Frontiers of Information Technology & Electronic Engineering
www.zju.edu.cn/jzus; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

TextGen: a realistic text data content generation method for modern storage system benchmarks[*]

Long-xiang WANG[1], Xiao-she DONG[1], Xing-jun ZHANG[‡1],
Yin-feng WANG[2], Tao JU[1], Guo-fu FENG[3]
([1]*School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China*)
([2]*Shenzhen Institute of Information Technology, Shenzhen 518172, China*)
([3]*College of Information Technology, Shanghai Ocean University, Shanghai 201306, China*)
E-mail: wanglongxiang@stu.xjtu.edu.cn; xsdong@mail.xjtu.edu.cn; xjzhang@mail.xjtu.edu.cn;
wangyinfeng@gmail.com; jutao2011@stu.xjtu.edu.cn; jt_f@163.com

Received Oct. 13, 2015; Revision accepted Mar. 21, 2016; Crosschecked Sept. 18, 2016

Abstract: Modern storage systems incorporate data compressors to improve their performance and capacity. As a result, data content can significantly influence the result of a storage system benchmark. Because real-world proprietary datasets are too large to be copied onto a test storage system, and most data cannot be shared due to privacy issues, a benchmark needs to generate data synthetically. To ensure that the result is accurate, it is necessary to generate data content based on the characterization of real-world data properties that influence the storage system performance during the execution of a benchmark. The existing approach, called SDGen, cannot guarantee that the benchmark result is accurate in storage systems that have built-in word-based compressors. The reason is that SDGen characterizes the properties that influence compression performance only at the byte level, and no properties are characterized at the word level. To address this problem, we present TextGen, a realistic text data content generation method for modern storage system benchmarks. TextGen builds the word corpus by segmenting real-world text datasets, and creates a word-frequency distribution by counting each word in the corpus. To improve data generation performance, the word-frequency distribution is fitted to a lognormal distribution by maximum likelihood estimation. The Monte Carlo approach is used to generate synthetic data. The running time of TextGen generation depends only on the expected data size, which means that the time complexity of TextGen is $O(n)$. To evaluate TextGen, four real-world datasets were used to perform an experiment. The experimental results show that, compared with SDGen, the compression performance and compression ratio of the datasets generated by TextGen deviate less from real-world datasets when end-tagged dense code, a representative of word-based compressors, is evaluated.

Key words: Benchmark, Storage system, Word-based compression
http://dx.doi.org/10.1631/FITEE.1500332 **CLC number:** TP311.1

1 Introduction

Benchmarking is an important method for measuring the performance of storage systems, and has long been an important research area for the storage community. A large volume of work has been carried out in this area, including benchmarks for file systems (Agrawal *et al.*, 2009; Tarasov *et al.*, 2011), cloud storage (Cooper *et al.*, 2010; Li *et al.*, 2010; Drago *et al.*, 2013), and databases (Armstrong *et al.*, 2013; Difallah *et al.*, 2013).

To obtain accurate performance results, a benchmark simulates the I/O behaviors of real-world applications. Because real-world proprietary datasets are too large to be copied onto a test storage system, and because most data cannot be shared due to privacy issues, a benchmark synthetically generates data

[‡] Corresponding author
[*] Project supported by the National Natural Science Foundation of China (Nos. 61572394 and 61272098), the Shenzhen Funda mental Research Plan (Nos. JCYJ20120615101127404 and JSGG20140519141854753), and the National Key Technologies R&D Program of China (No. 2011BAH04B03)
ORCID: Xing-jun ZHANG, http://orcid.org/0000-0003-1434-7016

that are used to perform I/O operations under simulated workloads. However, research on storage benchmarks has focused mainly on how to simulate storage workloads (Anderson *et al.*, 2004; Chilan, 2005; Traeger *et al.*, 2008), whereas studies on data generation have not received enough attention. Agrawal *et al.* (2009) first pointed out the importance of generating data for benchmarks, and proposed a method focusing mainly on metadata generation. The metadata included the directory structure, the number of directories, the size of files, and the number of files. Experimental results showed that the metadata significantly influenced the accuracy of benchmark results.

However, the 'impressions' solution proposed by Agrawal *et al.* (2009) generates unrealistic data content. Gracia-Tinedo *et al.* (2015) pointed out that data content significantly influences the benchmark results in modern storage systems, such as the B-tree file system (BTRFS) (Rodeh *et al.*, 2013) and Zettabyte file system (ZFS) (Bonwick *et al.*, 2003). These systems compress data before storing it to improve their performance and capacity. The data content significantly affects the compression performance and the compression ratio, as well as the performance and space consumption of these storage systems. Thus, these storage systems are called content-sensitive. The data generated by 'impressions' (Agrawal *et al.*, 2009) cannot obtain accurate benchmark results in content-sensitive storage systems. To address this problem, researchers have proposed SDGen (Gracia-Tinedo *et al.*, 2015), which is an open and extensible framework to generate realistic storage benchmarking content. SDGen captures the byte-level properties that influence the compression performance of real-world data, and uses them to create a characterization file, which saves the byte frequency and repetition, and is used to generate synthetic data content. The characterization file guarantees that the byte frequency and repetition of synthetic data content are close to those of real-world datasets, and can be shared by other researchers or practitioners to obtain a reproducible benchmark result. By using the data generated by SDGen, the storage system's benchmark result is accurate when the byte-based Ziv-Lempel family (Ziv and Lempel, 1977) compressors are enabled. Thus, SDGen deviates from real-world data by less than 10% in compression ratio and less than 20% in compression performance.

However, for other text-data-oriented word-based compressors, such as end-tagged dense code (ETDC) (Brisaboa *et al.*, 2003) or the word-based Lempel-Ziv-Welch (LZW) (Horspool and Cormack, 1992), the data generated by SDGen cannot ensure that the compression ratio and compression performance results are accurate. This is because SDGen generates data content based on the byte-level characterization of real-world data. SDGen lacks word-level characterizations, which are the basic factors that influence the compression ratio and compression performance of word-based compressors. Text is an important data type in storage systems. Agrawal *et al.* (2007) analyzed the datasets of over 10 000 file systems on Windows desktop computers at the Microsoft Corporation over five years. The results showed that the text data types, cpp, html, h, and txt, accounted for 7%, 5%, 3%, and 3% of data, respectively, and this indicated that text comprises a high proportion of data in the common working environment. A document database, which is also a common storage system, has been widely used in various applications. It contains vast quantities of text data generated endlessly by newspaper reporters, academics, lawyers, and government agencies. As described above, text data are extremely important in common storage systems. Compared with the traditional Ziv-Lempel family of compressors, word-based compressors can achieve higher compression performance and compression ratios with text data, and have been the subject of many studies (Yoshida *et al.*, 1999; Brisaboa *et al.*, 2003; 2007; 2010). A large number of storage systems use word-based compressors to improve the data compression ratio and the performance (Moffat *et al.*, 1997; Dvorský *et al.*, 1999; Fariña *et al.*, 2012). Thus, research on generating text data content more accurately at the word level is meaningful for modern storage system benchmarks.

To address the problem that the data content generated by existing methods is inaccurate at the word level, we present TextGen, a realistic text data content generation method for modern storage system benchmarks. The key idea behind TextGen is to capture the word-level properties that influence the compression performance and compression ratio of word-based the compressors, and to use these characterizations to generate content. Moreover, to improve the performance of content generation, the word-frequency distribution is fitted to a lognormal distribution using maximum likelihood estimation (MLE).

To compare the accuracy of compression performance and the compression ratio between SDGen and TextGen using a word-based compressor, we used the following four text datasets to conduct an experiment: Java source code, C source code, Calgary, and 20News. ETDC was used in the experiment as a representative word-based compressor. Experimental results indicated that the compression performance and compression ratio of ETDC deviated less from those of real-world datasets, when synthetic datasets were generated and compressed by TextGen rather than SDGen.

2 Framework of TextGen

2.1 Overview

The general idea of TextGen is similar to that of SDGen (Gracia-Tinedo *et al.*, 2015), whereas TextGen generates data content based on characterizations at the word level instead of the byte level. An overview of TextGen is shown in Fig. 1.

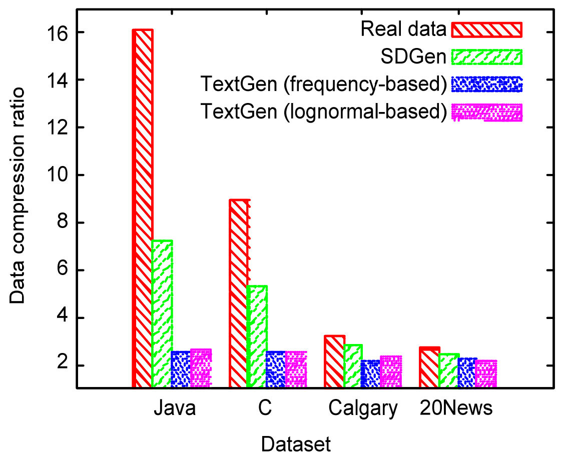
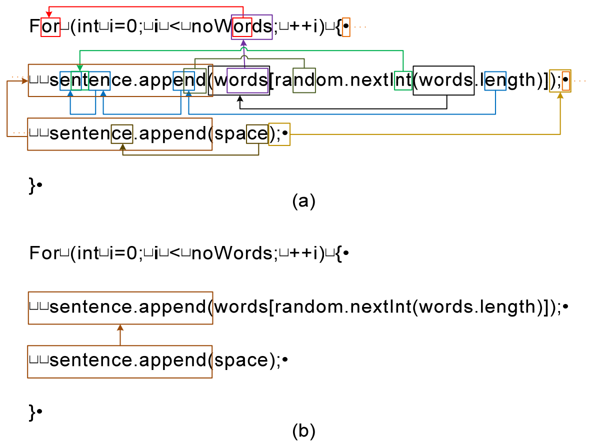


Fig. 1 TextGen framework

Gracia-Tinedo *et al.* (2015) pointed out that the byte frequency and repetition are the most important factors that influence the performance of most byte-based compressors. This idea also works for word-based compressors. However, repetition at the word level is much less frequent than that at the byte level. An example is shown in Fig. 2. Repetition means that a byte or word sequence has a longest match in the previous data stream. In Fig. 2, the arrowed lines point from the current sequences to the previous repeated ones. Note that the repetition length should be equal to or longer than two. Thus, we consider that the most important factor that influences the performance of word-based compressors is the word-frequency distribution. To capture this factor, we first segment the real-world text datasets into words, and then count the frequency of each word to build a corpus, which has the form <word, frequency>.



```
For⊔(int⊔i=0;⊔i⊔<⊔noWords;⊔++i)⊔{•

⊔⊔sentence.append(words[random.nextInt(words.length)]);•

⊔⊔sentence.append(space);•

}•
```

```
For⊔(int⊔i=0;⊔i⊔<⊔noWords;⊔++i)⊔{•

⊔⊔sentence.append(words[random.nextInt(words.length)]);•

⊔⊔sentence.append(space);•

}•
```

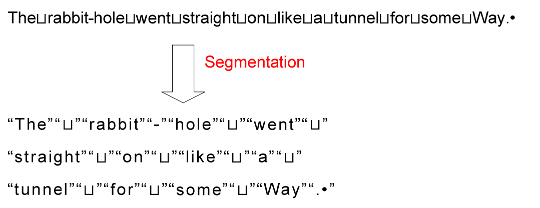
Fig. 2 Examples of repetition: (a) byte-level repetition; (b) word-level repetition

Second, we arrange all the words in the corpus in descending order by frequency. We give each word a rank, starting from 1, to obtain the rank-frequency distribution and the rank-word dictionary. To improve the data generation performance, we fit the rank-frequency distribution to a lognormal one by MLE.

Finally, we use the Monte Carlo method to generate dataset content. During the generation, a random number is continually generated based on the fitted lognormal distribution, and is used as the word rank. With the word rank, we can obtain the word from the rank-word dictionary which has the form <rank, word>.

2.2 Word segmentation and establishment of the corpus

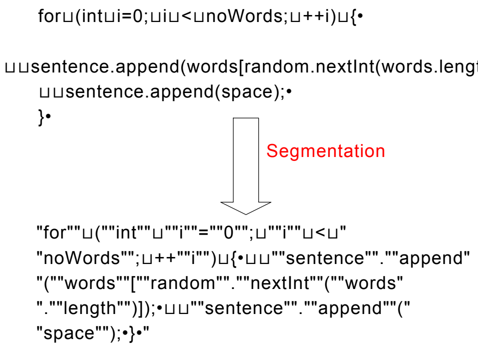
The dominant property that influences the compression performance and compression ratio of word-level compressors is the word distribution. Therefore, we need to segment text data into words. We use the segmentation method described by Horspool and Cormack (1992) and Salomon (2006). In this method, a word is defined as a maximal string of either alphanumeric characters (letters and digits) or other characters (punctuations and spaces). We denote all the alphanumeric words by A, and all the other words by P. As a result, the words from A and P strictly alternate. Two simple segmentation examples are shown in Figs. 3 and 4, where '•' indicates the end-of-line character (CR, LF, or both).



```
The⊔rabbit-hole⊔went⊔straight⊔on⊔like⊔a⊔tunnel⊔for⊔some⊔Way.•
```

```
“The”“⊔”“rabbit”“-”“hole”“⊔”“went”“⊔”
“straight”“⊔”“on”“⊔”“like”“⊔”“a”“⊔”
“tunnel”“⊔”“for”“⊔”“some”“⊔”“Way”“.•”
```

Fig. 3 Natural language segmentation example



```
for⊔(int⊔i=0;⊔i⊔<⊔noWords;⊔++i)⊔{•
⊔⊔sentence.append(words[random.nextInt(words.length)]);•
⊔⊔sentence.append(space);•
}•
```

```
"for""⊔(""int""⊔""i""=""0"";⊔""i""⊔<⊔"
"noWords"";⊔++""i"")⊔{•⊔⊔""sentence"".""append"
"(""words""[""random"".""nextInt""(""words"
".""length"")]);•⊔⊔""sentence"".""append""("
"space"");•}•"
```

Fig. 4 Java source code segmentation example

This segmentation method can be implemented by a simple finite-state automaton. With this method, real-world text datasets are recursively traversed to conduct word segmentation. The segmented words and their frequencies are saved in a hash table structure during segmentation. After word segmentation, the hash table is saved to a file as the corpus, which has the form <word, frequency>. If the datasets are too large, a random sampling method such as reservoir sampling (Vitter, 1985) can be used to reduce the segmentation time.

2.3 Word-frequency distribution fitting

The word-frequency distribution is established by the corpus. The Monte Carlo approach is then used to generate a data content. This approach continually generates a random number as the word rank, which maps the word content, and writes the word content into currently generating file until the expected file size is matched. Fitness proportional selection (Bäck, 1996) can be used to generate the random number based on the word-frequency distribution. However, a search operation is needed during the random number generation process. Even when the most efficient binary search is used, random number generation performance is still low. This is because the time complexity of a binary search is $O(\log m)$, which makes the time complexity of the data content generation performance become $O(n\log m)$. Here, n and m represent the word count in the expected dataset and the corpus, respectively. Thus, data content generation performance based on fitness proportional selection needs to be improved.

Inspired by previous work on natural languages (Li, 1992), we propose the use of a probability distribution model to generate the random numbers. To fit the model to the word-frequency distribution, we first arrange the words in the corpus in descending order by frequency. We assign the word that has the highest frequency to rank 1, the second highest word to rank 2, and so on. In previous natural language studies, the Zipf-Mandelbrot distribution (Powers, 1998) was the most widely used distribution to fit a word's rank-frequency distribution. However, the performance of generating random numbers based on this distribution is low. Thus, the lognormal distribution, which is also frequently used to fit a word distribution in natural language studies (Baayen, 1992), is used to fit a word's rank-frequency distribution. The probability density function (pdf) of the lognormal distribution is defined as

$$f(x\,|\,\mu,\sigma)=\frac{1}{x\sigma\sqrt{2\pi}}\exp\frac{-(\ln x-\mu)^2}{2\sigma^2}, \qquad (1)$$

where μ is the mean and σ is the standard deviation, which are the parameters that need to be estimated. MLE is used to estimate μ and σ. The principle of MLE is to seek the value of the parameter vector by maximizing the likelihood function of the desired probability distribution (Myung, 2003). The likelihood function is defined as the joint density treated as a function of the parameters θ:

$$L(\theta \mid x_1, x_2, \cdots, x_n) = \prod_{i=1}^{n} f(x_1; \theta). \tag{2}$$

By maximizing the likelihood function of the lognormal distribution, we obtain

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n}(\ln x_i)^2, \tag{3}$$

$$\hat{\sigma} = \frac{1}{n}\sum_{i=1}^{n}(\ln x_i - \hat{\mu})^2, \tag{4}$$

where n is the observed word's count, and x_i is the observed rank of the ith word. With Eqs. (3) and (4), MLE can be implemented by a very simple program to estimate μ and σ. After estimating the parameters, we use the Box-Muller transform method (Box and Muller, 1958) to generate a random number based on the lognormal distribution.

Supposing U_1 and U_2 are two independent random variables that are uniformly distributed in the interval [0, 1), we define two variables Z_1 and Z_2 with U_1 and U_2 in Eqs. (5) and (6), respectively:

$$Z_1 = \sqrt{-2\ln U_1 \cos(2\pi U_2)}, \tag{5}$$

$$Z_2 = \sqrt{-2\ln U_1 \sin(2\pi U_2)}. \tag{6}$$

Box and Muller (1958) have proved that Z_1 and Z_2 are both independent random variables, each with a standard normal distribution. Since Z (Z_1 or Z_2) is a standard normal random variable, Y will have a normal distribution with a mean deviation μ and a standard deviation σ:

$$Y = Z\sigma + \mu. \tag{7}$$

The normal and lognormal distributions are closely related. If X is distributed lognormally with the μ and σ, then $\log X$ is distributed normally with μ and σ. Thus, the random variables X with lognormal distribution can be obtained by

$$X = \exp(Y). \tag{8}$$

The mean deviation m and standard deviation υ of a normal random variable are functions of μ and σ of a lognormal random variable, respectively, expressed as

$$m = \ln\left(\mu^2 \Big/ \sqrt{\mu^2 + \sigma^2}\right), \tag{9}$$

$$\upsilon = \sqrt{\ln[(\mu^2 + \sigma^2)/\mu^2]}. \tag{10}$$

According to the above analysis, we can implement lognormal random number generation by Algorithm 1.

Algorithm 1 Lognormal random number generation

```
Input: mean deviation $\mu$ and standard deviation $\sigma$ of the lognormal distribution.
Output: random number $x$ with lognormal distribution.
1: $m$=ln($\mu$*$\mu$/sqrt($\mu$*$\mu$+$\sigma$*$\sigma$));
2: $\upsilon$=sqrt(ln(($\mu$*$\mu$+$\sigma$*$\sigma$)/($\mu$*$\mu$)));
3: generate random number $u$~$U$(0, 1) with a uniform
   distribution;
4: $z$←sqrt(-2*ln $u$*cos(2π*$u$));
   /* or $z$←sqrt(-2*ln $u$*sin(2π*$u$)); */
5: $y$←$z$*$m$+$v$;
6: $x$←exp($y$);
7: return $x$;
```

Algorithm 1 has a time complexity of $O(1)$ because it can be finished in a constant time. Thus, the time complexity of data content generation is $O(n)$, which is less than that of the fitness proportional selection approach.

2.4 Dataset generation algorithm

We integrate the metadata generation approach impressions (Agrawal *et al.*, 2009*)* into our content generation method. The dataset generation algorithm is described in Algorithm 2. First, the dataset structure tree is generated (line 1). The rank-word map is initialized with the rank-word dictionary C (line 2). For each file in the dataset, the file name is generated by giving a natural number, and the file size ss is generated based on the size distribution model. Then, the size ss and file name are added to the fileInfoList until the accumulated file size reaches the expected dataset size s (lines 3–8). Second, the fileInfoList is traversed. For each traverse, an empty file is created based on the file name (lines 9–17). The current generated file size is defined as curSize (line 10). The file contents are generated as follows: a random number wordRank is generated based on the lognormal distribution as the word rank, and then the word content is looked up by wordRank in rankWordMap. Finally, the content of the word is written to the current generating file and the word size is added to curSize. This process is repeated until curSize is equal to or larger than ss (lines 11–16). When traversing fileInfoList is finished, the dataset is successfully created.

Algorithm 2 Dataset generation

```
Input: parameters of the metadata generator and expected dataset size $s$; parameters of lognormal distribution $\sigma$, $\mu$; rank-word dictionary $C=\{(k, v)|k$ is the word rank, $v$ is the word$\}$.
Output: dataset $D=\{x|x$ is file or directory$\}$.
 1: generate a dataset structure tree;
 2: rankWordMap←initMap($C$)
 3: while curSize<$s$ do
 4:     generate the file name with a natural number;
 5:     generate the file size ss;
 6:     curSize←curSize+ss;
 7:     fileInfoList←<name,ss>;
 8: end while
 9: for <name, ss> in fileInfoList do
10:     curSize←0;
11:     while curSize<ss do
12:         wordRank←lognrand($\mu$, $\sigma$);
13:         word←lookup(rankWordMap,wordRank);
14:         writeToFile(fd, word);
15:         curSize←curSize+length(word);
16:     end while
17: end for
18: return $D$;
```

3 Implementation

TextGen is implemented in Java based on the source code of SDGen (Fig. 5).

TextScanner is extended from the abstract class AbstractScanner in SDGen. It is composed of the dataset traverse module, the word segmentation module, and the distribution fitting module. The dataset traverse module recursively traverses the real-world datasets by depth-first search (DFS). For each traversed file, Algorithm 1 is used to segment the file and build the word-frequency distribution. Then, the distribution fitting module is called to estimate the parameters of the lognormal distribution.

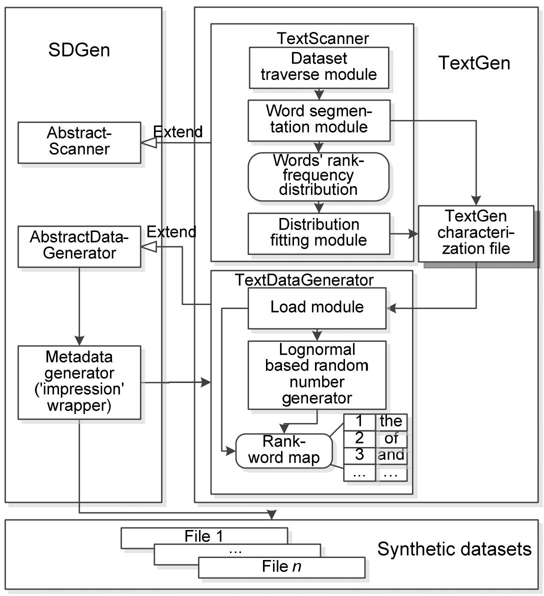


Fig. 5 Implementation architecture of TextGen

The characterization file is used to save the rank-word dictionary established after word segmentation, as well as the parameters of the lognormal distribution estimated by MLE. Unlike SDGen, the characterization file saves the properties that influence the performance of compressors at the word level instead of the byte level.

The metadata generator is used to generate the metadata of the dataset, including the directory structure tree, and the file number, size, and name. SDGen uses a wrapper class to call impressions to implement the metadata generator. More information about the metadata generator can be found in Agrawal *et al.* (2009).

TextDataGenerator is extended from the abstract class AbstractDataGenerator in SDGen. It is responsible for generating realistic data content at the word level. The lognormal-based random number generator is implemented by the Monte Carlo method as described in Section 2.4. The rank-word map, which contains the word rank and its content, is implemented by a HashMap data structure in Java and is initialized with the characterization file.

4 Evaluation

4.1 Experimental setup and datasets

All the experiments in this section were run on a single server, with the following configuration:

1) CPU: 2 Intel Xeon E5-2650 v2 2.6 GHz 8-core processors;
2) RAM: 128 GB DDR3;
3) Disk: 1.2 T MLC PCIe SSD card;
4) Operating system: CentOS release 6.5 (final).

Four text datasets were used to conduct the experiment:

Java: collection of Java source code from the GitHub (https://github.com/) and Sourceforge (http://sourceforge.net/) websites, including multiple versions of source codes, such as Hadoop, ZooKeeper, Hbase, and Lucene. The total size was 6.46 GB.

C: collection of C source code from GitHub and the Sourceforge websites including multiple versions of source code, such as httpd, memcached, nginx, and Subversion. The total size was 6.3 GB.

Calgary/Canterbury corpus (Arnold and Bell, 1997): collection of text and binary data files, commonly used for comparing data compression algorithms. The total size was 9.05 MB.

20News: collection of natural language documents from more than 20 000 newsgroups. It was originally collected by Lang (1995). The total size was 29.2 MB.

4.2 Lognormal distribution fitting results

In this section, the word rank-frequency distribution of real-world datasets was fitted to a lognormal distribution by MLE. The parameters of the lognormal fitting results are shown in Table 1.

Table 1 Parameters of lognormal fitting

Dataset	$\hat{\mu}$	$\hat{\sigma}$
Java	3.59	3.01
C	3.68	2.79
Calgary	3.05	3.10
20News	3.52	3.41

In Figs. 6–9, the pdf fitting curves of four real-world datasets and their word probability distributions are compared in log–log scale coordinates. The results show that the fitted lognormal pdf captures the characterization of the real-world datasets well at the word level. Thus, the fitted lognormal distribution can be used to replace the word probability distribution to generate a random number without significant loss of accuracy.

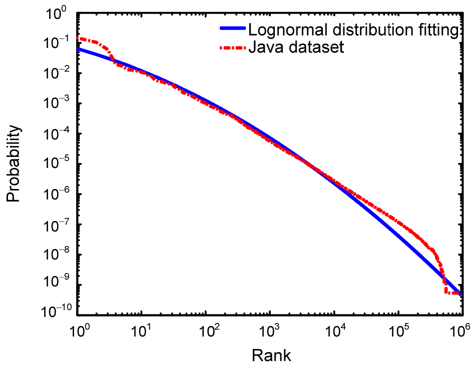


Fig. 6 Java dataset's word probability distribution and its lognormal distribution fitting

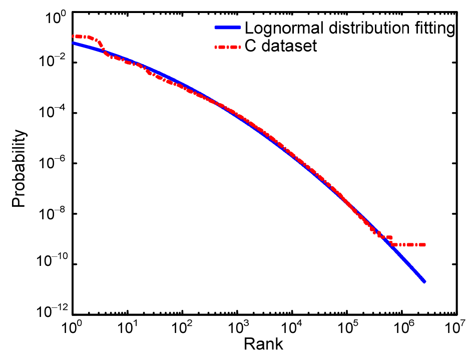


Fig. 7 C dataset's word probability distribution and its lognormal distribution fitting

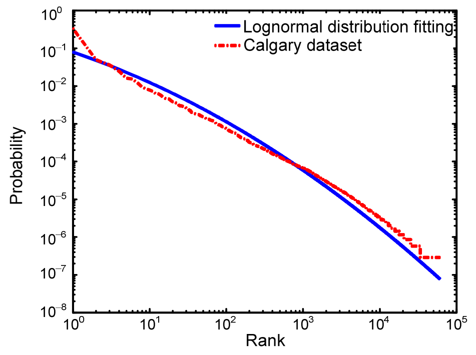


Fig. 8 Calgary dataset's word probability distribution and its lognormal distribution fitting

4.3 Dataset generation throughput evaluation

In Fig. 10, the dataset generation throughput between SDGen and TextGen is compared. We also compare two random number generation approaches in TextGen: (1) using fitness proportional selection based on word-frequency distribution; (2) using the lognormal distribution. The throughput is defined as the size of data generated per second (MB/s). There are two parameters that influence generation performance: uniqueness and generation thread number. The uniqueness is defined as

$$\text{uniqueness} = \frac{1}{\text{compression ratio}}. \tag{11}$$

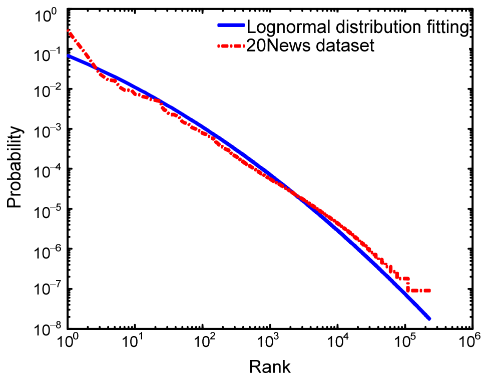


Fig. 9 20News dataset's word probability distribution and its lognormal distribution fitting

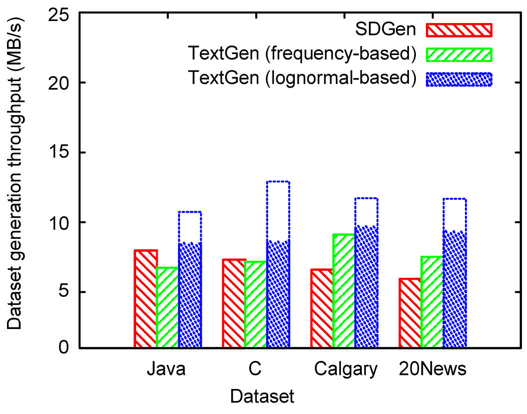


Fig. 10 Comparison of dataset generation throughput

An explanation of how uniqueness works is shown in Algorithm 3.

Algorithm 3 Generating a dataset with a predefined compression ratio

Input: uniqueness.
Output: dataset with a predefined compression ratio.

```
1:  while (generation not finished)
2:      double randomDouble=get a random double
3:      number in the interval [0, 1];
4:      boolean createUniqueData=true;
5:      if (randomDoubl<=uniqueness)
6:          createUniqueData=true;
7:      else
8:          createUniqueData=false;
9:      end if
10:     if (createUniqueData)
11:         generate a new chunk;
12:     else
13:         copy data from previously generated chunks;
14:     end if
15: end while
```

From the above pseudocode, we know that uniqueness is used to decide whether the current chunk is newly generated or a copy of previously generated chunks. By doing this, the compression ratio of the generated dataset will satisfy that defined in the configuration file. Obviously, this parameter will significantly influence the generation performance. The generation thread number is used to indicate how many threads would be used to generate the dataset. SDGen divides the file into chunks; thus, multiple chunks can be generated simultaneously by multiple threads. TextGen can also be executed in a similar multi-thread way to improve performance.

For a fair comparison of the results, both uniqueness and the generation thread number of the two methods were set to 1. The expected generation size was set to 5 GB. The datasets were generated by first scanning the four real-world text datasets and building their characterization files. The results show that TextGen (lognormal distribution-based) has the highest throughput of dataset generation. This can be explained as follows: both SDGen and TextGen (frequency-based) use fitness proportional selection to generate the random number using binary search. The time complexities of these two approaches to generate datasets are both $O(n\log m)$, significantly larger than the time complexity $O(n)$ of TextGen (lognormal-based). The results also indicate that the dataset generation throughput of SDGen is slightly lower than that of TextGen (frequency-based) in the Calgary and 20News datasets. This is because the basic generation unit of TextGen is word, whereas that of SDGen is byte, and most data of these two datasets are natural languages in which the word size is larger than 1 byte.

4.4 A comparison of word-based compressors

We compared three word-based compressors: ETDC (Brisaboa *et al.*, 2003), dynamic ETDC (DETDC) (Brisaboa *et al.*, 2008), and dynamic lightweight ETDC (DLETDC) (Brisaboa *et al.*, 2010), in terms of data compression throughput and ratio, with four real-world datasets. The compression ratio results were very close among three compressors (Fig. 11). This is because the three compressors are all based on word-based Huffman coding. DETDC and DLETDC outperformed ETDC in compression throughput (Fig. 12). This is because DETDC and DLETDC are optimized to improve their performance compared with ETDC. We chose ETDC as a representative word-based compressor to conduct our experiment for two reasons: (1) Both DETDC and DLETDC were developed based on ETDC; (2) ETDC can represent a large family of word-based compressors that use Huffman coding. Moreover, we used the LZ77 compressor (Ziv and Lempel, 1977) to test how TextGen behaves in a scenario other than its design goal.

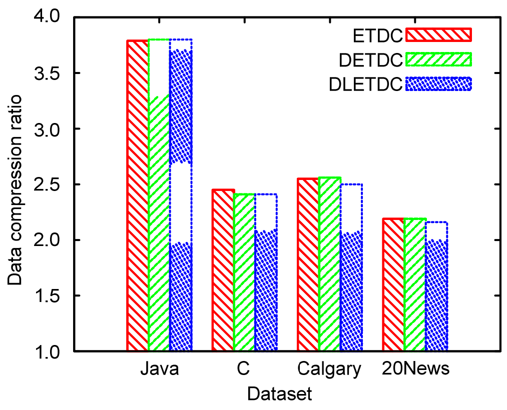


Fig. 11 Comparison of compression ratios of three word-based compressors
ETDC: end-tagged dense code; DETDC: dynamic ETDC; DLETDC: dynamic lightweight ETDC

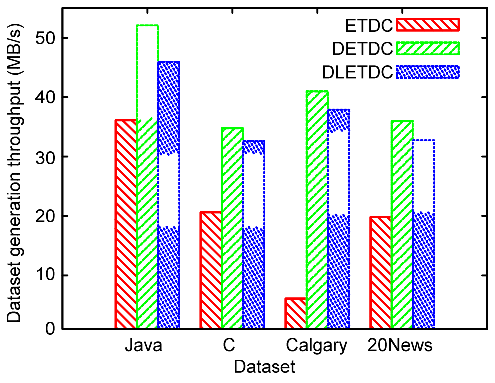


Fig. 12 Comparison of compression throughputs of three word-based compressors
ETDC: end-tagged dense code; DETDC: dynamic ETDC; DLETDC: dynamic lightweight ETDC

4.5 Compression throughput evaluation

In this section, we use word-based ETDC and byte-based LZ77 compressors to compare the rates of deviation of compression throughput between the real-world datasets and synthetic datasets generated by SDGen and TextGen, respectively. The compression throughput is defined as the size of data compressed per second (MB/s). TextGen includes frequency-based random number generation and lognormal-based approaches. The rate of deviation is defined as

$$d = \frac{|r - s|}{r} \times 100\%, \qquad (12)$$

where r represents the real-world dataset compression throughput, and s represents the synthetic dataset compression throughput.

In Fig. 13, ETDC is used to evaluate the compression throughput. The rates of deviation between real-world and synthetic datasets generated by using different methods in the four text dataset tests by ETDC are: for SDGen, 62.77%, 60.32%, 16.17%, and 29.41%, respectively; for TextGen (frequency-based), 23.96%, 11.13%, 2.18%, and 4.44%, respectively; for TextGen (lognormal-based), 23.74%, 4.55%, 0.05%, and 3.11%, respectively. The results show that the rate of deviation of TextGen is significantly lower than that of SDGen. The reason is that SDGen captures the properties that influence the compression performance of the real-world datasets only at the byte level. This causes ETDC, a word-based compressor that uses word as its basic compression unit, to behave completely differently on such synthetic datasets than on real-world datasets. However, TextGen captures enough characterizations at the word level to ensure that the deviation ratio in the test is much lower than that of SDGen. Compared with the frequency-based random generation approach, the lognormal-based approach results show no significant difference, validating that the lognormal distribution fits the word-frequency distribution of real-world datasets well.

In Fig. 14, LZ77 is used to evaluate the compression throughput. The rates of deviation between real-world and synthetic datasets generated by different methods in the four text dataset tests are: for SDGen, 33.37%, 27.38%, 1.80%, and 12.73%, respectively; for TextGen (frequency-based), 80.73%, 72.31%, 37.23%, and 39.08%, respectively; for TextGen (lognormal-based), 79.65%, 71.41%, 39.43%, and 37.94%, respectively. The results show that the rate of deviation of TextGen is significantly higher than that of SDGen. The reason is that the datasets generated by TextGen lack byte-level properties which influence the compression throughput of LZ77.

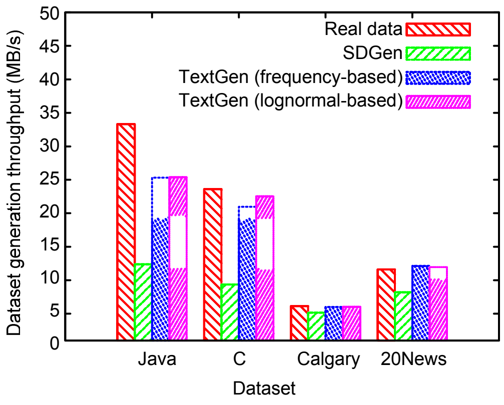


Fig. 13 Comparison of compression throughput by end-tagged dense code

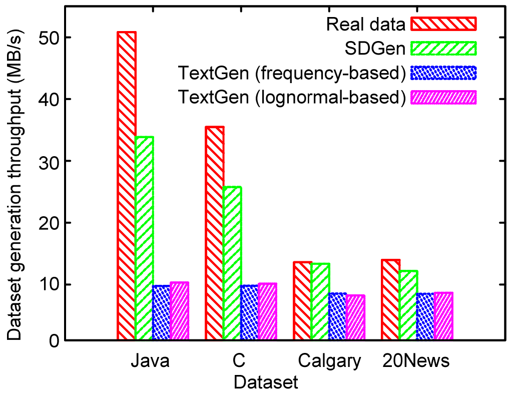


Fig. 14 Comparison of compression throughput by LZ77

4.6 Compression ratio evaluation

In this section, we use word-based ETDC and byte-based LZ77 compressors to compare the rates of deviation of the compression ratio between real-world and synthetic datasets generated by SDGen and TextGen, respectively. The compression ratio is defined as the ratio between uncompressed and compressed sizes. TextGen includes frequency-based random number generation and lognormal-based approaches. Eq. (4) can be used to define the rate of deviation of the compression ratio.

In Fig. 15, ETDC is used to evaluate the compression ratio. The rates of deviation between real-world and synthetic datasets generated by different methods in the four text dataset tests are: for SDGen, 53.12%, 46.90%, 54.36%, and 49.51%, respectively; for TextGen (frequency-based), 10.02%, 6.35%, 1.75%, and 1.17%, respectively; for TextGen (lognormal-based), 13.37%, 10.79%, 0.28%, and 7.13%, respectively. The results show that: (1) the rate of deviation of TextGen is significantly lower than that of SDGen; (2) the lognormal distribution fits the word-frequency distribution of real-world datasets well. The reason is similar to that given following the analysis of compression throughput in Section 4.5.

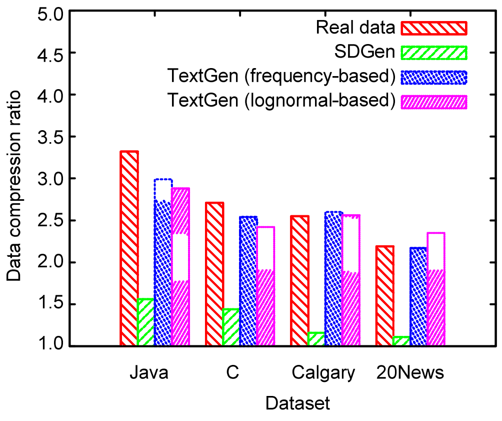


Fig. 15 Comparison of the compression ratio by end-tagged dense code

In Fig. 16, LZ77 is used to evaluate the compression ratio. The rates of deviation between real-world and synthetic datasets generated by the different methods are: for SDGen, 55.36%, 40.35%, 12.86%, and 8.87%, respectively; for TextGen (frequency-based), 84.13%, 71.71%, 33.09%, and 16.46%, respectively; for TextGen (lognormal-based), 83.91%, 71.72%, 26.91%, and 17.82%, respectively. The results show that the rate of deviation of TextGen is significantly higher than that of SDGen. The reason is that the datasets generated by TextGen lack the byte-level properties which influence the compression ratio of LZ77.

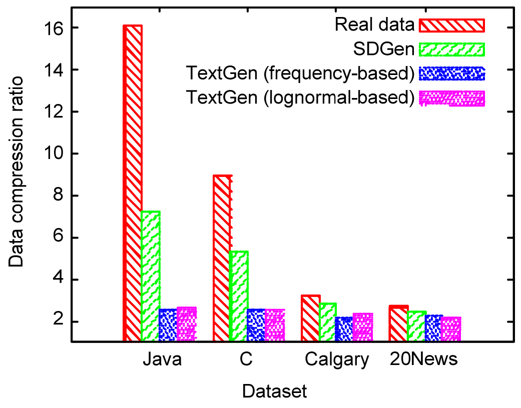


Fig. 16 Comparison of the compression ratio by LZ77

5 Conclusions

We present a lognormal-distribution-based text data content generation method for modern storage system benchmarks. The key idea behind TextGen is to capture the word-level properties that influence the compression performance and compression ratio of word-based compressors, and to use the characterizations to generate the content. To improve the performance of content generation, the lognormal distribution is fitted to the word-frequency distribution using maximum likelihood estimation (MLE). Four text datasets were used to evaluate TextGen. Experimental results show that compared with the real data in the end-tagged dense code (ETDC) compressor tests, the synthetic data perform accurately. Thus, TextGen can be used to generate data contents for word-based compressor-enabled storage system benchmarks. However, TextGen is not applicable to byte-based compressors. To solve this problem, we can integrate TextGen into SDGen as a text data generator module to face more complex situations in which both the Ziv-Lempel family and word-based compressors are enabled in the storage systems to be tested.

The source code of TextGen can be downloaded from https://github.com/wlx0419/TextGen.

References

Agrawal, N., Bolosky, W.J., Douceur, J.R., *et al.*, 2007. A five-year study of file-system metadata. *ACM Trans. Stor.*, **3**(3):9.1-9.32. http://dx.doi.org/10.1145/1288783.1288788

Agrawal, N., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., 2009. Generating realistic impressions for file-system benchmarking. *ACM Trans. Stor.*, **5**(4):16.1-16.30. http://dx.doi.org/10.1145/1629080.1629086

Anderson, E., Kallahalla, M., Uysal, M., *et al.*, 2004. Buttress: a toolkit for flexible and high fidelity I/O benchmarking. Proc. USENIX Conf. on File and Storage Technologies, p.4.

Armstrong, T.G., Ponnekanti, V., Borthakur, D., *et al.*, 2013. Linkbench: a database benchmark based on the Facebook social graph. Proc. ACM SIGMOD Int. Conf. on Management of Data, p.1185-1196. http://dx.doi.org/10.1145/2463676.2465296

Arnold, R., Bell, T., 1997. A corpus for the evaluation of lossless compression algorithms. Data Compression Conf., p.201-210. http://dx.doi.org/10.1109/DCC.1997.582019

Baayen, H., 1992. Statistical-models for word-frequency distributions—a linguistic evaluation. *Comput. Human.* **26**(5-6):347-363. http://dx.doi.org/10.1007/Bf00136980

Bäck, T., 1996. Evolutionary Algorithms in Theory and Practice. Oxford University Press, Oxford, UK, p.120.

Bonwick, J., Ahrens, M., Henson, V., *et al.*, 2003. The Zettabyte File System. Technical Report, Sun Microsystems, Inc., Santa Clara, USA.

Box, G.E.P., Muller, M.E., 1958. A note on the generation of random normal deviates. *Ann.. Math. Statist.*, **29**(2): 610-611. http://dx.doi.org/10.1214/aoms/1177706645

Brisaboa, N.R., Iglesias, E., Navarro, G., *et al.*, 2003. An efficient compression code for text databases. *Adv. Inform. Retriev.*, **2633**:468-481. http://dx.doi.org/10.1007/3-540-36618-0_33

Brisaboa, N.R., Fariña, A., Navarro, G., *et al.*, 2007. Lightweight natural language text compression. *Inform. Retriev.*, **10**(1):1-33. http://dx.doi.org/10.1007/s10791-006-9001-9

Brisaboa, N.R., Fariña, A., Navarro, G., 2008. New adaptive compressors for natural language text. *Softw.-Pract. Exper.*, **38**(13):1429-1450. http://dx.doi.org/10.1002/spe.882

Brisaboa, N.R., Fariña, A., Navarro, G., *et al.*, 2010. Dynamic lightweight text compression. *ACM Trans. Inform. Syst.*, **28**(3):1-32. http://dx.doi.org/10.1145/1777432.1777433

Chilan, C.M., 2005. IOzone: an Open Source File System Benchmark Tool. Technical Report, the National Center for Supercomputing Applications Hierarchical Data Format Group, University of Illinois at Urbana-Champaign, Illinois.

Cooper, B.F., Silberstein, A., Tam, E., *et al.*, 2010. Benchmarking cloud serving systems with YCSB. Proc. ACM Symp. on Cloud Computing, p.143-154. http://dx.doi.org/10.1145/1807128.1807152

Difallah, D.E., Pavlo, A., Curino, C., *et al.*, 2013. OLTP-bench: an extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, **7**(4):277-288.

http://dx.doi.org/10.14778/2732240.2732246

Drago, I., Bocchi, E., Mellia, M., *et al.*, 2013. Benchmarking personal cloud storage. Proc. Conf. on Int. Measurement, p.205-212. http://dx.doi.org/10.1145/2504730.2504762

Dvorský, J., Pokorný, J., Snášel, V., 1999. Word-based compression methods and indexing for text retrieval systems. *Adv. Database Inform. Syst.*, **1691**:76-84. http://dx.doi.org/10.1007/3-540-48252-0_6

Fariña, A., Brisaboa, N.R., Navarro, G., *et al.*, 2012. Word-based self-indexes for natural language text. *ACM Trans. Inform. Syst.*, **30**(1):1-34. http://dx.doi.org/10.1145/2094072.2094073

Gracia-Tinedo, R., Harnik, D., Naor, D., *et al.*, 2015. SDGen: mimicking datasets for content generation in storage benchmarks. Proc. USENIX Conf. on File and Storage Technologies, p.317-330.

Horspool, R.N., Cormack, G.V., 1992. Constructing word-based text compression algorithms. Data Compression Conf., p.62-71. http://dx.doi.org/10.1109/DCC.1992.227475

Lang, K., 1995. Newsweeder: learning to filter netnews. Proc. Int. Conf. on Machine Learning, p.331-339.

Li, A., Yang, X., Kandula, S., *et al.*, 2010. Cloudcmp: comparing public cloud providers. Proc. ACM SIGCOMM Conf. on Internet Measurement, p.1-14. http://dx.doi.org/10.1145/1879141.1879143

Li, W.T., 1992. Random texts exhibit Zipf-law-like word-frequency distribution. *IEEE Trans. Inform. Theor.*, **38**(6):1842-1845. http://dx.doi.org/10.1109/18.165464

Moffat, A., Zobel, J., Sharman, N., 1997. Text compression for dynamic document databases. *IEEE Trans. Knowl. Database Eng.*, **9**(2):302-313. http://dx.doi.org/10.1109/69.591454

Myung, I.J., 2003. Tutorial on maximum likelihood estimation. *J. Math. Psychol.*, **47**(1):90-100. http://dx.doi.org/10.1016/S0022-2496(02)00028-7

Powers, D.M.W., 1998. Applications and explanations of Zipf's law. Proc. Joint Conf. on New Methods in Language Processing and Computational Natural Language Learning, p.151-160.

Rodeh, O., Bacik, J., Mason, C., 2013. BTRFS: the Linux B-tree filesystem. *ACM Trans. Stor.*, **9**(3):1-32. http://dx.doi.org/10.1145/2501620.2501623

Salomon, D., 2006. Data Compression: the Complete Reference. Springer-Verlag New York, Inc., New York, USA, p.885.

Tarasov, V., Bhanage, S., Zadok, E., *et al.*, 2011. Benchmarking file system benchmarking: it *is* rocket science. Proc. USENIX Conf. on Hot Topics in Operating Systems, p.8-13.

Traeger, A., Zadok, E., Joukov, N., *et al.*, 2008. A nine year study of file system and storage benchmarking. *ACM Trans. Stor.*, **4**(2):1-56. http://dx.doi.org/10.1145/1367829.1367831

Vitter, J.S., 1985. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, **11**(1):37-57. http://dx.doi.org/10.1145/3147.3165

Yoshida, S., Morihara, T., Yahagi, H., *et al.*, 1999. Application of a word-based text compression method to Japanese and Chinese texts. Data Compression Conf., p.561. http://dx.doi.org/10.1109/DCC.1999.785718

Ziv, J., Lempel, A., 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theor.*, **23**(3): 337-343. http://dx.doi.org/10.1109/TIT.1977.1055714