IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 13, 2019, accepted April 2, 2019, date of publication May 22, 2019, date of current version June 4, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2910180

A Review on Role of Bloom Filter on DNA Assembly

SABUZIMA NAYAK AND RIPON PATGIRI
National Institute of Technology, Silchar 788010, India
Corresponding author: Ripon Patgiri (ripon@cse.nits.ac.in)

ABSTRACT The advancement of DNA assembly techniques has greatly boosted up the bioinformatics research and discovery. More precisely, DNA assembly has achieved tremendous popularity due to the ability to decode the hidden information in the DNA. DNA assembly is the process of finding the correct sequence of the nucleotide bases in DNA. The key challenges are a) size of the genomic data and, b) time to process the genomic data. Apparently, genomic data are voluminous consisting of many repeated fragments. The huge sized genomic data makes the DNA assembling a time consuming process. To address the space and time complexity, bloom filter is deployed in DNA assembling. Moreover, bloom filter plays a vital role in DNA processing to deal with the repeated data of DNA. A bloom filter is a probabilistic data structure for membership filter. Bloom filter uses a tiny amount of memory size to store information on the genomic data. However, DNA assembling is a very memory-intensive process. The whole process consists of many stages. In every stage, the repeated data need to be taken care of. Hence, bloom filter is deployed in every stage for its implementation. This paper presents the impact of bloom filter in DNA assembling process. It also gives a precise explanation on every aspect of the DNA assembling process. The focus of this paper is to review the techniques that implemented bloom filter.

INDEX TERMS Bloom filter, DNA sequencing, DNA assembly, de novo, de Bruijn Graph, bioinformatics, big data.

I. INTRODUCTION

DNA is the operating system of the living beings that operates all organs. It contains all the information about the functioning of cells to organs to keep the living being alive. The study of DNA is emerging due to its applicability in many fields; Forensic [1], Agriculture [2], Medicine [3], Cancer [4] and so on. DNA can help to know the reason for the occurrence of some diseases like cancer. Study on DNA also helps in understanding of mutation among living things. This knowledge will help to understand the various factors that initiate the cause of mutation. In the field of forensic science, DNA fingerprinting [5] is used to identify some criminals from the evidence left at the crime scene. In agriculture, DNA Assembling helps to make crops more resistant to crop diseases. In medicine sector, DNA Assembling helps to identify the defective gene that causes a disease and replace them with healthier genes.

Currently, next-generation sequencing (NGS) [6] techniques are used for DNA Assembling. NGS has enabled the production of reads at a high speed with low cost [7]. Moreover, NGS is a great help in de Novo sequencer. The genomic data are voluminous. The volume of the genomic database is getting double in every several months [8]. Moreover, the data contain a higher percentage of repeated regions [9]. Hence, its processing is a task-intensive and memory-intensive [10]. To handle such type of data on a large scale, an efficient data structure is required to boost up the performance and reduce space consumption. Apparently, the best answer is Bloom Filter [11]. It is a probabilistic data structure for membership filtering. It has great potential to be applicable in every stage of the DNA sequencer.

Bloom Filter [11] is a simple data structure used for membership query. The time complexity for insertion and query is $O(k)$ where k is the number of hash functions. However, r-Dimensional Bloom Filter (rDBF) has the time complexity for insertion and query of $O(1)$, since, it is independent from the number of hash function. Bloom Filter has gained popularity in the field having huge duplicated data. Such domains deploy Bloom Filter along with their application to reduce the unnecessary searching of missing data. Some examples are Network Traffic, [12], Network Security [13], [14], Routing

The associate editor coordinating the review of this manuscript and approving it for publication was Vincenzo Conti.

Algorithms [15], [16] Cloud Computing [17], Big Data [18]. Bloom Filter has two key issues, namely, false positive and false negative. To address the issues, variants of Bloom Filters have been proposed.

This article focuses on providing a precise description of the various stages of the DNA Assembly. Moreover, a rigorous reviews is provided on some techniques that have used Bloom Filter. Primary focus is given to implementation of Bloom Filter, hence, in some techniques whole process is not explained. The article is organized as follows- Section II provides a precise description about de Novo Assemby. In Section III, Bloom Filter data structure is explained in detail. It also includes the issues and challenges of Bloom Filter. In addition, the issues and challenges of the DNA sequencer are taken into account while implementing Bloom Filter. Section IV discusses about the variants of Bloom Filter proposed to handle genomic data. In the article, three main stages of the DNA Assembly is discussed, namely, preprocessing filtering, graph construction process, and post-processing filtering. Section V illustrates the preprocessing filtering, the stage that removes erroneous data. Several techniques are applied in preprocessing filtering. The article discusses three main techniques, namely, read compression, k-mer counting and error correction. Currently, de Bruijn graph is mostly used to represent the DNA sequences. Hence, in this article, for the graph construction stage a detailed discussion on de Bruijn graph is done. In Section VI, the working of de Bruijn graph is explained along with its variants. Moreover, various techniques implementing both de Bruijn graph and Bloom Filter is precisely written. Section VII explains some DNA Assembly techniques implementing Bloom Filter. Section VIII discuss mainly on the postprocessing filtering and Scaffolding. Section IX discusses about the opportunity Bloom Filter provides to improve the DNA Assembly technique. And, finally, the article is concluded in Section X.

II. DE NOVO ASSEMBLY

DNA Assembly is a method to identify the proper order of the nucleotide bases. The de Novo Assembly is defined as the sequencing of the DNA without any prior guidance. Hence, it is applicable to sequence DNA, which is not previously sequenced. DNA assembler is a device or tool to achieve DNA Assembling. Reads are the partition of the DNA fragments to shorter fragments. There are some issues associated with the reads which is presented as follows-

a Position: Correct read positioning depends on its neighbor. It is important because it affects the quality of the DNA Assembly.
b Number of reads: Increasing in the number of reads leads to complexity in positioning the reads at its correct place.
c Ambiguity: Similar reads are eligible for the same location, hence, their positioning is difficult.
d Unique read: Unique reads should be placed in unique places. Unique reads are those rare DNA fragments that carries important information about our body. But, it is very difficult to discover the correct location.

The assembler groups the reads to form contigs and then contigs to produce scaffolds. The sequencing size is determined statistically. The statistical calculation is done based on the maximum, average and combined total length of the contigs and scaffolds, and N50 [9].

The DNA Assembler takes a set of short reads as input. These reads contain errors which are introduced during the experimental sequencing procedures. Reads are connected together to form longer contiguous reads, called contigs. It is achieved by a DNA sequencer. The DNA sequencer takes two files as input, namely, the short read sequences, and their quality scores [19]. Often, single file contains both. However, the DNA Assembling technologies have to process a large volume of short reads which is a highly memory-intensive task. Hence, graph is used to simplify and increase the efficiency of any complex system. For example, de Bruijn graph [20] and overlap layout consensus graphs [9]. And, using Bloom Filter further reduces the usage of memory. The DNA Assembly has mainly three stages, particularly, preprocessing filtering, graph construction process, and post-processing filtering. The Figure 1 shows the whole process of the DNA sAssembling. The preprocessing filtering detects and corrects the erroneous reads. The Graph construction process creates a graph model. The postprocessing filtering construct the contigs, and identify the misassembled reads in contigs, and construct the scaffolds [21].

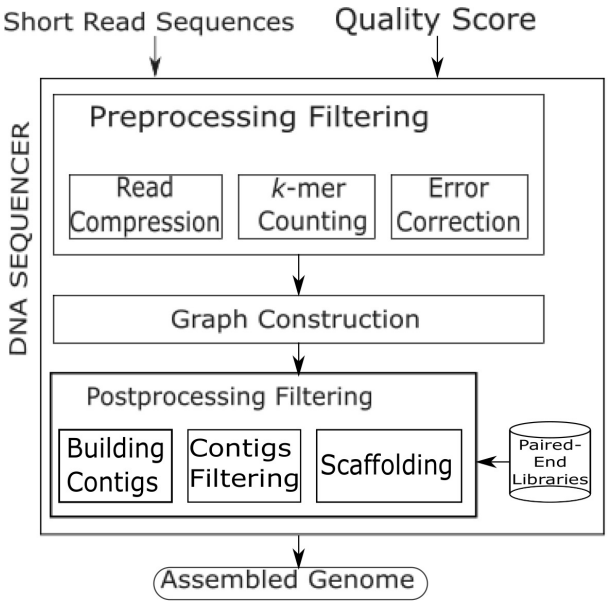

FIGURE 1. De Novo assembly.

III. BLOOM FILTER

Bloom Filter [11] is a data structure for approximate membership query to minimize memory. It is introduced by Burtorn Howard Bloom in 1970. Bloom Filter has got huge popularity since its inception. Also, it is deployed in diverse domains to

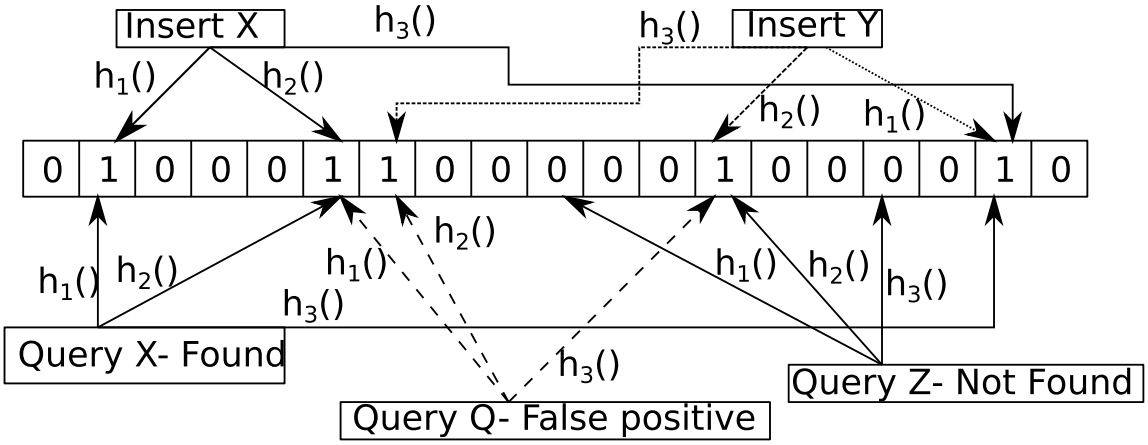

FIGURE 2. Architecture of bloom filter.

enhance time and space complexity. It is extremely efficient data structures, but has issues, namely, false positive and false negative. Hence, Bloom Filter cannot be deployed in Password filtering, and hard real-time systems. However, most of the systems can tolerate those issues. And, the occurrence of these errors is negligible.

Figure 2 depicts the architecture of Bloom Filter. Bloom Filter supports insert, query and delete operation. Most of the Bloom Filter avoids delete operation to avoid false negative. The time complexity of Bloom Filter for insertion and query operations is $O(k)$. A standard Bloom Filter is a bit array consisting of zeros and ones. As depicted in Figure 2, let us assume insertion of an input item X. The input item X is hashed using k hash functions to obtain k number of bit locations. In Figure 2, the hash functions are $h_1()$, $h_2()$ and $h_3()$ and the k value is 3. The arrows indicate the obtained locations. Those locations are set to bit '1'. In case of query operation, same procedure of hashing is followed to obtain the k locations. If all locations contain '1', then the item is present in Bloom Filter. But, if at least one location contains '0', then the item is not present in Bloom Filter. Standard Bloom Filer does not allow delete operation. But, the procedure followed in delete operation is the hashed locations are set to '0'. The response returned by Bloom Filter is classified into four types, namely, true positive, true negative, false positive and false negative.

a True positive: The item is inserted and the response given by Bloom Filter during query is true. It means that the input item is inserted and all the k locations contain '1'. For example, query of item X as given in Figure 2.

b True negative: The item is not inserted and the response given by Bloom Filter during query is false. It means that the input item is not inserted and at least one location contains '0'. For example, query of item Z as given in Figure 2.

c False positive: The item is not inserted, but the response given by Bloom Filter during query is true. It means that the input item is not inserted, but all the k locations contain '1'. For example, in case of item Q as depicted in Figure 2. The item Q is not inserted, but the locations obtained by hash functions are set to '1' by both item X and Y. Hence, Bloom Filter gives response as true.

d False negative: The item is inserted, but the response given by Bloom Filter during query is false. It means that the input item is inserted, but at least one location contains '0'. Such cases arise due to delete operation.

Bloom Filter suffers from two main issues, namely, false positive and false negative. But the occurrence of false negative is very rare. Moreover, many Bloom Filter variants do not allow delete operation to eliminate the false negative probability. The complete removal of the false positive is very difficult. Hence, the main focus of Bloom Filter variants is to reduce the false positive. Cuckoo Bloom Filter [22] stores fingerprint of the item. The size of the fingerprint is determined by the false positive probability. Smaller fingerprint gives a less false positive probability. The rfilter [23] reverses the item and stores it shatteringly to reduce the false positive. Scalable Bloom Filter [24] adds new Bloom Filter to increase the size. The new Bloom Filter size is finalized orthogonally to minimize the false positive. Dynamic Bloom filter [25] increases its size to manage the false positive. Bloofi [26] is a multidimensional Bloom Filter having a tree like structure. When the false positive probability of the root Bloom Filter becomes 1, the size of Bloom Filter is changed dynamically. The removal of false positive is inversely proportional to the performance of Bloom Filter. When a Bloom Filter become full it gives more false positive. So, scalability is also an issue.

Bloom Filter is classified mainly into four key categories [27], namely, standard Bloom Filter, counting Bloom Filter, Hierarchical Bloom Filter, and multidimensional Bloom Filter. Standard Bloom Filter has scalability and deletion issue, and that's why counting Bloom Filter is initiated. Counting Bloom Filter creates numbers of counter and increment the counter upon same item insertion. Similarly, counting Bloom Filter decrements the counter's value to delete an item. Therefore, the false positive and false negative

probabilities are more in counting Bloom Filter. However, counting Bloom Filter consumes lesser amount of memory than standard Bloom Filter but consumes more memory when counting Bloom Filter stores fingerprint of the item. Hierarchical Bloom Filter is a forest/tree structured Bloom Filter to increase more scalability than counting Bloom Filter while reducing the false positive and false negative probabilities. Thus, hierarchical Bloom Filter is deployed in very large scale data computing. For instance, Bioinformatics. On the contrary, the multidimensional Bloom Filter is developed in multidimensional array to speed up the lookup, insertion and deletion. It is similar to standard Bloom Filter except Bloom Filter array.

Bloom Filter is a simple membership checking data structure. Its deduplication ability is the most attracting feature. It found its applications in many domains. Bloom Filter is used in the flash memory to maximize the data storage [27]. Debnath et al. [28] proposed a flash memory based storage called BloomFlash. It utilizes Bloom Filter to reduce the RAM storage space. Droplet [29] is a distributed deduplication storage system. It uses Bloom Filter to avoid unnecessary disk access. In Metadata server, Bloom Filter helps to eliminate unnecessary checking by the nodes that does not contain the requested data [30]. Singh et al. [31] proposed a fuzzy-enabled folding approach using Bloom Filter to provide Bloom Filter-as-a-service. It is used for big data storage in the Cloud. In the field of security, for example DDoS (Distributed denial of Service) attack, Bloom Filter is used to determine legitimate users [32]. Another example is biometric template protection [33].

Bloom Filter is gaining popularity with the emerging of Big Data era. Searching or processing huge volume of data inversely affects the performance of the system. However, deploying Bloom Filter eliminates the processing of the duplicate data.

A. ANALYSIS

Let $\mathbb{B}$ be a Bloom Filter, $\mathbb{S}$ be a set, n be the total items entered into Bloom Filter, m be the size of $\mathbb{B}$, k be the total number of hash functions. The probability of bits to be '1' is

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right) \tag{1}$$

F. Grandi [34] derives the false positive probability. Let X be the random variable representing the total number set bits '1' which is conditioned by $X = x$, then Probability of false positive be

$$Pr(FP|X = x) = \left(\frac{x}{m}\right)^k \tag{2}$$

Then, the exact false positive probability of standard Bloom Filter is

$$FPP = \sum_{x=0}^{m} Pr(FP|X = x)\, Pr(X = x) \tag{3}$$

$$= \sum_{x=0}^{m} \left(\frac{x}{m}\right)^k f(x) \tag{4}$$

where $f(x)$ is probability mass function. F. Grandi [34] derive $f(x)$ using $\delta - transformation$ and derives the FPP as

$$FPP = \sum_{x=0}^{m} \binom{m}{x} \sum_{j=0} x\,(-1)^j \binom{x}{j} \left(\frac{x-j}{m}\right)^{kn} \tag{5}$$

Equation (5) is the exact FPP of the standard Bloom Filter.

B. ISSUES AND CHALLENGES

1) BLOOM FILTER

Along with false positive and false negative, some more issues need to be handled while implementing Bloom Filter. The issues And challenges are discussed below:

- False Positive: It is one of the main issues in Bloom Filter. False positive is the situation when data are not inserted into Bloom Filter, but upon query operation, it returns the response as true. It introduces an query overhead. Usually, false positive responses lead to overlook important data. For instance, deduplication filter remove unique data due to false positive. In Metadata server [30], the data requested by the client are checked with Bloom Filter present in every node in the cluster. After Bloom Filter returns true, the node search for the metadata and returns the metadata to the client. Suppose the Bloom Filter gives a false positive response, then during searching metadata is not found by the node. Hence, a false positive is an overhead for the application implementing Bloom Filter. Many variants of Bloom Filter are proposed with lower FPP, but complete elimination is impossible.
- Scalability: It is an important issue in Bloom Filter. Today's Big Data era, scalability of Bloom Filter is a necessity. The standard Bloom Filter is non-scalable. Bloom Filter gives more false positive response towards saturation of the data structure. Moreover, a large size Bloom Filter takes more memory. Hence, many variants use the hierarchical Bloom Filter to increase scalability. On the other hand, scaleBF deploys chaining hash data structure to increase the scalability [35].
- Deletion: The standard Bloom Filter [11] does not permit the deletion operation. Many variants of Bloom Filter permit deletion. For example, during deletion operation Counting Bloom Filter [36] decrements the count of the data. And, delete the data when the count becomes zero. However, deletion operation introduces the false negatives in Bloom Filter. But, Bloom Filter gives more false positive response when the data structure becomes saturated. Therefore, a Bloom Filter with efficient delete operation is required.
- Hashing: The primary efficiency of Bloom Filter depends on the hashing technique used to obtain the locations. Also, a different hashing technique is used to increase efficiency in a single Bloom Filter. Moreover, the complex hashing technique is used to mitigate the FPP. However, the utilization of the complex hashing techniques increases the overhead of Bloom Filter.

TABLE 1. Listing of features and limitations of bloom filters (I).

Bloom Filter	Features	Limitation
Counting Quotient Filter [38]	• A Quotient Bloom Filter variant • Smaller and faster with the skewed inputs compared to uniform random inputs • First creates RSQF and later convert to CQF • RSQF does only two operations, RANK and SELECT • In case of storage in SSD, insert and query operation takes $O(k)$ I/O • Adding counter to RSQF is called CQF • CQF added cache-efficient variable-sized counters • CQF performs in-memory insertion and query operation • CQF has good locality of reference, and supports deletion, resizing, merging and high concurrency	RSQF: • Not cache efficient • Maintains three Bloom Filters • Time complexity of insertion and query operations is $O(n)$ CQF: • Time complexity of insertion and query operations is $O(n)$
Blocked Bloom Filter [39]	• Cache-efficient • Consist of multiple small standard Bloom Filter • Standard Bloom Filter size=Cache-line size • Stored in the cache-line-aligned for better performance • One cache miss for every operation	• Usage of table for choosing precomputed k-bits may causes table collision in case of two items obtaining same hash values • Collision leads to increment in FPP • When table occupies whole cache, then every case causes cache fault

Therefore, the performance of Bloom Filter depends on the utilization of an efficient hash function.

- Dependent on number of hash functions: Bloom Filter also depends on the number of hash functions. The number of hash functions i.e k cannot be too large or too small. A large value of k leads to more hash functions. Hence, increasing the time complexity. Again, smaller values of k leads to more FPP. Therefore, every Bloom Filter determines the value of k experimentally.

2) DNA SEQUENCER

Some issues and challenges are outlined that a DNA sequencer encounters while implementing Bloom Filter.

- False Positive:
  - Inaccuracy: Bloom Filter gives a false positive response. It leads to introducing false edges and branches in the de Brujin Graph [37].
  - Large size: Some methods use the large Bloom Filter to tackle the issue of false positive. Due to the inability to identify the error-free and erroneous reads. Both need to be saved. However, the large Bloom Filter also requires more memory. This huge memory usage affects the efficiency of the Assembling system.
- Bloom Filter size:
  Bloom Filter size needs to be proportional to the number of unique k-Mars in red. However, identifying unique k-mers are mostly avoided as it requires more additional computation and memory. Therefore, finding Bloom Filter size is complex.

IV. VARIANTS OF BLOOM FILTERS

Bloom Filter improves the performance of the DNA Assembly methods. However, the available variants of Bloom Filter may not be able to efficiently support the DNA Assembling methods. Hence, new variants of Bloom Filter are introduced to handle such genomic reads. In this section, some variants of Bloom Filter used in DNA Assembly methods are explained. Moreover, the features and the limitations of Bloom Filter are also listed in the Table 1 and 2. The Table 3 illustrates the type of Bloom Filter, FPP, time complexity and the technique followed to increase the scalability of Bloom Filter.

A. COUNTING QUOTIENT FILTER

Pandey et al. [38] proposed a counting Bloom Filter, called Counting Quotient Filter (CQF). CQF is scalable, space-efficient and gives good performance even for highly skewed distributed input. An improved variant of the quotient Bloom Filter is also proposed, called Rank-and-select-based quotient filter (RSQF). RSBF is later converted to CQF. RSQF have three Bloom Filters, namely, quotient, remainder and occupied. RSQF divides the total hash bits into quotient(q) bits and, remainder (r)bits. The quotient (q bit string) and, remainder (r bit string) is stored in the quotient and remainder filters respectively. RSQF only have two operations, RANK and SELECT. RANK operation counts the number of slots having 1, till location l in quotient filter. l is the hash value of an item in quotient filter. SELECT operation finds the slot location of the lth 1 in quotient filter. The time complexity of RANK and SELECT operation is $O(n)$. To reduce the time complexity, RSQF uses offsets array. RSQF is not cache efficient. Moreover, the insertion and query time complexity is $O(n)$. RSQF uses many array for k-mer storage which leads to more space consumption.

CQF is RSQF with cache-efficient variable-sized counters. It saves space by replacing remainder of RSQF with counters. In CQF, counter of each slot is stored immediately after the slot. Counters count the frequency of the item, CQF does in-memory insertion and query operation. Size of CQF is $O(xlog\frac{nM}{\delta x^2})$ where n = capacity, δ = FPR, M = number of

TABLE 2. Listing of features and limitations of bloom filters (II).

Bloom Filter	Features	Limitation
Bloom Filter Trie [41]	• Extended version of the [42] • Based on a burst trie [43] • Allows any format for the input genomes • Has all operations for storage, traversal, and search of pan-genome • All implemented arrays are dynamic • BFT vertices are a list of containers (compressed or uncompressed)	• Usage of many arrays requires more processing and space complexity
k-mer Bloom filters [45]	• Two kBFs are defined, namely, one-sided kBF(1-kBF) and two-sided kBF(2-kBF) • Using the 2-kBF to lower the FPR • Uses neighbors of the k-mer to check the result of Bloom Filter	• Has edge k-mer problem • Edge k-mer gives false negative results • Compared to standard Bloom Filter, 1-kBF and 2-kBF have high query time complexity • In case of true positive results the query time increases further
Allsome sequence bloom trees [47]	• An improvement in the Sequence Bloom Tree (SBT) [46] data structure • Implements clustering technique for better tree construction • Has a different presentation of the internal nodes of the SBT tree for earlier pruning and faster coverage of the search space • Builds Bloom Filter on the query for faster execution of the queries • Bitvectors are compressed to reduce disk space using RRR [48]	• All Bloom Filters are stored in the disk • In large query, SBT-ALSO cannot perform bitwise operations efficiently on bitvectors due to RRR compression • SBT-ALSO maintains two compressed trees (ROAR and RRR) leading to waste of memory • SBT gives more false positive results towards the leaves as it gets saturated • Has false negative issues

TABLE 3. Comparison table of Bloom Filters. Some symbol meaning is mentioned in their respective section.

Bloom Filter	Type of Bloom Filter	False positive	Time Complexity	Scalability
Counting Quotient Filter	Counting	RSQF: FPP = 2^{-r} Where r = number of remainder bits	Insertion = Query = $O(n)$	Medium
Blocked Bloom Filter	Standard	FPP = $\sum_{i=0}^{\infty} Poisson_{B/c}(i) \cdot f_{inner}B, i, k$ B/c = Poisson distribution parameter $f_{inner}B, i, k$ = local FPP	Insertion = Query = $O(k)$	High
Bloom Filter Trie	Hierarchical	Yes	Insertion: • Compressed: $O(log_2 c)$ • Uncompressed: $O(d + 2^{len_{P1}} + 2a)$ Query: • Compressed: $O(log_2 c)$ • Uncompressed: $O(height_{max(t)} \cdot \lceil(\frac{\lvert A\rvert^l}{c})\rceil \cdot (f + 2^{\lvert\alpha\rvert} + q))$	Low
k-mer Bloom filters	Standard	1-kBF FPP = $(1 - (1 - \frac{1}{m})^{hn})^h$ m =length of Bloom Filter, h =number of hash functions, n = unique items 2-kBF FPP = $FPP_{1-kBF} \cdot (1 - (1 - FPP_{1-kBF})^4)^2$	–	Low
Allsome sequence bloom trees	Hierarchical	Yes	Insertion= Query= $O(log(n))$	Low

items inserted and x = number of distinct items. CQF has good locality of reference, and supports deletion, resizing, merging and high concurrency. However, the time complexity for insertion and query is $O(n)$ similar to RSQF.

B. BLOCKED BLOOM FILTER

Putze et al. [39] proposed a cache-efficient variant of Bloom Filter called Blocked Bloom Filter (BLO). BLO is implemented in a LightAssembler DNA Assembly technique [40]. BLO consist of multiple small standard Bloom Filters refereed as Bloom Filter blocks. Each Bloom Filter block can fit into a single cache-line. Moreover, Bloom Filter blocks are stored in the cache-line-aligned for better performance. During the insertion of an item, hash value selects a Bloom Filter block. And, insertion in Bloom Filter block follows standard Bloom Filter insertion procedure.

This technique leads to one cache miss per operation. Instead of multiple hash functions m, a single hash function is used to choose m-bit pattern from a table of random precomputed pattern. This method helps to implement the random precomputed pattern using few SIMD (Single instruction, multiple data) instructions. In addition, multi-blocking method is also implemented. During query operation, multi-Bloom Filter blocks are searched. The use of table for choosing precomputed m-bits may cause table collision when two item obtain same hash values. This collision leads to increase in FPP. Moreover, when the table occupies whole cache, then every case causes cache fault.

C. BLOOM FILTER TRIE

Bloom Filter Trie (BFT) [41] is the extended version of [42]. It is a novel succinct data structure used for indexing and compressing the pan-genome. Pan-genome have hundreds or thousands of strains that share same sub-sequence genes. BFT is based on a burst trie [43]. A bit array called *Color* represents the colors with bit set to 1. The arrays in BFT are dynamic. In BFT, the vertex is a list of container. BFT has zero or more compressed containers, and zero or one uncompressed containers. An uncompressed container for vertex v is a set of tuples $< s, color_{ps} >$ where s = suffix, p = prefix which is the path from root to vertex v and $|s| + |p| = k$. A new compressed container stores the prefix. And, a new uncompressed container stores the suffixes with its attached colors as the children of the compressed container. Uncompressed container replaces by compressed container by bursting. It does the following (a) $a \geq cap$ where a = number of suffix prefixes and cap = capacity (b) links to children containing the suffixes, and (c) reconstruct the suffix prefixes and finds the corresponding children. A compressed container has four data structures, namely, *quer*, *pref*, *suf*, and *clust*. *quer* is a bit array Bloom Filter. It is used to record and filter suffix prefixes(a). The *pref* is a bit array recording the presence of the prefix. The *suf* is an array storing the suffix of the s_{pref} sorted in ascending lexicographic order. The *clust* is an array representing cluster starting point.

In query operation, a k-mer(say m) presence is checked by traversing BFT from the root. A k-mer is a short fragment of the DNA of length k [44]. It is a subsequence of k length. It is explained in detail in the article in the de Bruijn graph section. Each container is checked to find m_{suf}. In case of compressed container, the *quer* array is searched. If the response is true, the corresponding children are searched. If the response is false, then m is also not present in other containers. When the container is uncompressed, *BinarySearch*() is used. During insertion, first the presence of k-mer is checked. If present, then its color is modified. If absent, the query algorithm stops the trie traversal on the container c which should contain the k-mer. If c is uncompressed it takes $O(log_2 c)$ time for insertion. In compressed container, the worst time complexity is $O(d+2^{len_{p_1}}+2a)$. The query time is $O(height_{max(t)} \cdot \lceil (\frac{|A|^l}{c}) \rceil \cdot (f + 2^{|\alpha|} + q))$.

D. K-MER BLOOM FILTERS

Pellow et al. [45] used the non-independence of the k-mers to develop a k-mer Bloom Filter (kBFs). Two kBFs are defined, namely, one-sided kBF(1-kBF) and two-sided kBF(2-kBF). The 1-kBF searches for the presence of a single overlapping neighbor during a set containment query. First, a read is searched in 1-kBF. If returns true, then all reads in the set are inserted to 1-kBF. The 2-kBF lowers the FPP than 1-kBF. However, 2-kBF has edge k-mer issue. The edge k-mer is the k-mer at the boundary of the read. The edge k-mer gives false negative results. 2-kBF avoids such situation by including a hash table. The hash table maintains a list that stores the edge k-mers. During insertion of a read, the first and last k-mers are stored. After insertion of all reads, the stored k-mers are verified for an edge k-mers. If the result is true, it is stored in the final edge k-mers table. This table also includes the k-mers adjacent to zero coverage regions. However, compared to standard Bloom Filter, both kBFs have high query time complexity. Moreover, in case of true positive results the query time increases further.

E. ALLSOME SEQUENCE BLOOM TREES

Sun et al. proposed an improvement in the Sequence Bloom Tree (SBT) [46] called the AllSome Sequence Bloom Tree (SBT-ALSO) [47]. It combines three ideas: (a) use of clustering for better tree construction, (b) different presentation of the internal nodes of the BST tree for earlier pruning and faster coverage of the search space, and (c) building Bloom filter on the query for faster execution of the queries. SBT is a binary tree of compressed Bloom Filter. Repeatedly, k-mers are inserted to SBT during the Bloom Filter construction. To insert a new k-mer (say x), first Bloom Filter $BF(x)$ is found. Finding Bloom Filter is based on calculating the Hamming distance between the associated database element of the leaf nodes. After finding $BF(x)$, walking from the root to the leaves is performed and x is inserted at the bottom of SBT by following some rules. The rules are (a) node (n) has a single child, then insert x as second child (b) n has two children, then BF(x) is compared with the $BT(left(\text{n}))$ and $BT(right(\text{n}))$ of the left left(n) and right right(n) children of n. The child having more similarity becomes the current node. This process is repeated till x at its appropriate place. (c) n has no children, then x becomes n's sibling.

SBT-ALSO combines three ideas to improve the SBT (a) using clustering for better tree construction (b) different presentation of the internal nodes for earlier pruning and faster coverage of the search space, and (c) building Bloom Filter on the query for faster query execution. In SBT-ALSO, $A_{\cup}(x)$ is divided into three sets: $A_{all}(x)$, $A_{some}(x)$ and $A_{\cap}(parent(x))$. When read set L is given for a query operation, then each k-mer is hashed to find the list of Bloom Filters. These position indices are stored in an array called the list of *unresolved* bit positions. SBT-ALSO maintains two counters, namely, present and absent counter. The present counter counts the number of present bit positions i.e. 1. The absent counter

counts the number of absent bit positions i.e. 0. The query operation does comparison in a recursive manner. The L is compared with each node n, all *unresolved* 1 valued bit in A_{all} are removed from the *unresolved* list and the present counter is incremented. Similarly, all *unresolved* 0 valued bits in A_{some} are removed from the *unresolved* list and the absent counter is incremented. If the present counter is at least $\theta|\mathrm{L}|$ where $0 \leq \theta \leq 1$, then add *leaves*(n) to the list of θ-matches and terminate the search. The θ-match is defined as L θ-matches B if $|m \in L : m \text{ exits in } B|/|L| \geq \theta$. If the absent counter$> (1-\theta)|L|$ then L will not θ-match any of the leaves of the subtree of the n and terminate the search. When both the counter does not satisfy the conditions, then the two counters and list of *unresolved* bits are recursively passed to its children. Reaching the leaf the *unresolved* list becomes empty and the search terminates.

SBT-ALSO compress the bitvectors to reduce disk space using RRR [48]. In large query, SBT-ALSO cannot perform bitwise operations efficiently on bitvectors due to RRR compression. So, ROAR [49] compression algorithm is preferred. However, ROAR is not efficient as RRR, because it takes longer I/O. Hence, SBT-ALSO maintains two compressed trees (ROAR and RRR). But this results in huge memory wastage as the SBT-ALSO tree have a huge size.

V. PREPROCESSING FILTERING

Many errors are introduced during the genomic data collection. Some examples of such errors are indels, ambiguous bases and substitutions [50], [51]. Indel [52] refers to the insertion or deletion of a nucleotide base in a DNA sequence. Substitutions means exchange of a single nucleotide. The preprocessing filtering stage tries to remove such errors to eliminate the possibility of erroneous output. Moreover, the errors depend on the Assembling platform. Many techniques are implemented in preprocessing filtering stage. Some examples are error correction, read compression, indexing, k-mer counting and many more. However, in this section only three processes are explained (Figure 1), namely, read compression, k-mer counting and error correction.

A. READ COMPRESSION

A large-scale DNA Assembly project results in obtaining tens to several thousands of genomes per species. Hence, it is creating concern for storage and transmission of genomic data. It also hinders the collaboration between research teams [53]. Moreover, it needs to be stored for a long-duration as published results may be reproduced in the future. In the year 2011 national Center for Biotechnology Information (NCBI) in the US declared to stop data submission and slow withdrawal of support from Sequence Read Archive (SRA). One of the main reasons is the huge size of data, making it unable to download, utilize, transfer [54]. Furthermore, DNA sequences contain highly similar and redundant sequences, known as pan-genomes. Hence, DNA sequences are compressed to solve the problem. However, DNA compression consists of three main issues [53], namely, compression of read IDs, base sequence and quality score.

- Read IDs: Read IDs are compressed by standard methods as they are very similar.
- DNA Sequences: It is difficult to compress due to the high redundancy across reads when the depth of sequencing is high. Moreover, they are spread over the whole file and must be lossless. The methods are classified into reference-based and de novo. Reference-based methods use the concept of similarity between the reads and a reference genome. For examples, PATHENC [55], FASTQZ [56], and QUIP [57]. The de novo methods use the concept of similarity between the reads. For instance, ORCOM [58], SCALCE [59], and FASTQZ [56].
- Quality score: The methods are classified into lossless or lossy. In lossless methods, the decompressed data matches perfectly with the original data. The methods use- (a) Correlation property: quality values correlated to their position in the read and nearby quality values. For examples, FASTQZ [56], DSRC [60], and FQZCOMP [56], and (b) convert the quality scores to values which uses fewer bits for coding. The lossy method tries to maintain higher compression rates. For example, FQZCOMP [56] and LIBCSAM [61].

DARRC [62] is a dynamic alignment-free and reference-free compression method. It extends ORCOM read sequencing [63]. It follows four steps for compression- (a) Preprocessing (b) Spanning Super Read (SSR) encoding, (c) Partition encoding and (d) Meta data and guided de Bruijn graph (gdBG) compression. SSR encoding phase uses Bloom Filter. DARRC uses a variant of de Bruijn graph, called gdBG. Keeping the growth of gdBG small while inserting a new SSR requires determining the path of k-mer extraction from a point called *start position*. Following the path from the *start position* maximizes the number of k-mers stored in the gdBG. DARRC uses Bloom Filter to implement a greedy algorithm. The algorithm detects the optimal *start position* for each SSR. The optimal *start position* is compared with other *start position* using Bloom Filter. Then, Bloom Filter updates itself with extracted k-mers.

Bloom filter Alignment-free Reference-based COmpression and DEcompression (BARCODE) [64] is a DNA sequence compression method by using cascading Bloom Filters [65]. During encoding, it assumes that all reads are unique. All reads are inserted into the first Bloom Filter of cascade Bloom Filters FP_1. A set FP stores all false positive responses returned by Bloom Filter. Another set FN stores the reads that mutated or some error occurred during storage of genomic data. All items of FP are stored in second Bloom Filter of the cascade Bloom Filter BF_2. Again FP_1 is constructed by performing query to BF_2. The process is repeated for a desired number. After the construction of the cascade Bloom Filter, the unique reads are determined by performing a query to all Bloom Filters. If a Bloom filter returns false and its index is even, then the read is considered unique. After encoding, all Bloom Filters are compressed using an off-the-shelf compression tool.

LEON [53] is a novel de novo lossless sequence compression and lossy quality compression method. It uses probabilistic de Bruijn Graph for representation of the sequences. First the de Bruijn Graph is constructed. It uses Bloom Filter for faster and memory-efficient graph construction. Using a small Bloom Filter takes less space for compression, but storage space for each read increases. Hence, Bloom Filter size depends on the depth of sequencing. The solid k-mers are inserted into Bloom Filter. Bloom Filter size is $L * b$, where L=size of large genome and b = number of bits per solid k-mer.

B. K-MER COUNTING

Speeding the generation of genomic data is achieved by producing short reads. The short read size is roughly 100 base pairs. But, it contains errors due to underlying chemical and electrical processing [66]. These errors can be removed by multiple reading in the same region. But, this method increases the processing time. Another solution is overlapping reads using coverage. Coverage is the number of reads of a given portion of the sequenced genome. In de novo Assembly, at least 10x average coverage is required for obtaining high quality results. Furthermore, biased coverage requires higher average to have sufficient representation of all regions [67]. However, this solution requires huge processing time. Many Assembling techniques address these issues by deploying k-mer counting stage. The k-mer counting is a data reduction and error removal step. In k-mer counting, the short reads are partitioned into k-mers having length 20-70 bases. Then, their frequency is counted. The k-mer having count lower than a threshold value is eliminated. This step is expensive because in some assembling technique, it takes half of the total computation time.

Mcvicar et al. [66] proposed a k-mer counting system based on FPGA-attached HMC Hybrid Memory Cube (HMC) [68]. The system uses HMC based on counting Bloom Filter. Each HMC has its own small Bloom Filter. In the system, uncompressed k-mers enter the Bloom Filter. The k-mer is hashed and kept in a bypass FIFO to output. After obtaining two-three hash values, the hash addresses are tagged to identify their origin k-mer. Then, the k-mer is checked in Bloom Filter. If present, the counter is incremented. A buffer of addresses inflight is also maintained. It is used to stall an access that targets the same Bloom Filter counter. It helps in maintaining atomicity. After Bloom Filter construction, Bloom Filter finds the minimum count and updates the counters that are not saturated. Then, those addresses are deleted from the buffer. Finally, Bloom Filter counts the actual k-mer stored in the bypass FIFO.

Turtle [69] is a cache-efficient k-mer counting tool that minimize cache misses. It tries to balance time, space and accuracy requirements to increase efficiency. It uses blocked Bloom Filter [39] to eliminate less frequent k-mers. Instead of hash, a novel sort-and-compact scheme is followed to speed up the counting process. The blocked Bloom Filter reduces cache misses by localizing the bits set for an item to a few consecutive bytes. This method takes the k-mer and its reverse complement as two representations of the same object. Two tools are proposed, namely, scTurtle and cTurtle. *scTurtle* : When a k-mer is read, Bloom Filter checks its presence. If found, the count is set as 1. This method identifies k-mer presence, at least twice. The tool maintains an array with k-mer and its count. When the number of items exceeds a threshold value, the sort-and-compact scheme is followed. The array is sorted. It places the same k-mer in consecutive slots. Then a linear traversal of the array replaces multiple items having same k-mer with one item. In this tool, the count field refers to how many items are merged. Furthermore, the sort-and-compact scheme is speedup using one-producer, multiple-consumer model with a pool of buffers. Each consumer has its own Bloom Filter. The producer extracts k-mers from the read and give to consumers. The consumers do the sort-and-compact. However, having many consumers is a bottleneck because the producer need to be more efficient than consumers. The time complexity of sort-and-compact is $\left(\frac{x}{x-1}(NlogN+N)\right)$, where N = number of k-mers and x =array size. *cTurtle* : This tool is used when the number of frequent k-mers are huge and keeping their count is infeasible. It uses counting Bloom Filter. When a k-mer is read, its presence is checked in counting Bloom Filter. If found, it means the k-mer is seen twice and it is frequent so the k-mer is written to disk. If not found the k-mer is inserted into the counting Bloom Filter. It is made cache efficiency by implementing the counting Bloom Filter as blocked Bloom Filter. The scTurtle tool shows some false positive errors and the cTurtle tool shows both false positive and false negative errors.

Squeakr (Simple Quotient filter-based Exact and Approximate k-mer Representation) [70] is an in-memory k-mer counter based on the counting quotient filter(CQF). Squeakr reads data from the disk, parses and extracts k-mer and finally insert into CQF. These operations are performed by threads. Same RAM usage is maintained irrespective of number of threads. Single thread-safe CQF helps in scaling the increasing number of threads for smaller low skewed dataset. However, it fails to scale in highly skewed datasets. Due to the presence of highly repetitive k-mers that makes the threads to acquire the same locks repetitively. This limits the usage of higher number of threads. Squeakr maintains two types of CQF, namely, local and global. In global CQF, one operation is executed to apply in many k-mers. Each thread has a local CQF. When it becomes full, the content is written to global CQF. During de Bruijn graph traversal, the Squeakr performs four queries for each k-mer. Among the four, one is true, hence, three-fourth time it spends in processing false query operations. In a large data sized DNA Assembly, wastage of such huge time is a big drawback to the efficiency of the system.

C. ERROR CORRECTION

Among all preprocessing processes, error correction is the most important step. It removes the majority errors to

implement the DNA sequencer on error free reads. During genomic data collection, error occurs due to misreading of some bases, adding new DNA fragment or omission of any DNA fragment [50], [51]. It degrades the data quality. Error correction methods are used to remove the errors. However, they take very long processing time. But, correcting one read is independent of other reads. Hence, these methods can use parallelism to achieve speedup. But, one big issue is the diverse nature of errors. They make it challenging to utilize the same method to eliminate all errors.

Heo et al [71] proposed Bloom-filter-based Error correction Solution for high-throughput sequencing reads (BLESS) based on standard Bloom Filter having higher tolerance of FPP. First, BLESS counts the multiplicity of each k-mer. Then, solid k-mers are found and stored in a hash table. All solid k-mers are inserted into Bloom Filter. Bloom Filter size is set to the total number of solid k-mers. However, finding all solid k-mers is a difficult task. Hence, finding the size of Bloom Filter is also difficult. Then, weak k-mers are converted into solid k-mers using Bloom Filter. Furthermore, BLESS removes any corrections performed due to false positive results of Bloom Filter. The k-mers are extracted that contains modified bases. The hash table is referred to check the count of the extracted k-mer. If the value differs, then it is concluded that it is caused by the false positive result of Bloom Filter. Therefore, those errors are reversed. BLESS spent the majority of processing time in multiplicity counting. Moreover, it has not considered parallelism in lots of steps. The multiplicity counting, error correction, k-mers distribution to multiple files is performed in parallel to reduce the time complexity. Furthermore, BLESS prefers longer k value, but it does not guarantee to give better results.

FADE (FPGA Accelerated DNA Error Correction) [72] is an FPGA-based error correction tool for Illumina reads. Illumina reads refers to the paired-end sequencing. Paired-end sequencing does the sequencing in both ends of a fragment and generate alignable, high-quality sequence data. BLESS [71] is the base algorithm of FADE. BLESS is the most accurate DNA error-correction tools for Illumina reads. Moreover, in FADE it reduces the false positive rate exponentially. FADE uses counting Bloom Filter (CBF) [73]. The CBF is present in the DDR3 memory. The k-mers are generated from the reads and inserted to the CBF. The CBF is implemented as a blocked Bloom Filter [39]. A blocked Bloom Filter reduces the number of DDR3 memory accesses. FADE uses a main hash function(h_{main}) to determine the solid k-mer.

LIGHTweight ERror corrector (Lighter) [74] is a voluminous and memory-efficient tool for correcting sequence errors. It is a spectral alignment method that uses pattern-blocked Bloom Filter [39] for error correction. Bloom Filters reduce the number of cache misses and improves efficiency. The input to Bloom Filter is canonicalized k-mer. The canonicalized k-mer refers to either the k-mer itself or its reverse complement based on lexicographical prior. The input reads go through three passes in Lighter. First pass takes a sample of input reads and insert into Bloom Filter, say, BF_1. In the second pass, BF_1 filters the solid k-mers and inserts into another Bloom Filter, say, BF_2. The k-mer position is classified as trusted or untrusted. If the k-mer is present in BF_1 and the number of k-mers overlapping the position are less than a certain threshold, then the position is called untrusted. Otherwise, a trusted position. All trusted k-mers are stored in BF_2. In third pass, a greedy error correction method similar to BLESS [71] is followed using BF_1 and BF_2.

VI. DE BRUIJN GRAPH

The de Bruijn graph [20] is a directed graph used to represent the overlaps between the symbols in a sequence. It was named after the Nicolaas Govert de Bruijn. The de Bruijn graph is now widely used to produce the overlap in the chromosome fragments. For the construction of the standard de Bruijn graph, first, the length of the substring (k-mer) is determined. Based on the k value, the shotgun fragments are divided into substring called reads. Each read is divided into two $k-1$-mers. The left $k-1$-mer, l, starts from the first symbol of the read. And, the right $k-1$-mer, r, starts from the second symbol of the read. Then, the de Bruijn graph is constructed by taking each unique $k-1$-mer as vertex. The edge is drawn from l to r for each read. Two vertices have multiple edges due to multiple repeated symbols. Figure 3 depicts the de Bruijn graph constructed from the given sequence (ACTACGTACGTACGA). Traversing an Eulerian walk in the graph gives the original sequence. The time complexity of the graph construction is $O(N)$ (assuming a perfect sequencing) where N is the length of the sequencing. Perfect sequencing is defined as the occurrence of a read exactly once in the fragment without any error. There are two key issues with de Bruijn graph which are- (a) when a de Bruijn graph contains multiple Eulerian walk, only one is a correct fragment. But, it is impossible to guess in de novo Assembly. (b) Gap and coverage in fragments lead to holes in the graph making the graph disconnected.

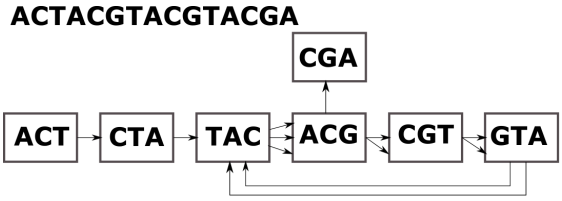

FIGURE 3. De Bruijn graph construction.

A. WHY DE BRUIJN GRAPH SHIFTED ITS FOCUS TO BLOOM FILTER?

Bloom Filter gained its popularity in the field of Bioinformatics when Melsted and Pritchard (2011) coupled Bloom Filter with traditional hash tables. It reduced the memory size drastically. Later Bloom Filter also gained popularity in the representation of de Bruijn graph. Nowadays, de Bruijn graph is widely used for representation of genome and transcriptome assembling [75]. The compacted de Bruijn graph

is appropriate to represent and index repetitive sequences. It stores the k length subsequence once in the graph in the set of non-branching and unique path [20]. For query operation indexing, the compacted de Bruijn graph requires a large memory size. In the hash table, k-mers takes at least 8 bytes, the unipath in which it occurs need 4 to 8 bytes and the offset needs 4 to 8 bytes. In addition, more data structures are also required, which incur more additional memory [76]. Hence, the focus got shifted to Bloom Filter to enhance time and space complexity for query operations. Bloom Filter is an excellent choice to solve the huge space requirement issue in de Bruijn graph.

B. VARIANTS OF DE BRUIJN GRAPH

The de Bruijn graph helps to represent the DNA sequences efficiently. Subsequently, various variants of de Bruijn graph have been proposed. In this section some variants are discussed.

1) WEIGHTED DE BRUIJN GRAPH

The weighted de Bruijn graph [77] is defined as $G = \{V, E, W\}$ where V = set of vertices, E = set of edges and W = set of weights. In the weighted de Bruijn graph, weight is associated with each edge. The number of edges between two vertices is counted and that count is assigned as the weight. In Figure 4, the number placed above the edge indicates the number of the edges between the vertices in the standard de Bruijn graph 3. The association of the weight reduces the number of edges. Moreover, the data structure of the weighted de Bruijn graph requires a tiny memory compared to the standard de Bruijn graph.

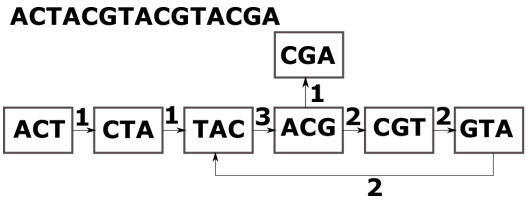

FIGURE 4. Weighted de Bruijn graph. Weight represents the number of edges.

The weighted de Bruijn graph helps in de novo transcriptome assembling. This type of assembling have to process complex graphs. The standard de Bruijn graph has many edges between the vertices, hence, it requires more memory. Therefore, the weighted de Bruijn graph is preferred.

2) COLORED DE BRUIJN GRAPH

Iqbal et al. [78] introduced the colored de Bruijn graph, a variant of the classic de Bruijn graph, which detects genotype simple and complex genetic variants in an individual or population. The colored de Bruijn graph is defined as $G = \{V, E, C\}$ where V = set of vertices, E = set of edges and C = set of colors. The set $C = \{c_1, c_2, \ldots .c_n\}$ where $c_1, c_2, \ldots .c_n$ are unique colors. The color helps in keeping track of the origin of the k-mers and $k-1$-mers as depicted in Figure 5.

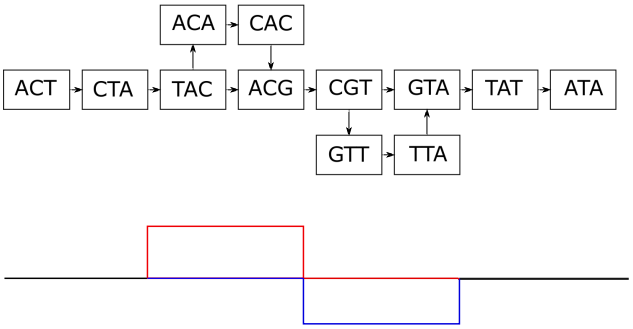

FIGURE 5. Colored de bruijn graph. black color represents common path, and red and blue color represents different path.

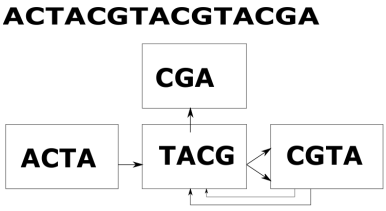

FIGURE 6. Compressed De Bruijn graph.

3) COMPACTED/ SUCCINCT DE BRUIJN GRAPH

A compressed de Bruijn graph is called compacted de Bruijn graph [79]. The compression is performed by replacing non-branching paths with a single edge. The uncompacted and its compacted de Bruijn graph are equivalent. Suppose, u and v are two nodes. If u is the only predecessor of v and v is the only successor of u, then merge u and v to a single node. After maximum compaction, every node may have at least two distinct predecessors (excluding the root node) or that node's successor has at least two distinct successors (excluding the stopping node) or its single successor has at least two distinct predecessors.

C. GRAPH CONSTRUCTION TECHNIQUES

Salikhov *et al.* [80] proposed a method to represent the de Bruijn graph using Bloom Filter. It uses cascading Bloom Filters. The set of k-mers i.e. the nodes of the graph is stored in Bloom Filter A_0. This method has bargained lower RAM usage with more disk accesses. The k-mers are read from the disk and stored in A_0. Bloom Filter B_1 is constructed by inserting k-mers from the A_0 successively. Then, each A_0 k-mer extension is checked with B_1. If it returns true the k-mer is written to the disk. Otherwise, kept in A_0 by performing the set difference operation. This operation is done iteratively by loading a part of the A_0 in RAM as the whole Bloom Filter cannot be accommodated. This operation gives the A_1 set containing the critical false positive. The critical false positive are the neighbours caused due to false positive response of Bloom Filter. Next, the k-mers from A_1

TABLE 4. Bloom filter features comparison of DNA assemblers.

Name	Type of Graph	Error correction algorithm	In-memory	Scaffolding Tool	Name of Bloom Filter	False positive
LightAssembler [40]	de Bruijn	No	No	SSPACE	Pattern-blocked	Yes
Kollector [84]	de Bruijn	No	Yes	GMAP	Progressive	Yes
ABySS 2.0 [86]	de Bruijn	Yes	No	Own	Cascading	High

are inserted to B_2 which gives the A_2 set. This process is repeated till a fixed number of times, which is determined experimentally. This method saves data on disk to reduce the RAM usage. However, the continuous interaction with the disk increases the processing time. Moreover, the k-mers are sorted externally and searching is performed using binary search. This helps in reducing the construction time, but in case of huge data set this gain is lost due to more number of disk accesses.

Faucet [81] is a two-pass streaming algorithm to assembly compacted de Bruijn graph construction. The Faucet builds the de Bruijn graph incrementally with the processing of each read. It maintains two Bloom Filters B_1 and B_2. Firstly, a read is checked with B_1. If present, insert the k-mer into B_2 otherwise into B_1. B_1 is discarded after the scanning of all reads. Then, query to B_2 gives the junction nodes. A junction node is k-mer having in-degree or out-degree > 1 or a having at least one extension which differs from the next base on the read. All junctions are inserted into a hashmap M. M maps the k-mer junction to vectors that record the each extension count. Moreover, Faucet also maintains the adjacency between the pairs of junctions using Bloom Filter. Faucet generates the compacted de Bruijn graph by a) traversing each forward extension for every special k-mer, and b) traversing in the reverse complement direction if the node is not reached, then start traversing from another node. To achieve it, Faucet query B_2 for the extensions. This is continued till next special node is reached. This traversal from special node u to v, constructs a unitig sequence c_{uv}. During construction of the C_{uv} starting from the special node u base is added at each extension till v. Cleaning, tip removal, chimera removal, collapsing of bulges, and disentanglement are performed after obtaining the raw graph.

deBGR [82] is a method to exactly represent the compacted weighted de Bruijn Graph. It is built on the Squeakr k-mer counting system [70]. It deploys counting quotient filter (CQF). The CQF helps to have a simpler traversal algorithm that allows the in-memory creation of the weighted de Bruijn Graph. This graph is manageable, simple and completely available in RAM. However, more space is utilized for efficient representation. The deBGR uses an algorithm that corrects the topology of the approximate representation and the misestimated value of abundance that caused due to the collisions from Bloom Filter. The algorithm calls three functions for each read. These three functions are represented using CQF. Bruijn Graph may contain small length cycles which hinder the correct estimated occurrences of an edge. A separate exact CQF is maintained to detect the edges involved in the cycle. The time complexity is $O(n^{1+1/log(1/4\epsilon)})$. The time complexity is much higher as compared to the weighted de Bruijn Graphs. On the other hand, the deBGR is appropriate for the applications that requires static and navigational weighted de Bruijn Graph overlooking the space complexity.

VII. DNA ASSEMBLY TECHNIQUE

In this section, various DNA Sequencers are discussed. Tabel 4 illustrates the various Bloom Filter related feature comparisons. Table 5 provides the list of advantages and drawbacks. Moreover, Table 6 exposes a list of all the genomic datasets used by the various DNA sequencer.

A. LIGHTASSEMBLER

El-Metwally et al. [40] proposed a lightweight assembling algorithm, called LightAssembler. It is based on two cache oblivious Bloom Filters. The Bloom filters are Pattern-blocked Bloom Filter [39]. One Bloom Filter has the uniform sample of the sequenced k-mers, and second has the correct k-mer determined using a simple statistical test. The use of Pattern-blocked Bloom Filter reduces the cache misses. The LightAssembler algorithm has two main modules, graph construction and graph traversal. The graph construction module has two stages uniform k-mers sampling and trust/untrust k-mer filtering. In this module, set of sequenced reads is given to Bloom Filter. It also produces the trusted k-mer file. LightAssembler does not use an error correction technique rather it passes over the sequenced reads twice to identify the trusted nodes. The graph traversal module has two steps, computing branching k-mers and simplify the de Bruijn graph, and extending the branching k-mers. In this module, the input is Bloom Filter and trusted k-mers, and the output is set of sequenced contigs. Minia's [20] traversal algorithm is implemented for graph traversal. Minia uses disk space to handle the memory usage limitation. Hence, the graph traversal latency is more. Moreover, if the disk space is not sufficient, then it gives false sequence results. Also, when the whole k-mers in the simulated datasets is given as input to the Minia algorithm, its performance reduces drastically. Moreover, Bloom Filter sometime gives a false positive response.

B. KOLLECTOR

Kucuk et al. [83] proposed Kollector, an alignment-free targeted assembly pipeline. Kollector uses a BioBloom Tools (BBT) [84] in Bloom Filter, called PBF. It takes thousands

TABLE 5. Comparison table of DNA sequencers.

Name	Advantage	Disadvantage	Discussion
LightAssembler [40]	noitemsep • Use of Pattern-blocked Bloom Filter reduces the cache misses • Implemented simplified de Bruijn graph • Identifies trusted k-mer without using counting module. Also eliminates disk-space overhead • Multithreaded program is used to achieve parallelism	noitemsep • Graph traversal latency is more • Insufficient disk space result in false sequencing results • Minia algorithm performance, reduces drastically when the whole k-mers in the simulated datasets are given as input	noitemsep • Error correction is done only for the untrusted graph, hence, it reduces the error correction input domain • It uses a library Scaffolding technique. However, own technique needs to be designed to make it more compatible with other stages of the algorithm
Kollector [84]	noitemsep • Alignment free approach • Reads thousands of transcript sequences concurrently • Sequences more genes than both iterative and non-iterative recruitment read methods	noitemsep • Bloom Filter sensitive to the arrangement of reads • Absence of assembly algorithm • Absence of scaffolds algorithm • Fails to reconstruct long length reads • Presence of false positive response	noitemsep • Error correction should be considered as false results may lead to false assumptions in the application implementing it • To achieve more efficiency it needs to implement its own algorithm in various stages of DNA Assembly.
ABySS 2.0 [86]	noitemsep • Bloom Filter in unitig stage reduces the memory usage • Bloom Filter removes majority erroneous reads • A look-ahead mechanism during the graph traversal to eliminate short branches to reduce the false positives and sequencing errors	noitemsep • Erroneous sequences are ignored • Large sized Bloom Filter used to reduce false positive • Implementation of cascading Bloom Filter requires large memory • Inability of Bloom Filter to store edges leads to more query operations	noitemsep • More efficient Bloom Filter with lower false positive rate need to be considered. • Except last other Bloom Filter are deleted. However, during usage of the cascading Bloom Filter the memory requirement is huge.

TABLE 6. Table of dataset used by DNA sequencers.

Name	Benchmarked Dataset
LightAssembler [40]	S.aureus, R.sphae, H. chr14
Kollector [84]	H.sapiens, C.elegans, Picea glauca [88]
ABySS 2.0 [86]	C. elegans

of transcript sequences concurrently and inform the localized sequencer about the corresponding gene loci. First, Kollector performs tagging. A set of genomic reads is scanned to find read pairs having user-defined amount of k-mer overlap. The overlapping is based on a threshold parameter. This tagging stage continues till PBF have a user-defined maximum number of k-mers or all reads are scanned. Second stage is pipeline. PBF is used to select the read, having a k-mer overlap based on a fixed read length. However, PBF becomes bias as it is sensitive to the arrangement of reads in the input file. Kollector relies on ABySS and GMAP for scaffolding. Hence, its efficiency depends on the efficiency of these algorithms. Kollector uses many parameters to control the number of erroneous reads. However, Kollector fails to reconstruct the reads having long length, i.e approximately 20 kbp and become bias to sequence short genes. It is caused by the inability of Kollector in identification of reads for connection of exons separated by long introns. Kollector is a greedy algorithm, hence, it selects reads of off-target regions. Moreover, it is robust in selecting relatively divergent sequences. Because, PBF initially reads from conserved regions and later starts reading from more divergent regions.

C. ABYSS

ABySS 2.0 [85] is a multi-stage de novo sequencer pipeline. It is the new version of ABySS 1.0 [86]. Implementing Bloom Filter reduces the memory usage and also removes majority erroneous sequence in ABySS 2.0 compared to ABySS 1.0. ABySS 2.0 uses cascading Bloom Filter [80] to represent the de Bruijn graph. ABySS 2.0 has three stages for DNA Assembly; unitig, contig, and scaffold. During unitig stage, the de Bruijn graph is employed to assemble sequences. The reads are passed through two passes. First pass extracts the k-mers from the reads and insert them into the Bloom Filter. After completion of k-mers insertion into Bloom Filter, except the last chain Bloom Filter that contains the error-free reads, all other Bloom Filters are discarded. Second pass identifies error-free reads. These reads are used to create the unitigs. The Minia [20] algorithm is followed for graph traversal. However, Bloom Filter is unable to store edges that results in more query operations. In each step of graph traversal, it requires four query operations for neighbor k-mers. During scaffold stage, mate-pair reads are aligned to the contigs to orient and join them to produce scaffolds, and insertion of characters at coverage gaps and resolved repeats. To maintain a low FPP, a large sized Bloom Filter is utilized. Moreover, ABySS 2.0 keeps a large FPP to have a good balance between time and space complexity. But, ABySS 2.0 need to consider other efficient Bloom Filter with low FPP to increase efficiency.

VIII. POSTPROCESSING FILTERING

The postprocessing filtering stage of DNA Assembly consists of building contigs, contig filtering, removing misassembled

contigs and scaffolding. After the graph construction, it is traversed to construct longer reads called contigs. The contigs are filtered and simplified to construct correct scaffolds. DNA sequencer refers a guide map, called Paired-end libraries, to orient the contigs in scaffolding process. It provides the information such as the positions of the contigs, their orientation and expected size of insertion. It also helps in identifying chimeric contigs [21]. In this section only the scaffolding mechanism is discussed.

A. SCAFFOLDING

The DNA sequencer produces the contigs from constructed graph. These contigs contains gaps between them. The gaps are filled to obtain the scaffolds. The scaffolding is defined as the technique of linking non-contiguous series of genomic sequences to produce the scaffold. The scaffold contains known gap length. Recently, some Scaffolding techniques are proposed using Bloom Filter.

Long Interval Nucleotide K-mer Scaffolder (LINKS) [88] algorithm is proposed that implements Bloom Filter for scaffolding. It is a method that utilizes the sequence properties of nanopore sequence data and other error-containing sequence data. Using Bloom Filter, LINK has reduced the memory consumption by 3-fold as compared to its previous versions. In LINK, initially the contigs are partitioned into k-mers. The k-mer are used for constructing Bloom Filter. Bloom Filter size is taken based on the FPP. Long reads are processed and k-mer pairs are extracted. The extraction is done at input distance and window step. When the pair is found in Bloom Filter, it helps in tracking the original contig or scaffold and determining k-mer positions and the frequency of the observation. The k-mer pair is stored in memory if they are present in Bloom Filter. These found pairs are used for further processing to complete the scaffolding process. The LINK reuses Bloom Filter for iterative runs.

Sealer [89] is a scaffolder that closes the gaps by navigating de Bruijn graphs. The de Bruijn graph is based on Bloom Filter. Konnector implements Bloom Filter to store all k-mers from the reads. The k-mers that have multiplicity higher than a certain threshold is stored in Bloom Filter. Konnector produces erroneous long reads. From the paired-end sequencing data, Konnector fills the unknown sequence between read pairs. It is achieved using redundancy in sequence coverage. Bloom Filter helps to perform a bidirectional, depth-limited, breadth-first graph search for a path connecting the flanking reads. Sealer performs local sequencing runs in serial fashion to reduce the high memory usage. At a time, one Bloom Filter is loaded. Moreover, Bloom Filter permits a subtractive procedure. The subtractive procedure eliminates the closed sequence gaps from the input of subsequent iterations. Furthermore, it reduces the CPU run time.

Dnaasm-link [90] is an application for linking the contigs. It fills the gaps with an appropriate fragment of a long DNA read. Dnaasm-link implements the dnaasm sequencer [91] for producing the contigs. Then, dnaasm-link finds the adjacent contigs. The algorithm to find the adjacent contigs uses k-mer similarity. It has three stages. In the first stage, set of k-mers are generated from the input contig set. These k-mers are inserted into Bloom Filter. In the second stage, set of k-mer pairs with a given distance is obtained. These k-mer pairs are checked with Bloom Filter. If not found in Bloom Filter, then it is discarded. Third stage determines the set of unique k-mers.

IX. OPPORTUNITY

Bloom Filter is a probabilistic data structure. It is a very simple yet powerful membership filter. The small chromosome fragments produce a large volume of information. Moreover, most of the data are repeated. Hence, it gives Bloom Filter a great opportunity to implement. The DNA Assembly has many stages where Bloom Filter can be used for deduplication and counting of the DNA fragments. Usually, the DNA assembling techniques utilize Bloom Filter in some stages. However, Bloom Filter qualifies to be used in every stage as discussed in the article. Bloom Filter is limited to membership filter yet its full potential can be utilized by the proper implementation in the DNA assembling. Moreover, same Bloom Filter can be used in every stage. Reusability of Bloom Filter reduces the processing time and saves space. Bloom Filter has the false positive issues. However, nowadays many variants of Bloom Filter are available that has lower FPP.

X. CONCLUSION

The world of Science is now focused toward solving the mystery of the living beings. The DNA is the puzzle the Science world wants to solve. The solution will provide answers to many questions and will solve many issues. Hence, DNA Assembly has gained its popularity drastically. But, it is complex and requires huge storage. The storage issue can be solved by the Big Data technology. DNA Assembly consist of mainly three stages, namely, preprocessing filtering, Graph construction and postprocessing filtering. These stages need to deal with huge volume with many repeated data. The simple and efficient Bloom Filter is capable to deal with the huge and repeated data. The DNA Assembling is very data intensive and task intensive process. Bloom Filter has less time and space complexity which can be utilized to reduce the whole processing of the DNA Assembling. Hence, DNA Assembly technique should explore thoroughly on Bloom Filter to utilize its advantages.

REFERENCES

[1] F. C. Brito, M. R. Nunes, D. R. Prata, S. F. Martha, C. Bottino, and R. G. Garrido, "DNA extraction of urinary bladder swabs collected from carbonized and decomposing corpses: Possible application in disaster victim identification," *Legal Med.*, vol. 37, pp. 15–17, Mar. 2018.
[2] A. Sofo *et al.*, "Evaluation of the possible persistence of potential human pathogenic bacteria in olive orchards irrigated with treated urban wastewater," *Sci. Total Environ.*, vol. 658, pp. 763–767, Mar. 2019.
[3] M. C. Ergören, G. Söyler, H. Sah, and E. Becer, "Investigation of potential genomic biomarkers for obesity and personalized medicine," *Int. J. Biol. Macromolecules*, vol. 122, pp. 493–498, Feb. 2019.
[4] S. Sayed, M. Nassef, A. Badr, and I. Farag, "A nested genetic algorithm for feature selection in high-dimensional cancer microarray datasets," *Expert Syst. Appl.*, vol. 121, pp. 233–243, May 2019.

[5] M. Kayser and P. De Knijff, "Improving human forensics through advances in genetics, genomics and molecular biology," *Nature Rev. Genet.*, vol. 12, no. 3, pp. 179–192, 2011.

[6] S. Goodwin, J. D. McPherson, and W. R. McCombie, "Coming of age: Ten years of next-generation sequencing technologies," *Nature Rev. Genet.*, vol. 17, no. 6, p. 333, 2016.

[7] V. Jalili, M. Matteucci, M. Masseroli, and S. Ceri, "Indexing next-generation sequencing data," *Inf. Sci.*, vol. 384, pp. 90–109, Apr. 2017.

[8] Z. D. Stephens *et al.*, "Big data: Astronomical or genomical?" *PLoS Biol.*, vol. 13, no. 7, 2015, Art. no. e1002195.

[9] J. R. Miller, S. Koren, and G. Sutton, "Assembly algorithms for next-generation sequencing data," *Genomics*, vol. 95, no. 6, pp. 315–327, 2010.

[10] D. Zerbino and E. Birney, "Velvet: Algorithms for de novo short read assembly using de Bruijn graphs," *Genome Res.*, vol. 18, no. 5, pp. 821–829, 2008.

[11] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[12] A. Singh, S. Garg, S. Batra, N. Kumar, and J. J. Rodrigues, "Bloom filter based optimization scheme for massive data handling in IoT environment," *Future Gener. Comput. Syst.*, vol. 82, pp. 440–449, May 2017.

[13] S. Geravand and M. Ahmadi, "Bloom filter applications in network security: A state-of-the-art survey," *Comput. Netw.*, vol. 57, no. 18, pp. 4047–4064, Dec. 2013.

[14] P. Xiao, Z. Li, H. Qi, W. Qu, and H. Yu, "An efficient DDoS detection with Bloom filter in SDN," in *Proc. IEEE Trustcom/BigDataSE/ISPA*, Aug. 2016, pp. 1–6.

[15] F. Guo and P. Efstathopoulos, "Building a high-performance deduplication system," in *Proc. USENIX Conf. Annu. Tech.*, 2011, p. 25.

[16] I. Nikolaevskiy, A. Lukyanenko, T. Polishchuk, V. Polishchuk, and A. Gurtov, "isBF: Scalable in-packet Bloom filter based multicast," *Comput. Commun.*, vol. 70, pp. 79–85, Oct. 2015.

[17] S. Xiong *et al.*, "kBF: Towards approximate and Bloom filter based key-value storage for cloud computing systems," *IEEE Trans. Cloud Comput.*, vol. 5, no. 1, pp. 85–98, Jan. 2017.

[18] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.

[19] C. Arnold, K. Edwards, M. Desai, J. Green, and D. Conway, "Setup, validation, and quality control of a centralized whole-genome-sequencing laboratory: Lessons learned," *J. Clin. Microbiol.*, vol. 56, no. 8, 2018, Art. no. e00261.

[20] R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson, and P. Medvedev, "On the representation of de Bruijn graphs," in *Proc. Int. Conf. Res. Comput. Mol. Biol.* Pittsburgh, PA, USA: Springer, 2014, pp. 35–55.

[21] S. El-Metwally, T. Hamza, M. Zakaria, and M. Helmy, "Next-generation sequence assembly: Four stages of data processing and computational challenges," *PLoS Comput. Biol.*, vol. 9, no. 12, 2013, Art. no. e1003345.

[22] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than Bloom," in *Proc. 10th ACM Int. Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, 2014, pp. 75–88.

[23] R. Patgiri, S. K. Borgohain, and A. Bhattacharjee, "rFilter: A scalable and space-efficient membership filter," in *Proc. 5th Int. Conf. Signal Process. Integr. Netw. (SPIN)*, Feb. 2018, pp. 478–485.

[24] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, "Scalable Bloom filters," *Inf. Process. Lett.*, vol. 101, no. 6, pp. 255–261, 2007.

[25] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The dynamic Bloom filters," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 1, pp. 120–133, Jan. 2010.

[26] A. Crainiceanu and D. Lemire, "Bloofi: Multidimensional Bloom filters," *Inf. Syst.*, vol. 54, pp. 311–324, Dec. 2015.

[27] R. Patgiri, S. Nayak, and S. K. Borgohain, "Role of Bloom filter in big data research: A survey," *Int. J. Adv. Comput. Sci. Appl.*, vol. 9, no. 11, pp. 655–661, 2018.

[28] B. Debnath, S. Sengupta, J. Li, D. J. Lilja, and D. H. C. Du, "BloomFlash: Bloom filter on flash-based storage," in *Proc. 31st Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2011, pp. 635–644.

[29] Y. Zhang, Y. Wu, and G. Yang, "Droplet: A distributed solution of data deduplication," in *Proc. 2012 ACM/IEEE 13th Int. Conf. Grid Comput.*, Sep. 2012, pp. 114–121.

[30] S. Nayak and R. Patgiri, "Dr. Hadoop: In search of a needle in a haystack," in *Distributed Computing and Internet Technology*, G. Fahrnberger, S. Gopinathan, and L. Parida, Eds. Cham, Switzerland: Springer, 2019, pp. 99–107.

[31] A. Singh, S. Garg, K. Kaur, S. Batra, N. Kumar, and K.-K. R. Choo, "Fuzzy-folded Bloom filter-as-a-service for big data storage on cloud," *IEEE Trans. Ind. Informat.*, vol. 15, no. 4, pp. 2338–2348, Apr. 2019.

[32] R. Patgiri, S. Nayak, and S. K. Borgohain. (2018). "Preventing DDoS using Bloom filter: A survey." [Online]. Available: https://arxiv.org/abs/1810.06689

[33] M. Gomez-Barrero, C. Rathgeb, J. Galbally, C. Busch, and J. Fierrez, "Unlinkable and irreversible biometric template protection based on Bloom filters," *Inf. Sci.*, vols. 370–371, pp. 18–32, Nov. 2016.

[34] F. Grandi, "On the analysis of Bloom filters," *Inf. Process. Lett.*, vol. 129, pp. 35–39, Jan. 2018.

[35] R. Patgiri, S. Nayak, and S. K. Borgohain, "scaleBF: A high scalable membership filter using 3D Bloom filter," *Int. J. Adv. Comput. Sci. Appl.*, vol. 9, no. 12, pp. 1–7, 2018. doi: 10.14569/IJACSA.2018.091277.

[36] O. Rottenstreich, Y. Kanizo, and I. Keslassy, "The variable-increment counting Bloom filter," *IEEE/ACM Trans. Netw.*, vol. 22, no. 4, pp. 1092–1105, Aug. 2014.

[37] V. Crawford, A. Kuhnle, C. Boucher, R. Chikhi, and T. Gagie, "Practical dynamic de Bruijn graphs," *Bioinformatics*, vol. 34, no. 24, pp. 4189–4195, 2018.

[38] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, "A general-purpose counting filter: Making every bit count," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 775–787.

[39] F. Putze, P. Sanders, and J. Singler, "Cache-, hash- and space-efficient Bloom filters," in *Proc. Int. Workshop Experim. Efficient Algorithms*. Rome, Italy: Springer, 2007, pp. 108–121.

[40] S. El-Metwally, M. Zakaria, and T. Hamza, "LightAssembler: Fast and memory-efficient assembly algorithm for high-throughput sequencing reads," *Bioinformatics*, vol. 32, no. 21, pp. 3215–3223, 2016.

[41] G. Holley, R. Wittler, and J. Stoye, "Bloom filter trie: An alignment-free and reference-free data structure for pan-genome storage," *Algorithms Mol. Biol.*, vol. 11, no. 1, p. 3, 2016.

[42] G. Holley, R. Wittler, and J. Stoye, "Bloom filter trie—A data structure for pan-genome storage," in *Proc. Int. Workshop Algorithms Bioinformatics*. Berlin, Germany: Springer, 2015, pp. 217–230.

[43] S. Heinz, J. Zobel, and H. E. Williams, "Burst tries: A fast, efficient data structure for string keys," *ACM Trans. Inf. Syst.*, vol. 20, no. 2, pp. 192–223, 2002.

[44] C. Sun and P. Medvedev, "Toward fast and accurate SNP genotyping from whole genome sequencing data for bedside diagnostics," *Bioinformatics*, vol. 35, no. 3, pp. 415–420, 2018.

[45] D. Pellow, D. Filippova, and C. Kingsford, "Improving Bloom filter performance on sequence data using k-mer Bloom filters," *J. Comput. Biol.*, vol. 24, no. 6, pp. 547–557, 2017.

[46] B. Solomon and C. Kingsford, "Fast search of thousands of short-read sequencing experiments," *Nature Biotechnol.*, vol. 34, no. 3, p. 300, 2016.

[47] C. Sun, R. S. Harris, R. Chikhi, and P. Medvedev, "Allsome sequence Bloom trees," in *Proc. Int. Conf. Res. Comput. Mol. Biol.* Springer, 2017, pp. 272–286.

[48] R. Raman, V. Raman, and S. S. Rao, "Succinct indexable dictionaries with applications to encoding k-ary trees and multisets," in *Proc. 13th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2002, pp. 233–242.

[49] S. Chambi, D. Lemire, O. Kaser, and R. Godin, "Better bitmap performance with Roaring bitmaps," *Softw.-Pract. Exper.*, vol. 46, no. 5, pp. 709–719, 2016.

[50] S. Meader, L. W. Hillier, D. Locke, C. P. Ponting, and G. Lunter, "Genome assembly quality: Assessment and improvement using the neutral indel model," *Genome Res.*, vol. 20, no. 5, pp. 675–684, 2010.

[51] J.-H. Choi, S. Kim, H. Tang, J. Andrews, D. G. Gilbert, and J. K. Colbourne, "A machine-learning approach to combined evidence validation of genome assemblies," *Bioinformatics*, vol. 24, no. 6, pp. 744–750, 2008.

[52] G. Lunter, C. P. Ponting, and J. Hein, "Genome-wide identification of human functional DNA using a neutral indel model," *PLoS Comput. Biol.*, vol. 2, no. 1, p. e5, 2006.

[53] G. Benoit *et al.*, "Reference-free compression of high throughput sequencing data with a probabilistic de Bruijn graph," *BMC Bioinf.*, vol. 16, no. 1, p. 288, 2015.

[54] GB Editorial Team, "Closure of the NCBI SRA and implications for the long-term future of genomics data storage," *Genome Biol.*, vol. 12, no. 3, p. 402, Mar. 2011.

[55] C. Kingsford and R. Patro, "Reference-based compression of short-read sequences using path encoding," *Bioinformatics*, vol. 31, no. 12, pp. 1920–1928, 2015.

[56] J. K. Bonfield and M. V. Mahoney, "Compression of FASTQ and SAM format sequencing data," *PLoS ONE*, vol. 8, no. 3, 2013, Art. no. e59190.
[57] D. C. Jones, W. L. Ruzzo, X. Peng, and M. G. Katze, "Compression of next-generation sequencing reads aided by highly efficient de novo assembly," *Nucleic Acids Res.*, vol. 40, no. 22, p. e171, 2012.
[58] S. Grabowski, S. Deorowicz, and Ł. Roguski, "Disk-based compression of data from genome sequencing," *Bioinformatics*, vol. 31, no. 9, pp. 1389–1395, 2014.
[59] F. Hach, I. Numanagić, C. Alkan, and S. C. Sahinalp, "SCALCE: Boosting sequence compression algorithms using locally consistent encoding," *Bioinformatics*, vol. 28, no. 23, pp. 3051–3057, 2012.
[60] S. Deorowicz and S. Grabowski, "Compression of DNA sequence reads in FASTQ format," *Bioinformatics*, vol. 27, no. 6, pp. 860–862, 2011.
[61] R. Cánovas, A. Moffat, and A. Turpin, "Lossy compression of quality scores in genomic data," *Bioinformatics*, vol. 30, no. 15, pp. 2130–2136, 2014.
[62] G. Holley, R. Wittler, and J. E. A. Stoye, "Dynamic alignment-free and reference-free read compression," in *Proc. Int. Conf. Res. Comput. Mol. Biol.*, Hong Kong, 2017, pp. 50–65.
[63] S. Deorowicz and S. Grabowski, "Data compression for sequencing data," *Algorithms Mol. Biol.*, vol. 8, no. 1, p. 25, 2013.
[64] R. Rozov, R. Shamir, and E. Halperin, "Fast lossless compression via cascading Bloom filters," *BMC Bioinf.*, vol. 15, no. 9, p. S7, 2014.
[65] K. Salikhov, G. Sacomoto, and G. Kucherov, "Using cascading Bloom filters to improve the memory usage for de Brujin graphs," *Algorithms Mol. Biol.*, vol. 9, no. 1, p. 2, 2014.
[66] N. Mcvicar, C.-C. Lin, and S. Hauck, "K-mer counting using Bloom filters with an FPGA-attached HMC," in *Proc. IEEE 25th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2017, pp. 203–210.
[67] N. S. Movahedi, E. Forouzmand, and H. Chitsaz, "De novo co-assembly of bacterial genomes from multiple single cells," in *Proc. IEEE Int. Conf. Bioinf. Biomed. (BIBM)*, Oct. 2012, pp. 1–5.
[68] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *Proc. IEEE Hot Chips 23 Symp. (HCS)*, Aug. 2011, pp. 1–24.
[69] R. S. Roy, D. Bhattacharya, and A. Schliep, "Turtle: Identifying frequent k-mers with cache-efficient algorithms," *Bioinformatics*, vol. 30, no. 14, pp. 1950–1957, 2014.
[70] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, "Squeakr: An exact and approximate k-mer counting system," *Bioinformatics*, vol. 34, no. 4, pp. 568–575, 2017.
[71] Y. Heo, X.-L. Wu, and D. E. A. Chen, "BLESS: Bloom filter-based error correction solution for high-throughput sequencing reads," *Bioinformatics*, vol. 30, no. 10, pp. 1354–1362, 2014.
[72] A. Ramachandran, Y. Heo, and W.-M. E. A. Hwu, "FPGA accelerated DNA error correction," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, 2015, pp. 1371–1376.
[73] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area Web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.
[74] L. Song, L. Florea, and B. Langmead, "Lighter: Fast and memory-efficient sequencing error correction without counting," *Genome Biol.*, vol. 15, no. 11, p. 509, 2014.
[75] B. J. Haas *et al.*, "*De novo* transcript sequence reconstruction from RNA-seq using the Trinity platform for reference generation and analysis," *Nature Protocols*, vol. 8, no. 8, p. 1494, 2013.
[76] F. Almodaresi, H. Sarkar, A. Srivastava, and R. Patro, "A space and time-efficient index for the compacted colored de Bruijn graph," *Bioinformatics*, vol. 34, no. 13, pp. i169–i177, 2018.
[77] A. Golovnev, A. S. Kulikov, and I. Mihajlin, "Approximating shortest superstring problem using de Bruijn graphs," in *Proc. Annu. Symp. Combinat. Pattern Matching*. Bad Herrenalb, Germany: Springer, 2013, pp. 120–129.
[78] Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean, "De novo assembly and genotyping of variants using colored de Bruijn graphs," *Nature Genet.*, vol. 44, no. 2, p. 226, 2012.
[79] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya, "Succinct de Bruijn graphs," in *Algorithms Bioinformatics*, B. Raphael and J. Tang, Eds. Berlin, Germany: Springer, 2012, pp. 225–235.
[80] K. Salikhov, G. Sacomoto, and G. Kucherov, "Using cascading bloom filters to improve the memory usage for de brujin graphs," *Algorithms Mol. Biol.*, vol. 9, no. 1, p. 2, 2014.
[81] R. Rozov, G. Goldshlager, E. Halperin, and R. Shamir, "Faucet: Streaming *de novo* assembly graph construction," *Bioinformatics*, vol. 34, no. 1, pp. 147–154, 2017.
[82] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, "deBGR: An efficient and near-exact representation of the weighted de Bruijn graph," *Bioinformatics*, vol. 33, no. 14, pp. i133–i141, 2017.
[83] E. Kucuk, J. Chu, B. P. Vandervalk, S. A. Hammond, R. L. Warren, and I. Birol, "Kollector: Transcript-informed, targeted *de novo* assembly of gene loci," *Bioinformatics*, vol. 33, no. 12, pp. 1782–1788, 2017.
[84] J. Chu *et al.*, "BioBloom tools: Fast, accurate and memory-efficient host species sequence screening using Bloom filters," *Bioinformatics*, vol. 30, no. 23, pp. 3402–3404, 2014.
[85] S. D. Jackman *et al.*, "ABySS 2.0: Resource-efficient assembly of large genomes using a Bloom filter," *Genome Res.*, vol. 27, no. 5, pp. 768–777, 2017.
[86] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, J. M. Jones, and I. Birol, "ABySS: A parallel assembler for short read sequence data," *Genome Res.*, vol. 19, no. 6, pp. 1117–1123, 2009.
[87] I. Birol *et al.*, "Assembling the 20 Gb white spruce *(Picea glauca)* genome from whole-genome shotgun sequencing data," *Bioinformatics*, vol. 29, no. 12, pp. 1492–1497, 2013.
[88] R. L. Warren *et al.*, "LINKS: Scalable, alignment-free scaffolding of draft genomes with long reads," *GigaScience*, vol. 4, no. 1, p. 35, 2015.
[89] D. Paulino *et al.*, "Sealer: A scalable gap-closing application for finishing draft genomes," *BMC Bioinf.*, vol. 16, no. 1, p. 230, 2015.
[90] W. Kuśmirek, W. Franus, and R. Nowak. (2018). "Linking *de novo* assembly results with long DNA reads by dnaasm-link application." [Online]. Available: https://arxiv.org/abs/1811.05456
[91] W. Kuśmirek and R. Nowak, "De novo assembly of bacterial genomes with repetitive DNA regions by dnaasm application," *BMC Bioinf.*, vol. 19, no. 1, p. 273, 2018.

SABUZIMA NAYAK is currently a Research Scholar with the Department of Computer Science & Engineering, National Institute of Technology, Silchar, Assam, India. She has published numerous papers in reputed journal, conferences, and books. Her research interests include bioinformatics, bloom filter, big data, and distributed systems.

RIPON PATGIRI is currently an Assistant Professor with the Department of Computer Science & Engineering, National Institute of Technology, Silchar, Assam, India. He has published numerous papers in reputed journal, conferences, and books. His research interests include distributed systems, file systems, Hadoop and MapReduce, big data, bloom filter, storage systems, and data-intensive computing. He is a member of ACM, EAI, and IETE.