Sub-linear Sequence Search via a Repeated And Merged Bloom Filter (RAMBO)

Minghao Yan  
Department of Computer Science  
Rice University  
Houston, U.S.  
mimgyang@rice.edu

Gaurav Gupta  
Department of Electrical and Computer Engineering  
Rice University  
Houston, U.S.  
gaurav.gupta@rice.edu

Benjamin Coleman  
Department of Electrical and Computer Engineering  
Rice University  
Houston, U.S.  
ben.coleman@rice.edu

Todd Treangen  
Department of Computer Science  
Rice University  
Houston, U.S.  
treangen@rice.edu

Anshumali Shrivastava  
Department of Computer Science  
Rice University  
Houston, U.S.  
anshumali@rice.edu

Abstract—Whole-genome shotgun sequencing (WGS), especially of microbial genomes, has been the core of recent research advances in large-scale comparative genomics. The data deluge has resulted in exponential growth in genomic datasets over the past years and has shown no sign of slowing down. Several recent attempts have been made to tame the computational burden of read classification and sequence search on these ultra large-scale datasets, including both raw reads and assembled genomes. A notable recent method is BItsliced Genomic Signature Index (BIGSI). It is a data-structure of array of Bloom Filters and offers very efficient query sequence search times. However, querying with BIGSI still requires probing Bloom Filters (or sets of bitslices) which scales linearly with the number of datasets. As a result, querying complexity of BIGSI scales linearly with the number of files in a dataset. In this paper, we propose a sequence search method based on Repeated and Merged BloOm Filter (RAMBO). Here the number of Bloom filter probes is significantly less than BIGSI due to sub-linear scaling for the same false-positive rate. Our idea is theoretically sound and inspired by the count-min sketch data structure, a popular streaming algorithm. RAMBO provides a significant improvement over BIGSI in terms of query time when evaluated on real genome datasets. Furthermore the insertion and query process supports parallelism. Due to the sub-linear scaling of query time, the larger the size and number of datasets, the bigger the gains are with RAMBO over BIGSI.

Index Terms—Sequence search, Online, Bloom filter, Genome data, Sub-linear query time

I. INTRODUCTION

Whole-genome shotgun sequencing (WGS) of microbial genome is now a standard component of any comparative genomic study and has been used in countless settings, from foodborne outbreak tracking [25] to infectious disease diagnostics [22]. However, as DNA sequencing has become standard [10], genome sequence data has doubled in size every 2 years and is likely to grow at an increasing pace [5], [21]. Owing to this exponential increase, it is computationally prohibitive to search these vast archives for DNA sequences of interest, including genes and polymorphisms of interest. It is beyond doubt that efficient search functionality across all the genomic datasets, similar to web search, would facilitate rapid identification of already-sequenced organisms that are highly similar to an outbreak strains. This capability would further enable studies of antibiotic resistance mutations, genes, or plasmids.

The similarity of the sequence search problem with the problem of web search, both in terms of objective and scale, has triggered a flurry of ideas borrowed from the information retrieval community. In the seminal work on BLAST, also popularly known as the Google of biological research [1], the authors provided the first attempt to align sequences in a database using efficient search trees. However, due to the reliance on computationally expensive local sequence alignment, the method does not scale to large query datasets [2]. In general, index construction time and query time become prohibitive for early bioinformatics tools applied to modern databases, whose size is still growing on a daily basis.

Compressive-sensing and MinHash based approaches have also been applied to global sequence comparison [16], [18] and have recently been extended to containment [15], [17]. However these computational approaches require an index (or sketching) of known sequences, and were not designed to return individual reads or subsequences that comprise a query sequence but rather for searching for known genes or genomes within vast databases.

The Sequence Bloom Tree (SBT) was the first approach to solve the sequence search problem on the scale of the entire sequence read archive [14] by creating an index for the database [23]. SBT utilized a hierarchy of Bloom filters to compress the database while still allowing for sequence queries. Thanks to this hierarchical bloom filter approach, SBTs allowed users to search for query sequences, such as genomic regions and transcripts, within the entire SRA for the first time. However, SBTs rely on tree data structure, similar to R trees [12], which for high dimensional inputs are known
to require memory growth exponential with dimensions. In Section VI-A we provide a discussion on this approach. Not surprisingly, experimental results from [5] suggests that SBT become less scalable when time and evolution of species are factored in, which is the case for bacteria and viruses. Several follow-ups using ideas similar to SBT also suffer from the same issues [19], [24], [26].

The recently introduced (BitSliced Genomic Signature Index) BIGSI [5] method simplified SBT by removing the hierarchy and instead using vanilla Bloom filter compression on each of the datasets independently. By avoiding trees (hierarchy), BIGSI obtained substantial memory savings compared to SBT. The paper demonstrated huge improvements over all previous methods in terms of query efficiency. The authors proposed the notion of a bitsliced signature, which essentially creates a bit array signature for a dataset by constructing a Bloom filter for each data file [4]. The simplicity of the algorithm, combined with clever bit manipulation tricks enabled BIGSI to report slightly faster time compared to SBT.

However, BIGSI has one main limitation; it keeps one Bloom filter for each sample. The size of the bitslices during query time grows linearly with the number of datasets. Thus, performing BIGSI operations over large bitslices is likely prohibitive, both in terms of latency and energy, when there are tens of millions (or higher) archived datasets.

Our Focus: The primary focus of this paper is to reduce the query cost of sequence search. In particular, unlike BIGSI, we do not want the bitslices (or signatures) of the query to be of the same size as the number of datasets, which can run into several million. At the same time, we also want an algorithm that maintains all the other features of BIGSI. In particular, we are looking at the following three properties: 1. Low false positive rate, 2. Cheap updates for streaming inputs, and 3. A simple, systems-friendly data structure that is straightforward to parallelize.

Insights from Computer Science Literature: There is a fundamental algorithmic barrier at the heart of this problem. The classical data structure literature for avoiding linear query time problems in search provides tree-based solutions that mainly implement SBT. However, trees complicate the query process and have issues with balanced partitions, especially when dimensionality blows up. Fortunately, the Count-Min Sketch (CMS) Algorithm [7] from the data streaming literature provides a fundamental workaround. Our proposal, RAMBO [11], for sequence search, combines Bloom Filters with CMS, leading to a better trade-off in practice.

Our Contributions: We leverage several critical observations from the count-min sketch literature. Instead of having separate Bloom filters for each dataset, we partition all the datasets into a small number of random groups. We keep one Bloom filter for each partition. According to the theory of the count-min sketch [7], if we repeat the partition process independently a small number of times, we can still answer search queries with high accuracy. The ideas behind RAMBO [11] lead to exponential improvements in query cost and hence would allow effortless scaling to millions of datasets.

To summarize, we propose sequence search based on RAMBO [11], a Bloom filter based data structure that has significantly less query cost compared to BIGSI. We show a rigorous experimental evaluation of RAMBO and its comparison with BIGSI. Our experiments show that RAMBO is faster to query than BIGSI for any level of accuracy. Furthermore, the speedup with RAMBO gets better with an increase in the number of datasets, clearly indicating superior scaling.

Organization This paper is organized as follows. In the next section, we introduce the problem with the background on Bloom filters and BIGSI. Then in section III we provide the intuition behind our approach. In section IV we provide the details of proposed architecture, RAMBO. We will see the method for dataset insertion and query process in the same section. In section V we present an empirical evaluation of our method and BIGSI [5]. This is followed by a brief discussion of future work and conclusion.

II. Notations and Problem Definition

We are concerned with the standard k-mer search problem over sequence read archive datasets (FASTQ format) or assembled genomes (FASTA format) consisting of raw sequences either specific to the output of a DNA sequencer or genome assembler, respectively. We denote the number of datasets with \( N \). Each dataset is represented by a set of k-mers (unique k-contiguous strings). Given a query k-mer (often denoted by \( q \)), our goal is to find a subset of datasets, from among these \( N \) datasets, that contain the query k-mer. Figure 1 provides a high-level schematic of our approach to sequence search.

A. Bloom Filters

The Bloom filter is a standard data structure that is widely used for set compression. In the context of k-mer search, Bloom filters can be used to compress a dataset FASTA of k-mers in the FASTA formats. As shown in Figure 2 a Bloom filter consists of an \( m \)-bit array (or bitslice column) and \( \eta \) independent random hash functions \( h_1, h_2, \ldots, h_\eta \). Each hash function takes integer values between 0 and \( m - 1 \), i.e., \( h_i: S \rightarrow \{0, 1, \ldots, m - 1\} \). At the start of dataset processing, each element in the bit array is initialized to 0. For every item \( x \in S \), the bit value of \( h_i(x) \), for all \( i \in \{0, 1, \ldots, \eta\} \) (\( \eta \) selected rows), is set to 1.

To check the membership status of an item \( x' \) in the dataset FASTA, we return true if all the bits \( h_i(x') \), for all the hash functions \( h_i, i \in \{0, 1, \ldots, \eta\} \), are 1. Bloom filters have zero false-negative rates and cannot return false if \( x' \) is actually in the set. However, due to random hash function collisions, \( x' \) may accidentally map to locations in the bitslice that are set to 1. In this case, we report that \( x' \in S \) when \( x' \notin S \), leading to a nonzero false positive rate. It can be shown that for an appropriate choice of \( \eta \), the number of hash functions, the false positive rate can be made negligibly small. Thus, Bloom filters are a highly efficient data structure for determining dataset membership.
Fig. 1. Illustration of the raw sequence search problem. First, we index sets of $k$-mers into $N$ raw read datasets. Given a query sequence, the task is to determine which of the $N$ raw read datasets contains the $k$-mers present in the query. The recent BIGSI approach poses this as a multiple dataset membership problem, where we test whether a $k$-mer is present in each of the $N$ datasets.

**B. BItsliced Genomic Signature Index (BIGSI)**

Figure 2 describes the BIGSI insertion and query phase. The data structure is a matrix of bits with $N$ columns, one for each dataset. The bit column represents the Bloom filter for the corresponding particular dataset. The dataset is assumed in FASTA format.

While adding a new dataset, say FASTA-1, we first break it into a set of $k$-mers. For that we take $k$-mers with sliding window of size 31. A new bit column, say col-1, is added to the sketch. Every $k$-mer in the FASTA-1 is hashed $\eta$ times using a predetermined set of random hash functions, and the corresponding hashed row in col-1 is set to 1. It should be noted that all $N$ Bloom filters share the same hash functions for consistency, which comes handy during the query process.

During querying with a $k$-mer $q$, we first compute the $\eta$ hash values of $q$ using the same hash functions used for construction. Each of these $\eta$ values corresponds to a row in the matrix sketch of BIGSI, named a *bitslice*. Overall, we get at most $\eta$ bitslices. Bitwise AND operation (row-wise) of these $\eta$ bitslices are computed to obtain a $N$ bit vector. The location of 1s in this $N$-bit vector corresponds to the dataset containing the given $k$-mer $q$.

**III. INTUITION BEHIND OUR APPROACH**

We draw inspiration from the count-min sketch (CMS) data structure from the data streaming and the sparse recovery literature [6]. Our goal is to reduce the computational cost associated with $k$-mer search queries. With the BIGSI approach (section II-B), we need to query one Bloom filter for every dataset. Thus, if we have $N$ datasets, we need to query $N$ Bloom filters. Instead, by randomly merging datasets and repeating, RAMBO can determine which datasets contain a given $k$-mer using far fewer Bloom filter queries.

To see how this might work, suppose that we randomly partition $N$ datasets into $W$ groups, where $2 \leq W \ll N$. Now, given a query $k$-mer $q$, if we query each partition, we can determine which partition contains $q$. We refer to this partition as $A$. Thus, with only $W$ Bloom filter queries, we have reduced the number of candidate datasets from $N$ to $\frac{N}{W}$ in expectation. With more repetitions, we progressively rule out more and more options until we are left with only the datasets that contain $q$. For example, if we independently repeat this process again, we find another partition $B$ that contains $q$. Our pool of candidate datasets is now the set intersection of $A$ and $B$, which in expectation has size $\frac{N}{W^2}$.

In summary, with $2W$ Bloom filter queries, we have reduced the search space from $N$ down to $\frac{N}{W^2}$. The critical insight is that each repetition reduces the number of candidates by a factor of $\frac{1}{W}$, which decreases exponentially with the number of repetitions. RAMBO uses this observation to identify the correct datasets using an exponentially smaller number of Bloom filter queries.

This method is guaranteed to be successful because RAMBO is an extension of the CMS data structure [7]. We replace the counters in the CMS with Bloom filters. Instead of adding counters in the CMS, we merge genomic sequencing datasets. The querying procedure of the CMS is replaced with a set intersection over the merged datasets to determine which datasets contain a query sequence.

We detail the algorithm in the next section.
IV. ALGORITHM/METHOD

A. Adding a Dataset to RAMBO

With RAMBO, in addition to η Bloom filter hash functions, we also have different universal hash functions which partition the set of datasets into small disjoint subsets. We call these functions partition hash function denoted by \( ph \), which are essentially universal hash functions that map a dataset to \( \{0, 1, ..., W\} \). Given a dataset FASTA-1 we use the value of \( ph(FASTA-1) \) to determine which partition dataset FASTA-1 is assigned. Clearly, with such a function, our set of datasets is partitioned into \( W \) disjoint sets. We have \( D \) such independent repetition of partition hash functions \( \{ph_1, ph_2, ..., ph_D\} \).

Figure 2 illustrates RAMBO data structure creation. The figure shows RAMBO with \( D = 2 \) and \( W = 2 \), i.e., two repetitions with each repetition having two partitions.

Each dataset, say FASTA-1, is therefore mapped to exactly \( D \) partitions, one in each repetition.

Now, unlike BIGSI (in contrast with Figure 2) with one column (bit-array for Bloom filter) for each dataset, we have a column (bit-array for Bloom filter) for each partition in each repetition. In total, have total \( D \times W \) columns (bit matrix), which can be even smaller than \( N \). Let’s say we want to add dataset FASTA-4 to RAMBO. First, we determine the columns that will be affected using \( D \) partition hash function. FASTA-4 maps to \( D \) partitions. In particular, it is mapped to partition number \( ph_i(FASTA-4) \) in repetition \( i \), \( \forall \ i \in \{0, 1, ..., W\} \).

Note each partition has an associated column (Bloom filter) with it. Unlike BIGSI, where we create a new Bloom filter (column) and add FASTA-4 to it, we instead add FASTA-4 to all these selected \( D \) columns (corresponding to the partitions).

These Bloom filter additions are just like BIGSI where we first compute all \( k \)-mers of FASTA-4, compute \( \eta \) hash values using the \( \eta \) hash function required by the Bloom filters and set the corresponding \( \eta \) bits for all the selected Bloom filters to 1. Overall, we set exactly \( \eta \times D \) bits.

As illustrated in Figure 3, FASTA-4, is mapped to partition 1 in repetition 1 and partition 2 in repetition 2. It has two \( k \)-mers: \{AAA\} and \{AAT\} which hashes to \{0, 5, 9\} and \{3, 7, 5\} respectively. As a result, we set the bits 0, 3, 5, 7, 9 in both column 1 (partition 1, repetition 1) and column 4 (partition 1, repetition 1).

An exciting advantage of RAMBO over BIGSI is that the number of columns does not grow with the data. To add a new dataset, we only need to insert the dataset into existing Bloom filters and do not have to add a new column! Our method avoids any dynamic data structure growth. Also, the algorithm works for any values of \( D \) and \( W \). When \( D \times W \ll N \), we have substantial memory savings over BIGSI and can still process additional datasets using far less memory. Primarily, we can work with any fixed, predetermined memory budget. This property makes RAMBO ideal for mobile devices. Of course, reducing the RAMBO memory incurs an increase in the false-positive rate.

The pseudo-code for the overall process is summarized in Algorithm 1.

Algorithm 1: Algorithm for insertion in RAMBO architecture

Input: Set \( D \) of \( N \) databases of \( k \)-mers

Given: Parameters \( W \times D \) and false positive rate \( p \)

Result: RAMBO (size: \( W \times D \))

Generate \( D \) partition hash functions \( ph_1(\cdot), ..., ph_D(\cdot) \)

RAMBO ← \( W \times D \) array of Bloom filters

while Input FASTA-\( i \)

for \( k \)-mer \( x \in FASTA_i \) do

for \( d = 1, ..., D \) do

insertBloomFilter(\( x \), RAMBO[\( ph_d(x)\), \( d \)])

end for

end for

end while

In the following section we present a method to perform gene sequence queries using RAMBO.

B. Querying with RAMBO

The query procedure in RAMBO is illustrated in Figure 4 on example query \( k \)-mer “AAT”. It uses the same bit array structure used for illustrating creation (compare with Figure 3).

Give a query \( k \)-mer \( q \), the initial few steps are precisely like BIGSI. First, we compute \( \eta \) hashes of \( q \). Again, using these hash values, we select rows of our data structure, which is a \( D \times W \) bit matrix. These rows form our \( \eta \) bit slices.
Again, similar to BIGSI, we perform bit-wise AND operation on these $\eta$ bitslices to obtain a $D \times W$ bit vector.

The critical difference is how we interpret this vector. In BIGSI, we report all the datasets corresponding to columns that have 1 in this final bit vector. In RAMBO, the columns only determine a partition in a repetition. The trick is to report the set of datasets which corresponds to the intersection of all the identified partition.

Clearly, the cost of querying requires bitslices of size $D \times W$ which can be much smaller than $N$.

As illustrated in Figure 3, our final bit vector is 1001, which indicates that the query comes from partition 1 in repetition 1 and partition 2 in repetition 2. Resolving this into actual datasets, we look at partition 1 in repetition 1 which as {FASTA-1, FASTA-4, FASTA-5} while partition 2 in repetition 2 has {FASTA-2, FASTA-4, FASTA-5} (see Figure 3). Therefore, the query must belong to the intersection of these two sets, which comes out to {FASTA-4, FASTA-5}.

The pseudo-code for the query process is summarized in Algorithm 2.

Algorithm 2 Algorithm for query using RAMBO architecture

Input: gene sequence $q$

Given: RAMBO bit matrix $M$ (size: $W \times D$)

Result: Set of datasets, each of which contains $q$.

1. UNION = 0

2. for $d = 1 : D$

3.  ID = $h_d(q)$

4.  bitslice$_d$ = get row number ID from $M$

5.  UNION = bitslice$_d$ (Bitwise OR) UNION

6. end for

7. INTERSECTION = Everything

8. for $i$ such that UNION[$i$] = 1 do

9.  INTERSECTION = INTERSECTION $\cap$ Partition[$i$]

10. end for

11. return INTERSECTION

V. EXPERIMENTS

In this section, our goal is to evaluate the query time of RAMBO and BIGSI. Since both are probabilistic approaches that trade efficiency with false positives, it is imperative to compare query time over the complete spectrum of false-positive rate for a fair comparison.

The focus of this section is twofold. First, we want to get a complete empirical characterization of the query time vs. error trade-off. Besides, we want to evaluate how both methods scale with the number of datasets.

Datasets We use the available NCBI (National Center for Biotechnology Information) hosted genome assemblies from the NCBI Assembly Resource database [9], which included
161,023 individual genome assemblies from both GenBank [3] and RefSeq [20].

Since there exist duplicates in the dataset, we pre-processed the dataset by running a Locality Sensitive Hashing (LSH) scheme to de-duplicate files with Jaccard similarity greater than 0.9. This pre-processing step required less than 12 CPU hours and left us with 136, 602 “unique” files. LSH is a method to hash the similar examples in same/nearby bins of the table. It facilitates a probabilistic similarity based sampling. More information about LSH can be found in [13]. A discussion of duplicate detection using LSH is beyond the scope of this paper.

For a fair evaluation, we need to obtain query times for a complete spectrum of false-positive rates. Computing the full spectrum requires running hundreds of experiments with both BIGSI and RAMBO. To ensure that these hundreds of experiments finish in a reasonable time, we generated two random subsets from the full de-duplicated genomic datasets, one with 3480 files and another 2500 files. We sample both subsets uniformly from the full dataset. All of our experiments were performed on these two samples.

**Baselines** We compare our method against the state-of-the-art genome search method BIGSI [5]. BIGSI is a recent method which has been evaluated and shown to have superior query time compared to all existing state-of-the-art methods, including SBT [26] and SSBT [24].

As mentioned before, both BIGSI and RAMBO trade false-positive rates for efficiency. Furthermore, both methods depend on a set of hyper-parameters, which can be tuned for good performance. To ensure that we provide a fair comparison, we selected hyper-parameters as described below and ran both BIGSI and RAMBO for a wide variety of choices. We compare all the false positive rate and query time results. For problem specific hyper-parameters like the k-mer length, we use the standard values from the literature. We plot the complete trade-off between query time with false-positive rate in Figures 5 and 6.

**Parameter Settings** There are five hyper-parameters in our methods. The first three are same hyper-parameters needed for our baseline BIGSI: length of k-mer, the number of hash functions (η) used in a Bloom filter , and the size of the Bloom filters. The two hyper-parameters specific to RAMBO are the size of the partition and number of repetitions.

We set the length of k-mer to 31, a common choice and the same value used in BIGSI. In practice, k-mers of longer length, like 31-mers, eliminate most of the random noise that may occur in the dataset.

For the number of hash functions, we used common values
Fig. 5. Query Time (s) vs FP Rate (%) tradeoff on subset of 3480 files with BIGSI and RAMBO. Both the values of $\eta = 3, 4$ are shown.

Fig. 6. Query Time (s) vs FP Rate (%) tradeoff on subset of 2500 files with BIGSI and RAMBO. Both the values of $\eta = 3, 4$ are shown.

3 and 4 (denoted by $k$ in Fig. 1 and Fig. 2) for both BIGSI and RAMBO.

For the first experiment with 3480 files (Figure 5), the size of the partition ($D$) is kept as 100 and number of repetitions ($W$) as 3. For the 2nd experiment (Figure 6), $D = 84$ and $W = 3$. In our experiments, we vary the size of Bloom filters to test BIGSI’s and RAMBO’s query speed with different false positive rates [5]. The size of BIGSI and RAMBO for different false positive rates is shown in plot.

Evaluation Metrics The false positive rate is data-dependent. It is the ratio of false positives and total negatives. We empirically measured the false positive rate of our method with 30000 randomly selected k-mer queries. Using these k-mers, we measured the query time under various false-positive rates by changing the size of the Bloom filters. The results are averaged over all the queries.

For the number of hash functions used in a Bloom filter, we use two values $\eta = 3, 4$.

Fig. 7. Memory (s) vs FP Rate (%) trend on subset of 3480 files with BIGSI and RAMBO. Both the values of $\eta = 3, 4$ are shown.

Fig. 8. Memory (s) vs FP Rate (%) trend on subset of 2500 files with BIGSI and RAMBO. Both the values of $\eta = 3, 4$ are shown.

System and Platform Details For both methods, the core algorithm is the Bloom filter. To ensure a fair comparison, we use Python3 for our implementation and inherit the Bloom Filter class from BIGSI [5]. This way, we ensure that the query time improvements are not due to a faster Bloom filter implementation. We conducted our experiments on a Linux server with the Intel(R) Xeon(R) CPU E5-2660 v4 CPU. The system has a 503 GB memory.

Results Fig. 5 and Fig. 6 contain the results of our experiments with a subset of 3480 and 2500 datasets, respectively. Both plots show that RAMBO outperforms BIGSI by a significant margin at all false-positive rates using the same value of $\eta$, the number of hash functions. The trade-off comes in the memory requirement, which is very minimal if the required false positive rates are low (for many practical purposes) and there is significant multiplicity of the k-mers. Refer Fig. 7 and Fig. 8. These plots confirm the aforementioned improvements of RAMBO over BIGSI for obtaining faster query time.

While a speedup of 20% is noteworthy, our most exciting
result is the scaling with an increase in the number of datasets. From the experiments, we observe that with a larger number of files (Fig 5), the speedup with RAMBO is substantially more than when compared to experiments with fewer files (Fig 6). As argued before, the number of Bloom filter probes needed by RAMBO scales sub-linearly with the number of datasets. Conclusively, the query time for RAMBO grows much more slowly than when compared to experiments with fewer files (Fig 6).

VI. DISCUSSION AND FUTURE WORK

Our method leverages the redundancy within the data and thus achieves better query speed.

The offline construction step of the Bloom filter is time-consuming. Construction time increases linearly in the number of datasets. However, due to the independent nature of the Bloom filters, we can efficiently parallelize the construction steps, which reduces the query time from days or even weeks to minutes. We can extract more speed up by having a GPU implementation in a distributed manner. With a parallel implementation, we will be able to explore many more biological data archives in a much more convenient fashion.

A. Contrast with Sequence Bloom Trees and BIGSI

RAMBO enjoys several advantages over other methods for genomic sequence search, such as sequence Bloom trees (SBT) and BIGSI. The popular BIGSI algorithm represents each dataset as a Bloom filter. As discussed previously, this method requires $O(K)$ query time but is easy to parallelize.

Several methods are based on SBTs. The SBT is a tree structure where each node in the tree is a merged Bloom filter. To query a sequence, SBTs traverse the tree in $O(\log N)$ query time. These operations are inherently sequential and cannot be done in parallel, which fails to take advantage of massively parallel modern computing hardware. SBTs also encounter another fundamental problem with trees - that of balancing the tree during construction. This construction typically involves a prohibitively large number (quadratic in the number of datasets) of Hamming distance computations over bitslices, making it difficult to obtain a balanced and optimal tree unless all datasets are known ahead of time. Updating the index in an online manner is not straightforward.

Finally, SBT-based approaches can require substantially more memory. While there are only $K$ Bloom filters in the SBT, each filter must be able to accommodate all of the k-mers in all of the datasets [8]. This is not a problem when there is a substantial overlap between datasets. However, when the datasets contain vastly different sequences, the tree needs space that is near $O(N^2)$. This phenomenon causes SBTs to be of limited use for applications with considerable genomic diversity such as bacterial and viral sequencing [26].

RAMBO provides a solid trade-off between false positive rate and query time while retaining all of the desirable properties of BIGSI and the bitslice data structure. Furthermore, due to sub-linear scaling, RAMBO becomes increasingly more efficient when compared to existing methods as the size of the dataset increases. We expect that this property will allow RAMBO to analyze extremely large-scale genomic datasets.

VII. CONCLUSION

In this paper, we propose a novel approach for querying sequences within massive genomic datasets that achieves improvements w.r.t query speed when compared to state-of-the-art approaches such as BIGSI. We take advantage of the redundancy of genome sequences among datasets (which is very prevalent in genomic datasets) to form a more compressed data representation. From the best of our knowledge, this is the only method with low memory cost and query time increases sub-linearly in the number of datasets. We empirically evaluated the effectiveness of our methods and compared against the state-of-the-art method in genomic search and demonstrates its potential in handling ultra large scale datasets.

REFERENCES

[1] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. Journal of molecular biology, 215(3):403–410, 1990.
[2] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. Journal of molecular biology, 215(3):403–410, 1990.
[3] Dennis A Benson, Karen Clark, Irena Karsch-Mizrachi, David J Lipman, James Ostell, and Eric W Sayers. Genbank. Nucleic acids research, 43(Database issue):D30–5, 2015.
[4] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7):422–426, 1970.
[5] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: A practical perspective. In STOC, pages 604–613. ACM, 1998.
[6] Yuichi Kodama, Martin Shumway, and Rasko Leinonen. The sequence read archive: explosive growth of sequencing data. Nucleic acids research, 40(D1):D54–D56, 2011.
[7] Brian D Ondov, Gabriel J Starrett, Anna Saphigna, Aleksandra Kostic, Sergey Koren, Christopher B Buck, and Adam M Phillippy. Mash screen: High-throughput sequence containment estimation for genome discovery. BioRxiv, page 557314, 2019.
