DNAContainer: An object-based storage architecture on DNA

Alex El-Shaikh[1] Bernhard Seeger[2]

Abstract: The digital data volumes produced worldwide per year are ever-increasing. Estimates show that by 2025, we will have reached 175 zettabytes of globally created digital data. Despite today's advancements in storage devices, current database management systems cannot cope with these amounts of data. More than the recent improvements in storage technologies are needed to meet the ever-accelerating growth of generated data. This problem is further exaggerated when considering that current storage technologies such as HDD and tape require replacement every few years. To combat this deficiency, deoxyribonucleic acid (DNA) offers a novel durable (millennia scale), extremely dense, and energy-efficient storage medium. However, current DNA systems lack support for random access and more expressive query support beyond key-value lookups. In this paper, we present DNAContainer, a novel storage architecture on DNA that spans an ample virtual address space on objects, enabling random access to DNA at a large scale while adhering to required biochemical constraints. The interface of DNAContainer also facilitates the implementation of common external data structures, such as arrays and lists that store data in blocks of fixed size.

Keywords: DNAContainer; DNA Storage; Random Access; DNA Data Structures

1 Introduction

Current database management systems rely on solid state disks (SSDs), magnetic hard disk drives (HDDs), and tapes as their primary persistent memory devices [Bo16]. However, due to the dramatic increase in data produced daily, these devices will no longer cope with the amount of data soon. As stated in [Li20a], the increase in capacity of current data storage devices is already behind that of the data created. In addition, these traditional storage devices are rather expensive [Ma20] and require continuous replacement every few years due to their low durability [Bo16]. To address these severe problems, deoxyribonucleic acid (DNA) has recently been considered 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 [Bo16]. DNA endures several centuries and consumes around eight orders of magnitude less energy than traditional storage devices [Li20a, Zh16, Al12]. 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

[1] University of Marburg, Department of Mathematics and Computer Science, Hans-Meerwein-Straße 6, D-35043 Marburg, Germany elshaika@mathematik.uni-marburg.de

[2] University of Marburg, Department of Mathematics and Computer Science, Hans-Meerwein-Straße 6, D-35043 Marburg, Germany seeger@mathematik.uni-marburg.de

doi:10.18420/BTW2023-50

information worldwide is considered cold [Ap19, QSH22], 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 have declined dramatically over the past years [Li22].

Among the severe drawbacks to using DNA as storage are its unsatisfying direct access ability and its poor interface for reading and writing dedicated objects. Similar to blocks on traditional devices, DNA consists of oligonucleotides (oligos) that are contiguous subsequences, generally of fixed size. Furthermore, an oligo consists of a payload between a pair of DNA addresses, the so-called *primer pair*. The primers are addresses used for random access via Polymerase Chain Reaction (PCR) [Or18]. Due to strict biochemical restrictions on primers, only a few hundred primers exist in a DNA library, leading to a very small address space. However, there is a second approach to direct accessing oligos that uses microarrays with *barcodes* that are unique prefixes of the payloads. Then, the available address space grows up to several million [El22].

In this paper, we primarily address the second drawback that current DNA storage systems do not provide a coherent interface to write and read random data objects. The reason is that DNA does not offer a natural linear address space as it is known from disks and tapes. Instead, there is only a key-value approach that maps a data object identifier (DOI) directly to a barcode or primer. To access a requested data object, its DOI has to be translated into the DNA address before performing the actual read operation. A so-called *routing table* can manage the mappings on a traditional storage device. Furthermore, an insertion of a new object first generates a new barcode (or primer) and updates the routing table before storing the actual object on DNA. This direct approach of current DNA storage systems to accessing data on DNA has serious drawbacks. For example, the generation of barcodes is non-trivial because they have to be sufficiently different from the others.

In this paper, we introduce DNAContainer, a novel DNA storage architecture that offers a virtual address space of objects on DNA, including *put* and *get* operations, addresses translation, and rerouting of invalid addresses. Our system reads objects from DNA via microarrays that allow using the ample available address space. As a special case, DNAContainer also offers the same interface as block storage when the objects are defined as fixed-sized blocks. Furthermore, a block storage facilitates the direct implementation of essential external data structures like arrays and lists on DNA, hiding the actual complexity of a DNA device. For example, DNAContainer internally checks if the generated addresses are too similar by utilizing locality-sensitive hashing (LSH) and approximating the Jaccard similarity of two DNA sequences [IM98, Br97, Bu01, Be15]. In addition, DNAContainer supports error correction mechanisms such as Reed Solomon [RS60] to address the inherent problem of error-prone reading from and writing to DNA. Users of DNAContainer do not have to deal with these problems anymore and instead use the common interface of storage systems as it is known from other devices.

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. Section 3 introduces notation and terminology commonly used in the context of DNA storage. Section 4 provides the design and implementation of DNAContainer and its components. It further shows how to generate DNA addresses and payloads that adhere to certain biochemical constraints. In Sect. 5, we detail the implementation of basic data structures like array and list on DNAContainer. Section 6 presents experimental results of a simulation with DNAContainer managing millions of oligos. Finally, Sect. 7 concludes the paper.

2 Related Work

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

The approach in [Ap19] encodes relational data objects (records) interleaved with meta-information as oligos. The meta information contains, e.g., the table name of the record and its primary key. Reading oligos is achieved by utilizing primers and PCR. Since only a few primers are available due to the biochemical restrictions, i.e., the address space is small, the same primer is used to address multiple records. For example, to read a specific record from DNA, a pre-known primer is used to fetch all oligos tagged with that primer. The encoded meta-information is further used to return the desired record, e.g., by its primary key. Moreover, the meta-information encoded within each oligo significantly decreases the information density per oligo, and the realized storage capacity is around $\approx 16.5\%$.

In [Or18], 35 different files were placed in a separate DNA pool each, resulting in 200MB of information. Since PCR utilized random access, this physical separation of files was necessary to overcome the imposed restrictions on the limited available primers. 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.

Fountain codes were used in [EZ17] to encode 2.15MB of data plus 7% redundancy. Similar to our previous work [El22], fountain codes provide a direct way to tune redundancy and are very practical for DNA encoding. Nevertheless, the work in [EZ17] utilizes PCR for retrieving data and does not support random access at a large scale.

So far, PCR is still the standard technology to read data from a DNA pool. In [Li20b], an alternative technology called *DORIS* is proposed to overcome PCR limitations yielding a larger address space at around 12, 000 available addresses. However, even 12, 000 addresses are not sufficient to exploit the massive storage capacity of DNA.

The random access approach presented in [Ba20] 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, only labeled files can be detected. Additionally, special equipment, such as optical channels, is needed.

According to [CNS19, Xu21], most recent studies do not support random access to their DNA storage system. These systems require a 5 to 3, 000-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 long-time archival. In particular, many DNA systems fail to restore the original data objects after reading the DNA [Wa19]. Furthermore, we are unaware of a system with virtual address space to access a data object. Instead, a user has to provide a primer for reading a data object. These primers must be managed on a traditional storage device. 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. Moreover, we use barcodes to exploit the large available address space [El22], whereas most current systems still rely on PCR and primers, and thus only support a small address space.

There is a plethora of work related to virtual address spaces in computer systems. For example, a few object-oriented database systems like O2 [De90] 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 [MFL14]. However, the designs of these approaches do not consider the unique features of DNA storage and thus are not directly applicable.

A common problem of today's storage technologies is successfully restoring archived data after several decades of writing the data to storage devices [AJ20]. For example, as mentioned above, current storage technologies such as flash memory rely on FTL, which requires storing meta-information about the corrupted memory cells to keep the device functioning. Current DNA systems manage the used primers (or barcodes) on a traditional storage device and face a similar problem. That is, if the used primers and barcodes are lost, the data on DNA cannot be restored. However, for DNA, storing the required meta-information also on DNA might solve this problem. DNA is an omnipresent material, and its principal building structure has never changed over millions of years and is expected to be ubiquitous for millions of years in the future.

3 Preliminaries

DNA is a long molecule found in all known living organisms. It carries the genetic code, such as instructions, functions, and reproduction in living organisms, including some viruses. Moreover, DNA is composed of smaller units called nucleotides. A nucleotide contains one of the following nucleobases: Adenine (A), Thymine (T), Cytosine (C), or Guanine (G). These nucleotides' specific combination and order make up living organisms' different instructions and functions. Finally, DNA is composed of two polynucleotide strands of the same length that loop and twist around each other to form a double-helix. Each nucleotide of one chain pairs and forms hydrogen bonds with the corresponding nucleotide from the other strand. According to the canonical Watson-Crick pairing [Sp59], `A` binds to `T` and `G` to `C`. Hence, we say `A` is complementary to `T`, `C` is complementary to `G`, and vice versa.

In the following subsections, we will introduce key terms, such as *DNA pool* and *hybridization*, typically used in the DNA storage context.

3.1 DNA Pool and Library

A DNA pool is a collection of one or more double-stranded DNA fragments held *in-vitro*, i.e., outside of a living organism, in a single container or test tube. Typically, one container refers to a single pool, whereas multiple pools represent a library. Nevertheless, a single pool can also be referred to as a library.

3.2 Encoding and Decoding

The term encoding is used to describe the process of transforming binary data to DNA, i.e., instead of bits, the information is represented by nucleotides. On the other hand, decoding describes the transformation of nucleotides back to the *original* binary representation.

3.3 Denaturation and Hybridization

Double-stranded DNA is generally stable under physiological conditions, meaning the bonds forming the double-helix will remain bonded [Ch99, YPFK06]. However, as illustrated in Fig. 1, raising the surrounding temperature, e.g., in a laboratory, will cause the strands to separate as single-stranded DNA (ss-DNA). This process is called denaturation. Therefore, lowering the temperature will allow the ss-DNA to bind together as double-stranded DNA (ds-DNA), which is called hybridization.

3.4 DNA Synthesizing and Sequencing

DNA synthesis is writing DNA by linking and joining nucleotides together, forming a single-stranded sequence. Today's technologies allow near-perfect DNA synthesis for over 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 [KC14] 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 [Ap19]. However, sequencing machines are designed for reading an entire DNA and not for random access so far.

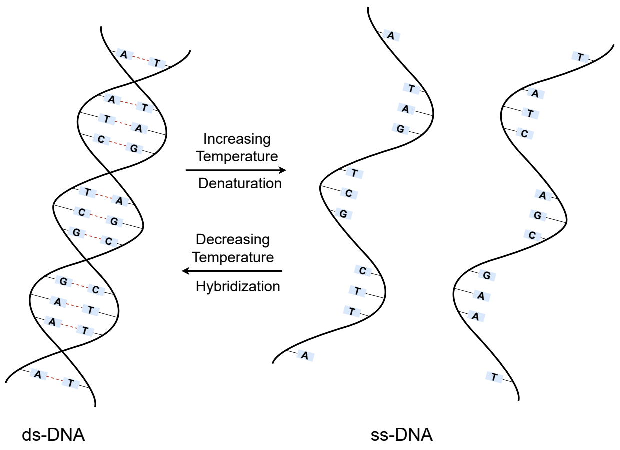


Fig. 1: DNA denaturation and hybridization.

3.5 Reading DNA with a Microarray

A microarray consists of a small surface (in the size of today's HDD) usually made of glass. It contains DNA *sites* to which DNA sequences can be immobilized or printed [Ku01]. The array can fetch a DNA sequence by printing its complementary sequence to one of its sites. These printed DNA sequences are often called *barcodes* or *probes*, but we simply call them DNA addresses. For example, a microarray with 20 printed DNA addresses can simultaneously fetch 20 oligos from an oligo pool. This procedure is done as follows. First, the oligo pool's temperature is raised, denaturing the contained DNA. Next, the denatured oligos are placed onto the microarray. Then, the temperature is lowered to allow the single-stranded oligos to hybridize to their complementary sites on the array. Finally, the array is washed, removing all the remaining oligos that did not hybridize, i.e., bind to any of the array's sites. The obtained bonded oligos are sequenced, and the data is transferred to a computer for further analysis. Note that a microarray can fetch a sequence s, even if only a complementary subsequence of s is printed to the array. Today's microarrays can contain up to several million sites [Bu13], allowing access to millions of DNA sequences simultaneously.

3.6 DNA Constraints

As described above, sequencing and synthesizing DNA is error-prone. For example, it is well-known that DNA sequences with a too high or low number of `G`'s and `C`'s causes a high error probability in the sequencing process [Sc20]. 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 and second constraints considerably reduce sequencing and synthesizing errors [Sc20]. Constraint (3) ensures that the microarray treats every DNA address uniquely. Finally, constraint (4) guarantees that a DNA oligo does not carry a DNA address as a payload.

4 The Design of DNAContainer

This section describes the architecture and functionality of DNAContainer. DNAContainer provides an interface for writing binary data to and reading it back from DNA into the memory of a computer system. It manages a DNA pool consisting of oligos of the same length L_{oligo}, similar to a block on common storage devices. Each oligo is composed of an address and a payload. Addresses are of the same length $L_{address}$, and payloads are then of length $L_{payload} = L_{oligo} - L_{address}$. Current DNA synthesis and sequencing costs are typically lower for shorter oligos ($L_{oligo} \leq 250$) than for longer ones [HMG19, GMM16]. Thus, the size of an oligo is substantially smaller than a typical block size. Figure 2 provides an example of an oligo of $L_{oligo} = 18$, $L_{address} = 6$, and $L_{payload} = 12$.

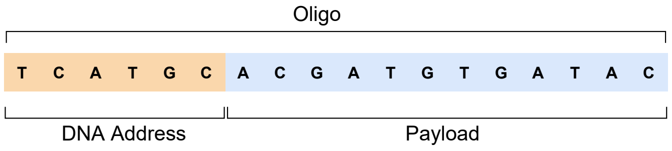


Fig. 2: The composition of an oligo in DNAContainer.

Suppose a large data object like a block has to be written to DNA, exceeding the size of an oligo. In that case, DNAContainer splits the data object into multiple segments, each of which fits 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 microarray 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 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. Rather than using block, we prefer using the generic term objects instead.

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` ↔ `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 virtual addresses are usable, which is not valid for the underlying DNA addresses. This process is further explained in Sect. 4.1.2. Furthermore, the data object, i.e.,

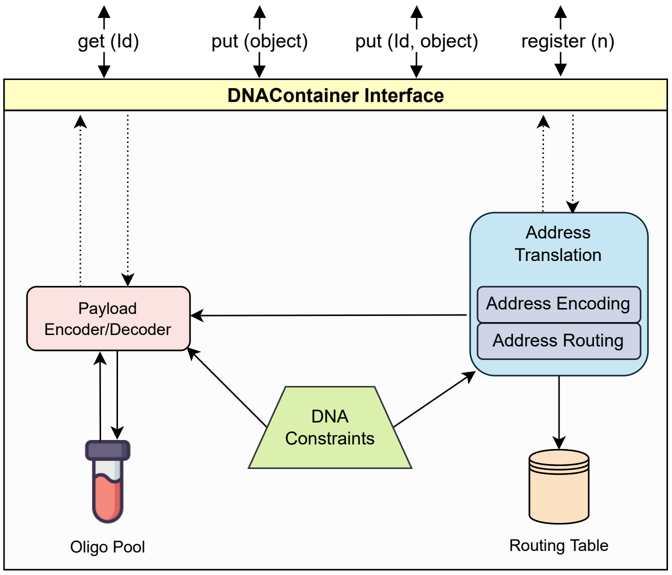


Fig. 3: Overview of DNAContainer and its components.

the information in an oligo's payload, can be encoded with different methods that we mention in more detail in Sect. 4.2. Even if the payload's encoder returns a payload that does not adhere to the constraints in Sect. 3.6, DNAContainer implements additional optimizations such that it fulfills the required constraints, which are discussed in Sect. 4.3. Figure 3 provides an overview of DNAContainter and its components. To sum up, DNAContainer is composed of the following main components: address translation to map an Id to a DNA address and vice-versa, address routing to map a DNA address to a new valid DNA address, the payload encoder, the payload decoder, and the DNA pool where the data is stored.

Our interface provides an abstraction layer to the methods mentioned above. In particular, to write a data object to DNA, we use the function `put`, the function `get` to read a data object from DNA, and `register` to pre-register an Id that can be used to write a data object at some point in the future, using that Id. The following discusses each of these functions in more detail.

register. As mentioned above, our DNA system assigns each data object a unique Id. The function `register` returns a unique Id that is not assigned a data object yet. In other words, this Id is reserved for a future data object that can be stored later. Note that the returned Id is not yet mapped to a DNA address and is only done once an actual data object is to be stored in the DNA storage. We implement the function `register(n)` that returns n newly registered consecutive Id numbers.

put. This function represents the write operation on DNA. It is used to store a data object in the DNA storage. We implement two variants of put. The first variant takes a data object, stores it in the DNA storage, and returns a newly registered unique Id. This newly registered Id is calculated by calling `register(1)`. The second variant takes an Id that was previously registered along with a data object and stores this data object in the DNA storage given the Id. We can extract the data object from the DNA storage for both variants by calling `get(Id)`. The call `put(Id,` d`)` can be used to replace the currently stored data object at Id with d. This is done by rerouting the virtual address Id (see Sect. 4.1.2), and the oligos of the old data object are not physically removed by default. However, to physically remove a data object from DNAContainer, the corresponding oligos can be fetched using `get` and discarded. Hence, the used addresses of physically removed objects can be reused.

get. This operation represents the read operation from DNA. By providing an Id to the function, `get(Id)` returns the data object associated with that Id. Hence, `get` is the inverse of `put`. For example, the following equality d = `get(put(`d`))` holds for any data object d.

4.1 Address translation

DNAContainer provides its interface based on the 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 sections detail the encoding of data objects as payloads and the translation of Ids to DNA addresses.

4.1.1 Address Encoding

We utilize the method described in [Go13] 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. The obtained DNA sequence could be longer than $L_{address}$; thus, we set $L_{address}$ to be sufficiently large.

Note that this method is reversible, i.e., following each mentioned step backward leads to the Id used.

Furthermore, the obtained DNA sequence could be shorter than $L_{address}$ or even violate the constraints mentioned in Sect. 3.6. In that case, we apply the optimizations explained in Sect. 4.3. The optimizations always return a DNA address of length $L_{address}$. However, if the sequence after optimizations does not adhere to the required constraints, we route the used Id to a new Id, which is explained in the following section.

4.1.2 Address Routing

Suppose a DNA address obtained after encoding and optimizations fails to fulfill the given constraints in Sect. 3.6. In that case, the used Id is mapped (routed) to a new Id as shown in Algorithm 1. Let us refer to the Id as Id, the new Id as Id^R, and the mapping $Id \mapsto Id^R$ as the routing table. As depicted in Fig. 3, the address translation manages the routing table $Id \mapsto Id^R$, which is stored on a traditional storage device. The routing table must be read before accessing the DNA storage. The new Id^R is encoded with the same method mentioned in the section above.

Algorithm 1: Routing Id to Id^R

```
Input: $Id$
Output: $Id^R$
1 $Id^R := Id$
2 $addr$ := asDnaAddress($Id^R$)
3 while not $constraints.adhere(addr)$ do
4     $Id^R := Id^R + 1$
5     $addr$ := asDnaAddress($Id^R$)
6 routingTable.put($Id \mapsto Id^R$)
7 return $Id^R$
```

As shown in Algorithm 1, the routing finishes once a new Id^R that fulfills the constraints is found. Note that Id and Id^R could be equal if the Id already adheres to the constraints. The algorithm iterates over the integers $Id^R = Id, Id + 1, Id + 2, \ldots$, translating each to a DNA address, only stopping once it finds Id^R of which the DNA address fulfills all the required constraints.

4.2 Payload Encoding

There are several approaches to how to encode a data object as DNA. We refer to the data object as a stream of bytes, which can be mapped to DNA nucleotides. For example, a straightforward method is to map every two consecutive bits 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 would get mapped to homopolymers, violating the required constraints in Sect. 3.6. More sophisticated methods [Go13, EZ17, Do20, El22] have been proposed, providing DNA codes that adhere to some or all the required constraints regardless of the input byte stream. For DNAContainer, any method encoding the data object to DNA nucleotides can be used because we apply optimizations (see Sect. 4.3) to return payloads that adhere to all of the mentioned constraints. However, by utilizing a fountain code [EZ17] to encode the payload, we already obtain DNA codes that obey the constraints (1) and (2) and include error correction, which leaves optimizing for the remaining constraint (4).

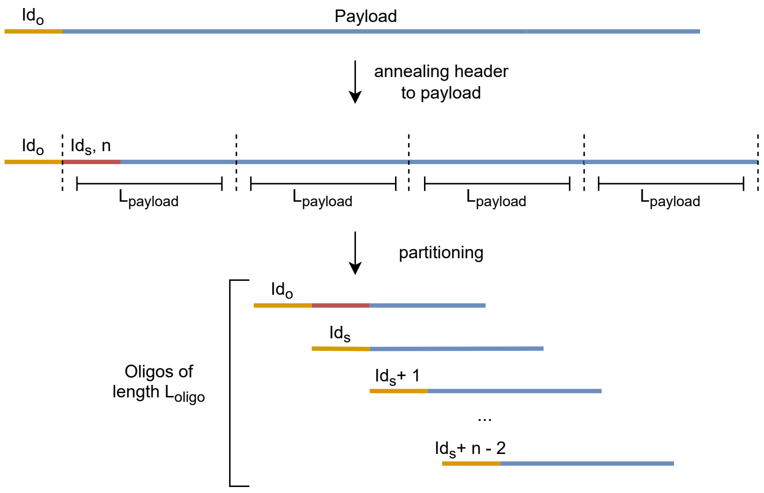


Fig. 4: Partitioning long oligos in DNAContainer.

Furthermore, if the given data object is too large, meaning that the payload is longer than $L_{payload}$, then the payload is partitioned among multiple oligos. This procedure is illustrated in Fig. 4 and is explained in the following. Let us assume that Id_o is the registered Id used to store the data object. First, we need to calculate the number of oligos n to which the long payload is split. Next, we register $n-1$ consecutive Ids, referred to as $Id_s + i$ for $i = 0, \ldots, n-2$. Then, we anneal Id_s and n to the left end of the long payload. We will refer to these two numbers as the payload's *header*, marked in red in Fig. 4. Note that we map the integers to DNA by representing them in base four and finally map each base four digit to a corresponding nucleotide. Therefore, the occupied space of the payload's header is always the same. After that, we split the payload (payload plus header) into partitions $p_i, i = 0, \ldots, n-1$ of the size $L_{payload}$. Finally, we need to address the obtained partitions. The first partition p_0 is addressed by Id_o, and each following partition p_i, $1 \leq i \leq n-1$ is addressed with $Id_s + i - 1$ as shown in Fig. 4. This method only works if the length of the payload plus its header are divisible by $L_{payload}$. In the other case, the last split partition p_{n-1} would be smaller than $L_{payload}$. To solve this issue, we add specific padding to p_{n-1}, resulting in all obtained oligos having the same length. Note that the function `put` implements the partitioning procedure and returns Id_o for the given data object.

To further optimize the oligos (after partitioning), we initially split a long payload into smaller payloads, each shorter than $L_{payload}$, and further add specific padding to each of them. This added padding is used to adjust the GC content and is further explained in the following section.

To decode the obtained oligos, i.e., to read the encoded data object, we read the first oligo given by Id_o. Next, we decode Id_s and n from the obtained payload to obtain the next virtual addresses $Id_s + i, i = 0, \ldots, n-2$. Finally, we read the corresponding oligos, assemble the payloads, and decode the data object.

4.3 Optimizing DNA Sequences

This section details the optimizations applied for DNA addresses and payloads. We implement two optimization steps. The first adds specific padding to a given DNA sequence, correcting its GC content closer to 50%. The second generates a number of permutations of the DNA sequence, selecting the one that adheres to all the constraints in Sect. 3.6.

4.3.1 Padding

If a DNA sequence is too short and the GC content is not 50%, we append additional nucleotides until the desired length is reached. For padding, we use 11 pre-computed DNA sequences $s_i, i = 0, \ldots, 10$ where s_i has a GC content of $\frac{i}{10}$. To add padding to a given DNA sequence a, we append nucleotides from the padding sequence s_i that best corrects the GC content of a to 50%. To mark the position at which the padding starts, we first append a specific delimiter sequence that is not contained as a subsequence in any s_i. Let us illustrate this with an example. Suppose $a = (\texttt{TCATT})$ with a GC content of 20%, and the target length of a is $t = 12$. Let the delimiter be $d = (\texttt{GT})$. The sequence a after appending the delimiter sequence is $a_d = (\texttt{TCATTGT})$ with a GC content of $\approx 28.5\%$. There are 5 remaining nucleotides to add from one of the pre-computed padding sequences. Since a_d has a GC content lower than 50%, we need to add padding information with a GC content higher than 50% to obtain a sequence with an overall GC content of nearly 50%. We evaluate the index i of the ideal padding sequence s_i as:

$$i = \left\lfloor 10 \cdot \max\left\{0, \frac{\frac{t}{2} - |q|_{\{\texttt{G},\texttt{C}\}}}{t - |q|}\right\}\right\rfloor \tag{1}$$

The expression $|q|$ returns the number of nucleotides in q, and $|q|_{\{\texttt{G},\texttt{C}\}}$ returns the number of nucleotides in q that are either a G or a C. Plugging in $t = 12$ and $q = a_d$ in Eq. (1), we obtain the padding sequence's index $i = 7$. Let us assume p_7 was pre-computed as $p_7 = (\texttt{TGCGGCTCCA})$. Hence, to reach $t = 12$, we append the first 5 nucleotides from p_7 to s_d resulting in (TCATTGTTGCGG) with a GC content of 50%.

The obtained DNA sequence after padding is likely to fulfill the constraint (1) in Sect. 3.6 but could still violate the remaining constraints. To adhere to all constraints, we further optimize the sequence after padding by permutations, which is explained in the following section.

4.3.2 Permutation

The DNA sequence obtained after padding could still contain homopolymers and have mutual overlaps with, e.g., other DNA addresses or payloads. To fulfill the remaining constraints (2), (3), and (4), we generate m permutations of the given DNA sequence, selecting the permutation that fulfills the constraints. We use the classical Fisher-Yates method [Du64] to compute a permutation. This method generates $k - 1$ index pairs to be swapped, where k is the length of the sequence. This method requires sampling random numbers from a random numbers generator (*RNG*) that requires a *seed* for initialization. Two RNG instances initialized with the same seed produce the same random numbers in the same order. To calculate the seed of a DNA sequence q, we evaluate:

$$seed(q) = |q|_{\{\texttt{A}\}} \cdot |q|_{\{\texttt{C}\}} \cdot |q|_{\{\texttt{T}\}} \cdot |q|_{\{\texttt{G}\}} \tag{2}$$

where $|q|_{\{b\}}$ counts the number of $b \in \{\texttt{A}, \texttt{C}, \texttt{T}, \texttt{G}\}$ in q. Note that $seed(q)$ is invariant of permutation, i.e., the seed of q and all its permutations is the same. Hence, we can reverse a permuted sequence to its original by knowing the seed. Finally, to compute the m permutations of q, we initialize m RNGs with $seed(q) + i, 0 \leq i < m$ that are used to permute q. Additionally, we append the offset i to each permuted sequence. Therefore, to reverse a permuted sequence, first, we decode and remove the encoded offset i from the DNA sequence to obtain the permuted DNA sequence $\hat{q}$ without offset. Next, we initialize an RNG with $seed(\hat{q}) + i$. Finally, using this initialized RNG, we swap the $k - 1$ generated index pairs in reverse, i.e., starting from the $(k - 1)$-th index pair to the first index pair.

After permutation, the obtained permuted sequence has its GC content corrected and is not likely to contain any homopolymers. Constraints (3) and (4) are further checked by approximating the Jaccard distance between two DNA sequences using LSH according to [El22]. The Jaccard distance is a metric between 0 and 1. A Jaccard distance of 0 means that the given two DNA sequences are as similar as possible, and a Jaccard distance of 1 is the maximum dissimilarity two DNA sequences can have. Hence, we select the permuted sequence that simultaneously contains no homopolymers and maximizes the Jaccard distance to the other DNA sequences.

This optimization is applied to both DNA addresses and payloads. If a DNA address after padding and permutation still does not fulfill all constraints (1), (2), (3), and (4), we route its corresponding Id to a new Id^R as detailed above in Sect. 4.1.2. If a payload does not fulfill the constraints, even after optimizations, we have the following options: (i) increase the number of permutations m, and (ii) incorporate the constraints into the payload's encoder and do not apply any optimizations to payloads. The first option to increase the number of permutations m is straightforward. More permutations increase the probability of finding a permuted DNA sequence that adheres to the constraints. The second option shifts the problem of generating payloads that adhere to certain constraints to the payload's encoder and is illustrated to work by utilizing fountain codes in [El22].

5 Implementing Data Structures on DNAContainer

Data structures provide useful abstractions and are necessary for efficient data access [Co22]. It is the basis for implementing efficient algorithms and even allows the integration of index structures directly on DNA. Hence, we implement three basic data structures: (i) Reference, (ii) Array, and (iii) List on DNAContainer, showcasing its usability.

5.1 Reference

This data structure (or data type) is implemented by the function `put`. In particular, by calling `put(`d`)`, the data object d is written to DNAContainer, returning a unique Id. We call this Id the *reference* to d. Therefore, essentially, every data object in DNAContainer is stored by a reference.

5.2 Array

Arrays are a well-known construct that current programming languages implement and data management algorithms rely on. We implement the array construct on DNAContainer, enabling concurrent access to its elements using only one Id. Let Id_o refer to an m-elements array. We further assume that every element of this array is encoded to DNA, e.g., by a fountain code. To write this array to DNAContainer, we generate m consecutive Ids by calling `register(`m`)`. Each of these Ids is used as a reference to an array's element. Let us refer to these Ids as $Id_a, Id_a + 1, \ldots Id_a + m - 1$, and to the i-th element of the array as e_i where the first element is e_0 and the last is e_{m-1}. As illustrated in Fig. 5, we utilize the `put` function with which we store each array's element calling `put(`$Id_a + i$`, `e_i`)`. Note that by calling `put`, the element could be partitioned to multiple oligos as explained in Sect. 4.2. Furthermore, Id_o is used to store Id_a and m as payload information.

To read an element of the array, we first call `get(`Id_o`)` to obtain the payload encoding Id_a and m. After that, we can access any index i of the array by calling $get(Id_a + i)$, $i = 0, \ldots, m - 1$, returning the element e_i of the array. Moreover, we can read the entire array in parallel by calling `get` on each index of the array simultaneously.

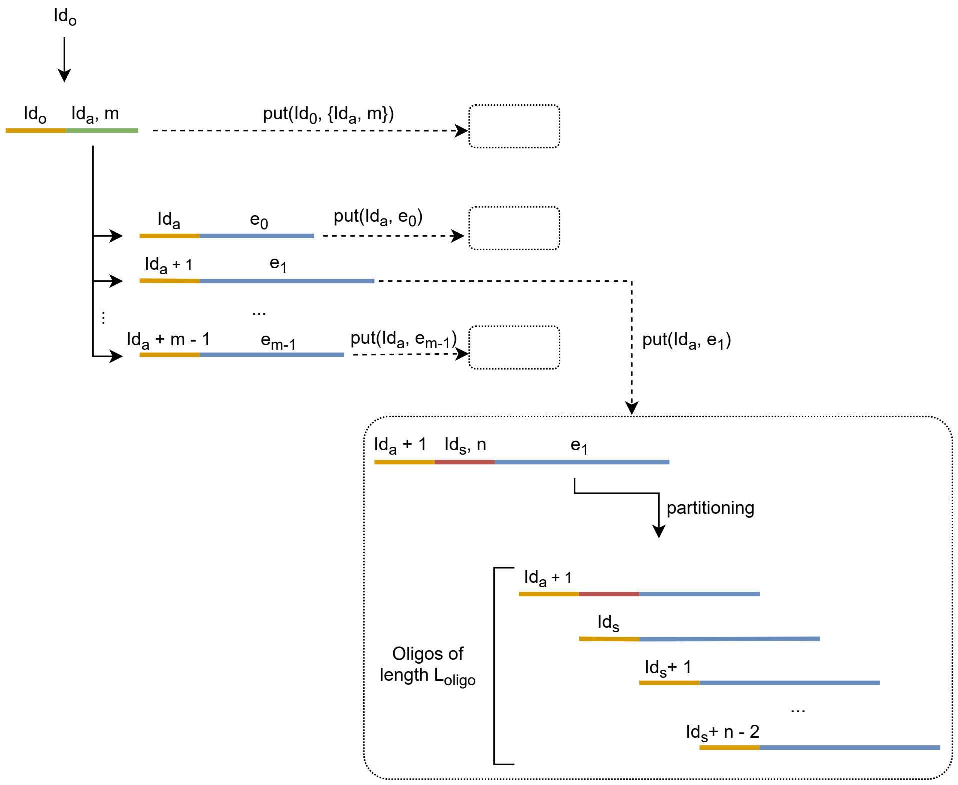


Fig. 5: Implementation of the array construct on DNAContainer.

5.3 List

Like an array, a list is a collection of elements where we assume the elements are mapped to DNA. However, unlike arrays, lists do not have a fixed size. DNAContainer implements a list as a chain of elements where each element points to the next one. Let us illustrate the implementation on DNAContainer by the example given in Fig. 6. Id_o is the Id used to reference the list of elements e_0 and e_1 marked as blue. The first element e_0 is stored in DNAContainer along with a newly registered Id_1 by calling `put(`Id_o`,` $\{Id_1, e_0\}$`)` where Id_1 is used to reference the next element of the list. Hence, element e_1 is stored in DNAContainer along with the next newly registered Id_2 and is referenced by Id_1 by `put(`Id_1`,` $\{Id_2, e_1\}$`)`. Since we invoke a `put` operation every time we append an element to the list, each element could be partitioned as detailed in Sect. 4.2. We repeat this procedure for every element appended to the list. For example, adding a third element e_2 to the list is done by storing e_2 along with a newly registered Id_3 and referenced by Id_2. This is a crucial difference to an array, where the array structure does not support adding elements after referencing the array.

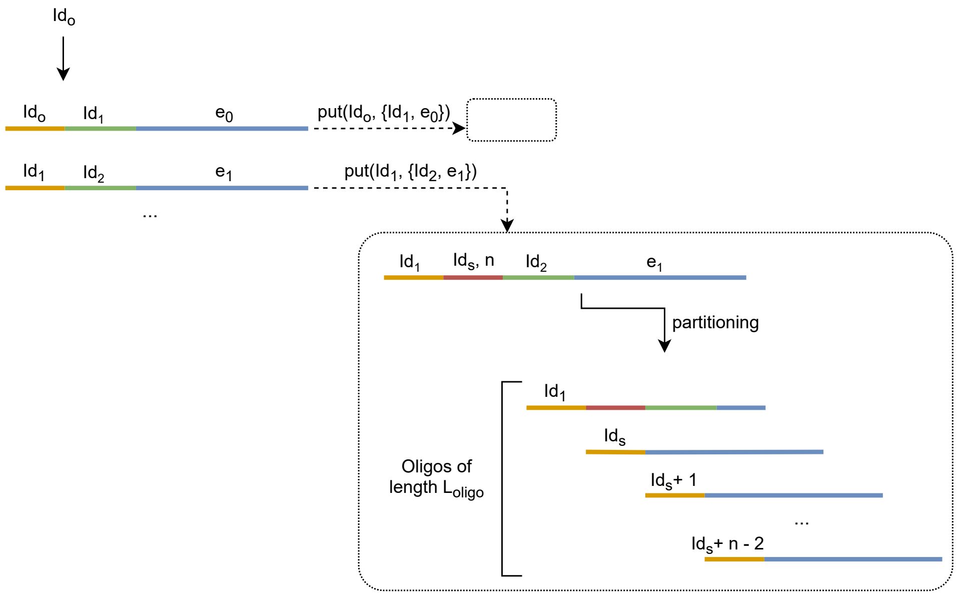


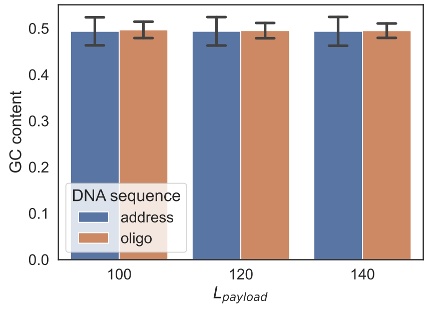
Fig. 6: Implementation of the list construct on DNAContainer.

To read an element from the list, we first call `get(`Id_o`)` to obtain the payload encoding Id_1 and e_0. After that, we could return the first element e_0. Otherwise, we iterate through the list by sequentially calling `get(`Id_i`)`, $i = 1, \ldots$ until we obtain the desired element.

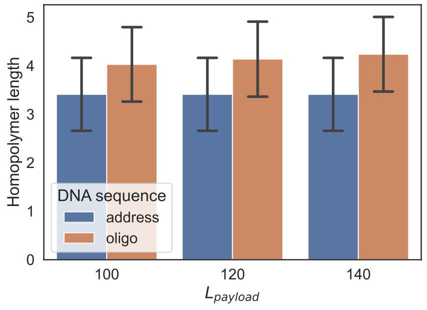
6 Experiments

We implemented DNAContainer in Java and tested it by simulating several million `put` and `get` operations. All the experiments were run on a computer with 256 logical cores (1.5 – 2.25 GHz each) and 1 TB of RAM. We used the data set from the OpenSky Network in (https://opensky-network.org/datasets/publication-data/climbing-aircraft-dataset/trajs/A321_valid.csv.xz), a relational table containing the tracking information of aircraft. We inserted the first 100, 000 lines (records) into DNAContainer (without error correcting codes), varying the address and payload sizes, resulting in millions of oligos.

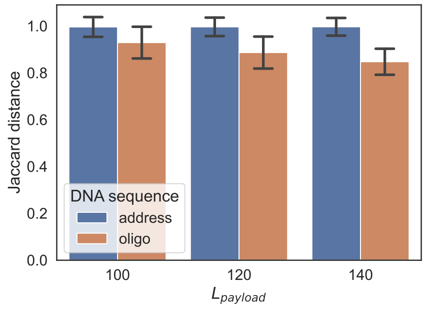
As shown in Fig. 7a, the obtained oligos' and DNA addresses' `GC` content is at ≈ 50%, adhering to the first constraint in Sect. 3.6. Furthermore, the average longest homopolymer's length is between 3 and 4, fulfilling the second required constraint as depicted in Fig. 7b. Moreover, to check the remaining constraints (3) and (4), we used LSH according to [El22] and set $k = 5$ for the Jaccard similarity. Hence, our DNA addresses and oligos do not significantly overlap and fulfill the remaining constraints as shown in Fig. 7c.



(a) The GC content.



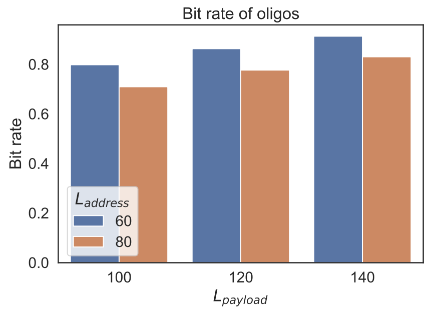
(b) The longest homopolymer length.



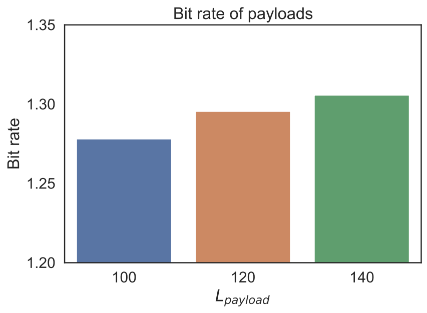
(c) The distance of every address and oligo to the other addresses and oligos calculated by LSH, respectively.

Fig. 7: GC content, the longest homopolymer length, and the mutual Jaccard distance of oligos and addresses.

The DNA optimization parameters are set as follows. For padding, we inserted ≈ 16% of the payload's size as padding information to each payload. Furthermore, we used $m = 16$ permutations for DNA addresses and payloads. After optimization, $L_{payload}$ ranged in 100, 120 and 140, and $L_{address}$ in 60, and 80. Note that by increasing the number of permutations, we could, e.g., reduce the mutual overlaps of the oligos. However, setting the permutations count to $m = 16$ was sufficient to obtain DNA without significant mutual overlaps. We repeated this experiment and turned off the permutations ($m = 0$), resulting in longer homopolymers, and the largest difference was that the mutual overlaps increased significantly. In particular, the average mutual Jaccard distance of the DNA addresses dropped from ≈ 1.0 to ≈ 0.5, and the average mutual Jaccard distance of the oligos dropped from ≈ 1.0 to ≈ 0.75.



(a) The bit rate of oligos.



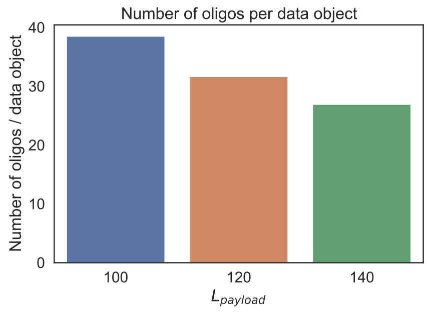
(b) The bit rate of payloads.

Fig. 8: The bit rate of oligos and payloads by varying the payload size.

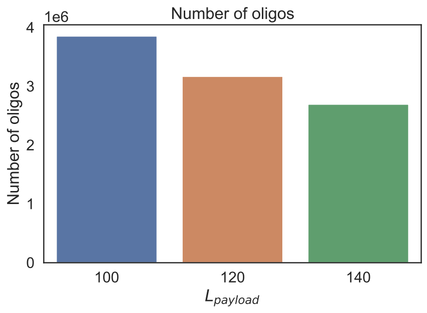
Figure 8a depicts the *bit rate* of oligos, i.e., the information density for each oligo. The bit rate is calculated as the total number of digital bits divided by the total number of nucleotides. As expected, the bit rate for oligos with an address size of 80 is lower than that of 60 because the address does not encode any information. Fig. 8b presents the bit rate of payloads instead of oligos, i.e., ignoring the DNA address. The bit rate also increases by increasing the payload's size because more information can fit into the same number of oligos as depicted in Fig. 9a and Fig. 9b. Thus, by increasing the payload's size, we obtain fewer oligos encoding the same information. A bit rate of 2.0 bits/nucleotide being optimal, our system encodes the information to payloads at around 1.3 bits/nucleotide, which is a $\approx$ 65% capacity utilization of payloads. Since a payload of length $L_{payload}$ optimally encodes two bits for every nucleotide and a single byte contains 8 bits, then the maximum capacity in bytes is calculated as:

$$\text{maximum capacity (bytes)} = 2 \cdot \frac{L_{payload}}{8} \tag{3}$$

For example, by setting $L_{payload} = 140$, the payload could encode up to 35 bytes. Plugging in our system's capacity utilization of 65% yields $\approx$ 23 bytes per payload. We repeated the same experiment above, stored the records in arrays and lists, and obtained similar results. We also tested turning on the Reed Solomon error correcting code, where the parameters are set such that up to 4 nucleotide erasures are corrected. The resulting capacity utilization was slightly lower at $\approx$ 59%. Despite that, DNAContainer manages to out-compete many recent DNA systems by supporting large-scale random access capabilities while maintaining a relatively high bit rate [Xu21, Do20, CNS19, Xu21].



(a) The number of oligos encoding a single data object.



(b) The number of oligos encoding all data objects.

Fig. 9: The number of oligos representing a single data object and the number of all oligos representing all the data objects.

Finally, to test the scalability of our approach, we translated 100 million addresses, i.e., we mapped 100 million Ids to corresponding DNA addresses (see Sect. 4.1) with $L_{address} = 80$, adhering to every constraint in Sect. 3.6. As shown above, our payloads with $L_{payload} = 140$ reach a bit rate of 1.3, i.e., carry 23 bytes of information. Therefore, we could store up to $23 \cdot 10^8$ bytes or 2.3 GB of information using these addresses, storing more information than the recently proposed DNA systems while providing sophisticated random access capabilities [Xu21]. Furthermore, only $\approx$ 4000 addresses were considered invalid of the generated 100 million addresses, and more addresses could be computed.

To store more information, e.g., in the terabyte range and beyond, in DNAContainer, more addresses must be generated, or a larger payload must be used. Current DNA synthesis technologies support synthesizing relatively short DNA sequences, whereas longer sequences are costly or not supported yet [HA19]. Additionally, we approximate the DNA overlaps using LSH, which requires extensive memory amounts. The computation time for generating the mentioned 100 million addresses took $\approx$ 32.5 hours with our computer, of which most of the time was spent on synchronizing checking constraint (3) for each generated permutation in parallel. Hence, the computation times could be significantly higher with a computer equipped with less memory or fewer processing cores.

Nevertheless, sequencing and especially synthesis technologies are constantly evolving. Certain synthesis technologies are developed, allowing the synthesis of several thousand nucleotides [Pi19]. For example, if we choose $L_{payload} = 5,000$ and use 10^{14} addresses, then the theoretical storage capacity of DNAContainer is $\approx$ 11.5 EB, which could be extended further by using more addresses or larger payloads.

7 Conclusion

This paper presents DNAContainer, an interface for DNA storage similar to a traditional storage device. DNAContainer offers an abstraction layer by providing simple put and get operations instead of synthesizing and sequencing DNA. Furthermore, we implement the common data structures array and list on DNAContainer, making data management more accessible. DNAContainer uses a virtual address space mapped to physical DNA addresses, facilitating random access to the data objects using traditional methods. Moreover, DNAContainer is aware of the required biochemical constraints. In particular, it encodes data objects as DNA oligonucleotides that are stable for long archival times and enables randomly accessing the data with the used virtual addresses. We tested our approach by simulating the insertion of several thousand data objects into DNAContainer, proving its scalability of managing up to millions of oligonucleotides addressed by millions of addresses.

In our future work, we will study multiple extensions of DNAContainer. In particular, we are interested in supporting more advanced index structures supporting expressive filter queries like range queries on DNA. Furthermore, we are designing a prototype of a physical DNA storage system where DNAContainer will be the primary interface.

Acknowledgment

We thank the Hessian Ministry for Science and the Arts (LOEWE) for funding this work.

Bibliography

[AJ20] Appuswamy, Raja; Joguin, Vincent: Universal layout emulation for long-term database archival. arXiv preprint arXiv:2009.02678, 2020.

[Al12] Allentoft, Morten E; Collins, Matthew; Harker, David; Haile, James; Oskam, Charlotte L; Hale, Marie L; Campos, Paula F; Samaniego, Jose A; Gilbert, M Thomas P; Willerslev, Eske et al.: 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, 2012.

[Ap19] Appuswamy, Raja; Lebrigand, Kevin; Barbry, Pascal; Antonini, Marc; Madderson, Oliver; Freemont, Paul; MacDonald, James; Heinis, Thomas: OligoArchive: Using DNA in the DBMS storage hierarchy. In: Biennal Conference on Innovative Data Systems Research (CIDR 2019). p. p98, 2019.

[Ba20] Banal, James L; Shepherd, Tyson R; Berleant, Joseph D; Huang, Hellen; Reyes, Miguel; Ackerman, Cheri M; Blainey, Paul; Bathe, Mark: Random access DNA memory in a scalable, archival file storage system. bioRxiv, 2020.

[Be15] Berlin, Konstantin; Koren, Sergey; Chin, Chen-Shan; Drake, James P; Landolin, Jane M; Phillippy, Adam M: Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. Nature biotechnology, 33(6):623–630, 2015.

[Bo16] Bornholt, James; Lopez, Randolph; Carmean, Douglas M; Ceze, Luis; Seelig, Georg; Strauss, Karin: 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, 2016.

[Br97] Broder, Andrei Z: On the resemblance and containment of documents. In: Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171). IEEE, pp. 21–29, 1997.

[Bu01] Buhler, Jeremy: Efficient large-scale sequence comparison by locality-sensitive hashing. Bioinformatics, 17(5):419–428, 2001.

[Bu13] Bumgarner, Roger: Overview of DNA microarrays: types, applications, and their future. Current protocols in molecular biology, 101(1):22–1, 2013.

[Ch99] Chalikian, Tigran V; Völker, Jens; Plum, G Eric; Breslauer, Kenneth J: A more unified picture for the thermodynamics of nucleic acid duplex melting: a characterization by calorimetric and volumetric techniques. Proceedings of the National Academy of Sciences, 96(14):7853–7858, 1999.

[CNS19] Ceze, Luis; Nivala, Jeff; Strauss, Karin: Molecular digital data storage using DNA. Nature Reviews Genetics, 20(8):456–466, 2019.

[Co22] Cormen, Thomas H; Leiserson, Charles E; Rivest, Ronald L; Stein, Clifford: Introduction to algorithms. MIT press, 2022.

[De90] Deux, O et al.: The story of O2. IEEE Transactions on Knowledge & Data Engineering, 2(01):91–108, 1990.

[Do20] Dong, Yiming; Sun, Fajia; Ping, Zhi; Ouyang, Qi; Qian, Long: DNA storage: research landscape and future prospects. National Science Review, 7(6):1092–1107, 2020.

[Du64] Durstenfeld, Richard: Algorithm 235: random permutation. Communications of the ACM, 7(7):420, 1964.

[El22] El-Shaikh, Alex; Welzel, Marius; Heider, Dominik; Seeger, Bernhard: High-scale random access on DNA storage systems. NAR genomics and bioinformatics, 4(1):lqab126, 2022.

[EZ17] Erlich, Yaniv; Zielinski, Dina: DNA Fountain enables a robust and efficient storage architecture. science, 355(6328):950–954, 2017.

[GMM16] Goodwin, Sara; McPherson, John D; McCombie, W Richard: Coming of age: ten years of next-generation sequencing technologies. Nature Reviews Genetics, 17(6):333–351, 2016.

[Go13] Goldman, Nick; Bertone, Paul; Chen, Siyuan; Dessimoz, Christophe; LeProust, Emily M; Sipos, Botond; Birney, Ewan: Towards practical, high-capacity, low-maintenance information storage in synthesized DNA. Nature, 494(7435):77–80, 2013.

[HA19] Heinis, Thomas; Alnasir, Jamie J: Survey of information encoding techniques for dna. arXiv preprint arXiv:1906.11062, 2019.

[HMG19] Heckel, Reinhard; Mikutis, Gediminas; Grass, Robert N: A characterization of the DNA data storage channel. Scientific reports, 9(1):1–12, 2019.

[IM98] Indyk, Piotr; Motwani, Rajeev: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the thirtieth annual ACM symposium on Theory of computing. pp. 604–613, 1998.

[KC14] Kosuri, Sriram; Church, George M: Large-scale de novo DNA synthesis: technologies and applications. Nature methods, 11(5):499–507, 2014.

[Ku01] Kurella, Manjula; Hsiao, Li-Li; Yoshida, Takumi; Randall, Jeffrey D; Chow, Gary; Sarang, Satinder S; Jensen, Roderick V; Gullans, Steven R: DNA microarray analysis of complex biologic processes. Journal of the American Society of Nephrology, 12(5):1072–1078, 2001.

[Li20a] Li, Bingzhe; Song, Nae Young; Ou, Li; Du, David HC: 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). 2020.

[Li20b] Lin, Kevin N; Volkel, Kevin; Tuck, James M; Keung, Albert J: Dynamic and scalable DNA-based information storage. Nature communications, 11(1):1–12, 2020.

[Li22] Lin, Yi-Syuan; Liang, Yu-Pei; Chen, Tseng-Yi; Chang, Yuan-Hao; Chen, Shuo-Han; Wei, Hsin-Wen; Shih, Wei-Kuan: How to Enable Index Scheme for Reducing the Writing Cost of DNA Storage on Insertion and Deletion. ACM Transactions on Embedded Computing Systems (TECS), 21(3):1–25, 2022.

[Ma20] Ma, Tian J; Garcia, Rudy J; Danford, Forest; Patrizi, Laura; Galasso, Jennifer; Loyd, Jason: Big data actionable intelligence architecture. Journal of Big Data, 7(1):1–19, 2020.

[MFL14] Ma, Dongzhe; Feng, Jianhua; Li, Guoliang: A survey of address translation technologies for flash memories. ACM Computing Surveys (CSUR), 46(3):1–39, 2014.

[Or18] Organick, Lee; Ang, Siena Dumas; Chen, Yuan-Jyue; Lopez, Randolph; Yekhanin, Sergey; Makarychev, Konstantin; Racz, Miklos Z; Kamath, Govinda; Gopalan, Parikshit; Nguyen, Bichlien et al.: Random access in large-scale DNA data storage. Nature biotechnology, 36(3):242–248, 2018.

[Pi19] Ping, Zhi; Ma, Dongzhao; Huang, Xiaoluo; Chen, Shihong; Liu, Longying; Guo, Fei; Zhu, Sha Joe; Shen, Yue: Carbon-based archiving: current progress and future prospects of DNA-based data storage. GigaScience, 8(6):giz075, 2019.

[QSH22] Quah, Jasmine; Sella, Omer; Heinis, Thomas: DNA data storage, sequencing data-carrying DNA. arXiv preprint arXiv:2205.05488, 2022.

[RS60] Reed, Irving S; Solomon, Gustave: Polynomial codes over certain finite fields. Journal of the society for industrial and applied mathematics, 8(2):300–304, 1960.

[Sc20] Schwarz, Michael; Welzel, Marius; Kabdullayeva, Tolganay; Becker, Anke; Freisleben, Bernd; Heider, Dominik: MESA: automated assessment of synthetic DNA fragments and simulation of DNA synthesis, storage, sequencing and PCR errors. Bioinformatics, 36(11):3322–3326, 2020.

[Sp59] Spencer, M: The stereochemistry of deoxyribonucleic acid. I. Covalent bond lengths and angles. Acta Crystallographica, 12(1):59–65, 1959.

[Wa19] Wang, Yixin; Zhang, Jingyun; Gunawan, Erry; Guan, Yong Liang; Poh, Chueh Loo et al.: High capacity DNA data storage with variable-length Oligonucleotides using repeat accumulate code and hybrid mapping. Journal of biological engineering, 13(1):1–11, 2019.

[Xu21] Xu, Chengtao; Zhao, Chao; Ma, Biao; Liu, Hong: Uncertainties in synthetic DNA-based data storage. Nucleic acids research, 49(10):5451–5469, 2021.

[YPFK06] Yakovchuk, Peter; Protozanova, Ekaterina; Frank-Kamenetskii, Maxim D: Base-stacking and base-pairing contributions into thermal stability of the DNA double helix. Nucleic acids research, 34(2):564–574, 2006.

[Zh16] Zhirnov, Victor; Zadegan, Reza M; Sandhu, Gurtej S; Church, George M; Hughes, William L: Nucleic acid memory. Nature materials, 15(4):366–370, 2016.