Exact and Approximate Range Mode Query Data Structures in Practice

Meng He

Dalhousie University, Halifax, Canada

Zhen Liu

Dalhousie University, Halifax, Canada

Abstract

We conduct an experimental study on the range mode problem. In the exact version of the problem, we preprocess an array A, such that given a query range $[a,b]$, the most frequent element in $A[a,b]$ can be found efficiently. For this problem, our most important finding is that the strategy of using succinct data structures to encode more precomputed information not only helped Chan et al. (Linear-space data structures for range mode query in arrays, Theory of Computing Systems, 2013) improve previous results in theory but also helps us achieve the best time/space tradeoff in practice; we even go a step further to replace more components in their solution with succinct data structures and improve the performance further.

In the approximate version of this problem, a $(1+\varepsilon)$-approximate range mode query looks for an element whose occurrences in $A[a,b]$ is at least $F_{a,b}/(1+\varepsilon)$, where $F_{a,b}$ is the frequency of the mode in $A[a,b]$. We implement all previous solutions to this problems and find that, even when $\varepsilon = \frac{1}{2}$, the average approximation ratio of these solutions is close to 1 in practice, and they provide much faster query time than the best exact solution. These solutions achieve different useful time-space tradeoffs, and among them, El-Zein et al. (On Approximate Range Mode and Range Selection, 30th International Symposium on Algorithms and Computation, 2019) provide us with one solution whose space usage is only 35.6% to 93.8% of the cost of storing the input array of 32-bit integers (in most cases, the space cost is closer to the lower end, and the average space cost is 20.2 bits per symbol among all datasets). Its non-succinct version also stands out with query support at least several times faster than other $O(\frac{n}{\varepsilon})$-word structures while using only slightly more space in practice.

2012 ACM Subject Classification Information systems → Data structures

Keywords and phrases range mode query, exact range mode query, approximate range mode query

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.19

Supplementary Material *Software (Source Code)*: `https://github.com/Kolento777/RangeModeQueries`, archived at `swh:1:dir:5d61144576ed7d45a2e424ae08b6b010c1a6e90c`

Funding This work is supported by NSERC.

1 Introduction

The *mode*, or the most frequent element, in a dataset is a widely used descriptive statistic. In the *range mode query problem*, we preprocess an array A of length n, such that, given a query range $[a,b]$, the *mode* in $A[a,b]$ can be computed efficiently. Many problems in data analytics and retrieval can be abstracted to range mode. For example, an online shopping platform may be interested in the most popular item purchased by customers over a certain period, which can be found by a range mode query over the sales records in its database.

Range mode is also connected to matrix multiplication; the product of two $\sqrt{n} \times \sqrt{n}$ Boolean matrices can be computed by answering n range mode queries in an array of length $\mathcal{O}(n)$ [7]. This reduction provides a conditional lower bound showing that, with current knowledge, the time required to preprocess an array and answer n range mode queries must be $\Omega(n^{\omega/2})$, where $\omega < 2.3726$ is the best exponent in matrix multiplication [2].

21st International Symposium on Experimental Algorithms (SEA 2023).
Editor: Loukas Georgiadis; Article No. 19; pp. 19:1–19:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Furthermore, since the best combinatorial algorithm for Boolean matrix multiplication is only a polylogarithmic factor better than cubic [4], with current knowledge, we cannot use pure combinatorial approaches to solve range mode in $O(n^{3/2-\delta})$ preprocessing time and $O(n^{1/2-\delta})$ query time simultaneously for any constant $\delta \in (0, 1/2)$. To speed up queries, researchers further define the $(1+\varepsilon)$-*approximate range mode query problem*, where $\varepsilon \in (0, 1)$. Given a query range $[a, b]$, let $F_{a,b}$ denote the frequency of the mode in $A[a, b]$. A $(1 + \varepsilon)$-approximate range mode query then asks for an element whose occurrences in $A[a, b]$ is at least $F_{a,b}/(1 + \varepsilon)$.

Due to the importance in both theory and practice, range mode has been studied extensively [22, 29, 7, 6, 18, 12, 13, 32, 31, 19]. Despite these efforts, we are not aware of any experimental studies on them. Hence, to connect theory to practice, we conduct an empirical study of exact and approximate range mode structures using large practical datasets.

Related Work. Krizanc et al. [22] first considered the exact range mode problem and introduced an $\mathcal{O}(n + s^2)$-word solution with $\mathcal{O}((n/s) \lg n)$ query time for any $s \in [1, n]$, and setting $s = \sqrt{n}$ yields a linear space solution with $\mathcal{O}(\sqrt{n} \lg n)$ query time. They also presented another solution with constant query time and $\mathcal{O}(n^2 \lg\lg n/ \lg n)$ words of space cost. Later Petersen et al. [29] proposed an $\mathcal{O}(n^2 \lg\lg n/ \lg^2 n)$-word structure with constant query time. Chan et al. [7] further improved the time-space tradeoff of Krizanc et al. by designing an $O(n + s^2/w)$-word data structure with $O(n/s)$ query time, where w is the number of bits in a word. This result implies a linear space solution in words with $\mathcal{O}(\sqrt{n/w})$ query time.

Regarding $(1 + \varepsilon)$-approximate range mode, Bose et al. [6] first used persistent search trees to design an $O(\frac{n}{\varepsilon})$-word solution with $\mathcal{O}(\lg\lg n + \lg \frac{1}{\varepsilon})$ query time. Greve et al. [18] provided another structure with $\mathcal{O}(\lg \frac{1}{\varepsilon})$ query time and $\mathcal{O}(\frac{n}{\varepsilon})$ words of space, and they used succinct data structures. More recently, El-Zein et al. [12] designed an encoding data structure occupying only $\mathcal{O}(\frac{n}{\varepsilon})$ bits, and without accessing the original array, it can also report the position of a $(1 + \varepsilon)$-approximate mode in the query range in $\mathcal{O}(\lg \frac{1}{\varepsilon})$ time.

Our Work. We first study linear-space exact range mode structures [22, 7]. Much of this study focuses on these two data structures of Chan et al. [7]: a simple linear word structure with $\mathcal{O}(\sqrt{n})$ query time, and a linear word structure with $\mathcal{O}(\sqrt{n/w})$ query time. They both outperform other previous exact solutions, and the latter, which is their final structure, essentially combines the former with succinct data structures to encode more precomputed information. However, in practice, constant-time operations over succinct data structures are usually slower than operations over their non-succinct counterparts when all solutions fit in memory [15, 9, 27, 3]. To see whether the use of succinct data structures by Chan et al. improves performance in practice, we compare different tradeoffs of both structures and find that, when the same amount of space is used, the latter indeed provides much faster query support than the former. This is because the query algorithm only performs a constant number of succinct structure operations, and their execution time is dominated by other steps. Encouraged by this observation, we further use succinct structures to swap out more components, and our variant achieves even better time/space tradeoffs. These results are exciting, as they confirm that, when the same space cost is incurred, careful use of succinct data structures may potentially improve query efficiency in practice.

Regarding $(1+\varepsilon)$-approximate range mode, we focus on solutions by Bose et al. [6], Greve et al. [18] and El-Zein et al. [13], as well as a non-succinct version of the $\mathcal{O}(\frac{n}{\varepsilon})$-bit encoding structure of El-Zein et al. which stores the sequences they encode succinctly in plain arrays instead. When setting $\varepsilon = 1/2$, all these data structures provide much faster query time than the best exact solution (which already answers a query in microseconds), and the average approximation ratio is between 1.00001 and 1.02630. They also typically use less than $5n$ words and are thus excellent solutions when high average quality of answers is sufficient. When encoded using compressed bit vectors, the space cost of the succinct encoding structure of El-Zein et al. [13] is only 35.6% to 93.8% of the input array of 32-bit integers (the average space cost is 20.2 bits per symbol among all datasets). Its non-succinct version also stands out with query support at least several times faster than other $O(\frac{n}{\varepsilon})$-word structures while using only slightly more space. When decreasing ε to improve worst-case approximation, query times increase at a logarithmic rate, but space costs tend to be proportional to $1/\varepsilon$.

2 Data Structure for Range Mode

We review the data structures that we will implement. When describing them, we adopt the word RAM model with word size w bits and assume that the input is an array $A[1..n]$ of integers from $\{1, 2, \dots, \Delta\}$, where $\Delta \leq n$. Some solutions use succinct bit vectors as building blocks. These operations are defined over a bit vector $B[1..n]$: $\texttt{rank}_b(i)$, which returns the frequency of bit $b \in \{0, 1\}$ in $B[1..i]$, and $\texttt{select}_b(i)$, which returns the index of the i-th occurrence $b \in \{0, 1\}$ in B. Pătraşcu [28] showed how to represent B in $\lg \binom{n}{t} + \mathcal{O}(\frac{n}{\lg^c n}) \leq n + \mathcal{O}(\frac{n}{\lg^c n})$ bits, where t is the number of 1s in B and c is an arbitrary positive constant, to support `rank` and `select` in $\mathcal{O}(1)$ time. A folklore approach encodes a monotonically increasing sequence of n nonnegative integers upper bounded by u by encoding the difference between consecutive elements in unary and performs `rank` and `select` operations over the concatenated bit vector to compute any entry. This lemma summarizes its bounds.

▶ **Lemma 1** (folklore). *A monotonically increasing sequence of n nonnegative integers upper bounded by u can be represented in $\lg \binom{n+u}{n} + \mathcal{O}(\frac{n+u}{\lg^c (n+u)}) \leq n + u + \mathcal{O}(\frac{n+u}{\lg^c (n+u)})$ bits for any positive constant c such that any entry in the sequence can be computed in $\mathcal{O}(1)$ time.*

2.1 Exact Range Mode in Linear Space and $\mathcal{O}(\sqrt{n} \lg n)$ Time

To design a solution, Krizanc et al. [22] divide A into s blocks each of size $\lceil \frac{n}{s} \rceil$ for an integer parameter $s \in [1, n]$ and precompute an $s \times s$ table S. For any integers $i, j \in [1, s]$, $S[i, j]$ stores the mode of the subarray consisting of blocks $i, i+1, \dots, j$. They also construct, for each integer $\alpha \in \{1, 2, \dots, \Delta\}$, a sorted array Q_α of the positions of the occurrences of α in array A. All these structures occupy $\mathcal{O}(n + s^2)$ words and can be built in $\mathcal{O}(ns)$ time. With them, the mode in $A[a, b]$ can be computed by decomposing $[a, b]$ into up to three subranges: the *span* consists of all the blocks that are entirely contained in $[a, b]$, while the *prefix* and the *suffix* are the two subranges of $[a, b]$ before and after the span, respectively. The mode, c, of the span can be retrieved from S in $\mathcal{O}(1)$ time. The answer to the query is either c, or an element in the prefix or the suffix. We call each of these up to $2\lceil \frac{n}{s} \rceil - 1$ elements a candidate, and the frequency of each candidate $A[x]$ in the query range is computed by a binary search in $Q_{A[x]}$. Then the total query time is $\mathcal{O}((n/s) \lg n)$. Hence, setting $s = \lceil \sqrt{n} \rceil$ yields a linear-word structure with $\mathcal{O}(\sqrt{n} \lg n)$ query time and $\mathcal{O}(n^{3/2})$ preprocessing time.

2.2 Exact Range Mode in Linear Space and $\mathcal{O}(\sqrt{n})$ Time

Chan et al. [7] improved the solution of Krizanc et al. [22] by constructing two additional data structures: A rank array A' in which $A'[i]$ is the index of the entry of $Q_{A[i]}$ that stores i, and an additional $s \times s$ table S' in which $S'[i, j]$ stores the frequency of the mode in blocks $i, i+1, \dots, j$. With the addition of A', we can determine, in constant time, whether $A[i]$ occurs at least q times in $A[i..j]$ for any given i, j and q, by checking if $Q_{A[i]}[A'[i] + q - 1] \leq j$.

The query algorithm again decomposes the query range $[a, b]$ into the span, the prefix and the suffix. Using S and S', we can find the mode, c, of the span and its frequency, f_c, in the span in $\mathcal{O}(1)$ time. This is one candidate of the mode in $A[a, b]$. We then look for the elements in the prefix or the suffix whose frequencies in $A[a, b]$ are greater than f_c: We scan the prefix, and for each element $A[x]$ in it, we find out whether we have seen it before by checking whether $Q_{A[x]}(A'[x] - 1)$ is at least a. If not, we determine whether $A[x]$ occurs more than f_c times in $A[x, b]$ in $O(1)$ time by the approach discussed before. If it does, then $A[x]$ is a candidate, and we compute its frequency in $A[a, b]$ by skipping the next $f_c - 1$ occurrences in $Q_A[x]$ and then continuing the scan of $Q_A[x]$ to find its remaining occurrences in $A[a, b]$. Since the number of times that $A[x]$ occurs in the span is at most f_c, the number of scanned entries of $Q_{A[x]}$ is at most the number of occurrences of $A[x]$ in the prefix and the suffix. Therefore, the frequencies of all candidates can be computed in time linear in the lengths of the prefix and the suffix, which is $O(n/s)$. We scan the suffix in a similar manner, and the candidate with the highest frequency in $A[a, b]$ is the answer. This way the query time is improved to $O(n/s)$, implying a linear-word tradeoff with $\mathcal{O}(\sqrt{n})$ query time.

2.3 Exact Range Mode in Linear Space and $\mathcal{O}(\sqrt{n/w})$ Time

The final solution of Chan et al. [7] divides the input array A into two subsequences B_1 and B_2 as follows: We scan A. If the current element appears at most s times in A, we append it to B_1. Otherwise, it is appended to B_2. Additionally, we define two $2 \times n$ tables $I_\beta[i]$ and $J_\beta[i]$, in which, for every $\beta \in [1, 2]$ and each $i \in [1, n]$, $I_\beta[i]$ (or $J_\beta[i]$) stores the index in B_β of the closest element in A to the left (or right) of $A[i]$ that lies in B_β. Then a range mode query in A can be answered by querying both B_1 and B_2.

A compact version of the structure in Section 2.2 is built over B_1 which consists of Q_α for each α and a compact encoding of S' in $\mathcal{O}(s^2)$ bits, or $\mathcal{O}(s^2/w)$ words. The latter uses Lemma 1 to encode each row of S' in $\mathcal{O}(s)$ bits, as it contains at most s positive integers upper bounded by s. Furthermore, Chan et al. use this structure to infer any entry of S in $\mathcal{O}(n/s)$ time without storing S. This decreases storage to $\mathcal{O}(n + s^2/w)$ words and can answer range mode over B_1 in $\mathcal{O}(n/s)$ time. As for B_2, since each element occurs more than s times, the number of distinct elements, Δ', is at most n/s. They mark every Δ' positions in B_2 and use n words to encode the number of occurrences of each distinct element from the start of B_2 to each marked position, so that the frequency of any element between two marked positions can be computed in $\mathcal{O}(1)$ time. Together with a walk from each endpoint of the query range $[a, b]$ to the nearest marked position inside $[a, b]$, we can compute the frequencies of all Δ' distinct elements in $[a, b]$ in $O(\Delta')$ time, thus answering range mode over B_2. Combing the structures for B_1 and B_2, we have an $\mathcal{O}(n + s^2/w)$-word structure with $\mathcal{O}(n/s)$ query time and $\mathcal{O}(ns + n \lg(n/s))$ preprocessing time. Setting $s = \lceil \sqrt{nw} \rceil$ yields a linear word structure with $\mathcal{O}(\sqrt{n/w})$ query time and $\mathcal{O}(n^{3/2}\sqrt{w})$ preprocessing time.

Remarks. We can further decrease the space overhead by replacing I_β and J_β, where $\beta \in \{1, 2\}$, with a bit vector F, in which $F[i] = 0$ if $A[i]$ is stored in B_1 and $F[i] = 1$ otherwise. Then, the elements in a query range $[a, b]$ are in $B_1[\texttt{rank}_0(a-1)+1, \texttt{rank}_0(b)]$ and $B_2[\texttt{rank}_1(a-1)+1, \texttt{rank}_1(b)]$. This decreases the space cost to $n + o(n) + \mathcal{O}(s^2/w)$ words. We will study both the original approach and our variant experimentally.

2.4 $(1+\varepsilon)$-Approximation in $O(\frac{n}{\varepsilon})$ Words and $\mathcal{O}(\lg\lg n + \lg\frac{1}{\varepsilon})$ Time

To design approximate solutions, Bose et al. [6] first presented a simple approach: For each $i \in \{1, 2, \ldots, n\}$, build a table T_i in which $T_i[r]$ stores the smallest index $j \geq i$ such that $A[j]$ occurs $\lceil (1+\varepsilon)^r \rceil$ times in $A[i,j]$. Given a query range $[a,b]$, they perform a binary search in T_a to find the entry $T_a[k]$ with $T_a[k] \leq b < T_a[k+1]$, and $A[T_a[k]]$ is a $(1+\varepsilon)$-approximate answer. This algorithm uses $\mathcal{O}(\lg\lg n + \lg\frac{1}{\varepsilon})$ time, and the space cost is $\mathcal{O}(\frac{n\lg n}{\varepsilon})$ words.

In a more advanced solution, Bose et al. define two number series, f_{low} and f_{high} by the recurrence $f_{low_1} = f_{high_1} = 1$, $f_{low_{r+1}} = f_{high_r} + 1$ and $f_{high_{r+1}} = \lfloor (1+\varepsilon) f_{low_r} \rfloor + 1$. They then construct a table T_i for each $i = 1, 2, \ldots, n$ as follows. In T_1, an entry $T_1[r]$ stores the smallest index $j \geq i$ such that $A[j]$ occurs f_{high_r} times in $A[1,j]$. To compute an entry $T_i[r]$ for any $i \geq 2$, we first determine whether $T_{i-1}[r]$ occurs at least f_{low_r} times in $A[i, T_{i-1}[r]]$. If it does, then we set $T_i[r] = T_{i-1}[r]$. Otherwise, $T_i[r]$ stores the smallest index $j \geq i$ such that $A[j]$ occurs f_{high_r} times in $A[i,j]$. To answer a query, observe that, the frequency of the mode of any query range $[a,b]$ with $T_a[r] \leq b < T_a[r+1]$ is at most $f_{high_{r+1}} - 1$. Since $A[T_a[r]]$ occurs at least f_{low_r} times in $A[a, T_a[r]] \subseteq A[a,b]$, the ratio of its frequency in $A[a,b]$ to $F_{a,b}$ is at least $f_{low_r}/(f_{high_{r+1}} - 1) = f_{low_r}/\lfloor (1+\varepsilon) f_{low_r} \rfloor \leq 1/(1+\varepsilon)$. [1] Therefore, $A[T_a[r]]$ is a $(1+\varepsilon)$-approximate answer.

Each table has at most $2\lceil \lg_{1+\varepsilon} n \rceil$ entries. To reduce storage costs, Bose et al. view $T_1, T_2, \ldots, T_n$ as n different versions of the same table T, and, to obtain T_i from T_{i-1}, an update is needed for each r with $T_i[r] \neq T_{i-1}[r]$. They proved that the total number of updates over all versions is $\mathcal{O}(n/\varepsilon)$, so these tables can be stored in a persistent binary search tree [11] in $\mathcal{O}(n/\varepsilon)$ words while supporting the search in any table in $\mathcal{O}(\lg(2\lceil \lg_{1+\varepsilon} n \rceil)) = \mathcal{O}(\lg\lg n + \lg\frac{1}{\varepsilon})$ time. They also maintain frequency counters [10] to achieve $O(\frac{n\lg n}{\varepsilon})$ preprocessing time.

2.5 $(1+\varepsilon)$-Approximation in $\mathcal{O}(\frac{n}{\varepsilon})$ Words and $\mathcal{O}(\lg\frac{1}{\varepsilon})$ Time

Let $\varepsilon' = \sqrt{(1+\varepsilon)} - 1$. The structures of Greve et al. [18] consist of the following two parts.

Low Frequency. For each $i = 1, 2, \ldots, n$, we precompute a table Q_i of length $\lceil \frac{1}{\varepsilon'} \rceil$, in which $Q_i[r]$ stores the rightmost index j such that $F_{i,j} = r$. Given a query range $[a,b]$, we perform a binary search to look for the index, s, of the successor of b in Q_a. If s does not exist, then $F_{a,b} > \lceil \frac{1}{\varepsilon'} \rceil$, and we use the structures for high frequencies to compute an answer. Otherwise, $F_{i,j} = s$, and, as observed by El-Zein et al. [12], $A[Q_a[s-1]+1]$ is the answer. [2]

High Frequency. For each $i = 1, 2, \ldots, n$, we precompute a table T_i of length at most $\lceil \lg_{1+\varepsilon'}(\varepsilon' n) \rceil$: For each $r \in [1, \lceil \lg_{1+\varepsilon'}(\varepsilon' n) \rceil$, if $i > 1$ and $F_{i, T_{i-1}[r]} \geq \lceil \frac{1}{\varepsilon'}(1+\varepsilon')^k \rceil + 1$, we set $T_i[r] = T_{i-1}[r]$. Otherwise, $T_i[r]$ stores the rightmost index j with $F_{i,j} \leqslant \lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{k+1} \rceil - 1$. We also build a table L_i for each i; $L_i[r]$ stores $A[i+j-1]$ where j is the smallest positive integer such that $T_{i+j}[r] \neq T_i[r]$. Then, $L_i[r]$ occurs at least $\lceil \frac{1}{\varepsilon'}(1+\varepsilon')^k \rceil + 1$ times in $A[i, T_i[r]]$. With these tables, given a query range $[a,b]$ with $F_{a,b} > \lceil \frac{1}{\varepsilon'} \rceil$, the query algorithm finds the successor, $T_a[s]$, of b in T_a. Then $F_{a,b} \leq F_{a, T_a[s]} \leqslant \lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{s+1} \rceil - 1$ and the frequency of $L_a[s-1]$ in $A[a,b]$ is at least $\lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{s-1} \rceil + 1$, so $L_a[s-1]$ is a $(1+\varepsilon)$-approximate mode.

---

[1] Bose et al. [6] originally defined $f_{high_{r+1}} = \lceil (1+\varepsilon) f_{low_r} \rceil + 1$. However, with their definition, the ratio of the frequency of $A[T_a[r]]$ in $A[a,b]$ to $F_{a,b}$ is at least $f_{low_r}/\lceil (1+\varepsilon) f_{low_r} \rceil$ which is not guaranteed to be at least $1/(1+\varepsilon)$. Therefore, we fix this issue by defining $f_{high_{r+1}} = \lfloor (1+\varepsilon) f_{low_r} \rfloor + 1$ instead.

[2] To return the mode, Greve et al. augments the low frequency structure by storing the mode in $A[i, Q_i[k]]$ with each $Q_i[k]$. This approach does not break asymptotic bounds, but, when implementing this data structure, we do not store these mode elements and use the observation in [12] to save space.

Hence, the total query time is $\mathcal{O}(\lg\lg n + \lg \frac{1}{\varepsilon})$. To speed it up to $\mathcal{O}(\lg \frac{1}{\varepsilon})$, Greve et al. design a 3-approximate structure to narrow down the initial range of binary search over high frequency structures. This structure performs constant-time lowest common ancestor (LCA) queries over a tree of a small $O(\lg\lg n)$ height. Unfortunately, experiments [5] show that, for trees with small heights, structures with constant LCA queries in theory are outperformed by naive approaches. Hence, we implement their solution without this speedup.

Regarding space, the bottleneck is the high frequency structures. Greve et al. view the T_i tables as n version of the same table T as in [6] and bound the total number of updates to T by $\mathcal{O}(\frac{n}{\varepsilon})$. A similar argument apples to L_i's. It is possible to store T_i's and L_i's in a persistent search tree, but this does not allow the speedup. Instead, Greve et al. design an $\mathcal{O}(n/\varepsilon)$-word scheme which samples some table entries and encodes updates between them compactly. It supports the retrieval of an arbitrary entry in constant time. Here we sketch the scheme of storing T_i's; the entries of L_i's can be paired with those of T_i's and stored as additional fields in the same structures. In this scheme, we explicitly store T_l in an array S_l if $l \bmod t = 1$, i.e., we sample and store one out of every t versions of T. Let r be an arbitrary integer in $[1, \lceil \lg_{1+\varepsilon'}(\varepsilon' n)\rceil$. Between two consecutive sampled versions, $T_l[r]$ and $T_{l+t}[r]$, of $T[r]$, there may be updates to $T[r]$. If $r \geq 1 + \lceil \log_{1+\varepsilon'} t\rceil$, then there can only be at most one update to $T[r]$ between versions l and $l+t$. In this case, we store with each sampled entry $T_l[r]$ the next update to $T[r]$. If $r \leq \lceil \log_{1+\varepsilon'} t\rceil$, then, for each sampled entry $T_l[r]$, construct a bit vector of length t with constant-time support for `rank` which uses one bit for each of the next t versions to encode whether an update to $T[r]$ is performed. We also store the (distinct) values used to update $T[r]$ in an array.

Preprocessing. As Greve et al. did not provide information on preprocessing, we also also design an algorithm to construct their data structure in $\mathcal{O}((n\lg n)/\varepsilon)$ time.

The low frequency structure can be constructed in $\mathcal{O}(n/\varepsilon)$ time using frequency counters [10] as was done by Bose et al. [6] to compute similar tables. For the high frequency structure, if we have already computed the content of T_i's and L_i's, we can encode them in time linear in the total number of entries in T_i's and L_i's, and there are $\mathcal{O}(n\lceil \lg_{1+\varepsilon'}(\varepsilon' n)\rceil) = \mathcal{O}((n\lg n)/\varepsilon)$ entries.

What remains is to compute the entries of T_i's and L_i's, and for this we scan A $\lceil \lg_{1+\varepsilon'}(\varepsilon' n)\rceil$ times. In the r-th scan, we compute $T_i[r]$ and $L_i[r]$ for all $i \in [1, n]$ in increasing order of i as follows. We maintain an array $C[1..\Delta]$ of counters; initially all entries of C are $0s$. We use an integer m to keep track of the number of entries of C that are greater than or equal to $\lceil \frac{1}{\varepsilon'}(1+\varepsilon')^k\rceil + 1$; m can be updated each time an entry of C is updated. During the scan, we maintain the following invariant: immediately after computing $T_i[r]$, each counter $C[j]$ stores the number of occurrences of j in $A[i, T_i[r]]$. To compute $T_1[r]$, we retrieve $A[k]$ for $k = 1, 2, ...$, and for each k, we increment $C[A[k]]$. We repeat until $C[A[k]]$ is the first counter in C that reaches $\lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{k+1}\rceil$. This means $A[1..k-1]$ is the longest prefix of A whose mode has frequency $\lceil \frac{1}{\varepsilon'}(1+\varepsilon')^{k+1}\rceil - 1$ in it. Therefore, we set $T_1[r] = k-1$. Then we put the entry $A[k]$ back to the unscanned portion of A by decrementing $C[A[k]]$ and then k. To compute $T_i[r]$ for any $i > 1$, we first decrement $C[A[i-1]]$ and then check whether m is still greater than 0. If it is, then there is at least one element whose frequency in $A[i, T_{i-1}[r]]$ is $\lceil \frac{1}{\varepsilon'}(1+\varepsilon')^k\rceil + 1$, and we set $T_i[r] = T_{i-1}[r]$. Otherwise, we resume the scanning of A to compute $T_i[r]$ using the approach used to compute $T_1[r]$. We also store $A[i-1]$ in $L_r[u], L_r[u+1], \ldots, L_r[r-1]$, where u is the smallest integer such that $T_u[r] = T_{r-1}[r]$. With this implementation, we need to scan the input array A $\mathcal{O}(\lceil \lg_{1+\varepsilon'}(\varepsilon' n)\rceil)$ times, and hence the total preprocessing time is $\mathcal{O}(n\lceil \lg_{1+\varepsilon'}(\varepsilon' n)\rceil) = \mathcal{O}((n\lg n)/\varepsilon)$.

2.6 $(1+\varepsilon)$-Approximation in $\mathcal{O}(\frac{n}{\varepsilon})$ Bits and $\mathcal{O}(\lg \frac{1}{\varepsilon})$ Time

The encoding data structure of El-Zein et al. [12] also consists of two parts: The low frequency structure contains, for each integer $k \in [1, \lceil \frac{1}{\varepsilon} \rceil]$, a table Q_k of length n, in which $Q_k[i]$ stores the rightmost index j such that $F_{i,j} = k$. Q_k can be encoded by Lemma 1 in $2n + o(n)$ bits, so all tables use $\mathcal{O}(\frac{n}{\varepsilon})$ bits. Then, for a range $[a, b]$, we perform a binary search in $Q_1[a], Q_2[a], \ldots, Q_{\lceil \frac{1}{\varepsilon} \rceil}[a]$ to check whether $F_{a,b} \leq \lceil \frac{1}{\varepsilon} \rceil$ and compute the index of a mode if so.

The high frequency structure contains, for each integer $k \in [1, \lfloor \log_{1+\varepsilon}(\varepsilon n) \rfloor]$, a data structure that can find in $\mathcal{O}(1)$ time one of these inequalities that holds for query range $[a, b]$: 1) $F_{a,b} < (1+\varepsilon)^k/\varepsilon$, 2) $F_{a,b} > (1+\varepsilon)^k/\varepsilon$, or 3) $(1+\varepsilon)^{k-1/2}/\varepsilon < F_{a,b} < (1+\varepsilon)^{k+1/2}/\varepsilon$. It finds in case 2 an element that occurs more than $(1+\varepsilon)^k/\varepsilon$ times in $A[a, b]$, and, in case 3, an element that occurs more than $(1+\varepsilon)^{k-1/2}/\varepsilon$ times in $A[a, b]$.

Let $\varepsilon' = \sqrt{1+\varepsilon} - 1$ and $f_j = (\varepsilon'/\varepsilon) \times (1+\varepsilon')^j$. This structure is designed based on four sequences s, s', r and r': For each integer $i \in [0, n/\lceil f_{2k-1} \rceil]$, s_i, the i-the element in s, is $i\lceil f_{2k-1} \rceil + 1$, and r_i is the smallest index such that $F_{s_i, r_i} \geqslant (1+\varepsilon')^{2k}/\varepsilon$. similarly, for each integer $i \in [0, n/\lceil f_{2k} \rceil]$, define $s'_j = i\lceil f_{2k} \rceil + 1$, and r'_j is the smallest index such that $F_{s'_j, r'_j} \geqslant (1+\varepsilon')^{2k+1}/\varepsilon$. Then, given a query range $[a, b]$, El-Zein et al. determine which case applies by comparing b to the entries of r and r' that correspond to the predecessors of a in s and s'. The high frequency structure can be encoded in $\mathcal{O}(\frac{n}{\varepsilon})$ bits by Lemma 1.

To use this trichotomy to answer queries in the high frequency case, perform a binary search in $\mathcal{O}(\lg \lg n + \lg \frac{1}{\varepsilon})$ time to compute a k such that either case 3 applies for the query range, or case 2 applies for k and case 1 applies for $k+1$. The element found by either case 3 or case 2 is a $(1+\varepsilon)$-approximate mode. Finally, to speed up the query time to $\mathcal{O}(\lg \frac{1}{\varepsilon})$, El-Zein et al. designed an $\mathcal{O}(n)$-bit structure that answers 4-approximate range mode queries in constant time, and used it to narrow down the initial range of binary search.

Remarks. The $\mathcal{O}(n)$-bit 4-approximate structure contains a network of fusion trees [16] and is not practical. Hence, our implementation does not include this speedup. El-Zein et al. did not discuss preprocessing, but we can build their structures using frequency counters [10] in $\mathcal{O}(n \lg n/\varepsilon)$ time. Finally, storing all structures in integer arrays without using Lemma 1 would yield a simple $\mathcal{O}(n/\varepsilon)$-word solution, which we also conduct experimental studies on.

3 Experimental Results

3.1 Experimental Setup

Table 1 gives an outline of the data structures we implemented. Among them, the first naive approach, $\texttt{nv}_1$, sorts the elements in the given range to answer a query, while the second one, $\texttt{nv}_2$, scans the elements in the range and uses an array of length Δ to count element frequencies. Four data structures, $\texttt{subsr}_1$, $\texttt{subsr}_2$, `sample` and `succ`, use succinct bit vectors, for which we use the implementation in the succinct data structures library, `sdsl-lite`, of Gog et al. [17]. Two types of bit vectors are used: a plain bit vector, `sdsl::bit_vector` and a compressed bit vector [30], `sdsl::rrr_vector`. To distinguish them, we combine $\texttt{subsr}_1$, $\texttt{subsr}_2$, `sample` or `succ` with superscripts `p` or `c`, e.g., $\texttt{succ}^\texttt{p}$ and $\texttt{succ}^\texttt{c}$, to respectively indicate whether plain or compressed bit vectors are used. Note that, even though $\texttt{subsr}_2^\texttt{c}$ uses compressed bit vectors to encode the table S', a plain bit vector is still used to represent F: we found that, due to the small space cost of F (n bits), compressing it would achieve negligible space savings at the cost of increasing query times by 4.5% to 25%. Finally, for a fair comparison, we modified the implementation of persistent search trees by Jansens [21] to remove the space overhead for generic programming and used it to implement `pst`.

Table 1 The data structures we implemented. The first half of the table present exact solutions, while the second half are $(1+\varepsilon)$-approximate structures with $\mathcal{O}(\lg\lg n + \lg\frac{1}{\varepsilon})$ query time.

abbr.	description
`nv`$_1$,`nv`$_2$	two naive solutions in Section 3.1
`supsr`	$\mathcal{O}(n)$-word, $\mathcal{O}(\sqrt{n}\lg n)$ query time structure for exact range mode in Section 2.1
`sqrt`	$\mathcal{O}(n)$-word, $\mathcal{O}(\sqrt{n})$ query time structure for exact range mode in Section 2.2
`subsr`$_1$	$\mathcal{O}(n)$-word, $\mathcal{O}(\sqrt{n/w})$ query time structure for exact range mode in Section 2.3
`subsr`$_2$	modifying `subsr`$_1$ with more succinct data structures; see the remarks in Section 2.3
`simple`	simple $\mathcal{O}(\frac{n\lg n}{\varepsilon})$-word approximate solution in Section 2.4
`pst`	$\mathcal{O}(\frac{n}{\varepsilon})$-word approximate solution with persistent search trees in Section 2.4
`sample`	$\mathcal{O}(\frac{n}{\varepsilon})$-word approximate solution with sampling in Section 2.5
`tri`	$\mathcal{O}(\frac{n}{\varepsilon})$-word approximate solution with the trichotomy in Section 2.6
`succ`	$\mathcal{O}(\frac{n}{\varepsilon})$-bit approximate solution with the trichotomy in Section 2.6

Table 2 The data sets used in our experiments, each stored as an array of n integers in $[1,\Delta]$.

data	n	Δ	$\lg\Delta$	H_0	
`reviews`	10,000,000	1,367,909	20.38	18.46	books of the first 10^8 book reviews by Amazon customers in 2018 [25]
`IPs`	8,571,089	135,542	17.04	7.96	source IP addresses of DDoS attacks [14]
`words`	6,715,122	127,886	16.96	12.74	words in a text string containing the 100 most frequently downloaded Project Gutenberg [1] e-books in July 2021, with stop words removed
`library`	10,000,000	314,358	18.26	15.75	first 10^8 call numbers in the 2016/17 Seattle Public Library checkout records [23]
`tickets`	10,000,000	79,027	16.27	11.10	street names of the first 10^8 parking tickets issued in New York in 2017 [26]

Five publicly available datasets are used; see Table 2. This table also shows the zeroth-order empirical entropy, H_0, of each dataset. Due to page limit, sometimes we only show figures and tables created for typical datasets, and a full set of tables/figures for all datasets is available in the second author's thesis [24]. To convert raw data into an integer array, we encode each element as an integer in $[1,\Delta]$. To generate a query range $[a,b]$, we adopt the method in [8, 20]: we pick an integer from $[1,n]$ uniformly at random (u.a.r.) and assign it to a, and b is chosen u.a.r. from $[a, a+\lceil\frac{n-a}{K}\rceil]$ for a parameter K. We generate three categories of queries, `large`, `medium` and `small`, by setting $K=1$, 10 and 100, respectively. To justify that this approach of generating queries is appropriate, Appendix C shows additional studies, including those performed over query ranges even smaller than `small` queries.

Our platform is a server with an Intel(R) Xeon(R) Gold 6234 CPU and 128GB of RAM, running Ubuntu 18.04.2. We complied programs using `g++` 7.4.0 with `-O2` flags.

3.2 An Initial Performance Study on Exact Mode

For exact range mode, we initially set $s=\sqrt{n}$ for `supsr` and `sqrt` and set $s=\sqrt{nw}$ for `subsr`$_1$ and `subsr`$_2$ to achieve linear space as in [22, 7]. Tables 3 and 4 present the query time, space usage and construction time of exact query structures. We measure space costs

Table 3 Average time to answer an exact range mode query, measured in micro seconds. Queries are categorized into `small`, `medium` and `large`, and each category has 10^6 queries.

	Query	$\texttt{nv}_1$	$\texttt{nv}_2$	supsr	sqrt	$\texttt{subsr}_1^{\texttt{p}}$	$\texttt{subsr}_1^{\texttt{c}}$	$\texttt{subsr}_2^{\texttt{p}}$	$\texttt{subsr}_2^{\texttt{c}}$
reviews	small	1134	442	338.93	51.70	10.74	11.56	10.74	11.56
	medium	13262	870	366.90	51.00	9.94	10.82	9.94	10.82
	large	144642	5686	363.42	50.85	9.39	10.20	9.39	10.20
IPs	small	532	51	218.75	15.58	3.93	4.40	3.94	4.48
	medium	5938	186	240.03	15.19	3.86	4.37	3.91	4.46
	large	66121	1531	239.35	14.48	3.53	4.01	3.60	4.07
words	small	678	45	298.83	31.49	8.01	8.75	8.14	9.06
	medium	7094	149	334.53	31.27	7.60	8.41	7.82	8.63
	large	73401	1235	349.22	28.28	6.53	7.24	6.67	7.38
library	small	1160	125	384.07	49.54	11.90	13.14	12.13	13.37
	medium	12960	408	422.32	47.11	10.66	11.87	10.71	11.98
	large	132605	3407	444.30	43.68	9.32	10.42	9.40	10.53
tickets	small	990	37	362.47	43.16	9.99	10.67	10.19	10.99
	medium	9931	187	414.76	42.39	9.92	10.65	10.15	10.97
	large	101281	1756	436.93	37.44	8.56	9.35	8.80	9.60

Table 4 Space (bits per symbol) and construction time (minutes) of exact range mode structures.

	Dataset	supsr	sqrt	$\texttt{subsr}_1^{\texttt{p}}$	$\texttt{subsr}_1^{\texttt{c}}$	$\texttt{subsr}_2^{\texttt{p}}$	$\texttt{subsr}_2^{\texttt{c}}$
space	reviews	109.1	173.2	174.3	144.1	174.3	144.1
	IPs	97.5	161.5	332.6	255.9	205.8	129.0
	words	97.8	161.9	329.1	284.1	202.2	157.2
	library	99.0	163.0	315.2	294.5	188.3	167.6
	tickets	96.7	160.8	311.0	289.9	184.1	163.0
construct time	reviews	0.911	0.911	7.205	7.460	7.205	7.460
	IPs	0.695	0.695	1.865	1.867	1.890	1.892
	words	0.438	0.438	2.755	2.760	2.762	2.765
	library	0.806	0.806	5.923	5.933	5.971	5.974
	tickets	0.720	0.720	4.251	4.275	4.756	4.809

in bits per symbol (bps), which is the space usage in bits divided by the length of the input array. Furthermore, the cost of the input array A (32 bps) is included in the space usage of `supsr` and `sqrt` but excluded for $\texttt{subsr}_1$ and $\texttt{subsr}_2$, because `supsr` and `sqrt` scan A when answering a query but $\texttt{subsr}_1$ and $\texttt{subsr}_2$ do not. Nevertheless, the space cost of A is not significant enough to affect our conclusions. These tables show that most data structures have much faster query time than both naive approaches, and `supsr` is the only exception in some cases. Between two naive approaches, $\texttt{nv}_2$ is faster because the number of distinct elements is relatively small compared to input array length.

Before comparing the performance of data structure solutions, we discuss how the distributions of the datasets affect $\texttt{subsr}_1$ and $\texttt{subsr}_2$, for which the array entries are stored in two subsequences B_1 and B_2 (see Section 2.3). Since B_2 stores elements of higher frequency, the lower the entropy of a dataset is, the larger the ratio of the length of B_2 to n tends to be. Indeed, for `reviews`, `words` and `library`, the ratios are 0, 0.037 and 0.010, respectively, while for `IPs` and `tickets`, the ratios are 0.58 and 0.14, respectively, which are higher. These ratios are consistent with the values of H_0 in Table 2. This immediately explains why, for `reviews`, there is no difference in costs between $\texttt{subsr}_1$ and $\texttt{subsr}_2$: These two solutions differ in the components used to map the query range to ranges in B_1 and B_2. Since $|B_2| = 0$ for `reviews`, no mapping is needed.

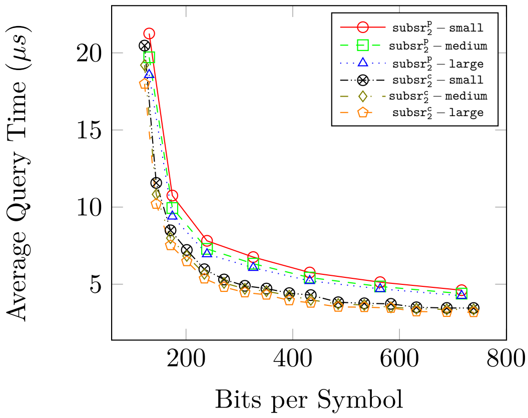
With this in mind, we now compare data structure solutions. We first find that the query time of `sqrt` is 6.0% to 15.3% of that of `supsr`, which is consistent with theoretical bounds. Then we observe that, by using a succinct bit vector to replace multiple arrays, $\texttt{subsr}_2$ saves much space compared to $\texttt{subsr}_1$ over all datasets except `reviews` (which does not require these components as discussed before). At the same time, there is almost no sacrifice in query performance. This is because we only perform `rank` over this bit vector a constant number of times to map query ranges to ranges in B_1 and B_2, and this cost is dominated by subsequent steps which use $O(\sqrt{n/w})$ time. The use of compressed bit vectors in $\texttt{subsr}_1^{\texttt{c}}$ and $\texttt{subsr}_2^{\texttt{c}}$ saves more space, albeit at the cost of a small increase in query time. Theoretical analysis indicates that, when we double s, the query time halves but tables S and S' use four times as much space. Hence, we predict that $\texttt{subsr}_2^{\texttt{p}}$ and $\texttt{subsr}_2^{\texttt{c}}$ achieve the best query-space tradeoffs, and more experiments will be run in Section 3.3 to confirm this.

The sizes of query ranges affect query times greatly for the naive approaches since they either sort or scan the elements in the range. On the other hand, these sizes only affect the query times of `supsr`, `sqrt`, $\texttt{subsr}_1$ and $\texttt{subsr}_2$ slightly. For `sqrt`, $\texttt{subsr}_1$ and $\texttt{subsr}_2$, larger queries even tend to take less time to answer. This is because the query algorithm of `sqrt` (which is also performed over B_2 in $\texttt{subsr}_1$ and $\texttt{subsr}_2$) keeps updating a candidate by a new candidate with higher frequency in the query range, until the mode of the range is found. The initial candidate is the mode of the span of the query. When the query range is larger, the span is also longer, and hence its mode tends to be a better candidate, thus decreasing the query time.

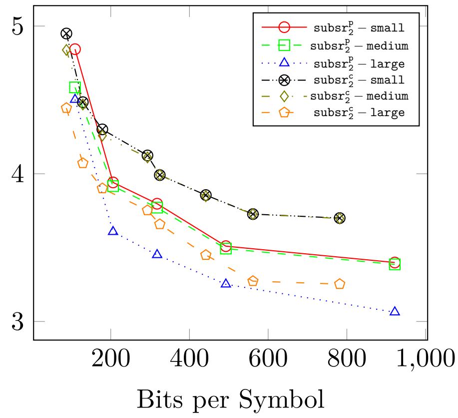
Regarding construction time, observe that the processing times of `supsr` and `sqrt` are about same. For `reviews`, `words`, `library` and `tickets`, the preprocessing time of `supsr` and `sqrt` is 12.2% to 16.9% of that of $\texttt{subsr}_1$ and $\texttt{subsr}_2$. This is because, with the choices of parameters, it takes $\mathcal{O}(n^{3/2}\sqrt{w})$ time to build $\texttt{subsr}_1$ and $\texttt{subsr}_2$, but the preprocessing time of `supsr` and `sqrt` is $\mathcal{O}(n^{3/2})$. However, the difference is much smaller for `IPs`. This is because, when constructing $\texttt{subsr}_1$ and $\texttt{subsr}_2$ for this dataset, 58% of array entries are in B_2, whose query structure can be built in linear time.

3.3 Different Parameter Values

We now choose different values of s to compare these structures thoroughly. First, we compare $\texttt{subsr}_2^{\texttt{p}}$ and $\texttt{subsr}_2^{\texttt{c}}$. The experimental results over `reviews` and `IPs` are shown in Figure 1, while the results over `words`, `library` and `tickets` are shown in Figure 4 in Appendix A. To draw the subfigure for either dataset, we initially set s to be $0.5\sqrt{nw}$ to construct $\texttt{subsr}_2^{\texttt{p}}$ or $\texttt{subsr}_2^{\texttt{c}}$, and each time we increase s by $0.5\sqrt{nw}$ until the space usage exceeds 640 bps. Each point in the figure represents a tradeoff achieved between space and the average query time of a category (`small`, `medium` or `large`) of queries. We then connect the points for the same data structure and query category into a polyline. Hence, over either dataset, we show how the query time changes when more space is used for either data structure using three plotted polylines, one for each query category. In Figure 1 (a), for the same category of queries, the polyline plotted for $\texttt{subsr}_2^{\texttt{p}}$ is always above that for $\texttt{subsr}_2^{\texttt{c}}$. This means, with the same space cost, $\texttt{subsr}_2^{\texttt{c}}$ uses less time to answer a query on average. Hence, $\texttt{subsr}_2^{\texttt{c}}$ outperforms $\texttt{subsr}_2^{\texttt{p}}$ over `reviews`. It is however the opposite for `IPs`, and there is no discernible differences over the three other datasets.



(a) `reviews`.



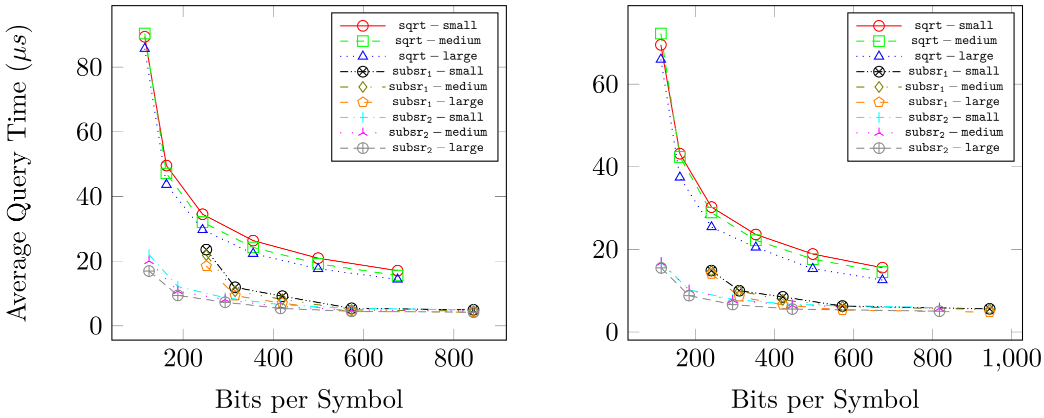
(b) `IPs`.

Figure 1 Time-space tradeoffs of $\texttt{subsr}_2^{\texttt{p}}$ and $\texttt{subsr}_2^{\texttt{c}}$ over `reviews` and `IPs`.

To discuss why they compare differently for different datasets, observe that whether to use plain or compressed bit vectors to encode S' in $\texttt{subsr}_2$ affects the structures built over B_1 only. Furthermore, when s increases, the block size decreases, and more adjacent entries of S' tend to store the same values, making S' more compressible. The dataset `reviews` has the largest entropy, which means the table S' constructed over it is less compressible than that over any other dataset for small s, so the increase of s makes it more compressible rapidly. All arrays entries of `reviews` are also stored in B_1 for all the values of s that we have used, making the compression more sensitive to the choice of the value of s. Hence, for `reviews`, the increase of s improves the compression ratio of $\texttt{subsr}_2^{\texttt{c}}$ at a faster rate than what it does for any other dataset. This allows S' to store much more precomputed information for $\texttt{subsr}_2^{\texttt{c}}$, speeding up the queries despite the increased operation time over compressed bit vectors. Other datasets perform differently when s changes, due to their smaller entropy which also affects the number of array entries distributed into B_1. In the extreme case of `IPs`, $\texttt{subsr}_2^{\texttt{p}}$ performs better, while for the rest, compression does not make a significant or consistent difference. Since it takes less time to construct plain bit vectors, we decide that $\texttt{subsr}_2^{\texttt{p}}$ is also a better solution for `words`, `library` and `tickets`.

We further conducted similar experiments to compare $\texttt{subsr}_1^{\texttt{p}}$ and $\texttt{subsr}_1^{\texttt{c}}$ and made the same observations. See Figure 5 in Appendix A for details. Hence, in the rest of this paper, when the context is clear, $\texttt{subsr}_1$ and $\texttt{subsr}_2$ respectively represent $\texttt{subsr}_1^{\texttt{c}}$ and $\texttt{subsr}_2^{\texttt{c}}$ for `reviews`, while they represent $\texttt{subsr}_1^{\texttt{p}}$ and $\texttt{subsr}_2^{\texttt{p}}$ for all other datasets.

After deciding on bit vector implementations, we compare `sqrt`, $\texttt{subsr}_1$ and $\texttt{subsr}_2$. We continue with the same parameters for $\texttt{subsr}_1$ and $\texttt{subsr}_2$, while for `sqrt`, the initial value of s is $0.5\sqrt{n}$, and each time we increase s by $0.5\sqrt{n}$ until the space usage exceeds 640 bps. Figure 2 shows the results over `library` and `tickets`, and the results are similar for the other three data sets (see Figure 6 in Appendix A), except that for `reviews`, $\texttt{subsr}_1$ and $\texttt{subsr}_2$ have the same performance because no structures are used to map query ranges as discussed before. Our results show that $\texttt{subsr}_1$ and $\texttt{subsr}_2$ have much better query performance than `sqrt` when the same storage costs are incurred. This matches the discussions and prediction in Section 3.2. Between $\texttt{subsr}_1$ and $\texttt{subsr}_2$, for `IPs`, `words`, `library` and `tickets`, our results show that $\texttt{subsr}_2$ achieves better time-space tradeoffs than $\texttt{subsr}_1$ does. The difference is significant for smaller values of s, but as s grows much larger, the plotted lines start to converge. This is because, for large enough s, the space savings by replacing four integer



(a) `library`. **(b)** `tickets`.

Figure 2 Time-space tradeoffs of `sqrt`, `subsr`$_1$ and `subsr`$_2$ over `library` and `tickets`.

arrays with a bit vector is dominated by the $\mathcal{O}(s^2)$-bit cost of S'. Nevertheless, when we require a reasonable space cost for data structures in practice, `subsr`$_2$ still improves `subsr`$_1$ significantly. Finally, we conducted similar experiments to confirm that `sqrt` outperforms `supsr` significantly; see Appendix A. Therefore, we conclude that `subsr`$_2$ preforms the best among all exact solutions.

3.4 Performance of Approximate Range Mode

Tables 5 and 6 present the query time, space usage and construction time of approximate range mode structures when $\varepsilon = 1/2$. Space costs do not include the cost of array A, since these structures can compute the indexes of approximate range modes without accessing A.

To measure accuracy, we compute the approximation ratio of each answer as the frequency of the actual mode in the query range divided by the frequency of the reported approximate mode in the range. Then, for each solution, we compute the average and the maximum of the approximation ratios of the answers for each query category over each dataset. We find that the average ratios range between 1.00001 and 1.02630, and the maximum ratios are closer to 1.5. To see why the average quality of the answers is high, recall that the approximate mode computed is the actual mode of a range having a significant overlap with the query range, so the probability of it being the mode of the query range is high. Since these results are consistent across datasets and query categories, we use Table 7 in Section 3.5 to provide a summary by reporting, for each data structure, the average and maximum ratios over all queries, together with results for some subsequent experiments. Since these structures have slower query support and higher space usage for smaller ε, setting $\varepsilon = 1/2$ is attractive to applications for which a high average approximation ratio is sufficient.

Another phenomenon is that larger queries tend to be faster with approximate solutions. This is because all query algorithms are essentially based on binary searches in lists of possible candidates, and in each list, the farther it is away from the list head, the larger the gaps between the indexes (in A) of two consecutive candidates are, benefiting larger query ranges.

We also observe that the space cost of `pst` can vary greatly among datasets, with the space cost of `library` being about 3.6% of that of `IPs`. Recall that in this solution, we view n different tables as versions of the same table T to store them in a persistent search tree, and each tree node corresponds to an update to the table (the initial version of the table is not

Table 5 Average time to answer an approximate query for $\varepsilon = 1/2$, measured in microseconds. Queries are categorized into `small`, `medium` and `large`, and each category has 10^8 queries.

	Query	`simple`	`pst`	`sample`$^{\mathrm{p}}$	`sample`$^{\mathrm{c}}$	`tri`	`succ`$^{\mathrm{p}}$	`succ`$^{\mathrm{c}}$
`reviews`	`small`	0.098	0.861	0.869	1.016	0.191	1.122	2.970
	`medium`	0.095	0.714	0.598	0.610	0.135	1.009	3.178
	`large`	0.089	0.556	0.440	0.453	0.116	0.864	3.703
`IPs`	`small`	0.110	1.561	0.545	0.796	0.138	1.003	4.003
	`medium`	0.113	1.343	0.358	0.430	0.105	0.696	3.198
	`large`	0.120	1.120	0.285	0.304	0.091	0.581	3.030
`words`	`small`	0.102	0.986	0.809	1.166	0.168	1.126	3.642
	`medium`	0.098	0.780	0.486	0.585	0.127	0.967	3.754
	`large`	0.105	0.546	0.281	0.309	0.095	0.595	2.547
`library`	`small`	0.099	0.760	1.017	1.164	0.200	1.230	3.508
	`medium`	0.099	0.581	0.603	0.629	0.144	1.152	3.809
	`large`	0.106	0.434	0.360	0.370	0.112	0.766	3.023
`tickets`	`small`	0.112	1.072	0.773	1.108	0.172	1.281	3.861
	`medium`	0.109	0.817	0.460	0.585	0.129	0.997	3.371
	`large`	0.119	0.580	0.300	0.327	0.105	0.634	2.669

Table 6 Space (bits per symbol) and construction time (minutes) of approximate structures when $\varepsilon = 1/2$.

	Dataset	`simple`	`pst`	`sample`$^{\mathrm{p}}$	`sample`$^{\mathrm{c}}$	`tri`	`succ`$^{\mathrm{p}}$	`succ`$^{\mathrm{c}}$
space	`reviews`	680.0	100.6	225.4	204.7	291.2	56.9	11.4
	`IPs`	1038.6	1051.5	327.9	311.3	291.5	82.9	30.0
	`words`	787.8	146.3	240.7	220.5	291.4	67.1	21.5
	`library`	769.6	37.6	231.6	210.8	291.3	65.6	13.9
	`tickets`	896.6	115.8	248.2	228.1	291.5	74.2	24.2
construct time	`reviews`	0.084	0.142	0.655	0.668	0.050	0.082	0.085
	`IPs`	0.075	0.172	0.564	0.568	0.031	0.063	0.067
	`words`	0.050	0.082	0.412	0.418	0.018	0.038	0.040
	`library`	0.084	0.136	0.663	0.673	0.042	0.065	0.067
	`tickets`	0.081	0.122	0.648	0.649	0.027	0.068	0.070

stored explicitly since $T[i] = i$ for all $i \in [1, n]$). Thus, we recorded the number of updates to T for each dataset, and it is $1,380,391$ for `reviews`, $10,773,911$ for `IPs`, $1,232,046$ for `words`, $485,498$ for `library` and $1,386,886$ for `tickets`. The difference in the numbers of updates is consistent with the difference in space costs. To see why there is such a difference in updates, recall that an update to T happens when the frequency of a candidate within a certain range $A[i, j]$ drops below a threshold when we increment i. This happens more often when the entropy of the dataset is lower or when the locality of reference is higher, since a lower entropy or higher locality of references means we are more likely to decrease the frequency of this candidate each time we increment i. Indeed, `IPs` has the lowest entropy by Table 2, and since the same subset of IPs occur frequently in a DDoS attack event, it has high locality of reference. This explains the high space cost of `pst` over IPs. On the other hand, `library` has the second highest entropy, and unlike `reviews` whose entropy is higher, due to the limited number of copies that a library has for each book, the borrowing records tend to be less affected by trends such as “best sellers of the month” than Amazon reviews are. This explains the low space usage for `library`. The space cost of `sample` also fluctuates among datasets for similar reasons, but due to sampling, the difference is small.

We now compare approximate structures. Among them, `simple` has the fastest query time due to its simplicity, but its space cost is high. Among more sophisticated, $O(n/\varepsilon)$-word solutions which are not succinct, `tri` stands out as its query time is comparable to that of `simple` (it even beats `simple` in some cases), but its space cost is only 28.1% to 42.8% of that of `simple`. Compared to `pst` and `sample`, it has the smallest worst-case space cost; it is not based on persistence and is thus not sensitive to entropy or locality of reference. On the other hand, for most datasets, `pst` and `sample` provide useful tradeoffs with lower space usage but slower query time, with `pst` especially attractive for datasets of high entropy but low locality of reference such as `library`. Finally, $\texttt{succ}^{\texttt{p}}$ and $\texttt{succ}^{\texttt{c}}$ provide compact solutions; $\texttt{succ}^{\texttt{p}}$ uses $0.89n$ to $1.30n$ words, with query slightly slower than `pst` and `sample` in most cases, while $\texttt{succ}^{\texttt{c}}$ is highly compact, with space costs only 35.6% to 93.8% of the array of 32-bit integers (in most cases, the space cost is closer to the lower end, and the average is 20.2 bps over all datasets), while the query time is 265% to 522% of that of $\texttt{succ}^{\texttt{p}}$. Regarding preprocessing, we observed that the construction of `tri` is the fastest while that of `sample` is the slowest.

3.5 Different Values of ε and Comparisons to Exact Queries Structures

We further conduct experiments by setting ε to $1/4$, $1/8$ and $1/16$. Table 7 shows that average approximation ratios decrease when ε decreases, though they are already close to 1 for $\varepsilon = 1/2$. Maximum approximation ratios are close to $1 + \varepsilon$.

Table 7 Average and max approximation ratios for different ε.

ε	Average				Maximum			
	`simple`	`pst`	`sample`	`tri/succ`	`simple`	`pst`	`sample`	`tri/succ`
1/2	1.00644	1.00464	1.00644	1.00192	1.49977	1.5	1.48879	1.47826
1/4	1.00218	1.00164	1.00188	1.00085	1.24952	1.25	1.24701	1.25
1/8	1.00075	1.00068	1.00055	1.00019	1.12474	1.125	1.12148	1.11765
1/16	1.00020	1.00017	1.00016	1.00006	1.06240	1.0625	1.06107	1.05882

We also measure the performance of each solution for different ε. Tables 8 and 9 present the performance and accuracy of approximate query structures for different values of ε over the `words` dataset. We observe that query times increase slowly as ε decreases, fitting the growth of the function of $\lg \frac{1}{\varepsilon} + \lg\lg n$. The space costs, however, grows at a much faster rate, proportional to $1/\varepsilon$. For different values of ε, how different solutions compare to each other is similar to the case where $\varepsilon = 1/2$. The main notable difference is that, due to persistence or compression, the space costs of `pst`, `sample`, and $\texttt{succ}^{\texttt{c}}$ grow more slowly than other data structures.

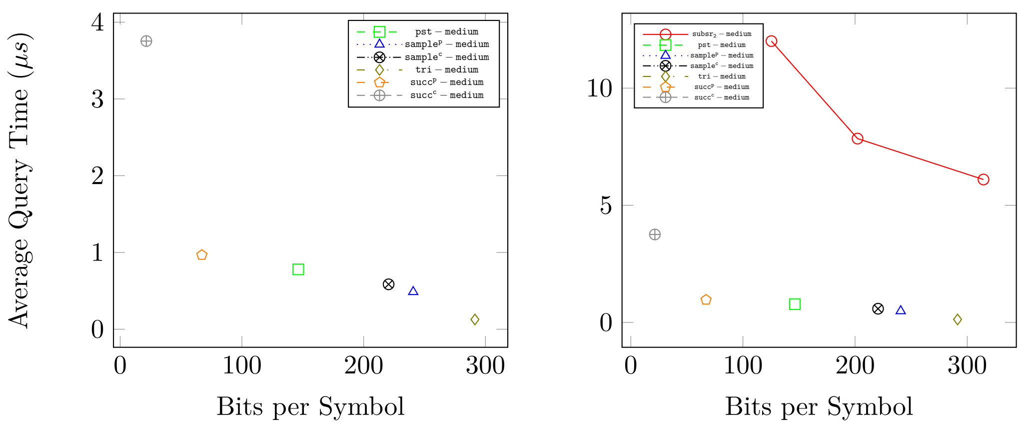
Finally, for $\varepsilon = 1/2$, we plotted figures to compare approximate structures to the best exact structure, `subsr2`. Due to its high space costs, `simple` is not included. To better compare approximate solutions, we plot a subfigure without `subsr2`, before plotting another one with `subsr2`. As a typical example, Figure 3 shows the tradeoffs achieved for medium queries over `words`, while Figure 8 in Appendix B shows the tradeoffs for all three types of queries over `reviews`. From them, we can tell approximate structures outperform exact structures greatly, making them suitable for applications that require good average approximations. They still achieve better time/space tradeoffs over `subsr2` for $\varepsilon = 1/4$, but may lose the appeals when we keep decreasing ε due to the increase in space costs.

Table 8 Average time to answer an approximate query over the `words` datasets for $\varepsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$, measured in microseconds. Queries are categorized into `small`, `medium` and `large`, and each category has 10^8 queries.

ε	Query	simple	pst	sample^p	sample^c	tri	succ^p	succ^c
1/2	small	0.102	0.986	0.809	1.166	0.168	1.126	3.642
	medium	0.098	0.780	0.486	0.585	0.127	0.967	3.754
	large	0.105	0.546	0.281	0.309	0.095	0.595	2.547
1/4	small	0.122	1.178	1.069	1.486	0.248	1.568	4.537
	medium	0.119	0.924	0.738	0.841	0.184	1.371	4.597
	large	0.119	0.639	0.357	0.386	0.142	0.906	3.512
1/8	small	0.148	1.637	1.277	1.809	0.349	2.231	5.778
	medium	0.138	1.586	1.253	1.280	0.269	2.128	5.940
	large	0.133	1.090	0.543	0.563	0.194	1.402	4.598
1/16	small	0.178	1.704	1.439	2.061	0.468	2.993	6.849
	medium	0.170	1.479	1.459	1.690	0.383	3.104	7.230
	large	0.161	0.935	1.025	1.077	0.261	2.095	5.894

Table 9 Space (bits per symbol) and construction time (minutes) when answering approximate queries over the `words` datasets for $\varepsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$.

	ε	simple	pst	sample^p	sample^c	tri	succ^p	succ^c
space	1/2	787.8	146.3	240.7	220.5	291.4	67.1	21.5
	1/4	1418.9	264.7	393.0	352.4	547.8	117.4	35.9
	1/8	2677.9	657.9	704.3	619.5	1063.9	212.2	62.6
	1/16	5185.2	753.6	1337.7	1157.9	2074.5	389.5	111.7
construc -tion	1/2	0.050	0.082	0.412	0.418	0.018	0.038	0.040
	1/4	0.091	0.166	0.744	0.746	0.032	0.066	0.077
	1/8	0.171	0.318	1.610	1.636	0.058	0.148	0.150
	1/16	0.335	0.554	3.224	3.247	0.108	0.229	0.238



(a) `words` – `medium` without `subsr`$_2$. **(b)** `words` – `medium` with `subsr`$_2$.

Figure 3 Time-space tradeoffs of different data structures for medium queries over `words`.

References

1 Project Gutenberg. (n.d.), retrieved in July 2021. Available from `https://www.gutenberg.org/`.

2 Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 522–539. SIAM, 2021. `doi:10.1137/1.9781611976465.32`.

3 Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. Succinct trees in practice. In Guy E. Blelloch and Dan Halperin, editors, *Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments, ALENEX 2010, Austin, Texas, USA, January 16, 2010*, pages 84–97. SIAM, 2010. `doi:10.1137/1.9781611972900.9`.

4 Nikhil Bansal and Ryan Williams. Regularity lemmas and combinatorial algorithms. *Theory of Computing*, 8:69–94, 2012.

5 Michael A Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

6 Prosenjit Bose, Evangelos Kranakis, Pat Morin, and Yihui Tang. Approximate range mode and range median queries. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 377–388. Springer, 2005.

7 Timothy M. Chan, Stephane Durocher, Kasper Green Larsen, Jason Morrison, and Bryan T. Wilkinson. Linear-space data structures for range mode query in arrays. *Theory of Computing Systems*, 55(4):719–741, March 2013.

8 Francisco Claude, J Ian Munro, and Patrick K Nicholson. Range queries over untangled chains. In *International Symposium on String Processing and Information Retrieval*, pages 82–93. Springer, 2010.

9 O'Neil Delpratt, Naila Rahman, and Rajeev Raman. Engineering the LOUDS succinct tree representation. In Carme Àlvarez and Maria J. Serna, editors, *Experimental Algorithms, 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006, Proceedings*, volume 4007 of *Lecture Notes in Computer Science*, pages 134–145. Springer, 2006. `doi:10.1007/11764298_12`.

10 Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. Frequency estimation of internet packet streams with limited space. In *European Symposium on Algorithms*, pages 348–360. Springer, 2002.

11 James R Driscoll, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. Making data structures persistent. *Journal of computer and system sciences*, 38(1):86–124, 1989.

12 Hicham El-Zein, Meng He, J Ian Munro, Yakov Nekrich, and Bryce Sandlund. On approximate range mode and range selection. In *30th International Symposium on Algorithms and Computation (ISAAC 2019)*, volume 149, page 57. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

13 Hicham El-Zein, Meng He, J Ian Munro, and Bryce Sandlund. Improved time and space bounds for dynamic range mode. In *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112, page 25. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

14 Derya Erhan. Boğaziçi university DDoS dataset, 2019. Available from `https://dx.doi.org/10.21227/45m9-9p82`.

15 Luca Foschini, Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Trans. Algorithms*, 2(4):611–639, 2006. `doi:10.1145/1198513.1198521`.

16 Michael L Fredman and Dan E Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the twenty-second annual ACM symposium on Theory of Computing*, pages 1–7, 1990.

17 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014. `doi:10.1007/978-3-319-07959-2_28`.

18 Mark Greve, Allan Grønlund Jørgensen, Kasper Dalgaard Larsen, and Jakob Truelsen. Cell probe lower bounds and approximations for range mode. In *International Colloquium on Automata, Languages, and Programming*, pages 605–616. Springer, 2010.

19 Yuzhou Gu, Adam Polak, Virginia Vassilevska Williams, and Yinzhan Xu. Faster monotone min-plus product, range mode, and single source replacement paths. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 75:1–75:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ICALP.2021.75`.

20 Meng He and Serikzhan Kazi. Path query data structures in practice. In *18th International Symposium on Experimental Algorithms*, volume 160, pages 27:1–27:16, 2020.

21 D. Jansens. *Persistent Binary Search Trees.* `https://cglab.ca/~dana/pbst/`.

22 Danny Krizanc, Pat Morin, and Michiel H. M. Smid. Range mode and range median queries on lists and trees. *Nordic Journal of Computing*, 12(1):1–17, 2005.

23 Seattle Public Library. Seattle library checkout records, 2017. Available from `https://www.kaggle.com/seattle-public-library/seattle-library-checkout-records`.

24 Zhen Liu. Exact and approximate range mode query data structures in practice. Master's thesis, Dalhousie University, 2023. URL: `http://hdl.handle.net/10222/81772`.

25 Jianmo Ni, Jiacheng Li, and Julian McAuley. Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 188–197, 2019.

26 City of New York. NYC parking tickets, 2017. Available from `https://www.kaggle.com/datasets/new-york-city/nyc-parking-tickets`.

27 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*. SIAM, 2007. `doi:10.1137/1.9781611972870.6`.

28 Mihai Patrascu. Succincter. In *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 305–313, 2008.

29 Holger Petersen and Szymon Grabowski. Range mode and range median queries in constant time and sub-quadratic space. *Information Processing Letters*, 109(4):225–228, 2009.

30 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)*, 3(4):43–es, 2007.

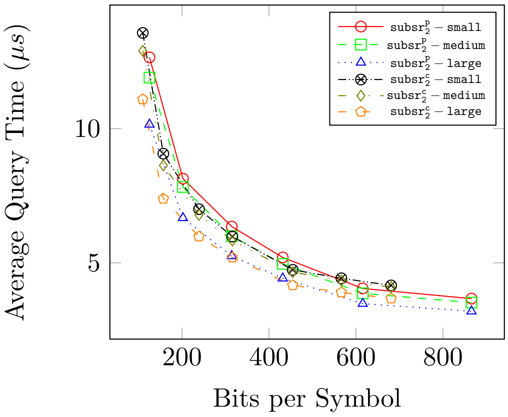
31 Bryce Sandlund and Yinzhan Xu. Faster dynamic range mode. In *47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

32 Virginia Vassilevska Williams and Yinzhan Xu. Truly subcubic min-plus product for less structured matrices, with applications. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 12–29. SIAM, 2020.

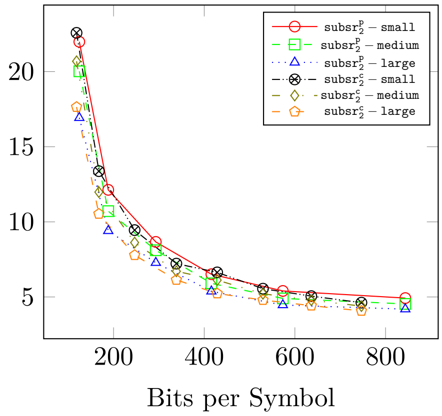
A Details Omitted from Section 3.3

Figures 4, 5 and 6 are omitted figures from Section 3.3.

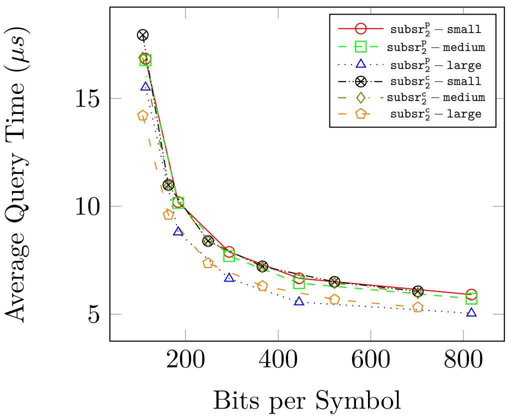
We also compare the time-space tradeoffs that can be achieved by `supsr` and `sqrt` with different parameters. Figure 7 shows our experimental results over `reviews` and `IPs`, in which a subfigure is used for either dataset. The results for other datasets are similar. To draw each subfigure, we construct `supsr` (and similarly `sqrt`) over each dataset for different values of s. The initial value of s is $0.5\sqrt{n}$, and each time we increase s by $0.5\sqrt{n}$ until the space usage of the data structure exceeds 640 bits per symbol. In Figure 7, our experimental study shows that `sqrt` use less query time than `supsr` when these data structures use the same space. Therefore, `sqrt` outperforms `supsr`.



(a) `words`.

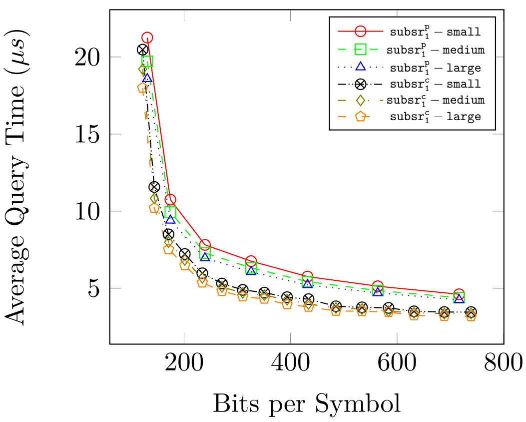


(b) `library`.

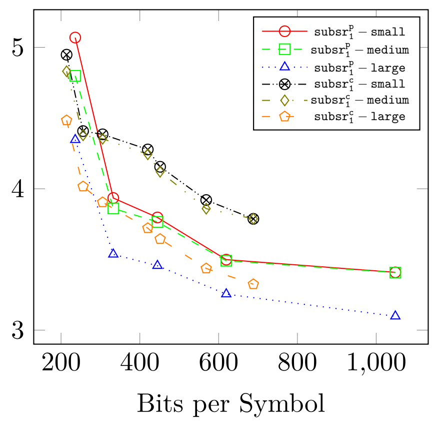


(c) `tickets`.

Figure 4 Time-space tradeoffs achieved by subsr$_2^p$ and subsr$_2^c$ over `words`, `library` and `tickets`.

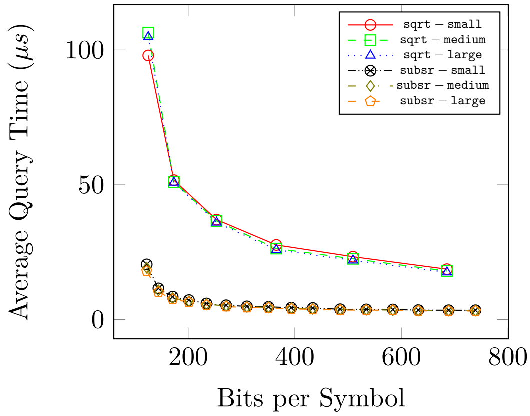


(a) `reviews`.

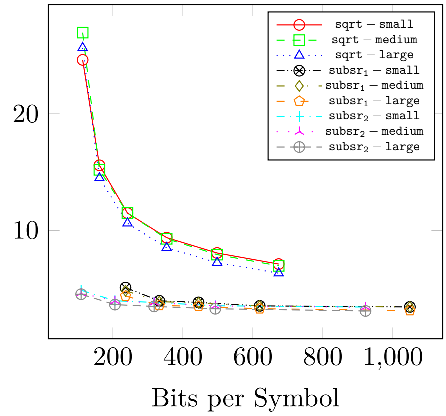


(b) `IPs`.

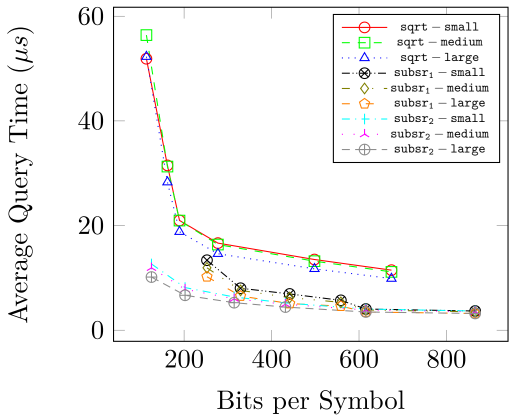
Figure 5 Time-space tradeoffs achieved by subsr$_1^p$ and subsr$_1^c$ over `reviews` and `IPs`



(a) `reviews`.

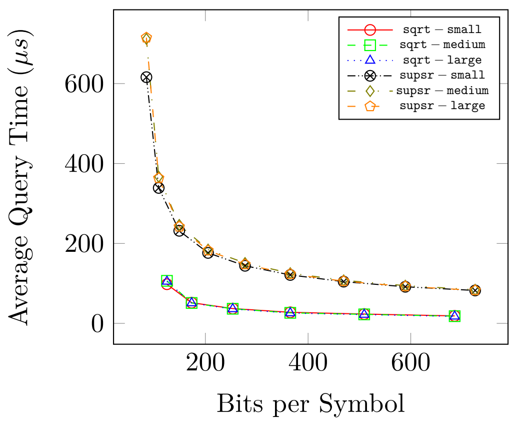


(b) `IPs`.

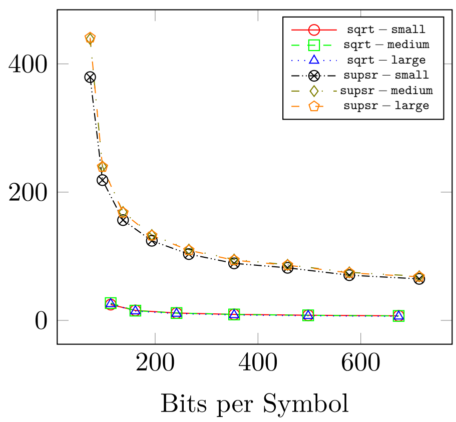


(c) `words`.

Figure 6 Time-space tradeoffs achieved by `sqrt`, `subsr`$_1$ and `subsr`$_2$ over `reviews`, `IPs` and `words`.



(a) `reviews`.



(b) `IPs`.

Figure 7 Time-space tradeoffs achieved by `supsr` and `sqrt` over `reviews` and `IPs`.

B Comparing exact range mode and approximate range mode data structures on `reviews`

Figure 8 compares tradeoffs achieved by exact and approximate range mode structures over `reviews`. Due to its high space costs, the figures do not show `simple`. We also omit some tradeoffs with low space cost that can be achieved using `subsr`2, because their query times are so large that, with them, it would not be possible to tell how other tradeoffs compare to each other in the same figure.

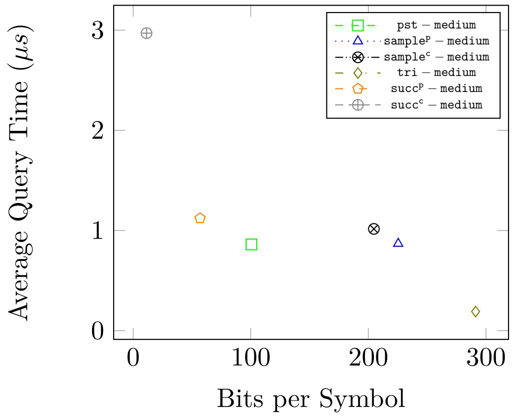
C Even Smaller Queries Ranges

In the experimental studies reported in Section 3, we adopt the method in [8, 20] to generate `small`, `medium` and `large` queries. To confirm whether this is appropriate for our experimental studies, we further perform additional studies using query ranges of sizes 10^1, 10^2, 10^3, 10^4 and 10^5, most of which are even smaller than the average size of our `small` queries, to see whether exact and approximate solutions still compare similarly for these query ranges. To run these experiments, for each $i \in \{1, 2, 3, 4, 5\}$, we generate 10^6 query ranges of size 10^i by choosing the starting positions of the ranges uniformly at random.

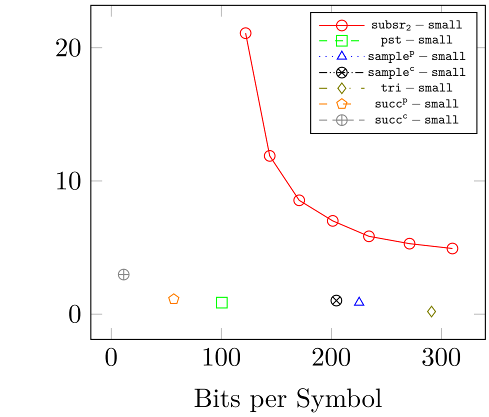
Exact query structures can achieve different time-space tradeoffs when setting the parameter s to different values. For a fair comparison, we binary search on s to make space costs as close to 300bps as possible. For example, for `words`, we set s to 4613, 16792 and 29748 for `sqrt`, $\texttt{subsr}_1$ and $\texttt{subsr}_2$ to achieve space costs of 300.7bps, 300.3bps and 300.3bps, respectively. For `library`, we set s to 5614, 22853 and 38859 for `sqrt`, $\texttt{subsr}_1$ and $\texttt{subsr}_2$ to achieve space costs of 300.7bps, 297.3bps and 300.0bps, respectively. We also include $\texttt{nv}_1$ to find out when data structure solutions outperform this naive solution. The other naive solution, $\texttt{nv}_2$, is not included; since it uses an array of size Δ, smaller query sizes will make it compare more poorly to others. Figure 9 presents our experimental results on `words` and `library`, and the results on other datasets are similar. These figures show that, for small query sizes under 100, the query times of all solutions including $\texttt{nv}_1$ are close, but after query sizes exceed 100 or so, data structure solutions start to outperform $\texttt{nv}_1$ significantly, and they compare to each other similarly as they did during the studies in Section 3.3. We also observe that, when query sizes increase, all data structure query times first increase due to the scan of more entries of A. Later, when query ranges are big enough (starting from somewhere between 10^3 and 10^4) to include multiple blocks of A, the table S is used, so the query algorithms need not scan more array entries. Instead, the query times decrease slowly when query sizes increase due to the reasons discussed in Section 3.2.

Figure 10 shows the results for approximate range mode structures over `words` and `library`, and the results on other datasets are similar. It again shows that the conclusions in Section 3.4 apply to these query sizes. A new observation is that the query times of `sample` and `succ` decrease rapidly when query sizes drops below 10^3 and 10^2, respectively. This is because each of these solutions consists of a low frequency structure and a high frequency structure, and when query sizes are smaller, it is more likely that only the former is used which has much faster query time than the latter.

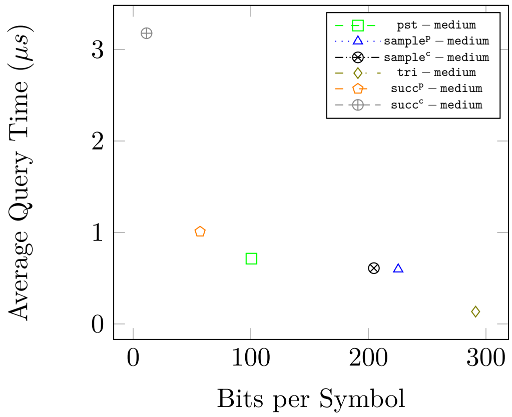
These experiments show that, when query sizes are big enough to justify the use of data structures (instead of merely using a naive solution), the same conclusions in Section 3 apply here. Hence, we conclude that it is appropriate to generate `small`, `medium` and `large` queries and use them throughout our studies.



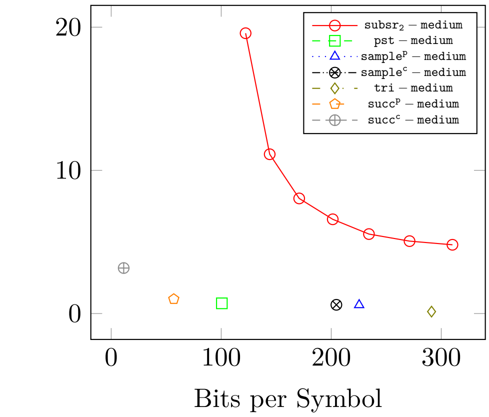
(a) `reviews` – `small` without $\mathtt{subsr}_2$.



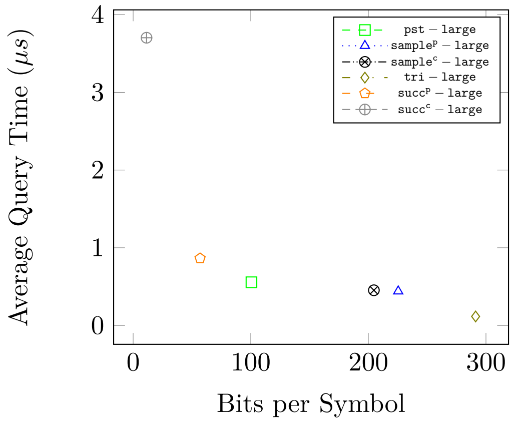
(b) `reviews` – `small` with $\mathtt{subsr}_2$.



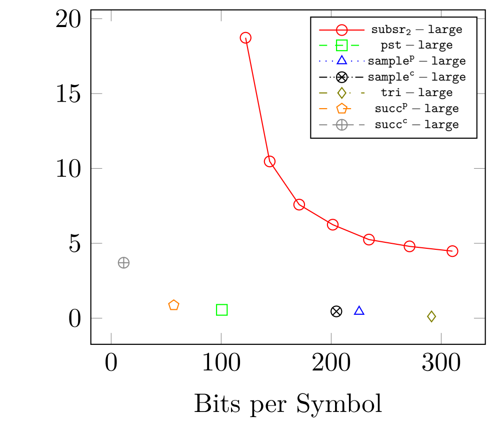
(c) `reviews` – `medium` without $\mathtt{subsr}_2$.



(d) `reviews` – `medium` with $\mathtt{subsr}_2$.

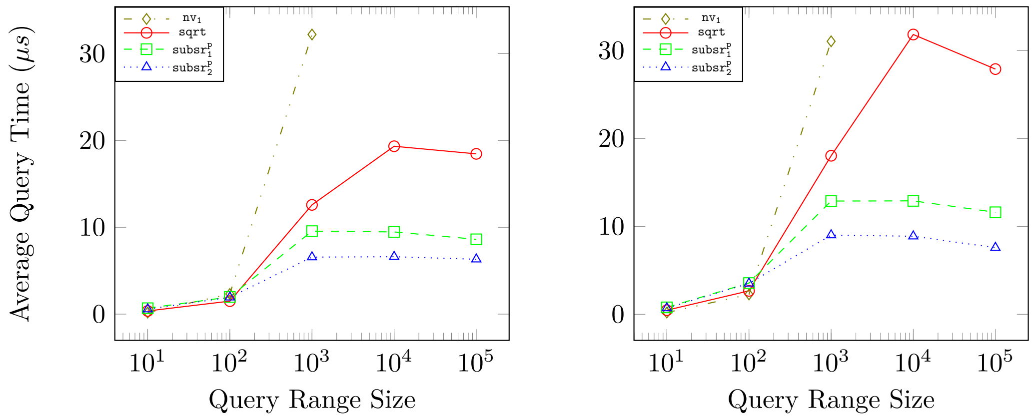


(e) `reviews` – `large` without $\mathtt{subsr}_2$.



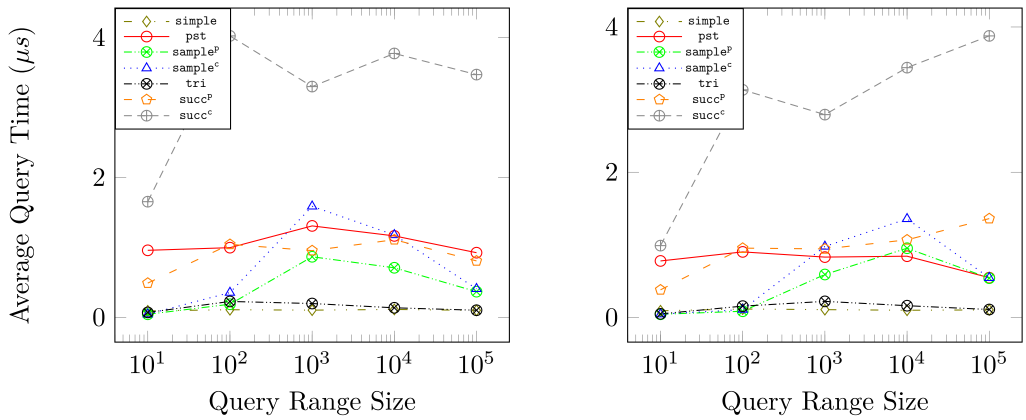
(f) `reviews` – `large` with $\mathtt{subsr}_2$.

Figure 8 Time-space tradeoffs achieved by $\mathtt{subsr}_2$, `pst`, $\mathtt{sample}^{\mathtt{P}}$, $\mathtt{sample}^{\mathtt{c}}$, `tri`, $\mathtt{succ}^{\mathtt{P}}$, and $\mathtt{succ}^{\mathtt{c}}$ on `reviews`.



(a) `words`. **(b)** `library`.

Figure 9 Query time of exact range mode query, for query ranges of sizes from 10^1 to 10^5.



(a) `words`. **(b)** `library`.

Figure 10 Query time of approximate range mode, for query ranges of sizes from 10^1 to 10^5.