D-Zipfian: A Decentralized Implementation of Zipfian*

Sumita Barahmand and Shahram Ghandeharizadeh

Database Laboratory Technical Report 2012-04

Computer Science Department, USC

Los Angeles, California 90089-0781

May 16, 2013

Abstract

Zipfian distribution is used extensively to generate workloads to test, tune, and benchmark data stores. With scalable multi-node database management systems, a centralized single node benchmarking framework that embodies Zipfian may utilize its resources fully and fail to generate work at a sufficiently high rate to evaluate its target system. This means the benchmarking framework must become decentralized and scalable. BG is one such framework. This paper presents BG's decentralized, parallel implementation of Zipfian named *D-Zipfian*. D-Zipfian employs multiple nodes that reference data items independently. To produce meaningful results, this scalable technique strives to produce a distribution that is independent of its degree of parallelism, i.e., number of employed nodes. Moreover, it supports heterogeneous nodes that reference data items at different rates. We characterize the behavior of D-Zipfian with different degrees of parallelism and skewness, population sizes, and heterogeneity of its employed nodes.

A Introduction

Benchmarks are a critical component of testing, tuning, and evaluating data stores. Over the years, several studies have argued for an application-directed approach to benchmarking that reflects the behavior of a particular application [2, 18, 14]. With most applications, a random distribution of access to data items is typically not realistic due to Zipf's law [21]. This law states that given some collection of data items, the frequency of any data item is inversely proportional to its rank in its frequency table. This means the most frequently referenced data item will occur more often than the second most frequent data item, the second most frequent data item will occur more often than the third most frequent data item, so on and so forth.

*A shorter version of this paper appeard in the Sixth International Workshop on Testing Database Systems (DBTest), Co-located with ACM SIGMOD, June 2013.

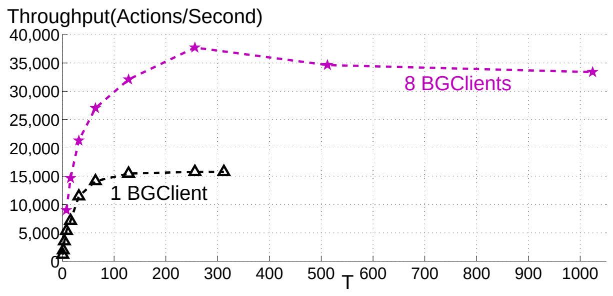


Figure 1: Performance of MongoDB with two different number of BGClients.

By manipulating the exponent[1] θ that characterizes the Zipfian distribution one may emulate different rules of thumb such as: 80% of requests (ticket sales [6], frequency of words [21], profile look-ups) reference 20% of data items (movies opening on a weekend, words uttered in natural language, members of a social networking site).

BG [3] is a benchmark that quantifies the processing capability of SQL, NoSQL and NewSQL [5, 20] data stores among others[2] in support of interactive social networking actions and sessions. (See [5] for a survey of alternative data stores.) It rates a data store using a pre-specified service level agreement, SLA. An example SLA may require 95% of issued requests to observe a response time faster than 100 milliseconds with the amount of produced unpredictable reads less than 0.1%. Given a data store, BG computes two different ratings named SoAR and Socialites. While SoAR pertains to the highest throughput (actions per second) supported by the data store, Socialites quantifies the maximum number of simultaneous threads that satisfy the specified SLA. Figure 1 illustrates these concepts using MongoDB version 2.0.6, a document store for storage and retrieval of JavaScript Object Notations, JSON. It shows the throughput (y-axis) of MongoDB as a function of the number of threads (x-axis). The two different curves pertain to the number of benchmarking nodes, termed *BGClients*, used to generate the workload for the data store. The curve labeled "8 BGClients" terminates at 1025 threads because the pre-specified SLA is violated with more than 1025 threads. This is the Socialites rating of MongoDB. The highest observed throughput, i.e., peak of this curve, is 36,043 actions per second and is realized with 264 threads. Hence, SoAR of MongoDB is 36,043.

Today's data stores process requests at such a high rate that one BGClient may not be sufficient to rate them accurately [3]. To address this challenge, BG utilizes multiple BGClients to generate work for its target data store. A coordinator, termed BGCoord, manages these BGClients and aggregates their results for final display. To illustrate, consider the curve labeled 1 BGClient in Figure 1. It corresponds to the same experiment as the one with 8 BGClients with one key difference: Only 1 BGClient is used to generate the workload. It computes SoAR of MongoDB to be 15,000 actions per second. This is inaccurate because the

[1]See Equation 1 in Section B.
[2]Such as cloud service providers and graph databases.

BGClient has utilized its Intel i7-2600 four core CPU fully while MongoDB's CPU is partially utilized. 8 BGClients resolve this limitation to accurately[3] quantify SoAR of MongoDB at 36,043 actions per second. This is more than two folds higher than the peak throughput observed with 1 BGClient. The Socialites rating with 8 BGClients (1025) is more than 3 times higher than that with one BGClient (317).

Use of multiple BGClients raises the following research question: How do BGClients produce requests such that their overall distribution conforms to a pre-specified Zipfian distribution? One solution, named Replicated Zipfian (*R-Zipfian*), requires each BGClient to employ the specified Zipfian distribution with the entire population independently. R-Zipfian is effective when BG produces workloads with read only references. It also accommodates heterogeneous nodes where each node produces requests at a different rate as each BGClient uses the entire population to generate the Zipfian distribution.

However, with BG, R-Zipfian introduces additional complexity in two cases. First, different BGClients might be required to reference a unique data item at an instance in time in order to model reality. For example, they might be required to emulate a unique user of a social networking site performing an action such as accepting friend request. R-Zipfian would require additional software to coordinate multiple BGClients to guarantee uniqueness of the referenced data items. Second, BG measures the amount of unpredictable data produced by a data store using workloads that are a mix of read and write references. It time stamps the read and write references to detect unpredictable reads. R-Zipfian would require BG to utilize synchronized clocks [11, 12, 7, 10, 15] to detect unpredictable reads. Both complexities are avoided by partitioning data items across BGClients.

With partitioning, BGCoord assigns a fraction of data items to each BGClient. A BGClient issues requests that reference its data items only. This ensures BGClients reference unique data items simultaneously. Moreover, the potential read-write and write-write conflicts are localized to each BGClient and its partition, enabling it to quantify its observed amount of unpredictable data using its own system clock and independent of the other BGClients.

With N partitioned BGClients, each BGClient must reference data items such that the overall distribution of references respects the Zipfian distribution. Moreover, given a θ exponent, the resulting distribution must remain constant as a function of N, i.e., the degree of parallelism employed by BG. This property is not trivial to realize because each BGClient has a subset of the original population and issues requests independently. As discussed in Section C, if each BGClient uses the original θ with a subset of the population to reference data items, the resulting distribution becomes more uniform as we increase the value of N. This is not desirable because it produces experimental results that are erratic and difficult to explain. For example, one may quantify the processing capability of a cache augmented SQL (CASQL) data store [9, 17, 1] with n_1 and n_2 BGClients ($n_1 < n_2$) and observe a lower processing capability with n_2 because its distribution

[3]The 8 BGClients impose a sufficiently high load to cause MongoDB to fully utilize the Intel i7-2600 four core CPU of its server. In [3], we report ratings with 16 BGClients to show no improvement when compared with 8 BGClients.

pattern is more uniform (which reduces the cache hit rate with a limited cache size). This is avoided by making the Zipfian distribution independent of N, enabling a true apple-to-apple comparison of benchmarking results obtained with different number of BGClients.

The rest of this paper is organized as follows. Section B formalizes the problem statement to parallelize a Zipfian distribution. Section C presents two intuitive solutions and quantifies their limitations using a small population of data items. D-Zipfian is presented and quantified in Section D. We discuss D-Zipfian in Section E and conclude in Section F.

B Problem Statement

With a Zipfian distribution, assuming M is the number of data items, the probability of data item i is:

$$p_i(M,\theta) = \frac{\frac{1}{i^{(1-\theta)}}}{\sum_{m=1}^{M}(\frac{1}{m^{(1-\theta)}})} \tag{1}$$

where θ characterizes the Zipfian distribution. Some studies use $(1-\theta)$ as the exponent, e.g., [6], and others use θ as the exponent, e.g., [4]. With the latter (former), the distribution becomes more skewed with larger (smaller) values of θ. The two definitions produce identical distributions based on the chosen θ value. For example, the distribution produced with one definition (say Equation 1) and $\theta = 0.27$ is identical to that produced using the alternative definition with $\theta = 0.73$. The concepts and techniques presented in this paper apply to both definitions. Without loss of generality and to simplify discussions, we use $(1-\theta)$ as the exponent of Zipfian for the rest of this paper.

Assuming data items are numbered 1 to M, a centralized implementation of Zipfian is as follows:

1. Compute the probability of each data item using Equation 1.

2. Compute array A consisting of M elements where the value of the first element is set to the probability of the first item, $A[1] = p_1(M,\theta)$, and the value of each remaining element m is the sum of its assigned probability and the probabilities assigned to previous $m-1$ elements, $A[i] = \sum_{j=1}^{i} p_j(M,\theta)$, $1 \leq i \leq M$. The last element of the array, $A[M]$, should have the value 1 because sum of the M probabilities equals one, $\sum_{j=1}^{M} p_j(M,\theta) = 1$. If this value is slightly lower than 1 then set it to 1.

3. Generate a random value r between 0 and 1. Identify the k^{th} element of the array that satisfies the following two conditions: a) A[k] is greater than or equal to r, and b) Either A[k-1] has a value lower than r or is non existent (because k is the first element of A). Produce k as the referenced data item, $1 \leq k \leq M$.

For an example, see discussions of Table 1 in Section C.

The challenge is how to parallelize this simple algorithm such that N BGClients reference data items and produce a distribution almost identical to that of one BGClient referencing data items. The final algorithm is correct as long as its resulting distribution is independent of its employed degree of parallelism, N. Below, we differentiate between local and global probability of a data item to provide a mathematical formulation of the problem.

Each data item i has a local and a global probability of reference. Its local probability specifies its likelihood of reference by its assigned BGClient k with m_k data items. One possible definition of the local probability of an object i is provided by Equation 1, $p_i(m_k, \theta)$. An algorithm may either use this definition or provide a new one, see Crude in Section C and D-Zipfian in Section D. The global probability of data item i assigned to BGClient k is a function of its local probability and the ratio of the number of references performed by BGClient k (O_k) relative to the total number of references (O) by N BGClients:

$$q_i(M, \theta, N) = \frac{O_k}{O} \times p_i(m_k, \theta) \tag{2}$$

With 1 BGClient, $N = 1$, local and global probability of a data item are identical, $q_i(M, \theta, 1) = p_i(m_k, \theta)$, because all data items are assigned to one BGClient, $m_k = M$, and that BGClient issues all requests, i.e., $\frac{O_1}{O} = 1$. With 2 or more BGClients, the global probability of a data item is lower than its local probability, $q_i(M, \theta, 1) \leq p_i(m_k, \theta)$. See discussions of Table 1 in Section C.

In sum, a parallel implementation of Zipfian with N BGClients may manipulate either the number of data items (m_k) assigned to each BGClient k and their identity, the definition of the local probability of an object i, the number of references (O_k) made by BGClient k, or all three. Note that by manipulating O_k, we are not shortening the execution time of one BGClient relative to the others, see Section E. To the contrary, as detailed in Section D.2, D-Zipfian manipulates O_k to require all BGClients to provide the same execution time with a platform consisting of heterogeneous nodes. This is important because, with N parallel BGClients, all BGClients should start and finish at approximately the same time. If one finishes considerably sooner than the others then the degree of parallelism is no longer N.

A mathematical formulation imposes the following constraint on a parallel implementation of Zipfian: $q_i(M, \theta, N) \approx q_i(M, \theta, 1)$ for all i and $N > 1$. It states the computed global probability of each data item i with two or more BGClients should be approximately the same as its computed probability with one BGClient.

The concepts presented in this section are demonstrated with an example in the next section using two naïve and intuitive ways to parallelize the centralized implementation of the Zipfian. They pave the way for the correct parallel implementation, D-Zipfian of Section D. The reader may skip to Section D for the final solution.

Data item	Zipfian/Crude with $N = 1$		Crude with $N = 3$	
i	$p_i(12, 0.01) = q_i(12, 0.01, 1)$	A[i]	$p_i(4, 0.01)$	$q_i(4, 0.01, 3)$
1	0.319014588	0.319014588	0.477558748	0.159186249
2	0.160616755	0.479631343	0.240440216	0.080146739
3	0.107512881	0.587144224	0.160944731	0.053648244
4	0.080866966	0.668011191	0.121056305	0.040352102
5	0.064838094	0.732849284	0.477558748	0.159186249
6	0.054130346	0.786979631	0.240440216	0.080146739
7	0.046469017	0.833448647	0.160944731	0.053648244
8	0.04071472	0.874163368	0.121056305	0.040352102
9	0.036233514	0.910396882	0.477558748	0.159186249
10	0.032644539	0.943041421	0.240440216	0.080146739
11	0.029705152	0.972746574	0.160944731	0.053648244
12	0.027253426	1	0.121056305	0.040352102

Table 1: Example with 12 data items and θ=0.01.

C Example and Two Naïve Approaches

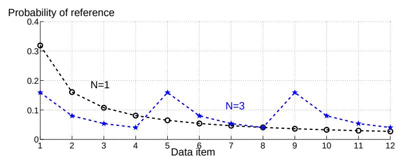
This section uses a small population consisting of twelve data items (M=12) to demonstrate the concepts presented in Section B. In addition, it describes two naïve techniques to parallelize Zipfian and their limitations.

Table 1 shows the local and global properties of the individual data items with 1 and 3 nodes, N=1 and N=3. Its first column shows the individual data items numbered from 1 to 12. Its second and third columns correspond to one node ($N = 1$) and show the local and global probabilities of each data item with the exponent 0.01, θ=0.01, and the values of Array A used by a centralized implementation to generate the Zipfian distribution, respectively. To implement Zipfian, an implementation generates a random value r between 0 and 1, say r=0.5. It produces data item 3 as its output because A[3] exceeds 0.5 and A[2] is less than 0.5. (See Step 2 of the pseudo-code to generate data items in Section B for a precise definition of selecting A[i].)

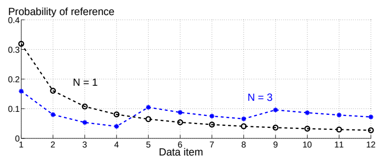
With N BGClients, say N=3, a technique named *Crude* range partitions data items across the BGClients as follows: BGClient 1 is assigned data items number 1 to 4, BGClient 2 is assigned data items number 5 to 8, and BGClient 3 is assigned data items number 9 to 12. It uses Equation 1 with $m_i = 4$ and original θ value (0.01) to compute the local probability of each data item, see the fourth column of Table 1. The fifth column of Table 1 shows the global probability of each data item with Crude using Equation 2 assuming each BGClient produces $\frac{1}{3}$ of references, i.e., $O = 3 \times O_k$. These are significantly different than those with 1 BGClient, compare 2nd and 5th columns, and do not satisfy the mathematical constraint presented in Section B.

Crude may assign data items to N BGClients in several other ways:

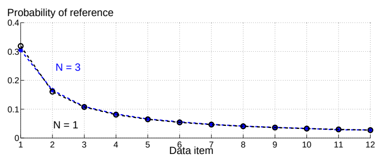
- Hash (instead of range) partitions data items using their id i to assign m_k data items to BGClient k.



a) Crude



b) Normalized-Curde



c) D-Zipfian

Table 2: $q_i(M, \theta, N)$ of data items with three different techniques, M=12, θ=0.01, and two different values of N.

- BGCoord may provide BGClient k with a list of m_k data items. Next, each BGClient would use the centralized implementation of Zipfian (see Section B) with M to generate a data item. If this data item is not on its list, it is discarded and a new data item is generated.

- BGCoord may utilize the centralized Zipfian with M data items to generate a trace file of the referenced data items and compute m_k data items for each BGClient k. Next, BGClient k reads the file and issues requests only for those m_k data items assigned to it.

And, other possible techniques. While these enable each BGClient to generate a Zipfian distribution independently, the resulting distribution (across all N BGClients) is dependent on the value of N. As we increase the value of N, the resulting distribution becomes more uniform, see Figure 2.a. Note that with N=3, the same distribution is repeated 3 times because each BGClient generates its distribution independently with m_k=4 and θ=0.01. Hence, a data item that was referenced infrequently with N=1 is now accessed more frequently. Unless Crude manipulates either its definition of local probability of a data item (p_i) or the number of references issued by a BGClient, the results of Table 1 remain unchanged.

A variant of Crude, named *Normalized-Crude*, defines the local probability of a data item i as follows: $p_i = \frac{p_i(M,\theta)}{\sum_{k=1}^{m_k} p_k(M,\theta)}$. This definition utilizes M (instead of m_k) to normalize the probability of data items assigned to each BGClient. With one node, N=1, it is identical to the centralized Zipfian because its denominator equals 1 ($m_i = M$ and the sum of the probability of data items equals 1). With more than one node, $N > 1$, the global probabilities produced by Normalized-Crude are more uniform than Crude, see Figure 2.b assuming $O = 3 \times O_k$. Note that the most popular data item with $N = 1$ has a global probability that is almost twice that with $N = 3$. However, Section D shows that with a minor adjustment, Normalized-Crude is transformed into the final solution.

D D-Zipfian

We present D-Zipfian assuming BGClients are homogeneous and produce requests at approximately the same rate. Subsequently, Section D.2 extends the discussion to heterogeneous BGClients that produce requests at different rates.

D.1 Homogeneous BGClients

With N BGClients, D-Zipfian constructs N clusters such that the sum of the probability of data items assigned to each cluster is $\frac{1}{N}$. Given a cluster k consisting of m_k elements and assigned to BGClient k, D-Zipfian overrides the local probability of each data item i as follows:

$$p_i = \frac{p_i(M,\theta)}{\sum_{m=1}^{m_k} p_i(M,\theta)} \tag{3}$$

This definition of local probability is identical to that used by Normalized-Crude. D-Zipfian is different because it constructs clusters by requiring the sum of probability of data items assigned to one cluster to approximate $\frac{1}{N}$. Thus, denominator of Equation 3 approximates $\frac{1}{N}$. Details of D-Zipfian can be summarized in two steps.

In this first step, BGCoord computes the probability of access to the M data items using Equation 1. Next, it constructs N clusters of data items such that the sum of the probability of the m_k data items assigned to cluster k is $\frac{1}{N}$, $\sum_{i=1}^{m_k} p_i(M,\theta) = \frac{1}{N}$. Finally, it assigns cluster k to BGClient k by transmitting[4] the identity of its data items to BGClient k. (A heuristic to construct clusters is described in the following paragraphs.)

In the second step, each BGClient k adjusts the probability of its assigned data items using Equation 3. Note that its denominator of Equation 3 approximates $\frac{1}{N}$ because BGCoord assigned objects to BGClient k with the objective to approximate $\frac{1}{N}$. Finally, each BGClient uses its computed probabilities to generate array A to produce data items, see Section B. Generation of the requests by each BGClient is independent of the other BGClients.

One may construct clusters of Step 1 using a variety of heuristics. We use the following simple heuristic. After BGCoord computes the quota for each BGClient k, $Q_k = \frac{1}{N}$, it assigns data items to the BGClients in a round-robin manner starting with the data item that has the highest probability. Once it encounters a BGClient whose Q_k is exhausted, BGCoord attempts to assign the data item with the lowest probability to this BGClient as long as its Q_k is not exceeded. Otherwise, it removes this BGClient from the list of candidates for data item assignment. It proceeds to repeat this process until it either assigns all data items to BGClients or runs out of BGClients. If the later, the coordinator assigns the remaining data items to one of the BGClients[5].

Figure 2.c shows D-Zipfian's produced probability with 1 and 3 BGClients and 12 data items. When compared with Figures 2.a and 2.b, D-Zipfian approximates the original distribution closely.

We use chi-square statistic to compare the distributions obtained with $N = 1$ with those obtained using

[4]Alternatively, with a deterministic technique to partition data items into clusters, each BGClient may execute the same technique independently to compute its m_k assigned objects.

[5]With the discussions of Section D.2, this is the fastest BGClient always.

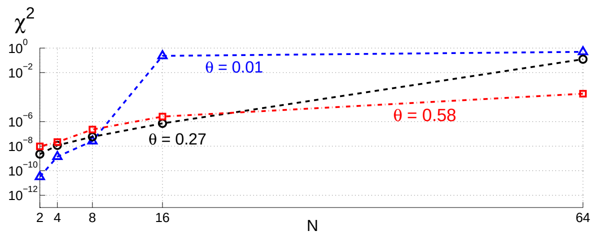


Figure 2: χ^2 analysis of centralized Zipfian with D-Zipfian as a function of N, M=10K.

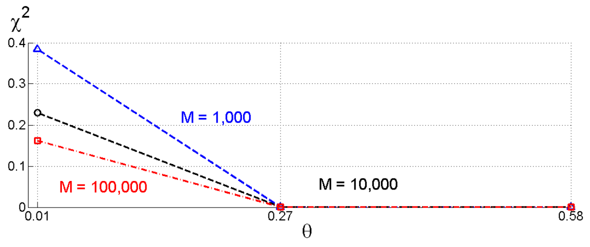


Figure 3: χ^2 analysis of centralized Zipfian with D-Zipfian as a function of θ with different number of data items, M.

$N > 1$. The chi-square statistic with $N > 1$ is computed as follows: $\chi^2 = \sum_{i=1}^{M} \frac{(q_i(M,\theta,N)-q_i(M,\theta,1))^2}{q_i(M,\theta,1)}$. A smaller value of χ^2 is more desirable. When $\chi^2 = 0$, it means the probability distribution with $N > 0$ is identical to that with $N = 1$.

Figure 2 shows the χ^2 statistic as a function of N BGClients with 10,000 data items and three different θ values. A smaller θ value results in a more skewed distribution. Obtained results show distributions with a handful of BGClients ($N \leq 8$) are almost identical to $N = 1$ as χ^2 value is extremely small. With tens of BGClients, the χ^2 value is greater because there is a higher chance of the sum of probabilities assigned to each BGClient to deviate from $\frac{1}{N}$. This is specially true with a more skewed distribution, θ=0.01. One way to enable D-Zipfian to better approximate a probability of $\frac{1}{N}$ for each BGClient is to increase the number of data items, M. This is shown in Figure 3 with three different values of M and θ=0.01. As we increase the value of M, the χ^2 statistic becomes smaller and approaches zero, showing a better match with a centralized implementation of Zipfian.

D.2 Heterogeneous BGClients

It is rare for one to purchase PCs that provide identical performance. As an example, on January 24, 2012, we purchased four identical Desktop computers from ZT systems configured with Intel i7-2600 processors, 16 Gigabyte of memory, and 1 TB of disk storage. When using them as BGClients, we observed one node to be considerably faster than the others. This fast node is almost twice faster than the slowest node. This discrepancy violates the assumption of Section D.1 that with N BGClients, each BGClient issues $\frac{1}{N}$ of

R_1	R_2	R_3	R_4	χ^2
1	1	2	2	0.11
1	1.25	1.5	2	0.07
1	2	2	2	0.06
1	1	1	2	0.12
1	4	4	4	0.16

Table 3: Processing rate of four BGClients and their impact on the χ^2 statistic, N=4, M=10K, θ=0.27.

R_1	R_2	R_3	R_4	χ^2
1	1	2	2	1.91E-08
1	1.25	1.5	2	1.49E-10
1	2	2	2	1.08E-09
1	1	1	2	1.19E-10
1	4	4	4	6.13E-09

Table 4: χ^2 improves dramatically with the refined D-Zipfian, N=4, M=10K, θ=0.27.

requests. This increases the error (χ^2) between the distributions observed with $N > 1$ and $N = 1$. As an example, Table 3 shows χ^2 observed with five different configurations of four heterogeneous BGClients. R_i denotes the rate at which a BGClient issues requests, see the first four columns of Table 3. The last column shows the χ^2 value when θ=0.27, comparing the observed theoretical[6] probabilities with 1 BGClient, i.e., $N = 1$. Each row corresponds to a different configuration of BGClients. For example, the first corresponds to a mix of 4 BGClients where two BGClients are twice faster than the other two BGClients. This results in errors (χ^2 values) significantly (orders of magnitude) higher than those shown in Figure 2.

To address this limitation, we change the first step of D-Zipfian (see Section D.1) to construct clusters for each BGClient such that their total assigned probability is proportional to the rate at which they can issue requests. Its details are as follows. Step 1 assigns objects to BGClient k with the objective to approximate a total probability of $\frac{R_k}{\sum_{j=1}^{N} R_j}$ for this BGClient (instead of $\frac{1}{N}$). With this change, the distribution with N BGClients becomes almost identical to that of one BGClient, see Table 4.

E Discussion

Section D.2 used the observed theoretical probabilities by considering the local probability of a data item in combination with the number of requests, $\frac{O_k \times p_i(M,\theta)}{O \times \sum_{j=1}^{m_k} p_j(M,\theta)}$. This study does not consider the actual generation of requests using a random number generator because it would require a too long a diversion from our main topic. We do wish to note that the considered probabilities are the foundation of generating requests and, without them, it is difficult (if impossible) to generate references that produce a Zipfian distribution. An

[6]We compute the observed theoretical probabilities by requiring each BGClient k to multiply its computed probabilities for a data item with its number of issued requests divided by the total number of requests issued by all the BGClients, $\frac{O_k \times p_i(M,\theta)}{O \times \sum_{j=1}^{m_k} p_j(M,\theta)}$.

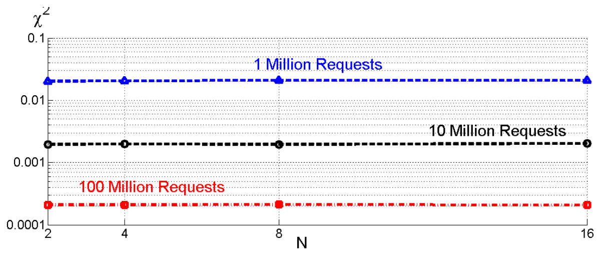


Figure 4: χ^2 analysis of an implementation of D-Zipfian generating requests. This analysis compares centralized Zipfian's probability for different data items with D-Zipfian as a function of different degrees of parallelism (x-axis). M=10,000, θ=0.27.

implementation of D-Zipfian with actual request generation is analyzed in Figure 4. The y-axis of this figure shows χ^2 statistic, quantifying the difference in observed probabilities with a centralized Zipfian when compared with D-Zipfian and different degrees of parallelism (x-axis). As we increase the number of issued requests, D-Zipfian resembles its centralized counterpart more closely.

With Section D.1, one may apply the concepts of Section D.2 to reduce the observed χ^2 values by several orders of magnitude and very close to zero. The idea is as follows. Once objects are assigned to the different BGClients, the number of references issued by a BGClient k is normalized relative to the total probability of its assigned objects. Thus, assuming the benchmark issues a total of O requests, each BGClient k would issue O_k requests:

$$O_k = O \times \frac{\sum_{i=1}^{m_k} p_i(M,\theta)}{\sum_{j=1}^{N} \sum_{i=1}^{m_j} p_i(M,\theta)} \tag{4}$$

While this enhances the χ^2 statistic dramatically, its potential usefulness is application specific. For example, a benchmarking framework may consist of a ramp-up, a ramp-down, and a steady state. Such a framework collects its observations during its steady state. The steady state might be defined as either a duration identified by conditions that mark the ramp-up and the ramp-down phases or a fixed number of requests. With the former, O is not known in advance and the system may not use Equation 4. Even when O exists, different values of O_k might be undesirable because different BGClients finish at different times. This is because participating nodes are assumed to be identical and those BGClients with the lowest O_k finish sooner, reducing the degree of parallelism.

We considered constructing V virtual BGClients ($V \geq N$) with several such BGClients mapped to one physical BGClient [8, 19, 16]. This is beneficial as long as it better approximates the quota assigned to each physical BGClient. In our experiments, we observed negligible improvement because approximating the appropriate quota for each virtual BGClient becomes more challenging as we increase the value of V, see discussions of Figure 2 in Section D.1.

F Conclusions

This paper presents D-Zipfian, a parallel algorithm that executes on N nodes and produces a Zipfian distribution that is independent of its degree of parallelism. D-Zipfian considers heterogeneity of participating nodes and the rate at which they produce requests in order to produce a distribution comparable to one node generating the distribution. D-Zipfian is decentralized and scales to a large number of nodes. It is an essential component of a scalable benchmarking framework (e.g., BG [3] or YCSB++ [13]) to test and tune the performance of a scalable data store. Its observed error (relative to one node) is dependent on how well it assigns data items to the participating nodes to approximate the quota of each node.

G Acknowledgment

We thank Jason Yap and anonymous reviewers of DBTest 2013 for their valuable comments.

References

[1] C. Aniszczyk. Caching with Twemcache, http://engineering.twitter.com/2012/07/caching-with-twemcache.html.

[2] Anon. A Measure of Transaction Processing Power. *Datamation*, April 1985.

[3] S. Barahmand and S. Ghandeharizadeh. BG: A Benchmark to Evaluate Interactive Social Networking Actions. *CoRR, Proceedings of 2013 CIDR*, abs/0913.1780, January 2013.

[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *INFOCOM*, pages 126–134, 1999.

[5] R. Cattell. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.*, 39:12–27, May 2011.

[6] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling Policies for an On-Demand Video Server with Batching. In *2nd ACM Multimedia Conference*, October 1994.

[7] R. Fan and N. Lynch. Gradient Clock Synchronization. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 320–327, 2004.

[8] S. Ghandeharizadeh and D. J. DeWitt. Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines. In *16th International Conference on Very Large Data Bases*, pages 481–492, 1990.

[9] S. Ghandeharizadeh and J. Yap. Cache Augmented Database Management Systems. In *DBSocial Workshop*, June 2013.

[10] K. Iwanicki, M. van Steen, and S. Voulgaris. Gossip-based Clock Synchronization for Large Decentralized Systems. In *Proceedings of the Second IEEE international conference on Self-Managed Networks, Systems, and Services*, pages 28–42, 2006.

[11] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, Jul 1978.

[12] D. L. Mills. On the Accuracy and Stablility of Clocks Synchronized by the Network Time Protocol in the Internet System. *SIGCOMM Comput. Commun. Rev.*, 20(1), December 1989.

[13] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. In *Cloud Computing*, New York, NY, USA, 2011. ACM.

[14] D. Patterson. For Better or Worse, Benchmarks Shape a Field. *Communications of the ACM*, 55, July 2012.

[15] Ratzel R and R. Greenstreet. Toward Higher Precision. *Commun. ACM*, 55(10):38–47, October 2012.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, August 2001.

[17] P. Saab. Scaling memcached at Facebook, https://www.facebook.com/note.php?note_id=39391378919.

[18] M. Seltzer, D. Krinsky, K. Smith, and X. Zhang. The Case for Application Specific Benchmarking. In *HotOS*, 1999.

[19] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *ACM SIGCOMM*, pages 149–160, San Diego, California, August 2001.

[20] M. Stonebraker. New Opportunities for New SQL. *Communications of the ACM, BLOG@ACM*, 55, November 2012.

[21] G. K. Zipf. Relative Frequency as a Determinant of Phonetic Change. Harvard Studies in Classified Philiology, Volume XL, 1929.