On the Compressibility of Large-scale Source Code Datasets

Antonio Boffa*[a], Roberto Di Cosmo[b], Paolo Ferragina[c,d], Andrea Guerra[d], Giovanni Manzini[d], Giorgio Vinciguerra[d], Stefano Zacchiroli[e]

[a] *École polytechnique fédérale de Lausanne (EPFL), Lausanne, Switzerland*
[b] *Inria and University Paris Cité, Paris, France*
[c] *Department L'EMbeDS, Sant'Anna School of Advanced Studies, Pisa, Italy*
[d] *Department of Computer Science, University of Pisa, Pisa, Italy*
[e] *LTCI, Télécom Paris, Institut Polytechnique de Paris, Palaiseau, France*

Abstract

Storing ultra-large amounts of unstructured data (often called objects or blobs) is a fundamental task for several object-based storage engines, data warehouses, data-lake systems, and key-value stores. These systems cannot currently leverage similarities between objects, which could be vital in improving their space *and* time performance. An important use case in which we can expect the objects to be highly similar is the storage of large-scale versioned source code datasets, such as the Software Heritage Archive [16]. This use case is particularly interesting given the extraordinary size (1.5 PiB), the variegated nature, and the high repetitiveness of the at-issue corpus.

In this paper we discuss and experiment with content- and context-based compression techniques for source-code collections that tailor known and novel tools to this setting in combination with state-of-the-art general-purpose compressors and the information coming from the Software Heritage Graph.

We experiment with our compressors over a random sample of the entire corpus, and four large samples of source code files written in different popular languages: C/C++, Java, JavaScript, and Python. We also consider two scenarios of usage for our compressors, called Backup and File-Access scenario, where the latter adds to the former the support for single file re-

*Work done while the author was at the Department of Computer Science of the University of Pisa

trieval. As a net result, our experiments show (i) how much "compressible" each language is, (ii) which content- or context-based techniques compress better and are faster to (de)compress by possibly supporting individual file access, and (iii) the ultimate compressed size that, according to our estimate, our best solution could achieve in storing *all* the source code written in these languages and available in the Software Heritage Archive: namely, in 3 TiB (down from their original 78 TiB total size, with an average compression ratio of 4%).

Keywords: Data compression; Source code; Storage systems; Locality-sensitive hashing; Software Heritage; Version control systems

1. Introduction

More and more organisations store and analyse growing amounts of data in cloud computing platforms. An important trend in this context is the separation of storage and computing, whereby data are stored in distributed cloud storage systems, whereas computing resources are spawned elastically on demand. This separation poses an amazing challenge as the surge in ultra-large unstructured data repositories — commonly referred to as object or blob storage — places unprecedented pressure on existing infrastructures in terms of storage and network capacity, energy consumption, and application requirements. This phenomenon is particularly evident across diverse storage engines [51], data warehouses [4], data-lake systems [60, 3], and key-value stores [34], posing the need for solutions that address their evolving and diverse demands. Moreover, the relentless growth of vast volumes of data managed by these systems further exacerbates the strain on their capacities, efficiency, and scalability.

A pivotal tool for mitigating the challenges posed by such a growth in data volume is data compression. In fact, data compression not only optimises storage utilization, but it also enhances data transfer speeds (as nodes exchange smaller amounts of data) and the effective use of computing resources (as more data can fit in main memory for efficient analysis), thereby improving the overall system performance.

Research papers about storage engines [5, 51, 45] mainly focus on managing large-scale data efficiently, emphasizing aspects like scalability, reliability, and metadata reduction, while exploiting the skew in access frequency of the stored objects. However, none of these papers explicitly discusses how to

leverage similarities among objects to improve compression effectiveness and, as a consequence, system performance. These benefits are especially prominent when considering the storage of *source code datasets*, which are highly repetitive, large, and ever-growing by nature. The significance of source code datasets is also witnessed by their increasingly pivotal role in AI [26], because of the growing interest in pre-trained models for code generation and summarization (e.g., GitHub Copilot[1] or Code Llama[2]) or for their surprising role in improving classic large language models [40]. Not to mention the flourishing industrial collaborations for building more effective AI-based solutions that deal with source code datasets,[3] which pose new challenges to these systems in terms of storage capacity, efficiency, and scalability.[4]

This motivates the study presented in this paper, which presents an ongoing effort at compressing the largest public archive of source code, known as the Software Heritage Archive [1]. In the remainder of this section, we discuss the Software Heritage Archive (Section 1.1), formalise the objectives of our study (Section 1.2), and summarise our contributions (Section 1.3).

1.1. The Software Heritage Archive

Software Heritage (SWH)[5] is an ambitious initiative launched in 2016 at Inria (France) aimed to collect, preserve, and share all the software available in source code form [1, 16].

At the time of this writing, the SWH Archive contains about 20 billion unique source code files gathered from more than 300 million projects, together with their full development history consisting of more than 4 billion unique commits created over 70 million authors. It is the largest existing collection of software source code, totalling 1.5 PiB of data, crawled over the years from several code hosting platforms such as GitHub, GitLab, Bitbucket, npm, and the (now-defunct) Gitorious and Google Code.

---

[1] https://github.com/features/copilot

[2] https://ai.meta.com/blog/code-llama-large-language-model-coding/

[3] See https://www.mongodb.com/press/mongodb-aws-collaborate-to-optimize-amazon-codewhisperer and https://www.softwareheritage.org/2023/12/04/hugging-face/

[4] We mention in this context the surge of the demand for high-capacity HDDs to support AI training, see https://www.theregister.com/2024/04/23/seagate_hdd_prices/.

[5] See https://www.softwareheritage.org

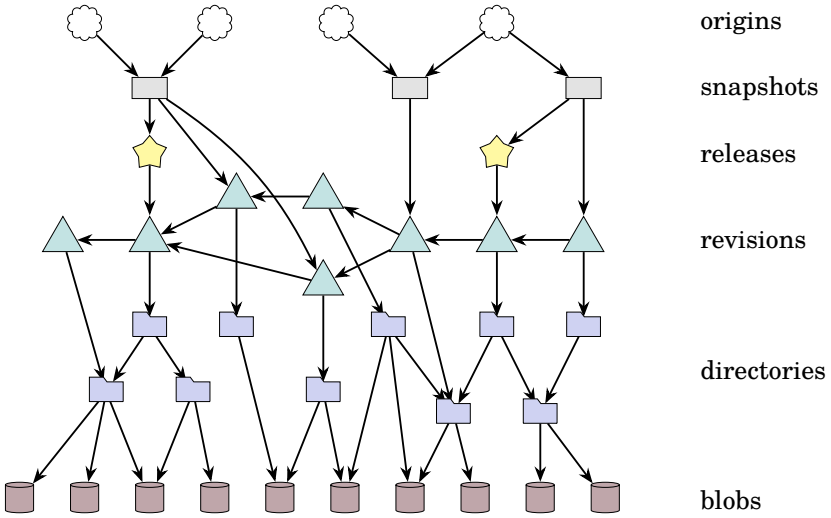

Figure 1: Data model of the Software Heritage Archive [52].

This enormous amount of information is logically structured as a direct acyclic graph (DAG), as depicted in Figure 1, whose nodes are software artefacts such as (i) *file contents* (aka "blobs"); (ii) *directories*, which enclose files and other directories; (iii) *revisions* (aka "commits"), which represent the state of a project's root directory at a certain time; and (iv) *releases* (aka "tags"), which denote revisions that are also project milestones. Then, there are some additional nodes offering provenance information such as (v) *snapshots*, which capture the full state of a project at a certain time; and (vi) *origins*, which are the URLs at which a snapshot has been observed. Any software artifact gets added to the archive only if it does not already exist, thus files and directories get deduplicated even if they come from entirely different projects.

Unsurprisingly, the actual file contents require the most storage space, namely 99% of the whole 1.5 PiB size, despite being deduplicated and individually compressed with `gzip` (using `-9` flag, so with maximum compression). Each file content is associated with the corresponding node identifier in the DAG and accessed via a key-value object store [16]. The remaining storage space (about 30 TiB) is taken by the DAG, which is kept both in a relational database with roughly one table per node type [16] and in a derived compressed in-memory representation [8].

This sheer and ever-growing size of the available source code demands

many storage devices and operational costs.[6] This poses a serious obstacle to the SWH missions, as it significantly impacts the tasks of source code collection (e.g., from new and ever-changing projects due to increasing storage requirements), preservation (e.g., replication becomes more and more costly, also in terms of transmission time) and sharing (e.g., impacting researchers that wish to download a local copy for data/code analysis under limited storage and networking resources).

1.2. Objective

The objective of this study is to investigate *how to achieve efficient and effective lossless compressed storage of the largest part of the SWH Archive — i.e., the file contents — while still supporting streaming decompression and random access to the compressed files.*

This is an ambitious goal not only due to the scale of SWH but also because of its variegated nature, as it comprises source code files (along with their development history) written in hundreds of different programming languages by many authors and for diverse purposes. Moreover, the archive includes other artifacts that can be found in software projects, such as documentation, datasets, images, compiled programs, trained ML models, etc.

Currently, files in the SWH Archive are compressed individually with gzip. This saves approximately half of the space on average [16] but fails to leverage both the history of these files, which are often originated by commits that add and modify just a few lines of code and the fact that some code blocks are commonly recurring among different files, e.g., they are programming idioms or snippets that are copy-pasted from other projects and Q&A coding websites. Detecting and exploiting these inter-file redundancies is a huge compression opportunity that SWH is missing as of now.

In this paper, to get insights about intra- and inter-file similarities over the variegated composition of SWH, we consider five datasets of 200 GiB each: a random sample of the entire corpus, and four samples of source code files written in four different popular languages, namely C/C++, Java, JavaScript, and Python. We also consider a smaller 50-GiB dataset of files from the most-starred repositories for the C and Python languages. Upon

[6] Add to this the environmental impact, both due to the energy consumption and the failure-prone nature of these storage devices, which often become e-waste.

these datasets, we investigate not only the (lossless) compression efficacy of the studied solutions, useful for the preservation issues mentioned in Section 1.1, but we also evaluate the decompression speed of individual files, useful for the fruition of the SWH archive.

1.3. Contributions

We design, implement, and experiment with a prototype compressed storage subsystem for the Software Heritage Archive, which we make publicly available at `https://github.com/acubeLab/PPC_utils4BigData`. Our prototype allows compressing collections of several billion files (not necessarily source-code files) by fully exploiting the power of the Permute-Partition-Compress (PPC) paradigm introduced in [12, 13, 22].

The PPC paradigm consists of three main steps: (i) the input files are *permuted* to bring the similar ones close to each other; (ii) the permuted files are *partitioned* into blocks of proper size to facilitate efficient access; and (iii) the individual blocks are *compressed* by a general-purpose compressor.

The most crucial and intriguing step is undoubtedly the permuting one. For it, we can follow two main approaches: *context-based* and *content-based*. The *context-based* approaches take advantage of the context of the files as available in the SWH graph such as provenance information, commit history, filename, and authors. They are inherently powerful because they can allow identifying files that belong to the same software development history and group them together to achieve very effective compression. The *content-based* approaches, instead, are designed to be agnostic to the context of the file and detect similarities among them by using Machine Learning and Locality Sensitive Hashing [35] techniques applied to their content, i.e., raw bytes. As such, they can be applied to file collections in which context information is not available, or it is costly to be derived.

Alongside these approaches, we experiment with known compressors such as the Burrows-Wheeler Transform [14, 43], compressed full-text indexes [47, 46, 28, 21], and the git-pack compressor from the Git revision control system.

Our experiments will consider five datasets of 200 GiB each: a random sample of the entire corpus, and four samples of source code files written in four different popular languages, namely C/C++, Java, JavaScript, and Python. We also experiment on a smaller 50-GiB dataset of files from the most-starred repositories for the C and Python languages. Our experiments show that:

- Compressing files individually (e.g., with gzip and zstd), as currently done in the SWH Archive, achieves an unsatisfactory compression ratio of around 30%.

- The git-pack approach achieves a compression ratio of around 20%.

- Compressed full-text indexes and the Burrows-Wheeler Transform can achieve on par or better compression ratios (i.e., up to 23% the former and 12% the latter), but they seem impractical for this scale of data due to their slow compression and decompression speeds (both no faster than 4 MiB/s).

- Our proposed PPC framework achieves with the context-based permuters, simultaneously, the most effective compression ratio (on average 8%, if file access needs to be supported and 4% if not), and good compression speeds (up to about 54 MiB/s when using zstd-22 as final-stage compressor), decompression speeds (up to about 470 MiB/s with zstd-22) and file access speed (up to about 100 MiB/s with zstd-12). The content-based permuters are much slower in compression speed and slightly worse in compression ratio, which suggests that they might be more appropriate in scenarios where the context information is not available, such as in classic storage systems, or it is costly to be derived.

- Interestingly, the most compressible source code files are those written in Python (2.2%), followed by JavaScript (3.6%), C/C++ (3.7%), and Java (7.8%). As expected, the compression ratios slightly increase when enabling random access to individual files (because of compression in small blocks), but interestingly, the relative compressibility of the different languages changes. This behavior can be partly explained by the repetitiveness introduced by code-generation tools, different developer habits, and intrinsic properties of each language/framework, but further research is needed to untangle these findings.

- When compressing a random sample of files from the SWH Archive, the PPC framework achieves a worse compression ratio due to the presence of many incompressible files (e.g., multimedia files, or zip files), but it is still around 10% thanks to an effective grouping based either on context information or by file type inferred from the file content.

In conclusion, since source code files in C/C++, Python, Java and JavaScript collected by SWH as of 2024 amount to 78 TiB of data, and given the 4% of average compression ratio we achieved, we can extrapolate that our ongoing effort to scale our techniques to all open-source code ever written in these languages and available in the SWH archive could be compressed in about 3 TiB, hence just within the storage capacity of a modern laptop.

Paper outline. Section 2 provides some background on techniques we use in this paper and discusses relevant literature. Section 3 introduces our datasets, compression techniques, and the hardware/software environment. Section 4 presents the results of our experimental evaluation. Section 5 includes a discussion on the limitations of the proposed techniques, a comparison of the compressibility of different programming languages, and its implications for SWH. Section 6 concludes the paper by summarising the key takeaways and outlining future research directions.

2. Background and related work

In this section, we describe the main known approaches for compressing a collection of files (Section 2.1). We also review the main techniques for detecting similar files (Section 2.2) since they play an important role within the PPC framework.

2.1. Compressing a collection of files

The problem of compressing a collection of files has been addressed in the literature with reference to web pages [22], genome repositories [25, 57, 30], file and storage systems [31], and (single) source code repositories [56, 44, 33, 59, 7]. We now review the main techniques underlying these diverse applications and refer the interested reader to [22, 54] for further information.

A folklore approach to exploit inter-file redundancies is to concatenate the files in the collection and then run a general-purpose compressor (e.g., gzip). Most general-purpose compressors work by succinctly encoding repeated data fragments that occur within a fixed-size window sliding through the file to be compressed. While classic compressors like gzip and bzip have been designed to have a small memory footprint (less than 1MB) and thus detect only relatively close data repetitions, more recent and sophisticated

compressors — like Brotli[7] by Google, LZMA[8], LZFSE[9] by Apple, and zstd[10] by Facebook — can detect repetitions at much longer distances, up to hundreds or thousands of MiB. However, it is clear that even these tools cannot detect and exploit all the redundancy present in a very large document collection, such as the Software Heritage Archive, where repetitions can be much farther apart because of the collection size. As a consequence, three main ad-hoc approaches have been proposed so far in the literature to tackle this issue.

The first approach is to use a more powerful compressor that is not limited by a fixed window or block size. For example, using a disk-based construction of the Burrows-Wheeler Transform (shortly bwt, see [20]) is possible to detect repetitions at *unlimited* distances using a *limited* internal memory footprint. This approach is very time-consuming both in the compression and the decompression stages, but its compression effectiveness can be quite significant, as demonstrated on collections of web pages in [22] and in our following experimental results.

The second approach is based on the idea of *delta-compression* in which a file is encoded by copying portions of a given reference file. Intuitively, the more similar are the two files, the more compact their delta-encoding. In the context of large document collections, this technique leads to the problem of finding an appropriate cycle-free assignment of the files to be compressed with respect to one or more reference files. This can be modelled as the search for an optimum branching in a complete directed graph in which the nodes are the files to be compressed and the edge-weights are the benefit of compressing the target file (destination node of the edge) with respect to the source file [55] (source node of the edge). The time complexity of this solution grows quadratically in the number of files because of the number of edges in the graph. To alleviate this computational limitation and thus scale that approach to larger collections, several authors (see e.g., [18, 50]) proposed sophisticated graph-pruning heuristics. The most interesting one was introduced in [50] by proposing two distinct *clusterings* of the input files that are syntactically similar and thus good candidates for delta-compression. One is based on shingles and MinHash (see Section 2.2) to speed up edge-

[7] https://github.com/google/brotli
[8] https://www.7-zip.org/
[9] https://github.com/lzfse/lzfse
[10] http://facebook.github.io/zstd/

weight computations, but still takes quadratic time overall; the other exploits Locality-Sensitive Hashing (LSH) to detect more efficiently subsets of similar files, which are however not necessarily the "best" ones.

The third approach to the compression of a collection of files is *Permute-Partition-Compress* [12, 13] (shortly PPC), already mentioned in the previous pages. Instead of trying to capture repetitions that are far away in the file collection, PPC tries to preliminarily recognize and group together similar files so that a compressor with a small memory footprint can be applied to squeeze them. More specifically, this is done in three phases: (i) the input files are *permuted* in a way that the similar ones end up close in the new ordering; (ii) the permuted collection is then *partitioned* into blocks of proper size, (iii) the individual blocks are compressed via a known tool with a reasonably-sized memory footprint (such as zstd, Brotli, LZMA, etc.). This approach is designed to boost the ability of general-purpose compressors to find repetitions that are far away in the original (non permuted) collection, and thanks to the use of blocks, to allow reasonably fast random access to the original files. The key step among the three above is undoubtedly the first one. For document collections consisting of web pages, there are well-known re-ordering heuristics based on their URLs which are simple, time efficient, and achieve impressive compression ratios (up to 4%) for some well-known Web collections [22]. As far as we know, no study has investigated the application of the PPC paradigm to source code files, that is what we do in this paper by proposing permuting strategies specifically designed to identify near-duplicate source-code files (see Section 2.2).

A representative approach for compressing source code files is the one implemented in Git,[11] the distributed version control system originally authored by Linus Torvalds in 2005 for the development of the Linux kernel. Git stores the tracked file contents and other "objects types" (such as the commits or the state of a directory tree[12]) using three main ingredients.[13] The first ingredient is deduplication: when creating a new commit, Git computes the SHA-1 hash of the contents of each tracked file, compares it to the hashes of all the objects it already has and, if there is a match, no new content is stored for that file. The second ingredient is a general-purpose

[11] https://git-scm.com

[12] https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

[13] https://github.blog/2022-08-29-gits-database-internals-i-packed-object-store/

compressor, namely gzip, to squeeze the individual files. The third and most sophisticated ingredient, used when the repository grows large or is pushed to a remote repository, is git-pack. The inner working of git-pack, according to an informal description by Linus Torvalds[14] and our inspection of the Git v2.44 implementation (see also Appendix A in [7]), is similar to PPC because it starts by permuting the files to be compressed by sorting them by object type, and then by sorting the concatenation of path and filename reversed (so all files with the same extension will sort together). If there is still a tie between two or more files, they are sorted by size in decreasing order. Next, git-pack scans the reordered files using a sliding window and searching for pairs inside the window that achieve a good delta compression. This gradually builds a chain of files that are delta-compressed in sequence. git-pack behaviour is controlled by the parameter window representing the window size, expressed in number of files (by default 10, higher values give better but slower compression), and depth representing the maximum length of the chains, again expressed in number of files (by default 50, higher values give better compression but slower decompression since to reconstruct the content of a single file we need to look at the content of up to depth previous files).

We point out that git-pack was designed to run on a single Git repository and there is no easy way to run it on other collection of files or the SWH Archive, which combines files from millions of repositories possibly coming also from other version control systems, like Mercurial and Subversion.

2.2. Fingerprinting to detect similar files

To detect which files have similar content, and therefore can be effectively compressed together, we use techniques from the field of near-duplicate document detection. This problem first arose in Web search applications where near-duplicate web pages cause an increase in the index space usage, in the latency of serving query results, and possibly impact the user experience with repeated results. Manber [41] and Heintze [29] were among the first to propose algorithms for detecting near-duplicate documents based on the concept of *fingerprint*, i.e., a compact representation of a document such that similar documents have a similar compact representation. The successive groundbreaking results by Broder [11] and Charikar [15] provided a solid theoretical foundation for the fingerprint technique. The concept of fingerprint is re-

[14] https://git-scm.com/docs/pack-heuristics

lated to "locality-sensitive hashing" (LSH), which is a technique introduced in 1998 [35] and used today in a growing number of applications [36, 58].

Within the PPC paradigm, we tested the following four fingerprinting techniques that, we believe, are representative of the different approaches proposed in the literature. The first two fingerprinting techniques (ssdeep, TLSH) work directly on the raw input files by means of a sliding window, and thus they are called "bytewise" [10, 32]. The other two techniques (MinHash, SimHash) can take into account the internal structure of the input files using the concept of "token", and thus we will specialize them to source codes. More details follow.

ssdeep. [37] deploys a sliding window of 7 bytes that moves byte-by-byte through the input and computes a rolling hash for each window position. Whenever the rolling hash produces a specific output, ssdeep identifies a "trigger point" that indicates the ending of a block and the beginning of the next one. All blocks are hashed using a cryptographic hash function, and the 6 least significant bits of each hash are encoded using a Base64 character. The final digest is the concatenation of all characters generated through the blocks. The trigger point above is selected in such a way that the final fingerprint is up to 80 bytes long. We test this fingerprinting technique (see Section 3.2.4) because it is one of the first context-triggered fingerprints ever introduced.

TLSH. [48] deploys a sliding window of 5 bytes, from which it selects six trigrams (triplets of window characters). Each trigram is converted using a Pearson hash to a 7-bit integer, which is used to update a vector of 128 counters. When the scanning is complete, a 256-bit fingerprint is obtained by encoding each counter with two bits, based on its quartile (the 25% smaller counters are encoded with 00, those in the 25% to 50% quartile with 01, and so on). The original TLSH scheme also defines a 24 bit header, but we do not use it since it is not related to file similarity. We test the TLSH fingerprint because experiments showed that it is more robust to random changes and adversarial manipulations than ssdeep and its variants [49].

SimHash. [15] estimates the *similarity* of two documents in two main steps. First, it identifies a set of "tokens" in the input documents, and maps each of them into a random vector drawn from $\{-1, 1\}^b$, where b is a properly chosen constant. This token-based mapping is the same for all documents.

Then, it obtains a b-dimensional vector for each document by adding the b-dimensional vectors of all of its tokens, possibly *weighting* each of them with a proper score (e.g., the TF-IDF in case of words [42]). The final fingerprint is created by setting every positive entry to 1 and every non-positive entry to 0, thus resulting in a random mapping of each document into the space $\{0,1\}^b$. Charikar [15] proved that the cosine similarity of two documents, in the classic IR-sense [42], is proportional to the number of bits in which the two fingerprints agree. In our experiments we did not use any weighting and set $b = 256$ so the final SimHash fingerprint is 256-bit long.

MinHash. [11] (see also the variant of [23]) computes the fingerprint of an input document d consisting of n tokens as follows. It slides a window of k tokens over d and, for each of them computes a 64-bit Karp-Rabin's rolling hash. This generates a set $S(d)$ of $n - k + 1$ integers called *shingles*. The *similarity* (a.k.a. *resemblance*) of two documents d and d' can be evaluated by the Jaccard's coefficient between the sets $S(d)$ and $S(d')$ defined as $J(d, d') = \frac{|S(d) \cap S(d')|}{|S(d) \cup S(d')|}$. To speed up the computation of $J(d, d')$, Broder [11] suggested to use m different fingerprinting functions f_i and proposed to compute, for each i, the minimum of each set of shingle fingerprints $S_i(d)$ according to f_i. These m minima form an m-dimensional vector that represents the final fingerprint of d, therefore called *MinHash*. Note that multiple occurrences of the same shingle will have the same effect on the min-values as a single occurrence, i.e., the multiplicity of shingles is ignored. Broder showed that the number of element-wise equal entries in the MinHash vectors of two documents d and d', divided by m, is an unbiased estimator of $J(d, d')$, and that m controls the robustness of the estimate. In our experiments, we set $m = 256$ so the final MinHash fingerprint is a 256-dimensional vector of 64-bit minima.

As a final note we observe that, although fingerprint techniques have some mathematical guarantees on their *average* performance, all of them may incur false positives (dissimilar document pairs returned as near-duplicates) as well as false negatives (near-duplicate document pairs not returned as near-duplicates). In addition, because of the size of our collections, computing the similarity between all pairs of files would be too expensive; for this reason our use of the fingerprints in the PPC framework will be different from their proposed use in the above-mentioned literature.

2.3. Large-scale analysis of software development artefacts

Various platforms and approaches deal with large-scale storage and analysis of software development artifacts, for software engineering research purposes. In support of their goals, they use compression and/or deduplication techniques. We briefly recall the most relevant work in this space, below.

LISA. [2] is a framework designed to minimize artifact redundancy in the analysis of source code stored in version control systems (VCS). Unlike our use case, LISA operates at a finer level of detail, down to the abstract syntax tree (AST) nodes. This allows for fine-grained deduplication, but it comes with the drawback of needing a proper parser. This parser is not always available and may fail on syntactically incorrect files, which might still be of interest for analysis—and need to be efficiently stored anyway.

Boa. [19] pioneered the idea of a shared infrastructure that hosts both data and compute resources for large-scale analyses of source code artifacts. Our measure of "large-scale" differs significantly, as the scale of the Software Heritage Archive is hundreds of times larger than the largest GitHub hosted on Boa, by several metrics, including projects, files, and commits. Similarly to LISA, Boa also parses source code files to AST but does not apply any compression technique.

World of Code (WoC). [39] has also created a mutualized infrastructure for large-scale VCS analyses. Unlike the Software Heritage case study, which spans GitLab and major package repositories, WoC is limited to GitHub. However, the overall size of the data stored by WoC for analysis purposes is in the same ballpark of the size of the SWH Archive. In terms of storage, WoC deduplicates all retrieved blobs, but does not apply any compression technique. It can benefit from the compression approaches that we develop in this work, specifically the ones designed for the File-Access scenario.

SWH-graph. [8] is a technical framework that builds upon WebGraph [9, 24] to compress the SWH graph in order to host and process it in main memory on a single server. swh-graph only compressed the graph structure of the SWH graph, leaving out of scope the compression of all other information stored in the SWH Archive, including metadata (e.g., filenames) and crucially file contents—we deal only with the latter in the present work.

3. Experimental setup

In this section, we detail our datasets (Section 3.1), compression techniques (Section 3.2), and the hardware/software environment used to run our experiments (Section 3.3).

3.1. Datasets

We prepared datasets of different sizes consisting mainly of source code files written in different programming languages but also some non-code files that are commonly found in software repositories. All datasets are extracted from the actual Software Heritage Archive and are available at the GitHub repository of our software library[15].

A dataset in our context consists of a set of *deduplicated* files (i.e., raw byte sequences that are different from each other in at least one byte) plus some *metadata* for each file consisting of an identifier derived from hashing its content, the file size, and a path that includes the filename. The distinction between actual file content and metadata will be useful in our experiments to distinguish compression methods that exploit just the first information (called *content-based* methods) or both of them (called *context-based* methods). In the case that the same file content appears under different paths, we use as its metadata the most popular path, as in [38]. Table 1 summarizes the characteristics of our datasets, which are commented on in the text below.[16]

Popular GitHub repos. We generated a 50-GiB dataset containing different versions of files present in the 143 most-starred repositories on GitHub for the C and Python languages as of October 2022. This includes popular repositories like redis, ngnix, zstd, scikit-learn, bert, and keras.[17] Roughly half of the dataset is taken by files from repositories written in C and the

[15] https://github.com/acubeLab/PPC_utils4BigData

[16] To measure file sizes we use kibibytes (KiB), mebibytes (MiB), gibibytes (GiB), and tebibytes (TiB), which represent 2^{10}, 2^{20}, 2^{30}, and 2^{40} bytes, respectively.

[17] See https://github.com/{redis/redis, nginx/nginx, facebook/zstd, scikit-learn/scikit-learn, google-research/bert, keras-team/keras}. To avoid skewing this bounded-size dataset towards a single, disproportionately large repository, we excluded just the Linux kernel repository from the popular repositories list, since it takes more than 70 GiB alone, and the other repositories are just a few GiB each.

Datasets	Size (GiB)	Num. files	File size (KiB)	
			Mean	Median
Popular GitHub repos (C and Python)	50.4	1 858 580	28.4	5.9
Python code	200.0	9 640 731	21.7	7.2
C/C++ code	200.0	6 437 613	32.5	9.0
JavaScript code	200.0	3 464 374	60.5	4.6
Java code	200.0	26 373 974	7.9	2.4
Random content	200.0	2 335 158	89.8	4.5

Table 1: Characteristics of our datasets.

other half in Python. As shown in Table 1, the average and median file sizes are 28.4 KiB and 5.9 KiB, respectively.

The purpose of this dataset is to fairly compare git-pack against the other compression techniques. Indeed, git-pack can only be run on individual Git repositories, and thus it cannot be applied to datasets of files gathered from the whole SWH Archive (such as the ones described in the next paragraphs) since these cannot be recombined into Git repositories due to the nature of SWH [1].

More precisely, SWH does not exclusively store Git repositories, but rather a diverse range of software origins, as described in Section 1.1. These origins represent different sources where software artifacts can be found, including repositories of various version control systems (e.g., Git, Mercurial, SVN, Bazaar), as well as source packages for various package managers and standalone tarball archives. As a result, the concept of a "repository" in SWH is broader and more abstract than that of Git-centric platforms, like GitHub. Furthermore, while one can reconstruct repositories from SWH starting from any given origin, this process is costly due to thorough deduplication (e.g., reconstructing the Git repository of the Linux kernel requires collecting tens of millions of objects). SWH deduplicates files across all archived origins, as the same files can be shared by multiple origins. This raises the question of which origin should take precedence when reassembling repositories. Reconstructing repositories by assigning files to them would require a high number of random accesses, significantly increasing costs and making the process inefficient at scale.

The experimental results on this dataset are in Section 4.1.1.

Datasets from the SWH Archive. We generated the following five datasets by randomly sampling files with specific extensions from the whole SWH Archive:[18]

- **Python code**, consisting of files with extension `.py` and `.pyi`.
- **JavaScript code**, consisting of files with extension `.js`.
- **C/C++ code**, consisting of files with extensions `.c`, `.C`, `.cc`, `.cpp`, `.CPP`, `.c++`, `.cp`, `.cxx`, `.h`, `.hpp`, `.hpp`, and `.HPP`.
- **Java code**, consisting of files with extension `.java`.
- **Random content**, consisting of files with any extension and thus may include images, videos, binary files, and PDF documents.

The purpose of the Random content dataset is to evaluate the experimented techniques in the presence of non-source-code files. We anticipate that the results on this dataset, provided in Section 4.1.3, suggest that grouping files of the same type together is crucial to achieving effective compression.

As shown in Table 1, the average file size of the experimented datasets ranges from 7.9 KiB for Java code to 60.5 KiB for JavaScript code. The median file size ranges from 2.4 KiB for Java to 9.0 KiB for C/C++ code.

3.2. Compression techniques

We apply different compression techniques on the datasets of Section 3.1 and evaluate them in terms of compression and decompression speed, expressed in mebibytes per second (MiB/s), and in terms of compression ratio, i.e., the ratio between the size of the compressed dataset and the size of the original uncompressed dataset, expressed as a percentage.

As anticipated in the introduction, we distinguish between a *Backup scenario* and a *File-Access scenario*. In the former, a dataset is stored in compressed form, and we only need to support streaming access to the whole compressed collection. This is relevant, e.g., for backup purposes or to offer fast downloads of the dataset. In the latter, in addition to providing

[18]The full list of files and their metadata in the snapshot of the SWH Archive we used is a massive 750-GiB csv file. To deal with it, sample files, and compute statistics, we leveraged the Postgres DBMS. Then, the contents of the sampled files have been downloaded from AWS S3.

Technique	SWH-applicable	File-Access	Context-based	Section
single_compress + *	✓	✓	✗	§3.2.1
random_order + *	✓	✓	✗	
bwt_rle	✓	✗	✗	§3.2.2
csa_sada	✓	✓	✗	
csa_wt	✓	✓	✗	
git-pack	✗	✓	✓	§3.2.3
tlsh + *	✓	✓	✗	§3.2.4
ssdeep + *	✓	✓	✗	
simhash + *	✓	✓	✗	
minhash_graph + *	✓	✓	✗	
type_minhash_graph + *	✓	✓	✗	
filename + *	✓	✓	✓	
path + *	✓	✓	✓	

Table 2: Compression techniques experimented in this paper; the symbol "+ *" denotes the additional use of a general-purpose compression algorithm such as gzip or zstd at the end of the PPC pipeline. The first group is formed by baseline techniques, the second group includes the current theoretical state of the art for compressing file collections, and the last group is a set of PPC-based techniques we specifically designed for the SWH archive.

space-efficient storage, we also need to support fast access to individual files. This is relevant, e.g., for browsing the contents of the SWH Archive through its web interface or API,[19] or to perform data/code analyses locally on a researcher's computer.

Given our literature review of Section 2, we chose to experiment with the compression techniques shown in Table 2. All of them except git-pack apply to the SWH Archive, most of them are appropriate also for the File-Access scenario, and only some of them are context-based in that they use not only the file content but also the metadata offered by the SWH Archive.

3.2.1. Baselines

The first two techniques we consider will be our baselines. The algorithm single_compress compresses the files individually with a general-purpose compressor, such as gzip or zstd. This is the solution SWH currently employs.

[19] See https://archive.softwareheritage.org and https://archive.softwareheritage.org/api/, respectively.

The algorithm random_order instead concatenates all the files from a dataset in random order and applies a general-purpose compressor, such as gzip or zstd. This solution can exploit some inter-file redundancies, as long as these redundancies are within the fixed-size window of the general-purpose compressor.

3.2.2. Techniques based on the Burrows-Wheeler transform

As we pointed out in Section 2.1, by applying the Burrows-Wheeler Transform (bwt) to the concatenation of the input files, we can take advantage of repetitions in the input no matter how far they are in the collection. We have tested this approach using the `bwtdisk` library[20], which computes the bwt of large files in external memory and subsequently compresses it using run-length encoding followed by arithmetic coding [20]. We call this technique bwt_rle. As already observed in Section 2.1, this solution can achieve quite significant compression ratios at the cost of very slow compression and decompression speeds. Furthermore, it does not support the File-Access scenario as it requires decompressing the full dataset to access a single file.

The bwt however is also at the core of the implementation of compressed full-text indexes [47, 46, 28, 21], which keep data in a compressed form that also supports pattern-matching queries and efficient extraction of arbitrary subsequences. This last feature makes them suitable for the File-Access scenario. We tested the following two full-text indexes offered by the *sdsl* library [27][21] applying them on the concatenation of all files from a dataset: csa_wt and csa_sada. The former represents the bwt with a Huffman-shaped wavelet tree with RRR-encoded bitvectors [53] with block size b. The larger b, the better the compression ratio because the RRR-encoding must keep track of longer blocks. At the same time the larger b, the worse the compression and decompression speeds because operations inside longer blocks are more expensive. We set and tested $b = 63$ and $b = 127$ bits because they provided the best Pareto performance in our experiments. csa_sada does not explicitly represent the bwt but is based on compressing the Ψ array using Elias δ-codes with sample rate s. The larger s is, the better the compression ratio because fewer samples are stored but, on the other hand, the slower are the compression and decompression speeds because fewer samples means more

[20] https://github.com/acubeLab/bwtdisk_2010
[21] https://github.com/simongog/sdsl-lite/

integers must be decoded. We set and tested $s = 128$ and $s = 256$ bits because they provided the best Pareto performance in our experiments.

3.2.3. The git-pack *technique*

As the representative compression method for (individual) source code repositories, we selected the git-pack tool discussed in Section 2.1 because of its widespread use. Unfortunately, git-pack must be run on individual Git repositories, so we have applied it only to the "Popular GitHub repos" dataset.

To test this technique, we run `git pack-objects`[22] on each repository obtaining a single `.pack` file containing all the repository's files starting from a fixed commit. We measure the compression speed by taking the time to build such `.pack` file. To measure the decompression speed, we extract back all the individual files by passing the `.pack` file to `git unpack-objects`[23]. Note that `git unpack-objects` extracts the objects from the `.pack` file creating the so-called "loose objects", i.e., one gzip-compressed file per object. To get back the uncompressed content, a final decompression phase is performed by the zlib [24] library on each file. The tool `git cat-file` can be used for this last phase, but we experimentally found out it can be up to twice as slow as simply decompressing the loose objects with zlib library. Finally, to measure the compression ratio, we divide the sum of the sizes of the per-repository pack files by the size of the whole uncompressed dataset.

3.2.4. PPC-based techniques

In this section, we describe a set of techniques that belong to the Permute-Partition-Compress (PPC) paradigm, described in Section 2.1. They permute the input files according to one of the strategies detailed below. Then, they use the tar tool[25] to concatenate the permuted input files into a single tar file (Backup scenario) or into multiple tar files of size B called "blocks" (File-Access scenario).

The choice of using tar archives for concatenating files is motivated by their widespread adoption and robust performance. For example, the tar

[22] https://www.git-scm.com/docs/git-pack-objects, version `2.43.0`
[23] https://git-scm.com/docs/git-unpack-objects
[24] https://www.zlib.net/
[25] https://www.gnu.org/software/tar/

format is the foundation of the WebDataset format [26], which is designed for efficient handling of large-scale datasets in deep learning applications. WebDataset is supported by the Huggingface ecosystem [27], making it an ideal choice for distributing and processing datasets in modern machine learning workflows. Leveraging tar archives ensures that our method benefits from a well-established, efficient, and versatile file aggregation approach.

To avoid splitting files larger than B bytes across blocks, we allow the last file in a block to exceed B. The mapping from the file identifier to the block containing it is implemented via a key-value store using RocksDB [17]. In the File-Access scenario, we experiment with different values of B: a higher value leads to better compression because more inter-file redundancies can be exploited, but also slower access to a single file because the full block containing the file must be decompressed.

The individual tar files (one in the Backup scenario or several in the File-Access scenario) are finally compressed with a general-purpose compression tool. We tested several compressors including lzma, gzip, brotli, zstd, xz, and lz4. We report only the results for zstd because of its generally good performance and its many options to trade off compression ratio with (de)compression time. On our datasets, by setting the highest compression level and memory usage, zstd achieves a compression ratio similar to (and even better than) the most effective compressors like xz and brotli. At the same time, decreasing the compression level and memory usage, zstd can be extremely fast in decompression speed (more than 400 MiB/s, similar to lz4). Precisely, we report the results for three compression levels: -3 (very fast compression speed, the default one), -12 (an effective compromise between compression ratio and compression speed), -22 –ultra -M1024MB –long=30 (the best compression ratio, but very slow in compression speed). For brevity, we call these configuration settings zstd-3, zstd-12, and zstd-22, respectively.

To compute the compression speed, we consider the whole pipeline of permuting the input files and creating the compressed tar file(s). For the decompression speed, in the Backup scenario we measure the time to decompress the whole compressed tar file, while in the File-Access scenario we measure the time to decompress a random subset of files (10% of the total number of files in a dataset), which includes the time to query RocksDB for

[26] https://github.com/webdataset/webdataset
[27] https://huggingface.co/docs/hub/datasets-webdataset

finding the block containing the desired input file, and decompressing it until the file is found.

Permutation algorithms. These following permutation algorithms exploit the fingerprinting techniques described in Section 2.2 to place similar files close to each other:

- **tlsh**: permute files according to the lexicographic order of their TLSH fingerprints [48], each of size 256 bits.
- **ssdeep**: permute files according to the lexicographic order of their ssdeep fingerprints [37], each of size up to 80 bytes.
- **simhash**: permute files according to the lexicographic order of their 256-bit SimHash fingerprints [15], each one computed by tokenising the file into lines (i.e., splitting it according to the `\n` character) and filtering out the lines with fewer than 10 characters. The rationale is that short lines of code often represent trivial or insignificant segments that may not contribute significantly to the overall structure of the code.
- **minhash_graph**: tokenize the input file into lines as in SimHash (above) and for each file i compute the MinHash fingerprint f_i consisting of 256 64-bit minima. Split each f_i into four 4096-bit chunks $f_i^{(c)}$ for $c = 1, \ldots, 4$ and create a graph whose vertices corresponds to the files and there is an edge between files i and j iff $f_i^{(c)} = f_j^{(c)}$ for some $c = 1, \ldots, 4$. Finally, find the connected components in this graph and, for each connected component, permute the corresponding files according to their sizes in descending order.[28] We tested different fingerprint/chunk lengths and the above scheme was the one giving the better compression. Since some connected components might be too large, and thus similar files would end up far away from each other, we permute the ones with overall size greater than 32 MiB and more than 3 files via the tlsh technique described above.

We also test a permutation algorithm designed for collections containing different file types (e.g., written in different programming languages):

[28] The same technique has been used by Hugging Face to deduplicate, but not compress, source code datasets. See https://huggingface.co/blog/dedup.

- **type_minhash_graph**: group files according to a type inferred from their contents via Google's Magika deep learning model,[29] and then apply the minhash_graph approach over each group if this consists of more than 3 files whose total size is larger than 32 MiB; if not, the files in a group are sorted just by their size in descending order.

All the above algorithms are *content-based* in that the final permutation is based only on the actual content of the input files. We also consider a class of algorithms that are *context-based* in that they compute a permutation without looking at the file content, but only at the metadata, such as filename, path, size, etc. Assuming the appropriate metadata is available, context-based methods are usually faster since they do not have to scan the file contents. However, it is not a priori obvious which of the two approaches is superior in terms of compression when used in combination with the PPC paradigm. To investigate this issue, we consider the following two context-based techniques, inspired by the git-pack algorithm described in Section 2.1:

- **filename**: permute files according to the lexicographic order of their reversed filenames, and if there is a tie, put the largest file first;
- **path**: as before but, instead of using only the filename, also use the name of the directory containing it.

We test both filename and path because we wish to evaluate how much adding one directory level impacts the compression performance.[30] The rationale is that we expect two files with the same name (e.g., `parser.c`) to be more similar if they belong to the same directory (say `src/` rather than `test/`).

3.3. Hardware setup

We test the techniques of Section 3.2 on a machine equipped with a 2.30 GHz Intel Xeon Platinum 8260M CPU and 384 GiB of RAM, running Ubuntu 20.04.3. The files are stored in an XFS filesystem mounted on SSD SAS disks RAID0. The compressed archives are stored an XFS filesystem mounted on traditional rotating disks RAID0. All the disks are attached to the same PERC H740P Adapter.

[29] https://google.github.io/magika/

[30] We only consider the parent directory because on SWH using full paths is very time-consuming [38] and does not seem to improve the clustering process.

3.4. Software setup: the compression library

All our compression techniques are part of a software library we publicly release at https://github.com/acubeLab/PPC_utils4BigData. The library is written mostly in Python and includes the list of the files composing our datasets, which can be downloaded from the SWH Archive. Although the library is specialized to the task of compressing a collection of source code files, it can be easily customized to other kinds of collections (e.g., by using a custom tokeniser) or adapted to use other general-purpose compressors.

After downloading the datasets and setting up the environment with the Python script available in the library, it is easy to benchmark the various permuters using different compressors on a properly specified list of files. More precisely, the script `bench_PPC.py` takes as input a list of files (contained in a csv file), permutes them according to one or more techniques (`-p` option), concatenates them, and compresses the resulting file as a unique blob or splits it into blocks to compress individually (`-b` option). This choice depends on the fact that one wants to test the Backup scenario or the File-Access scenario. The choice of the compressor(s) is delegated to option `-c`.

The script measures and displays several numbers: such as the compression ratio, the compression speed, the decompression speed, etc. For example, the command:

```
$./bench_PPC.py \
    C_files_metadata.csv -c zstd -p random \
    simhashsort filename
```

permutes the files according to the PPC pipeline random_order, simhash, and filename; then concatenates and compresses them with `zstd`, and outputs the results (in csv format) as shown in Table 3.

Our library is sufficiently flexible that one can customize also its tokenizer to implement different locality-sensitive hashing (LSH) techniques. In fact, LSH implementations (like Simhash, Minhash, etc...) view each file as a sequence of tokens. A token can be a sequence of bytes, words, lines, or arbitrarily long substrings. In this paper, we deal with source code files and thus we decided to use as "tokens" the single lines of the input files (i.e., delimited by the `\n` char). We also tested tokens as groups of consecutive lines (obtained from a sliding window of a certain width), but we experimentally evaluated that single lines perform better. Since we wanted to be robust with respect to tiny and irrelevant changes in the input files, we removed leading

Dataset	Permuter	Comp. ratio (%)	Comp. speed (MiB/s)	Decomp. speed (MiB/s)
C_files_metadata	random_order+zstd	21.41	209.32	300.05
C_files_metadata	simhash+zstd	13.57	16.42	361.74
C_files_metadata	filename+zstd	7.14	493.46	455.57

Table 3: The table reports the output of the `bench_PPC.py` script (part of the software library we publicly release with this paper). It shows the performance of the PPC framework on a 200GiB dataset made of source code files written in C/C++, permuting the files with random_order, simhash, and filename using as final compressor `zstd`.

and trailing tabs and white spaces and dropped too short lines (<10 chars). The code we used to tokenize the files is the following:

```
def get_tokens(file_content, len_limit=10):
    # get a list of lines from the content of the file
    tokens = file_content.split('\n')
    # remove tokens with less than 10 chars
    # and delete leading and trailing tabs and whitespaces
    tokens = [x.strip() for x in tokens if len(x) > len_limit]
    return tokens
```

As a technical note, our library uses just one thread on the relatively small "Popular GitHub repos" dataset while it processes the other (larger) datasets via more threads to speed up the computation. In particular, it extracts the fingerprints using 16 threads and then compresses 16 blocks in parallel in the File-Access scenario, while it deploys the parallel compression capabilities of zstd, whose option `-T#` spawns `#` compression threads, in the Backup scenario. While we rely on parallelization to reduce execution times, a thorough analysis of the scalability of our approach is beyond the scope of this paper.

4. Experimental results

We now describe the results of our experiments on compressing our file collections in both the Backup scenario (Section 4.1), where we only support full decompression of the whole compressed collection, and in the File-Access scenario (Section 4.2), where we also support fast access to the individually compressed files.

Technique	**Compression ratio** (%)	**Compression speed** (MiB/s)	**Decompression speed** (MiB/s)
single_compress + gzip	32.4	11.4	65.5
single_compress + zstd-3	33.4	79.5	102.9
single_compress + zstd-22	29.9	16.7	117.7
random_order + zstd-22	18.2	2.4	304.6
git-pack, $w = 10, d = 50$	20.9	49.2	54.4
git-pack, $w = 20, d = 100$	20.2	36.7	56.9
git-pack, $w = 40, d = 200$	19.9	28.3	66.9
bwt_rle	12.0	<1	<1
csa_sada, $s = 128$	34.8	3.6	3.7
csa_sada, $s = 256$	31.3	2.6	2.9
csa_wt, $b = 63$	25.8	2.2	<1
csa_wt, $b = 127$	23.4	1.8	<1
tlsh + zstd-22	10.5	12.4	329.6
ssdeep + zstd-22	10.7	13.0	331.7
simhash + zstd-22	10.8	11.0	330.7
minhash_graph + zstd-22	10.3	3.3	332.1
type_minhash_graph + zstd-22	9.5	1.7	327.4
filename + zstd-22	10.1	32.9	379.7
path + zstd-3	15.8	465.8	326.3
path + zstd-22	10.1	32.7	381.0

Table 4: The table reports the experimental results on the 50-GiB "Popular GitHub repos" dataset of all techniques investigated in this paper. We refer the reader to Section 3.2 for the description and parameter settings of these techniques.

4.1. Results for the Backup scenario

4.1.1. Results for the Popular GitHub repositories

As a first experiment, we compare all compression techniques on the 50-GiB "Popular GitHub repos" dataset, which is variegated and of relatively small size, thus allowing us to investigate the performance of all techniques in a reasonable time.

As the first two rows of Table 4 show, compressing the files individually with gzip (default compression level) achieves a compression ratio of 32.4% and a compression speed of 11.4 MiB/s, while the more recent and faster zstd-3 achieves a compression ratio of 33.4% and a compression speed of 79.5 MiB/s. The memory-hungry zstd-22 (third row) achieves a slightly improved compression ratio of 29.9% and a compression speed of 16.7 MiB/s.

This means that we can cut the storage space at more than one-third by just exploiting the redundancy within each file.

These techniques, however, do not exploit the inter-file redundancy among the source code files in the dataset. In fact, if we compress the concatenation of the files with random_order + zstd-22, we immediately obtain an improved compression ratio of 18.2% (i.e., –10% in absolute terms) and a faster decompression speed, at the cost of a significantly slower compression speed. This slowdown can be attributed to the expensive processing required by zstd-22 to identify and encode long repeated subsequences. On the other hand, compared to single_compress + zstd-22, the decompression speed is 2.6× faster due to the fact that the decompression does not need to restart at each individually-compressed file and thus it can proceed in a streaming-fashion.

Despite its 1024-MB window, zstd-22 cannot exploit redundancies among files that are far apart in the random concatenation order. git-pack tries to capture this redundancy by compressing together potentially-similar files inside the same repository. However, it achieves a relatively modest compression ratio of 20.9% with an improved (de)compression speed.[31] Increasing the value of the parameters `window` (from 10 to 20) and `depth` (from 50 to 100) allows git-pack to create longer and more compressed chains of deltified files (see Section 2.1), but this improves the compression ratio by just a relative 3.1% (going from 20.96% to 20.29%). Increasing the value of the parameters even more (setting `window` to 40 and `depth` to 200), the compression ratio becomes 19.90%, but the compression speed almost halves (it goes from 49.2 MiB/s to 28.3 MiB/s) and the decompression speed slightly decreases. Overall, git-pack gets almost the same compression ratio of random_order+zstd-22 with a 12× faster compression speed but at the cost of a 4.5× slower decompression speed. Summing up git-pack: (i) is not able to leverage similarities between files from different repositories, and (ii) its combined use of delta encoding and gzip is not as efficient as zstd-22.

The approaches we just discussed can exploit some inter-file redundancy but they fail when the files are far apart (i.e., random_order) or in distinct repositories (i.e., git-pack). Such distant redundancies are the target of the techniques based on the bwt (see Section 3.2.2), whose experimental results

[31] We observe that git-pack cannot deduplicate files across the different repositories in this dataset. However, since the non-deduplicated dataset takes 1.2 GiB more than the 50.4 GiB of the deduplicated one, the impact of the duplicates on the compression ratio will be at most 2.4%, which does not change our conclusions.

are shown in the third row-group of Table 4. We immediately notice that bwt_rle is very slow in compression and decompression (less than 1 MiB/s) and thus unusable in practice. Compared to bwt_rle, the performance of full-text indexes is worse in terms of compression ratio (i.e., more than double in percentage), but it is more than twice better in compression and decompression speed, even if it is still unacceptably slow. In detail, the compression ratio goes from 34.8% of csa_sada with sample rate $s = 128$ to 31.3% with sample rate $s = 256$; instead, csa_wt achieves 25.8% compression ratio with block size $s = 63$, and 23.4% with block size $b = 127$. The compression and decompression speeds are still on the order of a few MiB/s due to the random accesses made during the construction of the bwt or its inversion. We can thus conclude that, although bwt-based approaches are reasonably effective in compression ratio, their very-slow compression and decompression speeds makes them unsuitable to scale on bigger datasets.

We are now left with commenting on the PPC-based techniques whose performance, detailed in the last row-group of Table 4, is very promising in all three tested performance measures: compression ratio, and (de)compression speeds. In particular, the content-based approaches based on the sorting of the fingerprints (namely, tlsh, ssdeep, and simhash) reach the significant compression ratio of about 10%, a moderately fast compression speed (between 11 and 13 MiB/s), and a very fast decompression speed of more than 300 MiB/s. The approaches based on graphs (such as minhash_graph and type_minhash_graph) offer a slightly improved compression ratio and a comparable decompression speed, but a very slow compression speed: hence resulting not competitive. The reason resides in the fact that minhash_graph requires reading all the files from the filesystem (randomly accessing the disk), computing the MinHash fingerprint, constructing a graph, computing its connected components, and finally further permuting the bigger connected components with tlsh; while type_minhash_graph also performs a grouping pre-step using the file type inferred with a deep learning model.

The last group of PPC-based techniques are the context-based approaches filename and path that use the (reversed) file/path name as the sorting key of the permuting step. Despite their simplicity, these algorithms are better on all three performance measures (with the only exception of type_minhash_graphin compression speed). Table 4 also reports the performance of path + zstd-3: the reduced window size worsens the compression ratio to 15.8% and the decompression speed to 326 MiB/s (-55 MiB/s), but it significantly improves the compression speed (from 32.7 MiB/s to 465.8 MiB/s). Not surprisingly,

we see that among the PPC-based techniques, the content-based algorithms (tlsh, ssdeep, simhash, minhash_graph, and type_minhash_graph) are about 4× slower in compression speed than the context-based algorithms (filename and path), since the former need to scan the file content and compute the fingerprints while the latter permute the files by looking only at their filename/-path. Overall, we can state that the PPC-based approaches outperform all the other techniques in terms of compression ratio (which turns out to be half of the others) and decompression speed (which turns out to be up to 5× faster), thus making them a promising solution for the compressed storage of the SWH Archive.

Takehome message. Using a dataset of popular GitHub repositories, we found out that compressing the files individually achieves an unsatisfactory compression ratio, that git-pack does not improve much on this, and that bwt-based tools are impractical due to their slow (de)compression speed. Instead, the PPC framework with the context-based permuters combines the simplicity of its design with the most effective compression ratio (≈10%) at interesting compression (≈13 MiB/s) and decompression (≈330 MiB/s) speeds. If we adopt the content-based permuters, we reduce the compression speed by 3× without significantly changing the compression ratio and decompression speed, which seems to suggest they might be more appropriate in scenarios where the context information is not available. As a final note, we remark that changing the final-stage compressor (e.g., from zstd-22 to zstd-3) slightly reduces the compression ratio but significantly benefits the compression speed, at comparable decompression speed.

4.1.2. Results for the SWH source code datasets

The four SWH datasets we experiment with in this section consist of 200 GiB of source code files in four languages: Python, JavaScript, Java, and C/C++. We remind that git-pack cannot be applied here because it is designed to work on a set of files organised as a single repository. We also do not test gzip, whose compression ratio is never smaller than 16%, and the bwt-based techniques, since their compression speed makes them impractical. Moreover, we do not test type_minhash_graph because its ML-based approach to detect the file type is useless here since the datasets are homogeneous (as they contain files in the same language). For uniformity, the last stage of the compression in all of our PPC-based tools is always implemented through zstd-22.

We show the results of this section in Figure 2. Plots on the same row refer to the same dataset. The first column shows the compression ratio vs compression speed, while the second column shows the compression ratio vs decompression speed. The plots in the same row share the same vertical axis to better compare their (de)compression speeds. We also highlight the best compression ratio achieved by our tools with a vertical line, and with an arrow, we indicate the Pareto front of the best approaches. We use random_order as our baseline plotting its results in blue.

All the other algorithms shown in Figure 2 follow the PPC-based approach. The context-based techniques filename and path are plotted in red and achieve the best performance overall. Their compression ratio is indeed a remarkable 2.21% on Python, 3.64% on JavaScript, 7.78% on Java, and 3.73% on C/C++. They are also the best in (de)compression speeds (up to 500 MiB/s, with a potential writing speed of our disk of 1 GiB/s, see Section 3.3). As expected, the additional usage of the directory name by the path algorithm slightly improves the compression ratio (up to 0.4% on average), so in the following we experiment only with path.

The content-based techniques are plotted in green. The three algorithms that permute the input files according to the lexicographic order of the fingerprints (namely, simhash, ssdeep, and tlsh) almost halve, on average, the compression ratio obtained by random_order. Among these algorithms, tlsh consistently outperforms the others in terms of compression ratio, particularly excelling on the JavaScript dataset. However, sorting fingerprints has the disadvantage that similar files are placed far apart in the sorted order if the corresponding fingerprints differ only in the very first bits. minhash_graph avoids this pitfall with a graph-based approach, and indeed it achieves the third-best compression ratio after the context-based algorithms path and filename in three out of four datasets, but at the cost of a much slower compression speed.

Takehome message. Our experiments show that source code files from the SWH Archive can be compressed significantly by PPC-based techniques. The most compressible ones are those written in Python (2.2%), followed by JavaScript (3.6%), C/C++ (3.7%), and Java (7.8%). These different compression ratios are likely due to multifaceted reasons: such as the number of files (for example Java has 9× more files than JavaScript see Table 1), repetitiveness introduced by automatic tools and developers, language history, and intrinsic properties of each language/framework. We believe that a deeper insight on

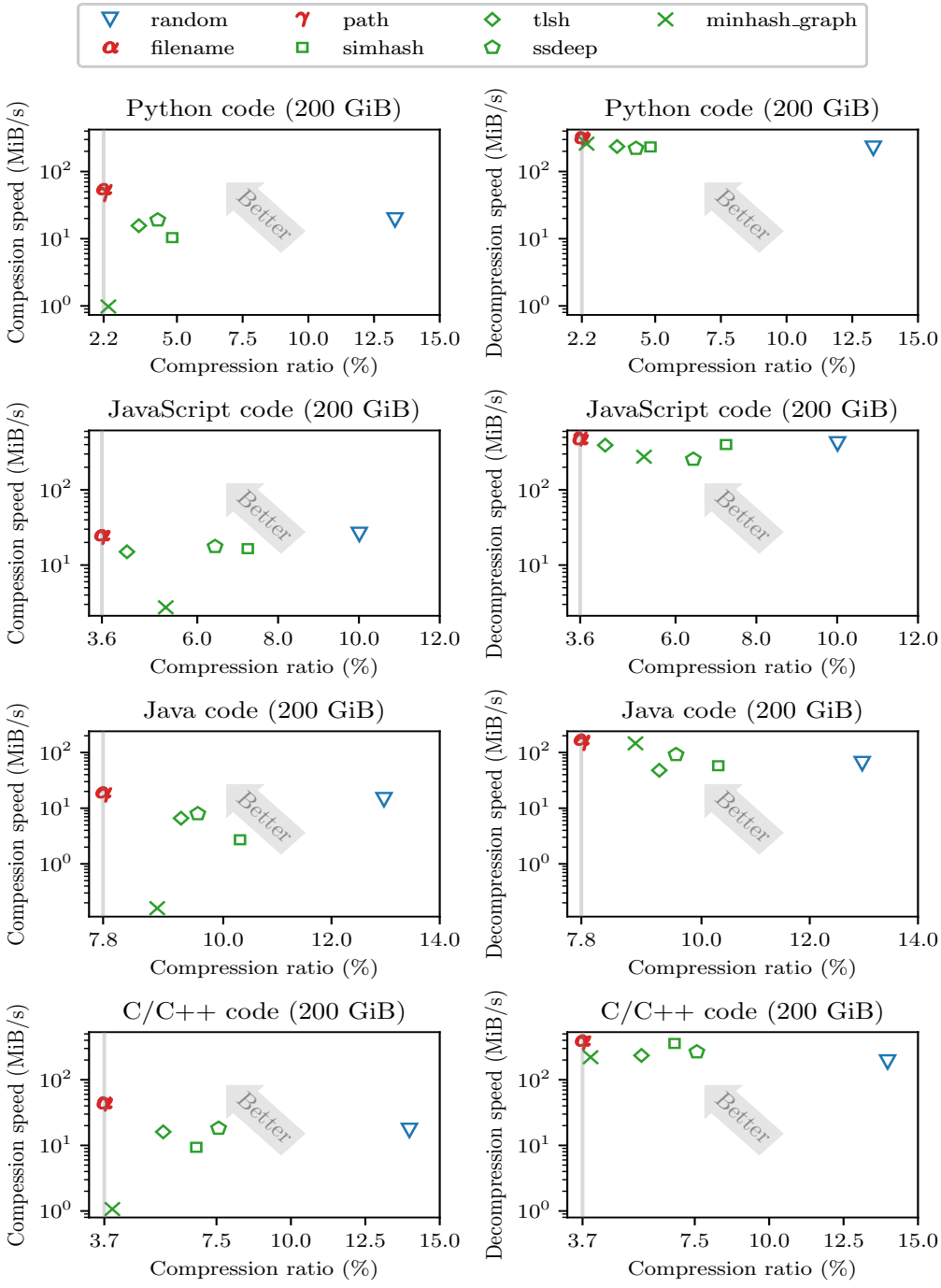

Figure 2: Compression and decompression speed vs compression ratio for the 200-GiB datasets in the Backup scenario, using zstd-22 as the final compressor.

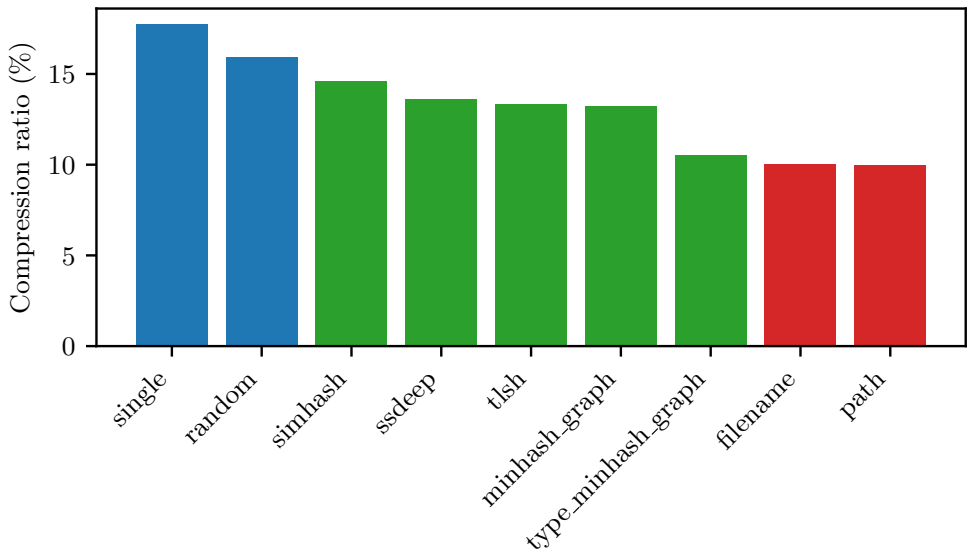

Figure 3: Compression ratio for the 200-GiB radom dataset in the Backup scenario, using different permuters and zstd-22 as the final compressor.

the above differences necessitates a comprehensive analysis involving also programming languages experts.

These compression ratios are achieved by the context-based permuter path (followed closely by filename), which is both simple and fast in (de)compression speed: a few tens of MiB/s in compression and a few hundreds of MiB/s in decompression. Among permuters that do not use context information but look just at the file content, the minhash_graph approach achieves the best compression ratio (close to that of filename) at the cost of a slow compression speed, because of the additional operations involving the fingerprint computations. Much faster compression speeds, but slightly worse compression ratios, are achieved by permuters that sort fingerprints of the file content, and here the tlsh is the most promising approach. As a final remark, we believe that the content-based permuters remain very relevant despite not achieving the best results in our experiments, especially because they can be applied in scenarios where context information is not available (i.e., classic key-value stores), or it is costly to be derived (i.e., SWH Archive).

4.1.3. Results for the SWH random dataset

Given the promising performance of the PPC-based compression techniques on source-code datasets, we move our attention to the whole SWH archive and consider a random sample of size 200 GiB (dataset Random Content in Section 3.1). This is intended to provide a glimpse of the performance of our techniques on the whole SWH archive, which includes files of many different types. A preliminary observation is that, with respect to the Popular GitHub repos dataset, the random sample from SWH dataset contains a larger amount of data different than source codes and texts (60% vs less than 50%). This is relevant for two reasons: 1) non textual data is often uncompressible (e.g., zipped files, multimedia files etc.), and 2) some of our fingerprinting techniques (simhash and minhash_graph) are text based in that they parse the input into text lines: they can be applied also to non-textual files but we can expect worse performance.

Figure 3 reports the compression performance of all tested PPC-based techniques and the two baselines single_compress and random_order. These experiments show a worse compression ratio, due to the presence of non source-code files, but confirm the validity of the PPC-based approach with a larger gap in compression ratio between type_minhash_graph and the other LSH-based techniques. This shows the effectiveness of the finer grouping by mime-types achieved by the ML-based classifier. Again, filename sorting (filename) and path sorting (path) achieve the best compression ratio of 9.99% and 9.98%, respectively.

4.2. Results for the File-Access scenario

In this section, we consider the File-Access scenario, where we aim to design a compression scheme that is succinct in space and is efficient in supporting access to the content of individual files. We restrict our test to four techniques: the three best PPC-based techniques tlsh, minhash_graph, and path, and random_order as a baseline. We excluded apriori git-pack since it only works on individual Git repositories and the bwt-based techniques which are unpractically slow. Since we experimented with the SWH source code datasets we also excluded type_minhash_graph since it behaves as minhash_graph but at a lower speed.

In the File-Access scenario, the partition phase of PPC-based approaches is crucial in that the larger the blocks to be compressed, the slower the time to access the individual files contained in them (see Section 3.2.4). The size B of those blocks thus influences the trade-off between compression ratio and

file-access speed. We experimented with $B = 256$ KiB and $B = 2$ MiB, and fix the block-compressor to zstd-12 because, as argued in Section 3.2.4, it offers an effective compromise between zstd-22 (very low compression ratio) and zstd-3 (very fast compression speed). As described in Section 3.2.4, we compute the file access speed by decompressing a random subset of 10% files in a dataset. Note that, given this random subset of files, the decompression time includes both (i) the time required to locate the block containing the target file (leveraging the mapping stored in RocksDB) and (ii) the time to decompress the block until the desired file is retrieved. Therefore, decompression speed is computed as the total number of bytes of the decompressed files divided by total decompression time.

Figure 4 shows the compression ratio, compression speed, and file access speed on the 200 GiB source-code datasets. As in Section 4.1.2, plots in the same row refer to the same dataset and share the vertical axis that allows to better compare the compression speed against the file access speed. Moreover, the plots show a vertical line that highlights the best compression ratio achieved by our tools. They also show the baseline solutions in blue, the content-based techniques in green, and the context-based ones in red. Notice that each PPC technique is shown twice, once with the block size set to $B = 256$ KiB (lighter colour) and once with $B = 2$ MiB (darker colour).

As expected, splitting and compressing the files block-wise impacts negatively on the compression effectiveness: the most effective solution in all datasets is still path+zstd-22 and achieves a compression ratio between 2.09% and 8.51% that is worse than the most effective solution in the Backup scenario. For example, compressing the Python code takes about 5.8% (instead of 2.2%), whereas compressing the Java code takes 9.9% (instead of 7.8%); hence, the more compressible the dataset, the larger the worsening gap.

Naturally, the parameter B induces some performance trade-off. Looking at the compression speed/compression ratio plots (first column in Figure 4), we see that a larger B is always better. This is because with a larger B we need to compress fewer blocks and zstd-12 can exploit more redundancies thus improving the compression ratio, which generates a win-win situation.[32] A different trade-off is observable when considering the decompression speed (second column of Figure 4). Not surprisingly the solution with $B = 2$ MiB is slower in file-access speed with respect to the one adopting $B = 256$ KiB be-

[32]Recall that a block may take more than B bytes if the last file surpasses that limit.

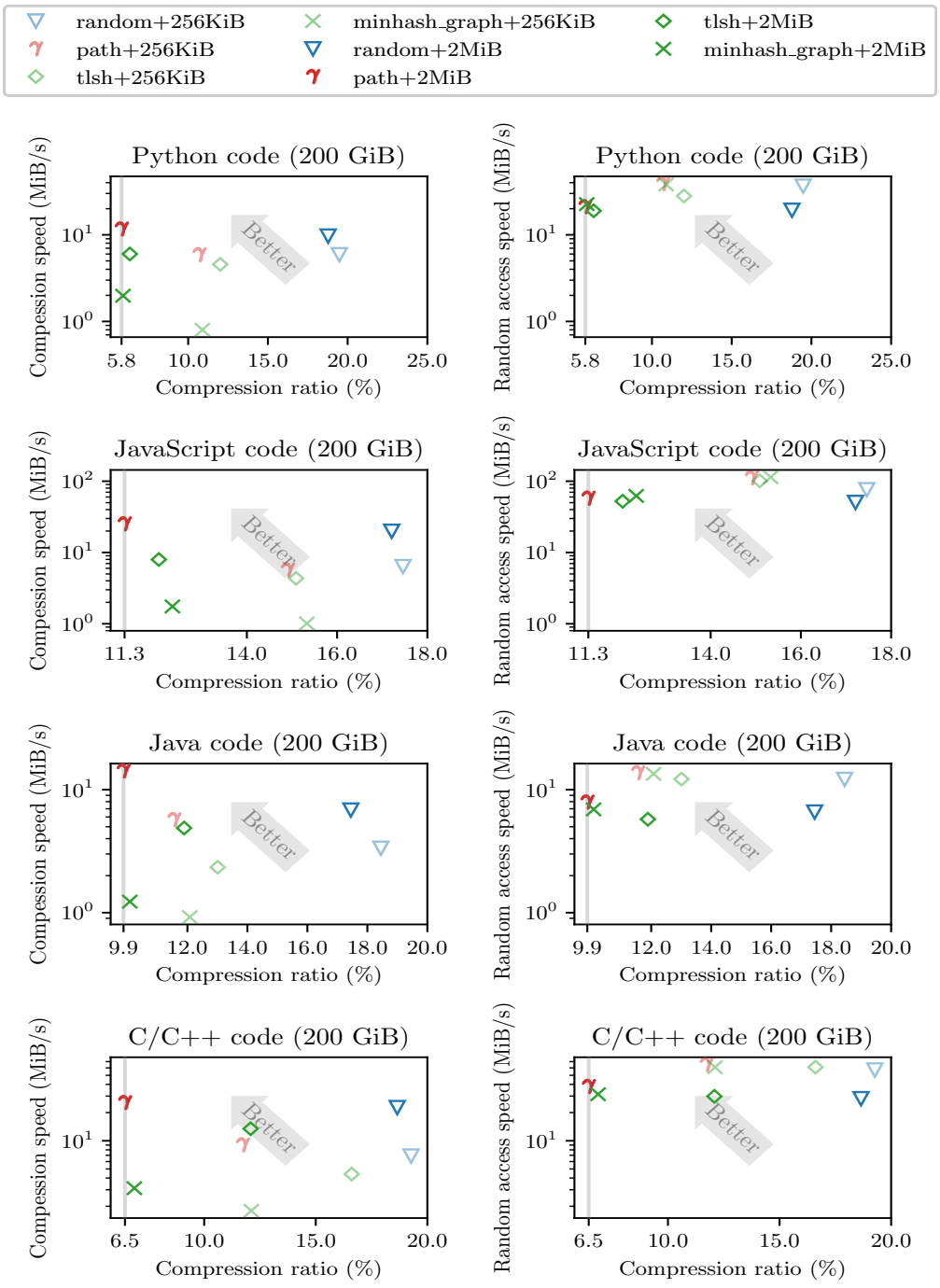

Figure 4: Compression and decompression speed vs compression ratio for the 200-GiB datasets in the File-Access scenario, using zstd-12 as the final compressor.

cause it needs to decompress a much larger block. We observe that the difference in access speed, for example between the JavaScript and Java datasets, has to be mainly attributed to the disparity in the number of files within each dataset (in fact, the Java dataset comprises approximately 26 million files, while the JavaScript dataset contains around 3 million files, see Table 1), which impacts on the storage infrastructure used to index and keep block-compressed these files.

Takehome message. Quite remarkably, the results on the File-Access scenario show that we can provide very fast file access speeds (up to 100 MiB/s), while achieving a significant compression ratio (with $B = 2$ MiB) of 5.82% on Python, 11.29% on JavaScript, 9.87% on Java, and 6.45% on C/C++. Using a smaller block size $B = 256$ KiB, we improve the file access speed by a factor 1.8×, at the cost of worsening the compression ratio by a factor 1.2× and the compression speed by a factor 2×.

Compared to the Backup scenario, as expected, the compression ratios deteriorate across all languages due to the files being compressed in blocks. Unexpectedly, the compressibility of different languages changes. In particular, JavaScript replaces Java as the least compressible language, while Python remains the most compressible language, and C/C++ the second best. This switch between Java and JavaScript is due to their different average file sizes. For the JavaScript dataset, the average file size is 60.5 KiB with a median of 4.6 KiB. In contrast, the Java dataset has an average file size of 7.9 KiB with a median of 2.4 KiB, see Table 1. Therefore, JavaScript files are, on average, more than 7 times longer than Java files. So, when using block-based compression in the File-Access scenario, we can fit into one block fewer JavaScript files than Java files. More precisely, the average number of JavaScript files in a 2-MiB block is 48, while for Java it is 268. Therefore, the final compression step based on `zstd` is penalized on JavaScript files because it can exploit less inter-file similarity.

5. Discussions

Limitations. We performed our experiments on a traditional POSIX-based filesystem. It is a simple, easily implementable, and extendable solution to evaluate the compression ratio of our compression techniques. We are aware it is not the best solution for evaluating compression and decompression speeds on massive amounts of small files. This is mainly because metadata

— such as path, permission, creation/modification dates, and location — are on the disk. These metadata must be read from disk into memory to find the file itself. While insignificant on a small scale, accessing metadata of billions of files and petabytes of data can be a bottleneck [5].

Furthermore, while we achieved our goal of investigating how to efficiently and effectively compress the contents of the SWH archive, there are some other important aspects that should be addressed in case of scaling up to its full size, such as parallel and distributed computation, fault tolerance, concurrency, etc., that we leave for future work.

As we leave to future work the fact that we restricted our analysis to four (very well-known) programming languages, although the SWH archive contains hundreds of different languages, each having its own compression ratio and inter-file similarity. Scaling to the compressed storage of the whole archive will require dealing with many other data types and file formats, so our space estimates of Section 4 (especially the ones for the Random dataset) have to be taken just as estimates, good and promising but estimates.

On the repetitiveness of programming languages. It is well-known, and often a topic of heated debates, that programs in some programming languages are more concise than in others. Some papers presented models for measuring the conciseness of programming languages in a consistent and objective way, eventually adopting information theoretical principles (see, e.g., [6] and references therein). Although interesting, those papers consider very small datasets (of about 13k files) and also clean the files by removing comments and spaces, which instead must be preserved in our lossless compression scenario. So they derive a different ordering of the most concise, and thus less expressive, programming languages among the ones considered in this paper. Overall, our experiments consider much larger datasets up to three to five orders of magnitude in number of compressed files. This makes them undoubtedly the largest examined in the literature, and our experiments show that the most compressible programming languages are Python (2.2%), followed by JavaScript (3.6%), C/C++ (3.7%), and Java (7.8%). These different compression ratios are likely due to multifaceted reasons: such as the number of files (for example Java has 9× more files than JavaScript see Table 1), repetitiveness introduced by automatic tools and by code developers, language history, and intrinsic properties of each language/framework.

Implications for the SWH Archive. Considering the entire SWH Archive as of now, the total size of source code written in C/C++ is 14.7 TiB, in Python

5.4 TiB, 50.3 TiB in JavaScript, and in Java 7.3 TiB. The experiments in Section 4.1 have shown that we can achieve a compression ratio that is about 4% averaging over those four programming languages. This is also confirmed by some preliminary experiments we conducted on snapshots of about 1 TiB. Thus, we can argue that we can compress all available public code in those four languages into just 3 TiB. In terms of (de)compression speed, the context-based permuters allow achieving a few tens of MiB/s in compression and a few hundreds of MiB/s in streaming-like decompression, and a few tens of MiB/s in single-file access. This file access speed implies that users can quickly retrieve and interact with even small portions of the compressed archive without the need for its full decompression. This capability is especially beneficial for tasks that require frequent access to different parts of the codebase, such as code analysis, software maintenance, and real-time collaboration in software development. The combination of high compression rates and fast file access speed makes our system not only space-efficient but also time-efficient, thus offering a scalable solution for managing and accessing large-scale source code repositories in various computing environments.

6. Conclusions

Our results have shown that, on the source codes of the SWH Archive, it is possible to achieve remarkable compression ratios (4% on average, or even less on specific programming languages, such as Python) by exploiting the inter-file redundancies that are naturally present in such file collections. The PPC-based techniques appear to be very effective for this task in that combined with standard compression tools outperform ad-hoc techniques, like git-pack, or compressors designed to handle very-long inputs, like bwt_rle. Among the different PPC variants, we ontained the best performance when contextual information, even only the file name, is available and exploited. When we lack such information, or the file name is not a good proxy of file-content similarity (e.g., blog or social-network posts, storage systems, key-value stores, etc.), we can still achieve good performances with the PPC-paradigm using proper fingerprints extracted from the file contents. We are currently investigating an example of such alternative PPC-use: for key-value stores we compute for each $\langle k, v\rangle$-pair, a proper fingerprint of the "value"-component and use it as a new key to create a different key-value store with pairs $\langle f(v), v\rangle$, and a satellite associative table $\langle f(v), k\rangle$ to keep track of the implemented key-mapping. This way, similar values would come close in

the (new) key-value store thus eventually leveraging its compressed plug-in functionalities. This application shows the robustness of the PPC framework, making it a versatile tool for compressing large file collections across a variety of situations and datasets. Further algorithmic research is still needed to make the content-based techniques faster and thus scalable to larger and larger data collections.

Our algorithmic framework can be immediately applied to all contexts where storing large amounts of source code files is required. At present, this is a concrete need for large archives like SWH (our main experimental use case), research infrastructures for empirical software engineering like WoC and Boa (see Section 2.3), as well as large source code datasets that are growing in popularity driven by the needs of large language models (LLMs). The state-of-the-art of practical storage solutions in those contexts remains pretty naive, relying on ad-hoc object storages and their built-in compression solutions that do not take into account specific aspects of software source code. We have shown how storage requirements for those use cases can be reduced significantly, while retaining a very simple implementation, based on state-of-the-art techniques and technologies.

Building on the promising outcomes of our research, the next crucial step involves applying our techniques to compress the whole collection of source files written in the most used programming languages, as well as the entire SWH Archive. This latter endeavor requires a strategic approach to distinguish between compressible and non-compressible objects within the archive to avoid spending computing time on incompressible files. The mentioned solutions, in particular type_minhash_graph, will arguably play a pivotal role in this regard.

Data availability

The code to download and decompress the datasets used for the current study, as well as the scripts to reproduce all the experiments, are available in the `PPC_utils4BigData` repository. It has a Software Heritage persistent Identifier (SWHIDs)[33] and is also available on GitHub at the link: https://github.com/acubeLab/PPC_utils4BigData.

[33] swh:1:dir:b428d05ec4bc45c9ebee8a39a9d08b6e197df587

Conflict of interest

The authors declare that they have no conflict of interest.

Acknowledgement

We express our gratitude to the staff of the Green Data Centre at the University of Pisa for generously providing us with the necessary machines and technical support crucial for executing the numerous experiments detailed in this paper.

This work was partially supported by the Alfred P. Sloan Foundation with the grant #G-2025-25193[34]; by the European Union – Horizon 2020 Program under the scheme "INFRAIA-01-2018-2019 – Integrating Activities for Advanced Communities", Grant Agreement n. 871042, "SoBigData++: European Integrated Infrastructure for Social Mining and Big Data Analytics" (http://www.sobigdata.eu); by the NextGenerationEU – National Recovery and Resilience Plan (Piano Nazionale di Ripresa e Resilienza, PNRR) – Project: "SoBigData.it - Strengthening the Italian RI for Social Mining and Big Data Analytics" – Prot. IR0000013 – Avviso n. 3264 del 28/12/2021; by the spoke "FutureHPC & BigData" of the ICSC – Centro Nazionale di Ricerca in High-Performance Computing, Big Data and Quantum Computing funded by European Union – NextGenerationEU – PNRR; and by the Italian Ministry of University and Research "Progetti di Rilevante Interesse Nazionale" project: "Multicriteria data structures and algorithms" (grant n. 2017WR7SHH).

We thank Valentin Lorentz from the Software Heritage engineering team for his help in obtaining relevant datasets from the Software Heritage archive.

References

[1] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. Building the universal archive of source code. *Commun. ACM*, 61(10):29–31, sep 2018.

[2] Carol V. Alexandru, Sebastiano Panichella, Sebastian Proksch, and Harald C. Gall. Redundancy-free analysis of multi-revision software artifacts. *Empir. Softw. Eng.*, 24(1):332–380, 2019.

[34] sloan.org

[3] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał undefinedwitakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. Delta lake: High-performance acid table storage over cloud object stores. *PVLDB*, 13(12):3411–3424, aug 2020.

[4] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. Amazon redshift re-invented. In *Proc. 2022 International Conference on Management of Data (SIGMOD)*, pages 2205–2217, New York, NY, USA, 2022. Association for Computing Machinery.

[5] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, October 2010.

[6] Lodewijk Bergmans, Xander Schrijen, Edwin Ouwehand, and Magiel Bruntink. Measuring source code conciseness across programming languages using compression. In *21st IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Luxembourg, September 27-28, 2021*, pages 47–57. IEEE, 2021.

[7] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya G. Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *Proc. VLDB Endow.*, 8(12):1346–1357, 2015. Version with the appendix available at https://arxiv.org/pdf/1505.05211.

[8] Paolo Boldi, Antoine Pietri, Sebastiano Vigna, and Stefano Zacchiroli. Ultra-large-scale repository analysis via graph compression. In *Proc. 27th International IEEE Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 184–194, 2020.

[9] Paolo Boldi and Sebastiano Vigna. The webgraph framework I: compression techniques. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 595–602. ACM, 2004.

[10] Frank Breitinger, Barbara Guttman, Michael McCarrin, Vassil Roussev, and Douglas White. Approximate matching : definition and terminology. *NIST Special Publication*, (800-168), May 2014.

[11] Andrei Z. Broder. On the resemblance and containment of documents. In *Proc. Compression and Complexity of SEQUENCES*, pages 21–29, 1997.

[12] Adam L. Buchsbaum, Donald F. Caldwell, Kenneth Ward Church, Glenn S. Fowler, and S. Muthukrishnan. Engineering the compression of massive tables: an experimental approach. In *Proc. 11th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 175–184, 2000.

[13] Adam L. Buchsbaum, Glenn S. Fowler, and Raffaele Giancarlo. Improving table compression with combinatorial optimization. *J. ACM*, 50(6):825–851, nov 2003.

[14] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[15] Moses Charikar. Similarity estimation techniques from rounding algorithms. In *Proc. 34th Annual ACM Symposium on Theory of Computing (STOC)*, pages 380–388, 2002.

[16] Roberto Di Cosmo and Stefano Zacchiroli. Software Heritage: why and how to preserve software source code. In *Proc. 14th International Conference on Digital Preservation (iPRES)*, 2017.

[17] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. RocksDB: evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, 17(4), oct 2021.

[18] Fred Douglis and Arun Iyengar. Application-specific delta-encoding via resemblance detection. In *Proc. General Track 2003 USENIX Annual Technical Conference*, pages 113–126, 2003.

[19] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: Ultra-large-scale software repository and source-code mining. volume 25, pages 7:1–7:34, 2015.

[20] Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, Jul 2012.

[21] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[22] Paolo Ferragina and Giovanni Manzini. On compressing the textual web. In *Proc. 3rd ACM International Conference on Web Search and Data Mining (WSDM)*, pages 391–400, 2010.

[23] Dennis Fetterly, Mark S. Manasse, and Marc Najork. On the evolution of clusters of near-duplicate web pages. In *Proc. 1st Latin American Web Congress (LA-WEB)*, pages 37–45, 2003.

[24] Tommaso Fontana, Sebastiano Vigna, and Stefano Zacchiroli. Webgraph: The next generation (is in rust). In Tat-Seng Chua, Chong-Wah Ngo, Roy Ka-Wei Lee, Ravi Kumar, and Hady W. Lauw, editors, *Companion Proceedings of the ACM on Web Conference 2024, WWW 2024, Singapore, Singapore, May 13-17, 2024*, pages 686–689. ACM, 2024.

[25] Markus Hsi-Yang Fritz, Rasko Leinonen, Guy Cochrane, and Ewan Birney. Efficient storage of high throughput dna sequencing data using reference-based compression. *Genome research*, 21(5):734–740, 2011.

[26] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The Pile: an 800GB dataset of diverse text for language modeling. *CoRR*, abs/2101.00027, 2021.

[27] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures.

In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014.

[28] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[29] Nevin Heintze. Scalable document fingerprinting. In *Proc. 2nd USENIX Workshop on Electronic Commerce (EC)*, 1996.

[30] Morteza Hosseini, Diogo Pratas, and Armando J. Pinho. A survey on data compression methods for biological sequences. *Information*, 7(4), 2016.

[31] Xiaokang Hu, Fuzong Wang, Weigang Li, Jian Li, and Haibing Guan. QZFS: QAT accelerated compression in file system for application agnostic and cost efficient data storage. In *Proc. 2019 USENIX Annual Technical Conference (USENIX ATC)*, pages 163–176, 2019.

[32] Vitor Hugo G Moia and Marco Aurelio Amaral Henriques. Similarity digest search: A survey and comparative analysis of strategies to perform known file filtering using approximate matching. *Security and Communication Networks*, 2017:1–17, 09 2017.

[33] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. Delta algorithms: an empirical analysis. *ACM Trans. Softw. Eng. Methodol.*, 7(2):192–214, apr 1998.

[34] Stratos Idreos and Mark Callaghan. Key-value storage engines. In *Proc. 2020 ACM International Conference on Management of Data (SIGMOD)*, pages 2667–2672, 2020.

[35] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. 30th Annual ACM Symposium on Theory of Computing (STOC)*, pages 604–613, 1998.

[36] Omid Jafari, Preeti Maurya, Parth Nagarkar, Khandker Mushfiqul Islam, and Chidambaram Crushev. A survey on locality sensitive hashing algorithms and their applications. *ArXiv*, abs/2102.08942, 2021.

[37] Jesse D. Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digit. Investig.*, 3(Supplement):91–97, 2006.

[38] Valentin Lorentz, Roberto Di Cosmo, and Stefano Zacchiroli. The Popular Content Filenames Dataset: Deriving most likely filenames from the Software Heritage Archive. working paper or preprint, July 2023.

[39] Yuxing Ma, Tapajit Dey, Chris Bogart, Sadika Amreen, Marat Valiev, Adam Tutko, David Kennard, Russell Zaretzki, and Audris Mockus. World of code: enabling a research workflow for mining and analyzing the universe of open source VCS data. *Empir. Softw. Eng.*, 26(2):22, 2021.

[40] Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. Language models of code are few-shot commonsense learners. In *Proc. 2022 Conference on Empirical Methods in Natural Language Processing*, pages 1384–1403. Association for Computational Linguistics, December 2022.

[41] Udi Manber. Finding similar files in a large file system. In *Proc. USENIX Winter 1994 Technical Conference*, pages 1–10, 1994.

[42] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[43] Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, may 2001.

[44] Angelos Molfetas, Anthony Wirth, and Justin Zobel. Using inter-file similarity to improve intra-file compression. In *Proc. 2014 IEEE International Congress on Big Data*, pages 192–199, 2014.

[45] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's warm BLOB storage system. In *Proc. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 383–398. USENIX Association, October 2014.

[46] Gonzalo Navarro. Indexing highly repetitive string collections, part ii: Compressed indexes. *ACM Comput. Surv.*, 54(2), feb 2021.

[47] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), April 2007.

[48] Jonathan Oliver, Chun Cheng, and Yanggui Chen. TLSH – a locality sensitive hash. In *Proc. 4th Cybercrime and Trustworthy Computing Workshop (CTC)*, pages 7–13, 2013.

[49] Jonathan Oliver, Scott Forman, and Chun Cheng. Using randomization to attack similarity digests. In *Applications and Techniques in Information Security*, pages 199–210, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[50] Zan Ouyang, Nasir D. Memon, Torsten Suel, and Dimitre Trendafilov. Cluster-based delta compression of a collection of files. In *Proc. 3rd International Conference on Web Information Systems Engineering (WISE)*, pages 257–268, 2002.

[51] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's Tectonic filesystem: Efficiency from exascale. In *Proc. 19th USENIX Conference on File and Storage Technologies (FAST)*, pages 217–231, February 2021.

[52] Antoine Pietri. *Organizing the graph of public software development for large-scale mining*. Theses, Université Paris Cité, November 2021.

[53] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):article 43, November 2007.

[54] Torsten Suel. Delta compression techniques. In Sherif Sakr and Albert Y. Zomaya, editors, *Encyclopedia of Big Data Technologies*. Springer, 2019.

[55] Torsten Suel and Nasir D. Memon. *Lossless Compression Handbook*, chapter Algorithms for delta compression and remote file synchronization. Academic Press, 2002.

[56] Walter F. Tichy. Rcs — a system for version control. *Software: Practice and Experience*, 15(7):637–654, 1985.

[57] Sebastian Wandelt, Marc Bux, and Ulf Leser. Trends in genome compression. *Current Bioinformatics*, 9(3):315–326, 2014.

[58] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for similarity search: A survey. *CoRR*, abs/1408.2927, 2014.

[59] Wen Xia, Chunguang Li, Hong Jiang, Dan Feng, Yu Hua, Leihua Qin, and Yucheng Zhang. Edelta: A word-enlarging based fast delta compression approach. In *Proc. 7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*. USENIX Association, 2015.

[60] Matei A. Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In *Proc. 11th Conference on Innovative Data Systems Research (CIDR)*, 2021.