Datenbank-Spektrum (2023) 23:211–220
https://doi.org/10.1007/s13222-023-00460-3

FACHBEITRAG

An Extension of DNAContainer with a Small Memory Footprint

Alex El-Shaikh[1] · **Bernhard Seeger**[1]

Received: 31 May 2023 / Accepted: 19 October 2023 / Published online: 30 October 2023

Abstract
Over the past decade, DNA has emerged as a new storage medium with intriguing data volume and durability capabilities. Despite its advantages, DNA storage also has crucial limitations, such as intricate data access interfaces and restricted random accessibility. To overcome these limitations, DNAContainer has been introduced with a novel storage interface for DNA that spans a very large virtual address space on objects and allows random access to DNA at scale. In this paper, we substantially improve the first version of DNAContainer, focusing on the update capabilities of its data structures and optimizing its memory footprint. In addition, we extend the previous set of experiments on DNAContainer with new ones whose results reveal the impact of essential parameters on the performance and memory footprint.

Keywords DNAContainer · DNA Storage · DNA Data Structures

1 Introduction

Due to the rapid increase in daily data produced, traditional storage devices like disks and tapes can no longer cope with these high storage demands. Even today, the overall capacity of existing data storage devices is already behind that of the data created [1]. In addition, these traditional storage devices are expensive [2] and require continuous replacement every few years due to their low durability [3]. To overcome these severe storage problems, it is of utmost importance to develop novel storage devices with substantially higher capacities and lower storage costs than existing ones.

Among recent developments, deoxyribonucleic acid (DNA) storage is one of the most promising technological trends for managing persistent data. DNA is an extremely dense biomaterial holding up to 455 exabytes per gram, and thus at least six orders of magnitude denser than current devices [3]. DNA endures several centuries and consumes around eight orders of magnitude less energy than traditional storage devices [1, 4, 5]. Despite these apparent advantages, current technologies for reading and writing DNA induce a high latency (from hours to days). However, around 80% of generated digital information worldwide is considered cold [6, 7], i.e., the data is not accessed frequently, making DNA storage a potential candidate for the management of cold data. In addition, the cost of reading and writing DNA has declined dramatically over the past years [8], and this trend is expected to continue. From a database perspective, one of DNA's serious problems is its poor support for scalable random access and the inappropriate interface for data access. To provide scalable random access, most previous approaches, such as [9], have essentially relied on the parallel use of multiple DNA storage devices, so-called tubes, as one logical storage device.

Recently, DNAContainer [10] has emerged to overcome these drawbacks of current approaches for single-tube DNA storage. DNAContainer provides an interface for reading objects from and writing to DNA storage similar to the ones of a traditional device. Based on a translation table, it overcomes the problem that there is no natural address scheme for DNA storage. Furthermore, it organizes DNA storage in units of oligonucleotides (oligos) that are contiguous subsequences on DNA and generally of fixed size. An oligo is split up into two sections. One consists of a barcode that serves as an address, and the other contains the content. Since similarities between different oligos cause contention when reading the DNA [11], DNAContainer in-

✉ Alex El-Shaikh
elshaika@mathematik.uni-marburg.de

Bernhard Seeger
seeger@mathematik.uni-marburg.de

[1] Department of Mathematics and Computer Science, University of Marburg, Hans-Meerwein-Straße 6, 35043 Marburg, Germany

troduces various coding steps to create only oligos that are sufficiently different from others. Thus, an efficient test for dissimilarity among oligos is critical in DNAContainer.

This paper addresses certain limitations and proposes further extensions to the original design of DNAContainer that has been published in [10]. First, the current interface of DNAContainer is limited to reading objects and appending new ones. So far, it is not possible to update or delete an object from DNA. This hinders the implementation of even basic data structures such as lists and arrays. Thus, we propose an extension of the interface supporting updates and deletions of objects, and we show how an arbitrary insertion in lists can be implemented using the new interface. Second, the creation of oligos is complex as it involves checks of biochemical constraints, and it also requires verifying that a new oligo is substantially different from all the other ones already kept on DNA. The current algorithm is expensive, especially regarding its memory footprint. Consequently, the size of directly accessible DNA storage is still small compared to that of traditional devices because DNAContainer hits the memory limit early. Thus, we develop new strategies for reducing the memory footprint. The basic idea is to introduce compression on locality-sensitive hashing (LSH) that is internally used for the dissimilarity check. Third, we conducted extensive experiments with varying parameter settings showcasing the usability and effectiveness of DNAContainer. We verified that DNAContainer generates DNA adhering to the required constraints for different data sets and induces only a small extra storage overhead. We also examined the scalability of DNAContainer, demonstrating its ability to generate and support up to billions of addresses, providing large-scale random access while utilizing storage efficiently.

The remainder of the paper is structured as follows. The following Sect. 2 discusses recent works and studies on DNA systems and virtual address spaces. Next, Sect. 3 introduces terminology commonly used in the context of DNA storage. Then, Sect. 4 provides an overview of the design and implementation of DNAContainer and its components. In Sect. 5, we introduce extensions and enhancements of DNAContainer. After that, Sect. 6 presents experimental results of a simulation with DNAContainer managing billions of oligos. Finally, Sect. 7 concludes the paper.

2 Related Work

In the following, we first discuss related work on DNA storage systems. After that, we focus on approaches with virtual address spaces.

In [6], relational data objects are encoded as oligos interleaved with meta-information, including table name and primary key. Polymerase Chain Reaction (PCR) is used for reading, but the same address is used for multiple records to overcome biochemical restrictions. Additional meta-information is used to filter records after fetching all oligos tagged with a specific address, resulting in a storage capacity utilization of around $\approx 16.5\%$.

In [9], 35 different files were placed in a DNA pool physically separated into tubes, storing a total of 200MB of information. Since PCR was utilized for random access, this physical separation of files was necessary to overcome the imposed limitations. Additionally, there are 35 physical addresses, each of which resembles a physical location of a single tube with one file, which significantly decreases information density over all tubes. To the contrary, DNAContainer is primarily designed for a single tube.

Fountain codes were used in [12] to encode 2.15MB of data plus 7% redundancy. Similar to our previous work [13], fountain codes provide a direct way to tune redundancy and are very practical for DNA encoding. Nevertheless, the work in [12] does not support random access at a large scale.

In [14], an alternative technology called *DORIS* is proposed to overcome the biochemical limitations yielding a larger address space at around 12,000 available addresses. However, even 12,000 addresses are insufficient to exploit the massive storage capacity of DNA.

The random access approach presented in [15] encodes data physically encapsulated in impervious silica capsules that are surface-labeled with selected DNA sequences called barcodes. These barcode labels re-emit light when excited. Hence, each file is labeled with specific barcodes and is detected by special optical channels. For example, the file “bird” can be detected with the barcode “can fly” and so on. However, special equipment is needed, and only labeled files can be detected. Nevertheless, using barcodes overcomes the severe limitations when utilizing PCR.

According to [16, 17], most recent studies do not support random access on their DNA storage system. These systems require a 5 to 3000-fold physical and logical redundancy to reduce errors, substantially reducing storage density. In addition, many DNA systems fail to encode information such that the resulting DNA is sufficiently stable for longtime archival [18]. Furthermore, we are unaware of a system with virtual address space to access data objects. Instead, a user has to provide a physical DNA address for reading an object. More complex queries beyond simple key-value queries are not supported on data collections. In particular, data structures like lists and arrays are not supported in any system, making data management difficult. Contrary to most previous approaches, we use barcodes to exploit the large available address space [13], whereas most current systems still rely on PCR and thus support only a small address space.

Computer systems have a plethora of work related to virtual address spaces. For example, a few object-oriented database systems like O2 [19] have used an address transformation table to convert unique object addresses visible to the user into internal addresses. In addition, a flash disk also offers a similar mapping known as the flash translation layer (FTL) to implement wear leveling [20]. However, the designs of these approaches do not consider the unique features of DNA storage and thus are not directly applicable.

3 Preliminaries

Today's technologies allow near-perfect DNA writing (**synthesis**) of thousands of DNA fragments in parallel. However, a small error can already lead to a significant decrease in product quality, and redundancy is introduced to avoid these errors. Thus, modern **sequencing** machines [21] read the same sequence multiple times. Both synthesizing and sequencing costs have been declining dramatically over the past years, and sequencing productivity has already outpaced Moore's law by 2008 [6]. However, sequencing machines are designed to read the entire DNA and not for random access so far.

3.1 DNA Constraints

As discussed above, sequencing and synthesizing DNA are error-prone. For example, it is well-known that DNA sequences with a too high or low number of `G`'s and `C`'s result in a high error probability in the sequencing process [22]. Hence, to reduce errors, our generated DNA codes must adhere to the following constraints:

1. The number of `G`'s and `C`'s (*GC content*) should be around 50%.
2. Consecutive repeats of the same nucleotide (*Homopolymer*) should be avoided.
3. Mutual overlaps of DNA addresses should be avoided.
4. Mutual overlaps of the oligos should be avoided.

The first two constraints considerably reduce sequencing and synthesizing errors [22]. Constraint (3) reduces contention of DNA addresses, i.e., ensures that every DNA address is treated uniquely. Finally, constraint (4) guarantees that a DNA oligo does not interact with others within a DNA library.

4 Review of DNAContainer

This section describes the architecture and functionality of DNAContainer. DNAContainer provides an interface for writing binary data to and reading it from DNA into the memory of a computer system. It manages a DNA pool consisting of oligos of fixed length L_{oligo}, similar to a block on common storage devices. Every oligo is composed of an address and a payload. Addresses are of fixed length L_{address}, and payloads are then of length $L_{\text{payload}} = L_{\text{oligo}} - L_{\text{address}}$. All sizes are given in units of nucleotides. Current DNA synthesis and sequencing costs are typically lower for shorter oligos ($L_{\text{oligo}} \leq 250$) than for longer ones [23, 24]. Thus, the size of an oligo is substantially smaller than a typical block size. Fig. 1 provides an example of an oligo of $L_{\text{oligo}} = 18$, $L_{\text{address}} = 6$, and $L_{\text{payload}} = 12$.

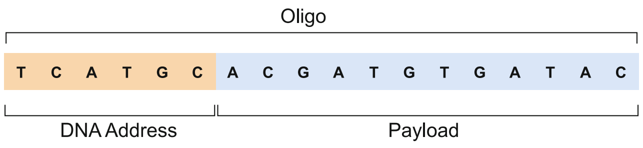

Fig. 1 An outline of an oligo in DNAContainer

Suppose a large data object like a block has to be written to DNA, exceeding the size of an oligo. Then, DNAContainer splits the data object into multiple segments, each fitting into an oligo's payload. To read the data object back from DNA, DNAContainer first computes all DNA addresses of the relevant oligos. Then, a special device such as a microarray [25] retrieves the corresponding oligos, and finally, the oligos are assembled and decoded such that the object (block) is in memory again.

In the following, we give an overview of the original functionality of DNAContainer, which can manage a set of objects in a linear address space. If objects refer to fixed-size blocks, DNAContainer offers the standard interface of block-based storage. In contrast to traditional devices, however, objects are not required to be of the same length.

Each data object written to the DNA storage is tagged with a unique integer number *Id* obtained from a linear virtual address space. Furthermore, the Id is translated to a DNA address and vice-versa (see Sect. 4.1), creating an unambiguous mapping `Id`$\leftrightarrow$`DNA address`. The Id is a virtual address visible to the user, while the associated DNA address refers to the root oligo of the object. In particular, a user can read the associated data object from DNA by simply using the virtual address. Similar to bad blocks on disks, this mapping ensures that all virtual addresses are usable, which is not valid for the underlying DNA addresses. This process is further explained in Sect. 4.1. Furthermore, the data object, i.e., the information in an oligo's payload, can be encoded with different methods that we discuss in more detail in Sect. 4.2. Fig. 2 provides an overview of the architecture of DNAContainer. DNAContainer is composed of the following main components: an address translation that maps a virtual integer identifier (Id) to a DNA address and vice-versa, an address routing that maps a DNA address to a new valid DNA address, the payload encoder,

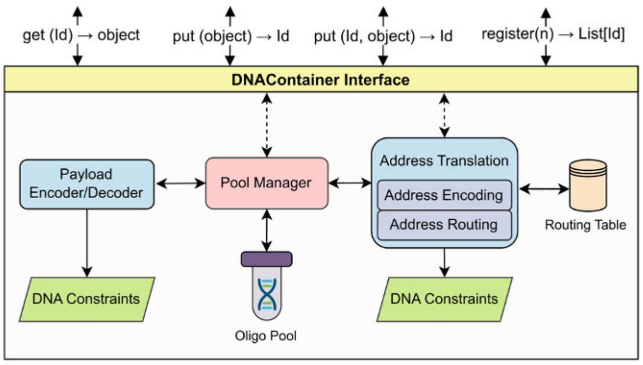

Fig. 2 Overview of DNAContainer

the payload decoder, and the DNA pool where the data is stored.

The DNAContainer interface provides an abstraction layer to the methods mentioned above. In particular, the function `put` writes a data object to DNA, the function `get` reads a data object from DNA, and `register` preregisters an Id that can be used to write a data object at a later point in time.

Given an Id, the function `get(Id)` reads the data object and returns it to the caller. Hence, `get` is the inverse of `put`. Thus, the following equality obj `= get(put(obj))` holds for every data object obj.

4.1 Address translation

The original interface of DNAContainer offers a virtual address space on integers. The put operation writes a data object into the DNA storage by generating a new Id, which is translated to a DNA address. The data object is encoded to the payload, and the oligo is formed by annealing the DNA address and the obtained payload. The following section explains the encoding of data objects as payloads and the translation of Ids to DNA addresses.

DNAContainer utilizes the method described in [26] to encode an Id to a DNA address. First, the Id is converted to a string of bytes by mapping every digit in base 10 to a byte character. Next, the string is compressed with a static Huffman code of base three. Then, each of the obtained Huffman digits is mapped to a nucleotide, forming a DNA sequence. To avoid DNA sequence being longer than L_{address}, we set L_{address} sufficiently large. Note that this method is reversible, i.e., following each mentioned step backward leads to the initial Id again.

In case the obtained DNA sequence is shorter than L_{address} or even violates the constraints mentioned in Sect. 3, we apply padding and permutations [10]. In particular, specific padding is inserted into short DNA sequences, adjusting their `GC` content and ensuring they reach the length L_{address}. Subsequently, we perform a certain number of permutations on the modified sequence, selecting the permutation that best conforms to the defined constraints. However, if none of the permutations align with the required constraints, we redirect the associated Id to a new identifier, prompting the generation of a new DNA address that complies with the constraints.

Suppose an encoded and optimized DNA address obtained for a given Id does not fulfill our constraints in Sect. 3. Then, a routing algorithm is called searching for a new Id termed Id^R such that the associated DNA address fulfills all the constraints. This search iterates Id^R over Id + 1, Id + 2, ... until it returns an appropriate DNA address. Because an Id is stable and the method register can deliver an Id without having generated a verified DNA address, a routing table maintains the mapping Id $\mapsto Id^R$. This table is kept on a traditional device. In order to read an object with Id from DNA, the method `get(Id)` first checks the routing table for a mapping Id $\mapsto Id^R$. If such a mapping exists, the DNA address is computed from Id^R. Otherwise, the DNA address is obtained from Id.

4.2 Payload Encoding

Given a data object as a stream of bytes, there are multiple encoding methods for mapping it to DNA nucleotides. For example, a straightforward method is to map every two consecutive bits of the stream to a respective DNA nucleotide, e.g., 00 $\mapsto$ `A`, 01 $\mapsto$ `C`, 10 $\mapsto$ `T`, and 11 $\mapsto$ `G`. In that case, a data object consisting of long runs of zeros or ones in its stream would result in homopolymers, violating the required constraints in Sect. 3. More sophisticated methods [12, 26–29] have been proposed, providing DNA codes that adhere to some or all the required constraints regardless of the input stream. DNAContainer allows using any of these encoding methods. Suppose the DNA code of a payload is not satisfactory. In that case, like improving DNA addresses, DNAContainer applies padding and permutations [10] to return payloads that adhere to all our constraints.

Furthermore, if the given data object is too large, meaning that the payload is greater than L_{payload}, the payload is partitioned among multiple oligos. This procedure is further detailed in [10].

4.3 Data Structures on DNAContainer

Recall that we refer to an object stored on DNAContainer as a *reference*. Moreover, DNAContainer offers support for data collections, such as arrays and lists. An array or a list is also addressed by a virtual Id, which is used to read the entire structure. The implemented methods are further explained in [10]. Furthermore, deletions and updates are not supported on references and objects within structures like arrays and lists, which we address in the following section.

5 DNAContainer Extensions

This section introduces two essential extensions of DNAContainer. First, we discuss methods for reducing its memory footprint. Then, we extend the interface of DNAContainer to support deletions and updates.

5.1 Reducing Memory Footprint of LSH

The produced DNA must adhere to every constraint in Sect. 3. In particular, constraints (3) and (4) require that DNA addresses and payloads show a sufficiently mutual dissimilarity. Thus, DNAContainer has to check the similarity of a new DNA sequence to all the previously generated sequences. If the similarity is too high, DNAContainer rejects the sequence and has to create a new one. Thus, the similarity check is a critical operation that must be performed quickly. In our original approach, we simply kept all generated sequences in memory. However, this causes high memory cost and is impractical for large databases. In the following, we briefly present our original approach, and after that, we introduce techniques for improving the memory footprint.

To efficiently support similarity searches on a set of addresses (and payloads), DNAContainer follows a common approach [30]. It employs the Jaccard similarity that splits up a DNA sequence into k-mers and uses the relative overlap between sets of k-mers for measuring similarity. Furthermore, locality-sensitive hashing (LSH) is used to approximate the Jaccard similarity [30–32]. LSH efficiently supports testing whether a newly generated DNA sequence, representing either an address or a payload, is too similar to the previously generated ones [33]. Our original approach used LSH directly to maintain the generated DNA sequences. For a new sequence S, DNAContainer first computes its hash address h using LSH, then calculates the Jaccard distance to the DNA sequences in the hash bucket h.

To improve the memory footprint, we consider the following approximate approaches. First, instead of maintaining a complete hash table, the first approach maintains a set (`Set_HA`) consisting of the hash values whose buckets contain at least one sequence. For a newly generated sequence, the hash value h is computed and tested if it is in `Set_HA`. Compared to the original approach, this approach rejects more sequences than necessary because the hash bucket h could only contain DNA sequences that are sufficiently different from S. However, as we will show in our experiments, this probability is low due to the design principle of LSH to keep only similar sequences in a bucket. Instead of maintaining a set of hash addresses, our second approach is to maintain a bitset where bit i is set if hash bucket i is occupied. In general, the memory footprint of this approach is smaller than that of `Set_HA`, but it could still become a problem for a very large number of sequences. To limit the number of bits to an upper bound n, where n is less than the number of buckets, we could use a simple mapping like $i \bmod n$. However, this would increase the false positive rate even more because one bit would no longer represent similar sequences. Another approach is to use a Bloom filter with multiple LSH hash functions instead. Though this approach would be a technical option, it is not meaningful because it would result in accepting a sequence that one of the LSH deciced to be similar to an existing one. Instead, our third approach is to use a Bloom filter on `Set_HA` and consider multiple independent uniform hash functions that map from `Set_HA` to $\{0, ..., n-1\}$. For a given false positive rate ε and a given number of sequences, we set the number of hash functions optimal such that the size n of the Bloom filter is minimized [34]. We call this method `BF_HA`. Note that our second approach is indeed a special Bloom filter with only one hash function, and thus, it is not optimal regarding memory use. In the following, `BF_HA (1)` refers to our second approach, denoting the use of one hash function.

5.2 Enhancing Data Structures

As outlined in Sect. 4, the original interface of DNAContainer only supports three operations `put`, `get`, and `register`. In the following, we extend the interface to enable updates and deletions of objects and sketch an implementation of these operations.

Recall that the routing table is responsible for mapping the virtual Ids of the objects to physical DNA addresses. Hence, to update or delete an object, e.g., stored within an oligo on DNA, its corresponding entry in the routing table is modified or deleted, i.e., the virtual address is mapped to a different DNA address. In addition, the associated oligo can be physically removed by fetching and discarding it [35, 36]. Furthermore, the original address could also be removed from LSH if DNAContainer uses the original approach for managing the addresses. However, as discussed above, the new version of DNAContainer uses a Bloom filter where the deletion of an address should have no effect. We currently recommend a complete rebuild of the Bloom filter in case of many deletions and updates. However, we examine more advanced Bloom filters [37, 38] that support deletions in our future work.

By introducing deletions and updates of objects in the interface, the new version of DNAContainer can also support dynamic and mutable data structures, further facili-

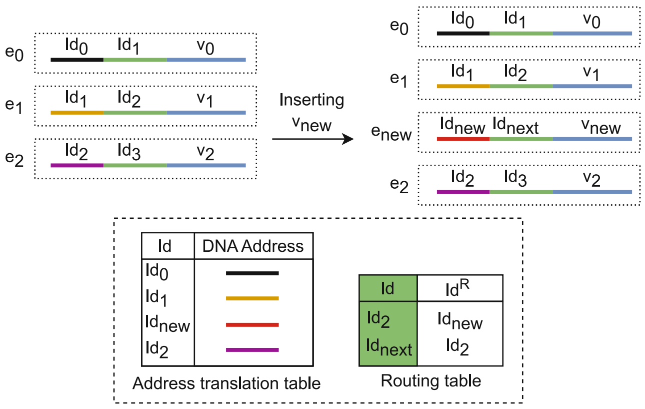

Fig. 3 Insertion of a new object into a list

tating data management. Let us consider, as an example, how to use updates in a list. In the original version of DNAContainer, it was only possible to append a new object to the tail of the list. Due to the update support in the new version, DNAContainer allows inserting objects at arbitrary positions into the list, as illustrated in Fig. 3. The initial list contains three elements e_0, e_1, and e_2, each consisting of a unique Id for addressing, an Id to reference the next element, and a value. The following steps are performed to insert a new value v_{new} into the list between e_1 and e_2. First, we create two unique Ids, Id_{next} and Id_{new}. Then, we write the pair $(v_{\text{new}}, \text{Id}_{\text{next}})$ to DNA and receive a new DNA address $addr$. We insert the pair $(\text{Id}_{\text{new}}, addr)$ into the translation table. Finally, we insert the pairs $(\text{Id}_2, \text{Id}_{\text{new}})$, $(\text{Id}_{\text{next}}, \text{Id}_2)$ into the routing table. This results in the new list displayed on the right-hand side of Fig. 3.

To read a list element for a given Id_S, we first check if an Id entry exists in the routing table. In that case, we use the associated Id for accessing the DNA storage. Otherwise, we use Id_S itself.

6 Experiments

We implemented DNAContainer in Java[1] and conducted experiments by simulating `put` and `get` operations. We used a real data set from the Global Biodiversity Information Facility (GBIF) in [39] consisting of a relational table representing information of over 500,000 species. All experiments were performed on an AMD computer with 256 logical cores (1.5–2.25 GHz each) and 1 TB of RAM. Table 1 shows the parameters and default settings in our experiments.

We examined the three different LSH structures in the following experiments and labeled them as follows. `HT` refers to the LSH structure using a hash table whose buckets contain the associated DNA sequences. `Set_HA` refers to the set of hash values from the occupied buckets of LSH only, without storing the DNA sequences. `BF_HA` refers to the optimal Bloom filter as outlined in Sect. 5.1 with default false positive probability $\varepsilon = 0.01$. Finally, `BF_HA (1)` refers to a Bloom filter with the same setting of `BF_HA`, but only one hash function is used instead of using the optimal number of hash functions.

In the following, we discuss the results from four of our experiments. First, we discuss the impact of our parameter settings for padding size and number of permutations on the `GC` content and homopolymers to satisfy our first two constraints. Then, we show the impact of these settings on the storage. Third, we show how often our generated addresses are rejected for various address sizes. Finally, we give proof of the scalability of our method. Except for the last one, all experiments first inserted 1 million records ($\approx$ 500 MB) into DNAContainer, resulting in about 14 million oligos. Every record is represented as an object with a reference in DNAContainer. A fountain code [40] is used to encode the payloads, and LSH is configured as described in [41].

First, let us discuss the results depicted in Fig. 4a and b where the average `GC` content and average longest homopolymer are plotted as a function of padding size and the number of permutations, respectively. An increase of the padding size leads to a `GC` content closer to 50%, adhering to the first constraint in Sect. 3. The padding size does not vary much because the fountain code already returns DNA codes satisfying this constraint. Thus, we decided to use 10% of the payload for padding. For other encodings, the variance of our results was indeed higher. Similarly, increasing the number of permutations lowered the homopolymers' lengths in the oligos, as shown in Fig. 5b. An increase from 8 to 16 did not significantly reduce the homopolymers' lengths. Setting the number of permutations to 8 seems to be a good choice because more permutations no longer significantly reduce the lengths of homopolymers. In addition to avoiding long homopolymers, permutations also impact the oligos' mutual overlaps. For 8 permutations, for example, we could reduce the mutual overlap compared to not using permutations by $\approx$ 25%.

In the second series of experiments, we examined the *bit rate* of oligos and payloads. The bit rate is defined as the

Table 1 Parameters and default settings

Parameter	Default Value
Address size L_{address}	80
Payload size L_{payload}	170
k-mer length	6
Number of Permutations	8
Padding size	10% of L_{payload}
False positive probability ε	1%

[1] https://github.com/alexelshaikh/DNAContainer.

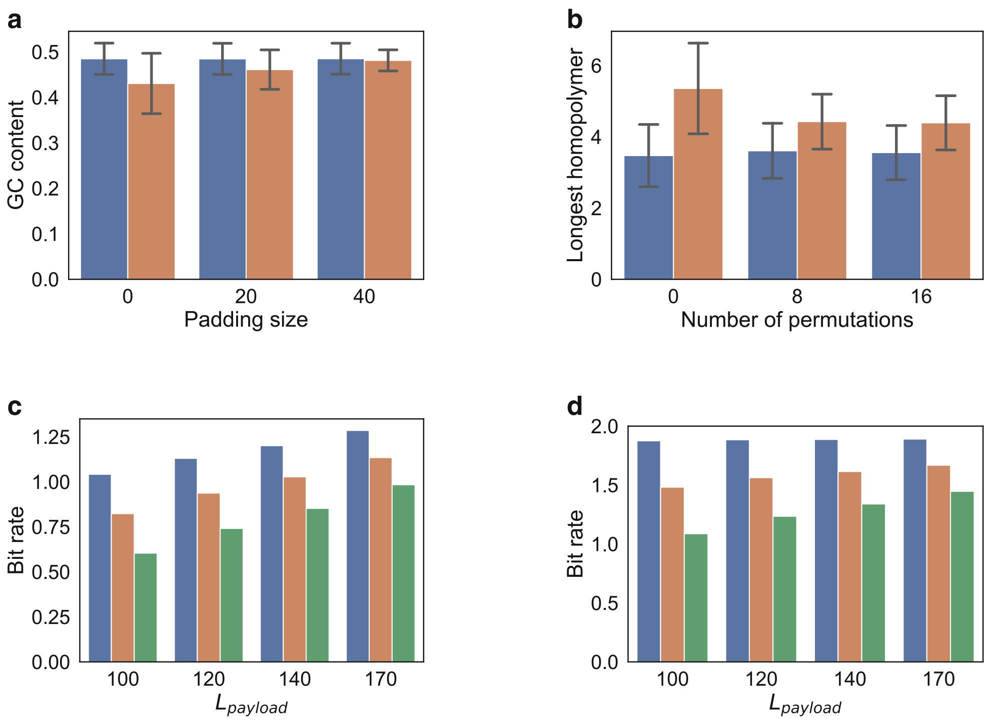

Fig. 4 The GC content, the hompolymer lengths, and the bit rate as a function of key parameters. **a** The GC content. **b** The longest homopolymer length. **c** The bit rate of oligos. **d** The bit rate of payloads

total number of bits for an object in memory divided by the total number of nucleotides used for storing the object in DNA. The upper bound of the bit rate is 2, but the bit rate will be lower due to error correction, padding, and the extra addresses in the oligos. In order to distinguish the bit rate with and without addressing, Fig. 4 depicts the bit rate of oligos and payloads in two different graphs where Fig. 4c and d show the results for the entire oligo and payload as a function of the payload size, respectively. Obviously, the bit rate of the payloads is always higher than the bit rate of oligos. As shown, the larger the payload, the higher the bit rate. DNAContainer achieves a near-optimal bit rate of $\approx$ 1.8 for payloads. The slight difference to the optimum is mainly because of padding that requires additional bits.

Since a payload of length L_{payload} represents at most $2 \cdot L_{\text{payload}}$ bits, the maximum capacity in bytes is calculated as $2 \cdot \frac{L_{\text{payload}}}{8}$. For example, by setting $L_{\text{payload}} = 170$, the payload could encode up to 42.5 bytes. Plugging in our system's capacity utilization of 1.8 bits/nucleotide (90%) yields $\approx$ 38.25 bytes per payload.

The next experiments reveal the number of invalid addresses among a given number of generated addresses for various settings of L_{address} (the size of an address). An address becomes invalid if one of the four quality constraints is not satisfied, e.g., if an address is too similar to an address previously generated. Fig. 5a displays the number of invalid addresses while creating a total of 10 million addresses for $L_{\text{address}} = 20, 40, 60, 80$. Even for the case of $L_{\text{address}} = 20$, only 50% of the computed addresses are invalid after all records are inserted. Thus, 20 million addresses are computed, of which 10 million are valid. For larger addresses, the number of invalid addresses is significantly smaller. It is interesting to note that a larger address does not always result in fewer invalid addresses. For example, our experiments return the lowest number of invalid addresses for $L_{\text{address}} = 60$. The reason is that the Jaccard distance delivers a higher similarity for larger sequences assuming a constant length of the k-mers.

Fig. 5b displays the rejection probability of a newly generated DNA address. When using a classical hash table (HT), the rejection rate is the lowest because the bucket to which a newly generated DNA address belongs might contain only dissimilar addresses. Then, the address is accepted and inserted into that bucket. This is different when Set_HA is used. It only accepts the address if the corresponding hash bucket is empty. Moreover, using a Bloom filter instead of Set_HA results in a higher rejection rate. For both Bloom filters, BF_HA and BF_HA (1), we use the common settings for the number of bits under the assumption that ε, the probability of false positives, is given [34]. However, the Bloom filter BF_HA (1), which makes use of one hash function only, shows a substantially higher rejection rate than BF_HA, the Bloom filter with the optimal number of hash functions. Thus, we no longer consider BF_HA (1) in the remaining experiments. The rejection rate of BF_HA is only slightly higher than the one of Set_HA, while the required space is significantly lower, as shown in Fig. 5c. Supplementary Figs. 1–3 further show the rejection rate and memory requirements by varying the parameters $L_{\text{address}} = 20, 40, 60, 80$ and $k = 6, 7, 8$.

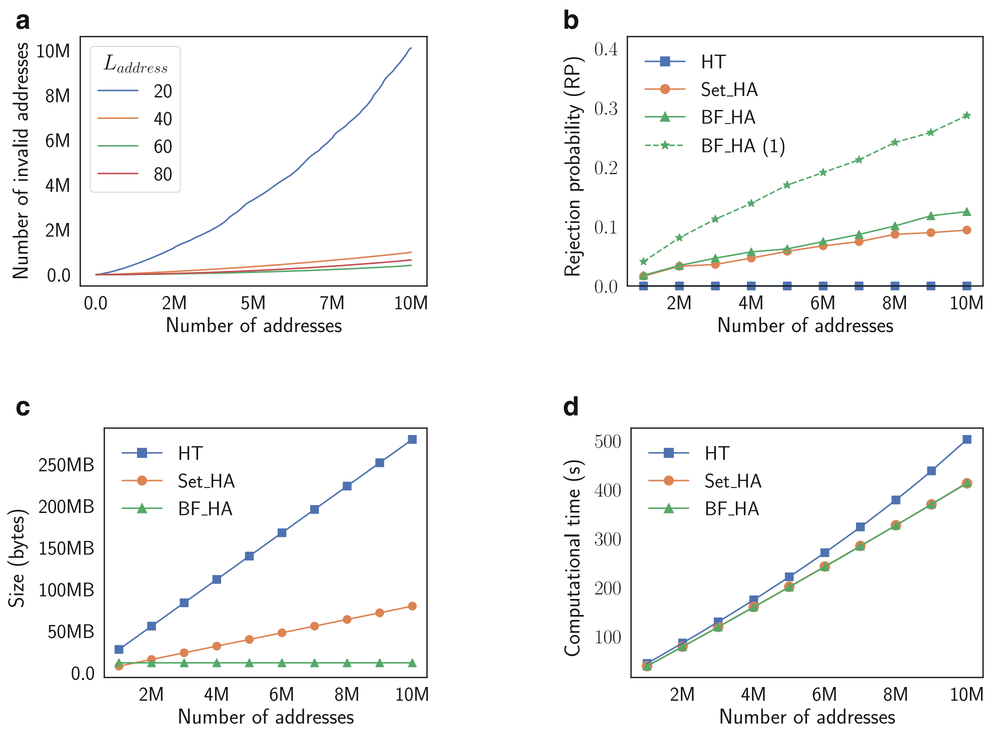

Fig. 5 The number of invalid addresses, rejection probability, size, and computational time as a function of the number of the generated addresses. **a** The number of invalid DNA addresses using `BF_HA` for $L_{\text{address}} = 20, 40, 60, 80$. **b** The address rejection probability by varying the LSH's structure. **c** The LSH structures' sizes in bytes. **d** The computational time in seconds

Furthermore, Fig. 5d depicts the required wall-clock time for generating 10 million DNA addresses for `HT`, `Set_HA`, and `BF_HA`. Despite the fact that `HT` offers the lowest rejection probability, the required computational time is the highest. The reason is the overhead of the Jaccard similarity computations when a newly generated address belongs to a non-empty bucket. Because the more addresses are generated, the higher the expected occupancy of a bucket, and thus, the additional overhead for similarity computations increases for `HT`.

In our final experiments, we examine the scalability of our approach, i.e., how many addresses DNAContainer is able to generate. For this experiment, we generated 10 billion addresses with $L_{\text{address}} = 80$, adhering to every constraint in Sect. 3. For `HT`, the memory of 1 TB was fully exhausted after generating $\approx$ 400 million addresses. With `BF_HA`, we generated 10 billion addresses in $\approx$ 6 days without running out of memory. For $L_{\text{payload}} = 170$ and a bit rate of 1.8, we could store up to $3.825 \cdot 10^{11}$ bytes or 382.5 GB of information. To the best of our knowledge, we are unaware of a larger DNA system with random access capabilities [17].

In order to increase the storage capacity even more, we could either increase the address space or exploit a larger payload for oligos. Regarding an increase in the payload, current DNA synthesis technologies only support synthesizing relatively short DNA sequences, whereas longer sequences are costly or not supported yet [42]. However, new synthesis technologies are currently under development, allowing the synthesis of several thousand nucleotides in an oligo [43]. For example, if we choose $L_{\text{payload}} = 6000$ and use 10^{10} addresses, then the theoretical storage capacity of DNAContainer would be $\approx$ 13.5 TB.

7 Conclusion

Due to its remarkable advantages regarding storage density and long-term persistence, DNA could be a promising alternative to tapes for managing cold data in the near future. In this paper, we present an extended version of DNAContainer that provides an interface for randomly accessing objects on DNA storage similar to a traditional storage device. DNAContainer provides an abstraction layer using a virtual address space with put, get, update, and delete operations on objects rather than interacting with DNA storage via low-level bio-chemical synthesizing and sequencing processes. Thus, it enables the implementation of common data structures, such as arrays and lists on DNA.

The extended version of DNAContainer provides new effective techniques with a small memory footprint for checking the similarity of DNA sequences, which is an essential requirement for achieving the stability of DNA storage. The memory savings are up to two orders of magnitude compared to the original version of DNAContainer without a major impact on the rejection rate of sequences. Thus, DNAContainer makes an essential step forward to support large-scale random access on DNA to billions of oligonucleotides.

In our future work, we will examine how to support filter queries like range queries on DNA and use a more advanced Bloom filter supporting deletions. We also continue improving the similarity computation with respect to time and memory. Finally, we are also extending DNAContainer to work on a real physical DNA device.

The online version of this article (https://doi.org/10.1007/s13222-023-00460-3) contains supplementary material, which is available to authorized users.

Funding This work was funded by the Hessian Ministry for Science and the Arts (LOEWE) in the MOSLA research cluster.

Funding Open Access funding enabled and organized by Projekt DEAL.

Conflict of interest No declared competing interests.

References

1. Li B, Song NY, Ou L, Du DHC (2020) Can we store the whole world's data in DNA storage? In: 12th USENIX workshop on hot topics in storage and file systems (hotstorage 20). USENIX Association, (https://www.usenix.org/conference/hotstorage20/presentation/li)
2. Ma TJ, Garcia RJ, Danford F, Patrizi L, Galasso J, Loyd J (2020) Big data actionable intelligence architecture. Journal of Big Data 7(1):1–19
3. Bornholt J, Lopez R, Carmean DM, Ceze L, Seelig G, Strauss K (2016) A DNA-based archival storage system. In: Proceedings of the twenty-first international conference on architectural support for programming languages and operating systems, pp 637–649
4. Zhirnov V, Zadegan RM, Sandhu GS, Church GM, Hughes WL (2016) Nucleic acid memory. Nature Materials 15(4):366–370
5. Allentoft ME, Collins M, Harker D, Haile J, Oskam CL, Hale ML et al (1748) The half-life of DNA in bone: measuring decay kinetics in 158 dated fossils. Proceedings of the Royal Society B: Biological Sciences, 279(1748), 4724-4733
6. Appuswamy R, Lebrigand K, Barbry P, Antonini M, Madderson O, Freemont P et al (2019) OligoArchive: Using DNA in the DBMS storage hierarchy. In: Biennal Conference on Innovative Data Systems Research (CIDR 2019), p 98
7. Quah J, Sella O, Heinis T (2022) DNA data storage, sequencing data-carrying DNA. arXiv preprint arXiv:220505488
8. Lin YS, Liang YP, Chen TY, Chang YH, Chen SH, Wei HW et al (2022) How to enable index scheme for reducing the writing cost of DNA storage on insertion and deletion. ACM Transactions on Embedded Computing Systems 21(3):1–25
9. Organick L, Ang SD, Chen YJ, Lopez R, Yekhanin S, Makarychev K et al (2018) Random access in large-scale DNA data storage. Nature Biotechnology 36(3):242–248
10. El-Shaikh A, Seeger B (2023) DNAcontainer: an object-based storage architecture on DNA. In: BTW 2023. Gesellschaft für Informatik e.V., Bonn, pp 773–795
11. Liu H, Bebu I, Li X (2010) Microarray probes and probe sets. Frontiers in Bioscience (Elite edition) 2:325
12. Erlich Y, Zielinski D (2017) DNA Fountain enables a robust and efficient storage architecture. Science 355(6328):950–954
13. El-Shaikh A, Welzel M, Heider D, Seeger B (2022) High-scale random access on DNA storage systems. NAR Genomics and Bioinformatics 4(1):lqab126
14. Lin KN, Volkel K, Tuck JM, Keung AJ (2020) Dynamic and scalable DNA-based information storage. Nature Communications 11(1):1–12
15. Banal JL, Shepherd TR, Berleant JD, Huang H, Reyes M, Ackerman CM et al (2020) Random access DNA memory in a scalable, archival file storage system. bioRxiv. https://doi.org/10.1101/2020.02.05.936369
16. Ceze L, Nivala J, Strauss K (2019) Molecular digital data storage using DNA. Nature Review Genetics 20(8):456–466
17. Xu C, Zhao C, Ma B, Liu H (2021) Uncertainties in synthetic DNA-based data storage. Nucleic Acids Research 49(10):5451–5469
18. Wang Y, Zhang J, Gunawan E, Guan YL, Poh CL et al (2019) High capacity DNA data storage with variable-length Oligonucleotides using repeat accumulate code and hybrid mapping. Journal of Biological Engineering 13(1):1–11
19. Deux O et al (1990) The story of O2. IEEE Transactions on Knowledge & Data Engineering 2(01):91–108
20. Ma D, Feng J, Li G (2014) A survey of address translation technologies for flash memories. ACM Computing Surveys 46(3):1–39
21. Kosuri S, Church GM (2014) Large-scale de novo DNA synthesis: technologies and applications. Nature Methods 11(5):499–507
22. Schwarz M, Welzel M, Kabdullayeva T, Becker A, Freisleben B, Heider D (2020) MESA: automated assessment of synthetic DNA fragments and simulation of DNA synthesis, storage, sequencing and PCR errors. Bioinformatics 36(11):3322–3326
23. Heckel R, Mikutis G, Grass RN (2019) A characterization of the DNA data storage channel. Scientific Reports 9(1):1–12
24. Goodwin S, McPherson JD, McCombie WR (2016) Coming of age: ten years of next-generation sequencing technologies. Nature Reviews Genetics 17(6):333–351
25. Heller MJ (2002) DNA microarray technology: devices, systems, and applications. Annual Review of Biomedical Engineering 4(1):129–153
26. Goldman N, Bertone P, Chen S, Dessimoz C, LeProust EM, Sipos B et al (2013) Towards practical, high-capacity, low-maintenance information storage in synthesized DNA. Nature 494(7435):77–80
27. Dong Y, Sun F, Ping Z, Ouyang Q, Qian L (2020) DNA storage: research landscape and future prospects. Nature Science Review 7(6):1092–1107
28. Welzel M, Schwarz PM, Löchel HF, Kabdullayeva T, Clemens S, Becker A et al (2023) DNA-Aeon provides flexible arithmetic coding for constraint adherence and error correction in DNA storage. Nature Communications 14(1):628
29. Park SJ, Park H, Kwak HY, No JS (2023) BIC codes: bit insertion-based constrained codes with error correction for DNA storage. IEEE Transactions Emerging Topics in Computing 11(3):764–777
30. Indyk P, Motwani R (1998) Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the thirtieth annual ACM symposium on theory of computing, pp 604–613
31. Broder AZ (1997) On the resemblance and containment of documents. In: Proceedings. Compression and Complexity of SEQUENCES 1997. Cat. No. 97TB100171. IEEE, pp 21–29

32. Anand R, David JU (2011) Mining of massive datasets. Cambridge University Press
33. Hua Y, Xiao B, Veeravalli B, Feng D (2011) Locality-sensitive Bloom filter for approximate membership query. IEEE Transactions on Computers 61(6):817–830
34. Luo L, Guo D, Ma RT, Rottenstreich O, Luo X (2018) Optimizing bloom filter: challenges, solutions, and comparisons. IEEE Communications Surveys & Tutorials 21(2):1912–1949
35. Mamanova L, Coffey AJ, Scott CE, Kozarewa I, Turner EH, Kumar A et al (2010) Target-enrichment strategies for next-generation sequencing. Nature Methods 7(2):111–118
36. Tomek KJ, Volkel K, Simpson A, Hass AG, Indermaur EW, Tuck JM et al (2019) Driving the scalability of DNA-based information storage systems. ACS Synthetic Biology 8(6):1241–1248
37. Tarkoma S, Rothenberg CE, Lagerspetz E (2011) Theory and practice of bloom filters for distributed systems. IEEE Communications Surveys & Tutorials 14(1):131–155
38. Rothenberg CE, Macapuna CA, Verdi FL, Magalhaes MF (2010) The deletable Bloom filter: a new member of the Bloom family. IEEE Communications Letters 14(6):557–559
39. GBIF Org User Occurrence download. The global biodiversity information facility. https://www.gbif.org/occurrence/download/0165113-230224095556074. Accessed: 26.10.2023
40. Shokrollahi A (2006) Raptor codes. IEEE Transactions on Information Theory 52(6):2551–2567
41. El-Shaikh A, Seeger B (2023) Content-based filter queries on DNA data storage systems. Scientific Reports 13(1):7053. https://doi.org/10.1038/s41598-023-34160-5
42. Heinis T, Alnasir JJ (2019) Survey of information encoding techniques for DNA. arXiv:190611062
43. Ping Z, Ma D, Huang X, Chen S, Liu L, Guo F et al (2019) Carbon-based archiving: current progress and future prospects of DNA-based data storage. GigaScience 8(6):giz75