GenoDedup: Similarity-Based Deduplication and Delta-Encoding for Genome Sequencing Data

Vinicius Cogo, João Paulo, and Alysson Bessani

Abstract—The vast datasets produced in human genomics must be efficiently stored, transferred, and processed while prioritizing *storage space* and *restore performance*. Balancing these two properties becomes challenging when resorting to traditional data compression techniques. In fact, specialized algorithms for compressing sequencing data favor the former, while large genome repositories widely resort to generic compressors (e.g., GZIP) to benefit from the latter. Notably, human beings have approximately 99.9% of DNA sequence similarity, vouching for an excellent opportunity for deduplication and its assets: leveraging inter-file similarity and achieving higher read performance. However, identity-based deduplication fails to provide a satisfactory reduction in the storage requirements of genomes. In this work, we balance space savings and restore performance by proposing GenoDedup, the first method that integrates efficient similarity-based deduplication and specialized delta-encoding for genome sequencing data. Our solution currently achieves 67.8% of the reduction gains of SPRING (i.e., the best specialized tool in this metric) and restores data 1.62× faster than SeqDB (i.e., the fastest competitor). Additionally, GenoDedup restores data 9.96× faster than SPRING and compresses files 2.05× more than SeqDB.

Index Terms—Storage, Deduplication, Compression, Genome Sequencing Data

1 Introduction

Personalized medicine brings medical decisions to the individual level propelling the use of specific procedures and treatments for each patient. Human genomics enables advances in this and many other critical applications that are increasing our health awareness and life expectancy [1]. Datasets produced in this subject are huge since its studies compare thousands to millions of biological samples, where hundreds of gigabytes of data are generated from each sequenced body cell [2].

This data deluge must be efficiently stored, transferred, and processed to avoid stagnating medical breakthroughs [3]. Cutting costs in storage space and achieving a high-throughput in restoring data are paramount for this domain. Our primary goal is to *increase data reduction gains and restore it faster than the generic compressors* used in practice (e.g., GZIP), while approaching the reduction gains to the ones from specialized tools.

Genomic data has three main representations, as shown in Figure 1. *Sequencing data* is the immediate output from genome sequencing machines [4] and is typically stored in the FASTQ format [5]. It contains millions of randomly-dispersed small DNA sequences with associated quality scores (QS) to attest the sequencing accuracy. *Aligned data* results from ordering the FASTQ entries based on a reference genome, and is stored in the SAM/BAM format [6]. *Assembled data* results from merging the aligned overlapping entries into contiguous DNA sequences, which are commonly stored in the FASTA format.

Humans have 99.9% of DNA sequence similarity since the *assembled genome* of any two individuals differ in less than 0.1% [7]. Additionally, this representation has a public blueprint (i.e., a reference genome) for humans[1]. It sizes ∼3GB of data from its 3.2 billion contiguous sequence of nucleobases. Assembled human genomes can be reduced ∼700× from ∼3GB to ∼4.2MB in 40 seconds [8] by storing only the genome differences to the mentioned blueprint in a process called referential compression. However, *sequencing data* is much bigger than *assembled data* and has particularities that prevent such compression ratio.

Sequencing data is the most critical representation in genomics because it contains the purest version of genomic data and is unbiased from subsequent processing steps [5]. On the contrary, the output from alignment and assembly is imprecise, lossy, and algorithm-dependent [9]. For instance, using *aligned data* from multiple sources means they presumably were aligned with different algorithms and reference genomes. It precludes subsequent analyses, except if one first converts data back to *sequencing data* and realigns it with the same algorithm and reference (see §2).

The main reasons *sequencing data* is harder to compress than *assembled data* are (i) the randomness on entries' locality (small data chunks sequenced in no specific order [10]); and (ii) the lack of a stable reference for quality scores [3] (e.g., a similar blueprint as the *hg38* available for human DNA). Corroborating these observations, specialized algorithms usually compress *sequencing data* no more than 7× (see §3 for details on FASTQ compression).

Many algorithms favor maximizing compression ratio, which usually comes with penalties in (de)compression speed. This decision is justifiable when data is intended to be archived. However, the decompression speed becomes a bottleneck in cases where compressed data is read from remote storage systems and needs to be decompressed and read several times. In fact, this issue justifies why many real-world solutions (e.g., 1000 Genomes Project [11]) prefer generic compression algorithms that decompress fast (e.g., GZIP) rather than those that only compress more.

- *VC and AB are with LASIGE, Faculdade de Ciências, Universidade de Lisboa, Lisboa, Portugal. JP is with HASLab - High-Assurance Software Lab, INESC TEC & U. Minho, Portugal. Authors e-mails: vielmo@lasige.di.fc.ul.pt, jtpaulo@inesctec.pt, and anbessani@ciencias.ulisboa.pt.*
- *This work was supported by the European Commission, through SUPERCLOUD project (H2020-ICT-643964), and by FCT, through projects IRCoC (PTDC/EEISCR/6970/2014) and LASIGE Research Unit (UIDB/00408/2020).*

1. *hg38*, http://genomereference.org/

Storage of sequencing data is an important, challenging mostly unexplored domain for the systems community [3]. It presents an excellent opportunity for deduplication and its assets: leveraging inter-file similarity and achieving high-performance in reading data. However, traditional identity-based deduplication fails to provide a satisfactory reduction in the storage requirements of genomes (see §4.1).

Solutions for similarity-based deduplication commonly cluster similar entries into buckets and use identity-based deduplication within them [12], or they focus mostly on the delta-encoding problem [13] while employing inefficient global indexes [14]. In this work, we balance space savings and restore performance by proposing GenoDedup, the first method that integrates scalable, efficient similarity-based deduplication and specialized delta-encoding for genome sequencing data.

Novelty in our approach encompasses (i) the proposal (§4.2) and implementation (§5.3.2) of GenoDedup, a similarity-based deduplication solution that integrates scalable, efficient Locality-Sensitive Hashing (LSH) with delta-encoding; and (ii) specializations on delta-encoding for genome sequencing data, namely:

- Circular deltas (§2);
- Delta-Hamming (§5.3.1);
- A scalable modeling of generic indexes for multiple genomes (§5.2).

Additionally, we introduce a converged characterization of aspects from sequencing data important to deduplication (§2) and justify why identity-based deduplication fails on it (§4.1). Our experimental results (§6) attest the feasibility of GenoDedup since it currently achieves 67.8% of the reduction gains of SPRING [15] (i.e., the best specialized tool in this metric) and restores data 1.62× faster than SeqDB [16] (i.e., the fastest competitor). Additionally, GenoDedup restores data 9.96× faster than SPRING and compresses files 2.05× more than SeqDB.

2 Genome Sequencing Files

Data obtained from sequencing genomes is stored in the FASTQ text format [5], which is usually written once and read many times later for processing. FASTQ is the standard format in both cold and hot storage systems for genomic sequencing data [5]. A discussion on other datasets and on why this work favors sequencing data rather than aligned or assembled representations is available in §7.

A FASTQ file contains many entries with four lines each—similar to the one presented at the top right corner of Figure 1. The first line is a *comment* about the entry starting with a "@" character. The second line contains the *DNA sequence* interpreted by the machine—e.g., A for adenine, C for cytosine, G for guanine, and T for thymine. The third line is another comment that starts with a "+" character to determine the end of the DNA sequence, and can optionally be followed by the same content as the first one. The fourth line contains *quality scores (QS)*, which measure the machine's confidence for each sequenced nucleobase.

The second (DNA) and fourth (QS) lines have the same length since one QS is attributed for each sequenced nucleobase. This length is configurable and may vary from file to file, but it is usually constant within the same file. In the following descriptions, we detail each portion of FASTQ entries.

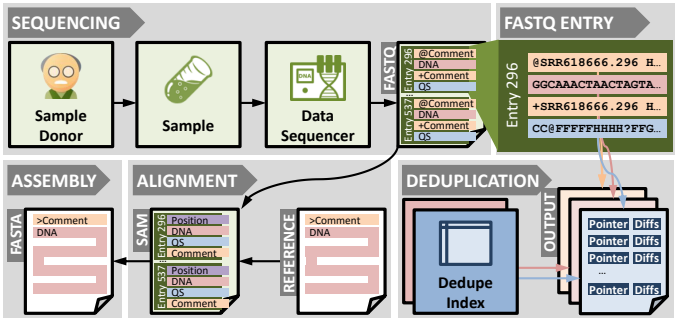

Figure 1. Genome sequencing overview, some subsequent workflows, and a FASTQ entry.

Comment Lines

The first and third lines of each FASTQ entry are comments that start with a "@" character in the former and a "+" in the latter. These lines usually contain: a sample identifier (e.g., SRR618666 in Figure 1), the entry identifier (e.g., 296), and some information about the sequencing run (e.g., HWI-ST483:151:C08KDACXX:7:1101:21215:2070/1). Comments follow a similar structure through the file, which can be determined if it contains numeric or alphanumeric fields, and if they are constant, incremental, or variable among entries [17].

DNA

The second line of each entry contains the DNA sequence interpreted by the sequencing machine. This sequence is composed of ℓ characters, where this length ℓ can be configured on each sequencing job. Nucleobases can be represented using different sets of characters, where the most commonly used is the $\{A,C,G,T,N\}$. It considers the four nucleobases (i.e., adenine, cytosine, guanine, and thymine) and a special character "N" to represent any of them when the machine is unsure on the sequenced nucleobase.

A contiguous human genome sizes 3.2 billion nucleobases and results in more than 3GB of data in text mode (e.g., UTF-8 encodes each character in 1 byte). However, NGS machines do not provide the whole genome in a single contiguous DNA sequence [10]. They generate millions of randomly-dispersed reads, which contain small pieces of DNA sequences with hundreds to thousands of nucleobases each [5].

A configurable sequencing parameter determines the coverage in which a genome is sequenced. It is equivalent to the average number of different entries in which every nucleobase position from a genome appears in. Common configurations consider coverage of 30–45× to increase accuracy. This redundancy results, for instance, in 96 to 144GB of DNA characters per whole sequenced human genome in the FASTQ format.

Quality Scores (QS)

The fourth line of each FASTQ entry contains the sequence of quality scores asserting the confidence level for each sequenced nucleobase. *Phred* quality score [18] is the typical notation in FASTQ files. QS values usually range from 0 to 93 (the higher, the better) and are encoded in ASCII (requiring seven bits per QS) [5]. QS roughly occupy the same storage space as DNA in FASTQ since there is one QS for each nucleobase, and standard text encoding (e.g., UTF-8) use eight bits per character.

Quality score sequences are the most challenging portion of FASTQ entries to compress, and as such, we concentrate

most of our efforts on it. There is no reference sequence for quality scores [3], but they do have patterns that can boost data reduction [19]. In this paper, we take into consideration three of them. The first pattern is that many NGS machines have a limited precision and generate QS only in the range between 0 and 40 [5], [15], which allows one to describe them using six bits instead of seven. Second, the longer the read DNA sequence is, the bigger the uncertainty at the end of the QS sequence. For instance, a practical implication from this pattern is that, in FASTQ files from Illumina HiSeq 2000[2] (the most common NGS machine in the world [20]), several QS sequences finish with a chain of "#" characters—i.e., a low *Phred* value equivalent to 0.

The third pattern is the fact that subsequent QS tend to vary little from one to the other [21]. It means that one may replace subsequent QS by a delta value, which results in the zero value most of the times [19], and convert data to a normal distribution between -40 and $+40$.

However, using delta values naively increases the number of bits required to describe a QS to seven bits again since there are eighty-one options between -40 and $+40$. With this in mind, we propose to use modular arithmetic to convert them to *circular deltas*, which distributes the mentioned range into a circular array from -20 to $+20$. Each circular delta can be translated into two different normal delta values. For instance, the circular delta -1 is equivalent to both -1 and $+40$ normal deltas. When solving circular deltas to restore the original QS sequence, the correct alternative can unambiguously be distinguished because only it results in a valid QS between 0 and 40. This transformation reduces the QS encoding back to six bits.

3 Sequencing Data Compression

Before presenting the challenges of deduplicating genomic sequencing data, we discuss the state-of-the-art on the compression of sequencing data, its limitations, and the opportunities it leaves open for deduplication. There is a well-known trade-off in data compression between compression ratio and throughput [22]. We selected ten relevant compression algorithms that achieve the best results in these properties [23], [24]: GZIP,[3] pigz,[4] BSC,[5] ZPAQ,[6] SeqDB [16], DSRC2 [25], Quip [26], FQZcomp [23], FaStore [27], and SPRING [15].

Our analyses use five representative FASTQ files of human genomes from the 1000 Genomes Project [11]: SRR400039, SRR618664, SRR618666, SRR618669, and SRR622458. Only the FASTQ file from the first end of these genomes are considered in our analyses, but they sum up 265GB of data and result in almost one billion FASTQ entries. Table 1 presents these files and the resulting compression ratio and restore throughput of each algorithm on them. More details on these files (e.g., number of entries, sequence lengths, and coverage) can be seen in §2 of our Supplementary Material.

GZIP is a generic compression tool employed in several application domains, including the storage of human genome sequencing data. For instance, the 1000 Genomes Project [11] stores their FASTQ files compressed with GZIP. Even recent frameworks for bioinformatics (e.g., Persona [28]) use GZIP to compress data. The main strength of GZIP is its decompression/restore throughput, which reaches 41MB/s on average in our files and 66MB/s in its parallel version (i.e., pigz), while ZPAQ, Quip, and Fqzcomp reach less than 10MB/s and SPRING reaches 20MB/s. FaStore and BSC reach a similar throughput as GZIP, but DSRC2 and SeqDB are the fastest (specialized) tools to decompress FASTQ files, reaching a throughput of approximately 125MB/s. We use GZIP and pigz as the baseline generic tools and SeqDB and DSRC2 as the baseline specialized tools in experiments that evaluate throughput.

Many specialized tools for FASTQ files focus on maximizing compression ratio. For instance, SPRING is the specialized tool that reaches the best compression ratio in our files (i.e., 6.023× on average). It is followed up by FaStore (i.e., 5.4×) and by the generic tool ZPAQ (i.e., 5.2×). We use ZPAQ as the baseline generic tool (together with GZIP and pigz due to their importance and restore throughput) and SPRING as the baseline specialized tool in experiments that evaluate FASTQ compression ratio.

We have evaluated other specialized (e.g., G-SQZ [29] and KIC [30]) and generic compression algorithms (e.g., BZIP2[7] and LZMA2[8]). However, they compress data less than SPRING [15] and restore data slower than pigz and SeqDB [16] in our experiments. Additionally, we have evaluated LFQC [31] and discarded its results because it uses LPAQ8 to compress the quality score sequences and LPAQ8 does not support files bigger than 2GB. The complete discussion on these alternative tools is available in §3 of our Supplementary Material.

Algorithms that align the DNA data before compressing it (e.g., SlimGene [19]) can reduce the DNA portion alone up to 20×, but they take considerable time (e.g., 8 hours per human genome) and consequently reduce the compression throughput. Nonetheless, our methods can work with aligned data (see § 7).

Finally, Zhou *et al.* [32] propose a similarity-based compression algorithm for quality scores from genome sequencing data. However, they use a non-scalable memetic algorithm to create a small codebook for each FASTQ file they want to compress and they inefficiently compare each QS sequence to all base chunks in the codebook to calculate the best delta-encoding. Additionally, we cannot compare the performance of our solution to theirs because they provide no implementation, but our work surpasses theirs in several other aspects, which are detailed in §5.

4 Human Genome Deduplication

Deduplication reduces the storage requirements by eliminating unrelated redundant data [33]. Additionally, deduplication has two advantages when compared to compression algorithms: it may leverage the inter-file similarities, while most compression algorithms consider only intra-file data or use a single generic contiguous reference; and it usually achieves a better restore performance than compression.

There are many deduplication approaches and systems available [12], and several of them rely on *index data structures* to lookup exact copies of data already stored in the system. This indexing mechanism maps the content of stored chunks to their actual storage location to efficiently find duplicate instances.

2. https://www.illumina.com/documents/products/datasheets/datasheet_hiseq2000.pdf
3. https://www.gzip.org/
4. https://zlib.net/pigz/
5. http://libbsc.com/
6. http://mattmahoney.net/dc/zpaq_compression.pdf
7. https://github.com/enthought/bzip2-1.0.6
8. https://www.7-zip.org/

Table 1
Genomes and compression tools. Per genome: its identifier and size in GB. Per algorithm: compression ratio (i.e., *original_size*/*compressed_size*) on each genome, write and read throughput (in MB/s), its version, and where it was published. ⊗ Generic compression algorithm. † We used only portions of this file to complete 100GB of DNA and of QS lines in our experiments. ⋆ See §6 for the complete analysis.

Genome (Size in GB)	GZIP⊗	pigz⊗	BSC⊗	ZPAQ⊗	SeqDB	DSRC2	Quip	Fqzcomp	FaStore	SPRING	GenoDedup
SRR400039_1 (34.3GB)	2.800	2.801	3.994	4.426	2.015	3.878	4.550	4.523	4.695	**5.179**	4.110
SRR618664_1 (64.6GB)	3.006	3.004	4.328	4.839	2.007	4.240	4.982	4.935	N/A	**6.038**	4.419
SRR618666_1 (62.3GB)	2.927	2.930	4.198	4.688	2.003	4.120	4.825	4.776	N/A	**5.841**	4.354
SRR618669_1 (79.6GB)	3.027	3.027	4.362	4.886	2.012	4.287	5.029	4.968	N/A	**6.187**	4.517
SRR622458_1† (23.6GB)	4.367	4.373	5.830	7.367	1.924	4.212	4.811	5.018	6.173	**6.869**	3.047
Avg. Comp. Ratio	3.225	3.227	4.543	5.241	1,992	4.148	4.839	4.844	5.434	**6.023**	4.089
Write (MB/s)	15.5	281.1	159.9	5.3	415.6	**1375.9**	28.7	60.5	25.5	43.1	0.3⋆
Read (MB/s)	41.4	66.1	46.2	1.1	127.9	125.3	3.4	9.6	45.2	20.9	**208.2**⋆
Version	1.6	3.1.0	7.15	2.00	0.2.1	1.1.8	4.6	1.0	0.8.0	9.22	0.1

4.1 Identity-based Deduplication

In this section, we discuss the strengths and limitations of common approaches for identity-based deduplication and present examples confronting them with FASTQ files. Given the particularities of FASTQ files (§2), this discussion is of extreme importance to clarify and caution the general deduplication community in the search for efficient solutions to the problem of interest. The next discussions encompass three approaches: file deduplication, block deduplication, and application-aware deduplication.

File deduplication

This approach identifies exact copies of the same file by comparing their content hashes (e.g., SHA-2) and replaces the redundant data with pointers to a single instance. It is ineffective in genome repositories because these facilities store data mostly from their unique samples [34] or because even sequencing the same sample results in files with different content [10].

Example. The 1000 Genomes Project [11] contains half a million files, in which more than 200k are FASTQ. We downloaded its current directory tree[9] and compared the content hashes (MD5) of all FASTQ files to obtain the duplicate ratio. These MD5 hashes are available in the last column of this directory tree, which means one does not need to download all FASTQ files to perform the present comparison. The result indicates that less than 0.007% of the FASTQ collection is composed of duplicate files, which validates the low interest for file deduplication in sequencing data.

Block deduplication

This approach splits files into fixed- or variable-size blocks, calculates their content hashes, and compares them to find duplicates. Systems with fixed-size block deduplication commonly adopt blocks of 4KiB for historical and compatibility reasons—e.g., this is the size of virtual memory pages in several computer architectures and of blocks in many filesystems. For variable-length blocks, the most common algorithms are the Rabin fingerprinting and the Two-Threshold Two-Divisor (TTTD).

Block deduplication fails to identify copies of FASTQ data chunks because they are unlikely to happen. Reasons for that include the fact FASTQ files contain the unique sample and entry identifiers; the DNA sequences contain mutations, transformations, and are sequenced in no specific order; and the distribution of QS varies from run to run.

Example. We have split three FASTQ files (SRR400039, SRR618664, and SRR618666) into 40 million fixed-size blocks of 4KiB, calculated the MD5 hash of each block, and verified that there are no duplicates on it. We executed the same experiment with variable-size chunks using the Rabin fingerprinting[10] (with blocks between 1–8KiB) to generate more than 23 million hashes, where no duplicates were found.

Application-Aware deduplication

A final strategy is to take into consideration the files' structure and content to increase the chances of deduplication. One may write each line type of FASTQ entries into different files—each one containing only (1) the "@" sequencing comments, (2) the DNA sequences, (3) the "+" comments, or (4) the quality scores—and deduplicate them separately. Both fixed- and variable-size block deduplication can be employed in this approach.

Example (Comment lines). Comments have an identifiable structure that can be parsed into fields—e.g., lines from the SRR618666 genome have ten fields each. Five of them are constant across the whole file, two are incremental numbers, and three are variable. One may replace the constant and incremental fields by a small encoding at the beginning of a compressed file. Then, the remaining variable fields can be placed in a file to be deduplicated separately. In SRR618666, the 231 million lines, with three variable fields each, can be replaced by pointers to only 48 unique values in the first field, 20k in the second, and 199k in the third. Bhola *et al.* [17] compresses comments 17× with this approach.

Example (DNA and QS blocks). We separate the lines from the three FASTQ files as previously mentioned, removed the newline character and performed the block deduplication previously presented. We split the DNA and the QS files into 4KiB blocks and separately compared their content hashes, which results in no duplicates. Similarly, executing the same workflow with Rabin fingerprinting does not find any redundant blocks.

9. http://ftp.1000genomes.ebi.ac.uk/vol1/ftp/current.tree

10. https://github.com/datproject/rabin

We execute the same block deduplication with the block size as 100 characters in SRR618664 and SRR618666 (i.e., the sequence length in these files). This approach is the first to provide a considerable number of duplicates. From the 471 million entries in these genomes, 44 million DNA lines (9.42%) are exact duplicates, as well as 468 thousand QS lines (0.01%). However, these values are unsatisfactory since spatial deduplication requires gains of 20–40% to be worth the invested cost and time [33].

Summary

The three selected FASTQ files used in these examples are enough to illustrate the inefficiency of traditional identity-based deduplication methods, whereas considering more genomes here leads to similar conclusions. Identity-based deduplication provided significant gains only in comment lines in our analyses. Based on the descriptions from the present section and the characteristics of FASTQ files, there are excellent opportunities for similarity-based deduplication, which we discuss in the next section.

4.2 Similarity-based Deduplication

Similarity-based deduplication matches resembling objects of any size using similarity search to deduplicate them [13]. We integrate similarity-based deduplication with delta-encoding, which stores (1) a *pointer* to the most similar entry together with (2) the *minimal list of modifications* to restore the original object from this entry. This most similar entry is known as the *base chunk* [12].

Associating this approach with the application-aware deduplication is intuitively a promising solution to deduplicate genomes. However, there are at least three challenges that need to be addressed: (1) *choosing a distance metric and encoding*, (2) *modelling the deduplication index*, and (3) *reducing the number of candidate comparisons*.

A *distance metric* is critical as it defines what makes entries similar and determines how to choose the best deduplication candidates. In this work, we consider three metrics and present experiments using them in §6.

- HAMMING: Counts the number of positions with different characters in two strings of the same size. The resulting list of edit operations is composed of only `UNMODIFIED` and `SUBSTITUTION` operations.
- LEVENSHTEIN: Calculates the minimal number of modifications to convert a string into another. It considers `UNMODIFIED`, `DELETE`, `INSERT`, and `SUBSTITUTION` operations. Since it considers insertions and deletes, it allows comparing strings with different sizes.
- JACCARD: Calculates the ratio between the intersection and the union of N-grams from the strings. It also is independent of the size of the to-be-compared strings.

The first two metrics return the distance value and a list of edit operations to restore the original data from the base chunk, whereas the last one provides only the distance.

After choosing the distance metric, one may *model the deduplication index* based on it. It is an optimization process that selects a subset of (real or synthetic) entries and results, for example, in the smallest distance sum to a known sample of sequences. As previously mentioned, human DNA sequences have a comprehensive reference (i.e., *hg38*) that can be used to create such an index, *but there is no such reference for QS sequences* [3]. To create the index for quality score sequences, one may resort to optimization, memetic (e.g., [32]), or clustering (e.g., K-Means [35]) algorithms to find the best codebooks to the deduplication task.

Another option is to choose the most frequent sequences from each file empirically. However, naively creating the index with entries exactly as they appear in FASTQ files is inefficient due to a combinatorial explosion. Finally, one may initiate the system with an empty deduplication index and dynamically insert every queried entry that has not found a similar enough neighbor (i.e., under a predefined threshold). However, the index may grow indefinitely if the threshold is too hard to achieve, or it will result in low reduction gains if the threshold is too easy to reach.

After obtaining a deduplication index that achieves satisfactory compression results, one may decide how to improve the scalability and performance of the system [36]. The human DNA reference provides nearly 3.2 billion base chunks. As mentioned before, QS sequences do not have a reference, and thus one may define the limits of the index size according to its capacity. For instance, storing 1 billion entries of 100 characters each in a simple key-value store, indexed by integers of 32-bits, results in at least 100GB of data. Keeping all data in main memory in a single node may become a burden, and thus partitioning data across several nodes [37] or using sparse indexes [38] emerge as desirable alternatives.

Finally, *reducing the number of candidate comparisons* is another crucial performance improvement to the system. One may achieve this goal through other auxiliary data structures such as K-mer tables [39], indexes for Locality-Sensitive Hashing (LSH) [40], or cluster deduplication [37]. However, these structures may interfere with the recall of the best deduplication entries, producing suboptimal search results depending on their configuration. It means that there is a trade-off in improving the performance that may compromise the deduplication gains.

5 GENODEDUP

In this section, we describe GenoDedup, which integrates scalable, efficient similarity-based deduplication and specialized delta-encoding for sequencing data. In Section 5.1, we present the main components of GenoDedup and how data flows among them. Sections 5.2 and 5.3 detail how we solve the three main challenges from Section 4.2.

5.1 Overview

The main components of GenoDedup can be seen in Figure 2. The similarity-based deduplication selects the nearest base chunk for each sequence in FASTQ entries using two auxiliary data structures. The first is a Locality-Sensitive Hashing (LSH) index, which enables the similarity search when the number of deduplication candidates is too big to perform optimal searches. Entries are blocks with a variable size similar to the length of the DNA and QS lines in the FASTQ files used in this work. The second data structure is a key-value store (KVS) indexing unique entries that are used in optimal similarity searches and to retrieve the value of deduplication candidates using their content hashes as keys. A data storage component is used to store the deduplicated files and provide them to readers. Readers use a restore module, which reads the pointers and delta-encoding from the deduplicated file and queries the deduplication index of unique entries to restore the original FASTQ file from it.

An offline setup phase, described in §5.2 but not shown in Figure 2, prepares the environment where the deduplication will take place. This phase populates the auxiliary data structures (i.e., LSH and KVS) with the previously generated list of deduplication candidates. For instance, the human reference genome (e.g., *hg38*) can be loaded to the LSH and KVS during this phase. At the end of this offline phase, data has been loaded to the appropriate data structures in a way that similarity search can be efficiently executed.

Data flow during a deduplication execution is composed of the numbered steps present in Figure 2. Steps 1–21 represent the deduplication process, while steps 22–34 represent the FASTQ restore process. Squared steps are processor-bounded tasks, circular steps are disk-bounded, and triangular ones are network-bounded.

When sequencing a genome, (1) NGS machines generate data at 0.3MB/s, which is (2) stored in a disk that supports this throughput. Similarity-based deduplication receives the sequenced data by (3) reading it from the disk and (4) transferring it through the network to the deduplication component. Then, (5) each FASTQ entry is parsed into the different line types, where comments are sent to Step 18 (see below), DNA to Step 7, and QS to Step 6. QS sequences are (6) converted to circular deltas, and QS and DNA sequences are used to (7) calculate the hashes that will be used to query the LSH. These hashes are (8) sent to the LSH component, which will (9) obtain the internal LSH keys from these hashes, query them in the respective LSH indexes, and join the lists of pointers to the candidates in a bigger list, which is (10) returned to the deduplication component.

The deduplication component (11) receives this list of pointers to candidates and (12) sends it to the KVS to obtain their content. The KVS (13) obtains the candidate value using each pointer as a key and (14) returns the list of candidates (their content, not the pointers). The deduplication algorithm (15) calculates the edit distance only (not the edit operations) between each candidate from the received list and the sequence from the FASTQ file and keeps track only on the pointer and value of the best candidate (i.e., the one with the smallest edit distance). After identifying the best candidate, it (16) calculates the edit operations between the sequence from the FASTQ file and the best candidate and (17) converts the edit operations to the delta-encoding using Huffman codes. In parallel to this process, the deduplication component (18) compresses the comment lines with an external algorithm (e.g.,Bhola *et al.* [17]). At the end, the component (19) joins the deduplicated and compressed version of the comment, DNA, and QS lines and (20) sends the reduced entry to a storage component, which (21) writes the entry in a deduplicated file.

When a client intends to read a deduplicated FASTQ file, he (22) reads the file from the disk and (23) transfers it to the FASTQ restore component. The restore module (24) converts, both for the DNA and QS sequences, the bytecode to the pointer to the best candidate, to the first character of the original QS sequence and the delta-encoding. For each sequence (25), the restore module (26) sends the pointer to the KVS, which (27) obtains the respective value indexed by the pointer as a key and (28) returns the value of the best candidate to the restore module. The restore module then (29) applies the edit operations from the delta-encoding to the returned candidate and (30) converts from circular delta QS to normal QS if it is a QS sequence. Finally, it (31) decompresses the comment lines using an external algorithm (e.g., Bhola *et al.* [17]) and (32) joins the restored comment, DNA, and QS lines. The restore entry is (33) sent to the client, which (34) stores it in a FASTQ file on disk.

Steps 12 and 14 can be avoided if the LSH index stores and returns the list of the actual content of the deduplication candidates instead of their content hashes. These content hashes are used as pointers to retrieve the candidate content from the KVS index with unique entries. We opted to store only the content hashes of candidates in the LSH because it makes the size of LSH index smaller and linearly proportional to the number of entries, independent on the candidate's size.

5.2 Offline Phase

Modelling the group of base chunks that will be inserted in the deduplication index is paramount to achieve satisfactory reduction gains. As previously mentioned, we suggest the use of the available human reference genome *hg38* as the deduplication index for DNA sequences. For QS sequences, we resort to representative entries that result in the smallest sum of distances to a group of real entries, e.g., from the SRR618666 genome.

GenoDedup converts the original input QS sequences to circular delta values (see §2) and employs clustering algorithms to distribute them into a predefined number of clusters. Centroids from the resulting clusters are stored in a file that is loaded to the deduplication index during an offline setup phase. GenoDedup employs the Bisecting K-means from Apache Spark, which is a faster and more scalable hierarchical divisive version of K-means [35]. Additionally, our solution can generate three orders of magnitude more base chunks than the memetic algorithm from Zhou *et al.* [32] in useful time. This scaling up allows us to create generic deduplication indexes from many genomes instead of generating one small codebook for each FASTQ file. In this paper, we deliberately select specific numbers k of clusters in the form of $k = 2^i$, where i ranges from 0 to 20, four by four.

At this point, the base chunks that compose the deduplication index are already defined and placed in the proper data structures.

5.3 Optimizations of the Online Phase

In this section, we describe two optimizations that balance storage space and performance in the similarity-based deduplication described in §5.1. The first one describes how the distance metric ant its encoding are implemented, whereas the second discusses how do we reduce the number of candidate comparisons.

5.3.1 Distance Metric and Encoding

Choosing a distance metric determines what makes entries similar while designing an optimal encoding provides reduction gains when describing entries as the differences to previously known sequences. GenoDedup includes all the three string distances (Hamming, Levenshtein, and Jaccard) mentioned in §4.2. Our implementation uses the *java-string-similarity* library,[11] which provides implementations for these distances. We employ Huffman codes to encode the divergent characters in all metrics.

The encoding sizes of Hamming and Levenshtein algorithms are presented in §1 of our Supplementary Material. We propose to extend the Hamming algorithm to aggregate subsequent matching characters—in an encoding dubbed *Delta-Hamming*—and replace them by a delta number that informs how many characters should be skipped before finding the next substitution. For example, applying this comparison algorithm between the

11. https://github.com/tdebatty/java-string-similarity

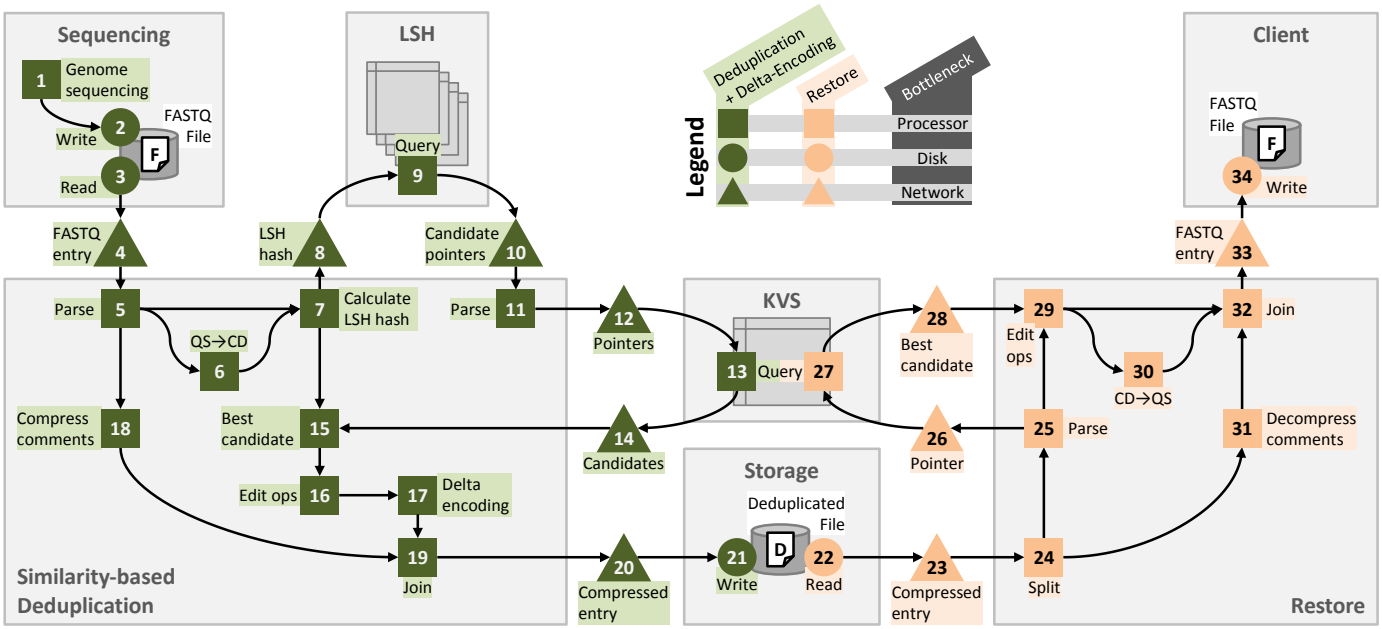

Figure 2. Overview of the architecture of GenoDedup.

strings "ABCDEFGH" and "AXCDEYZH" results in the follow operations: "1X3YZ", while the result in the original Hamming is: "USXUUUSYSZU". This algorithm results in the encoding size presented in Equation (1).

$$SizeDH = M + C_0 + (len(\diamond) * (5)) + (len(\circ) * (1 + huf(\bullet))) \quad (1)$$

The size of the candidate pointer M (in bits) corresponds to $M = log_2(N)$, where N is the expected number of entries in the deduplication index. C_0 describes the first character in the original sequence, which allows one to initiate a chain of conversions from circular delta values to the original quality score sequence. C_0 is unnecessary for DNA sequences since they do not use delta values. It can be a fixed-size value or a Huffman code based on the distribution of the first character observed from several FASTQ files. Function $len(\diamond)$ is the quantity of delta numeric characters (i.e., [0–9]) in the string, where each one is represented by five bits. Function $len(\circ)$ is the number of differing characters in the string, where each one is represented using Huffman codes.

One of the main advantages of this approach is that its encoding size is not lower-bounded by the length of the sequences ℓ. For instance, if two strings are identical, the encoding results only in a special code of five bits to inform that there is no additional edit operation in the comparison. Contrarily, the encoding of Hamming and Levenshtein algorithms are lower-bounded by ℓ bits informing that there are ℓ `UNMODIFIED` operations (see §1 of our supplemental material).

5.3.2 Number of Candidate Comparisons

The number of candidate comparisons executed on each query influences the search performance and directly depends on the employed algorithm and configuration. We implement two forms of similarity search: optimal and probabilistic.

In the former, the system loads all modeled base chunks to a list in main memory and compares each queried sequence to all entries in this list. This process is inefficient when the number of candidates is very large. However, it always finds the best candidate (i.e., the nearest neighbor) in the index and is a feasible solution for small indexes.

In the latter, the system inserts all base chunks into an efficient data structure, called Locality-Sensitive Hashing (LSH) index, and compares each queried sequence only to entries that belong to the same buckets as the queried sequence. It effectively reduces the number of candidates to be compared.

LSH is an algorithm that, given an entry, returns a content hash that has a high probability of colliding with the hash of similar objects—and a low probability of colliding with distinct ones [40]. This idea is the opposite of cryptographic hashes, where even very similar objects should generate very distinct content hashes.

The resulting hash from LSH is composed of a group of k smaller hashes (e.g., integers). The LSH index is composed of k multimaps—i.e., a KVS where each key maps to a list of values. Each smaller hash from the LSH hash is the key to one of these multimaps. In an insert operation, the LSH hash from the received sequence is obtained, and the object is appended in the list of values mapped by each small hash in the respective multimap.

In a query operation, the LSH hash is also obtained, and the result is the joint set of values mapped by the small hashes in the respective multimaps. The best candidate sequence is obtained by calculating the string distance of choice between the queried sequence and all base chunks present in the returned small joint set. Finally, the chosen base chunk is used to calculate the delta-encoding, which is the minimal list of edit operations necessary to restore the queried sequence from the base chunk.

In GenoDedup, we implement the LSH hash as a MinHash [41], which is proportional to the Jaccard distance—i.e., the ratio between the intersection and union of two sets. It means that sequences that are more similar than a given threshold will have a higher probability of being placed in the same bucket in at least one multimap. We also implement bitsampling techniques [42] in our LSH hash to reduce its size and to become even more efficient in space and time.

To implement the LSH index, we extended the Chronicle-

Map library[12] to provide a multimap instead of their original key-value store. GenoDedup benefits from Chronicle's principles and results in a well-engineered solution that provides: off-heap techniques to avoid garbage collection; efficient persistent storage to support data bigger than the available main memory; multi-threads and fine-grain locks to support multiple writers and readers; collections of objects as small as Java primitives to avoid space overhead; etc. Finally, our deduplication index supports four orders of magnitude more base chunks than the values reported by other solutions for similarity-based deduplication [13], [14].

6 Evaluation

We evaluate our Java prototype of GenoDedup to illustrate the strengths and limitations of similarity-based deduplication in genome sequencing data. It is open-source and publicly available on GitHub[13].

Experiments are divided into three parts: the encoding size of deduplicated entries; a performance evaluation; and an end-to-end scenario with a large workload. In the first two experiments (§6.1 and §6.2), our testing dataset is the first two hundred and fifty thousand FASTQ entries from the SRR618666 genome, which properly represents the diversity of its entries. We use this subset, instead of all portions of this genome, because we intend to evaluate an optimal (exhaustive) search algorithm that compares every queried sequence to all candidates in a deduplication index. Testing this optimal search with the whole genome (instead of using only these 250k reads) would make it infeasible to complete these tests in practical time when using indexes with more than 2^{16} deduplication entries. This optimal search is also important to identify the expected performance of the system given different number of candidates returned by the LSH optimization (§5.3.2).

Tests with DNA sequences are directly executed using the original FASTQ file and the human reference genome *hg38*. Tests with QS sequences first convert them to circular delta values (§2) and compare them to entries in the deduplication index, which also are encoded as circular deltas.

Experimental Setup

The experimental setup is composed of a Dell PowerEdge R430 server, equipped with 2 Intel Xeon E5-2670v3 processors (2.3GHz), 128GB of RAM (DIMM 2133MHz) and a 300GB disk of 15k RPM with an average sequential write and read throughput of 215MB/s. The operating system used was an Ubuntu 16.04.2 LTS x86_64.

6.1 Encoding Gains

In this experiment, we compare the average size (in bits) of delta-encoded entries using Hamming edit operations, Levenshtein, and Delta-Hamming ones. They include a pointer to the most similar deduplication entry and the encoded edit operations to transform it back into the original sequence.

To validate the differences on the data entropy of each portion of FASTQ entries, we separate and compress them individually with ZPAQ. For instance, the file with the first comment line of every FASTQ entry from the SRR618666 sizes 15.1GB. ZPAQ compresses it 6.43× to 2.3GB. The DNA and QS portions of this genome size 23.3GB each. ZPAQ compresses the former to 5.4GB (4.33×) and the latter to 7.4GB (3.14×).

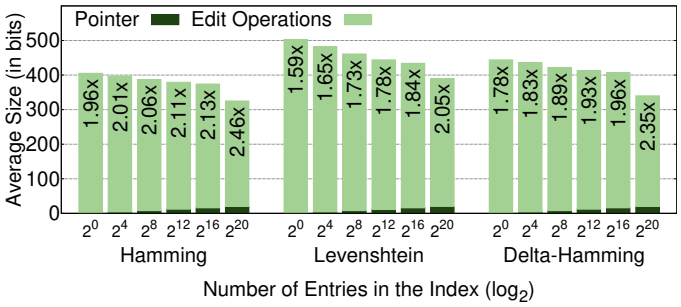

Figure 3. Average encoding size of deduplicated QS sequences (in bits) and its reduction ratio.

Every queried DNA and QS sequence has 100 characters, which means that each one of them occupies 800 bits in text mode originally. When using the human reference genome *hg38* as the deduplication index, DNA sequences are compressed 13.43× with our *Delta-Hamming* encoding, whereas ZPAQ compresses it only 4.33×. We used only the Delta-Hamming encoding for DNA sequences because the encoding of Hamming and Levenshtein are bounded up to 8× and 4×, respectively (see §1 of our Supplementary Material for more details on their encoding).

For QS sequences, the results from Figure 3 show that Hamming encoding achieves a smaller output size than Delta-Hamming, which is smaller than Levenshtein. Their best case (i.e., 2.46× considering the Hamming distance and 2^{20} index entries) already achieves nearly 80% of the reduction gains from the ZPAQ algorithm when considering only quality score sequences—3.14× for SRR618666. Our solution can obtain even better reduction gains with bigger indexes.

6.2 Performance

In this section, we evaluate the read and write performance of GenoDedup both for DNA and QS sequences. We discuss the performance of the deduplication and restoring processes only in the aspects that our algorithm and implementation may have a bigger impact or may represent a bottleneck to the workflow. More specifically, GenoDedup is compute-bound, mostly by finding the best candidate, which requires calculating the distance metric between the query and all returned candidates. For this reason, we *do not* evaluate in this section:

- Processing bottlenecks on services (e.g., LSH and KVS), because they can be placed in local memory if they are small enough or they can horizontally scale by using multiple nodes;
- Bottlenecks from parsing and direct data conversion, because they usually are significantly faster than the main processing steps we evaluate here;
- Disk bottleneck, because it is specific to the hardware used in the experimental environment and it can be avoided by processing entries from multiple files on different disks up to the point the processing becomes the main bottleneck again;
- Network bottleneck, because it is also specific to the experimental environment and can be avoided with faster networks (e.g., 10Gbps instead of 1Gbps).

12. https://github.com/OpenHFT/Chronicle-Map
13. https://github.com/vvcogo/GenoDedup

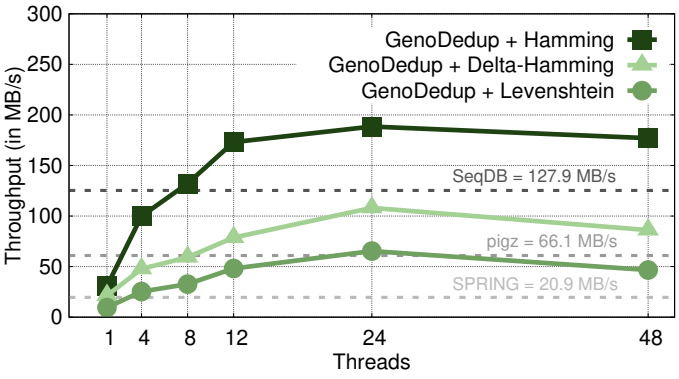

Figure 4. Average throughput of restore operations in GenoDedup with different numbers of threads.

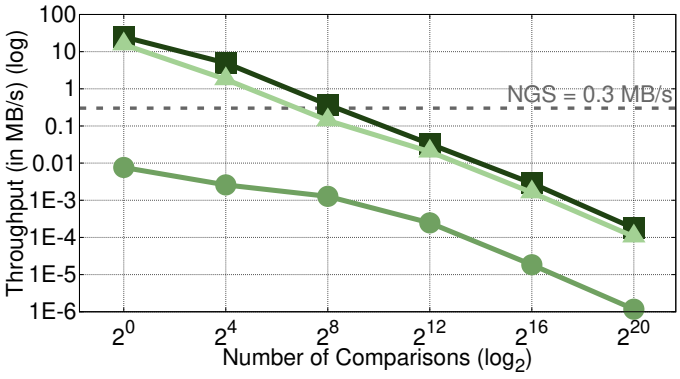

Figure 5. Average throughput of GenoDedup writes with different indexes for QS using a single thread.

6.2.1 Read Operations

Our next experiment aims to evaluate the performance of applying the list of edit operations (i.e., the delta-encoding) in a base chunk to restore the original sequences from FASTQ entries (i.e., steps 29 and 30 in Figure 2). In theory, it is directly related to the number of differences between the two sequences, where the less differences they have, the faster it is.

Figure 4 presents the throughput (in MB/s) that GenoDedup reaches in applying the identified edit operations with different parallelism configurations (from 1 to 48 threads). The length of the analyzed sequences is 100 characters. As expected, the more threads processing requests up to the number of physical cores, the bigger the throughput. For instance, GenoDedup restores entries in a single thread at a pace of 30.8MB/s with the Hamming encoding, 21.3MB/s with Delta-Hamming, and 9.5MB/s with Levenshtein. Since the machine in our experimental environment has 24 physical cores (i.e., two processors with 12 cores each), using 24 threads obtains the best results: 188MB/s with Hamming, 108.1MB/s with Delta-Hamming, and 65.3MB/s with Levenshtein.

The Hamming algorithm results in the best throughput because it is the simplest encoding to be restored. The Hamming and Levenshtein algorithms have the number of operations directly proportional to the length of the used sequences since they store `UNMODIFIED` operations when characters from both sequences match. However, the Delta-Hamming has potential in obtaining a higher throughput as the modeled index becomes better. If the best base chunk for each queried sequence results in less `SUBSTITUTION` operations, then the Delta-Hamming becomes proportionally smaller and reduces the restore time.

As mentioned in §3, the restore throughput from the ten selected compression algorithms range as follows: ZPAQ, Quip, and Fqzcomp reach less than 10MB/s; SPRING reaches 20MB/s; GZIP, BSC, and FaStore reach 40–50MB/s, pigz reaches 66MB/s, and DSRC2 and SeqDB reach 125MB/s. These values refer to decompressing the whole FASTQ file in the specialized tools, not only quality scores as in the results from Figure 4. Restoring only the QS data from GenoDedup is up to 2.84× faster than pigz, the fastest generic competitor and up to 170× faster than ZPAQ, the generic algorithm that with the best compression ratio for QS.

6.2.2 Write Operations

We evaluate the performance of string comparisons using different encoding algorithms and how do they interfere with the performance of GenoDedup (i.e., mainly steps 15–17 in Figure 2). Similarity-based deduplication directly depends on the number of comparisons necessary to find the nearest neighbor (i.e., the best candidate) of a queried sequence. Genome sequencing data is usually written once and read many times later for processing. Systems for genome sequencing data should support a write throughput of at least 0.3MB/s—the average write throughput from an NGS machine [4]—to not become a bottleneck in an NGS pipeline.

Figure 5 presents the throughput (in MB/s) obtained when comparing a single queried sequence to *all entries* in deduplication indexes of different sizes (from 2^0–2^{20}) using a single thread. As expected, the more entries to compare in the index, the smaller the throughput is. More specifically, GenoDedup reaches 25MB/s when comparing the queried sequence to a single candidate using Hamming encoding (i.e., 0.004ms per comparison), 15.5MB/s using Delta-Hamming (ie.e., 0.0064ms), and 0.0076MB/s using Levenshtein (i.e., 13.11ms). From these results, Levenshtein is two to three orders of magnitude slower than Hamming and Delta-Hamming algorithms.

GenoDedup must process at least 3000 queries (of 100 characters each) per second to support the 0.3MB/s throughput from NGS machines [4]. The maximum number of comparisons on each query, to reach at least 0.3MB/s, is 422 for Hamming and 113 for Delta-Hamming. Levenshtein requires a speedup of 40 to reach 0.3MB/s when comparing to a single entry, which makes this algorithm unappealing for our solution. Parallelization can improve GenoDedup to support higher write throughput in the future. Currently, it means that queries should return fewer deduplication candidates than these numbers; otherwise, the system does not support the required throughput of 0.3MB/s.

Conveniently, reducing the number of candidates returned in a query is the exact benefit LSH brings to GenoDedup. It makes Hamming and Delta-Hamming algorithms even more feasible. For instance, in an LSH with a similarity threshold of 0.95 and 128 permutations on the MinHash, it can reduce the number of candidate comparisons from 350 million entries (333× bigger than 2^{20}) to only 50. Such reduction contributes, for instance, to approximate the write throughput of GenoDedup with the one of the ZPAQ (i.e., 5.3MB/s). GenoDedup has the potential of reaching higher write throughput with parallelism since FASTQ entries are processed unrelated with each other.

These results consider only the string comparison. Usually, there are other steps (e.g., communication and parsing) that need

to be considered. However, 0.3MB/s per second is an achievable throughput for most modern service solutions.

6.3 Large End-to-End Workload

Our last experiment evaluates GenoDedup with a large workload in an end-to-end scenario. The evaluation considers approximately 265GB from all files in Table 1. We compare our results with the most prominent competitors in terms of compression ratio (i.e., SPRING [15]) and read throughput (i.e., SeqDB [16]). We also add to the comparison DSRC2 [25] and pigz). Table 2 presents the observed results. The complete set of components in our end-to-end solution encompasses: (1) the algorithm from Bhola *et al.* [17] to compress FASTQ comments with a ratio of 17.26× on average; and our similarity-based deduplication both (2) for DNA sequences, which compresses them 13.43×; and (3) for QS sequences (with 2^8 entries), which compresses them 1.88×.

SPRING compresses the mentioned FASTQ files 6.023×, which is the biggest compression ratio observed in our experiments. For the same dataset, SeqDB achieves a compression ratio of 1.992×, DSRC2 4.148×, and pigz 3.227×. Our end-to-end deduplication solution achieves a compression ratio of 4.089× using a deduplication index with the human reference genome *hg38* for the DNA sequences and with 2^8 entries for the QS sequences. This result corresponds to 67% the compression ratio of SPRING, 98.6% the ratio of DSRC2, while we compress 2.05× more than SeqDB and 1.26× more than pigz.

Our solution compresses data at 0.3MB/s with an index of 2^8 candidates for the QS sequences in a single thread. The other solutions perform better than us in terms of the compression speed. However, GenoDedup can reach higher speeds with multi-threading, with better deduplication indexes, and with the use of LSH to reduce the number of candidate comparisons (e.g., it reaches almost 10MB/s when comparing 2^4 entries).

In terms of restoring throughput, SeqDB decompressed the selected FASTQ files at 127.9MB/s (i.e., the fastest competitor), DSRC2 at 125.3MB/s, pigz at 66.1MB/s, and SPRING at 20.9MB/s. Our end-to-end solution achieves a read throughput of 208.25MB/s in the same dataset. This result makes GenoDedup 1.62× faster than SeqDB, 1.66× faster than DSRC2, 3.15× faster than pigz, and 9.96× faster than SPRING.

An important aspect of our solution is that achieving better compression ratios is possible in the future and it does not compromise our read throughput since it is independent of the number of candidates in the deduplication index. Furthermore, a better compression ratio implies more `UNMODIFIED` edit operations, which accelerates the restore process even more.

Applying these results to a repository like the 1000 Genomes Project [11] provides a better figure of the savings GenoDedup can bring to big genome data warehouses. The project currently stores approximately 115TB of FASTQ files compressed with GZIP (i.e., the equivalent to 370TB of uncompressed FASTQ files). By using our deduplication strategy, it would be able to store such files using only 90TB, which corresponds to 78% of the 115TB used today with GZIP. Perhaps even more importantly, GenoDedup would also improve their data sharing ecosystem by allowing data consumers to restore FASTQ files 5× faster than today (i.e., GZIP decompressed our files at 41.4MB/s).

7 Discussion

The methods proposed in this work are generic enough to support sequencing data from other species and NGS machines, as well as other data representations (e.g., aligned data) and file formats (e.g., SAM). Additionally, it can be explored in other highly-dimensional data where identity-based deduplication fails [13]. In this section, we discuss how our solution can work with or be adapted to support other datasets and methods.

Table 2
Comparison (i.e., the ratio *best*/*worst*) of the compression ratio (C.R.) and the (write and read) throughput between GenoDedup with 2^8 QS candidates and its main competitors. Brackets ({}) indicate where competitors are better than GenoDedup.

Algorithm	C.R.	Write	Read
SPRING	{1.47×}	{143.6×}	9.96×
DSRC2	{1.01×}	{4586.3×}	1.66×
SeqDB	2.05×	{1385.3×}	1.62×
pigz	1.26×	{937×}	3.14×

7.1 Other Data Representations

As previously mentioned, *sequencing data* is considered the purest unbiased version of genomic data coming from Next Generation Sequencing (NGS) machines [5]. Contrarily, the output from alignment and assembly processes is imprecise, lossy, and depend on the employed algorithm and reference [9].

For instance, aligned data in the 1000 Genomes Project [11] was generated using different algorithms and references in distinct phases of the project. Studies that use data from multiple of these phases must first reconvert the aligned data into sequencing data, and then realign all the data of interest using the same reference and algorithm before analyzing it. This rework in converting from aligned to sequencing data takes considerable additional time (e.g., 50–200 minutes for each 100GB [43]) and is even more likely to be required in studies that involve large quantities of samples and data from multiple sources. It is no surprise that the 1000 Genomes Project stores the original sequencing data (i.e., FASTQ) of every aligned data (i.e., SAM/BAM) they have.

Notwithstanding, our methods can be used with *aligned data* (e.g., in the SAM/BAM format [6]). The only difference is that this data representation already contains the pointer to the best candidate of DNA sequences in the aligned file. It only eliminates the need to execute the similarity-based deduplication for the matched DNA sequences. However, it still requires (1) a compression algorithm for the QS sequences, in which our similarity-based deduplication has shown its potential, (2) a delta-encoding (e.g., our Delta-Hamming) for the matched DNA sequences, and (3) a similarity-based solution like ours to the unmatched ones.

Another important data representation is the *assembled data*. However, recovering the original sequencing data file from assembled data is impossible because the resulting assembled genome file (i.e., FASTA) does not contain details such as: How many FASTQ entries were used to create the assembled genome? In which genome position each one of them started and ended? Which was the quality score for each sequenced nucleobase? Additionally, we consider the compression of assembled data a challenge that has been mostly addressed by different approaches. For instance, we devised a tool that reduces assembled human genomes ∼700× in 40 seconds [8].

7.2 Paired-end Sequencing

Paired-end sequencing digitizes both ends of DNA fragments to increase accuracy and help detecting repetitive sequences and

rearrangements. It produces two FASTQ files, one for each end, where the entry order matches among them (i.e., entry 100 from the second file is the reverse complement of the DNA sequence of entry 100 from the first file). GenoDedup may explore this additional redundancy in the future to eliminate the similarity-search for the DNA sequences of the second FASTQ file. In this case, GenoDedup may use the reverse complement of the best candidate of the DNA sequence from the first file also as the best candidate for the DNA sequence in the second file. It will reduce the size of our encodings and speedup the deduplication and restore since only one pointer is used as the best candidate of two sequences.

7.3 Other Species

In this work, we favor sequencing data from human genomes due to the the availability of a comprehensive reference genome and the potential impact of this data domain. However, the methods proposed in this work can easily be adapted to work with sequencing data from other species. In an extreme case, one will end up with one deduplication index for DNA of each species. Moreover, species with a representative reference genome have the advantage of using it as the deduplication index for DNA, but it is not a requirement since the same method of modelling the index of QS sequences can be used for DNA.

7.4 Other Sequencing Machines

Many related works for FASTQ compression select genomes from several species and sequencing machines. This choice usually results in selecting only a few genomes per species or selecting small FASTQ files with very low coverage. We intended to select more and bigger genomes from the same species and the same sequencing machine to reduce the influence from these two variables in our results.

Our datasets include only human genomes (due to the previously mentioned reasons) sequenced with the Illumina HiSeq 2000 platform. To the best of our knowledge, this machine was the most used NGS machine in sequencing laboratories around the world when we started this work [20]. Additionally, some of the selected datasets were used also in other papers on FASTQ compression (e.g., SRR400039 in Quip's paper [26]).

Datasets in our work had an expected alphabet of 40 possible QS.[14] Newer Illumina platforms have been binning QS into groups with reduced alphabets (e.g., seven groups in HiSeqX10[15] and four in NovaSeq[16]). This binning is similar to the initial approach of many lossy FASTQ compression algorithms [44]. These smaller alphabets reduce the size of our encoding and may benefit our index modelling since it reduces also the possible combinations. Notwithstanding, our methods can work with data from most of the modern NGS machines. The differences in QS distribution patterns and alphabet may require one to model new deduplication indexes and one may end up with one index per machine in an extreme case.

14. https://www.illumina.com/documents/products/technotes/technote_Q-Scores.pdf

15. https://www.illumina.com/content/dam/illumina-marketing/documents/products/appnotes/appnote-hiseq-x.pdf

16. https://www.illumina.com/content/dam/illumina-marketing/documents/products/appnotes/novaseq-hiseq-q30-app-note-770-2017-010.pdf

7.5 Other Sequence Lengths

Our methods support sequences of any length, while working even with indexes containing sequences with multiple lengths. The length influences several aspects in our solution (e.g., index size, string comparison time, chances of finding more differences). More specifically, LSH supports entries of different sizes when using MinHash. MinHash converts entries of any size into hashes of fixed size proportional to Jaccard distance (i.e., also independent on the entry size). Furthermore, Levenshtein edit operations can compare strings of different sizes since it includes insert and delete operations. In an extreme case, one may model a few different deduplication indexes for different entry sizes. However, big differences in the size of the query sequences and the modelled ones may reduce the compression ratio.

Additionally, the bigger the sequence length is, the bigger the chances of having more edit operations, which tends to reduce the compression ratio and throughput on the selection of the best candidate. The impact of this length in GenoDedup is proportional to the impact of the sequence length in the string distance calculation.

7.6 Reordering FASTQ entries

Reordering FASTQ entries to group similar entries is another pattern explored in the literature [15], [45]. The current version of our methods work entry by entry, without correlating them or their order. GenoDedup compresses the DNA and QS sequences independently and reordering them would reduce its compression ratio and performance since it requires storing and working with additional correlating metadata.

on a per-entry Reordering the QS sequences to group similar entries is another pattern explored in the literature [45]. However, we do not employ reordering because it has produced only minor gains in reducing sequencing data—e.g., less than 6% compared to compressing entries in the original order. Furthermore, reordering both DNA and QS sequences separately and storing them in an unmatched order nullify these reduction gains since it requires additional correlating metadata.

8 Conclusion

In this work, we presented GenoDedup, the first method that integrates efficient similarity-based deduplication and specialized delta-encoding for genome sequencing data. Our experimental results attested that our method currently achieves 67.8% of the reduction gains of SPRING (i.e., the best specialized tool in this metric) and restores data 1.62× faster than SeqDB (i.e., the fastest competitor). Additionally, GenoDedup restores data 9.96× faster than SPRING and compresses files 2.05× more than SeqDB.

References

[1] A. Alyass, M. Turcotte, and D. Meyre, "From big data analysis to personalized medicine for all: challenges and opportunities," *BMC Med Genomics*, vol. 8, no. 1, p. 33, 2015.

[2] V. Marx, "Biology: The big challenges of big data," *Nature*, vol. 498, no. 7453, pp. 255–260, 2013.

[3] D. Pavlichin, T. Weissman, and G. Mably, "The quest to save genomics: Unless researchers solve the looming data compression problem, biomedical science could stagnate," *IEEE Spectrum*, vol. 55, no. 9, pp. 27–31, 2018.

[4] L. Liu *et al.*, "Comparison of next-generation sequencing systems," *BioMed Research International*, vol. 2012, 2012.

[5] P. Cock *et al.*, "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants," *Nucleic Acids Res.*, vol. 38, no. 6, pp. 1767–1771, 2010.
[6] H. Li *et al.*, "The sequence alignment/map format and SAMtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.
[7] J. C. Venter *et al.*, "The sequence of the human genome," *Science*, vol. 291, no. 5507, pp. 1304–1351, 2001.
[8] F. Alves, V. V. Cogo, S. Wandelt, U. Leser, and A. Bessani, "On-demand indexing for referential compression of DNA sequences," *PLoS ONE*, vol. 10, no. 7, p. e0132460, 2015.
[9] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," *Brief. Bioinformatics*, vol. 11, no. 5, pp. 473–483, 2010.
[10] E. R. Mardis, "Next-generation DNA sequencing methods," *Annu. Rev. Genomics Hum. Genet.*, vol. 9, pp. 387–402, 2008.
[11] L. Clarke *et al.*, "The 1000 Genomes Project: data management and community access," *Nature methods*, vol. 9, no. 5, pp. 459–462, 2012.
[12] J. Paulo and J. Pereira, "A survey and classification of storage deduplication systems," *ACM CSUR*, vol. 47, no. 1, p. 11, 2014.
[13] F. Douglis and A. Iyengar, "Application-specific delta-encoding via resemblance detection," in *Proc. of the USENIX ATC*, 2003, pp. 113–126.
[14] L. Xu, A. Pavlo, S. Sengupta, and G. R. Ganger, "Online deduplication for databases," in *Proc. of the ACM SIGMOD*, 2017, pp. 1355–1368.
[15] S. Chandak, K. Tatwawadi, I. Ochoa, M. Hernaez, and T. Weissman, "SPRING: a next-generation compressor for FASTQ data," *Bioinformatics*, vol. 35, no. 15, pp. 2674–2676, 2018.
[16] M. Howison, "High-throughput compression of FASTQ data with SeqDB," *IEEE/ACM TCBB*, vol. 10, no. 1, pp. 213–218, 2013.
[17] V. Bhola, A. S. Bopardikar, R. Narayanan, K. Lee, and T. Ahn, "No-reference compression of genomic data stored in FASTQ format," in *Proc. of the IEEE BIBM*, 2011, pp. 147–150.
[18] B. Ewing, L. Hillier, M. C. Wendl, and P. Green, "Base-calling of automated sequencer traces using Phred. i. accuracy assessment," *Genome Research*, vol. 8, no. 3, pp. 175–185, 1998.
[19] C. Kozanitis, C. Saunders, S. Kruglyak, V. Bafna, and G. Varghese, "Compressing genomic sequence fragments using SlimGene," *Journal of Computational Biology*, vol. 18, no. 3, pp. 401–413, 2011.
[20] J. Hadfield, "NGS mapped," 2019, in: http://enseqlopedia.com/ngs-mapped/. Accessed on Feb. 10, 2020.
[21] F. Hach *et al.*, "Scalce: boosting sequence compression algorithms using locally consistent encoding," *Bioinformatics*, vol. 28, no. 23, pp. 3051–3057, 2012.
[22] M. Mahoney, "Data compression explained," 2013, in: http://mattmahoney.net/dc/dce.html. Accessed on Feb. 10, 2020.
[23] J. K. Bonfield and M. V. Mahoney, "Compression of FASTQ and SAM format sequencing data," *PLoS ONE*, vol. 8, no. 3, p. e59190, 2013.
[24] S. Deorowicz and S. Grabowski, "Compression of DNA sequence reads in FASTQ format," *Bioinformatics*, vol. 27, no. 6, pp. 860–862, 2011.
[25] Ł. Roguski and S. Deorowicz, "DSRC 2—industry-oriented compression of FASTQ files," *Bioinformatics*, vol. 30, no. 15, pp. 2213–2215, 2014.
[26] D. C. Jones, W. L. Ruzzo, X. Peng, and M. G. Katze, "Compression of next-generation sequencing reads aided by highly efficient de novo assembly," *Nucleic acids research*, vol. 40, no. 22, pp. e171–e171, 2012.
[27] L. Roguski, I. Ochoa, M. Hernaez, and S. Deorowicz, "FaStore: a space-saving solution for raw sequencing data," *Bioinformatics*, vol. 34, no. 16, pp. 2748–2756, 2018.
[28] S. Byma *et al.*, "Persona: a high-performance bioinformatics framework," in *Proc. of the USENIX ATC*, 2017, pp. 153–165.
[29] W. Tembe, J. Lowey, and E. Suh, "G-SQZ: compact encoding of genomic sequence and quality data," *Bioinformatics*, vol. 26, no. 17, pp. 2192–2194, 2010.
[30] Y. Zhang, K. Patel, T. Endrawis, A. Bowers, and Y. Sun, "A FASTQ compressor based on integer-mapped k-mer indexing for biologist," *Gene*, vol. 579, no. 1, pp. 75–81, 2016.
[31] M. Nicolae, S. Pathak, and S. Rajasekaran, "LFQC: a lossless compression algorithm for FASTQ files," *Bioinformatics*, vol. 31, no. 20, pp. 3276–3281, 2015.
[32] J. Zhou, Z. Ji, Z. Zhu, and S. He, "Compression of next-generation sequencing quality scores using memetic algorithm," *BMC Bioinformatics*, vol. 15, no. 15, p. 1, 2014.
[33] L. Freeman, R. Bolt, and T. Sas, "Evaluation criteria for data de-dupe," 2007, iNFOSTOR.
[34] V. V. Cogo and A. N. Bessani, "From data islands to sharing data in the cloud: the evolution of data integration in biological data repositories," *ComInG—Communications and Innovations Gazette*, vol. 1, no. 1, pp. 01–11, 2016.
[35] A. K. Jain, "Data clustering: 50 years beyond K-means," *Pattern Recognition Letters*, vol. 31, no. 8, pp. 651–666, 2010.
[36] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proc. of the USENIX Conf. on File and Storage Technologies (FAST)*, vol. 8, 2008, pp. 1–14.
[37] D. Frey, A.-M. Kermarrec, and K. Kloudas, "Probabilistic deduplication for cluster-based storage systems," in *Proc. of the 3rd ACM Symposium on Cloud Computing (SoCC)*, 2012, p. 17.
[38] M. Lillibridge *et al.*, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in *Proc. of the USENIX Conf. on File and Storage Technologies (FAST)*, vol. 9, 2009, pp. 111–123.
[39] Y. Zhang *et al.*, "Light-weight reference-based compression of FASTQ data," *BMC Bioinformatics*, vol. 16, no. 1, p. 188, 2015.
[40] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proc. of the 30th ACM Symposium on Theory of Computing (STOC)*, 1998, pp. 604–613.
[41] A. Z. Broder, "On the resemblance and containment of documents," in *Proc. of the IEEE Compression and Complexity of Sequences*, 1997, pp. 21–29.
[42] P. Li and C. König, "b-Bit minwise hashing," in *Proc. of the 19th Int. Conference on World Wide Web (WWW)*, 2010, pp. 671–680.
[43] G. Tischler and S. Leonard, "biobambam: tools for read pair collation based algorithms on BAM files," *Source Code for Biology and Medicine*, vol. 9, no. 1, p. 13, 2014.
[44] I. Ochoa, M. Hernaez, R. Goldfeder, T. Weissman, and E. Ashley, "Effect of lossy compression of quality scores on variant calling," *Briefings in bioinformatics*, vol. 18, no. 2, pp. 183–194, 2017.
[45] R. Wan and K. Asai, "Sorting next generation sequencing data improves compression effectiveness," in *Proc. of the IEEE BIBMW*, 2010, pp. 567–572.

Vinicius Cogo has a MSc in Informatics and is a PhD student from the Faculty of Sciences, University of Lisbon. He is member of the LASIGE research unit since 2009. His main research interests include the dependability of distributed systems and the efficient, secure storage of large-scale critical data.

João Paulo is an Assistant Researcher at HASLab, one of the research units of INESC TEC and University of Minho. He obtained his PhD degree, in the context of the MAP-i Doctoral Programme, from the universities of Minho, Aveiro and Porto in 2015. His current research is focused on large scale distributed systems with an emphasis on storage and database systems scalability, performance, security and dependability.

Alysson Bessani received the PhD degree in electrical engineering from Santa Catarina Federal University, Brazil, in 2006. He was a visiting professor at Carnegie Mellon University, in 2010 and was a visiting researcher with Microsoft Research Cambridge, in 2014. He is an associate professor in the Department of Computer Science, Faculty of Sciences, University of Lisbon, Portugal, and a member of LASIGE research unit and the Navigators research team. His main interests are distributed algorithms, Byzantine fault tolerance, coordination, and security monitoring. More information at http://www.di.fc.ul.pt/~bessani.