Hardness of Frequency-Related Queries on Compressed Strings

Rajat De
Department of Computer Science, Stony Brook University, NY, USA

Dominik Kempa
Department of Computer Science, Stony Brook University, NY, USA

Abstract

Compressed indexing is a recent trend in the design of data structures that aims to support fundamental string queries in space proportional to the size of the data in compressed form. One of the most popular compression frameworks in this field is *grammar compression*. A length-n string $T \in \Sigma^n$ (where Σ is any finite set of size up to $|\Sigma| = |T|^{\mathcal{O}(1)}$) represented using a context-free grammar of size $|G|$ can be augmented to support random access queries (given any $i \in [1\,..\,n]$, return $T[i]$) in $\mathcal{O}(|G| \log^{\mathcal{O}(1)} n)$ space and $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ time. Numerous other queries, including pattern matching, longest common extension, lexicographical predecessor/successor, Burrows–Wheeler Transform, suffix array, and even suffix tree queries, can also be supported within the same bounds.

Despite this progress, one fundamental class of queries has remained elusive: frequency-related queries, such as reporting the number of occurrences of a symbol $c \in \Sigma$ in a substring $T(b\,..\,e]$ (the so-called *rank* query), or simply checking whether c occurs in $T(b\,..\,e]$ (the *symbol occurrence* query). To date, no fully general structure achieving $\mathcal{O}(|G| \log^{\mathcal{O}(1)} n)$ space and $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ query time is known. In this work, we establish new conditional lower bounds for frequency-related problems:

- We prove that answering rank and symbol occurrence queries on grammar-compressed texts in polylogarithmic time using a $\mathcal{O}(|G| \log^{\mathcal{O}(1)} n)$-space structure that is constructible from the input grammar in $\mathcal{O}(|G| \log^{\mathcal{O}(1)} n)$ time would imply an $\mathcal{O}(n^2 \log^{\mathcal{O}(1)} n)$-time algorithm for Boolean Matrix Multiplication (BMM), where the best known algorithms achieve $\mathcal{O}(n^{2.371339})$ time. Our result is achieved using a more general lower bound for efficiently answering a batch of rank and symbol occurrence queries.
- We generalize the above result, showing that even *LZ78-compressed strings* cannot support efficient rank queries. Since LZ78 is provably weaker than grammar compression, this yields a stronger result: rank and symbol occurrence queries remain hard for a wider class of compressors. We further show that achieving even *additive* approximations of rank queries would imply faster BMM algorithms.
- After establishing hardness of rank and symbol occurrence queries, we consider a broader class of frequency-related queries and show that, under the popular Orthogonal Vectors (OV) conjecture, other problems, including range distinct counting and range mode frequency queries, also cannot be efficiently supported in compressed space.

In summary, we develop new techniques for reasoning about computation over compressed data, and establish tight connections between compressed indexing and long-standing problems in fine-grained complexity. This sheds new light on compressed indexing by isolating a new class of frequency-related queries whose complexity hinges on known hard problems.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Data compression; Information systems → Information retrieval

Keywords and phrases compressed indexing, grammar compression, Lempel-Ziv 78 compression, conditional lower bounds, rank queries, range distinct count queries, range mode frequency queries

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.143

Related Version *Full Version*: `https://arxiv.org/abs/2607.07366`

Funding Partially funded by the NSF CAREER Award 2337891.

1 Introduction

Text indexing is a classical problem that asks to preprocess a given length-n sequence (text, string) $T \in \Sigma^n$ over an alphabet Σ, so that we can efficiently answer various queries on T. To date, numerous indexes using $\mathcal{O}(n)$ space are known, supporting a wide range of queries with query times typically ranging from $\mathcal{O}(1)$ to $\mathcal{O}(\log^{\mathcal{O}(1)} n)$. These include classical queries such as suffix arrays/trees [52, 63, 40], longest common extension (LCE) [63, 41], pattern matching [49, 7, 6, 53], rank/select [32, 4], lexicographical predecessor/successor [34], and many others [35, 54, 59, 17, 50].

While classical indexes remain fundamental in many applications and are still frequently used in practice, the need to index massive, highly repetitive sequences arising from projects such as the 100,000 Genomes Project [29] or the ongoing 1+ Million Genomes Initiative [16] has led to the development of compressed variants. A compressed index for a length-n text T is a data structure of size close to $\mathcal{O}(C(T))$ (where $C(T)$ is the output size of some lossless compression algorithm, or a measure of the repetitiveness of T) that supports efficient queries (typically in $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ time) on the original uncompressed text T.

The design of a compressed index primarily depends on the underlying compression representation. One particularly popular framework used in text indexing is *grammar compression*. In this method, a text $T \in \Sigma^n$ is represented as a context-free grammar (CFG) whose language consists only of the text T. One reason grammar compression has become a popular framework is its strong theoretical guarantees: the smallest grammar can be efficiently approximated within a $\mathcal{O}(\log n)$ factor [10, 61, 38], and grammar sizes are closely related to Lempel–Ziv compression [61] and the run-length compressed Burrows–Wheeler transform [42]. More generally, grammar compression belongs to a broader family of repetitiveness measures used in compressed indexing. Besides the size of a smallest grammar, this family includes the LZ77 size [65], the run-length BWT size $r(T)$ [9], the size of the smallest string attractor [45], and the substring complexity $\delta(T)$ [48], to name a few. Across a series of works [61, 10, 25, 45, 42, 46, 48], it has been shown that for these and other standard measures, the worst-case gaps are only $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ factors. Thus, if one ignores polylogarithmic factors, using smallest grammar size, LZ77 size, run-length BWT size, smallest attractor size, or $\delta(T)$ as the space benchmark leads to the same notion of compressed space.

State-of-the-art compressed indexes in this repetitiveness-based setting support the majority of central string processing queries, including:

- random access [8, 27, 2, 45, 48, 46, 43][1];
- longest common extension (LCE) [37, 28, 58, 46, 43];
- pattern matching [12, 13, 14, 24, 23, 11, 47, 48];
- suffix/LCP array, suffix tree, or Burrows–Wheeler transform (BWT) queries [26, 43].

Using the polylogarithmic relations above, their space usage can be bounded with respect to any grammar G representing a text $T \in \Sigma^n$ (where Σ is an alphabet of size up to polynomial in $|T|$, i.e., $|\Sigma| = |T|^{\mathcal{O}(1)}$) by $\mathcal{O}(|G| \cdot \log^{\mathcal{O}(1)} n)$, with query times ranging from $\mathcal{O}(\log \log n)$ to $\mathcal{O}(\log^{\mathcal{O}(1)} n)$. We refer to surveys of Navarro [56, 57] for further details.

Within this common compressed-space regime, the lower-bound picture began with proving that random access requires $\Omega(\frac{\log n}{\log \log n})$ time in $\mathcal{O}(\delta(T) \log^{\mathcal{O}(1)} n)$ space [62]. More recent work in [44] extends this understanding to most of the above non-frequency queries and

[1] Achieving efficient random access queries in $\mathcal{O}(n)$ space is trivial: it suffices to store the text $T \in \Sigma^n$ in plain form. In the compressed setting, however, even random access becomes a non-trivial query.

establishes a clean dichotomy. In the cell-probe model with word size $\Theta(\log n)$, any such index requires $\Omega(\frac{\log n}{\log\log n})$ time for random access, LCP-array, suffix array, inverse suffix array, and LCE queries, whereas BWT, PLCP, LF, inverse LF, and lexicographical predecessor/successor queries require $\Omega(\log\log n)$ time. These bounds match known upper bounds and already hold over a binary alphabet. Thus, this work yields two optimal query-time classes, $\Theta(\frac{\log n}{\log\log n})$ and $\Theta(\log\log n)$, for much of the classical compressed-indexing toolkit.

Despite this progress, one fundamental class of queries has remained elusive: frequency-related queries. These include reporting the number of occurrences of a symbol $c \in \Sigma$ in a substring $T(b\,..\,e]$ (the so-called *rank* query), or simply checking whether c occurs in $T(b\,..\,e]$ (the *symbol occurrence* query). Rank queries are among the most widely used queries in string processing [21, 33, 54, 55, 31, 60]. In the uncompressed setting, these queries can be supported easily in $\mathcal{O}(\log n)$ time by storing the list of occurrences of each character, and more efficient solutions are known in the case of an integer alphabet, i.e., when $\Sigma = [0\,..\,\sigma)$ [32, 30, 5].

In the compressed setting, however, the understanding of these queries is significantly more limited due to their dependence on the alphabet size. The classical queries (such as random access, pattern matching, suffix array, LCE, or BWT) can be supported in compressed space independently of the alphabet size, i.e., even when $\Sigma = \{1, \ldots, \sigma\}$ and $|\Sigma| = |G|^{\mathcal{O}(1)}$ or $|\Sigma| = |T|^{\mathcal{O}(1)}$. Rank and symbol occurrence queries, however, appear to depend strongly on the alphabet size. In the small-alphabet regime, upper and lower bounds are well understood:

- Belazzougui et al. [3] describe a data structure that, for any SLP G representing a string $T \in \Sigma^n$ with $\Sigma = \{1, \ldots, \sigma\}$, uses $\mathcal{O}(|\Sigma||G|)$ words of space and answers rank queries in $\mathcal{O}(\log n)$ time. They also describe a more general trade-off using $\mathcal{O}(\tau|\Sigma||G|\log_\tau(n/|G|))$ space and $\mathcal{O}(\log_\tau(n/|G|))$ query time, for $2 \le \tau \le \log^\epsilon n$ and any constant $\epsilon > 0$. For $\tau = \log^\epsilon n$, this yields a structure using $\mathcal{O}(|\Sigma||G|(\log^\epsilon n)\cdot\log(n/|G|)/\log\log n)$ words of space, answering queries in $\mathcal{O}(\log(n/|G|)/\log\log n)$ time.
- On the other hand, Prezza [60] generalized the lower bound of Verbin and Yu [62] and demonstrated that any data structure using $\mathcal{O}(|G|\log^{\mathcal{O}(1)} n)$ space cannot support rank queries in $o((\log n)/\log\log n)$ time. The same paper also shows how to achieve trade-offs similar to those in [3] for a wide range of compressed representations by generalizing them to string attractors [45].

Consequently, when the alphabet is small, e.g., when $|\Sigma| = \mathcal{O}(\log^{\mathcal{O}(1)} n)$, the above solutions yield structures using $\mathcal{O}(|G|\log^{\mathcal{O}(1)} n)$ space that support rank queries in the optimal time $\mathcal{O}(\frac{\log n}{\log\log n})$. For large alphabets (say, when $|\Sigma| = |T|^{\mathcal{O}(1)}$), the situation is different:

- The trade-off from [3] in this case yields structures using $\widetilde{\mathcal{O}}(|\Sigma||G|)$ space, which, for example, when $|\Sigma| = |G|$ corresponds to quadratic space $\mathcal{O}(|G|^2)$. At the other extreme, a naive solution using $\mathcal{O}(|G|)$ space answers rank queries in $\mathcal{O}(|G|)$ time.
- On the hardness side, the authors of [3] showed that if we can preprocess a grammar of size g with g' nonterminals that generates a string of length N in $T(g, g', N)$ time and produce a data structure of size $S(g, g', N)$ that answers rank queries on the generated string in time $t(g, g', N)$, then, given a DAG with $|V|$ nodes, $|E|$ edges (possibly with multiedges), β sources, and σ sinks, we can, after $\mathcal{O}(|E| + T(g, g', N))$-time preprocessing, produce a data structure of size $\mathcal{O}(|E| + S(g, g', N))$ that counts the number of distinct paths from any node of the DAG to one of the σ sinks in time $\mathcal{O}(t(g, g', N))$, where N is the number of distinct paths that connect the β sources to the σ sinks.

In other words, when the alphabet is large, e.g., when $|\Sigma| = |T|^{\mathcal{O}(1)}$, it is currently not known whether rank queries can be supported in $\mathcal{O}(|G|\log^{\mathcal{O}(1)} n)$ space and $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ query time. Although [3] sheds some light on this hardness by connecting the problem to the

DAG path-counting problem, prior to this work, no precise quantitative lower bounds had been developed beyond this general reduction, and large-alphabet rank and symbol occurrence queries remain a central unresolved challenge in compressed indexing. Furthermore, the known hardness evidence [3] applies only to the relatively powerful rank queries, despite the fact that no indexes are known even for the much simpler symbol occurrence queries.

The *large-alphabet* case for rank and symbol occurrence queries has also recently been shown to be important for 2D string indexing. In [19], it is proved that if, for a 2D SLP G_M representing a 2D string (array, matrix, image) $M \in \{\mathtt{0},\mathtt{1}\}^{r\times c}$, there exists a data structure of size $\mathcal{O}(|G_M|\log^{\mathcal{O}(1)} n)$ (where $n = \max(r,c)$) that answers any of the basic 2D queries about subrectangles or subsquares (including sum, equality, longest common extension, or all-zero queries), then for any (1D) SLP G_T representing a (1D) string $T \in \Sigma^*$, where $|\Sigma| = |T|^{\mathcal{O}(1)}$, there exists a structure of size $\mathcal{O}(|G_T|\log^{\mathcal{O}(1)}|T|)$ that answers symbol occurrence queries in $\mathcal{O}(\log^{\mathcal{O}(1)}|T|)$ time. A similar reduction is proved for rank queries. In other words, a notion of hardness for rank or symbol occurrence queries on 1D compressed strings over *(polynomially) large alphabets* would imply hardness for 2D compressed indexing of 2D strings over a *binary alphabet*. Given the fundamental role of rank and symbol occurrence queries in many algorithms [55, 54, 31, 60, 21], accentuated further by the recent reductions in [19], we thus ask:

Can frequency-related queries (such as rank and symbol occurrence queries) on large-alphabet strings be efficiently supported in compressed space?

Our Results. We present a series of reductions showing that fully general support for fundamental frequency-related queries over large alphabets (including rank and symbol occurrence queries, as well as the related problems of range distinct counting and range mode frequency) would either break long-standing barriers in computational complexity or require substantially new approaches.

More specifically, we first prove that efficient support for rank and symbol occurrence queries would improve the state-of-the-art algorithms for Boolean matrix multiplication.[2] Given any matrices $A, B \in \{\mathtt{0},\mathtt{1}\}^{n\times n}$, the currently best algorithm for this task runs in $\mathcal{O}(n^{2.371339})$ time [1]. Although we are not aware of any substantial barriers ruling out the existence of a faster algorithm, and an $\mathcal{O}(n^2\log^{\mathcal{O}(1)} n)$-time algorithm for this problem may exist, our result nevertheless shows that obtaining fast rank or symbol occurrence queries over grammars would have consequences well beyond compressed indexing. In this sense, our work is similar in spirit to the conditional lower bounds for text indexing with mismatches and differences by Cohen-Addad et al. [15]. Specifically, we prove the following theorem.

▶ **Theorem 1.** *If there exists an algorithm that, given any SLG $G = (V, \Sigma, R, S)$ generating a string $T \in \Sigma^N$ (where $\Sigma = \{1,\ldots,\sigma\}$ and $\sigma = \Omega(N^{1/3})$), answers any batch of m symbol occurrence queries (Definition 13) on T in $\mathcal{O}((|G|+m)\log^{\mathcal{O}(1)} N)$ total time, then the Boolean matrix product of any two $n\times n$ Boolean matrices can be computed in $\mathcal{O}(n^2\log^{\mathcal{O}(1)} n)$ time.*

This immediately implies that unless we can multiply Boolean matrices in $\mathcal{O}(n^2\log^{\mathcal{O}(1)} n)$ time, there is no compressed index for symbol occurrence queries on grammar-compressed text that is simultaneously small, fast to query, and quickly constructible.

[2] Given any $A, B \in \{\mathtt{0},\mathtt{1}\}^{n\times n}$, Boolean matrix multiplication computes a matrix $C = AB$, where $C[i,j] = \bigvee_{k=1}^{n} A[i,k] \wedge B[k,j]$ holds for every $i, j \in [1\,..\,n]$.

▶ **Corollary 2.** *If there exists a data structure that, given any SLG G representing a string $T \in \Sigma^N$ (where $\Sigma = \{1, \ldots, \sigma\}$ and $\sigma = \Omega(N^{1/3})$), answers symbol occurrence queries (Definition 13) on T in $\mathcal{O}(\log^{\mathcal{O}(1)} N)$ time, and takes $\mathcal{O}(|G| \log^{\mathcal{O}(1)} N)$ time to construct, then the Boolean matrix product of any two $n \times n$ Boolean matrices can be computed in $\mathcal{O}(n^2 \log^{\mathcal{O}(1)} n)$ time.*

Since rank queries allow answering symbol occurrence queries, the above results also hold for rank queries. We state them for symbol occurrence queries, as this establishes the hardness of these easier queries (i.e., yields a stronger result). To our knowledge, these are the first hardness results for symbol occurrence queries, establishing a surprisingly strong barrier in indexing for these extremely basic frequency-related queries.

It is worth separating the above result from the well-understood rank queries used inside BWT-based indexes. The *Burrows–Wheeler transform (BWT)* [9] is a permutation of the text that plays a central role in data compression and text indexing [21, 26]: the FM-index of Ferragina and Manzini [21] relies on rank over the BWT stored in plain form, whereas the r-index of Gagie et al. [26] relies on rank over its run-length-compressed form, using $\mathcal{O}(r(T))$ or $\mathcal{O}(r(T) \log n)$ space, where $r(T)$ is the number of runs in the BWT of T. In both cases, the relevant primitive is rank on the BWT sequence itself, either uncompressed or only run-length-compressed, and this setting is well understood from the upper and lower bound perspectives [22]. The surprising point is that $r(T)$ and the smallest grammar size $g^*(T)$ are known to be within $\log^{\mathcal{O}(1)} n$ factors of each other in the worst case [42, 25]: thus, rank over the run-length-compressed BWT is understood, while rank over the original grammar-compressed text remains challenging.

The hardness is not confined to grammar compression: as explained next, the above conditional lower bounds for symbol occurrence queries hold even for significantly weaker compression methods.

Generalization to LZ78. LZ78 [66] is a classical compression method that, unlike other compression schemes such as LZ77 [65] or grammar compression, admits significantly faster algorithms and queries on the underlying text. For example, the complexity of random access queries on LZ78-compressed texts (allowing $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ overhead in space) is $\Theta(\log \log n)$ [20, 18] time, whereas, as noted above, for LZ77, the optimal query time for random access is $\Theta(\frac{\log n}{\log \log n})$ [3, 2, 27, 62]. This decrease in query time comes at the price of reduced compression ratio: while LZ77 and grammar compression are capable of exponential compression, LZ78 cannot compress a length-n string below $\Omega(\sqrt{n})$ bits. This motivates us to ask whether rank and symbol occurrence queries can also be answered more efficiently on LZ78-compressed texts. We answer this question negatively: we show that the above reduction from Boolean matrix multiplication holds even on LZ78-compressed text.

▶ **Theorem 3.** *If there exists an algorithm that, given the LZ78 representation (Definition 27) of a string $T \in \Sigma^N$ (where $\Sigma = \{1, \ldots, \sigma\}$ and $\sigma = \Omega(N^{1/3})$), answers any batch of m symbol occurrence queries (Definition 13) on T in $\mathcal{O}((z_{78}(T) + m) \log^{\mathcal{O}(1)} N)$ total time, then the Boolean matrix product of any two $n \times n$ Boolean matrices can be computed in $\mathcal{O}(n^2 \log^{\mathcal{O}(1)} n)$ time.*

As in the grammar-compressed case, this immediately implies that unless we can multiply any two Boolean matrices in $\mathcal{O}(n^2 \log^{\mathcal{O}(1)} n)$ time, there is no compressed index for symbol occurrence queries on LZ78-compressed text that is simultaneously small, fast to query, and quickly constructible. The same implication also holds for rank queries.

▶ **Corollary 4.** *If there exists a data structure that, given the LZ78 representation (Definition 27) of a string $T \in \Sigma^N$ (where $\Sigma = \{1, \ldots, \sigma\}$ and $\sigma = \Omega(N^{1/3})$), answers symbol occurrence queries (Definition 13) on T in $\mathcal{O}(\log^{\mathcal{O}(1)} N)$ time, and takes $\mathcal{O}(z_{78}(T) \log^{\mathcal{O}(1)} N)$ time to construct, then the Boolean matrix product of any two $n \times n$ Boolean matrices can be computed in $\mathcal{O}(n^2 \log^{\mathcal{O}(1)} n)$ time.*

Approximate Rank Queries. The above results raise a natural question of whether *approximating* rank queries is easier than computing rank values exactly. Since all of the above results hold even for symbol occurrence queries (which distinguish whether $\mathrm{rank}_T(b, e, c) = 0$ or $\mathrm{rank}_T(b, e, c) \geq 1$), we immediately obtain the hardness of multiplicative approximation (since it would distinguish between the two cases). This leaves open the possibility of an additive approximation of $\mathrm{rank}_T(b, e, c)$. We show that even additive approximation is hard.

▶ **Theorem 5.** *Let $\mu \in (0, 1)$ be a constant. If there exists an algorithm that, given any SLG $G = (V, \Sigma, R, S)$ generating a string $T \in \Sigma^N$ (where $\Sigma = \{1, \ldots, \sigma\}$ and $\sigma = \Omega(N^{\mu/3})$), computes an $\lfloor N^{1-\mu} \rfloor$-additive approximation of any batch of m two-sided rank queries (see Definitions 15 and 18 and Remark 17) in $\mathcal{O}((|G| + m) \log^{\mathcal{O}(1)} N)$ total time, then the Boolean matrix product of any two $n \times n$ Boolean matrices can be computed in $\mathcal{O}(n^2 \log^{\mathcal{O}(1)} n)$ time.*

▶ **Corollary 6.** *Let $\mu \in (0, 1)$ be a constant. If there exists a data structure that, given any SLG G representing a string $T \in \Sigma^N$ (where $\Sigma = \{1, \ldots, \sigma\}$ and $\sigma = \Omega(N^{\mu/3})$), computes an $\lfloor N^{1-\mu} \rfloor$-additive approximation of a given two-sided rank query (Definitions 15 and 18 and Remark 17) in $\mathcal{O}(\log^{\mathcal{O}(1)} N)$ time, and takes $\mathcal{O}(|G| \log^{\mathcal{O}(1)} N)$ time to construct, then the Boolean matrix product of any two $n \times n$ Boolean matrices can be computed in $\mathcal{O}(n^2 \log^{\mathcal{O}(1)} n)$ time.*

Hardness of Other Frequency-Related Queries. After establishing the hardness of the most basic frequency-related queries, we turn our attention to other related queries, namely, the range distinct counting and range mode frequency queries. Consider a length-n string $T \in \Sigma^n$. The *range distinct counting* query, given any $b, e \in [0 \,..\, n]$, returns $\mathrm{distinct}_T(b, e) := |\{T[i] : i \in (b \,..\, e]\}|$, i.e., the number of distinct elements in the block $T(b \,..\, e]$; see Definition 19. Similarly to the queries considered above, range distinct counting queries can be answered efficiently in the uncompressed setting (in [39], the authors describe an algorithm that achieves $\mathcal{O}(n \log n)$ preprocessing time and $\mathcal{O}(\log n)$ query time).

We prove that, assuming the popular *Orthogonal Vectors Conjecture* (Conjecture 32), answering a range distinct counting query on a grammar-compressed string essentially requires inspecting the entire grammar. As before, we obtain this result as a corollary of the following stronger batch lower bound:

▶ **Theorem 7.** *Assuming the Orthogonal Vectors Conjecture (Conjecture 32), there is no algorithm that, given any SLG $G = (V, \Sigma, R, S)$ representing a string $T \in \Sigma^N$ (where $\Sigma = \{1, \ldots, \sigma\}$ and $\sigma = \Omega((N/\log N)^{1/2})$), answers any batch of $m = \Omega(|G|/\log N)$ range distinct count queries (Definition 19) in $\mathcal{O}(m|G|^{1-\epsilon} \log^{\mathcal{O}(1)} N)$ time, for any constant $\epsilon > 0$.*

▶ **Corollary 8.** *Assuming the Orthogonal Vectors Conjecture (Conjecture 32), there is no data structure that, given any SLG G representing a string $T \in \Sigma^N$ (where $\Sigma = \{1, \ldots, \sigma\}$ and $\sigma = \Omega((N/\log N)^{1/2})$), answers range distinct counting queries (Definition 19) on T in $\mathcal{O}(|G|^{1-\epsilon} \log^{\mathcal{O}(1)} N)$ time, and takes $\mathcal{O}(|G|^{2-\epsilon} \log^{\mathcal{O}(1)} N)$ time to construct, for any $\epsilon > 0$.*

We complement this hardness result with essentially a matching upper bound, showing how to answer a batch of m range distinct counting queries in $\mathcal{O}(m|G| \log n)$ time.

We conclude our set of results by presenting an analogous hardness argument for *range mode frequency queries*. Given any $b, e \in [0\,..\,n]$ satisfying $b < e$, the range mode frequency query returns the frequency of the most common element in $T(b\,..\,e]$; see Definition 21.

▶ **Theorem 9.** *Assuming the Orthogonal Vectors Conjecture (Conjecture 32), there is no algorithm that, given any SLG $G = (V, \Sigma, R, S)$ representing a string $T \in \Sigma^N$ (where $\Sigma = \{1, \ldots, \sigma\}$ and $\sigma = \Omega((N/\log N)^{1/2})$), answers any batch of $m = \Omega(|G|/\log N)$ range mode frequency queries (Definition 21) in $\mathcal{O}(m|G|^{1-\epsilon}\log^{\mathcal{O}(1)} N)$ time, for any constant $\epsilon > 0$.*

▶ **Corollary 10.** *Assuming the Orthogonal Vectors Conjecture (Conjecture 32), there is no data structure that, given any SLG G representing a string $T \in \Sigma^N$ (where $\Sigma = \{1, \ldots, \sigma\}$ and $\sigma = \Omega((N/\log N)^{1/2})$), answers range mode frequency queries (Definition 21) on T in $\mathcal{O}(|G|^{1-\epsilon}\log^{\mathcal{O}(1)} N)$ time, and takes $\mathcal{O}(|G|^{2-\epsilon}\log^{\mathcal{O}(1)} N)$ time to construct, for any $\epsilon > 0$.*

Implications of our Hardness Results for Other Range Queries on Grammar-Compressed Strings. Our hardness results for symbol occurrence queries immediately imply the hardness of other popular fundamental queries (of which symbol occurrence is just a special case), such as position-restricted pattern matching introduced by Mäkinen and Navarro in [51]. These hardness results hold even on LZ78-compressed strings.

▶ **Corollary 11.** *If there exists a data structure that, given any SLG G representing a string $T \in \Sigma^N$ (where $\Sigma = \{1, \ldots, \sigma\}$ and $\sigma = \Omega(N^{1/3})$), answers position-restricted pattern matching queries (that, given any pattern $P \in \Sigma^M$ and any pair $b, e \in [0\,..\,N-M+1]$, checks whether there exists $i \in (b\,..\,e]$ satisfying $T[i\,..\,i+M) = P$)[3] on T in $\mathcal{O}(M^{\mathcal{O}(1)}\log^{\mathcal{O}(1)} N)$ time, and takes $\mathcal{O}(|G|\log^{\mathcal{O}(1)} N)$ time to construct, then the Boolean matrix product of any two $n \times n$ Boolean matrices can be computed in $\mathcal{O}(n^2\log^{\mathcal{O}(1)} n)$ time.*

▶ **Corollary 12.** *If there exists a data structure that, given the LZ78 representation (Definition 27) of a string $T \in \Sigma^N$ (where $\Sigma = \{1, \ldots, \sigma\}$ and $\sigma = \Omega(N^{1/3})$), answers position-restricted pattern matching queries (that, given any pattern $P \in \Sigma^M$ and any pair $b, e \in [0\,..\,N-M+1]$, checks whether there exists $i \in (b\,..\,e]$ satisfying $T[i\,..\,i+M) = P$) on T in $\mathcal{O}(M^{\mathcal{O}(1)}\log^{\mathcal{O}(1)} N)$ time, and takes $\mathcal{O}(z_{78}(T)\log^{\mathcal{O}(1)} N)$ time to construct, then the Boolean matrix product of any two $n \times n$ Boolean matrices can be computed in $\mathcal{O}(n^2\log^{\mathcal{O}(1)} n)$ time.*

2 Preliminaries

2.1 Basic Definitions

2.1.1 Strings

A *string* is a finite sequence of characters drawn from a given *alphabet* Σ. The length of a string S is denoted by $|S|$. For $i \in [1\,..\,|S|]$,[4] the ith leftmost character of S is denoted $S[i]$. A *substring* of S is any string of the form $S[i\,..\,j) = S[i]S[i+1]\cdots S[j-1]$ for some $1 \le i \le j \le |S|+1$. Substrings of the forms $S[1\,..\,j)$ and $S[i\,..\,|S|+1)$ are called *prefixes* and *suffixes*, respectively. The *concatenation* of two strings S_1 and S_2, namely the string

[3] We obtain symbol occurrence queries as a special case of position-restricted pattern matching queries simply by setting $M = 1$; see Definition 13.

[4] For $i, j \in \mathbb{Z}$, we define $[i\,..\,j] = \{k \in \mathbb{Z} : i \le k \le j\}$, $[i\,..\,j) = \{k \in \mathbb{Z} : i \le k < j\}$, and $(i\,..\,j] = \{k \in \mathbb{Z} : i < k \le j\}$.

$S_1[1]\cdots S_1[|S_1|]S_2[1]\cdots S_2[|S_2|]$, is denoted by S_1S_2 or $S_1 \cdot S_2$. For $k \in \mathbb{Z}_{\geq 0}$, we define $S^k = \bigodot_{i=1}^{k} S$ as the concatenation of k copies of S; by convention, $S^0 = \varepsilon$ denotes the *empty string*.

2.1.2 Matrices and Vectors

A matrix $A \in \{0,1\}^{n\times m}$ is a two-dimensional array with n rows and m columns. The entry at the intersection of row i and column j is denoted as $A[i,j]$ for every $(i,j) \in [1\,..\,n] \times [1\,..\,m]$. All matrix products used in this paper are over the Boolean semiring, i.e., for $A, B \in \{0,1\}^{n\times n}$, we define $C = AB$ as $C[i,j] = \bigvee_{k=1}^{n} A[i,k] \wedge B[k,j]$ for each $i,j \in [1\,..\,n]$.

A binary vector v of dimension d is an element of $\{0,1\}^d$. The ith coordinate of v is denoted as $v[i]$. For two vectors u, v of dimension d, we denote their dot product by $\langle u, v\rangle = \sum_{i=1}^{d} u[i]\cdot v[i]$. We say two vectors u and v are *orthogonal* if $\langle u, v\rangle = 0$.

2.2 Frequency-Based Queries

2.2.1 Symbol Occurrence and Rank Queries

▶ **Definition 13** (Symbol occurrence). *Let $T \in \Sigma^n$. For every $b, e \in [0\,..\,n]$ and every $c \in \Sigma$, we define*

$$\mathrm{occurs}_T(b,e,c) := \begin{cases} 1 & \text{if there exists } j \in (b\,..\,e] \text{ such that } T[j] = c, \\ 0 & \text{otherwise.} \end{cases}$$

▶ **Example 14.** For $T = \mathtt{acaaba}$ it holds $\mathrm{occurs}_T(1,4,\mathtt{a}) = 1$ and $\mathrm{occurs}_T(1,4,\mathtt{b}) = 0$.

▶ **Definition 15** (Rank). *Let $T \in \Sigma^n$. For every $b, e \in [0\,..\,n]$ and every $c \in \Sigma$, we define*

$$\mathrm{rank}_T(b,e,c) := |\{i \in (b\,..\,e] : T[i] = c\}|.$$

For every $j \in [0\,..\,n]$ and every $c \in \Sigma$, we also define

$$\mathrm{rank}_T(j,c) := \mathrm{rank}_T(0,j,c) = |\{i \in (0\,..\,j] : T[i] = c\}|.$$

▶ **Example 16.** For $T = \mathtt{acaaba}$, it holds $\mathrm{rank}_T(5,\mathtt{a}) = 3$ and $\mathrm{rank}_T(2,5,\mathtt{a}) = 2$.

▶ Remark 17. To distinguish between the two types of rank queries in Definition 15, by a *two-sided rank query*, we mean the computation of $\mathrm{rank}_T(b,e,c)$, and by *one-sided rank query*, we refer to the computation of $\mathrm{rank}_T(j,c)$. Note that, for every $T \in \Sigma^n$, $c \in \Sigma$, and $b, e \in [0\,..\,n]$ such that $b \leq e$, it holds $\mathrm{rank}_T(b,e,c) = \mathrm{rank}_T(e,c) - \mathrm{rank}_T(b,c)$.

▶ **Definition 18** (Additive rank approximation). *Let $T \in \Sigma^n$, $c \in \Sigma$, and $b, e \in [0\,..\,n]$. Let $k \in \mathbb{Z}_{\geq 1}$. We say that $x \in \mathbb{Z}$ is a* k-additive approximation *of $\mathrm{rank}_T(b,e,c)$ if it holds*

$$\mathrm{rank}_T(b,e,c) - k < x < \mathrm{rank}_T(b,e,c) + k.$$

2.2.2 Range Distinct Count Queries

▶ **Definition 19** (Range distinct count). *Let $T \in \Sigma^n$. For any $b, e \in [0\,..\,n]$, we define*

$$\mathrm{distinct}_T(b,e) := |\{T[i] : i \in (b\,..\,e]\}|.$$

▶ **Example 20.** For $T = \mathtt{acaaba}$, it holds $\mathrm{distinct}_T(0,4) = 2$ and $\mathrm{distinct}_T(1,5) = 3$.

2.2.3 Range Mode Frequency Queries

▶ **Definition 21** (Range mode frequency). *Let $T \in \Sigma^n$. For every $b, e \in [0\,..\,n]$, we define* mode-freq$_T(b, e)$ *as the frequency of the most common element in $T(b\,..\,e]$, i.e.,*

$$\text{mode-freq}_T(b, e) := \max_{c \in \Sigma} \text{rank}_T(b, e, c)$$

(see Definition 15).

▶ **Example 22.** For $T = \texttt{acaaba}$ it holds mode-freq$_T(0, 4) = 3$ and mode-freq$_T(3, 6) = 2$.

2.3 Compressed Representations

2.3.1 Grammar Compression

A *context-free grammar (CFG)* is a tuple $G = (V, \Sigma, R, S)$ such that $V \cap \Sigma = \emptyset$ and

- V is a finite nonempty set of *nonterminals* or *variables*,
- Σ is a finite nonempty set of *terminal* symbols,
- $R \subseteq V \times (V \cup \Sigma)^*$ is a set of *productions* or *rules*, and
- $S \in V$ is the special *starting nonterminal*.

We say that $u \in (V \cup \Sigma)^*$ *derives* v, and write $u \Rightarrow^* v$, if v can be obtained from u by repeatedly replacing nonterminals according to the rule set R. We then denote $L(G) := \{w \in \Sigma^* \mid S \Rightarrow^* w\}$.

By a *straight-line grammar (SLG)* we mean a CFG $G = (V, \Sigma, R, S)$ such that:

1. there exists an ordering $(N_1, \ldots, N_{|V|})$ of all elements in V such that, for every $(N, \gamma) \in R$, letting $i \in [1\,..\,|V|]$ be such that $N = N_i$, it holds $\gamma \in (\{N_{i+1}, \ldots, N_{|V|}\} \cup \Sigma)^*$, and
2. for every nonterminal $N \in V$, there exists exactly one $\gamma \in (V \cup \Sigma)^*$ such that $(N, \gamma) \in R$.

The unique string $\gamma \in (V \cup \Sigma)^*$ such that $(N, \gamma) \in R$ is called the *definition* or *right-hand side* of nonterminal N and is denoted $\text{rhs}_G(N)$. Note that in an SLG, for every $\alpha \in (V \cup \Sigma)^*$, there exists exactly one string $\gamma \in \Sigma^*$ satisfying $\alpha \Rightarrow^* \gamma$. Such γ is called the *expansion* of α and is denoted $\exp_G(\alpha)$. In particular, in an SLG, we have $|L(G)| = 1$. We define the size of an SLG as $|G| = \sum_{N \in V} \max(|\text{rhs}_G(N)|, 1)$.

An SLG in which, for every $N \in V$, it holds $\text{rhs}_G(N) = XY$, where $X, Y \in V$, or $\text{rhs}_G(N) = a$, where $a \in \Sigma$, is called a *straight-line program (SLP)*.

▶ **Observation 23.** *Every SLG $G = (V, \Sigma, R, S)$ satisfying $L(G) \neq \{\varepsilon\}$ can be transformed in $\mathcal{O}(|G|)$ time into an SLP $G' = (V', \Sigma, R', S')$ satisfying $|G'| = \Theta(|G|)$ and $L(G') = L(G)$.*

▶ **Observation 24.** *If G is an SLP such that $L(G) = \{T\}$, where $|T| = n$, then $|G| = \Omega(\log n)$.*

▶ **Lemma 25** ([27]). *Given any SLP $G = (V, \Sigma, R, S)$ representing a string $T \in \Sigma^n$, we can in $\mathcal{O}(|G|)$ time construct an SLP $G' = (V', \Sigma, R', S')$ of height $\mathcal{O}(\log n)$ that represents the same string T.*

2.3.2 Lempel–Ziv (LZ78) Compression

▶ **Definition 26** (Lempel–Ziv (LZ78) factorization [66]). *The* LZ78 factorization *of a string T is a factorization $T = f_1 f_2 \cdots f_k$ defined such that, letting $f_0 = \varepsilon$, for every $i \in [1\,..\,k]$, after $f_1 \cdots f_{i-1}$ has been parsed, f_i is the longest prefix of the remaining suffix $f_i f_{i+1} \cdots f_k$ such that there exists $j \in [0\,..\,i)$ satisfying $f_i = f_j \cdot c$ for some symbol c. The elements f_i of the factorization (where $i \geq 1$) are called* phrases. *We denote the number of phrases in the LZ78 factorization of T by $z_{78}(T)$ (i.e., $k = z_{78}(T)$).*

▶ **Definition 27** (Lempel–Ziv (LZ78) representation). *Let $T = f_1 f_2 \cdots f_k$ be the LZ78 factorization of T (Definition 26), and let $f_0 = \varepsilon$. The* LZ78 representation *of T is a sequence of pairs $((p_1, c_1), \ldots, (p_k, c_k))$ such that, for every $i \in [1\,..\,k]$, it holds $p_i \in [0\,..\,i)$ and $f_i = f_{p_i} \cdot c_i$.*

▶ Remark 28. Observe that if $T = f_1 f_2 \cdots f_k$ is the LZ78 factorization of T, then all phrases $f_1, \ldots, f_{k-1}$ are distinct. Consequently, the LZ78 representation (Definition 27) of T is unique.

▶ Remark 29. Note that if $T \in [0\,..\,\sigma)^n$ for some $\sigma \in \mathbb{Z}_{\geq 1}$, then the LZ78 representation of T (Definition 27) encodes T using $\mathcal{O}(z_{78}(T) \cdot (\log z_{78}(T) + \log \sigma)) = \mathcal{O}(z_{78}(T) \cdot (\log n + \log \sigma))$ bits of space.

▶ **Example 30.** The LZ78 factorization (Definition 26) of the string $T = \texttt{010212020}$ is $T = \texttt{0} \cdot \texttt{1} \cdot \texttt{02} \cdot \texttt{12} \cdot \texttt{020}$ with $z_{78}(T) = 5$ phrases, and the LZ78 representation (Definition 27) of T is: $((0, \texttt{0}), (0, \texttt{1}), (1, \texttt{2}), (2, \texttt{2}), (3, \texttt{0}))$.

2.4 Hardness Assumptions

2.4.1 Boolean Matrix Multiplication (BMM)

The first problem we use as the basis of our hardness arguments is the Boolean Matrix Multiplication problem.

Boolean Matrix Multiplication (BMM)	
Input:	Two matrices $A, B \in \{\texttt{0}, \texttt{1}\}^{n \times n}$.
Output:	The matrix $C = AB$, where $C[i,j] = \bigvee_{k=1}^{n} A[i,k] \wedge B[k,j]$ for every $i, j \in [1\,..\,n]$.

The state-of-the-art for the above problem is summarized below, and currently no algorithm is known that solves this problem in $\mathcal{O}(n^2 \log^{\mathcal{O}(1)} n)$ time.

▶ **Theorem 31** ([1]). *Given any two Boolean matrices $A, B \in \{\texttt{0}, \texttt{1}\}^{n \times n}$, we can compute their product in $\mathcal{O}(n^{2.371339})$ time.*

2.4.2 Orthogonal Vectors (OV)

The second problem we use as a basis for our hardness results is the Orthogonal Vectors problem, one of the most widely used tools in fine-grained complexity theory (see, e.g., [64] and references therein for a recent discussion).

Orthogonal Vectors (OV)	
Input:	A set of vectors $A \subseteq \{\texttt{0}, \texttt{1}\}^d$ with $\lvert A \rvert = n$.
Output:	Determine whether there exist $x, y \in A$ such that $\langle x, y \rangle = \sum_{i=1}^{d} x[i] \cdot y[i] = 0$.

▶ **Conjecture 32** (Orthogonal Vectors Conjecture). *For every constant $\epsilon > 0$, there exists a constant $c \geq 1$ such that OV cannot be solved in $\mathcal{O}(n^{2-\epsilon})$ time on instances with $d = c \log n$.*

2.5 Model of Computation

We use the standard word RAM model of computation [36] with w-bit machine words, where $w = \Theta(\log n)$, and all standard bitwise and arithmetic operations take $\mathcal{O}(1)$ time. Unless explicitly stated otherwise, we measure space complexity in machine words.

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

$$\mathrm{RowOnes}_{A,1} = 13 \quad \mathrm{RowOnes}_{B,1} = 2$$
$$\mathrm{RowOnes}_{A,2} = 2 \quad \mathrm{RowOnes}_{B,2} = 1$$
$$\mathrm{RowOnes}_{A,3} = 12 \quad \mathrm{RowOnes}_{B,3} = 23$$

$$AB = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

$$T_{A,B,1} = 223$$
$$T_{A,B,2} = 1$$
$$T_{A,B,3} = 21$$
$$T_{A,B} = \ 223121$$

Figure 1 Example showing matrices A and B along with strings $\mathrm{RowOnes}_{A,i}$, $\mathrm{RowOnes}_{B,i}$, $T_{A,B,i}$ for $i \in [1\,..\,3]$, and $T_{A,B}$.

3 Technical Overview

Reducing Boolean Matrix Multiplication to Symbol Occurrence Queries. Our main idea is to construct a large but compressible string that lets us compute a single entry of the product of two matrices using a *single* symbol occurrence query (see Definition 13). Given two Boolean matrices $A, B \in \{\mathtt{0},\mathtt{1}\}^{n\times n}$, we start by defining the following objects (see Figure 1 for an example containing every object defined below):

1. For each $i \in [1\,..\,n]$, we define $\mathrm{RowOnes}_{A,i}$ as the string containing (in increasing order) the indices of all columns j such that $A[i,j] = \mathtt{1}$.
2. For each $i \in [1\,..\,n]$, the string $\mathrm{RowOnes}_{B,i}$ is defined analogously, i.e., $\mathrm{RowOnes}_{B,i}$ contains (in increasing order) the indices of all columns j such that $B[i,j] = \mathtt{1}$.
3. For each $i \in [1\,..\,n]$, we define the string $T_{A,B,i}$ to be the concatenation of the strings $\mathrm{RowOnes}_{B,x}$ over all indices x appearing in $\mathrm{RowOnes}_{A,i}$, i.e.,

$$T_{A,B,i} := \bigodot_{t=1,\ldots,k} \mathrm{RowOnes}_{B,R[t]},$$

   where $R = \mathrm{RowOnes}_{A,i}$ and $k = |R|$.
4. Lastly, we define $T_{A,B} := \bigodot_{i=1,\ldots,n} T_{A,B,i}$.

We show that there is a direct correspondence between symbols appearing in $T_{A,B,i}$ and the positions of $\mathtt{1}$-entries in row i of the Boolean matrix product AB. Formally, for every $i,j \in [1\,..\,n]$, the symbol j appears in the string $T_{A,B,i}$ if and only if $(AB)[i,j] = \mathtt{1}$ holds. We illustrate this correspondence in Figure 1; for example, $(AB)[1,2] = (AB)[1,3] = \mathtt{1}$ corresponds to the symbols 2 and 3 occurring in $T_{A,B,1}$, and $(AB)[2,1] = \mathtt{1}$ corresponds to 1 occurring in $T_{A,B,2}$.

After concatenating the strings $T_{A,B,i}$ into $T_{A,B}$, we can compute the entire product AB using n^2 symbol occurrence queries, one for each pair (i,j), by querying the substring corresponding to $T_{A,B,i}$. We also construct an SLG of size $\mathcal{O}(n^2)$ that generates $T_{A,B}$. Thus, any algorithm that answers a batch of symbol occurrence queries (and consequently also rank queries; see Definition 15) on SLG-compressed texts in amortized polylogarithmic time per query yields a near-quadratic-time algorithm for Boolean matrix multiplication (see Theorem 1).

Hardness for Symbol Occurrence Queries on LZ78-Compressed Text. We use a new variant of the *grammar boosting* technique of [18] to transform the structured BMM instance (A,B) underlying the answer string $T_{A,B}$ into a new string $X_{A,B}$. The key idea is to add

prefix gadgets that force LZ78 to create, for every $i \in [1\,..\,n]$, the phrase $\$_i \cdot \mathrm{RowOnes}_{B,i}$ together with all of its prefixes. Once these phrases have been created, each later block of the form $\$_p \cdot \mathrm{RowOnes}_{B,p} \cdot \#_i$ is parsed as a single additional LZ78 phrase, because it extends an already existing phrase by one fresh delimiter. This yields an explicit description of the LZ78 representation of $X_{A,B}$ and allows us to compute it in $\mathcal{O}(n^2)$ time from A and B. Moreover, for every $i \in [1\,..\,n]$, the block $X_{A,B,i}$ preserves the occurrences of all symbols $j \in [1\,..\,n]$ from the corresponding block $T_{A,B,i}$. Hence, the symbol occurrence queries used in the BMM reduction can be simulated on $X_{A,B}$, and the same idea yields the corresponding hardness for rank queries (see Theorem 3).

Applying this technique to the example string $T_{A,B}$ from Figure 1 results in the following transformed string (with $\odot$ added for clarity):

$$\$_1\$_1 2 \$_2 \$_2 1 \$_3 \$_3 2 \$_3 23 \odot \$_1 2 \#_1 \$_3 23 \#_1 \odot \$_2 1 \#_2 \odot \$_1 2 \#_3 \$_2 1 \#_3$$

The LZ78 parsing of the above string is (with parentheses denoting each phrase):

$$(\$_1)(\$_1 2)(\$_2)(\$_2 1)(\$_3)(\$_3 2)(\$_3 23)(\$_1 2 \#_1)(\$_3 23 \#_1)(\$_2 1 \#_2)(\$_1 2 \#_3)(\$_2 1 \#_3)$$

Lastly, we highlight the portion of this transformed string that corresponds to the symbols of $T_{A,B}$ from Figure 1. All symbols appearing between the highlighted ones are auxiliary delimiters. This illustrates how symbol occurrence queries on the transformed string can simulate symbol occurrence queries on $T_{A,B}$ (and the same simulation applies to rank queries).

$$\$_1\$_1 2 \$_2 \$_2 1 \$_3 \$_3 2 \$_3 23 \ \$_1 2 \#_1 \ \$_3 23 \#_1 \ \$_2 1 \#_2 \ \$_1 2 \#_3 \ \$_2 1 \#_3$$

Hardness of Approximate Rank Queries. The hardness results for symbol occurrence queries imply that any multiplicative approximation for two-sided rank queries is also hard, since it would distinguish between the cases $\mathrm{rank}_T(b, e, c) = 0$ and $\mathrm{rank}_T(b, e, c) \geq 1$. We therefore consider additive approximations. We take any string $T \in \Sigma^N$ and replace every symbol with k copies of itself (call this transformed string $\mathrm{stretch}_k(T)$). After this transformation, a substring of T contains a symbol $c \in \Sigma$ if and only if the corresponding substring in $\mathrm{stretch}_k(T)$ contains at least k copies of c. If the original string T can be generated by a small grammar, then $\mathrm{stretch}_k(T)$ can also be generated by a small grammar, and such a grammar can be constructed efficiently. We thus obtain hardness results for additive approximations of two-sided rank queries over grammar-compressed texts (see Theorem 5).

Hardness for Range Distinct Count and Range Mode Frequency Queries. Let $A = (a_1, \ldots, a_n)$ be a sequence of $n \geq 1$ binary vectors of dimension $d \geq 1$. To show hardness of range distinct count queries (see Definition 19), we construct a string over alphabet $\{1, \ldots, n\}$ consisting of n *blocks* such that, for each i, the symbols missing from block i are exactly the indices of vectors orthogonal to a_i. We start by defining the following objects (see Figure 2 for an example showing all the objects defined below):

1. For each $j \in [1\,..\,d]$, we define $\mathrm{OneVectors}_{A,j}$ to be the string containing (in increasing order) the indices i of all vectors satisfying $a_i[j] = \mathtt{1}$.
2. For any vector $x \in \{\mathtt{0}, \mathtt{1}\}^d$, we define OnePos_x to be the string containing (in increasing order) the indices of all coordinates j satisfying $x[j] = \mathtt{1}$.
3. For every $i \in [1\,..\,n]$, we define the string $U_{A,i}$ as follows:

$$U_{A,i} := \bigodot_{j=1,\ldots,k} \mathrm{OneVectors}_{A,R[j]},$$

   where $R = \mathrm{OnePos}_{a_i}$ and $k = |R|$.
4. Lastly, we let $U_A := \bigodot_{i=1,\ldots,n} U_{A,i}$.

$a_1 = [1, 0, 0, 1]$ $\quad \text{OnePos}_{a_1} = 14$ $\quad \text{OneVectors}_{A,1} = 125$
$a_2 = [1, 1, 0, 0]$ $\quad \text{OnePos}_{a_2} = 12$ $\quad \text{OneVectors}_{A,2} = 23$
$a_3 = [0, 1, 0, 1]$ $\quad \text{OnePos}_{a_3} = 24$ $\quad \text{OneVectors}_{A,3} = 45$
$a_4 = [0, 0, 1, 1]$ $\quad \text{OnePos}_{a_4} = 34$ $\quad \text{OneVectors}_{A,4} = 134$
$a_5 = [1, 0, 1, 0]$ $\quad \text{OnePos}_{a_5} = 13$

$U_{A,1} = 125134$
$U_{A,2} = 12523$
$U_{A,3} = 23134$
$U_{A,4} = 45134$
$U_{A,5} = 12545$

$$U_A = 125134\ 12523\ \underbrace{23134}_{\text{Missing 5}}\ \overbrace{45134}^{\text{Missing 2}}\ 12545$$

Figure 2 Example showing our reduction from Orthogonal Vectors to range distinct count queries.

We prove that for every $i, j \in [1\,..\,n]$, the symbol j does *not* occur in $U_{A,i}$ if and only if $\langle a_i, a_j \rangle = 0$. Therefore, $\text{distinct}_{U_{A,i}}(0, |U_{A,i}|) < n$ holds if and only if a_i is orthogonal to some vector in A. Hence, after concatenating all blocks into U_A, we can use n queries $\text{distinct}_{U_A}(\cdot, \cdot)$ (Definition 19) to solve the orthogonal vectors problem on A.

We demonstrate this reduction in Figure 2. Here, the strings OnePos_{a_i} contain the indices of all coordinates j such that $a_i[j] = \texttt{1}$ for $i \in [1\,..\,5]$, and the strings $\text{OneVectors}_{A,j}$ contain the indices of all vectors a_i such that $a_i[j] = \texttt{1}$ for $j \in [1\,..\,4]$. The strings $U_{A,i}$ are defined as above. We observe that $U_{A,1}$ contains all elements from $\{1, \ldots, 5\}$, which matches the fact that a_1 is not orthogonal to any vector in A. The element 4 is missing from $U_{A,2}$ (and symmetrically, 2 is missing from $U_{A,4}$). This corresponds to the fact that $\langle a_2, a_4 \rangle = 0$. Similarly, the element 5 is missing from $U_{A,3}$ (and 3 is missing from $U_{A,5}$), which corresponds to $\langle a_3, a_5 \rangle = 0$. Thus, in this example, the only pairs of orthogonal vectors are (a_2, a_4) and (a_3, a_5).

The string U_A can be generated by a grammar G of size $\mathcal{O}(nd)$ that can also be constructed in $\mathcal{O}(nd)$ time. Thus, if we can answer a batch of n queries $\text{distinct}_{U_A}(\cdot, \cdot)$ with average time $\mathcal{O}(|G|^{1-\epsilon} \log^{\mathcal{O}(1)} N)$ per query (where N is the length of the input text), then we can solve the Orthogonal Vectors problem in sub-quadratic time (see Theorem 7).

The hardness for range mode frequency queries (see Definition 21) follows an analogous structure. We replace each string $\text{OneVectors}_{A,j}$ with its complement: we let $\text{ZeroVectors}_{A,j}$ be the string containing (in increasing order) the indices of all vectors a_i such that $a_i[j] = \texttt{0}$. We then define the corresponding blocks $W_{A,i}$. We show that symbol j appears in $W_{A,i}$ less than $|\text{OnePos}_{a_i}|$ times if and only if $\langle a_i, a_j \rangle \neq 0$. Thus, the maximum frequency in $W_{A,i}$ is at least $|\text{OnePos}_{a_i}|$ if and only if a_i is orthogonal to some vector in A. Proceeding as above, we obtain hardness of answering a batch of range mode frequency queries over substrings of grammar-compressed texts (see Theorem 9).

References

1 Josh Alman, Ran Duan, Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. More asymmetry yields faster matrix multiplication. In Yossi Azar and Debmalya Panigrahi, editors, *Proceedings of the 2025 ACM-SIAM Symposium on Discrete Algorithms (SODA 2025)*, pages 2005–2039. SIAM, 2025. doi:10.1137/1.9781611978322.63.

2 Djamal Belazzougui, Manuel Cáceres, Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Gonzalo Navarro, Alberto Ordóñez Pereira, Simon J. Puglisi, and Yasuo Tabei. Block trees. *Journal of Computer and System Sciences*, 117:1–22, 2021. doi:10.1016/j.jcss.2020.11.002.

3 Djamal Belazzougui, Patrick Hagge Cording, Simon J. Puglisi, and Yasuo Tabei. Access, rank, and select in grammar-compressed strings. In Nikhil Bansal and Irene Finocchi, editors, *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA 2015)*, pages 142–154. Springer, 2015. doi:10.1007/978-3-662-48350-3_13.

4 Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10(4):23:1–23:19, 2014. doi:10.1145/2635816.

5 Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms*, 11(4):31:1–31:21, 2015. doi:10.1145/2629339.

6 Oren Ben-Kiki, Philip Bille, Dany Breslauer, Leszek Gąsieniec, Roberto Grossi, and Oren Weimann. Towards optimal packed string matching. *Theoretical Computer Science*, 525:111–129, 2014. doi:10.1016/j.tcs.2013.06.013.

7 Philip Bille, Inge Li Gørtz, and Frederik Rye Skjoldjensen. Deterministic indexing for packed strings. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, pages 6:1–6:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.6.

8 Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015. doi:10.1137/130936889.

9 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994. URL: https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf.

10 Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005. doi:10.1109/TIT.2005.850116.

11 Anders Roy Christiansen, Mikko Berggren Ettienne, Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Optimal-time dictionary-compressed indexes. *ACM Transactions on Algorithms*, 17(1):8:1–8:39, 2021. doi:10.1145/3426473.

12 Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, 2011. doi:10.3233/FI-2011-565.

13 Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. In Liliana Calderón-Benavides, Cristina N. González-Caro, Edgar Chávez, and Nivio Ziviani, editors, *Proceedings of the 19th International Symposium on String Processing and Information Retrieval (SPIRE 2012)*, pages 180–192. Springer, 2012. doi:10.1007/978-3-642-34109-0_19.

14 Francisco Claude, Gonzalo Navarro, and Alejandro Pacheco. Grammar-compressed indexes with logarithmic search time. *Journal of Computer and System Sciences*, 118:53–74, 2021. doi:10.1016/j.jcss.2020.12.001.

15 Vincent Cohen-Addad, Laurent Feuilloley, and Tatiana Starikovskaya. Lower bounds for text indexing with mismatches and differences. In Timothy M. Chan, editor, *Proceedings of the 2019 ACM-SIAM Symposium on Discrete Algorithms (SODA 2019)*, pages 1146–1164. SIAM, 2019. doi:10.1137/1.9781611975482.70.

16 European Commission. 1+ Million Genomes Initiative. URL: https://digital-strategy.ec.europa.eu/en/policies/1-million-genomes.

17 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, Cambridge, UK, 2007. doi:10.1017/cbo9780511546853.

18 Rajat De and Dominik Kempa. Grammar boosting: A new technique for proving lower bounds for computation over compressed data. In David P. Woodruff, editor, *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms (SODA 2024)*, pages 3376–3392. SIAM, 2024. doi:10.1137/1.9781611977912.121.

19 Rajat De and Dominik Kempa. Optimal random access and conditional lower bounds for 2D compressed strings. In Kasper Green Larsen and Barna Saha, editors, *Proceedings of the 2026 ACM-SIAM Symposium on Discrete Algorithms (SODA 2026)*, pages 1903–1915. SIAM, 2026. doi:10.1137/1.9781611978971.69.

20 Akashnil Dutta, Reut Levi, Dana Ron, and Ronitt Rubinfeld. A simple online competitive adaptation of Lempel-Ziv compression with efficient random access support. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2013 Data Compression Conference (DCC 2013)*, pages 113–122. IEEE, 2013. doi:10.1109/DCC.2013.19.

21 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

22 José Fuentes-Sepúlveda, Juha Kärkkäinen, Dmitry Kosolobov, and Simon J. Puglisi. Run compressed rank/select for large alphabets. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2018 Data Compression Conference (DCC 2018)*, pages 315–324. IEEE, 2018. doi:10.1109/DCC.2018.00040.

23 Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. A faster grammar-based self-index. In Adrian-Horia Dediu and Carlos Martín-Vide, editors, *Proceedings of the 6th International Conference on Language and Automata Theory and Applications (LATA 2012)*, pages 240–251. Springer, 2012. doi:10.1007/978-3-642-28332-1_21.

24 Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. LZ77-based self-indexing with faster pattern matching. In Alberto Pardo and Alfredo Viola, editors, *Proceedings of the 11th Latin American Symposium on Theoretical Informatics (LATIN 2014)*, pages 731–742. Springer, 2014. doi:10.1007/978-3-642-54423-1_63.

25 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. On the approximation ratio of Lempel-Ziv parsing. In Michael A. Bender, Martin Farach-Colton, and Miguel A. Mosteiro, editors, *Proceedings of the 13th Latin American Symposium on Theoretical Informatics (LATIN 2018)*, pages 490–503. Springer, 2018. doi:10.1007/978-3-319-77404-6_36.

26 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67(1):1–54, 2020. doi:10.1145/3375890.

27 Moses Ganardi, Artur Jez, and Markus Lohrey. Balancing straight-line programs. *Journal of the ACM*, 68(4):27:1–27:40, 2021. doi:10.1145/3457389.

28 Pawel Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Lacki, and Piotr Sankowski. Optimal dynamic strings. In Artur Czumaj, editor, *Proceedings of the 2018 ACM-SIAM Symposium on Discrete Algorithms (SODA 2018)*, pages 1509–1528. SIAM, 2018. doi:10.1137/1.9781611975031.99.

29 Genomics England. The 100,000 Genomes Project. https://www.genomicsengland.co.uk/about-genomics-england/the-100000-genomes-project/.

30 Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: A tool for text indexing. In *Proceedings of the 2006 ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*, pages 368–373. ACM Press, 2006. URL: http://dl.acm.org/citation.cfm?id=1109557.1109599.

31 Rodrigo González and Gonzalo Navarro. Rank/select on dynamic compressed sequences and applications. *Theoretical Computer Science*, 410(43):4414–4422, 2009. doi:10.1016/j.tcs.2009.07.022.

32 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 2003 ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 841–850. ACM/SIAM, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

33 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 2003 ACM-SIAM Symposium on Discrete Algorithms, SODA 2003*, pages 841–850, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

34 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC 2000)*, pages 397–406. ACM, 2000. doi:10.1145/335305.335351.

35 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.

36 Torben Hagerup. Sorting and searching on the word RAM. In Michel Morvan, Christoph Meinel, and Daniel Krob, editors, *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1998)*, volume 1373 of *LNCS*, pages 366–398. Springer, 1998. doi:10.1007/BFb0028575.

37 Tomohiro I. Longest common extensions with recompression. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, pages 18:1–18:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.18.

38 Artur Jeż. A really simple approximation of smallest grammar. *Theoretical Computer Science*, 616:141–150, 2016. doi:10.1016/j.tcs.2015.12.032.

39 Haim Kaplan, Natan Rubin, Micha Sharir, and Elad Verbin. Counting colors in boxes. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *Proceedings of the 2007 ACM-SIAM Symposium on Discrete Algorithms (SODA 2007)*, pages 785–794. SIAM, 2007. URL: http://dl.acm.org/citation.cfm?id=1283383.1283467.

40 Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM 2001)*, pages 181–192, 2001. doi:10.1007/3-540-48194-X_17.

41 Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: Sublinear-time BWT construction and optimal LCE data structure. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC 2019)*, pages 756–767. ACM, 2019. doi:10.1145/3313276.3316368.

42 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler transform conjecture. In Sandy Irani, editor, *Proceedings of the 61st Annual IEEE Symposium on Foundations of Computer Science (FOCS 2020)*, pages 1002–1013. IEEE, 2020. doi:10.1109/FOCS46700.2020.00097.

43 Dominik Kempa and Tomasz Kociumaka. Collapsing the hierarchy of compressed data structures: Suffix arrays in optimal compressed space. In *Proceedings of the 64th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2023)*, pages 1877–1886. IEEE, 2023. doi:10.1109/FOCS57990.2023.00114.

44 Dominik Kempa and Tomasz Kociumaka. Tight lower bounds for central string queries in compressed space. In Kasper Green Larsen and Barna Saha, editors, *Proceedings of the 2026 ACM-SIAM Symposium on Discrete Algorithms (SODA 2026)*, pages 1824–1840. SIAM, 2026. doi:10.1137/1.9781611978971.65.

45 Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: String attractors. In Ilias Diakonikolas, David Kempe, and Monika Henzinger, editors, *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC 2018)*, pages 827–840. ACM, 2018. doi:10.1145/3188745.3188814.

46 Dominik Kempa and Barna Saha. An upper bound and linear-space queries on the LZ-End parsing. In Joseph (Seffi) Naor and Niv Buchbinder, editors, *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms (SODA 2022)*, pages 2847–2866. SIAM, 2022. doi:10.1137/1.9781611977073.111.

47 Tomasz Kociumaka, Gonzalo Navarro, and Francisco Olivares. Near-optimal search time in δ-optimal space. In Armando Castañeda and Francisco Rodríguez-Henríquez, editors, *Proceedings of the 15th Latin American Symposium on Theoretical Informatics (LATIN 2022)*, pages 88–103. Springer, 2022. doi:10.1007/978-3-031-20624-5_6.

48 Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Toward a definitive compressibility measure for repetitive sequences. *IEEE Transactions on Information Theory*, 69(4):2074–2092, 2023. doi:10.1109/TIT.2022.3224382.

49 Veli Mäkinen. Compact suffix array: a space-efficient full-text index. *Fundamenta Informaticae*, 56(1,2):191–210, October 2002. URL: `http://content.iospress.com/articles/fundamenta-informaticae/fi56-1-2-12`.

50 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Bioinformatics in the Era of High-Throughput Sequencing (2nd edition)*. Cambridge University Press, 2023. URL: `http://www.genome-scale.info/`.

51 Veli Mäkinen and Gonzalo Navarro. Position-restricted substring searching. In José R. Correa, Alejandro Hevia, and Marcos A. Kiwi, editors, *Proceedings of the 7th Latin American Symposium on Theoretical Informatics (LATIN 2006)*, pages 703–714. Springer, 2006. `doi:10.1007/11682462_64`.

52 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. `doi:10.1137/0222058`.

53 J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Text indexing and searching in sublinear time. In *Proceedings of the 31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*, pages 24:1–24:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.CPM.2020.24`.

54 Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014. `doi:10.1016/j.jda.2013.07.004`.

55 Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, Cambridge, UK, 2016. `doi:10.1017/cbo9781316588284`.

56 Gonzalo Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Computing Surveys*, 54(2):29:1–29:31, 2021. `doi:10.1145/3434399`.

57 Gonzalo Navarro. Indexing highly repetitive string collections, part II: Compressed indexes. *ACM Computing Surveys*, 54(2):26:1–26:32, 2021. `doi:10.1145/3432999`.

58 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. In Piotr Faliszewski, Anca Muscholl, and Rolf Niedermeier, editors, *Proceedings of the 41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, pages 72:1–72:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPIcs.MFCS.2016.72`.

59 Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013. URL: `http://www.oldenbusch-verlag.de/`.

60 Nicola Prezza. Optimal rank and select queries on dictionary-compressed text. In Nadia Pisanti and Solon P. Pissis, editors, *Proceedings of the 30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*, pages 4:1–4:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.CPM.2019.4`.

61 Wojciech Rytter. Application of Lempel–Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1–3):211–222, 2003. `doi:10.1016/S0304-3975(02)00777-6`.

62 Elad Verbin and Wei Yu. Data structure lower bounds on random access to grammar-compressed strings. In Johannes Fischer and Peter Sanders, editors, *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM 2013)*, pages 247–258. Springer, 2013. `doi:10.1007/978-3-642-38905-4_24`.

63 Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT/FOCS 1973)*, pages 1–11. IEEE Computer Society, 1973. `doi:10.1109/SWAT.1973.13`.

64 Ryan Williams. The orthogonal vectors conjecture and non-uniform circuit lower bounds. In *Proceedings of the 65th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2024)*, pages 1372–1387. IEEE, 2024. `doi:10.1109/FOCS61266.2024.00088`.

65 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. `doi:10.1109/TIT.1977.1055714`.

66 Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978. `doi:10.1109/TIT.1978.1055934`.