc. 6th International Conference on Algorithmic Aspects in Information and Management (AAIM), LNCS 6124, pages 180–190, 2010.

27 D. Hucke, M. Lohrey, and C. P. Reh. The smallest grammar problem revisited. In Proc. 23rd Annual Symposium on Combinatorial Pattern Matching (CPM), LNCS 9954, pages 35–49, 2016.

28 A. Jez. Approximation of grammar-based compression via recompression. Theoretical Computer Science, 592:115–134, 2015.

29 A. Jez. A really simple approximation of smallest grammar. Theoretical Computer Science, 616:141–150, 2016.

30 S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. Theoretical Computer Science, 483:115–133, 2013.

31 A. Lempel and J. Ziv. On the complexity of finite sequences. IEEE Transactions on Information Theory, 22(1):75–81, 1976.

32 D. Lopresti and A. Tomkins. Block edit models for approximate string matching. Theoretical Computer Science, 181(1):159–179, 1997.

33 V. Mäkinen, G. Navarro, J. Sirén, and N. Valimäki. Storage and retrieval of highly repetitive sequence collections. Journal of Computational Biology, 17(3):281–308, 2010.

34 D. Morrison. PATRICIA – practical algorithm to retrieve information coded in alphanumeric. Journal of the ACM, 15(4):514–534, 1968.

35 I. Munro. Tables. In Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), LNCS 1180, pages 37–42, 1996.

36 S. Muthukrishnan. Efficient algorithms for document retrieval problems. In Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 657–666, 2002.

37 J. C. Na, H. Park, M. Crochemore, J. Holub, C. S. Iliopoulos, L. Mouchard, and K. Park. Suffix tree of alignment: An efficient index for similar data. In Proc. 24th International Workshop on Combinatorial Algorithms (IWOCA), LNCS 8288, pages 337–348, 2013.

38 J. C. Na, H. Park, S. Lee, M. Hong, T. Lecroq, L. Mouchard, and K. Park. Suffix array of alignment: A practical index for similar data. In Proc. 20th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 8214, pages 243–254, 2013.

39 G. Navarro. Indexing highly repetitive collections. In Proc. 23rd International Workshop on Combinatorial Algorithms (IWOCA), LNCS 7643, pages 274–279, 2012.
G. Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys*, 46(4):article 52, 2014.

G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

G. Navarro, Y. Nekrich, and L. Russo. Space-efficient data-analysis queries on grids. *Theoretical Computer Science*, 482:60–72, 2013.

Gonzalo Navarro. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016.

D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70, 2007.

W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.

K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.

H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *Journal of Discrete Algorithms*, 3(2–4):416–430, 2005.

D. Shapira and J. A. Storer. Edit distance with move operations. *Journal of Discrete Algorithms*, 5(2):380–392, 2007.

W. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, 1984.

E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1):100–118, 1985.

E. Verbin and W. Yu. Data structure lower bounds on random access to grammar-compressed strings. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 7922, pages 247–258, 2013.

X. Yang, B. Wang, C. Li, J. Wang, and X. Xie. Efficient direct search on compressed genomic data. In *Proc. 29th IEEE International Conference on Data Engineering (ICDE)*, pages 961–972, 2013.