Space-Efficient Text Indexing with Mismatches using Function Inversion

Jackson Bibbens
jbibbens@umass.edu
University of Massachusetts
Amherst, MA, USA

Levi Borevitz
leviborevitz2023@u.northwestern.edu
Northwestern University
Evanston, IL, USA

Samuel McCauley
sam@cs.williams.edu
Williams College
Williamstown, MA, USA

Abstract

A classic data structure problem is to preprocess a string T of length n so that, given a query q, we can quickly find all substrings of T with Hamming distance at most k from the query string. Variants of this problem have seen significant research both in theory and in practice. For a wide parameter range, the best worst-case bounds are achieved by the "CGL tree" (Cole, Gottlieb, Lewenstein 2004), which achieves query time roughly $\widetilde{O}(|q| + \log^k n + \#occ)$, where $\#occ$ is the size of the output, and space $O(n \log^k n)$. The CGL Tree space was recently improved to $O(n \log^{k-1} n)$ (Kociumaka, Radoszewski 2026).

A natural question that arises is whether a high space bound is necessary. How efficient can we make queries when the data structure is constrained to $O(n)$ space? While this question has seen extensive research, all known results have query time with unfavorable dependence on the alphabet size, n and k. The state of the art query time from (Chan, Lam, Sung, Tam, Wong 2011) is roughly $\widetilde{O}(|q| + |\Sigma|^k \log^{k^2+k} n + \#occ)$ for alphabet Σ.

We give an $O(n)$-space data structure with query time roughly $\widetilde{O}(|q| + \log^{4k} n + \log^{2k} n \cdot \#occ)$, with no dependence on the size of the alphabet. Even for a constant-sized alphabet, this is the best known query time for linear space if $k \geq 3$ unless $\#occ$ is large. Our results give a smooth tradeoff between time and space. Interestingly, our results are the first to extend to the sublinear space regime: we give a succinct data structure using only $o(n)$ space in addition to the text itself, with only a modest increase in query time.

The main technical idea behind this result is to apply Fiat-Naor function inversion (Fiat, Naor 2000) to the CGL tree. Combining these techniques is not immediate; in fact, we revisit the exposition of both the Fiat-Naor data structure and the CGL tree to obtain our bounds. Along the way, we obtain improved performance for both data structures, which may be of independent interest.

CCS Concepts

• **Theory of computation** → **Pattern matching**; **Sorting and searching**.

Keywords

Data structures, Text Indexing, Pattern Matching, String algorithms

STOC '26, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-2536-4/2026/06
https://doi.org/10.1145/3798129.3800818

ACM Reference Format:
Jackson Bibbens, Levi Borevitz, and Samuel McCauley. 2026. Space-Efficient Text Indexing with Mismatches using Function Inversion. In *Proceedings of the 58th Annual ACM Symposium on Theory of Computing (STOC '26), June 22–26, 2026, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3798129.3800818

1 Introduction

In many string processing applications, it is crucial to be able to handle errors. After all, real-world methodologies for obtaining and querying data are subject to mistakes and to noise. With this motivation in mind, a classic string processing problem is finding substrings of a text that nearly match—but may not exactly match—a query string.

One common setting is building an indexing data structure. We begin with a preprocessing step, in which we build an index on a text T. After the preprocessing is complete, there is a sequence of queries q. All characters in the text and any query are from an alphabet Σ. The goal is to use the index to quickly find substrings of T that are approximately equal to q. In this work, we use Hamming distance as our notion of approximate matching: we want to find all substrings of T that differ from q in at most r locations for a specified parameter r. Throughout this paper, we will assume that during preprocessing, we are given a bound k on the maximum r of any query.

This is a classic problem, first introduced by Minsky and Papert in 1969 as the "best match" or "nearest neighbors" problem [22]. In their original exposition, the problem is defined slightly differently: the input is a dictionary of strings rather than a single large text, and the goal is to find if any string in the dictionary is approximately equal to q; we discuss this further and explain why both versions are equivalent in Section 2.2. We follow recent work (i.e. [9, 10]) in calling this the Text Indexing with Mismatches problem.

Past work has found a number of efficient data structures for this problem. A particularly performant data structure is the index of Cole, Gottlieb, and Lewenstein [10], which we refer to throughout this paper as the "CGL tree." If T has size n, and the maximum search radius is k, the CGL tree uses $O(n3^k\binom{\log n+k}{k})$ space and can answer queries in[1] $\widetilde{O}(|q|+6^k\binom{\log n+k}{k}+\#occ)$ time, where $\#occ$ is the number of substrings of T that match q. Perhaps most notably, these bounds exhibit optimal linear dependence on $|q|$ and $\#occ$, with no dependence on the size of the alphabet. Furthermore, the search time is polylogarithmic for $k = O(1)$, since $\binom{\log n+k}{k} = O(\log^k n)$.

[1] We use $\widetilde{O}$ notation to suppress log factors: $\widetilde{O}(f(n))$ is the same as $O(f(n)\text{polylog} f(n))$. Note that the log factors are in terms of the argument: for example, $\widetilde{O}(\log^k n)$ suppresses k and $\log\log n$ terms.

Table 1: Space-efficient data structures for Text Indexing with Mismatches. In this table, we simplify $\binom{x}{y} \approx x^y$; this replacement does not affect the asymptotic bounds when $k = O(1)$.

Space	Query Time	Reference
$O(n)$ words	$O(\lvert q\rvert^{k+1}\lvert\Sigma\rvert^k + \#occ)$	[8]
	$\widetilde{O}(\lvert q\rvert^k\lvert\Sigma\rvert^k \log n + \#occ)$	[15]
	$\widetilde{O}(\lvert q\rvert^k\lvert\Sigma\rvert^k \log\log n + \#occ)$	[20]
	$\widetilde{O}(\lvert q\rvert + \lvert\Sigma\rvert^k k^k 3^{k^2} \log^{k^2+k} n + \#occ)$	[7]
	$\widetilde{O}(\lvert q\rvert + 3^k \log^{4k} n + \log^{2k} n \cdot \#occ)$	Corollary 2
$O(\frac{n}{\sigma}\log^k n)$ words; $\sigma \le \log^k n$	$\widetilde{O}(\lvert q\rvert + \lvert\Sigma\rvert^k k^k 3^{k^2} \sigma^k \log^k n + \#occ)$	[7]
	$\widetilde{O}(\lvert q\rvert + 3^k \sigma^3 \log^k n + \sigma^2 \cdot \#occ)$	Theorem 1
$O(n \log\lvert\Sigma\rvert)$ bits	$\widetilde{O}(\lvert q\rvert^k\lvert\Sigma\rvert^k \log^2 n + \log n \cdot \#occ)$	[15]
	$\widetilde{O}(\lvert q\rvert^k\lvert\Sigma\rvert^k \log^\varepsilon n + \log^\varepsilon n \cdot \#occ)$	[20]
	$\widetilde{O}(\lvert q\rvert \log^\varepsilon n + \lvert\Sigma\rvert^k k^k 3^{k^2} \log^{k^2+k+\varepsilon} n + \log^\varepsilon n \cdot \#occ)$	[7]
$O(\frac{n}{\sigma}\log^k n)$ words; $\sigma \ge \log^k n$	$\widetilde{O}(\lvert q\rvert + 3^k \sigma^4 \log n + \sigma^3 \log^{1-k} n \cdot \#occ)$	Theorem 4

The drawback of the Cole, Gottlieb, and Lewenstein result is the prohibitive space usage: the CGL tree requires $\Theta(n3^k\binom{\log n+k}{k})$ words of space. There has been a long line of work on tradeoffs between space usage and query time [7, 8, 15, 20]. In particular, Chan et al. [7] obtained an $O(n)$-sized index with query time $\widetilde{O}(|q|+|\Sigma|^k(3\log n)^{k(k+1)} + \#occ)$.

Recently, Kociumaka and Radoszewski [17] were the first to improve the space of the CGL tree while maintaining essentially the same query time. For large alphabets, their data structure requires $O(n\log^{k-1} n)$ space; roughly $\log n$ better than the CGL tree. For small alphabets, they are able to save an additional log, achieving $O(n\log^{k-2+\varepsilon} n)$ space for any $\varepsilon > 0$ for sufficiently large k. Their parameter setting is significantly different from ours: they improve the CGL tree space as much as possible while maintaining the same query time, whereas we increase the query time noticeably in exchange for drastic space improvements.

The techniques used to obtain Kociumaka and Radoszewski's bounds in [17] differ substantially from the techniques in this paper. Our strategy is to augment a truncated version of the CGL tree with a function-inversion data structure. Their approach leverages the underlying problem structure more directly. They first give a data structure whose query time depends on the length of a query; these bounds are useful if the length of a query is short. For long queries, they carefully select "anchors" of the text to cover all matches; this improves space if the query is sufficiently long. It is possible that their results could be augmented with a function-inversion data structure as described in this work, improving our tradeoffs by roughly a $\log n$ factor space. We leave this to future work.

There has also been work on lower bounds for this problem. Cohen-Addad, Feuilloley, and Starikovskaya [9] showed that in the pointer machine model, any data structure with $O(|q| + (\frac{\log n}{2k})^k + \#occ)$ query time must use $\Omega(c^k n)$ space for some constant c.

Goal of This Work. In the following discussion, we assume $k = O(1)$ to allow us to substitute $c^{\text{poly}(k)}\binom{\log n+k}{k} = \Theta(\log^k n)$.

A significant gap remains in the known time-space tradeoffs for this problem. The best upper bound on the query time for a $\Theta(n)$ space data structure is $\widetilde{O}(|\Sigma|^k \log^{k^2+k} n + |q| + \#occ)$, whereas the best lower bound is $\Omega(\log^k n + |q| + \#occ)$. Furthermore, known space-efficient upper bounds all have a $|\Sigma|^k$ term in the query time (see Table 1), whereas the query time of the CGL tree has no dependence on the alphabet size.

In this paper, we make significant progress towards closing this gap, obtaining a $\widetilde{O}(\log^{4k} n + |q| + \log^{2k} n \cdot \#occ)$ query time with $O(n)$ space. This is the first linear-space result with a query time that is a polynomial factor larger than the lower bound, and the first without a dependency on the size of the alphabet.

Our time-space tradeoff also extends to a new parameter setting: a succinct data structure using *sublinear* extra space. Past work has used either $\Theta(n)$ words of space or $\Theta(n)$ bits of space [7]. We show that it is possible to achieve $o(n)$ bits of space in addition to the text T while still retaining competitive query times. This space savings is particularly well-motivated in the case that T can be significantly compressed, leading to sublinear overall space usage. At the extreme end of the time-space tradeoff, we maintain $o(n)$ query times while only storing $\widetilde{O}(n^{3/4}\log^{3k} n)$ words of metadata in addition to the text T.

Function Inversion for Data Structures. A classic problem in cryptography is function inversion: given a function $f : \{1,\dots,N\} \to \{1,\dots,N\}$, create a space-efficient data structure that allows us to quickly evaluate its inverse f^{-1}—that is to say, for some j, find an i such that $f(i) = j$. A space-inefficient solution is easy: we can store a lookup table for all possible queries in $\Theta(N)$ space; the goal is to obtain an $O(N/\sigma)$-space data structure for some $\sigma = \omega(1)$. The best-known bounds for this classic problem are an $\widetilde{O}(N/\sigma)$-space data structure that can evaluate f^{-1} in $\widetilde{O}(\sigma^3)$ time [12]. If f is random, or if there are few pairs of elements i, j with $f(i) = f(j)$, then the evaluation time can be improved to $\widetilde{O}(\sigma^2)$ time [12, 14].

A recent line of work has shown that function inversion can help improve time-space tradeoffs for data structures. The classic

example is 3SUM indexing [13, 16, 18]. Further work has included function inversion results for other data structure problems including collinearity testing [4], string indexing [6, 11], and similarity search [21].

The main idea behind this paper is to use function inversion to give a space-time tradeoff for Text Indexing with Mismatches. One interesting way this differs from past work is that we "open the black box" on both sides. We must significantly alter both the CGL tree, and Fiat-Naor's function inversion result, in order to obtain our bounds. Our exposition obtains improved bounds for each problem individually, and may be of independent interest.

1.1 Results

Our main result is the following time-space tradeoff.

Theorem 1. *For any σ satisfying $1 \le \sigma \le \binom{\log n}{k}$, there exists a Text Indexing with Mismatches data structure with $O(n\binom{\log n}{k}/\sigma)$ space that can be constructed in $O(nk^2\binom{\log n}{k}(\log n + k^2(\log\log n)^2))$ expected time and can answer queries in time*

$$\widetilde{O}\left(|q| + 3^k\sigma^3\binom{\log n}{k} + \sigma^2 \cdot (\#occ)\right).$$

This is a Las Vegas data structure: it is randomized, but the queries are always correct, and the space and query time bounds are worst-case. The randomization only affects preprocessing time.

We can immediately achieve linear space by setting $\sigma = \binom{\log n}{k}$.

Corollary 2. *There exists a Text Indexing with Mismatches data structure with $O(n)$ space that can be constructed in expected time $O(nk^2\binom{\log n}{k}(\log n + k^2(\log\log n)^2))$ and can answer queries in time*

$$\widetilde{O}\left(|q| + 3^k\binom{\log n}{k}^4 + \binom{\log n}{k}^2 \cdot (\#occ)\right).$$

Our data structure uses a new exposition of the CGL tree. This exposition slightly improves the performance of the CGL tree: we improve the space from $O(3^k n\binom{\log n+k}{k})$ to $O(nk\binom{\log n}{k})$, and the query time from $\widetilde{O}(|q|+6^k\binom{\log n+k}{k}+\#occ)$ to $\widetilde{O}(|q|+3^k\binom{\log n}{k}+\#occ)$. Specifically, we improve the base of the exponent (removing it entirely from the space term), which improves the asymptotics for $k = \omega(1)$, and we remove the additive k term in the binomial, which improves the asymptotics for $k = \omega(\sqrt{\log n})$.

Theorem 3. *There exists a Text Indexing with Mismatches data structure with $O(nk\binom{\log n}{k})$ space that can be constructed in expected time $O(nk^2\binom{\log n}{k}\log n)$ and can answer queries in time*

$$\widetilde{O}\left(|q| + 3^k\binom{\log n}{k} + \#occ\right).$$

We also extend our results to give the first succinct data structure for Text Indexing with Mismatches. This data structure has sublinear space in addition to the space required to store T. This is particularly motivated in contexts when T is compressed—our data structure can be built to match even a significantly sublinear-sized T. Our succinct results are Monte Carlo, and queries are correct with high probability—that is to say, with probability $\ge 1 - 1/n^c$ for any constant c.

Theorem 4. *For any $\sigma \ge \binom{\log n}{k}$, there exists a Text Indexing with Mismatches data structure that uses $O(n\binom{\log n}{k}/\sigma)$ space in addition to the text T, that can be constructed in $O(nk^2\binom{\log n}{k} + nk\sigma\log\sigma)$ expected time and can answer queries correctly with high probability in time*

$$\widetilde{O}\left(|q| + 3^k\sigma^4\log n + \frac{\sigma^3}{\binom{\log n}{k}}\log n \cdot (\#occ)\right).$$

Finally, we obtain slightly improved function inversion bounds. For our results (and for many data structure applications), it is crucial that we obtain *all* elements in the inverse $f^{-1}(j)$, whereas classic function inversion results [12, 14] give a single element of $f^{-1}(j)$. We show that if the elements we are inverting[2] have $|f^{-1}(j)| \le \sigma$, a few minor changes to the original Fiat-Naor data structure are sufficient to find all elements in the inverse with probability 1. Furthermore, we can avoid all $\log n$ terms in the space and running time bound, using only $\log\sigma$ terms instead.

Theorem 5. *For any $f : [n] \to [n-1] \cup \perp$, such that: (1) $f(i)$ can be evaluated in $E(f)$ time for all i, and (2) $|f^{-1}(j)| \le \sigma$ for all $j \ne \perp$; there exists a data structure using $O(n/\sigma)$ space such that for any $j \ne \perp$, we can find $f^{-1}(j)$ in $O(\sigma^3 \cdot (E(f) + \log\sigma) \cdot \log^4\sigma)$ time. The data structure requires $O(n \cdot (E(f) + \log\sigma) \cdot \log\sigma)$ expected preprocessing time.*

Past work on data structures for function inversion [3, 21] has found all elements in the preimage with sampling- or binary-search-based techniques on a black-box Fiat-Naor data structure, leading to (ignoring extra $\log n$ terms in [12]) a query time of $O(E(f)\cdot\sigma^4\log n)$.

A Note on Log Factors. We point out that the $\widetilde{O}$ notation in our bounds does not suppress $\log n$ factors. Specifically, in this paper we use $\widetilde{O}(f(n))$ as a shorthand for $O(f(n)\cdot\mathrm{polylog}(f(n)))$. Substituting the query time from Theorems 1, 3, or 4 for $f(n)$, we are suppressing terms polynomial in k, $\log|q|$, $\log\sigma$, or $\log\log n$. These terms are logarithmic compared to the query time, but $\log n$ terms are not—we need to track any $\log n$ terms that appear in the analysis, and design our data structures to avoid stray $\log n$ terms.

Comparing Our Results to Previous Space-Efficient Work. We give a full comparison of previous space-efficient results in Table 1. We give the first polynomial tradeoff between time and space: a space savings of σ results in a σ^3 increase in query time, compared to the previous state of the art σ^k. We are also the first space-efficient results without a $|\Sigma|^k$ term—in fact, our results have no dependence on Σ so long as each character fits in a machine word. We point out that $4k \le k^2 + k$ for $k \ge 3$, so the logarithmic term in the query time alone is also improved for most values of k.

The one area where we do not improve on past work is that the final term in our query time is $\sigma^2\cdot\#occ$, whereas past work generally has an additive $\#occ$ or $\#occ\cdot\log n$ term. This seems to be a downside of our approach: function inversion has, inherently, a high cost per element inverted (and therefore a high cost per element returned); see also [6]. That said, our gains elsewhere are significant enough that we still achieve improved performance unless the output size

[2]The theorem as stated assumes one high-indegree element; to generalize, we can store all elements with indegree $> \sigma$ in $O(n/\sigma)$ space and map them to $\perp$ manually.

is large. In particular, we give the best known query time for a linear-space data structure if $\#occ \leq |\Sigma|^k k^k 3^{k^2} \log^{k^2-k} n$.

We are the first to give $o(n)$-space results, significantly generalizing past research attaining $O(n)$ bits of space. To our knowledge, a succinct data structure was not known even for the $k = 1$ case.

1.2 Technical Overview

The CGL Tree and Function Inversion. The main idea of our results is to use function inversion to reduce the space necessary to store the CGL tree. The CGL tree, in short, is a combination of a trie and a binary search tree. Each node in the CGL tree has a suffix of T as a "pivot". As in a binary search tree, the query algorithm guarantees that if a substring s of T has distance at most k from q, then s is a prefix of a pivot of a node traversed by q.

The CGL tree, which we refer to as $\mathcal{T}_0$, has $O(n\binom{\log n}{k})$ leaves. We truncate the CGL tree to obtain a subtree with $O(n\binom{\log n}{k}/\sigma)$ leaves, each of which is the root of a subtree of size $O(\sigma)$ in the original tree. We denote the truncated tree as $\mathcal{T}$.

Consider the following strategy that is space- and time-inefficient, but maintains correctness: for each leaf of the truncated tree $\mathcal{T}$ we can keep a list of its $O(\sigma)$ descendants in $\mathcal{T}_0$. On a query, we begin by traversing $\mathcal{T}$ exactly as we would have traversed $\mathcal{T}_0$. When we reach the leaf, we can compare the query to all σ descendants.

To use function inversion, consider applying a *label* to each leaf of the truncated tree $\mathcal{T}$. We define a set of functions $f_1, \ldots, f_{\binom{\log n}{k}}$ such that if the ith suffix of T is a descendant of the leaf with label ℓ, there exists a λ such that $f_\lambda(i) = \ell$.

With this setup, we no longer need to store a list at each leaf of $\mathcal{T}$. Instead, we store a function inversion data structure on each f_λ. When the query reaches a leaf ℓ, rather than scanning through a list, we can calculate $f_\lambda^{-1}(\ell)$. This gives exactly the set of strings that were stored in the list.

Performing a function inversion query requires $O(\sigma^3)$ time using the strategy of Fiat and Naor [12]. This leads to query time roughly $O(\sigma^3\binom{\log n}{k})$. The truncated tree $\mathcal{T}$ and the function inversion data structure each require $O(n/\sigma)$ space.

Challenges to This Strategy. The basic strategy above is to combine [10] and [12], but their results do not work together immediately. We need to carefully address the following crucial details.

- We must find a truncated tree $\mathcal{T}$ with $O(|\mathcal{T}_0|/\sigma)$ leaves, each with $O(\sigma)$ descendants.
- We need to define the sequence of functions $f_1, \ldots, f_{\binom{\log n}{k}}$.
- We need to ensure that we can use the function-inversion data structure to recover *all* points stored in a given leaf, with probability 1 (Fiat-Naor inversion only returns one).
- We need a way to efficiently find the distance between the query q and a substring of T.
- We must avoid losing stray $\log n$ or 2^k terms while addressing these challenges.

To address these challenges, we "open the black box" on both sides. We give a new exposition of the CGL tree, and an exposition of a simplified Fiat-Naor function inversion data structure.

Changes to the CGL Tree. In the original exposition of [10], the starting point is a compressed trie on T, with two crucial changes. First, a centroid path decomposition is calculated on the tree. For each node, a new child is created, consisting of the union of all non-centroid path children of the node. This union allows for more efficient searching when the query has errors at this point. Then, the children of each node, and the nodes along each path, are each merged into a BST-like data structure; the "group tree." The group tree ensures that the height of the tree is at most $\log n$. Stitching these data structures together gives their $O(c^k\binom{\log n+k}{k})$ query time.

Our exposition proceeds essentially in reverse. We begin with a BST-like structure: at each recursive node, we partition into 4 sets, each of at most half the size. Then, to handle errors, rather than grouping trie nodes, we recursively alter both the query q and the suffixes of T. Each suffix of T is altered exactly once. We then store 8 children of each node:[3] 4 for the sets of altered suffixes, and 4 for the sets of unaltered suffixes.

The advantage of this change is that we have one recursive structure, rather than a combination of suffix tree nodes and two types of group tree nodes. This gives us our query time improvements: in short, the original exposition incurred a constant-factor loss in performance at most k times along each root-to-leaf path when the different structures were combined, leading to an extra c^k term in time and space, and a $\binom{\log n+k}{k}$ term compared to our $\binom{\log n}{k}$.

The disadvantage of this change comes from difficulties in removing a final $\log n$ term. Our goal is $\widetilde{O}(\log^k n)$ performance: in particular, when the recursive search radius $r = 0$, we want performance $O(k \log\log n)$, so we do not have time to traverse a tree of height $\Omega(\log n)$. In [10] this is handled by combining a compressed trie on T with an efficient predecessor data structure. The challenge is that we have recursively changed the suffixes and the query—the suffix tree of T might not reflect these recursive alterations.

Therefore, we must generalize the data structure in [10] to handle alterations. Our data structure has the same basic ideas as theirs, but our simplicity in the recursive exposition comes at the cost of significantly increased bookkeeping here. This generalization is the topic of Section 5.

1.3 Related Work

Known Lower Bounds. Cohen-Addad et al. [9] showed that there is a constant $c > 1$ such that any pointer-machine data structure for Text Indexing with Mismatches with $O((\frac{\log n}{2k})^k + \#occ)$ query time must have space $\Omega(c^k n)$—in particular, such a query time cannot be obtained for linear space if $k = \omega(1)$. For sufficiently small $\#occ$, our results are the first to give $O(\log^{O(k)} n)$ query time with $O(n)$ space, within a polynomial factor of this lower bound.

However, these bounds are not entirely comparable to ours. First, our data structure is not in the pointer machine model, as function inversion uses a lookup table. Second, the lower bounds of [9] are based on [1], and heavily rely on the query time being linear in the output, e.g. they rely on query performance of the form $O(Q(n,k) + \#occ)$. The bounds in this paper, on the other hand, have a significant cost per string being output. This downside seems intrinsic to the function inversion technique, at least using known techniques—space-efficient function inversion seems to require $\omega(1)$ time per element returned (see also [6]).

[3] In fact we will store 7 sets, as one of the altered sets will be unnecessary.

Function Inversion. Function inversion was first studied by Hellman [14], who showed that a *uniform random* function $f : [N] \to [N]$ can be inverted in $\widetilde{O}(N/\sigma)$ space and σ^2 time. Fiat-Naor generalized this result to general f, in exchange for an increase in cost: any $f : [N] \to [N]$ can be inverted in $\widetilde{O}(N/\sigma)$ space and σ^3 time. Recently Golovnev et al. improved this result for large σ (along with simplifying the previous analysis), inverting f in $\widetilde{O}(N/\sigma)$ space and $O(\min\{\sigma^3, \sigma N^{1/2}\})$ time [13]. We note that this final result uses either shared randomness or a non-uniform algorithm; therefore, we do not use their result here.

Using Function Inversion for Data Structures. Function inversion was originally applied to achieve time-space tradeoffs for an indexing variant of the classic 3SUM problem [13, 18]. More recently, these ideas have been extended to the idea of collinearity testing [4]. Aronov et al. [3] give a toolbox for using function inversion for a variety of data structure primitives.

The work of Bille et al. [6] also applies function inversion to a string indexing problem. Specifically, they solve the *gapped string indexing problem*: given two strings P_1 and P_2, find instances of P_1 and P_2 in T with distance (in this context: the difference in the index where each starts) within a given range $[\alpha, \beta]$. In terms of techniques, their result differs significantly from ours: their main technical contribution is a data structure that allows a reduction to a sequence of 3SUM indexing queries, after which it is possible to use the results of [13] as a black box to solve their problem.

Corrigan-Gibbs and Kogan [11] apply function inversion to obtain new time-space tradeoffs for the "systematic substring-search problem", which is essentially a special case of Text Indexing with Mismatches with $k = 0$, $\Sigma = \{0, 1\}$ and $|q| \leq \log n$. Interestingly, they show that this problem is *equivalent* to function inversion: any improvement on one implies improvement for the other.

The work of McCauley [21] also applies function inversion to a nearest neighbor problem; specifically, high-dimensional approximate similarity search under the ℓ_2 metric. At a high level, the techniques of their work are similar: they trim a tree to use less space, and then use function inversion to recover the missing pieces.

1.4 Paper Outline

In Section 2 we give a detailed description of our model and notation. We also give a data structure to help us compare suffixes of T and q more quickly, even suffixes that we have altered during recursive calls. In Section 3 we give our exposition of the CGL tree. Then, in Section 4, we describe how to achieve the space-efficient results given in Section 1.1, including an exposition of how we can adjust the Fiat-Naor function inversion strategy to obtain Theorem 5. Finally, in Section 5, we show how to traverse our data structure quickly, avoiding an extra $\log n$ term in Theorems 1 and 3.

Most proofs have been omitted for space; they can be found in the full version of the paper.

2 Preliminaries

String Notation and Definitions. The strings in this paper are assumed to consist of characters from a set Σ, each of which fits in $\Theta(1)$ machine words. For a string s, and for any $i \in \{1, \ldots, |s|\}$, we use $s[i]$ to denote character i of s; we 1-index strings.

We define the *Hamming distance* of two strings s_1 and s_2 to be the number of indices $i \in \{1, \ldots, \min(|s_1|, |s_2|)\}$ such that $s_1[i] \neq s_2[i]$. We denote this as $d_H(s_1, s_2)$. Note that this is the classic definition of Hamming distance if $|s_1| = |s_2|$; if the sizes are different, any further characters in the longer string are ignored. This generalized notion of Hamming distance will make our exposition slightly easier. For example, we can rephrase Text Indexing with Mismatches as: find all suffixes of T with distance at most r from q.

We append to T $2k + 1$ copies of a special character \$ that does not appear in q or T. When we refer to "suffixes" of T, we only refer to the n suffixes that do not start with \$. With these special characters appended, no suffix of T is a prefix of another, even if k characters of each suffix are changed. This has an added benefit that with our definition of distance, all suffixes of T with length less than $|q|$ have distance greater than k from q.

The *longest common prefix* between strings s_1 and s_2, denoted $LCP(s_1, s_2)$, is the largest i such that for all $j \leq i$, $s_1[j] = s_2[j]$. If s_1 is a prefix of s_2 or vice versa then $LCP(s_1, s_2) = \min\{|s_1|, |s_2|\}$.

Altered Strings. Our data structure is recursive. While recursing, we will make character substitutions on both the query string and suffixes of T. We will keep track of the characters that have been changed using *alterations*, which we define here.

An *alteration* consists of an index and a character. A *k-altered string* consists of a string s and a sequence of *at most* k alterations a; we also call the string *altered* without specifying k. If s is the query q, we refer to (s, a) as an *altered query*. The resulting string can be obtained by starting with s, and, for each alteration in a consisting of an index i and character c, replacing $s[i]$ with c. We emphasize that k is an upper bound on the number of alterations: much of our data structure, built with maximum search radius k, consists of k-altered suffixes of T. This means that each suffix has at most k alterations, but could have fewer. We denote an altered string using the notation (s, a).

Other Notation and Definitions. For a function f, we use $f^{-1}(y)$ to denote the set of all x such that $f(x) = y$.

All logs in this paper are base 2. All bounds in this paper are in the word RAM model with words of size $\Omega(\log n)$, and all space bounds are in the number of required words.

We use $[x]$ to denote the set $\{1, \ldots, x\}$ for any x.

We use n to denote the length of T, k the maximum search radius of the data structure, and σ the space-saving parameter: we want $O(n\binom{\log n}{k}/\sigma)$ space. For Theorem 4, our goal is $o(n)$ space, i.e. $\sigma > \binom{\log n}{k}$. In this case, it is often useful to use a variable $\tau = \lceil \sigma / \binom{\log n}{k} \rceil$; that way, the space bound can be rewritten $O(n/\tau)$.

Our results assume that any character of T can be accessed in $O(1)$ time. If T is stored in such a way that a longer time is required (e.g., if it is compressed) the bounds still hold with proportionally increased preprocessing and query time.

2.1 First $k + 1$ Mismatches Data Structure

Our results rely on a crucial subroutine, the *first $k + 1$ mismatches*: given two strings $\widehat{s_1}$ and $\widehat{s_2}$, where $\widehat{s_2}$ is a substring of T with at most $k + 1$ alterations, and $\widehat{s_1}$ is either a substring of q with at most $k + 1$ alterations or a substring of T with at most $k + 1$ alterations, find the first $k + 1$ indices i where $\widehat{s_1}[i] \neq \widehat{s_2}[i]$. We fix $\widehat{s_1}$ and $\widehat{s_2}$

for the remainder of this section. Let $\widehat{s_1} = (s_1, a_1)$ and $\widehat{s_2} = (s_2, a_2)$, where s_1 is a substring of T or q, and s_2 is a substring of T.

This subroutine generalizes two queries we need for our data structure: (1) finding the first place where $\widehat{s_1}$ and $\widehat{s_2}$ differ (i.e., computing $LCP(\widehat{s_1}, \widehat{s_2})$), and (2) checking if $d_H(\widehat{s_1}, \widehat{s_2}) \leq k$.

Note that the strings are $(k+1)$-altered rather than k-altered; this is for Section 5. While Sections 3 and 4 only use k-altered strings, Section 5 uses $(k+1)$-altered strings.

Our strategy for this data structure is to use, as a black box, a known data structure to find the longest common prefix between two *unaltered* suffixes: either two suffixes of T, or a suffix of T and a suffix of q. We then recursively alter each string to compute the first $k+1$ mismatches between the altered strings. This strategy is similar to the strategy referred to as "kangaroo jumps" in the literature; see [23]. For $\tau = 1$ a similar result (using a similar strategy) was given independently in [17, Lemma 3.3].

Finding the Mismatches. We first find the LCP without taking alterations into account: that is to say, we find $LCP(s_1, s_2)$ (we will specify how to do this momentarily). If the LCP is before the first alteration in a_1 or a_2, we have found a place where $\widehat{s_1}$ and $\widehat{s_2}$ differ, and we append it to the list of mismatches. If it is at or after the first alteration to one of the strings, we can check if the strings differ at that location manually, appending to the list if so. Then, we recurse on the suffix of each string after the first alteration.

Since each string has $\leq k+1$ alterations, and we only list out $\leq k+1$ mismatches, we recurse at most $3k+3 = O(k)$ times.

Longest Common Prefix Data Structures. We use one of the following two data structures to find $LCP(s_1, s_2)$.

First, consider the case where the total space used is $\Omega(n)$, as in Theorem 1. This means we can use $O(n)$ space for the longest common prefix data structure. In this case, Cole, Gottlieb, and Lewenstein [10] show that it is possible to combine a suffix array on T with a lowest common ancestor data structure to allow longest common prefix queries in $O(1)$ time. They further point out that all suffixes of q can be added to the tree in $O(|q|)$ time.

Now, consider the case with $O(n/\tau)$ space for $\tau > 1$, as in Theorem 4. We split into two further cases.

To find the longest common prefix between suffixes of T and suffixes of q, we use the Monte-Carlo data structure of Bille et al. [5]. During preprocessing, we preprocess T using their data structure in $O(n)$ time. On each query q, we also preprocess q using their data structure in $O(|q|)$ time. In the full version of the paper, we show how the data structure for T and q can be merged in $O(|q|)$ time, allowing us to answer longest common prefix queries between suffixes of the two strings—we also discuss why other longest common prefix data structures do not seem to work for our use case. Each query requires $O(\tau + \log n)$ time and gives a correct answer with high probability.

To find the longest common prefix between two suffixes of T, we use the data structure of Kosolobov and Sivukin [19], which requires $O(n/\tau)$ space and preprocessing, and $O(\tau)$ query time.

These results immediately imply the following lemma.

Lemma 6. *Let $\widehat{s_1}$ be either a $(k+1)$-altered substring of T or a $(k+1)$-altered substring of q, and let $\widehat{s_2}$ be a $(k+1)$-altered substring of T. There is an $O(n/\tau)$ space data structure that can be built with $O(n)$ preprocessing on T and $O(|q|)$ preprocessing on q such that:*

- *If $\tau = 1$ or s_1 is from T, then the data structure can find the first $k+1$ mismatches between $\widehat{s_1}$ and $\widehat{s_2}$ in $O(k\tau)$ time.*
- *If $\tau > 1$ and s_1 is from q, the data structure can find the first $k+1$ mismatches between $\widehat{s_1}$ and $\widehat{s_2}$ with high probability in $O(k\tau + k \log n)$ time.*

With high probability means with probability $\geq 1 - 1/n^c$ for any constant c (increasing c increases the constant in the big-O bounds). During any query, the number of calls to Lemma 6 can be loosely upper bounded by $O(n)$. Therefore, by a standard union bound, *all* calls to Lemma 6 during a given query are correct with probability $\geq 1 - 1/n^{c-1}$, i.e. with high probability after incrementing c.

2.2 Dictionary Queries

In this work, we phrase the problem as finding substrings of a text T at Hamming distance k from a query q. However, many applications involve a slightly different problem: we are given a set of strings $s_1, s_2, \ldots, s_x$, and the goal is to find all strings s_i with distance at most k from q. Following [10], we refer to this as the *dictionary queries* variant of the problem, as opposed to the *text indexing* variant used elsewhere in the paper. In the dictionary queries variant, n is the total length of all strings in the set.

It is well known that the dictionary queries variant reduces to the text indexing variant. Assume that characters $1, 2, \ldots$ are not in Σ. We interleave these characters with those of q to create a new string: $q[1]\ 1\ q[2]\ 2\ q[3]\ 3 \ldots$ and so on. We modify $s_1, \ldots, s_x$ the same way, e.g. s_1 becomes $s_1[1]\ 1\ s_1[2]\ 2\ s_1[3]\ 3 \ldots$ and so on. We then concatenate the resulting strings to obtain T. The transformed q has distance $\leq k$ from a substring of T exactly when the original query has distance $\leq k$ from a string in $s_1, \ldots, s_x$.

3 A Recursive Formulation of the CGL Tree

In this section, we describe RecursiveCGL, our exposition of the CGL tree, and analyze it to prove Theorem 3.

Before formally defining our data structure, we give intuition behind the data structure, and state the key observations we use in its construction.

The Plan. To begin, we describe our strategy using an analogy with a static binary search tree (BST). In a BST, each node stores a pivot p. To ensure height $\log_2 n$, the pivot p is the median of the node's descendants; the node has one child for the elements $< p$, and another for the elements $> p$. A query traverses the tree recursively. At each node, q is compared to p; the left node is traversed if $q < p$ and the right node is traversed if $q > p$.

We use a similar structure to allow us to find the suffixes of T with Hamming distance at most k from the query q. We define the data structure as a recursive tree, where each node stores a pivot p.

Now, the goal is to replicate a BST-style recursion for Hamming distance. First, when constructing the data structure, we need to partition the strings s in each node using their relationship with the pivot p—we will need to generalize beyond simply storing strings $< p$ and strings $> p$. Second, when querying the data structure, our plan will be to compare q to p, and use the result of this comparison to make recursive queries.

Pivot-Altering Strings. One way that our data structure differs from both the static BST and the classic CGL tree is that we recursively change the strings when building our tree, and we recursively change the query string itself during queries. We define these changes using the pivot string p.

For any string s, we can *pivot-alter* s to match the pivot string p by one more character: in particular, the *pivot-altered* s can be obtained by calculating $i = LCP(s, p)$, and by appending the alteration $(i+1, p[i+1])$ to s. We will generally denote the result of this process $\widehat{s}$ for a fixed p.

Recursive Idea of the Tree: Intuition. Recall that the goal of RecursiveCGL is to, for each tree node, compare q to p, and recursively traverse the tree using the result of this comparison. The following observation is the heart of this process in RecursiveCGL (and is essentially the main idea behind the CGL Tree [10]). We first state our observation intuitively, then give a more formal statement.

For any query q and any string $s \neq q$, let ℓ be the first index where $q[\ell] \neq s[\ell]$. If $p[\ell] = q[\ell]$, then altering q makes it more similar to s: $d_H(\widehat{q}, s) = d_H(q, s) - 1$. Alternatively, if $p[\ell] = s[\ell]$, then altering s makes it more similar to q: $d_H(q, \widehat{s}) = d_H(q, s) - 1$. If $p[\ell] \neq q[\ell]$ and $p[\ell] \neq s[\ell]$, then altering both q and s makes them more similar: $d_H(\widehat{q}, \widehat{s}) = d_H(q, s) - 1$. We can rephrase these conditions using the LCP: if $LCP(q, p) < LCP(s, p)$, then $p[\ell] = s[\ell]$ but $p[\ell] \neq q[\ell]$.

Let us briefly discuss why this observation is useful. When creating the tree, we partition all strings into $O(1)$ sets according to their LCP with the pivot string p. On a query q, we begin by calculating $LCP(q, p)$. Let's look at two examples of how we recurse.

(1) if a set contains only strings s with $LCP(s, p) > LCP(q, p)$, we can recursively query that set using $\widehat{q}$, reducing the search radius by 1.
(2) if a set contains only strings s with $LCP(s, p) < LCP(q, p)$ we can recursively query all altered strings in the set using q, again reducing the search radius by 1.

Since the search radius can only be decreased k times, this process only searches a bounded number of nodes in the tree.

Pivot-Altered String Structure. Now we state our observation more formally. We begin with the case that q is not a prefix of p.

Observation 7. *Let q and s be arbitrary strings, and let $\widehat{q}$ and $\widehat{s}$ be the pivot-altered q and s respectively for a pivot string p. Let $i = LCP(q, p)$ and $j = LCP(s, p)$. Then if $i < |q|$ and $j < \min\{|s|, |p|\}$:*[4]

(1) If $i < j$, then $d_H(\widehat{q}, s) = d_H(q, s) - 1$;
(2) if $i > j$, then $d_H(q, \widehat{s}) = d_H(q, s) - 1$;
(3) if $i = j$ and $q[i+1] \neq s[j+1]$, then $d_H(\widehat{q}, \widehat{s}) = d_H(q, s) - 1$.

If $i = |q|$, then the query is a prefix of the pivot string, and the structure changes. Crucially, if $LCP(s, p) > |q|$, then $d_H(q, s) = 0$.

Observation 8. *Let q and s be arbitrary strings, and let $\widehat{q}$ and $\widehat{s}$ be the pivot-altered q and s respectively for a pivot string p. Let $i = LCP(q, p)$ and $j = LCP(s, p)$. Then if $i = |q|$ and $j < \min\{|s|, |p|\}$:*

(1) If $j \geq |q|$ then $d_H(q, s) = 0$;
(2) if $j < |q|$ then $d_H(q, \widehat{s}) = d_H(q, s) - 1$.

[4]The requirement that $j < \min\{|s|, |p|\}$ will always be satisfied: s and p will be k-altered suffixes of T; since T ends with $2k+1$ copies of a special character \$, there must be some place where s and p differ.

Table 2: An example for Observation 7. The first character mismatched between q or s and p—that is, the $i+1$st character of q and $j+1$st character of s, which are altered in $\widehat{q}$ and $\widehat{s}$—is highlighted in red.

$i < j$	$i > j$	$i = j$
p: BANANA	p: BANANA	p: BANANA
q: CANANC	q: BANACC	q: BACCNA
s: BACACA	s: BCNANA	s: BADANA
$\widehat{q}$: BANANC	q: BANACC	$\widehat{q}$: BANCNA
s: BACACA	$\widehat{s}$: BANANA	$\widehat{s}$: BANANA

3.1 The Recursive CGL Tree

In this section we formally describe how to build a recursive data structure, RecursiveCGL, to solve the Text Indexing with Mismatches problem. The goal of the data structure is to store all suffixes of T to handle queries with distance bound at most k. However, as mentioned above, as we recurse we alter the stored suffixes and change the search radius. Therefore, we describe the data structure recursively as storing a set S of k-altered suffixes of T, which can handle queries with distance bound at most k' for some $k' \leq k$; we denote this structure RecursiveCGL(S, k').

To begin, build the First $k+1$ Mismatches Data Structure from Lemma 6 on T with $\tau = 1$. (In Section 4, when we make the data structure space-efficient, we may use larger values of τ.)

Our data structure is recursive. The initial recursive call is to RecursiveCGL(S, k) where S consists of all suffixes of T (no alterations). In the base case $|S| \leq 1$, RecursiveCGL(S, k') stores S.

From now on we assume that $|S| > 1$ and describe how to build RecursiveCGL(S, k') recursively. First, we calculate the median of S in lexicographic order; we call this the *pivot string* p.

We will now use p to partition the strings in S. Our partition is defined with two goals in mind: (1) the partition should allow us to search for strings using Observation 7, and (2) the size of each partition should be at most $|S|/2$ to ensure tree height $\lceil \log_2 n \rceil$.

Let m be the median $LCP(s, p)$ for all strings $s \in S$. We partition $S \setminus \{p\}$ into the following four sets.

Definition 9 (Recursive Subsets)**.** For any set of strings S with pivot p, where m is the median value of $LCP(s, p)$ for $s \in S$,

- $S_{<m}$ consists of all strings $s \in S$ with $LCP(s, p) < m$;
- $S_{>m}$ consists of all strings $s \in S$ with $LCP(s, p) > m$;
- $S_{<\ell}$ consists of all strings $s \in S$ with $LCP(s, p) = m$ that occur before p in lexicographic order; and
- $S_{>\ell}$ consists of all strings $s \in S$ with $LCP(s, p) = m$ that occur after p in lexicographic order.

Lemma 10. *$S_{<m}, S_{>m}, S_{<\ell}, S_{>\ell}$ all have size at most $|S|/2$.*

For each of the four recursive subsets, we create an *altered subset* by pivot-altering every string in the recursive set (recall that to "pivot-alter" an altered suffix s, we alter it so that it matches p by one more character). Thus, $\widehat{S}_{<m}$ stores the pivot-altered s for all $s \in S_{<m}$; we similarly define $\widehat{S}_{<\ell}$, $\widehat{S}_{>\ell}$, and $\widehat{S}_{>m}$.

We are ready to define our recursive data structure.

Table 3: An example of recursive subsets and altered subsets for p = BANANA\$. In this example, $m = 3$ and $S_{<\ell}$ is empty. In reality, there are $2k+1$ copies of \$ at the end of each string, but these are truncated for simplicity.

S = AN\$, BACDA\$, BANAAAA\$, BANANA\$, BANAZAAAA\$, BANCNAB\$, BANCZZ\$, CAT\$

p = BANANA\$

$S_{<m}$	$S_{<\ell}$	$S_{>\ell}$	$S_{>m}$
CAT\$		BANCNAB\$	BANAAAA\$
AN\$		BANCZZ\$	BANAZAAAA\$
BACDA\$			

$\widehat{S}_{<m}$	$\widehat{S}_{<\ell}$	$\widehat{S}_{>\ell}$	$\widehat{S}_{>m}$
BAT\$		BANANAB\$	BANANAA\$
BN\$		BANAZZ\$	BANANAAAA\$
BANDA\$			

- For each recursive subset, we build the data structure recursively with distance bound k': RecursiveCGL$(S_{<m}, k')$, RecursiveCGL$(S_{<\ell}, k')$, RecursiveCGL$(S_{>\ell}, k')$, and RecursiveCGL$(S_{>m}, k')$.
- If $k' > 0$, for each altered subset except $\widehat{S}_{>m}$, we build the data structure recursively with distance bound $k' - 1$: RecursiveCGL$(\widehat{S}_{<m}, k'-1)$, RecursiveCGL$(\widehat{S}_{<\ell}, k'-1)$, and RecursiveCGL$(\widehat{S}_{>\ell}, k'-1)$. We do not need to build RecursiveCGL$(\widehat{S}_{>m}, k'-1)$; see the discussion below.

These recursive data structures form a tree structure: if $k' > 0$, RecursiveCGL(S', k') consists of the pivot string, and 7 pointers to the recursive data structures defined above (four recursive subsets, and three altered subsets). The pivot string consists of a pointer to a suffix of T and at most k alterations, which requires $O(k)$ space. Combining with the space to store the $O(1)$ pointers, we can store RecursiveCGL(S', k') in $O(k)$ space.

If $k' = 0$, we recurse on the recursive subsets, but we omit the altered sets; thus RecursiveCGL$(S', 0)$ consists of the pivot string, and 4 pointers to recursive data structures on the recursive subsets. Again, we can store the pivot using a pointer and the list of operations, giving $O(k)$ space per node.

The Recursive Data Structure as a Tree. For the remainder of this paper, we will treat this data structure as a degree 7 tree, which we refer to as[5] $\mathcal{T}_0$. The root of $\mathcal{T}_0$ represents RecursiveCGL(S, k), where S consists of all suffixes of T, each with an empty set of alterations; the children of each node represent the recursive data structures on the recursive subsets and altered subsets. This is essentially the same idea as a standard binary search tree—each child of a node is the root of a recursive tree on a subset of the data. We refer to a node of a tree, and the recursive data structure it represents, interchangeably. The height of $\mathcal{T}_0$ is at most $\lceil \log n \rceil$ by Lemma 10.

[5]The subscript 0 is to differentiate it from the space-efficient versions of this data structure we will look at later.

We say that a node RecursiveCGL(S, k) *contains* a suffix s of T if $(s, a) \in S$ for some (possibly empty) set of alterations a. We define the *size* of a node to be $|S|$.

3.1.1 Queries. We describe how to perform a query q with search radius $r \le k$. The query is recursive, traversing the tree $\mathcal{T}_0$; as the tree is traversed, the query is altered and the search radius decreases. In Section 3.1.1, we use the notation q solely to refer to the original query string, and q' to a recursive (possibly altered) query; similarly, r is the original search radius, and r' is the recursive search radius. The query begins at the root of $\mathcal{T}_0$.

We now describe how a query is performed. We split into two cases based on if the recursive search radius $r' > 0$ or $r' = 0$. For the case where $r' > 0$, we split into two further cases based on whether or not q' is a prefix of the current node. For the case where $r' = 0$, we give two strategies. First, we explain how to traverse the tree recursively. Then, we summarize how storing additional data structures allows us to traverse the tree much more quickly; we bound the performance of this strategy in Section 5.

Positive Search Radius $r' > 0$. We describe how to traverse a node v representing RecursiveCGL(S, k') for some S; let $r' > 0$ be the current search radius. First, we determine whether the distance from q' to the pivot string p of v is $\le r'$ in $O(k)$ time using Lemma 6. If so, we return p. In the base case, if $|S| \le 1$, we are done.

Otherwise, we find $i := LCP(q', p)$ using Lemma 6. Once we find i, we can pivot-alter q' in $O(1)$ time; denote the result $\widehat{q}$. We can also determine if q' is before or after p in lexicographic order in $O(1)$ time.

We split into two subcases based on if $i < |q'|$ or $i = |q'|$.

If $i < |q'|$, we split into four cases based on i and whether q' is before or after p in lexicographic order. In all cases, we perform at most 4 recursive queries: at most 1 recursive query with search radius r, and at most 3 with search radius $r - 1$. These recursive queries can be obtained immediately from Observation 7; we list them explicitly in Table 4.

If $i = |q'|$, then q' is a prefix of the pivot string p. This changes our recursive calls since *any* string $s \in S$ with $LCP(s, p) > i$ has $d_H(q', s) = 0$. We handle these items by performing a depth-first search (DFS) on the tree to recover S: we recursively search all unaltered children of each node; the pivots of these nodes are exactly the elements of S. (We chose DFS arbitrarily; this could be handled equally well by BFS or any other kind of tree traversal.)

Traversing the Tree with $r' = 0$. If $r' = 0$, we can simply ignore all recursive calls with search radius $r - 1$. There is only one non-DFS recursion for each node.

For the remainder of this paragraph, we define traversing the nodes in more detail, giving some structural definitions that will be useful throughout the rest of the paper. Then, we give a corollary that significantly speeds up our algorithm: rather than traversing the tree one node at a time for $r' = 0$, we can quickly "jump" to the final result of the traversal.

Let us look at the recursive calls with search radius r. If $i < m$ we recurse in $S_{<m}$, if $i = m$ and $\widehat{q} < p$ we recurse on $S_{<\ell}$, and so on. In general, we traverse to the node whose recursive set would contain the query $\widehat{q}$.

Table 4: The recursive calls made based on the query if $i < |q'|$ using Observation 7. We use $q' < p$ to denote that q' is before p in lexicographic order.

	$S_{<m}$	$\widehat{S}_{<m}$	$S_{<\ell}$	$\widehat{S}_{<\ell}$	$S_{>\ell}$	$\widehat{S}_{>\ell}$	$S_{>m}$	$\widehat{S}_{>m}$
$i < m$	q', r		$\widehat{q}, r-1$		$\widehat{q}, r-1$		$\widehat{q}, r-1$	
$i = m, q' < p$		$q', r-1$	q', r			$\widehat{q}, r-1$	$\widehat{q}, r-1$	
$i = m, q' > p$		$q', r-1$		$\widehat{q}, r-1$	q', r		$\widehat{q}, r-1$	
$i > m$		$q', r-1$		$q', r-1$		$q', r-1$	q', r	

Table 5: The recursive calls made based on the query if $i = |q'|$ using Observation 8.

	$S_{<m}$	$\widehat{S}_{<m}$	$S_{<\ell}$	$\widehat{S}_{<\ell}$	$S_{>\ell}$	$\widehat{S}_{>\ell}$	$S_{>m}$
$i < m$	q', r		DFS		DFS		DFS
$i = m$		$q', r-1$		$q', r-1$		$q', r-1$	DFS
$i > m$		$q', r-1$		$q', r-1$		$q', r-1$	q', r

This traversal process is currently most important for q', but in fact it is well-defined for any string s: starting at any node $v \in \mathcal{T}_0$, we traverse to the descendant of v whose recursive set would contain s, continuing until reaching a leaf. We call this process a *manual traversal*; we call the last node reached in a manual traversal the *traversal destination*.

Definition 11 (Traversal Destination; Manual Traversal)**.** Let $\mathcal{T}'$ be a subtree of $\mathcal{T}_0$, and let s be an arbitrary string. We define a path through $\mathcal{T}'$ which we call the *manual traversal*. We define one recursive subset to recurse on. If $LCP(s, p) < m$ we recurse on $S_{<m}$; if $LCP(s, p) > m$ we recurse on $S_{>m}$; if $LCP(s, p) = m$ and $s < p$ we recurse on $S_{<\ell}$; if $LCP(s, p) = m$ and $s > p$ we recurse on $S_{>\ell}$. We define the *traversal destination* to be the leaf of $\mathcal{T}'$ reached by this process.

Our goal is to avoid the manual traversal, which may traverse $\Omega(\log n)$ nodes. In Section 5 we show that we can indeed find the traversal destination in $\text{poly}(k, \log\log n)$ time.

However, this result is not sufficient to answer a query—we cannot just jump to the final node traversed, as we may skip matching strings found along the way. Specifically, as we traverse the tree, we compare the query q to the pivot of each node along the traversal path. If they match—q is a prefix of p—then we must return p as a solution to the problem. Then, using Table 5, we may call one or more DFS subroutines to return further matching pivots.

The following definition address this gap: we define the set of nodes which we must traverse to correctly answer a query with radius $r' = 0$.

Definition 12. Let $\mathcal{T}'$ be a subtree of $\mathcal{T}_0$, let v be a node in $\mathcal{T}'$, and let q' be a k-altered query. The *Matching Nodes for v and q'*, denoted $P(v, q', \mathcal{T}')$, are the set of nodes $v' \in \mathcal{T}'$ such that: (1) v' is a descendant of v in $\mathcal{T}'$, and (2) q' is a prefix of p.

The following corollary shows that we can find the Matching Nodes of a query quickly, without manually traversing the tree. This corollary is a special case of Lemma 27, stated in Section 5.

Corollary 13. *There is a data structure using $O(n\binom{\log n}{k})$ space that, for any k-altered query q' can find $P(v, q', \mathcal{T}_0)$ in time*

$$O\left(k^2 + k\log\log n + k|P(v, q', \mathcal{T}_0)|\right).$$

With this data structure, we can finish our queries. To query a node v of the tree with query q' and search radius $r' = 0$, we use Corollary 13 to find $P(v, q', \mathcal{T}_0)$. Each pivot (s_i, a) of a node in $P(v, q', \mathcal{T}_0)$ satisfies $d_H(s_i, q) \le r$; we return all such s_i.

3.2 Analysis

We analyze the query time and space of RecursiveCGL to prove Theorem 3. Both can be analyzed by substituting into the recurrence implied by the data structure's definition (as was done in [10]). However, it will be helpful for our later analysis to have additional structure. In particular, we give an analysis which labels the edges of $\mathcal{T}_0$; then we use these labels to obtain our bounds.

3.2.1 Query Time. We begin by bounding the number of times we invoke Corollary 13. This lemma forms the core of our running time proof. Let $\mathcal{T}_q$ be the subtree of $\mathcal{T}_0$ traversed using recursive calls that (1) have radius $r > 0$, and (2) are not a DFS call; thus we invoke Corollary 13 once for each leaf of $\mathcal{T}_q$.

Lemma 14. $|\mathcal{T}_q| = O\left(3^k\binom{\log n}{k}\right)$.

To begin bounding the other terms, we show that each string is only considered once by the query—this is crucial for the query time to be linear in $(\#occ)$.

Lemma 15. *Let s_i be a suffix of T, and let Q be the nodes of $\mathcal{T}_0$ traversed during a query. Then there is at most 1 node in Q with pivot (s_i, a) for some set of alterations a.*

Now, we can bound the query time.

Lemma 16. *The total time to perform a query q with output size $\#occ$ is*

$$O\left(|q| + 3^k\binom{\log n}{k}(k^2 + k\log\log n) + k \cdot \#occ\right).$$

3.2.2 Space. To bound the space, we assign labels to each edge of the tree, and bound the number of possible labels combinatorially. For now, this is an analysis tool. However, this definition will be important for our space-efficient approach in Section 4, where we will maintain path labels explicitly.

Definition 17. The *path label* of any node $v \in \mathcal{T}_0$ is a string consisting of characters a and u. It is defined recursively as follows. The path label of the root is the empty string. Let ℓ be the path label of a node v. For any child c of v, the path label of c can be

obtained by adding u to ℓ if c is an (unaltered) recursive subset, and by adding a to ℓ if c is an altered subset.

Observation 18. *The length of a path label is at most $\lceil \log_2 n \rceil$, and each path label contains at most k a's.*

The path labels are useful for bounding the space (and will be useful for defining the functions in Section 4) since they are a way to track, for each suffix s_i of T, where altered suffixes (s_i, a) are stored in $\mathcal{T}_0$.

Observation 19. *For some suffix s_i of T, consider two nodes v_1 and v_2 in $\mathcal{T}_0$ (and alterations a_1 and a_2) such that (s_i, a_1) is a pivot of v_1 and (s_i, a_2) is a pivot of v_2. Then the path labels of v_1 and v_2 are distinct; furthermore, the path label of one is not a prefix of the path label of the other.*

With this observation in hand the space bound can be obtained, in short, by multiplying the number of suffixes of T by the number of possible path labels.

Lemma 20. *Any suffix s_i of T is the pivot of $O(k\binom{\log n}{k})$ nodes of $\mathcal{T}_0$; therefore, RecursiveCGL requires $O(nk^2\binom{\log n}{k})$ total words of space.*

3.2.3 Preprocessing Time. The data structures for Lemma 6 and Corollary 13 can be built in $O(nk^2\binom{\log n}{k}\log n)$ time. All that remains to prove Theorem 3 is to bound the time to build $\mathcal{T}_0$.

Lemma 21. *$\mathcal{T}_0$ can be built in time $O(nk^2\tau\binom{\log n}{k}\log n)$.*

4 The Space-Efficient Data Structure

In this section we describe the data structure satisfying Theorems 1 and 4. We first describe how to truncate the tree to save space, then we describe how function inversion can help us perform queries on the truncated tree. Then we describe how to perform function inversion efficiently, proving Theorem 5, and finally we put it all together to achieve the final bounds.

4.1 The Truncated Tree

We define the *truncated tree* similarly to RecursiveCGL, but with a key difference in the base case. If $|S| \le \sigma$, then the node consists only of an integer *label*; nothing else is stored, and we do not recurse further. We use $\mathcal{T}$ to denote the truncated tree constructed on all suffixes of T.

Since the nodes of $\mathcal{T}$ are also nodes of $\mathcal{T}_0$, we can define path labels using Definition 17; we will use path labels both for space analysis and to define the functions for function inversion.

Consider all leaves in $\mathcal{T}$ with a given path label λ. These leaves can be found by traversing $\mathcal{T}$ recursively. We label all leaves in $\mathcal{T}$ with path label λ using increasing integers starting at 1. The following lemma implies that the maximum label of any node is $O(n/\sigma)$.

Lemma 22. *For any path label λ, $\mathcal{T}$ has $O(n/\sigma)$ leaves with path label λ. $\mathcal{T}$ has $O(nk\binom{\log n}{k}/\sigma)$ nodes in total.*

4.2 Defining the Functions We Will Invert

We define a function f_λ for each path label λ. For any $i \in [n]$, $f_\lambda(i)$ is the label of the leaf $\ell \in \mathcal{T}$ such that ℓ has path label λ and ℓ contains s_i. There is at most one such ℓ by Observation 19 since each leaf ℓ of $\mathcal{T}$ contains exactly the suffixes that are pivots of the descendants of ℓ in $\mathcal{T}_0$; if no such ℓ exists then we define $f_\lambda(i) = \perp$.

Thus, $f_\lambda(i) : [n] \to [O(n/\sigma)] \cup \{\perp\}$. In the following discussion, we will pad the codomain of $f_\lambda(i)$ to have size exactly n; in other words, $f_\lambda(i) : [n] \to [n-1] \cup \{\perp\}$.

From this definition, we immediately observe that all elements of the codomain except $\perp$ have a small preimage: since every leaf in $\mathcal{T}$ contains $O(\sigma)$ elements, we have $|f_\lambda^{-1}(j)| = O(\sigma)$ for all $j \in [n-1]$. This observation will be important in the subsequent analysis.

Evaluating the Functions. Let $E(f_\lambda)$ be the worst-case time to evaluate $f_\lambda(i)$ for any λ, i. We can evaluate $f_\lambda(i)$ in $O(k\tau \log n)$ time by traversing $\mathcal{T}$ (the height of the tree is $O(\log n)$; using Lemma 6 we can determine which subset contains s_i in $O(k\tau)$ time since s_i and the pivot p are altered suffixes of T).

However, the key idea behind function inversion [12] is that we repeatedly calculate $f_\lambda(\cdot)$ (in the forward direction) in order to find an element's inverse $f_\lambda^{-1}(j)$. Therefore, the time spent calculating $f_\lambda(\cdot)$ has a significant impact on our bounds—we would like to avoid spending $\Omega(\log n)$ time traversing $\mathcal{T}$. We point out that this very similar to the goal of Corollary 13: we want to "jump" right to the result of the traversal.

In Section 5, we improve the time to evaluate $f_\lambda(\cdot)$ to $O(k^3 + k^2 \log\log n)$; see Lemma 26. The space bound of Lemma 26 has an additive $\Theta(n)$ term; therefore, we only use this lemma if $\tau = 1$. Our succinct data structure ($\tau > 1$) evaluates $f_\lambda(i)$ by traversing $\mathcal{T}$ in $O(k\tau \log n)$ time.

4.3 Proof of Theorem 5

In this section, we prove Theorem 5, giving a data structure to efficiently invert an arbitrary function f satisfying the requirements of Theorem 5. Later, we will use this result with $f = f_\lambda$ for the functions f_λ defined above to achieve our final bounds.

This proof is in two parts: first, we describe how the classic Fiat-Naor data structure extends to our setting; then, we extend their analysis to find all points in the inverse to Theorem 5.

4.3.1 Recalling the Fiat-Naor Data Structure. Let's begin with a review of the data structure from [12]. For the most part, our exposition here is identical to theirs; however, we tweak their parameters slightly to fit our analysis, and we omit the data structure they use to store high-indegree elements.

We briefly describe the idea behind their data structure before giving a detailed exposition. Consider an element $i \in [n]$; apply f repeatedly to i to obtain $f(i), f(f(i))$, and so on for σ operations; call this a "chain." If we store a mapping from the last element to the first element in the chain, we can invert all elements in the chain as follows. To invert an element j, we start applying f to j until reaching the last element in the chain; following the pointer to i, we continue applying f until reaching j again. The penultimate element in this strategy is in $f^{-1}(j)$.

The challenge is ensuring that all elements are in some chain, and ensuring this works for any f. This is done by composing f with a random hash function, and creating a large number of chains (called a "cluster") for the composed function. Repeatedly creating clusters, with a new hash each time, is sufficient to lower bound the probability that j is in some chain.

Creating the Chains. We create a sequence of *clusters*. For each cluster c, we select a 2σ-independent hash function $g_c : [2n] \to [n]$ from the family given in [12, Section 4.4]. This hash family can be stored in $O(1)$ amortized space, and can be evaluated in $O(\log \sigma)$ amortized time. (The amortization is over all hashes for all clusters; since each query is always repeated for each cluster, we can treat these bounds as worst-case on a given cluster.) We define a function h_c that composes g_c and f, avoiding the high-indegree element $\perp$:

$$h_c(i) = \begin{cases} g_c(f(i)) & \text{if } f(i) \neq \perp; \\ g_c(n+i) & \text{otherwise.} \end{cases}$$

This function differs from the one in [12]: they avoid high-indegree elements by repeatedly hashing to avoid elements sampled in a lookup table. Since we know that $\perp$ is our only high-indegree element we can instead handle it explicitly. This hash function is defined so that $\Pr_{i \neq j}[h_c(i) = h_c(j)] = O(\sigma/n)$, where the probability is taken over the choice of g_c.

For each cluster, we define a set of *chains*. For any $x \in [n]$, the chain of elements starting at x is the sequence of σ elements starting with x, where each element is obtained by applying h_c to the previous element:

$$x, h_c(x), h_c(h_c(x)), \ldots, \underbrace{h_c(h_c(h_c(\ldots h_c}_{\sigma \text{ iterations}}(x)))).$$

For any $x, i \in [n]$, we say that x *finds* i if i is in the chain of elements starting at x.

For each cluster, we sample n/σ^3 elements x uniformly at random from $[n]$. For each such x, we store a key-value pair in a dictionary, where the key is the last element in the chain starting at x, and the value is x. We use a linear-space dictionary which achieves constant-time insert in expectation and constant-time worst-case lookup, e.g., a cuckoo hash table suffices, as does [2]. Thus, in $O(1)$ time, for any element ℓ, we can: (1) determine if ℓ is the last element in the chain starting at some sampled element x, and (2) if so, find x. A cluster consists of the hash function g_c and this dictionary.

Now, we will describe how to use a cluster to (attempt to) find $f^{-1}(j)$ for some $j \in [n-1]$. First, we calculate the chain starting at j using h_c, exactly as we did when creating the cluster. For each x_c in the chain starting at j, we look up x_c in the dictionary. If none are found, the query fails. If we find some x_c in the dictionary, it is the last element in the chain starting at some element x; x is stored as the key in the dictionary. We then calculate the chain starting at x. If it contains j, then the element preceding j in the chain starting at x is $f^{-1}(j)$; we return that element. If it does not contain j, the query fails. Note that we only perform the above for one x_c, so we only calculate two chains.

Lemma 23. *Each cluster c can be stored in $O(n/\sigma^3)$ space and created in $O(n(E(f) + \log \sigma)/\sigma^2)$ time. We can query c to attempt to find $f^{-1}(j)$ for some $j \in [n]$ in $O(\sigma(E(f) + \log \sigma))$ time.*

The following is the main correctness lemma for function inversion. Its proof is closely adapted from the proof of [12, Lemma 4.2]—it is essentially the same proof, but we do not need to condition on having sampled high-indegree elements since we handle them explicitly in h_c.

Lemma 24. *Let i and j satisfy $f(i) = j$. Then the probability that i is returned by querying a given cluster c is $\Omega(1/\sigma^2)$.*

Creating Multiple Clusters. We begin with a method that does not quite work. Consider creating $\Theta(\sigma^2 \log \sigma)$ clusters. For each cluster, we sample a new hash function g_c to construct a new h_c, and randomly sample new elements to start the chains. The extra $\log \sigma$ term is for Lemma 25 below: in short, Fiat-Naor created $\Theta(\sigma^2)$ clusters to ensure each element is returned with constant probability, so repeating an extra σ times ensures that each element is returned with probability $1 - O(1/\sigma)$. Unfortunately, this strategy has space usage $\Theta(n(\log \sigma)/\sigma)$ rather than $O(n/\sigma)$. Therefore, we adjust parameters to retain the final Lemma 25 result while still meeting the $O(n/\sigma)$ space bound; in exchange, the preprocessing and query times will be slightly higher.

We create $\Theta(\sigma^2 \log^3 \sigma)$ clusters with parameter $\sigma' := \sigma \log \sigma$. By Lemma 23, each cluster requires $O(n/(\sigma \log \sigma)^3)$ space, answers queries in $O(\sigma \log \sigma(E(f) + \log \sigma))$ time, and can be created in $O(n(E(f) + \log \sigma)/(\sigma \log \sigma)^2)$ time. By Lemma 24, a given $i \in f^{-1}(j)$ is returned by querying a given cluster with probability $\Omega(1/(\sigma \log \sigma)^2)$.

With this adjustment, we have $O(n/\sigma^3)$ total space over all clusters, and we can give our lemma showing that a given element is likely to be inverted by some cluster.

Lemma 25. *For any j, if $i \in f^{-1}(j)$, with probability $1 - O(1/\sigma)$ there exists a cluster c such that there exists a sampled x in cluster c where x finds i.*

4.3.2 Inverting All Points. Now, we extend the data structure to obtain *all* points in a given element's preimage.

Lemma 25 leaves us with $O(n/\sigma)$ expected $i \in [n]$ that are not inverted by the data structure: $O(n/\sigma)$ elements i such that $f(i) = j$ for some $j \in [n]$, but i is not returned when finding $f^{-1}(j)$ using the above strategy. We call these the *missing items*.

We want to be sure that the missing items will also be returned during a function inversion query. To this purpose, we store another linear-space dictionary with constant-time lookup and expected linear preprocessing time. The keys will consist of all j such that there is a missing item in $f^{-1}(j)$. The value for each such j will be all missing i in $f^{-1}(j)$. (Note that we only store the missing elements as values, not all elements in $f^{-1}(j)$.)

Now, let's describe how to calculate these values. The items not in any chain can be found while creating the chains, by keeping an array on $[n]$ and marking all elements found by each chain. (The preprocessing space can be reduced to $O(n/\sigma)$ in exchange for increasing the preprocessing time by σ, by repeating this process for each set of n/σ elements.) Then, we iterate through each i not found in any chain, and calculate $f(i)$ in $O(E(f))$ time. If the key $f(i) \neq \perp$ is not already a stored in the dictionary, we add it (initialized with the empty set as its value). Then we add i to the set stored as the value for key $f(i)$.

With this extra dictionary for elements that are not inverted by a chain, we add an extra step to the query. On a query j, we first query the dictionary using j to find any missing items in $f^{-1}(j)$ (this takes time $O(1 + |f^{-1}(j)|)$), and then search for j in the chains.

We have now proven Theorem 5.

4.4 Putting It All Together

What to Store. Our data structure consists of the following parts. First, we store the First $k+1$ Mismatches Data Structure from Lemma 6 built on T. Then, we store the truncated tree $\mathcal{T}$. Then, we store the function inversion data structure from Theorem 5 for each f_λ.

Queries. We describe the query as constructing a list L consisting of leaves of $\mathcal{T}$; our plan is to find all suffixes contained in L using function inversion, and return all that are close to q. Note that we do not need to actually store L explicitly as the leaves can be handled one by one during a query.

We construct L using the following recursive process:

- to recurse with $r' > 0$, we use the process outlined in Section 3.1.1, outputting any matching pivots found, and adding any leaf of $\mathcal{T}$ reached to L.
- The base case is when $r' = 0$ for some altered query q' and node $v \in \mathcal{T}$. When $r' = 0$ we first find the traversal destination and the matching nodes of $v \in \mathcal{T}$ and q'. If $\tau > 1$, this is done using a manual traversal (Definition 11); if $\tau = 1$ this is done using Lemma 27 (see Section 5).
  Once we have found the traversal destination and matching nodes, we process them. For each node $v' \in P(v, q', \mathcal{T})$, if v' is a leaf of $\mathcal{T}$, we add v' to L; if v' is an internal node of $\mathcal{T}$, we add its pivot to the output.

Now, for each leaf in L, let ℓ be the label of the leaf and λ be its path label. We use the function inversion data structure from Theorem 5 to find $f_\lambda^{-1}(\ell)$. Then, for each $i \in f_\lambda^{-1}(\ell)$, we use Lemma 6 to find $d_H(q, s_i)$; if $d_H(q, s_i) \le r$ then we add s_i to the output.

5 Improving Final Log Factors

In the full version of the paper, we prove the following two lemmas, which improve our query performance by a $\log n$ factor if $\tau = 1$.

We show how to evaluate $f_\lambda(i)$ in $\text{poly}(k, \log\log n)$ time.

Lemma 26. *There exists a data structure using $O(n\binom{\log n}{k}/\sigma + n)$ space that can be built in $O(kn\binom{\log n}{k}/\sigma + kn)$ expected time such that for any given λ and i, the data structure can find $f_\lambda(i)$ in $E(f_\lambda) = O(k^3 + k^2 \log\log n)$ time.*

We show how to answer queries with search radius $r' = 0$ in $\text{poly}(k, \log\log n)$ time.

Lemma 27. *There is a data structure using $O(n\binom{\log n}{k}/\sigma + n)$ space that can be built using $\mathcal{T}$ in $O(n\binom{\log n}{k}/\sigma + n)$ expected time and can be initialized for q in $O(|q|)$ time such that, for any k-altered query q', can find the matching nodes $P(v, q', \mathcal{T})$ and the traversal destination v_T of q' and v in $O\left(k^2 + k\log\log n + k|P(v, q', \mathcal{T})|\right)$ time.*

Acknowledgments

This research is supported in part by National Science Foundation grants CAREER-2045641, CNS-2325956, and CCF-2103813. We would like to thank the anonymous reviewers for their helpful comments and feedback on previous versions of this paper.

References

[1] Peyman Afshani. 2012. Improved pointer machine and I/O lower bounds for simplex range reporting and related problems. In *Proc. 28th Annual Symposium on Computational Geometry (SoCG)*. ACM, New York, NY, USA, 339–346.
[2] Yuriy Arbitman, Moni Naor, and Gil Segev. 2010. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *Proc. 51st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, IEEE Computer Society, 2001 L Street N.W., Suite 700, Washington, DC, 787–796.
[3] Boris Aronov, Jean Cardinal, Justin Dallant, and John Iacono. 2024. A general technique for searching in implicit sets via function inversion. In *Proc. 7th Annual Symposium on Simplicity in Algorithms (SOSA)*. SIAM, Philadelphia, PA USA, 215–223.
[4] Boris Aronov, Esther Ezra, Micha Sharir, and Guy Zigdon. 2023. Time and space efficient collinearity indexing. *Computational Geometry* 110 (2023), 101963.
[5] Philip Bille, Inge Li Gørtz, Mathias Bæk Tejs Knudsen, Moshe Lewenstein, and Hjalte Wedel Vildhøj. 2015. Longest common extensions in sublinear space. In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*. Springer, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 65–76.
[6] Philip Bille, Inge Li Gørtz, Moshe Lewenstein, Solon P. Pissis, Eva Rotenberg, and Teresa Anna Steiner. 2024. Gapped String Indexing in Subquadratic Space and Sublinear Query Time. In *Proc. 41st International Symposium on Theoretical Aspects of Computer Science (STACS) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 289)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:21. doi:10.4230/LIPIcs.STACS.2024.16
[7] Ho-Leung Chan, Tak-Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Swee-Seong Wong. 2011. A linear size index for approximate pattern matching. *Journal of Discrete Algorithms* 9, 4 (2011), 358–364. doi:10.1016/j.jda.2011.04.004
[8] Archie L. Cobbs. 1995. Fast approximate matching using suffix trees. In *Proc. 6th Annual Symposium on Combinatorial Pattern Matching (CPM)*. Springer, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 41–54.
[9] Vincent Cohen-Addad, Laurent Feuilloley, and Tatiana Starikovskaya. 2019. Lower Bounds for Text Indexing with Mismatches and Differences. In *Proc. 30th Annual Symposium on Discrete Algorithms (SODA)*. ACM-SIAM, SIAM, Philadelphia, PA USA, 1146–1164.
[10] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. 2004. Dictionary matching and indexing with errors and don't cares. In *Proc. 36th Annual ACM Symposium on Theory of Computing (STOC)*. ACM, ACM, New York, NY USA, 91–100.
[11] Henry Corrigan-Gibbs and Dmitry Kogan. 2019. The function-inversion problem: Barriers and opportunities. In *Proc. 17th Theory of Cryptography Conference (TCC)*. Springer, Springer-Verlag, Berlin, Heidelberg, Germany, 393–421.
[12] Amos Fiat and Moni Naor. 2000. Rigorous time/space trade-offs for inverting functions. *SIAM J. Comput.* 29, 3 (2000), 790–803.
[13] Alexander Golovnev, Siyao Guo, Thibaut Horel, Sunoo Park, and Vinod Vaikuntanathan. 2020. Data structures meet cryptography: 3SUM with preprocessing. In *Proc. 52nd Annual ACM Symposium on Theory of Computing (STOC)*. ACM, ACM, New York, NY USA, 294–307.
[14] Martin Hellman. 1980. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory* 26, 4 (1980), 401–406.
[15] Trinh N.D. Huynh, Wing-Kai Hon, Tak-Wah Lam, and Wing-Kin Sung. 2006. Approximate string matching using compressed suffix arrays. *Theoretical Computer Science* 352, 1-3 (2006), 240–249.
[16] Yael Kirkpatrick, John Kuszmaul, Surya Mathialagan, and Virginia Vassilevska Williams. 2026. Preprocessed 3SUM for Unknown Universes with Subquadratic Space. https://arxiv.org/abs/2602.11363
[17] Tomasz Kociumaka and Jakub Radoszewski. 2026. Space-Efficient k-Mismatch Text Indexes. In *Proceedings of the 37th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. ACM, SIAM, Philadelphia, PA USA, 1873–1902. doi:10.1137/1.9781611978971.68
[18] Tsvi Kopelowitz and Ely Porat. 2019. The strong 3SUM-INDEXING conjecture is false. https://arxiv.org/abs/1907.11206
[19] Dmitry Kosolobov and Nikita Sivukhin. 2024. Construction of Sparse Suffix Trees and LCE Indexes in Optimal Time and Space. In *Proc. 35th Annual Symposium on Combinatorial Pattern Matching (CPM)*, Vol. 296. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 20:1–20:18.
[20] Tak-Wah Lam, Wing-Kin Sung, and Swee-Seong Wong. 2008. Improved approximate string matching using compressed suffix data structures. *Algorithmica* 51, 3 (2008), 298–314.
[21] Samuel McCauley. 2024. Improved Space-Efficient Approximate Nearest Neighbor Search Using Function Inversion. In *Proc. 32nd European Symposium on Algorithms (ESA) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 308)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 88:1–88:19.
[22] Marvin Minsky and Seymour A. Papert. 1969. *Perceptrons*. MIT Press, Cambridge, MA USA.
[23] Marius Nicolae and Sanguthevar Rajasekaran. 2017. On pattern matching with k mismatches and few don't cares. *Inform. Process. Lett.* 118 (2017), 78–82.