Approximate query processing over static sets and sliding windows

Ran Ben Basat a, Seungbum Jo b, Srinivasa Rao Satti c,*, Shubham Ugare d

a University College London, UK
b Chungbuk National University, South Korea
c Norwegian University of Science and Technology, Norway
d IIT Guwahati, India

A R T I C L E   I N F O

Article history:
Received 15 March 2021
Received in revised form 8 June 2021
Accepted 13 June 2021
Available online 17 June 2021
Communicated by M. Zehavi

Keywords:
Streaming Algorithms
Sliding window
Lower bounds

A B S T R A C T

Indexing of static and dynamic sets is fundamental to a large set of applications such as
information retrieval and caching. Denoting the characteristic vector of the set by $B$, we
consider the problem of encoding sets and multisets to support approximate versions of the
operations rank$(i)$ (i.e., computing $\sum_{j\leq i} B[j]$) and select$(i)$ (i.e., finding min\{$p \mid \text{rank}(p) \geq i$\}) queries. We study multiple types of approximations (allowing an error in the query or
the result) and present lower bounds and succinct data structures for several variants
of the problem. We also extend our model to sliding windows, in which we process a stream
of elements and compute suffix sums. This is a generalization of the window summation
problem that allows the user to specify the window size at query time. Here, we provide
an algorithm that supports updates and queries in constant time while requiring just $(1 + \epsilon)$ factor more space than the fixed-window summation algorithms.

© 2021 The Author(s). Published by Elsevier B.V. This is an open access article under the
CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

Given a bit-string $B[1\ldots n]$ of size $n$, one of the fundamental and well-known problems proposed by Jacobson [16], is
to construct a space-efficient data structure which can answer rank and select queries on $B$ efficiently. For $b \in \{0, 1\}$, these
queries are defined as follows.

- rank$b(i, B)$: returns the number of $b$’s in $B[1\ldots i]$.
- select$b(i, B)$: returns the position of the $i$-th $b$ in $B$.

A bit vector supporting a subset of these operations is one of the basic building blocks in the design of various succinct
data structures. Supporting these operations in constant time, with close to the optimal amount of space, both theoretically
and practically, has received a wide range of attention [17,19–21,24]. Some of these results also explore trade-offs that allow
more query time while reducing the space.

* Preliminary version of these results have appeared in the proceedings of the 29th International Symposium on Algorithms and Computation (ISAAC 18) [7].
* Corresponding author.
E-mail addresses: r benbasat@cs.ucl.ac.uk (R.B. Basat), sbjo@chungbuk.ac.kr (S. Jo), srinivasa.r.satti@ntnu.no (S.R. Satti), ugare.dipak@iitg.ac.in (S. Ugare).

https://doi.org/10.1016/j.tcs.2021.06.015
0304-3975/© 2021 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license
(http://creativecommons.org/licenses/by/4.0/).
We also consider related problems in the streaming model, where a quasi-infinite sequence of integers arrives, and our algorithms need to support the operation of appending a new item to the end of the stream. For $i \in [1, \ldots, n]$, let $S_i$ be the sum of the last $i$ integers. Here, $n$ is the maximal suffix size we support queries for. For streaming, we consider processing a stream of elements, and answering two types of queries, suffix sum (ss) and inverse suffix sum (iss), defined as:

- $ss(i, n)$: returns $S_i$ for any $1 \leq i \leq n$.
- $iss(i, n)$: returns the smallest $j$, $1 \leq j \leq n$, such that $ss(j, n) \geq i$.

In this paper, our goal is to obtain space efficient data structures for supporting a few relaxations of these queries efficiently using an amount of space below the theoretical minimum (for the relaxed versions), ideally. To this end, we define approximate versions of rank and select queries, and propose data structures for answering approximate rank and select queries on multisets and bit-strings. We consider the following approximate queries with an additive error $\delta > 0$.

- $rank_A(\delta i, \delta j, \delta B)$: returns any value $r$ which satisfies $rank_b(i - \delta, B) < r \leq rank_b(i, B)$. If $rank_b(i - \delta, B) = rank_b(i, B)$, then $rank_A(\delta i, \delta j, \delta B) = rank_b(i, B)$.
- $drank_A(\delta i, \delta j, \delta B)$: returns any value $r$ which satisfies $rank_b(i, B) - \delta < r \leq rank_b(i, B)$.
- $select_A(\delta i, \delta j, \delta B)$: returns any position $p$ which satisfies $select_b(i - \delta, B) < p \leq select_b(i, B)$.
- $dsselect_A(\delta i, \delta j, \delta B)$: returns any position $p$ which satisfies $select_b(i, B) - \delta < p \leq select_b(i, B)$.
- $ss_A(\delta i, \delta j, \delta B)$: returns any value $r$ which satisfies $ss(i, n) - \delta < r \leq ss(i, n)$.
- $iss_A(\delta i, \delta j, \delta B)$: returns any value $r$ which satisfies $iss(i - \delta, n) < r \leq iss(i, n)$.

We propose data structures for supporting approximate rank and select queries on bit-strings efficiently. Our data structures uses less space than that is required to answer the exact queries, and most of the data structures use optimal space. We also propose a data structure for supporting $ss_A$ and $iss_A$ queries on binary streams while supporting updates efficiently. Finally, we extend some of these results to the case of larger alphabets. For all these results, we assume the standard word-RAM model [18] with word size $\Theta(\lg n)$ if it is not explicitly mentioned.

1.1. Previous work

Rank and Select over bit-strings. Given a bit-string $B$ of size $n$, it is clear that at least $n$ bits are necessary to support rank and select queries on $B$ (since the bit vector can be reconstructed by using the answers to these queries). Jacobson [16] proposed a data structure for answering rank queries on $B$ in constant time using $n + o(n)$ bits. Clark and Munro [8] extended it to support both rank and select queries in constant time with $n + o(n)$ bits. For the case when there are $m$ 1’s in $B$, at least $B(n, m) = \lg \binom{n}{m}$ bits are necessary to support rank and select on $B$. Raman et al. [24] proposed a data structure that supports both operations in constant time while using $\frac{B(n, m) + o(n)}{\lg m}$ bits. Golynski et al. gave an asymptotically optimal time-space trade-off for supporting rank and select queries on $B$ [14]. A slightly related problem of approximate color counting has been considered in El-Zein et al. [10].

Algorithms that Sum over Sliding Windows. A natural generalization of the static case is answering queries with respect to a sliding window over a data stream. The sliding window model was extensively studied for multiple problems including summing [4,9], heavy hitters [25], Bloom filters [1] and counting distinct elements [11]. Our ss queries for streaming are a generalization of the problem of summing over sliding windows. That is, window summation is a special case of the suffix sum problem where the algorithm is always asked for the sum of the last $i \leq n$ elements. Approximating the sum of the last $n$ elements over a stream of integers in $[0, 1, \ldots, \ell]$, was first introduced by Datar et al. [9]. They proposed a $(1 + \epsilon)$ multiplicative approximation algorithm that uses $O\left(\epsilon^{-1} \left(\lg^2 n + \lg \ell \cdot (\lg n + \lg \ell)\right)\right)$ bits and operates in $O\left(\lg \ell / \lg n\right)$ amortized time, or $O\left(\lg (\ell \cdot n)\right)$ worst case time. In [12], Gibbons and Tirthapura presented a $(1 + \epsilon)$ multiplicative approximation algorithm that operates in constant worst case time while using similar space for $\ell = n^{O(1)}$. Ben-Basat et al. [4] studied the potential memory savings one can get by replacing the $(1 + \epsilon)$ multiplicative guarantee with a $\delta$ additive approximation. They showed that $\Theta(\ell \cdot n / \delta + \lg n)$ bits are required and sufficient. Recently, [3] showed the potential memory saving of a bi-criteria approximation, which allows error in both the sum and the time axis, for sliding window summation. The data structure of [6] looks at a generalization of the ssA queries to general alphabet, where at query time we also receive an element $x$ and return an estimate for the frequency of $x$ in the last $i$ elements.

It is worth mentioning that these data structures do allow computing the sum of a window whose size is given at the query time. Alas, the query time will be slower as they do not keep aggregates that allow quick computation. Specifically, we can compute a $(1 + \epsilon)$ multiplicative approximation in $O(\epsilon^{-1} \lg (tn))$ time using the data structures of [9] and [12]. We can also use the data structure of [4] for an additive approximation of $\delta$ in $O(n(\ell / \delta))$ time.

1 $B(n, m)$ bits are the information-theoretic lower bound on space for storing a subset of size $m < n$ from the universe $\{1, 2, \ldots, n\}$. Note that $B(n, m) = m \lg (en/m) - O(\lg m) - \Theta(m^2/n)$ [24].
1.2. Our results

In this paper, we obtain the following results for the approximate rank, select, ss and iss queries with additive error. Let $B$ be a bit-string of size $n$.

1. rank and select queries with additive error $\delta$:

- We first show that $\lceil n/\delta \rceil$ bits are necessary for answering $\text{rank}_A$ and $\text{select}_A$ queries on $B$ and propose a $(\lceil n/\delta \rceil + o(n/\delta))$-bit data structure that supports $\text{rank}_A$ and $\text{select}_A$ queries on $B$ in constant time. For the case when there are $m$ 1’s in $B$, we show that $B(\lceil n/\delta \rceil, \lceil m/\delta \rceil)$ bits are necessary for answering $\text{rank}_A$ and $\text{select}_A$ queries on $B$, and obtain $B(\lceil n/\delta \rceil, \lceil m/\delta \rceil) + o(n/\delta)$-bit data structure that supports $\text{rank}_A$ and $\text{select}_A$ queries on $B$ in constant time.

- We also obtain data structures for answering approximate rank and select queries on a given multiset $S$ from the universe $U = \{1, 2, \ldots, n\}$ with additive error $\delta$, where $\text{rank}(i, S)$ query returns the value $|\{j \in S : j \leq i\}|$, and $\text{select}(i, S)$ query returns the $i$-th smallest element in $S$. We consider two different cases: (i) rankA, $\text{rank}_A$, $\text{select}_A$, and $\text{select}_A$ queries when $|S| = m$, and (ii) $\text{rank}_A$ and $\text{select}_A$ queries when the frequency each element in $S$ is at most $\ell$. For case (ii), we first show that at least $\lceil n/ \lfloor \ell/\delta \rfloor \rceil \log \max(\lfloor \ell/\delta \rfloor, 1) + 1$ bits are necessary for answering $\text{rank}_A$ queries, and obtain an optimal space structure that supports $\text{rank}_A$ queries in constant time, and an asymptotically optimal space structure that supports both $\text{rank}_A$ and $\text{select}_A$ queries in constant time when $\ell = O(\delta)$.

- We also consider the $\text{rank}_A$ and $\text{select}_A$ queries on strings over large alphabets. Given a string $A$ of length $n$ over the alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$ of size $\sigma$, we obtain a $(2\lceil n/\delta \rceil \log (\sigma + 1) + o(n/\delta) \log (\sigma + 1))$-bit data structure that supports $\text{rank}_A$ and $\text{select}_A$ on $A$ in $O(\log \sigma)$ time. We summarize our results for bit-strings in Table 1.

2. ss and iss queries with additive error $\delta$:

- We first consider a data structure for answering ss and iss queries on binary stream, i.e., all integers in the stream are 0 or 1. For exact ss and iss queries on the stream, we propose an $n + o(n)$-bit data structure for answering those queries in constant time while supporting constant time updates whenever a new element arrives from the stream. This data structure is obtained by modifying the data structure of Clark and Munro [8] for answering rank and select queries on bit-strings. Using the above structure, we obtain an $(n/\delta + o(n/\delta) + O(\log n))$-bit structure that supports ssA and issA queries on the stream in constant time while supporting constant time updates. Since at least $\lceil n/\delta \rceil$ bits are

### Table 1

<table>
<thead>
<tr>
<th>Query</th>
<th>Space (in bits)</th>
<th>Query time</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lower bounds</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{rank}_A$, $\text{select}_A$</td>
<td>$\lceil n/\delta \rceil$</td>
<td>$\delta$, additive</td>
<td>$\delta$, additive</td>
</tr>
<tr>
<td>$\text{rank}_A$, $\text{select}_A$</td>
<td>$B(\lceil n/\delta \rceil, \lfloor m/\delta \rfloor)$</td>
<td>$\Omega(\log n)$</td>
<td></td>
</tr>
<tr>
<td>$\text{select}_A$</td>
<td>$O(\lceil n/\delta \rceil \log^{O(1)} \delta)$</td>
<td>$\Omega(\log n)$</td>
<td></td>
</tr>
<tr>
<td>Upper bounds</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{rank}_A$, $\text{select}_A$</td>
<td>$n/\delta + o(n/\delta)$</td>
<td>$O(1)$</td>
<td>$\delta$, additive</td>
</tr>
<tr>
<td>$\text{select}_A$</td>
<td>$B(n/\delta, m/\delta) + o(n/\delta)$</td>
<td>$O(1)$</td>
<td>$\delta$, additive</td>
</tr>
<tr>
<td>$\text{rank}_A$</td>
<td>$(n/\delta) \log \delta + o((n/\delta) \log \delta)$</td>
<td>$t(n/\delta, n)$</td>
<td></td>
</tr>
</tbody>
</table>

### Table 2

Comparison of data structures for $ss$ queries over stream of integers in $\{0, \ldots, \ell\}$. All works can answer fixed-size window queries (where $i = n$) in $O(1)$ time. Worst case times are specified.

<table>
<thead>
<tr>
<th>Guarantee</th>
<th>Space (in bits)</th>
<th>Update time</th>
<th>Query time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGIM02 [9]</td>
<td>$(1 + \epsilon)$-multiplicative</td>
<td>$O(\epsilon^{-1} \log(\ell(n) \log(\ell(n))))$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>GT02 [12]</td>
<td>$(1 + \epsilon)$-multiplicative</td>
<td>$O(\epsilon^{-1} \log(\ell(n)))$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>BEFK16 [4]</td>
<td>$\delta$-additive, for $\delta = \Omega(\ell)$</td>
<td>$O(\ell(n/\delta + \log n))$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>BEFK16 [4]</td>
<td>$\delta$-additive, for $\delta = o(\ell)$</td>
<td>$O(\log(\ell(n)))$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>This paper</td>
<td>$\delta$-additive</td>
<td>Same as in [4]</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
necessary for answering `drankA₁` (or `selectA₁`) queries on bit-strings, and `[lg n]` bits are necessary for answering `ss(n, n)` queries [4], the space usage of our data structure is succinct (i.e., optimal up to lower-order terms) when `n/δ = o(lg n)`, and asymptotically optimal otherwise.

- We then consider the generalization that allows integers in the range `[0, 1, ..., ℓ]`, for some ℓ ∈ N. First, we present an algorithm that uses the optimal `n lg (ℓ + 1) + o(1)` bits for exact suffix sums. Then, we provide a second algorithm that uses `n/⌊δ/ℓ⌋ lg(max(ℓ/δ, 1 + 1) + o(1)) + O(lg n)` bits for solving `ssA`. Specifically, our data structure is succinct when `n/δ = O(lg n)`, and is asymptotically optimal otherwise, and improves the query time of [4] while using the same space. Table 2 presents this comparison.

2. Queries on binary strings and streams

In this section, we first consider data structures for answering approximate rank and select queries on bit-strings and multisets. We also show how to extend our data structures on static bit-strings to sliding windows on binary streams, for answering approximate `ss` and `iss` queries.

2.1. Approximate rank and select queries on bit-strings

We consider the approximate rank and select queries on bit-strings with additive error δ. We only show how to support `rankA₁`, `drankA₁`, `dselectA₁`, and `selectA₁` queries. To support `rankA₀`, `drankA₀`, `dselectA₀`, and `selectA₀` queries, one can construct the same data structures on the bit-wise complement of the original bit-string. We first introduce a few previous results which will be used in our structures. The following lemmas describe the optimal structures for supporting rank and select queries on bit-strings.

**Lemma 2.1 ([8]).** For a bit-string `B` of length `n`, there is a data structure of size `n + o(n)` bits that supports `rank₀`, `rank₁`, `select₀`, and `select₁` queries in `O(1)` time.

**Lemma 2.2 ([24]).** For a bit-string `B` of length `n` with `m` 1’s, there is a data structure of size

- (a) `B(n, m) + o(m)` bits that supports `select₀` queries in `O(1)` time, and
- (b) `B(n, m) + o(n)` bits that supports `rank₀`, `rank₁`, `select₀`, and `select₁` queries in `O(1)` time.

We use results from [15] and [23], which describe efficient data structures for supporting the following queries on integer arrays. For the standard word-RAM model with word size `O(lg U)` bits, let `A` be an array of `n` non-negative integers. For `1 ≤ i ≤ n` and any non-negative integer `x`, (i) `sum(i)` returns the value `∑_{j=1}^{i} A[j]`, and (ii) `search(x)` returns the smallest `i` such that `sum(i) > x`. In what follows, we use the following function to state the running time of some of the (Searchable Partial Sum) queries.

\[ SPS(n, U) = \begin{cases} 0(1) & \text{if } n = \text{polylog}(U) \\ O(\min(\lg \lg n, \frac{\lg U}{\lg \lg U}, \sqrt{\frac{\lg n}{\lg \lg n}})) & \text{otherwise} \end{cases} \]

**Lemma 2.3 ([15], [23]).** An array of `n` non-negative integers, each of length at most `α` bits, can be stored using `αn + o(αn)` bits, to support `sum` queries on `A` in constant time, and `search` queries on `A` in `SPS(n, n^{2α})` time. Moreover, when `α = O(lg lg n)`, we can answer both queries in `O(1)` time.

**Supporting `drankA` and `selectA` queries.** We first consider the problem of supporting `drankA₁` or `selectA₁` queries with additive error `δ` on a bit-string `B` of length `n`, and prove a lower bound on space used by any data structure that supports either of these two queries.

**Theorem 2.4.** Any data structure that supports `drankA₁` or `selectA₁` queries with additive error `δ` on a bit-string of length `n` requires at least `[n/δ]` bits. Also if the bit-string has `m 1’s` in it, then at least `B([n/δ], [m/δ])` bits are necessary for answering the above queries.

**Proof.** Consider a bit-string `B` of length `n` divided into `[n/δ]` blocks `B_1, B_2, ..., B_{n/δ}` where for `1 ≤ i < [n/δ], B_i = B[i(i−1)+1, ..., iδ]` and `B_{n/δ} = B[⌈(n/δ)−1⌉+1, ..., n]` (the last block may contain more than `δ`, but less than `2δ` bits). Let `S` be the set of all possible bit-strings satisfying the condition that all the bits within a block are the same (i.e., each block contains either all zeros, or all ones), and hence |`S`| = `2^{n/δ}`. We now show that any two distinct bit-strings in `S` will have different answers for some `drankA₁` query (and also some `selectA₁` query). Consider two distinct bit-strings `B` and `B′` in `S`, and let `i` be the leftmost index with `B_i ≠ B′_i`. Then by the construction of `B` and `B′`, all the possible answers of `drankA₁(iδ, B, δ)` are different from all the possible answers of `drankA₁(iδ, B′, δ)`, for all values of `i` (note that the answer to a `drankA₁` query is not unique). By the same argument, there exists no position which can be an answer of both
selectA_{j, Reb, \delta} and selectA_{j, Reb, \delta} queries, where j is the number of 1's in \( B[1 \ldots \delta] \). Thus, any structure that supports either of these queries must distinguish between every element in S, which implies \( \lceil n/\delta \rceil \) bits are necessary to answer \text{drankA}_1 or selectA_1 queries.

For the case when the number of 1's in the bit-string is fixed to be m, we choose \( \lceil m/\delta \rceil \) blocks from each bit-string and make all bits in the chosen blocks to be 1's (and the rest of the bits as 0's). Since there are \( \binom{\lceil n/\delta \rceil}{\lceil m/\delta \rceil} \) ways for select such \( \lceil m/\delta \rceil \) blocks in a bit-string of length n, it implies that \( B(\lceil n/\delta \rceil, \lceil m/\delta \rceil) \) bits are necessary to answer \text{drankA}_1 or selectA_1 queries in this case. □

The following theorem gives a data structure for supporting \text{drankA}_1 and selectA_1 queries in constant time, using optimal space.

**Theorem 2.5.** For a bit-string B of length n, if there is an \( s(n) \)-bit data structure which supports \text{rank}_1 and select1 queries in \( t(n) \) time, then there exists an \( s(n/\delta) \)-bit data structure which supports \text{drankA}_1 and selectA_1 queries with additive error \( \delta \) in \( t(n) \) time.

**Proof.** We divide B into \( \lceil n/\delta \rceil \) blocks \( B_1, B_2, \ldots, B_{\lceil n/\delta \rceil} \) where for \( 1 \leq i < \lceil n/\delta \rceil \), \( B_i = B[\delta(i - 1) + 1 \ldots \delta i] \) and \( B_{\lceil n/\delta \rceil} = B[\delta \lceil (n/\delta) - 1 \rceil + 1 \ldots n] \). Now we define another bit-string \( B' \) of length \( n/\delta \) where for \( 1 \leq i \leq \lceil n/\delta \rceil \), \( B'[i] \) is 1 if \( B_i \) contains \( \delta \)-th 1 in B for any integer \( j \leq i \), and \( B'[i] = 0 \) otherwise. Note that any block of B has at most one such position in B. Then by constructing the \( s(n/\delta) \)-bit data structure on \( B' \), we can support \text{rankA}_1 and selectA_1 queries on B in \( t(n) \) time.

Now we claim that \( C = \delta \cdot \text{rank}_1([[1/\delta], B')] + (i \mod \delta) B'[\lceil i/\delta \rceil] \) gives an answer of the \text{drankA}_1(i, B, \delta) query, which can be computed in \( t(n) \) time. Let \( D = \delta \cdot \text{rank}_1([[1/\delta], B']) \), and let \( d \) be the position of \( D \)-th 1 in B. By the definition of \( B' \), if \( B'[\lceil i/\delta \rceil] = 0 \) or \( i \mod \delta = 0 \), the claim holds since there are less than \( \delta \) 1's in \( B[d \ldots i] \). Now consider the case when \( B'[\lceil i/\delta \rceil] = 1 \) and \( i \mod \delta \neq 0 \). Then there are at most \( \delta + (i \mod \delta - 1) \) 1's in \( B[d \ldots i] \), which is the case when \( \delta \lceil i/\delta \rceil + 1 \) is the position of \( \delta \)-th 1 in B and all the values in \( B[\delta \lceil i/\delta \rceil + 2 \ldots i] \) are 1. Thus, \( C \) gives a valid answer for \text{drankA}_1(i, B, \delta) in this case. Also by the same argument, one can answer the selectA_1(i, B, \delta, \beta) query in \( t(n) \) time by returning \( \delta \cdot \text{select}_1([[1/\delta], B')] - 1 + (i \mod d) \). □

By combining the data structures of Lemma 2.1 and 2.2 with Theorem 2.5, we obtain the following corollary.

**Corollary 2.6.** For a bit-string B of length n, there is a data structure that uses \( n/\delta + O(n/\delta) \) bits and supports \text{drankA}_1 and selectA_1 queries with additive error \( \delta \), in constant time. If there are \( m \) 1's in B, the data structure uses \( B(n/\delta, m/\delta + O(n/\delta) \) bits and supports the queries in \( O(1) \) time.

Note that the proof of Theorem 2.5 implies that any data structure that supports \text{rank}_1 (or select1) queries on \( B' \) can be used to answer \text{drankA}_1 (or selectA_1) queries on B. For example if B is very sparse, i.e., when \( B(n/\delta, m/\delta) \ll O(n/\delta) \) (in this case, the space usage of the structure of Corollary 2.6 is sub-optimal), one can use the structure of [21] that uses \( (m/\delta) \lg (m/n) + O(m/\delta) \) bits (asymptotically optimal space), to support \text{drankA}_1 queries in \( O(min(\lg m, \lg (n/m))) \) time, and selectA_1 queries in constant time.

**Supporting \text{rankA}_1 and dselectA_1 queries.** Now we consider the problem of supporting \text{rankA}_1 and dselectA_1 queries with additive error \( \delta \) on bit-strings of length n. The following theorem describes a lower bound on space.

**Theorem 2.7.** Any data structure that supports \text{rankA}_1 or dselectA_1 queries with additive error \( \delta \) on a bit-string of length n requires at least \( \lceil n/2\delta \rceil \lg \delta \) bits.

**Proof.** We first construct a set V of bit-strings of length n as follows. We divide each bit-string B into \( \lceil n/\delta \rceil \) blocks \( B_1, B_2, \ldots, B_{\lceil n/\delta \rceil} \). Such that for \( 1 \leq i < \lceil n/\delta \rceil \), \( B_i = B[\delta(i - 1) + 1 \ldots \delta i] \) and \( B_{\lceil n/\delta \rceil} = B[\delta \lceil (n/\delta) - 1 \rceil + 1 \ldots n] \). Now for every \( 1 \leq i \leq \lceil n/\delta \rceil \), we set all bits in \( B_i \) to 0 if i is odd. If i is even, we fill \( B_i \) to \( k \leq \delta \) 1’s followed \( (\delta - k) \) 0’s. Thus there’s only one choice of blocks \( B_i \) if i is odd, and \( \delta \) choices for blocks \( B_j \) if i is even. Hence |V| = \( \delta \cdot \lceil n/2\delta \rceil \). Now consider two distinct bit-strings B and \( B' \), and let i be the smallest even index which satisfies \( B_i \neq B'_i \), where \( B_1 \) and \( B'_1 \) has k and \( k' \) 1’s. Without loss of generality, assume k is less than \( k' \). Then by the construction of B and \( B' \), there exists no value which can be an answer of both \( \text{rankA}_1([[i + 1/\delta], B, \delta]) \) and \( \text{rankA}_1([[i + 1/\delta], B', \delta]) \) queries. By the same argument, there exists no position which can be an answer of both \( \text{dselectA}_1(\ell, B, \delta) \) and \( \text{dselectA}_1(\ell, B', \delta) \) queries, where \( \ell \) is number of 1’s upto i-th block in \( B' \). Thus, any structure that supports either of these queries must distinguish between every element in S, which implies \( \lg |V| = \lceil n/2\delta \rceil \lg \delta \) bits are necessary to answer \text{rankA}_1 or dselectA_1 queries. □

We now show that for some values of \( \delta \), any data structure that uses up to a \( \lg^{O(1)} \delta \) factor more than the optimal space cannot support dselectA_1 queries in constant time.
Theorem 2.8. Any \((n/\delta)\log^{O(1)} n\)-bit data structure that supports \(dselectA_1\) queries with an additive error \(\delta = O(n^c)\), for some constant \(0 < c < 1\) on a bit-string of length \(n\) requires \(\Omega(\log \log n)\) query time.

Proof. We reduce the predecessor search problem to the problem of supporting \(dselectA_1\) queries. Given a set \(S \subseteq \{1, \ldots, n\}\), a predecessor query, \(\text{pred}(i, S)\) returns the largest element in \(S\) that is smaller than \(i\), for \(1 \leq i \leq n\). Now suppose we want to support \(\text{pred}\) queries on \(S\) with \(|S| = n/\delta\), where \(\delta = O(n^c)\) with some constant \(0 < c < 1\). For this range of parameters, Patrascu and Thorup [22] showed that any data structure that represents \(S\) using \(O(n \log^{O(1)} n)\) bits needs \(\Omega(\log \log n)\) time to support \(\text{pred}\) queries. We now show that any data structure that supports \(dselectA_1\) queries can be used to obtain a data structure that supports \(\text{pred}\) queries, using asymptotically the same space and query time. The theorem immediately follows from this reduction.

Let \(S' = \{k\delta [1 \leq k \leq \lfloor n/\delta \rfloor] \cup [n]\}\). We call the elements in \(S'\) as the dummy elements. Next, let \(S_1 = S \cup S'\) with \(|S_1| = \ell\), and let \(x_1, x_2, \ldots, x_\ell = n\) be the set of all elements of \(S_1\) in increasing order (note that \(n/\delta \leq \ell \leq 2n/\delta\) since both \(S\) and \(S'\) have size \(n/\delta\)). The dummy elements in \(S_1\) ensure that \(x_1 \leq \delta\) and \(x_i - x_{i-1} \leq \delta\), for \(1 < i \leq \ell\). Now consider the bit-string \(B = B_1 B_2 \ldots B_\ell\), where the block \(B_1 = \mathsf{O}^{\delta - x_1} 1^{x_1}\) and \(B_i = \mathsf{O}^{\delta - x_{i-1}} 1^{x_i - x_{i-1}}\) for \(1 < i \leq \ell\). Hence, we encode the differences between successive elements of \(S_1\) using fixed-length right-justified unary codes of size \(2\delta\). Note that \(B\) contains \(x_i = n\)’s, and has length \(2\delta \ell \leq 2n\). In addition, we store an array \(A\) of length \(\ell\) where \(A[i] = \text{pred}(x_i, S)\) using \((n/\delta)\log n) = O((n/\delta)\log n)\) bits, since \(\delta = O(n^c)\).

Suppose that there is a data structure \(X\) that uses \(s(n, \delta)\) space, and supports \(dselectA_1\) queries on \(B\) in \(t(n, \delta)\) time. To answer the query \(\text{pred}(x, S)\), we first perform the \(dselectA_1(x, B, \delta)\) on \(X\). Let \(B_i\) be the block to which this answer belongs. Since each block starts with a sequence of at least \(\delta\) zeros, and since \(dselectA_1(x, B, \delta) \leq \text{select}_1(x, B)\), it follows that \(x_i \leq x < x_{i+1}\). Hence we return \(A[x_i]\) as the answer of \(\text{pred}(x)\). Thus, from the assumption about the data structure \(X\), we can obtain a structure that uses \(s(n, \delta) + O((n/\delta)\log n)\) bits and supports \(\text{pred}\) queries in \(t(n, \delta) + O(1)\) time. The theorem follows from this reduction, and the predecessor lower bound mentioned above.

The following theorem describes a simple data structure for supporting \(\text{rankA}_1\) and \(dselectA_1\) queries.

Theorem 2.9. For a bit-string \(B\) of length \(n\), there is a data structure of size \((n/\delta)\log \delta + o((n/\delta)\log \delta)\) bits, which supports \(\text{rankA}_1\) queries on \(B\) using \(O(1)\) time and \(dselectA_1\) queries on \(B\) using \(\text{SPS}(n/\delta, n/\delta)\) time.

Proof. We divide the \(B\) into \([n/\delta]\) blocks \(B_1, B_2, \ldots, B_{[n/\delta]}\), as in the proof of Theorem 2.5. Also we define an array \(C[1 \ldots [n/\delta]]\) of length \([n/\delta]\) where for \(1 \leq i \leq [n/\delta]\), \(C[i]\) is the number of 1’s in \(B_i\). We store the structure of Lemma 2.3 on \(C\) using \((n/\delta)\log \delta + o((n/\delta)\log \delta)\) bits, to support sum and search queries on \(C\). Then since \(j - \delta \leq \delta \lfloor j/\delta \rfloor \leq j\), the answer of \(\text{sum}(\lfloor j/\delta \rfloor)\) query on \(C\) gives the answer of \(\text{rankA}_1(j, B, \delta)\), which can be answered in \(O(1)\) time by Lemma 2.3. To answer the query \(\text{dselectA}_1(j, B, \delta)\), we first find the block \(B_i\) in \(B\) which contains the position \(\text{select}_1(j, B)\) by answer the \(i = \text{search}(j)\) query on \(C\). Since \(\text{select}_1(j, B) - \delta < (i - 1)\delta \leq \text{select}_1(j, B), (i - 1)\delta\) gives an answer of the \(\text{dselectA}_1(j, B, \delta)\) query.

2.2. Approximate rank and select queries on multisets

In this section, we describe data structures for answering approximate rank and select queries on a multiset with additive error \(\delta\). Given a multiset \(S\) where each element is from the universe \(U = \{1, 2, \ldots n\}\), the rank and select queries on \(S\) are defined as follows:

- \(\text{rank}(i, S)\): returns the number of elements in \(S\) whose value is at most \(i\).
- \(\text{select}(i, S)\): returns the \(i\)-th smallest element in \(S\).

One can define approximate rank and select queries on multisets (also denoted as \(\text{rankA}, \text{drankA}, \text{selectA}, \text{dselectA}\)) analogously to the queries on strings [24]. Any multiset \(S\) of size \(m\) from a universe of size \(n\) can be represented as a characteristic vector \(B_S\) of size \(m + n\), where \(B_S = 1^{m_1} 0^{m_2} 1^{m_3} 0 \ldots 1^{m_k} 0\) when the element \(k\) has multiplicity \(m_k\) in \(S\), for \(1 \leq k \leq n\). Then one can answer \(\text{rank}(i, S)\) and \(\text{select}(i, S)\) queries using rank and select queries on \(B_S\), by \(\text{rank}(i, S) = \text{rankA}(\text{selectA}(i, B_S), B_S) = \text{selectA}(i, B_S) - i\) and \(\text{select}(i, S) = \text{selectA}(\text{selectA}(i, B_S), B_S)\). We now describe our data structures for answering approximate rank and select queries on \(S\) in the following two settings:

1. \(\text{rankA}, \text{drankA}, \text{selectA}, \text{dselectA}\) queries when \(|S| = m\) is fixed: We construct a bit-string \(B_S'\) of length \([m/\delta] + n\) where \(B_S'\) only keeps every \(i\delta\)-th 1 in \(B_S\), for \(1 \leq i \leq n/\delta\), and removes all the other 1’s in \(B_S\). To answer the \(\text{rankA}(i, S, \delta)\) query, we first compute \(\text{selectA}(i, B_S') - i = \lceil \text{rank}(i, S)/\delta \rceil\). Since \(\text{rank}(i, S)/\delta \leq \delta \leq \lceil \text{rank}(i, S)/\delta \rceil\), we return \(\delta \lceil \text{selectA}(i, B_S') - i \rceil\) as the answer. Similarly, we can answer the \(\text{selectA}(i, S, \delta)\) query by returning \(\text{rankA}(\text{selectA}(i/\delta, B_S'), B_S') + 1\). By storing the structure of Lemma 2.2(b) on \(B_S'\), using \(B(n + [m/\delta], [m/\delta]) = \Omega(n + [m/\delta])\) bits, we can support \(\text{rankA}\), \(\text{rankA}\), \(\text{selectA}\), and \(\text{selectA}\) queries on \(B_S'\) in constant time, which implies both \(\text{drankA}\) and \(\text{selectA}\) queries on \(S\) also can be supported in constant time.
For answering rankA and dselectA queries on S, we first construct the data structure of Theorem 2.9 on B δ to support dselectA queries. In answering rankA queries to that, we maintain the data structure of Lemma 2.3 to support sum and search queries on the arrays D[1 . . . [(n + m)/δ]] and E[1 . . . [(n + m)/δ]] where for 1 ≤ i ≤ [(n + m)/δ], D[i] and E[i] store the number of 0’s and 1’s, respectively, in the block B δ i (as defined in the proof of Theorem 2.9). Thus, the total space of the data structure is O([(n + m)/δ](log n)) bits. To answer the rankA(i, S, δ) query, we first find the block B δ i of B δ which contains i-th 0 by computing j = search(i) on D, and then return sum(j − 1) on E, which gives an answer of the rankA(i, S, δ) query by the similar argument in the proof of Theorem 2.9. To answer dselectA(i, S, δ) query, we first find the j-th block B δ j which contains an answer of the dselectA(i, B δ j, δ) query, and then return sum(j − 1) on D. Note that if j = 1, we return 0 for both queries. By Theorem 2.9, the total running time is O(S(n + m/δ,n + m)) for both rankA and dselectA queries. For special case when min([(n + m)/δ], δ) = polylog(n + m), we can answer rankA and dselectA queries on S in constant time.

(2) dRankA and selectA queries when the frequency of each element in S is at most ℓ: We describe a data structure for answering dRankA and selectA queries on S in O(1) time. We then show that at least [n/⌊δ/ℓ⌋] log(⌊δ/ℓ⌋, 1) + 1) bits are necessary for supporting dRankA queries on S. Thus, for supporting only dRankA queries, the data structure uses the optimal space. We consider the following two cases: (a) δ ≤ ℓ, and (b) δ > ℓ.

- Case 2a. δ ≤ ℓ: In this case, we first observe that |S| ≤ n(ℓ. Hence, B δ is a bit-string with n 0’s and at most n(ℓ 1’s. Let B δ' be a bit-string defined as in the Case (1); B δ has n 0’s and at most nδ/ℓ 1’s. To support dRankA on S, we need to support selectB δ on B δ'. We represent the bit-wise complement of B δ' using the structure of Lemma 2.2(a), which takes at most B(n + nδ/ℓ), nδ/ℓ) + o(n) bits and supports selectB δ' on B δ' in O(1) time. Using this structure, we can achieve optimal space usage, and support dRankA queries on S in O(1) time. Alternatively, we can represent B δ' using the structure of Lemma 2.2(b), which takes at most B(n + nδ/ℓ), nδ/ℓ) + o(n + nδ/ℓ) bits, and supports rankB δ, rankB δ, selectB δ and selectB δ queries on B δ' in O(1) time. Using this structure, we can support both dRankA and selectA queries on S in O(1) time, while using asymptotically optimal space when δ = Θ(δ).

- Case 2b. δ > ℓ: In this case, we first set μ = [δ/ℓ], and define a bit-string B'[(n/μ)] of length [n/μ] if and only if there exists a 1 between the positions of the (i − 1)μ-th 0 and the (min(iμ, n))−th 0 in B δ'. Note that there exists at most a single 1 between those two positions since μℓ ≤ δ. Now using Lemma 2.1, we construct an (n/μ + o(n/μ)) bits data structure which supports rankB δ and selectB δ queries on B' in constant time. Then one can show that δ(rankB δ[1/μ] , B')) + (ℓ mod μ)B δ'[1/μ] is an answer to the query dRankA(i, S, δ), using an argument similar to the one in the proof of Theorem 2.5. For selectA(i, S, δ) queries, we set μ = [δ/2ℓ] and construct a same structure as above, using 2nδ/ℓ + o(nδ/ℓ) bits. Since selectB δ(i, S) = μ(selectB δ[1/μ] , B')) − 1 ≤ 2μℓ ≤ δ, we can answer selectA query in constant time by returning μ(selectB δ[1/μ] , B')). Therefore, our data structure supports dRankA queries in constant time with optimal space, and twice the optimal space for supporting both dRankA and selectA queries in constant time (note that at least [n/⌊δ/ℓ⌋] bits are necessary in this case).

We now show a matching lower bound on space for supporting dRankA queries on S.

Theorem 2.10. Given a multisets S where each element is from the universe U = {1, 2, . . . , n} of size n such that the maximum frequency of each element in S is at most ℓ, any data structure that supports dRankA queries on S requires at least [n/⌊δ/ℓ⌋] log(max(⌊δ/ℓ⌋, 1) + 1) bits.

Proof. Note that S can be represented by a sequence S1, S2, . . . , Sn of size n, where Si denotes a frequency of i in S. Now we first set μ = δ/ℓ and denote I as {min(δk, ℓ)|k ∈ {0, 1, 2, . . . , max(1/μ, 1)|} ⊂ {0, 1, 2, ..., ℓ}}, and denote δ as |σ/μ| |σ ∈ I; that is, consider I = n[μ] . 0n−(n mod |μ|) . It is easy to show that every input of I gives a representation of S. We show that every two distinct inputs in I must lead to distinct answer of a dRankA query, thereby implying an [log|I|] bits lower bound as required. Let two distinct sets S1 and S2 be represented by the sequences in I such as x1 = x1,1x1,2 · · · x1,|n/μ|0n−(n mod |μ|) and x2 = x2,1x2,2 · · · x2,|n/μ|0n−(n mod |μ|) respectively such that xα,β = I for any α ∈ {1, 2}, β ∈ {1, 2, . . . , |n/μ|}. Also let t be a leftmost index which satisfies x1,t ̸= x2,t. Now consider dRankA[μ, t, S1] and dRankA[μ, t, S2] queries. If μ ≥ 1 then t = |n/μ| and (due to the definition of I) |x1,t − x2,t| ≥ δ, which implies that there is no answer which satisfies both dRankA[μ, t, S1] and dRankA[μ, t, S2] queries. On the other hand, if μ > 1 then I = {0, ℓ} and thus either x1,t = 0[μ], x2,t = ℓ[μ] or x1,t = ℓ[μ], x2,t = 0[μ]. In either case, |dRankA[μ, t, S1] − dRankA[μ, t, S2]| ≥ δ. We established that if two inputs in I lead to the same configuration of dRankA queries, the error for one of them would be at least δ while we assumed it is strictly lower. □

2.3. Approximate ss and iss queries on binary streams

In this section, we consider a data structure for answering ssA and issA queries over a stream of binary symbols {0, 1}, when sliding window size is fixed to n. We first describe a data structure for answering exact ss(i, n) and iss(i, n) queries in constant time using n + o(n) bits, while supporting updates in constant time, which is based on the data structure of Lemma 2.1.
Theorem 2.11. There exists a \( n + o(n) \)-bit data structure which supports \( ssA(i, n) \) and \( iss(i, n) \) queries in \( O(1) \) time while supporting \( O(1) \)-time updates whenever a new element arrives from the stream.

Proof. We partition the stream into frames of size \( n \), and maintain the current \( n \) elements in the sliding window, which span at most two consecutive frames, in a circular array \( A \) using \( n \) bits. Clark and Munro \cite{Clark-Munro} showed that one can answer both \( ss \) and \( iss \) queries in \( O(1) \) time on a static bit-string of size \( n \) by storing substructures of total size \( o(n) \) bits, which consist of (i) some auxiliary arrays, and (ii) a fixed lookup (precomputed) table. The lookup tables are independent of the frames, and can be shared between the frames. The auxiliary arrays store the answers to the \( ss \) and \( iss \) queries for a subset of the positions/values (i.e., a subset of all possible queries). Furthermore, these auxiliary arrays can be constructed in \( O(1) \) time per entry by scanning the original bit-string. Thus, the auxiliary arrays can be easily computed incrementally (i.e., by appending elements at the end), using \( O(1) \) time per insertion. Thus, by maintaining the substructures for the two frames spanning the current sliding window using \( o(n) \) bits in addition to the \( O(\log n) = o(n) \)-bit counter for storing the number of 1’s in the current frame, we can support both \( ss \) and \( iss \) queries in \( O(1) \) time, while supporting updates in \( O(1) \) time.

Next, we consider a data structure for answering \( ssA(i, n, \delta) \) and \( issA(i, n, \delta) \) queries on a binary stream in constant time, using \( \lceil n/\delta \rceil + O(\log \delta) + o(n/\delta) \) bits, while supporting constant time updates. We first divide the stream into consecutive chunks of size \( \delta \). For each chunk, we assign a bit as follows: a chunk is assigned the bit 1 if it contains \( j \delta \)-th 1 of the stream, for some \( j \leq n/\delta \), and 0 otherwise. Next, we consider the values assigned to the chunks as a stream, and maintain the data structure of Theorem 2.11 for supporting \( ss \) and \( iss \) queries in \( O(1) \) time over the stream of values of chunks. Also, to support \( O(1) \)-time update, we maintain two counters \( c = (f_1 \mod \delta) \) and \( t_c = (n_t \mod \delta) \) using \( O(\log \delta) \) bits, where \( f_1 \) and \( n_t \) are the number of elements and the number of 1’s in the current from the (original) stream up to position \( i \), respectively. Now we describe how to answer \( ss \) and \( iss \) queries.

- \( ssA(i, n, \delta) \): We return 0 if \( i < \delta \). Otherwise, we return \( t_c + \delta \cdot \text{ss}((i-f_1)/\delta), \lceil n/\delta \rceil + \kappa\left(\lceil (i-f_1)/\delta \rceil \right) \cdot (i-f_1) \mod \delta \), where \( \kappa \) is an indicator function defined as \( \kappa(k) = 1 \) if and only if \( k \mod \delta \neq 0 \), and the value assigned to the \( k \)-th latest chunk is 1. The correctness can be proved by the same argument as in the proof of Theorem 2.5.
- \( issA(i, n, \delta) \): We return 0 if \( i < \delta \). Otherwise, we return \( f_1 + \delta \cdot \text{iss}((i-t_c)/\delta), \lceil n/\delta \rceil + ((i-t_c) \mod \delta) \). Again, the correctness can be proved by the same argument as in the proof of Theorem 2.5.

In conclusion, we obtain the following theorem.

Theorem 2.12. For a binary stream, there exists a data structure that uses \( \lceil n/\delta \rceil + O(\log \delta) + o(n/\delta) \) bits and supports \( ssA \) and \( issA \) queries on the stream with additive error \( \delta \), in constant time, while supporting constant time updates.

Compared to the lower bound of Theorem 2.4 for answering \( \text{rankA} \) and \( \text{selectA} \) queries on bit-strings (which also gives a lower bound for answering \( ssA \) and \( issA \) queries), the above data structure takes \( \Omega(n/\delta) \) bits when \( n/\delta = o(\log \delta) \). However for a sliding window of size \( n \), at least \( \lceil \log n \rceil \) bits are necessary \cite{Algorithms} for answering \( ssA \) queries, even in the case when \( i \) is always equal to \( n \). Therefore the data structure of Theorem 2.12 supports \( ssA \) and \( issA \) queries with optimal space when \( n/\delta = o(\log \delta) \), and with asymptotically optimal space otherwise.

3. Queries on strings and streams over a large alphabet

In this section, we consider non-binary inputs. First, we look at general alphabet and derive results for approximate rank and select. Then we consider approximate \( ss \) queries over integer streams.

3.1. \( \text{rankA} \) and \( \text{selectA} \) queries on strings over general alphabet

Let \( A \) be a string of length \( n \) over the alphabet \( \Sigma = \{1, 2, \ldots, \sigma\} \) of size \( \sigma \). Then, for \( 1 \leq j \leq \sigma \), the query \( \text{rank}_j(i, A) \) returns the number of \( j \)'s in \( A[1 \ldots i] \), and the query \( \text{select}_j(i, A) \) returns the position of the \( i \)-th \( j \) in \( A \) (if it exists). Similarly, the queries \( \text{rankA}_j(i, A, \delta) \) and \( \text{selectA}_j(i, A, \delta) \) are defined analogous to the queries \( \text{rankA} \) and \( \text{selectA} \) on bit-strings. By extending the proof of Theorem 2.4 to strings over larger alphabets, one can show the following.

Corollary 3.1. Any data structure that supports \( \text{rankA} \) or \( \text{selectA} \) queries with additive error \( \delta \) on a string of length \( n \) over an alphabet of size \( \sigma \) requires at least \( \lceil n/\delta \rceil \log \sigma \) bits.

Proof. We divide the string \( B \) of length \( n \) into \( \lceil n/\delta \rceil \) blocks \( B_1, B_2, \ldots, B_{\lceil n/\delta \rceil} \) where for \( 1 \leq i < \lceil n/\delta \rceil, B_i = B[\delta(i-1) + 1 \ldots, \delta i] \) and \( B_{\lceil n/\delta \rceil} = B[\delta\lceil n/\delta \rceil - 1 + 1 \ldots, n] \) (the last block may contain more than \( \delta \), but less than \( 2\delta \) characters). Let \( S \) be the set of all possible strings satisfying the condition that all the characters within a block are the same. Then there are \( \sigma^{\lceil n/\delta \rceil} \) possible strings in \( S \), where for any two distinct strings from \( S \), they have different answers for some \( \text{rankA}_1 \) query (and also some \( \text{selectA}_1 \) query), by the same argument as the proof of Theorem 2.4. \( \square \)
In this section, we describe a data structure that supports \( \text{rankA} \) and \( \text{selectA} \) queries in \( O(\lg \lg \sigma) \) time, using twice the optimal space. We make use of the following result from [13] for supporting \( \text{rank} \) and \( \text{select} \) queries on strings over large alphabets.

**Lemma 3.2 ([13])**. Given a string of length \( n \) over the alphabet \( \Sigma = \{1, 2, \ldots, \sigma\} \), one can support \( \text{rank}_j \) queries in \( O(\lg \lg \sigma) \) time and \( \text{select}_j \) queries in \( O(1) \) time, using \( n \lg \sigma + o(n \lg \sigma) \) bits, for any \( 1 \leq j \leq \sigma \).

The following theorem shows we can construct a simple data structure for supporting \( \text{rankA}_j \) and \( \text{selectA}_j \) queries on \( A \) using the above lemma.

**Theorem 3.3.** Given a string of length \( n \) over the alphabet \( \Sigma = \{1, 2, \ldots, \sigma\} \), one can support \( \text{rankA}_j \) and \( \text{selectA}_j \) queries for any \( 1 \leq j \leq \sigma \) on the string in \( O(\lg \lg \sigma) \) time, using \( 2n \lg (\sigma + 1)/\delta + o(\lg \sigma) \) bits.

**Proof.** We first divide the input string, say \( A \), into \( \lceil n/\delta \rceil \) blocks \( A_1, A_2, \ldots, A_{\lceil n/\delta \rceil} \) where for \( 1 \leq i \leq \lceil n/\delta \rceil \), \( A_i = A[\delta(i-1)+1 \ldots \delta i] \) and \( A_{\lceil n/\delta \rceil} = A[\lceil n/\delta \rceil - 1 \ldots n] \) (if \( n \) is not multiple of \( \delta \)). Then we construct a new string \( A' = A_1S_2S_3\ldots S_{\lceil n/\delta \rceil} \) of length \( n + \lceil n/\delta \rceil \), where \( S_i \) is a symbol not in \( A \). Now, we construct yet another string \( A'' \) of length at most \( \lceil n/\delta \rceil + \lceil n/\delta \rceil \), which is a subsequence of \( A' \), obtained by only keeping every \( i\delta \)-th occurrence of each symbol from \( A \), for \( 1 \leq i \leq \lceil n/\delta \rceil \), and also all the occurrences of \( S_i \) in \( A' \), while removing all the other characters in \( A' \). We then represent \( A'' \) using the structure of Lemma 3.2, using \( 2n \lg (\sigma + 1)/\delta + o(n \lg \sigma) \) bits to support \( \text{rankA}_j \) and \( \text{selectA}_j \) queries in \( O(\lg \lg (\sigma)) \) and \( O(1) \) time, respectively.

Now we describe how to support the queries. For answering the \( \text{rankA}_j(i, \delta) \) query, we first compute the position, \( b_{(i/\delta)} \), of the \( (i/\delta) \)-th \( \Sigma \) in \( A'' \), in constant time, using \( b_{(i/\delta)} = \text{select}(i/\delta) \). Then by an argument similar to the one in the proof of Theorem 2.5, one can show that \( \delta \cdot \text{rank}_j(b_{(i/\delta)}) + (i \text{ mod } \delta) \) gives an answer of the \( \text{rankA}_j(i, \delta) \) query, where \( \text{rank}_j(b_{(i/\delta)}) = \text{rank}_j(b_{(i/\delta)+1}, A') - \text{rank}_j(b_{(i/\delta)}, A') \), which is either 0 or 1. Thus, \( \text{rankA}_j(i, \delta) \) query can be answered in \( O(\lg \sigma) \) time. Similarly, it is easy to see how we can answer the \( \text{selectA}_j(i, \delta) \) query in \( O(\lg \sigma) \) time by returning \( \delta \cdot \text{rank}_j(\text{select}(i/\delta) - 1, A') + (i \text{ mod } \delta) \). \( \square \)

### 3.2. Supporting \( \text{ssA} \) queries over non-binary strings

In this section, we consider the problem of computing suffix sums over a stream of integers in \( \{1, 2, \ldots, \ell\} \). This generalizes the result of Theorem 2.12 for \( \text{ssA} \). For such streams, one can use \( \text{ssA} \) binary search to solve \( \text{issA} \), while a constant time \( \text{issA} \) queries are left as future work. Specifically, we show a data structure that requires \( n (\ell/\epsilon)^{\Theta(\log n)} = n^{(\ell/\epsilon)^{\Theta(\log n)}} \) bits, which is not even asymptotically optimal for non-constant \( \ell \) values. While one may think that this is solvable by further breaking the sub-blocks into sub-sub-blocks, sub-sub-sub-blocks, etc., it is not the case. To see this, consider a lookup table for sequences of length 2. Then its space requirement will be \( (\ell + 1)^2 \) bits. If \( \ell \) is large (say, \( \ell \geq n \)) then this becomes \( \Omega(n\epsilon) = o(n \lg (\ell + 1)) \), which is not even asymptotically optimal.

**Theorem 3.4.** There exists a data structure that requires \( n \lg (\ell + 1) (1 + o(1)) \) bits and supports \( O(1) \) time \( \text{ssA} \) queries and updates.

**Proof.** Same as in the proof of Theorem 2.11, we break the stream into \( n \)-sized frames, and store auxiliary structures for each frame to support the queries efficiently. We first divide each frame into \( \lceil n/\lg^2 n \rceil \) equal-sized blocks. The blocks are then sub-divided into \( \sqrt{n/\lg^2 n} \) sized sub-blocks. For each frame, we keep a \( \lceil n/\lg^2 n \rceil \)-sized array \( C \) that stores the sum of elements from the beginning of the frame, and a \( \lceil n/\lg^2 n \rceil \)-sized array \( S \) with the sums from the beginning of the corresponding blocks. The number of bits required for \( C \) is \( n/\lg^2 n \lg (n\ell + 1) + [n/\lg n] \lg (\ell + 1) = o(n \lg (\ell + 1)) \). Similarly, \( S \) requires \( n/\lg^2 n \lg (\ell + 1) = O \left( \frac{n \lg (\ell + 1)}{\lg n} \right) \) bits. Next, we consider the two cases (i)
\( \ell + 1 \leq 2^{\frac{\sqrt{\log n}}{\log \log n}} \), and (ii) \( \ell + 1 > 2^{\frac{\sqrt{\log n}}{\log \log n}} \), and give separate data structures to handle each case. In both the cases, we store the arrays \( C \) and \( SC \) for the two frames spanning the current (sliding) window. In case (i), we also maintain all the elements in the current window in a circular array \( A \), using \( n \log (\ell + 1) \) bits. Since the sub-block size is \( \sqrt{\log n} \), we can store a look-up table for all sequences of size \( \leq \sqrt{\log n} \) as this only requires \( O((\ell + 1)^{\frac{1}{\log \log n}} \log (\ell \log n)) = O(2^{\frac{1}{2} \log n / 6} \log (\ell \log n)) = o(n \log (\ell + 1)) \). To handle an update, we simply need to add the value of the newly added element to the last two entries in the \( C \) and \( SC \) arrays (or create new entries if needed). Thus, by maintaining the array \( A \) and the lookup table in addition to the \( C \) and \( SC \) arrays for the two frames, we get a solution that requires \( n \log (\ell + 1)(1 + o(1)) \) bits and supports succinct sums queries and updates in constant time. In case (ii), the lookup-table approach fails. Thus, we do not use such a table, nor do we keep the last \( n \) elements window. Instead, we keep an \( n \)-sized circular array \( B \) in which every entry is the cumulative sum from the beginning of the sub-block. The space required for keeping the array is then

\[
\begin{align*}
n \log \left( \frac{\ell}{\sqrt{\log n}} + 1 \right) &\leq n \log (\ell + 1) \left( 1 + \frac{\sqrt{\log n}}{\ell + 1} \right) = n \log (\ell + 1)(1 + o(1)).
\end{align*}
\]

Thus, by keeping this array \( B \) in addition to the \( C \) and \( SC \) arrays for the two frames, we get a solution that takes \( n \log (\ell + 1)(1 + o(1)) \) bits and supports constant time updates and queries. \( \square \)

**General \( ssA \) queries** Here, we consider the general problem of computing \( ssA \) (i.e., up to an additive error of \( \delta \)). Intuitively, we apply the exact solution from the previous section on a compressed stream that we construct on the fly. A simple approach would be to divide the streams into consecutive chunks of size \( \max((\mu \cdot 1), 1) = \max((\delta/\ell \cdot 1), 1) \) and represent each chunk's sum as an input to an exact suffix sum algorithm, same as in the binary stream case. However, this fails to achieve succinct space. For example, summing \( \delta/\ell \) integers requires \( O((\delta/\ell/\ell) \log (\ell + 1)) = \Omega(\ell \log \ell \) bits. However, \( \ell \log \ell \) bits may be asymptotically larger than \( \log (1/\mu) \log (1/\mu) \cdot 1 + 1 \) bits lower bound of Theorem 2.10. We alleviate this problem by rounding the arriving elements. Namely, when adding an input \( x \in \{0, 1, \ldots, \ell\} \), we first round its value to \( \text{Round}_b(x) = 2^{b - b} \mu \cdot \left\lfloor \frac{x}{2^{b - b}} \right\rfloor \) so it will require \( \log (\mu) + \log (n) \) bits.\(^2\) The rounding allows us to sum elements in a chunk (using a variable denoted by \( r \)), but introduces a rounding error. To compensate for the error, we use chunks of size \( \nu = \max(\{\mu \cdot (1 / (1 - 1/\log n)) \cdot 1\}) \). We also consider \( \delta = \delta \cdot (1 - 1 / \log n) \) that is slightly lower than \( \delta \) for compensating the rounding error when \( \mu \leq 1 \).\(^3\) We then employ the exact suffix sums construction of Theorem 3.4 for window size of \( s = \lceil n/\nu + 1 \rceil \) (the number of chunks that can overlap with the window) over a stream of integers in \( \{0, 1, \ldots, z\} \), where \( z = \max(\{\mu^{-1} \nu \cdot 1\}) \) is a upper bound on the resulting items. We use \( \rho \) to denote the input that represents the current block.

The query procedure is also a bit tricky. First, we introduce the following variables used in our algorithm for \( ssA \) queries:

- \( A \) - an exact suffix sum algorithm, as described in the previous section. It allows computing suffix sums over the last \( s = \lceil n/\nu + 1 \rceil \) elements on a stream of integers in \( \{0, 1, \ldots, z\} \).
- \( r \) - tracks the sum of elements that is not yet recorded in \( A \).
- \( o \) - the offset within the chunk.

Intuitively, we can estimate the sum of the last \( i \) items by querying \( A \) for the sum of the last \( i/\nu \) inserted values and multiplying the result by \( \delta \); but there are a few things to keep in mind which were not considered in the binary case. First, \( i/\nu \) may not be an integer. Next, the values within the current chunk (that has not ended yet) are not recorded in \( A \). Finally, we are not allowed to overestimate, so \( r \)'s propagation may be an issue. To address the first issue, we weigh the oldest chunk's \( \rho \) value by the fraction of that chunk that is still in the window. For the second, we add the value of \( r \) to the estimation, where \( r \) is the sum of rounded elements. Notice that we do not reset the value of \( r \) but rather propagate it between chunks. Finally, to assure that our algorithm never overestimates we subtract \( \delta - 1/2 \) from the result. A pseudo code of our method appears in Algorithm 1. The space usage of the algorithm is analyzed in the following lemma.

**Lemma 3.5.** Algorithm 1 requires \((1 + o(1)) \cdot \lceil n/\max((\mu \cdot 1), 1) \rceil \cdot \log (\lceil (\mu^{-1})^{-1} + 1 \rceil) + O(\log n)\) bits.

**Proof.** The algorithm utilizes three variables: \( A \) that requires \((1 + o(1)) \cdot \lceil n/\nu + 1 \rceil \) bits, \( r \) that uses \( O(\log \nu) \) space, and \( o \) that is allocated with \( \lceil \log n \rceil \) bits. Recall that \( s = \lceil n/\nu + 1 \rceil \) is the number of blocks that can overlap with the maximal \( n \)-sized window and \( z = \mu^{-1} \nu \) is a bound on \( \rho \). Overall, the number of bits used by our construction is

\[
(1 + o(1)) \cdot \lceil n/\nu + 1 \rceil \log (\lceil (\mu^{-1} \nu + 1 \rceil) + 1) + O(\lceil \log (n/\mu) + \log \log \lceil \log (\nu) \rceil \log \nu) + O(\log n).
\]

Since \( \nu = \max((\mu \cdot (1 - o(1))) \cdot 1) \), we get the desired bound. \( \square \)

\(^2\) That is, we encode the integer \( \lceil x^b \rceil \) and whenever needed multiply the value by \( 2^{-b} \).

\(^3\) If \( \delta = 1 \), then we simply apply the exact algorithm from the previous subsection.
Algorithm 1 Algorithm for ssA and update on stream.

1: Initialization: \( r \leftarrow 0, o \leftarrow 0, A.\text{init}() \)
2: function \( \text{Add(element } x) \) \( \triangleright \) Add an element \( x \) into sliding window
3: \( o \leftarrow (o + 1) \mod v \)
4: \( r \leftarrow r + \frac{\lceil \frac{\delta}{\ell} \rceil}{\ell} \)
5: if \( o = 0 \) then \( \triangleright \) End of a chunk
6: \( \rho \leftarrow [2^{-b} \cdot \ell \cdot \tilde{\delta}^{-1} \cdot r] \)
7: \( r \leftarrow r - [\tilde{\delta} \cdot 2^{-b} \cdot \ell^{-1} \cdot \rho] \)
8: \( A.\text{Add}(\rho) \)
9: function \( \text{ssA}(i, n, \delta) \)
10: if \( i \leq o \) then \( \triangleright \) Queried within the current chunk
11: return \( r - (\tilde{\delta} - 1/2) \)
12: else
13: \( \text{numElems} \leftarrow \left\lceil \frac{i - o}{v} \right\rceil \)
14: \( \text{totalSum} \leftarrow A.\text{ss}(\text{numElems}, s) \)
15: \( \text{oldest}_{\rho} \leftarrow \text{totalSum} - A.\text{ss}(\text{numElems} - 1, s) \)
16: \( \text{out} \leftarrow (v - ((i - o) \mod v)) \)
17: return \( r \cdot 2^{-b} \cdot \ell - (\tilde{\delta} - 1/2) + \tilde{\delta} \cdot \text{totalSum} - \ell \cdot \text{oldest}_{\rho} \cdot \text{out} \)

Thus, we conclude that our algorithm is succinct if the (additive) error \( \delta \) satisfies \( o(\ell \cdot n/\lg n) \). We note that a \( [\lg n] \) bits lower bound for Basic-Summing with an additive error was shown in [4], even when only fixed sized windows (where \( i \) is always \( n \)) are considered. Thus, our algorithm always requires \( O(B_{\ell, n, \delta}) \) space, even if \( \delta = \Omega(\ell \cdot n/\lg n) \). Here, \( B_{\ell, n, \delta} = [n/\lceil \delta/\ell \rceil] \lg(\max(\lceil \ell/\delta \rceil, 1) + 1) \) is the lower bound for static data shown in Theorem 2.10.

Corollary 3.6. Let \( \ell, n, \delta \in \mathbb{N}^+ \) such that \( \mu = \delta/\ell \) satisfies

\[
(\mu = o(\ell \cdot n/\lg n)) \land \left( (\mu = o(1)) \lor (\mu = o(1)) \lor (\mu \in \mathbb{N}) \lor (\mu^{-1} \in \mathbb{N}) \right),
\]

then Algorithm 1 is succinct. For other parameters, it uses \( O(B_{\ell, n, \delta}) \) space.

We now state the correctness of our algorithm.

Theorem 3.7. Algorithm 1 solves ssA while processing elements and answering queries in \( O(1) \) time.

Proof. For the proof, we recall a few quantities that we also use in Algorithm 1: \( \text{numElems} = \left\lceil \frac{i - o}{v} \right\rceil \), \( \text{totalSum} = A.\text{ss}(\text{numElems}, s) \), \( \text{oldest}_{\rho} = \text{totalSum} - A.\text{ss}(\text{numElems} - 1, s) \) and \( \text{out} = (v - ((i - o) \mod v)) \). We assume that the index of the most recent element is \( h = \text{out} + 1 \), such that \( x_h \) is the first element in the chunk of \( \text{oldest}_{\rho} \) and \( o = (\text{oldest}_{\rho} \mod v) \) is the offset within the current chunk. We also denote \( g = h - o \), such that \( x_g \) is the last element of the most recently completed chunk. Fig. 1 illustrates the setting. By the correctness of the \( A \) exact suffix sum algorithm, and as illustrated in Fig. 1, we have that \( \text{totalSum} \) is the sum of the last \( \text{numElems} \) added to \( A \), that \( \text{oldest}_{\rho} \) is the value of the element that represents the last chunk that overlaps with the queried window. Also, notice that \( \text{out} \) is the number of elements in that chunk that are not a part of the window. For any \( i \in \mathbb{N} \), we denote by \( \tau_i \) the value of \( \tau \) after the \( i \)th item was added; e.g., \( \tau_o \) is the value of \( \tau \) at the time of the query and \( \tau_g \) is its value before the current chunk. Notice that \( \tau_o \) is also at the end of a chunk (that does not overlap with the queried interval). For other variables, we consider their value at query time.

When a chunk ends (Line 5), we effectively perform \( \tau \leftarrow \tau \mod \tilde{\delta} \) (lines 6 and 7), thus:

\[
0 \leq \tau_o \leq \tilde{\delta} - 1.
\]

Our goal is to estimate the quantity

\[
S_i = \sum_{d=h-i+1}^{h} x_d = \sum_{d=\text{out}+1}^{h} x_d.
\]

Recall that our estimation (Line 17) is:

\[
\tilde{S}_i = \tau_{h} \cdot 2^{-b} \cdot \ell - (\tilde{\delta} - 1/2) + \tilde{\delta} \cdot \text{totalSum} - \ell \cdot \text{oldest}_{\rho} \cdot \text{out}
\]

\[
= \tau_{g} + \sum_{d=g+1}^{h} \text{Round}_{\epsilon}(x_d) - (\tilde{\delta} - 1/2) + \tilde{\delta} \cdot \text{totalSum} - \ell \cdot \text{oldest}_{\rho} \cdot \text{out},
\]

where the last equality follows from the fact that within a chunk we simply sum the rounded values (Line 4). Next, observe that we sum the rounded values in each chunk and that if \( \tau \) is decreased by \( k \cdot \tilde{\delta} \) (for some \( k \in \mathbb{N} \)) at Line 7, then we set one of the last \( \text{numElems} \) elements added to \( A \) to \( k \). This means that:

11
Similarly, we can express the algorithm’s error as:

\[
\tilde{S}_i - S_i = r_0 + \sum_{d=1}^{\ell} \text{Round}_b(x_d) + \sum_{d=\text{out}+1}^{h} \left( \text{Round}_b(x_d) - x_d \right) - \left( \delta - 1/2 \right) - \ell \cdot \text{oldest}_\rho \cdot \text{out}.
\]

Joining (5) with (2), we can express the algorithm’s error as:

\[
\tilde{S}_i - S_i = r_0 + \sum_{d=1}^{\ell} \text{Round}_b(x_d) + \sum_{d=\text{out}+1}^{h} \left( \text{Round}_b(x_d) - x_d \right) - \left( \delta - 1/2 \right) - \ell \cdot \text{oldest}_\rho \cdot \text{out}.
\]

where \( \xi \) is the rounding error, defined as \( \xi = \sum_{d=\text{out}+1}^{h} \left( \text{Round}_b(x_d) - x_d \right) \).

Since each rounding of an integer \( x \in \{0, 1, \ldots, \ell\} \) has an error of at most \( \ell \cdot n / 2^b \), and as we round \( i \leq n \) elements, we have that the rounding error satisfies

\[
0 \geq \xi \geq 0 - \frac{\ell \cdot n}{2^b} \geq -\delta / \lg n,
\]

where the last inequality is immediate from our choice of the number of bits \( b = \lceil \lg (n/\mu) + \lg \lg n \rceil \). We now split to cases based on the value of \( \mu \). We start with the simpler \( \mu < 2 \cdot (1 - 1/\lg n) \) case, in which \( \nu = 1 \) (and consequently, \( \text{out} \equiv 0 \)). This allows us to express the algorithm’s error of (6) as

\[
\tilde{S}_i - S_i = r_0 + \xi - \left( \delta - 1/2 \right).
\]

We now use (1), (7), and the definition of \( \delta \) to obtain:

\[
\tilde{S}_i - S_i = r_0 + \xi - \left( \delta - 1/2 \right) \leq -1/2.
\]

Similarly, we can bound it from below:

\[
\tilde{S}_i - S_i = r_0 + \xi - \left( \delta - 1/2 \right) \geq \xi - \left( \delta - 1/2 \right) \geq -\delta + 1/2.
\]

We established that if \( \nu = 1 \) we achieve the desired approximation. Henceforth, we focus on the case where \( \mu \geq 2 \cdot (1 - 1/\lg n) \), which means that \( \nu = \lfloor \mu \cdot (1 - 1/\lg n) \rfloor > 1 \) and \( \text{oldest}_\rho \in \{0, 1\}. \) We now consider two cases, based on the value of \( \text{oldest}_\rho \).
1. \textit{oldest}_{p} = 1 \ case.
In this case, we know that after the processing of element \( x_v \), the value of \( \tau \) was at least \( \tilde{\delta} \) (Line 6). This implies that
\[
\tau_{o} + \sum_{d=1}^{\nu} \text{Round}_b(x_d) \geq \tilde{\delta} \quad \text{and equivalently}
\]
\[
\tau_{o} + \sum_{d=1}^{\nu} \text{Round}_b(x_d) \geq \tilde{\delta} - \sum_{d=\text{out}+1}^{\nu} \text{Round}_b(x_d).
\]
Substituting this in (6), and applying (7), we get that:
\[
\hat{S}_1 - S_1 = \tau_{o} + \sum_{d=1}^{\nu} \text{Round}_b(x_d) + \xi - (\tilde{\delta} - 1/2) - \ell \cdot \text{out}
\]
\[
\geq \tilde{\delta} - \sum_{d=\text{out}+1}^{\nu} \text{Round}_b(x_d) + \xi - (\tilde{\delta} - 1/2) - \ell \cdot \text{out}
\]
\[
\geq - \left( \sum_{d=\text{out}+1}^{\nu} \ell \right) + \xi + 1/2 - \ell \cdot \text{out}
\]
\[
\geq -\delta/\lg n - \ell (1 - \mu (1 - 1/\lg n)) + 1/2 \geq -\delta + 1/2.
\]
In order to bound the error from above we use (1) and (7):
\[
\hat{S}_1 - S_1 = \tau_{o} + \sum_{d=1}^{\nu} \text{Round}_b(x_d) + \xi - (\tilde{\delta} - 1/2) - \ell \cdot \text{out}
\]
\[
\leq \tilde{\delta} - 1 + \ell \cdot \text{out} - (\tilde{\delta} - 1/2) - \ell \cdot \text{out} \leq -1/2.
\]

2. \textit{oldest}_{p} = 0 \ case.
Here, since the value of \( \text{oldest}_{p} \) is 0, we have that \( \tau_{o} + \sum_{d=1}^{\nu} \text{Round}_b(x_d) < \tilde{\delta} \) and thus
\[
\tau_{o} + \sum_{d=1}^{\nu} \text{Round}_b(x_d) \leq \tilde{\delta} - \sum_{d=\text{out}+1}^{\nu} \text{Round}_b(x_d) - 1.
\]
We use this for the error expression of (6) to get:
\[
\hat{S}_1 - S_1 = \tau_{o} + \sum_{d=1}^{\nu} \text{Round}_b(x_d) + \xi - (\tilde{\delta} - 1/2)
\]
\[
\leq \tilde{\delta} - \sum_{d=\text{out}+1}^{\nu} \text{Round}_b(x_d) - 1 + \xi - (\tilde{\delta} - 1/2) \leq -1/2
\]
We now use (1), (7), and the fact that \( \text{out} \leq \nu \) to bound the error from below as follows:
\[
\hat{S}_1 - S_1 = \tau_{o} + \sum_{d=1}^{\nu} \text{Round}_b(x_d) + \xi - (\tilde{\delta} - 1/2)
\]
\[
\geq \xi - (\tilde{\delta} - 1/2) \geq -\delta + 1/2.
\]
Finally, we need to cover the case of \( i < o \). In this case, we can return \( \tau - (\tilde{\delta} - 1/2) \) as the estimation. This directly follows from (1) and the fact that within a chunk we simply sum the rounded values (Line 4). We established that in all cases \(-\delta < \hat{S}_1 - S_1 \leq 0. \) □

4. Conclusion
In this paper, we considered the problem of supporting approximate versions of the well-studied rank/select problems over static sets and suffix sums and inverse suffix sums over sliding windows. We studied the generalization of these problems to multi-sets and examined different error guarantees, such as \( \text{error} \) in the input vs. \( \text{error} \) in the result. Most of our results include lower bounds and matching upper bounds that are asymptotically optimal and often succinct. The resulting solutions require considerably less space compared to the space required for supporting exact queries, which makes them more likely to be implemented in practice.
Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

References