SD*Gen:* Mimicking Datasets for Content Generation in Storage Benchmarks

Raúl Gracia-Tinedo, *Universitat Rovira i Virgili;* Danny Harnik, Dalit Naor, and Dmitry Sotnikov, *IBM Research Haifa;* Sivan Toledo and Aviad Zuck, *Tel-Aviv University*

https://www.usenix.org/conference/fast15/technical-sessions/presentation/gracia-tinedo

This paper is included in the Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15).

February 16–19, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-201

SDGen: Mimicking Datasets for Content Generation in Storage Benchmarks

Raúl Gracia-Tinedo
Universitat Rovira i Virgili (Spain)
raul.gracia@urv.cat

Danny Harnik, Dalit Naor, Dmitry Sotnikov
IBM Research-Haifa (Israel)
{dannyh, dalit, dmitrys}@il.ibm.com

Sivan Toledo, Aviad Zuck
Tel-Aviv University (Israel)
{stoledo, aviadzuc}@tau.ac.il

Abstract

Storage system benchmarks either use samples of proprietary data or synthesize artificial data in *simple ways* (such as using zeros or random data). However, many storage systems behave completely differently on such artificial data than they do on real-world data. This is the case with systems that include data reduction techniques, such as compression and/or deduplication.

To address this problem, we propose a benchmarking methodology called *mimicking* and apply it in the domain of data compression. Our methodology is based on *characterizing the properties of real data* that influence the performance of compressors. Then, we use these characterizations to generate new synthetic data that mimics the real one in many aspects of compression. Unlike current solutions that only address the *compression ratio* of data, mimicking is flexible enough to also emulate *compression times* and *data heterogeneity*. We show that these properties matter to the system's performance.

In our implementation, called *SDGen*, characterizations take at most 2.5KB per data chunk (e.g., 64KB) and can be used to efficiently share benchmarking data in a highly anonymized fashion; sharing it carries few or no privacy concerns. We evaluated our data generator's accuracy on compressibility and compression times using real-world datasets and multiple compressors (`lz4, zlib, bzip2` and `lzma`). As a proof-of-concept, we integrated *SDGen* as a content generation layer in two popular benchmarks (LinkBench and Impressions).

1 Introduction

Benchmarking is a fundamental building block for researchers and practitioners to measure the performance of storage systems in a reproducible manner [35]. Indeed, the research community has taken remarkable steps towards accurately emulating observed workloads into experimental assessments. A myriad of examples can be found in the literature, such as benchmarks for file systems [7, 31], cloud storage [13, 16, 22] or databases [9, 15], to name a few.

However, most storage benchmarks do not pay particular attention to the *contents* generated during their execution [35] (see examples in Table 1). For instance, Impressions [7] implements accurate statistical methods to model the structure of a file system, but the contents of files are by default zeros or statically generated by third-party applications. Another example is OLTPBench [15], which provides a rich suite of database workloads and access patterns; however, the payload of queries is filled with random data. Clearly these contents are not realistic. Thus, the following question arises: *does the content matter to the performance analysis of systems?* The answer is a definitive *yes* when data reduction is involved.

Data reduction and performance sensitivity: To improve performance and capacity, a variety of storage systems integrate data reduction techniques [11, 21, 23, 25, 27]. This can have two crucial effects on the performance of the storage system: (i) When data is highly compressible, the amount of bytes actually written to the storage diminishes and performance can improve dramatically. (ii) The running time of compression algorithms varies greatly for different data types (e.g. [18]) and hence can affect the overall throughput and latency of the system. As a result, the performance of many systems with data reduction techniques is extremely content-sensitive.

To illustrate this, we measured the transfer times of ZFS, a file system with built-in compression [27, 38]. We copied sequentially 1GB files filled with low (random) and high (zeros) compressible data. The results in Fig. 1 support our claim: *the transfer times of ZFS greatly vary depending on the file contents*. Thus, two executions of the same benchmark may report disparate performance results of ZFS, just depending on the data used.

Current solutions: One solution to benchmarking a content-sensitive storage system is to use real life datasets for executing the tests. However, this practice is limiting due to the burden of copying large amounts of data onto the testing system. Even more so, privacy con-

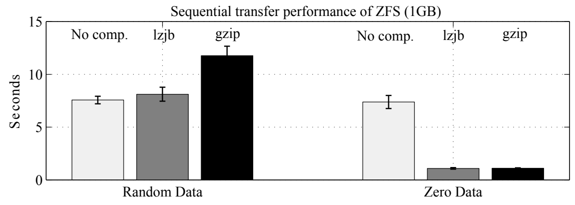

Figure 1: Sequential transfer times of ZFS depending on file contents with and without compression.

cerns greatly inhibit this approach as end users typically are unwilling to share their proprietary data [33].

Another practice which is gradually being adopted is generating synthetic data with definable compressibility. For example, *VDBench* [2], *Fio* [4] and *LinkBench* [9] all offer synthetic data with tunable compression ratio. This tuning is achieved by mixing incompressible and highly compressible data at appropriate and variable proportions. The shortcoming of this approach is that it only considers a single dimension —that of compressibility. It ignores the *time* required for actual compression and does not support *heterogeneity* of the data within files.

To exemplify this, we tested data created in *LinkBench* for its compression properties. Thus, using `zlib` we calculated the compression ratio of the original chunks [10], defined as $\frac{original\_size}{compressed\_size}$, to generate synthetic chunks of similar compressibility with LinkBench. The results, shown in Fig. 2, confirm that (i) compression ratios are fairly accurate but, unlike what happens with real data, insensitive to the compressor, and (ii) compression times are very inaccurate —affecting the system's performance.

Thus, we face a situation where most storage benchmarks generate unrealistic contents, whereas representative datasets cannot be shared with the community. This reflects a need for a *common substrate for generating realistic and reproducible benchmarking data*.

1.1 Our Contributions

We present Synthetic Data Generator (*SDGen*): an open and extensible framework for generating realistic storage benchmarking contents. *SDGen* is devised to produce data that *mimics* real-world data. In this paper we focus on mimicking compression related properties, but we view it as a wider framework that can be used for other properties as well. The framework consists of a pluggable architecture of *data generators and characterizations*. Basically, characterizations capture properties of datasets and are then used by data generators to create arbitrary amounts of similar synthetic data. A salient feature of *SDGen* is that researchers can *share dataset characterizations instead of actual contents* to generate realistic synthetic data in a reproducible manner.

The main features of our solution include:

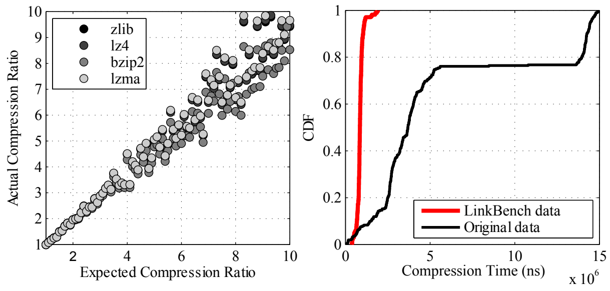

Figure 2: Compression ratios of LinkBench synthetic chunks for 4 compression engines (left). Compression times cumulative distribution function (CDF) of LinkBench data vs Canterbury corpus data (right).

Mimicking compression: Our first contribution is to identify the *properties of data* that are key to the performance of compressors, and therefore, to the performance of systems. Naturally, finding a universal solution to mimic data for *all* compressors is hard. The reason is that different compressors are guided by distinct heuristics in order to compress data. We therefore chose to focus on the most common category of compressors used in storage systems. Specifically, we target lossless compression that is either based on *byte level repetition finding* (Lempel-Ziv style algorithms [41, 42]) and/or on *entropy encoding* (e.g. Huffman encoding [20]).

As a second contribution, *SDGen* generates data that *mimics* both *compression ratios* and *compression times* of the original dataset for several compression engines. Moreover, our synthetic data exhibits similar *variability* of these parameters compared to the original datasets. Our tests exhibit that, on average, *SDGen* generates synthetic data for which compression ratio deviates less than 10% *and* its processing time deviates less than 15% from the original data. This was shown for disparate data types and with several different compression engines (`lz4`, `zlib-1`, `zlib-6`). For other compressors that vary in their core methods (`lzma`, `bzip2`) the results are less tight but also acceptable. We also verify the mimicking effect of working with our synthetic data over ZFS.

Compact and anonymized representation: Our third contribution is to design a *practical and private way of sharing benchmarking data*. *SDGen* users can reproduce a synthetic dataset by sharing a compact characterization file, being agnostic to the original dataset contents. This approach benefits from easy mobility coupled with the privacy of not sharing actual data. The characterization takes just 2.5KB per data chunk (for arbitrary chunk size, e.g. 64KB). We also explore the option to use random sampling to efficiently scan very large datasets, creating a constant size characterization while maintaining high accuracy in mimicking the entire dataset. In our tests, the overall characterization does not to exceed 8.5MB irrespective of the dataset size, which can be Terabytes.

Storage Domain	*Article/Benchmark*	*Data Generation Method*
File System	FileBench [5]	(**R**) Internally, FileBench gets random offsets of the internal memory to fill write buffers.
	Impressions [7]	(**C/D**) Binary files are zeroed whereas text files are filled with a static list of words sorted by popularity (English language). Impressions relies in third party applications to generate specific file types (mp3, jpeg).
Micro-benchmarks	IOzone [26]	(**R**) Random data that can be specified with a ratio of change per-operation to specify deduplication.
	VDBench [2]	(**M**) Compression ratio supported by mixing random and zero data and block repetition for deduplication.
	Bonnie++ [3]	(**R**) Write/update operations are filled with non-initialized char arrays.
	fio [4]	(**M**) Mix of random and zero data to fill IO operations with compression defined data.
Database/KV Store	OLTP [15]	(**R**) Payloads of queries are filled with random data.
	LinkBench [9]	(**M**) Static configuration of data compressibility that mixes new and existing data in each query.
	YCSB [13]	(**C**) The values of each table field are a string of ASCII characters of predefined length.
Cloud Storage	CloudCmp [22]	(**R**) Blobs for `put` operations are filled with random data.
	Drago et. al. [16]	(**R**) Benchmarking files are either random or with random words to emulate text.
	COSBench [39]	(**R**) API `put` operations are filled with random data.
	ssbench [1]	(**C**) Uploaded objects are filled with a single character (e.g. 'A').
Deduplication	Tarasov et. al. [33]	(**D**) Generation of deduplication workloads based on a Markov model. Initial contents are delegated to a first dataset image.
	DEDISBench [9]	(**D**) Initial contents are delegated to a first dataset image.

Generated contents are: (**R**)andom data, (**C**)onstant/zeros data, (**M**)ix of compressible and non-compressible data, (**D**)elegated to application/assumes initial dataset

Table 1: Data generation approaches for several widely adopted benchmarks in various storage domains.

Usability and integration: We plan to release *SDGen* to the community[1] as well as a set of public characterizations for some popular data types. Users can either use a public characterization, or create a new one in order to mimic their proprietary data. As a proof-of-concept, we also integrated *SDGen* as a data generation service into two well-known benchmarks suites: *Impressions* [7] (file system) and *LinkBench* [9] (social graph).

Paper organization: The rest of the paper is structured as follows. Section 2 discusses related work on synthetic data generation. Section 3 presents the *SDGen* architecture and in Section 4 we present our data characterization and generation methods for compression techniques. We evaluate *SDGen* in sections 5 and 6. In section 7 we describe the integration of *SDGen* with popular benchmarking tools. We draw some conclusions in Section 8.

2 Related Work

Benchmarking storage systems has long been an important research topic for the storage community. A vast amount of microbenchmarks and domain-specific benchmarks have been proposed in the last decade [5, 7, 9, 26, 28]. However, although these solutions provide flexible and realistic [8, 32] workload modeling, they do not consider the generated content as this is not their goal.

In Table 1, we summarize how many of these benchmarks generate data as stated in the official documentation or as we inferred by inspecting their source code. As mentioned in the introduction, some of these directly offer compression ratio tuning and some offer simple implementation also for deduplication ratio. We also refer the reader to [35] for an excellent overview of the state-of-the-art in storage benchmarking.

Tay [34] advocates for an application-specific benchmarking approach [29]. This work aims to augment an empirical dataset to an arbitrary size for database benchmarking. It provides a theoretical study of how to keep the internal database structure. For RDF databases Schmidt et. al. [28] suggested to include document and query generation modules based on a study of the DBLP system. Similarly to LinkBench [9], they emulate the behavior of users to guide the workload execution. Adir et al. presented an approach for benchmarking databases in which data is generated according to the customer's specifications in order to match his proprietary settings and data types [6]. They can optionally scan specified database columns to collect statistics on used words and use them in the benchmark.

Generating realistic contents for system benchmarking seems to be gaining momentum in the field of big data [36]. The authors of [36] propose BigDataBench, a complete benchmark for systems such as Hadoop and key-value stores. In particular, BigDataBench provides a data generation module that emulates predefined data types (e.g. text, graphs). In contrast, *SDGen* analyzes any given dataset to generate similar synthetic data. That is, a text file can be very compressible or not, depending on its contents. *SDGen* is able to capture this characteristic and generate synthetic data accordingly.

The closest work to the present paper we are aware of is that of Tarasov et. al. [33], which identified the relevance of generating realistic workloads for benchmarking deduplication systems. They propose a framework to capture and share the *updates* of datasets; traces representing *update operations* can be reproduced over other datasets to emulate deduplication. Clearly, *SDGen* shares the same spirit of [33]. However, in practice Tarasov et. al. delegate the actual dataset contents to an *initial image/snapshot*. *SDGen* fills this gap by providing synthetic initial contents similar to the original ones, which is a preliminary step to the deduplication benchmarking.

3 *SDGen*: Framework Architecture

SDGen is designed to capture characteristics of data that can affect the outcome of applying data reduction techniques on it. As we show next, *SDGen* works in two phases: A priming *scan* phase which build data characterizations to be used by a subsequent *generation* phase.

[1] Available at `https://github.com/iostackproject/SDGen`.

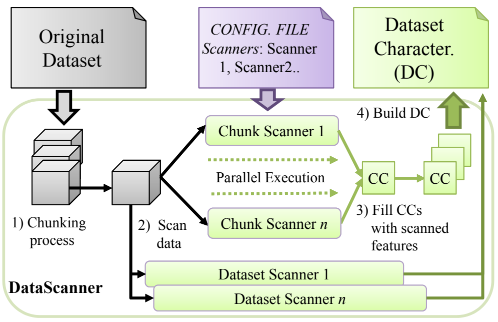

Figure 3: *SDGen* dataset scanning and characterization.

3.1 Scan Phase: Characterizations

To capture the characteristics of data, *SDGen* implements a two-level scan phase: *chunk level* and *dataset level*.

Many compression algorithms (e.g. `lz4`, `zlib`) partition the input data stream into chunks, and apply compression separately for every chunk [41]; such algorithms try to exploit redundancy which stems from locality of data (repetitions, common bytes) while minimizing the size of their internal data structures. Therefore, a central element in our design is the **chunk characterization** (*CC*). A CC is a user-defined module that contains the necessary information for every data chunk. *SDGen* scans a given dataset by splitting its contents into chunks (e.g., from 8KB to 128KB, configurable by the user) that are characterized individually (step 1 and 2, Fig. 3). We depict our CC design in Section 4.2.

In a higher level, *SDGen* builds **dataset characterizations** (DC), which provide a more holistic characterization. In the current version of *SDGen*, DCs store the deduplication ratio of the entire dataset as well as a list of all the previously generated CCs.

To support the above scans, *SDGen* applies two modules: `Chunk scanners` and `Dataset scanners`. These modules are loaded from the configuration in a manager class (`DataScanner`), which processes the dataset, and concurrently uses it as input for the scanners in order to build the characterization. The `DataScanner` life-cycle appears in Fig. 3.

The scan phase ends by persistently storing a DC (step 4, Fig. 3). *SDGen* also includes a way of transparently storing and loading DCs, enabling users to easily creating and sharing them.

3.2 Generation Phase

Once in possession of a DC, users may load it in *SDGen* to generate synthetic data similar to the original dataset.

The heart of the generation phase is the *generation algorithm*. This algorithm is designed by the user and receives as input a CC filled with the data characteristics captured by chunk scanners (see Section 4.3). Since CCs are read-only and independent of each other, the generation algorithm can utilize parallelism for faster data generation. A module called `DataProducer` orchestrates the content generation process. The `DataProducer` is also responsible for taking into account dataset-level characteristics during the generation process. Currently, this is mainly used for generating duplicated data. However, we concentrate on data compression, leaving the analysis of deduplicated data for future work.

The `DataProducer` module generates data using two API calls: `getSynData()` and `getSynData(size)`. The first call retrieves entire synthetic chunks with the same size as the original chunk. This is adequate for generating large amounts of content, such as file system images. The second call specifies the size of the synthetic data to be generated. This call is an optimization to avoid wasting synthetic data in benchmarks that require small amounts of data per operation (e.g. OLTP, databases). Technically, successive executions of this method will retrieve subparts of a synthetic chunk until it is exhausted and a new one is created.

3.3 Sampling at Chunk-level

SDGen generates a chunk characterization data structure for each data chunk. However, the time to scan a very large dataset can be prohibitively long and the size of the characterizations can grow excessively. To remedy this, we resort to sampling, i.e. scanning only a random fraction of a given dataset.

The crux is that random sampling is a good estimator for many properties of the data, and specifically for properties that can be expressed by averages and sums such as compression ratio or compression time. Harnik et. al. [18] show that using random sampling on chunks is a good estimator for compression ratio (within an additive percentage factor). The same also holds for estimating the fraction of data with specific compressibility or within specific compression time limits. Note that compression time of blocks is not bounded the way that compression ratio is. Compression time typically has higher variance, so typically its estimation is less tight than that of compression ratio. Still we argue, and corroborate through experimentation, that the sampling's accuracy is well within what is required for benchmarking.

The actual number of samples is a constant regardless of the size of the entire data set. In our tests we took $\sim 3,500$ chunks (this number meets accuracy guarantees provided in [18][2]), where each chunk in the data set is chosen with equal probability. This sampling can be done in a simple manner when dealing with large files or block devices, or by using the methodology of [17] for the case of file systems and other complex structures. For

[2]Sample size is set by *confidence* and *accuracy* parameters. *Accuracy* (we use the value 0.05) measures the additive distance that the estimation can vary from the actual compression ratio, while *confidence* (we use 10^{-6}) bounds the probability of falling outside this accuracy range (probability is taken over the randomness of the sampling).

each of the chosen chunks, a characterization is created and stored. In the data generation phase, data is created by taking characterizations in a round robin fashion. We test the accuracy of this sampling strategy in Section 6.6.

3.4 How to Extend *SDGen*

SDGen enables users to integrate novel data generation methods in the framework. To this end, one should follow three steps:

1. *Characterization*: Create a CC extending the `AbstractChunkCharacterization` class. This user-defined characterization should contain the required information for the data generation process.

2. *Scanners*: Provide the necessary scanners to fill the content of CCs and DCs during the scan process. Chunk-level scanners should extend from `AbstractChunkScanner` and implement the method `setInfo`, to set the appropriate CC fields.

3. *Generation*: Design a data generation algorithm according to the properties captured during the scan phase. This algorithm should be embedded in a module extending `AbstractDataGenerator`, to benefit from the parallel execution offered by `DataProducer`. Concretely, a user only needs to override the `fill(byte[])` method to fill with synthetic data the input array.

SDGen manages the life-cycle of the user-defined modules to scan/generate data, which are loaded from a simple configuration file. Finally, *SDGen* consists of 5,800 lines of Java code, including the framework architecture, our generation methodology (plus `Deflate` algorithm), the integration with LinkBench, and 200 lines of C++ code for integration with Impressions.

4 Compression-oriented Synthetic Data

Creating an efficient and accurate *mimicking method* is a non-trivial task (i.e. characterization, generation). In this section, we describe the research insights that guided the design of our method. We evaluate it in Section 6.

4.1 Generation Method Rationale

Mimicking data for compressors requires a basic understanding of how compressors work. In this work we target compressors that utilize repetition elimination to reduce data size. We also target compressors that use entropy coding (such as Huffman codes), typically on top of repetition elimination.

With this in mind, and based on empirical tests we identified two main characteristics that affect the performance and behavior of compression algorithms: *repetition length distribution* and *frequencies of bytes*.

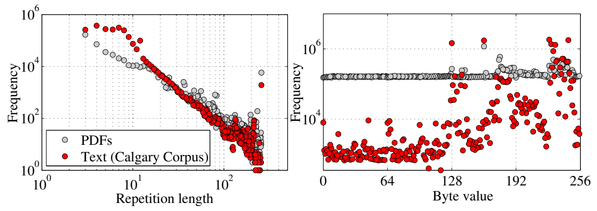

Figure 4: Repetition length distribution (left) and byte frequency (right) in PDFs and text data.

In repetition elimination a data byte is either represented by the byte itself (termed *literal*) or as part of a repetition. Each repetition is represented by its length and a back pointer (distance parameter). The repetition length is key since longer repetitions contribute to better compression ratio as well as to significantly better performance of compressors and decompressors. Note that the typical distribution of repetitions tends to follow a power-law [12], as observed in empirical tests (see Fig. 4 (left)). The majority of repetitions in such distributions are short ones (< 10 bytes) and consequently, compression algorithms exert effort in order to exploit these small repetitions, which in turn has an impact on performance. On the other hand we found that the effect of repetition distances on compression ratio and time is minor[3].

Entropy encodings utilize non-uniformity of byte level frequencies to encode data in a more compact representation at the bit level. In essence, the encoding associates bit level identifiers to bytes so that the most frequent bytes are represented by the shorter identifiers, saving storage space. This process is mimicked by capturing the distribution of bytes during the scan process. As we observe in Fig. 4 (right), the skew in the distribution of byte frequency changes significantly from text files to random-like data (PDFs). This has a strong impact on compressibility and may also impact the encoding process speed. These observations guided the design of our mimicking method for compression algorithms.

4.2 Data Characterization

To capture the aforementioned data characteristics, in our method every Chunk Characterization (CC) contains:
Byte frequency histogram. We build a histogram that relates the bytes that appear in a data chunk with their frequencies, encoding it as a `<byte, frequency>` map that we use to generate synthetic data that mimics this byte distribution. This information is key to emulate the entropy of the original data, among other aspects.
Repetition length histogram. Our aim is to mimic the distribution of lengths of repetitions as they would be found by a compressor. Note that different compressors will find different repetitions, depending on how much they invest in this task. Since there is no absolute answer here, we take a representative example of a compressor

[3]An entire data window typically fits in the L1 cache and thus a longer distance does not incur a performance penalty.

(default `zlib`'s `Deflate` algorithm) and work according to repetitions found by this compressor. To encode the repetitions as a histogram, we use a map whose keys represent the length of repetitions found in a chunk and the values are frequencies of repetitions of a given length.
Compression ratio. Every CC also includes the compression ratio of the original data chunk. In the generation phase, *SDGen* will try to create a synthetic chunk with similar compressibility.

Note that the CC design only reveals statistical properties of data, but not data itself. This provides a high degree of data privacy, as we discuss in Section 8.

Characterization space complexity: In our method, the space complexity of a CC data structure is *bounded* irrespective of the scan chunk size. Specifically, to represent a data chunk every CC contains 2 histograms whose data is stored in a map data structure. In these maps, keys can be encoded in *byte* data type (repetition length, bytes) whereas values are expressed as integers (32-bit integer). Therefore, in the worst case a single map requires $1,280$ bytes (256 keys $\cdot$ 4 bytes/value). In addition, we add the compression ratio (64-bit double), as well as the inherited fields from `ChunkCharacterization` (*size*, 32-bit integer and *seed*, 64-bit long).

The data chunk size can be arbitrarily large and can be chosen according to the application at hand (see more in Section 5). As a rule, we suggest to make it at least as long as the size of a compressor's compression window (e.g., 32-128KB are typical granularities). Altogether, a CC consumes at most 3.93% the space of a 64KB chunk.

4.3 Synthetic Content Generation

In order to generate `dataSize` bytes of synthetic data (`synData`), we sequentially pick CCs from the list contained in the dataset characterization. The scanned features in every CC are the input values for our data generation algorithm, which is described in Algorithm 1.

Algorithm 1 generates synthetic data that mimics some key properties of the original data (see Section 4.2):
Byte frequency: Algorithm 1 generates both unique and repeated sequences with the function `randomData` that outputs random bytes based on the histogram of byte frequencies extracted from the original chunk (`byteFrq`).
Repetition length: We insert both random and repeated data in sequences of length `seqLen`, whose values are drawn by the repetition length histogram of the original chunk (`repLenFrq`$\rightarrow$`seqLengths`). Normally, to generate repeated data we use a single sequence (`repSeq`) of length *MAX_LEN*. Thus, every time we need to insert a repetition, we select the first `seqLen` bytes of `repSeq`.
Compression ratio: For mimicking compressibility, Algorithm 1 interleaves repeated or unique sequences of bytes based on random trials (line 19) against the normalized compression ratio (`cr`) of the original chunk[4].

Algorithm 1: High-level data generation algorithm

```
   Data: dataSize, repLenFrq (Map), byteFrq (Map), cr
   Result: synData
1  $synData \leftarrow []$;
2  $uniqueBytes \leftarrow |byteFrq.keys()|$;
3  /*No need for renewal by default*/
4  $renewalRate \leftarrow \infty$;
5  $repCount \leftarrow 1$;
6  $i \leftarrow 0$;
7  /*Special treatment for extreme data types*/
8  if $uniqueBytes < MIN\_BYTES$ then
9      $renewalRate \leftarrow uniqueBytes$
10 /*Initialize repetition and set it as prefix*/
11 $repSeq \leftarrow randomData(byteFrq, MAX\_LEN)$;
12 /*Fill the repetitions distribution list*/
13 $seqLengths \leftarrow getDescOrderSeqLen(repLenFrq)$;
14 $previousWasRep \leftarrow False$;
15 while $i < dataSize$ do
16     if $seqLengths = []$ then
17         $seqLengths \leftarrow$
           $getDescOrderSeqLen(repLenFrq)$;
18     $seqLen \leftarrow seqLengths.popFirst()$;
19     if $randomTrial() < 1/cr$ then
20         $synData[i : i + seqLen] \leftarrow$
           $randomData(byteFrq, seqLen)$;
21         $previousWasRep \leftarrow False$;
22     else
23         /*Break to avoid repetition concatenation*/
24         if $previousWasRep$ then
25             $synData[i] \leftarrow randomData(byteFrq, 1)$;
26             $i \leftarrow i + 1$;
27         /*Add repeated data*/
28         $synData[i : i + seqLen] \leftarrow repSeq[0 : seqLen]$;
29         /*Renew repetition if necessary*/
30         if $(repCount \textbf{ mod } renewalRate) = 0$ then
31             $repSeq \leftarrow$
               $randomData(byteFrq, MAX\_LEN)$;
32         $repCount \leftarrow repCount + 1$;
33         $previousWasRep \leftarrow True$;
34     $i \leftarrow i + seqLen$;
```

Algorithm 1 includes several implementation nuances resulting from our empirical insights. First, this algorithm generates batches of repeated/unique byte sequences that are appended to the synthetic chunk in decreasing order by length (`getDescOrderSeqLen` in lines 13, 17). This choice allows algorithms with light repetition search (e.g. `lz4`) to find the correct synthetic repetitions. We empirically found that compressors with deeper repetitions search are insensitive to such ordering.

[4]Since we employ the `zlib` repetition finding algorithm, we also make use of the `zlib` compression ratio in our characterization. `zlib` sets the maximum repetition length at 258 (*MAX_LEN*, Algorithm 1).

Second, we avoid the concatenation of several repetitions by adding a random separator byte after every repetition (line 25). The reason is that two consecutive repetitions of the same length may occur more than once. Given that we normally use a single sequence as a source of repetitions (`repSeq`), these concatenations would be interpreted as a single longer repetition.

Note that our method is susceptible to mutual effects caused by the interplay between repetitions and byte distribution. Repetitions can slightly skew the byte distribution while a short alphabet can also affect the repetition counts (typically making them longer). In general, our evaluation showed that such these interplays have a minor effect on other metrics. An exception is with data formed with a very short alphabet (e.g. DNA sequencing)[5] which require special treatment. Algorithms like `zlib` exhibit degraded results on such data, probably because the internal Huffman tree should be constantly updated for very short sequences. To overcome this problem, we refresh the repeated sequence often during the generation process (`renewalRate`), to degrade the performance of compressors as in the original data (line 9).

Albeit simple, in Section 6 we show that our method provides an attractive trade-off between characterization complexity and accuracy, for disparate datasets. We also show that the accuracy of our synthetic data, in terms of compression ratios and times, is because it mimics the key properties defined in our characterizations (e.g., repetition lengths, byte distribution).

5 Experimental Setup

For clarity, our evaluation is divided into two parts: i) evaluation of the accuracy of the synthetic data generated by *SDGen* (Section 6), and ii) analysis of the benefits of *SDGen* integrated with real benchmarks (Section 7).

Methodology. To quantify the accuracy of the synthetic data that *SDGen* generates, we proceeded as follows. First, we scanned an original dataset in fixed size chunks (32KB) to build a full characterization file. For practicality, the scan process was done over a single `.tar` file containing all the files of a dataset. Subsequently, we generated a synthetic file as large as the original one.

Then, we analyzed the behavior of compression engines (compression ratios, times) on a *dataset and per-chunk basis*. For inspecting chunks, we instantiated a fresh compressor object to digest every data chunk. Compressors were executed sequentially, to avoid artifacts and interferences in compression times. This fine-grained perspective enabled us to capture the heterogeneity of datasets and the reaction of compression engines. Dataset compression times are averages of 30 executions.

We compared *SDGen* with LinkBench data generation, which is a representative case of solutions to generate data with predefined compressibility by simply mixing compressible/incompressible sequences (see Table 1). To this end, we used `zlib` to obtain the compression ratio of the original chunks. We then generated similarly compressible data chunks with LinkBench, using its default data generation mechanism. Note that this goes far beyond the standard implementation, which targets a preconfigured mean data compressibility.

Setting. The evaluation was performed using a server running a Debian 7.4 operating system, equipped with an i5-3470 processor (4 cores), 8GB DDR-3 memory and a HDD of $7,200$ rpm and 1TB of storage capacity. Since SSDs are becoming increasingly popular for databases, in the integration tests of LinkBench we used a Samsung 840 SSD with 250GB of storage capacity.

We ran our sampling experiments on an Ubuntu 14.04 server equipped with an Intel Xeon x5570 processor (4 cores) with 8GB RAM. We read the files from a local disk, compressed them and stored them to an 8-disk raid-10 array (10K RPM SAS drives), via a 4Gbit FC port.

5.1 Compression Similarity Metrics

We evaluate the accuracy of our synthetic generation method by targeting several metrics. First and foremost, we aim to hit the two main parameters that are relevant to the performance of a system:

- **Compression ratio**: This metric refers to the ratio between the size of the original data to the size of the compressed data. Thus, given a compression algorithm, we want that a chunk of synthetic data will be as compressible as the original data chunk.
- **Compression time**: This metric captures the computation time taken by a compression algorithm to compress a single data chunk.

We also analyze two properties that helped us to devise our generation algorithm. They can serve as good indicators to the potential success of *SDGen* in mimicking data for other compressors that are not tested here:

- **Repetition length**: We compare the length of repetitions in both the original and synthetic data.
- **Entropy**: It is often associated with the compressibility of data [18] and quantifies how uniformly distributed the bytes are within a data chunk. From a sample $X = \{x_0, ..., x_n\}$ of byte values with a probability mass function $P(X)$, we calculate its entropy $H(X) = -\sum P(x_i) log_2 P(x_i)$ [30]. In a byte basis, an entropy value of 8 (highest) means that the data is completely random and therefore not compressible. A value of 0 means the opposite.

In addition, we performed several experiments over ZFS, a well-known file-system with built-in compression to measure transfer throughput. These experiments helped us to compare and evaluate the behavior of a real system when using synthetic/original data [23].

[5]We empirically found alphabet size 8 to be a good threshold value.

		Compression Ratio				*Compression Time (secs.)*			
Dataset	*Source*	`zlib`	`lz4`	`bzip2`	`lzma`	`zlib`	`lz4`	`bzip2`	`lzma`
Calgary C.	OR	3.30	1.80	4.03	4.25	1.128	0.082	1.692	10.251
	SD	3.15(−4.5%)	1.82(1.26%)	3.10(−23.29%)	3.58(−15.76%)	1.096(−2.8%)	0.072(−12.19%)	1.554(−8.15%)	9.064(−11.67%)
	LB	3.70(12.3%)	2.79(55.39%)	3.8(5.48%)	3.71(−12.72%)	0.329(−70.8%)	0.043(−47.57%)	3.113(83.98%)	4.350(−57.56%
PDFs	OR	1.16	1.15	1.15	1.19	1.431	0.109	6.554	15.758
	SD	1.13(−2.59%)	1.10(−4.34%)	1.10(−4.34%)	1.12(−5.88%)	1.551(8.38%)	0.117(7.33%)	6.514(−0.61%)	15.783(0.15%)
	LB	1.46(25.85%)	1.12(−2.61%)	1.47(27.82%)	1.46(22.69%)	1.658(15.86%)	0.106(−2.75%)	5.086(−22.39%)	18.526(17.56%)
Media	OR	1.01	1.00	1.01	1.01	10.809	1.971	40.18	113.38
	SD	1.00(−0.99%)	1.00(0%)	1.00(−0.99%)	1.00(−0.99%)	11.083(2.53%)	1.960(0.56%)	43.02(7.07%)	110.895(2.19%)
	LB	1.30(28.71%)	1.00(0%)	1.30(28.71%)	1.29(27.72%)	11.601(7.32%)	1.941(1.52%)	30.12(−25.04%)	129.87(14.54%)
Silesia C.	OR	3.11	2.08	3.88	4.31	8.214	0.902	21.022	102.82
	SD	2.82(−9.22%)	2.03(−2.40%)	2.73(−29.63%)	3.32(−22.97%)	8.826(7.45%)	0.833(−7.65%)	19.200(−8.67%)	82.33(−19.92%)
	LB	3.57(14.79%)	2.69(29.32%)	3.65(−5.93%)	3.56(−17.40%)	4.002(−51.27%)	0.476(−47.23%)	35.202(67.45%)	52.07(−49.35%)
Sensor	OR	4.86	2.88	5.93	8.33	55.014	17.795	121.03	1238.3
	SD	4.52(−6.99%)	2.70(−6.25%)	4.80(−19.05%)	5.67(−31.93%)	53.799(−2.21%)	16.643(−6.47%)	145.65(20.30%)	909.2(−26.57%)
	LB	5.32(9.46%)	3.99(38.54%)	5.54(−6.57%)	5.34(−35.89%)	26.659(−51.54)	12.486(−29.83)	304.96(151.9%)	381.6(−69.18%)

OR=Original Dataset, **SD**=*SDGen* Synthetic Data, **LB**=LinkBench Synthetic Data. Relative errors compared to **OR** appear in parentheses.

Table 2: Dataset-level compression ratios and times of *SDGen* and LinkBench data (non-sampling datasets).

5.2 Datasets and Compression Engines

Next, we briefly describe the datasets used to assess the accuracy of our data generation method. Note that we stress the importance for a method to be accurate in the presence of diverse and heterogeneous datasets.

- **Calgary/Canterbury corpus (Text)**: Collection of text and binary data files, commonly used for comparing data compression algorithms (18.5MB).
- **PDFs**: Proceedings of the last 5 editions of the Usenix FAST conference (48.7MB).
- **Silesia corpus**: Standard set of files that covers the typical data types used nowadays [14] (211.9MB).
- **Media**: Media files collected from the home directories of 4 IBM engineers including photos (`.jpg`), music (`.mp3`) and video (`.avi`) (300.3MB).
- **Sensors dataset**: GPS trajectory dataset collected in Microsoft Research Geolife project by 182 users during three years [40](1.7GB).
- **Mix** (*sampling test only*): A private collection of mix of files of various data types e.g. html, xml, txt, database files and VM images (14GB).
- **Enwiki9** (*sampling test only*): Common measuring stick for compression methods consisting of the first 10^9 bytes of the English Wikipedia [37].

We measured the accuracy of our synthetic data mimicking these datasets by analyzing the behavior of 4 compression engines: `lz4`, `zlib` (level 1, 6), `bzip2` and `lzma`. We used the compressors' default implementation available on Unix distributions and the analogous Java libraries included in *SDGen*. It is worth mentioning that these algorithms belong to *different families and adopt disparate heuristics* to find redundancies within a data stream. This can give a sense about the universality of our generation method. Specifically, `lz4` targets speed over compression ratio and has repetition elimination only. `zlib` is medium speed, adopting Huffman encoding in addition. `bzip2` and `lzma` target compression ratio over speed deploying various advanced, yet time consuming methods such as the Burrows Wheeler transform [24] (`bzip2`) or a large dictionary based variant of LZ77 (`lzma`) .

6 Evaluation

Next, we describe the results comparing original and synthetic (*SDGen*, LinkBench) data using the aforementioned metrics. We compare the results of two compression engines that we *specifically target* (`zlib`, `lz4`) with other two of distinct families (`bzip2`, `lzma`).

6.1 Compression ratio

In Table 2 and Fig. 5, we compare the obtained compression ratios of the original and synthetic datasets for all compression engines at both dataset and chunk levels.

Table 2 shows that *SDGen* closely mimics the compressibility of real datasets for the targeted engines (`zlib`, `lz4`). For `zlib` and `lz4`, *SDGen* does not deviate more than 10% in compression ratio compared to the real data. This demonstrates that our synthetic data is *sensitive to the algorithm*; that is, our synthetic data exhibits the same behavior than the real data, depending on the compression engine used. This confirms that analyzing the *structure of data* is an appropriate approach to generate realistic synthetic data.

At the dataset level, we observe that *SDGen* is less accurate for the non-targeted compressors (`bzip2`, `lzma`). Surprisingly, this contrasts with Fig. 5 that shows how *SDGen* closely reproduces the compression ratio distributions of real dataset chunks. The reason for this behavior is related with the *chunk size*.

That is, `zlib` and `lz4` digest data in small chunks (e.g. 32KB) to reduce the size of their internal data structures. Conversely, `bzip2` and `lzma` digest data in window sizes that can reach 900KB and 1GB, respectively [23]. We do not focus on scanning data features in this broader scope. This encourages us to research new scanners at larger granularities to cope with other compressors.

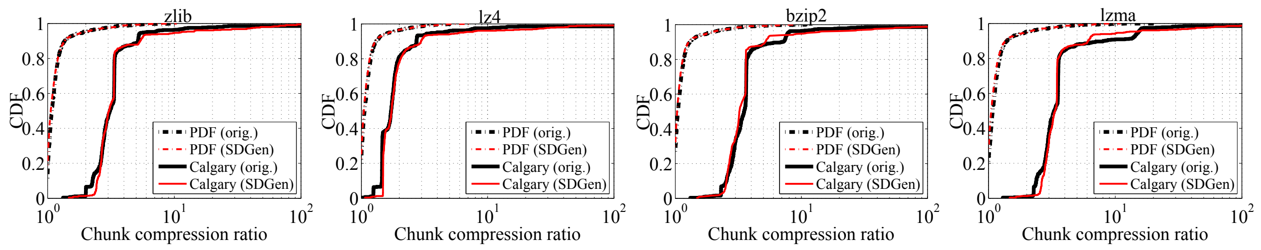

Figure 5: Original vs *SDGen* per-chunk compression ratio distributions for various datasets and compressors.

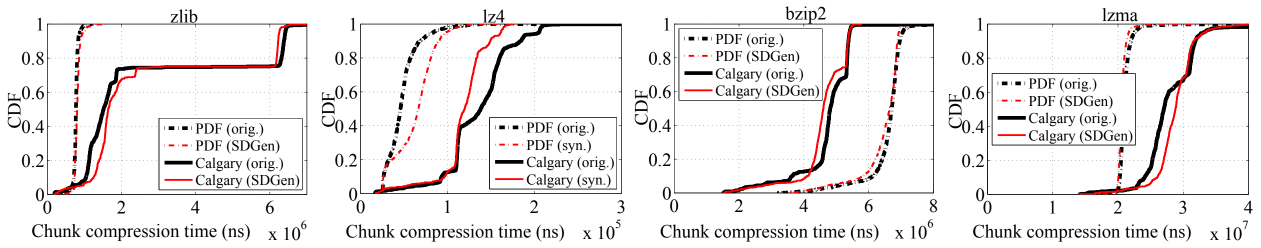

Figure 6: Original vs *SDGen* per-chunk compression time distributions for various datasets and compressors.

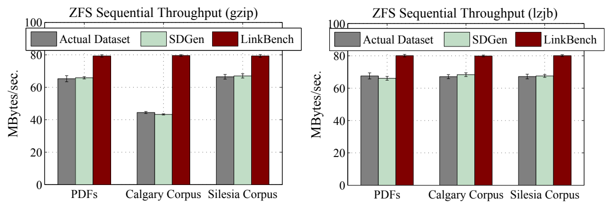

Figure 7: Sequential write throughput of ZFS depending on data type. Clearly, ZFS behaves similarly when processing *SDGen* synthetic data and the original one.

In general, the one-dimensional data generation approach of LinkBench deviates importantly from the compressibility of the actual data, even though LinkBench datasets were generated chunk-by-chunk to capture the heterogeneity (Table 2). The reason is that the proportion of compressible/incompressible data is determined by one algorithm in the generation phase (`zlib`). Such a synthetic data becomes inaccurate when compressed by other engines, compared to the original dataset.

6.2 Compression time

Table 2 shows that *SDGen* achieves high accuracy mimicking compression times for `zlib` and `lz4`. At the dataset level, our synthetic data does not deviate by more than 13% in compression times compared to the original data. At chunk level, on average the 70% of *SDGen* chunks deviate less than 20% in terms of compression time (Fig. 6) w.r.t. the real dataset —it is harder to be accurate for `lz4` since it is the fastest compressor. Table 2 also illustrates that for the non-targeted compressors (`bzip2`, `lzma`) our synthetic data does not deviate by more than 26%, which we consider acceptable.

Note that in the Calgary corpus there is a particular file that accounts for the 23% of samples, which exhibit a much higher compression time for `zlib` (Fig. 6, left). This file is the DNA sequencing of the E. coli bacteria. The particularity of this file is that it is formed by very few distinct bytes (4 in most chunks, since DNA sequences are composed by 4 nucleotides) and very short repetitions. This makes compression algorithms that use Huffman codes to perform worse, since the Huffman tree should be constantly updated for only very short sequences. Our generation algorithm detects these situations and reacts by increasing the repeated sequence renewal rate. Thus, the performance of Huffman codes becomes also worse, similar to the original data.

Unsurprisingly, Table 2 shows that LinkBench datasets deviate importantly from the compression times of real datasets. Such a deviation —in many cases higher than 50%— may induce important impact on the performance of a storage system with built-in compression.

6.3 Performance of ZFS

We want to stress the importance of using an appropriate data generation method in a real system. Thus, we augmented 3 datasets (PDFs, Calgary and Silesia) by replicating them to be 1GB in size. Then, we copied 50 times each from memory into a ZFS partition with compression enabled (`lzjb`, `gzip`) capturing the sequential write throughput. We repeated the experiment with two synthetic datasets: i) a dataset generated with LinkBench creating chunks of the same compressibility than the original one, and ii) a dataset generated with our method.

In Fig. 7, our synthetic data makes the system to exhibit virtually the same write throughput as the original dataset. That is, the difference in throughput of ZFS between the original dataset and our data is at most 1.9% in average for `lzjb` and `gzip`. However, considering datasets generated with LinkBench, ZFS exhibits a variation in write throughput between +12.5% and +19% in most cases w.r.t. the original dataset. Interest-

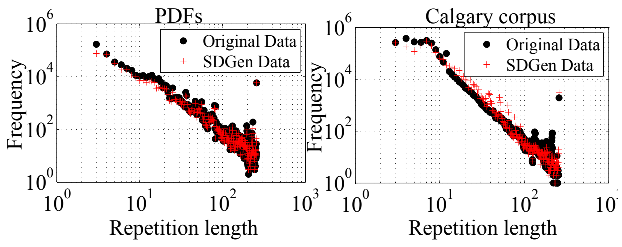

Figure 8: Repetition length distributions of *SDGen* and original datasets.

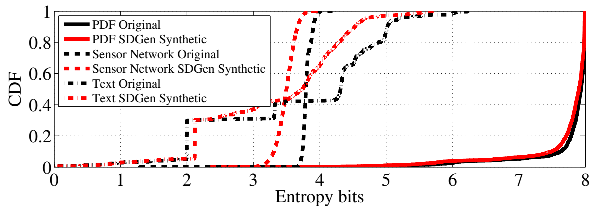

Figure 9: Entropy CDF of *SDGen* and original data.

ingly, irrespective of the dataset, ZFS achieves a similar throughput writing LinkBench data. The reason is that LinkBench generates data that is easy to compress for most algorithms, for a wide range of compression ratios.

`gzip` (as `zlib` in Fig. 6) performs worse digesting the Calgary corpus. Our method handles this behavior, whereas the LinkBench data makes ZFS write throughput to deviate by $+44\%$ compared to the original dataset.

Therefore, compared with current approaches, our method provides a much more realistic substrate to benchmark storage systems with compression built-in.

6.4 Similarity of Mimicked Characteristics

To emulate compression ratios and times, our mimicking method captures the *repetition length distribution* and the *frequencies of bytes* of the original data (Section 4). Now, we inspect how close *SDGen* mimics these properties.

Repetition length. Fig. 8 shows the distribution of repetition lengths for both original and synthetic datasets (PDFs, text). We observe that the repetitions in both cases are similar in terms of distribution shape and absolute frequency numbers. This characteristic plays a key role on the accuracy of the synthetic chunks compression ratios and times, suggesting that mimicking it is an effective way of generating realistic data for compression.

Entropy. In Fig. 9 we depict the entropy distribution of original and synthetic chunks for two datasets. As we can infer, the entropy distribution of the original dataset is roughly followed by the synthetic one. The reason for this is that we capture the byte histogram distribution in original chunks to generate bytes according to it. This property is also interesting because our synthetic data would be useful for techniques that estimate the compressibility of data based on entropy [18].

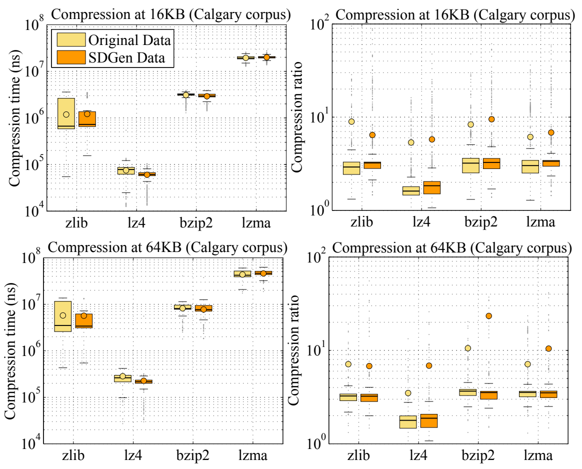

Figure 10: Compression times and ratios of 16KB and 64KB synthetic chunks mimicking Calgary Corpus data. Original data was scanned at 32KB chunk granularity (boxplot circles represent mean values).

6.5 Chunk Size Sensitivity

We obtained very similar results when varying the chunk size from 8KB to 128KB, which are the typical size limits for compression (16KB for MySQL, 8KB to 128KB for ZFS). However, as the scan chunk size gets smaller, the characterization file grows linearly. Normally, compression algorithms tend to avoid small window sizes since data compressibility decays. Thus, we recommend to scan data in chunks of 16KB to 64KB.

In *SDGen* we scan a dataset and generate data with a configurable chunk size. A question that arises here is: *how does the synthetic data behave when it is compressed to different granularities than the one used in the scan/generation process?* To answer this question, we compressed the Calgary corpus and the synthetic dataset at 16KB and 64KB granularities (Fig. 10). Note that the synthetic dataset has been scanned/generated at 32KB granularity. We selected the Calgary corpus since it is very heterogeneous (compression times, ratios), potentiating differences depending on the scan granularity.

Interestingly, in Fig. 10 we observe that our method is not sensitive to the scan granularity. That is, for both scan chunk sizes, the compression times and ratios of compression algorithms follow a similar trend. Although the distribution tails are harder to model, we observe that in most cases the boxplots for both datasets present a similar shape. Therefore, we conclude that we can safely mimic a dataset using granularities that are different than the one used during the original data scan phase, while maintaining the original content behavior.

6.6 Sampling: Scaling Characterizations

Previously, we performed full scans on the original data (32KB chunks). Full dataset scans let us reproduce the compressibility of data, and even the locality of compres-

Dataset	Compressor	Compression Time (sec.)				Decompression Time (sec.)				Compression Ratio			
		Orig.	*SDGen*	Zero	Rand.	Orig.	*SDGen*	Zero	Rand.	Orig.	*SDGen*	Zero	Rand.
Mix	`gzip-6`	443.47	451.98	136.48	589.82	97.30	95.74	72.52	95.64	1.86	1.82	1030.5	1.00
	`gzip-1`	378.28	380.45	136.34	574.46	88.30	87.21	48.64	96.32	1.84	1.82	229.3	1.00
	`lz4`	161.82	143.21	137.15	139.93	42.11	43.50	5.98	66.63	1.79	1.78	254.7	1.00
Enwiki9	`gzip-6`	50.31	57.16	9.41	39.32	8.20	9.05	4.84	6.32	3.09	2.83	1030.5	1.00
	`gzip-1`	22.08	22.92	9.45	37.93	9.18	9.27	6.05	3.23	2.64	2.54	229.3	1.00
	`lz4`	10.00	9.60	9.40	9.39	3.20	2.93	0.41	3.92	1.97	1.91	254.7	1.00

Orig=Original Dataset, **SDGen**=*SDGen* Dataset, **Zero**=Zero Dataset, **Rand**=Random Dataset.

Table 3: Dataset-level compression ratios and times of *SDGen* data using sampling in the scan process.

sion with high precision. In fact, this can be achieved with moderate characterization space requirements. For instance, compared to the original datasets, the size of characterization files are 4.08% (Silesia corpus), 2.14% (Calgary corpus), and 6.38% (PDFs) —the theoretical maximum is 7.86% for a 32KB chunk. However, considering large datasets, the size of characterizations and the memory requirements for the scan may be too high.

Next, we want to inspect the accuracy of our synthetic data when we only use a subset of data in the scan process. To this end, we make use of real datasets that are large enough to justify the use of sampling. As described in Section 3.3, we use characterizations formed by $3,500$ samples. We also tested a larger chunk size (128KB).

Table 3 shows the compression/decompression times, as well as the compression ratios for the real and synthetic datasets. "Random" and "Zeros" are datasets as large as the real one, whose content is self-explanatory.

In general, we see in Table 3 that our sampling approach is an effective way of mimicking large datasets. That is, *SDGen* datasets do not deviate more than 13.6% and 10% in compression and decompression times, respectively. *SDGen* compression ratios are also accurate.

Interestingly, we find that decompression times are similar in both original and *SDGen* datasets; this suggest that if the synthetic data mimics correctly compression times, decompression times become also mimicked.

The most relevant point in this experiment is the size of the characterizations needed to achieve these results. That is, the Mix dataset (14GB) characterization file produced by *SDGen* was only 7.3MB in size (0.052%). We conclude that *SDGen* provides a novel and attractive way of sharing large datasets with very low data exchange, high mimicking accuracy and preserving data anonymity.

6.7 Data Generation Throughput

We evaluate the throughput of *SDGen* generating synthetic data with our method. First, *SDGen* utilizes the available cores to increase the generation throughput (Fig. 11). That is, making use of 4 cores instead of 1, the throughput of *SDGen* is x3.56 and x3.45 times higher in the case of text and media data, respectively.

Second, we noticed that the generation throughput varies depending on the data type being generated. This effect is caused by the behavior of our data generation method. That is, Algorithm 1 is faster generating highly compressible data since it reuses repetitions more frequently to generate synthetic chunks. In the case of random-like data, Algorithm 1 performs more random decisions to generate new bytes, which is slower than copying an existing sequence. Note that several *SDGen* servers can run in parallel to increase throughput.

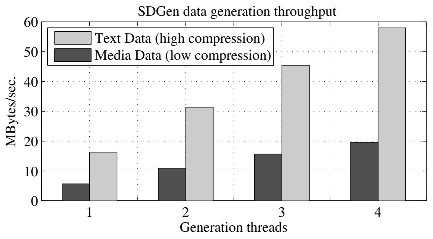

Figure 11: Throughput of our data generation algorithm integrated in *SDGen* depending on the data type.

7 Integration with Benchmarks

7.1 LinkBench

LinkBench [9] is a graph benchmarking tool developed by Facebook. LinkBench provides performance predictions on databases used for persistent storage of Facebook's production data. We integrated *SDGen* with LinkBench to explore the benefits of our synthetic data.

Integration. Internally, LinkBench permits to adjust the compressibility of data used in every experiment run. To decouple the actual data generation from the benchmark execution, LinkBench provides an interface called `DataGeneration`. We followed this contract by creating an adapter class which transforms LinkBench calls into API calls offered by our data generation system. This clean design permits a user to choose the way synthetic data is generated from the configuration file.

Setting. We executed LinkBench on top of MySQL 5.5 and ZFS with compression enabled (`lzjb` and `gzip`). We evaluated the differences in performance that are measured by LinkBench when query payloads are filled with realistic and non-realistic synthetic data given a target dataset (text, Calgary corpus). We filled query payloads with random offsets of datasets loaded in memory prior to the benchmark execution, to avoid the potential bias caused by the generation overhead.

We executed a write-dominated workload to observe the effects of content on insert latencies. We evaluated the performance of inserts due to their higher cost compared to reads, since they cannot be cached. We executed

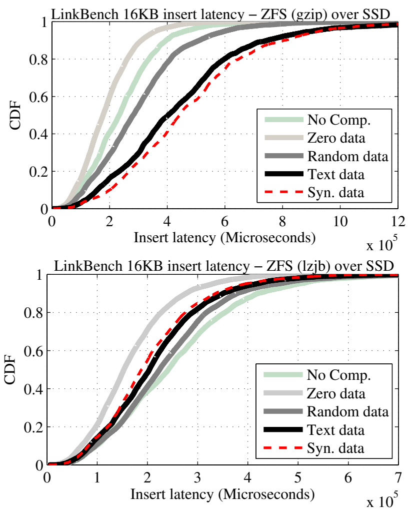

Figure 12: Insert latency of LinkBench for 16KB payloads on top of ZFS with `gzip` and `lzjb` compression.

LinkBench with 100 parallel threads for 30 minutes, resulting in various millions of inserts (18GB database). We used a Samsung 840 SSD as storage layer (250GB).

Results. In Fig. 12, we illustrate the insert latency that LinkBench experiences depending on the payload content. First, we want to emphasize the benefits of *SDGen* compared to use non-realistic data. Observably, the insert latency distributions are very similar for both the Calgary corpus and the corresponding *SDGen* synthetic dataset. This probes that *SDGen* produces *representative and reproducible data*, since the Calgary corpus characterization is ready to be shared and serve others.

Furthermore, we observe that employing naive contents may lead to disparate performance results. For instance, the median insert time of LinkBench is −55% filling payloads with zeros than using the corpus (`gzip`). `lzjb` also presents such performance variations, but they are less significant since the algorithm is much faster.

7.2 Impressions

File systems are an important field to apply data reduction techniques. Thus, we integrated our data generation system in the Impressions file system benchmark [7].

Integration. The integration has been done as follows. We set up a named pipe during the initial phase of the execution of Impressions to connect with *SDGen*. From that point onwards, Impressions delegates the generation of file contents to *SDGen* by writing in the pipe the size and the canonical path of the files to be created.

Additionally, we added a special type of dataset characterization (DC) in *SDGen* called *file system characterization* (Fig. 13). A file system characterization internally contains a set of regular DCs, each one associated to what we call a *file category*. File categories represent a group of file types —based on their extension— that usually contain similar contents. For instance, we

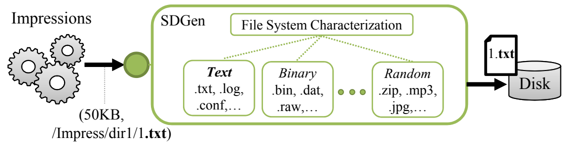

Figure 13: Integration of *SDGen* with Impressions.

can define a file category called "random data" containing compressed/encrypted files, such as `.zip`, `.mp3` and `.jpg`. We found this approach very convenient to treat the vast amount of existing file extensions, also produced by Impressions. We provide an initial set of file categories grouping the most popular file extensions[6].

To illustrate this, we recommend to see Fig. 13. In this figure, Impressions notifies *SDGen* that a new file called `/Impress/dir1/1.txt` of size 50KB should be created. *SDGen* looks up for the file extension `.txt`, which belongs to the "Text" file category. Subsequently, *SDGen* generates the file content resorting to the DC of this file category, which should be built by scanning representative files with the corresponding extensions.

Open Trial. We release a ready-to-use trial of *SDGen* integrated with Impressions. In the *SDGen* webpage, we release various DCs to be loaded into the file system characterization, which internally associates them to the appropriate file category. These DCs come from scanning file systems of our company engineers (text, images, etc.). We also provide the modified code of Impressions, as well as the execution instructions.

8 Discussion and Conclusions

A word on dataset privacy leakage: One of the main concerns in data sharing, or lack thereof, is privacy limitations on proprietary data. Our method relieves many of the privacy concerns and allows free sharing of data since *no actual data is shared*. More precisely, the data being created is a random combination of bytes, albeit with specific probabilities on byte occurrences and repetitions. It should be noted, however, that the characterizations are not entirely free of information. That is, the frequencies of bytes are revealed and these can actually tell us information about the data at hand. For instance, using such information one can distinguish the underlying *data type* (e.g. text, image).

Inherently, the mimicking approach is susceptible to this sort of "higher order information leakage" since the properties we are characterizing are *per se* an indication about the data at hand. However, we believe that this novel way of sharing provides an *attractive trade-off* between dataset privacy and benchmarking accuracy.

Beyond compression: In this paper we focused on compression related properties, but view *mimicking* as

[6] `http://en.wikipedia.org/wiki/List_of_file_formats`.

a general approach of data generation for benchmarking content-sensitive systems. One obvious extension is deduplication which is extremely data sensitive. Deduplication is more challenging for a number of reasons: (i) It is a *global dataset property*, rather than being dictated by local behavior of data. For this reason, fast scanning using sampling is much harder to achieve when deduplication is involved [19]; and (ii) deduplication ratios achieved in storage systems are typically affected by the *order and timing* in which data was written to the system (and not only by the content). Such a time based dependency is very hard to mimic. Our future research includes extending *SDGen* to also mimic data de-duplicability.

Conclusions: Most workload generators for benchmarks do not focus on the contents used in their execution, and they typically generate unrealistic data (zeros, random data). Storage systems with *built-in compression* behave differently on such naively-synthesized data than they do on real-world data. Current solutions create data with variable compression ratio, but they ignore other properties such as *compression time and heterogeneity*, which are critical to the performance of these systems.

We have therefore extended the basic methodology that underlies workload generators to the data itself. Current workload generators try to *mimic* real-world situations in terms of files, offsets, read/write balance and so on; we have designed and implemented an orthogonal component, called *SDGen*, to generate data that mimics the compressibility and compression times of real data.

For mimicking real-world data *SDGen* produces characterizations that are compact, sharable and essentially completely anonymized. We plan to release both *SDGen* and the characterizations that we have produced with it. We hope that others will use these tools and that others will share additional characterizations of data with the systems research community.

Acknowledgements

We first thank our shepherd Xiaosong Ma and anonymous reviewers. This work has been partly funded by the EU projects *CloudSpaces* (FP7-317555) and *IOStack* (H2020-644182) and Spanish research projects *DELFIN* (TIN-2010-20140-C03-03) and *Cloud Services and Community Clouds* (TIN2013-47245-C2-2-R) funded by the Ministry of Science and Innovation.

References

[1] Ssbench: Benchmarking tool for swift clusters. https://github.com/swiftstack/ssbench.

[2] Vdbench users guide. www.oracle.com/technetwork/server-storage/vdbench-1901683.pdf.

[3] Bonnie++. http://www.coker.com.au/bonnie++/, 2001.

[4] Fio. http://freecode.com/projects/fio, 2005.

[5] Filebench. http://sourceforge.net/projects/filebench/, 2008.

[6] ADIR, A., LEVY, R., AND SALMAN, T. Dynamic test data generation for data intensive applications. In *7th International Haifa Verification Conference, HVC 2011* (2011), pp. 219–233.

[7] AGRAWAL, N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Generating realistic impressions for file-system benchmarking. In *USENIX FAST '09* (2009), pp. 125–138.

[8] ANDERSON, E., KALLAHALLA, M., UYSAL, M., AND SWAMINATHAN, R. Buttress: A toolkit for flexible and high fidelity i/o benchmarking. In *USENIX FAST'04* (2004), pp. 4–4.

[9] ARMSTRONG, T. G., PONNEKANTI, V., BORTHAKUR, D., AND CALLAGHAN, M. LinkBench: a database benchmark based on the facebook social graph. In *ACM SIGMOD'13* (2013), pp. 1185–1196.

[10] BELL, T. Canterbury corpus. http://corpus.canterbury.ac.nz, 1997.

[11] BURROWS, M., JERIAN, C., LAMPSON, B., AND MANN, T. On-line data compression in a log-structured file system. In *ACM ASPLOS'92* (1992), pp. 2–9.

[12] CLAUSET, A., SHALIZI, C. R., AND NEWMAN, M. E. Power-law distributions in empirical data. *SIAM review 51*, 4 (2009), 661–703.

[13] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *ACM SoCC'10* (2010), pp. 143–154.

[14] DEOROWICZ, S. Silesia corpus. http://www.data-compression.info/Corpora/SilesiaCorpus/, 2003.

[15] DIFALLAH, D. E., PAVLO, A., CURINO, C., AND CUDRE-MAUROUX, P. OLTP-Bench: An extensible testbed for benchmarking relational databases. *VLDB Endowment 7*, 4 (2013).

[16] DRAGO, I., BOCCHI, E., MELLIA, M., SLATMAN, H., AND PRAS, A. Benchmarking personal cloud storage. In *ACM SIGCOMM IMC'13* (2013), pp. 205–212.

[17] GOLDBERG, G., HARNIK, D., AND SOTNIKOV, D. The case for sampling on very large file systems. In *IEEE MSST'14* (2014), pp. 1–11.

[18] HARNIK, D., KAT, R., SOTNIKOV, D., TRAEGER, A., AND MARGALIT, O. To zip or not to zip: Effective resource usage for real-time compression. In *USENIX FAST'13* (2013).

[19] HARNIK, D., MARGALIT, O., NAOR, D., SOTNIKOV, D., AND VERNIK, G. Estimation of deduplication ratios in large data sets. In *IEEE MSST'12* (2012), pp. 1–11.

[20] HUFFMAN, D. A. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers 40*, 9 (September 1952), 1098–1101.

[21] J. TATE, B. TUV-EL, J. Q. E. T., AND WHYTE, B. Real-time compression in SAN volume controller and Storwize V7000. Tech. rep., REDP-4859-00. IBM, 2012.

[22] LI, A., YANG, X., KANDULA, S., AND ZHANG, M. CloudCmp: comparing public cloud providers. In *ACM SIGCOMM IMC'10* (2010), pp. 1–14.

[23] LIN, X., LU, G., DOUGLIS, F., SHILANE, P., AND WALLACE, G. Migratory compression: coarse-grained data reordering to improve compressibility. In *USENIX FAST'14* (2014), pp. 257–271.

[24] MICHAEL BURROWS AND DAVID WHEELER. A block-sorting lossless data compression algorithm. Tech report.

[25] NIMBLE STORAGE. Nimble storage: Engineered for efficiency. Tech. rep., WP-EFE-0812, Nimble Storage, 2012.

[26] NORCOTT, W. D., AND CAPPS, D. IOzone filesystem benchmark. http://www.iozone.org.

[27] ORACLE. What is ZFS? http://docs.oracle.com/cd/E19253-01/819-5461/zfsover-2/.

[28] SCHMIDT, M., HORNUNG, T., LAUSEN, G., AND PINKEL, C. SP^{2}Bench: a SPARQL performance benchmark. In *IEEE ICDE'09* (2009), pp. 222–233.

[29] SELTZER, M., KRINSKY, D., SMITH, K., AND ZHANG, X. The case for application-specific benchmarking. In *USENIX HotOS'99* (1999), pp. 102–107.

[30] SHANNON, C. E. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review 5*, 1 (2001), 3–55.

[31] TARASOV, V., BHANAGE, S., ZADOK, E., AND SELTZER, M. Benchmarking file system benchmarking: It *IS* rocket science. *USENIX HotOS'11* (2011).

[32] TARASOV, V., KUMAR, S., MA, J., HILDEBRAND, D., POVZNER, A., KUENNING, G., AND ZADOK, E. Extracting flexible, replayable models from large block traces. In *USENIX FAST'12* (2012), p. 22.

[33] TARASOV, V., MUDRANKIT, A., BUIK, W., SHILANE, P., KUENNING, G., AND ZADOK, E. Generating realistic datasets for deduplication analysis. In *USENIX ATC'12* (2012), pp. 1–12.

[34] TAY, Y. Data generation for application-specific benchmarking. *VLDB, Challenges and Visions* (2011).

[35] TRAEGER, A., ZADOK, E., JOUKOV, N., AND WRIGHT, C. P. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS) 4*, 2 (2008), 5.

[36] WANG, L., ZHAN, J., LUO, C., ZHU, Y., YANG, Q., HE, Y., GAO, W., JIA, Z., SHI, Y., ZHANG, S., ZHENG, C., LU, G., ZHAN, K., LI, X., AND QIU, B. BigDataBench: A big data benchmark suite from internet services. In *IEEE HPCA'14* (2014), pp. 488–499.

[37] WIKIPEDIA FOUNDATION. English wikipedia. https://dumps.wikimedia.org/, 2014.

[38] ZHANG, Y., RAJIMWALE, A., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. End-to-end data integrity for file systems: A zfs case study. In *USENIX FAST'10* (2010), pp. 29–42.

[39] ZHENG, Q., CHEN, H., WANG, Y., DUAN, J., AND HUANG, Z. COSBench: A benchmark tool for cloud object storage services. In *IEEE CLOUD'12* (2012), pp. 998–999.

[40] ZHENG, Y., ZHANG, L., XIE, X., AND MA, W.-Y. Mining interesting locations and travel sequences from GPS trajectories. In *WWW'09* (2009), pp. 791–800.

[41] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on information theory 23*, 3 (1977), 337–343.

[42] ZIV, J., AND LEMPEL, A. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory 24*, 5 (September 1978), 530–536.