D-Zipfian: A Decentralized Implementation of Zipfian

Sumita Barahmand, Shahram Ghandeharizadeh
Computer Science Department
University of Southern California
Los Angeles, California
{barahman,shahram}@usc.edu

ABSTRACT
Zipfian distribution is used extensively to generate workloads to test, tune, and benchmark data stores. This paper presents a decentralized implementation of this technique, named D-Zipfian, using \( N \) parallel generators to issue requests. A request is a reference to a data item from a fixed population of data items. The challenge is for each generator to reference a disjoint set of data items. Moreover, they should finish at approximately the same time by performing work proportional to their processing capability. Intuitively, D-Zipfian assigns a total probability of \( \frac{1}{N} \) to each of the \( N \) generators and requires each generator to reference data items with a scaled probability. In the case of heterogeneous generators, the total probability of each generator is proportional to its processing capability. We demonstrate the effectiveness of D-Zipfian using empirical measurements of the chi-square statistic.

Categories and Subject Descriptors
C.4 [Performance of systems]: Measurement techniques, Modeling techniques; G.3 [Probability and statistics]: Distribution functions, Experimental design

General Terms
Design, Experimentation, Measurement, Performance

Keywords
Benchmarking, Zipfian distribution, Social networks, Distributed architectures

1. 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, 17, 13]. With most applications, a uniform random distribution of access to data items is typically not realistic due to Zipf’s law [20]. 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 data item with the lowest rank in the frequency table will occur more often than the data item with the second lowest rank, the data item with the second lowest rank in the frequency table will occur more often than the one with the third lowest rank, and so on and so forth. By manipulating the exponent \( \theta \) that characterizes the Zipfian distribution one may emulate different rules of thumb such as: 80% of requests (ticket sales [5], frequency of words [20], 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 [4, 19] data stores among others\(^2\) in support of interactive social networking actions and sessions. (See [4] 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 BG\( \text{Clients} \), used to generate the workload for the data store. The curve labeled “8 BG\( \text{Clients} \)” 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 BG\( \text{Client} \) may not be sufficient to rate them accurately [3]. To address this challenge, BG utilizes multiple BG\( \text{Clients} \) to generate work for its target data store. A coordinator, termed BG\( \text{Coord} \), manages these BG\( \text{Clients} \).

\(^{1}\)See Equation 1 in Section 2.
\(^{2}\)Such as cloud service providers and graph databases.
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 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 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 actions. It time stamps these to detect unpredictable reads. R-Zipfian would require BG to utilize synchronized clocks [10, 11, 6, 9, 14] to detect unpredictable reads. Both complexities are avoided by partitioning data items across BGClients.

With partitioning, BGCoord assigns a disjoint set of data items to each BGClient. A BGClient issues requests that reference its assigned 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 BGClients, each BGClient must reference data items such that the overall distribution of references conforms to a Zipfian distribution with a pre-specified \( \theta \). Moreover, 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 3, if each BGClient uses the original \( \theta \) with a subset of the population, 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 [8, 16, 1] with \( n_1 \) and \( n_2 \) BGClients (\( n_1 < n_2 \)) and observe a lower processing capability with \( n_2 \) because its distribution 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 \( N \).

The primary contribution of this paper is D-Zipfian, a novel technique that uses \( N \) parallel BGClients to issue requests that reference data items from a fixed population. Each BGClient references a disjoint set of data items. D-Zipfian ensures (1) the \( N \) BGClients finish at approximately the same time even when they issue requests at different rates, and (2) the overall distribution of references to the entire population is independent of \( N \) and conforms to a pre-specified \( \theta \).

The rest of this paper is organized as follows. Section 2 formalizes the problem statement to parallelize a Zipfian distribution. Section 3 presents two intuitive solutions and quantifies their limitations using a small population of data items. D-Zipfian is presented and quantified in Section 4. We discuss D-Zipfian in Section 5 and conclude in Section 6.

2. PROBLEM STATEMENT

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

\[
P_i(M, \theta) = \frac{\frac{1}{\theta}}{\sum_{m=1}^{M} \left( \frac{1}{m^{1+\theta}} \right)}
\]

where \( \theta \) characterizes the Zipfian distribution.

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 the 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 equal 1 because sum of the \( M \) probabilities equals one. 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 3.

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. 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 3 and D-Zipfian in Section 4. 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., \( O_k = 1 \). 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 3.

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 5. To the contrary, as detailed in Section 4.2, D-Zipfian manipulates \( O_k \) to require a mix of fast and slow BGClients to complete at approximately the same time. This is important because if one BGClient 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 4. The reader may skip to Section 4 for the final solution.

3. TWO NAÏVE APPROACHES

This section uses a small population consisting of twelve data items (\( M=12 \)) to demonstrate the concepts presented in Section 2. 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, \( \theta=0.01 \), 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 2 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 the original \( \theta \) 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 2.

Crude may assign data items to \( N \) BGClients in several other ways including:

- Hash (instead of range) partition data items using their id \( i \) to assign \( m_k \) data items to BGClient \( k \).
- Provide BGClient \( k \) with \( m_k \) assigned data items. Next, each BGClient would use the centralized implementation of Zipfian (see Section 2) with the entire population to reference a data item. If the referenced data item is not one of the \( m_k \) data items then BGClient \( k \) discards this request and generates a new one.

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 \( m_i = 4 \) and \( \theta=0.01 \). 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 \( i \) to normalize the probability of data items assigned to each BGClient. With one node, 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, the global probabilities produced by Normalized-Crude are more uniform than Crude, see Figure 2b 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 \( N = 3 \). Section 4...
shows that with a minor adjustment, Normalized-Crude is transformed into the final solution.

4. D-ZIPFIAN

We present D-Zipfian assuming BG Clients are homogeneous and produce requests at approximately the same rate. Subsequently, Section 4.2 extends the discussion to heterogeneous BG Clients that produce requests at different rates.

4.1 Homogeneous BG Clients

With $N$ BG Clients, 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 BG Client $k$, D-Zipfian overides the local probability of each data item $i$ as follows:

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

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 $\frac{1}{N}$.

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

Finally, it assigns cluster $k$ to BG Client $k$ by transmitting the identity of its data items to BG Client $k$. (A heuristic to construct clusters is described in the following paragraphs.)

In the second step, each BG Client adjusts the probability of its assigned data items using Equation 3. Note that the denominator of Equation 3 approximates $\frac{1}{N}$ because BG Coord assigned objects to each BG Client with the objective to approximate $\frac{1}{N}$. Finally, each BG Client uses its computed probabilities to generate array $A$ to produce data items, see Section 2. Generation of the requests by each BG Client is independent of the other BG Clients.

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

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

$^4$Alternatively, with a deterministic technique to partition data items into clusters, each BG Client may execute the same technique independently to compute its $m_k$ assigned objects.

$^5$With the discussions of Section 4.2, this is the fastest BG Client always.

<table>
<thead>
<tr>
<th>Data item $i$</th>
<th>Zipfian/Crude with $N = 1$</th>
<th>Normalized-Crude</th>
<th>Crude with $N = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.319014585</td>
<td>0.319014585</td>
<td>0.477558748</td>
</tr>
<tr>
<td>2</td>
<td>0.160616755</td>
<td>0.479631343</td>
<td>0.159444731</td>
</tr>
<tr>
<td>3</td>
<td>0.107512881</td>
<td>0.587442241</td>
<td>0.160944731</td>
</tr>
<tr>
<td>4</td>
<td>0.080866966</td>
<td>0.668011191</td>
<td>0.121056305</td>
</tr>
<tr>
<td>5</td>
<td>0.046469017</td>
<td>0.732849284</td>
<td>0.121056305</td>
</tr>
<tr>
<td>6</td>
<td>0.029705152</td>
<td>0.874163368</td>
<td>0.121056305</td>
</tr>
<tr>
<td>7</td>
<td>0.02753426</td>
<td>0.910396882</td>
<td>0.121056305</td>
</tr>
<tr>
<td>8</td>
<td>0.02753426</td>
<td>0.910396882</td>
<td>0.121056305</td>
</tr>
<tr>
<td>9</td>
<td>0.02753426</td>
<td>0.910396882</td>
<td>0.121056305</td>
</tr>
<tr>
<td>10</td>
<td>0.02753426</td>
<td>0.910396882</td>
<td>0.121056305</td>
</tr>
<tr>
<td>11</td>
<td>0.02753426</td>
<td>0.910396882</td>
<td>0.121056305</td>
</tr>
<tr>
<td>12</td>
<td>0.02753426</td>
<td>0.910396882</td>
<td>0.121056305</td>
</tr>
</tbody>
</table>

Table 1: Example with 12 data items and $\theta=0.01$.
We compute the observed theoretical probabilities by requiring each BGClient \( k \) to multiply its computed probabilities by 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 not impossible) to generate references that produce a Zipfian distribution. An implementation of D-Zipfian with actual request generation is analyzed in Figure 5. The y-axis of this figure shows \( \chi^2 \) 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 4.1, one may apply the concepts of Section 4.2 to reduce the observed theoretical 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 (\( \chi^2 \) values) significantly higher than those shown in Figure 3.

To address this limitation, we change the first step of D-Zipfian (see Section 4.1) 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 \( \sum_{j=1}^{N} R_j \) 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 3.

5. DISCUSSION

Section 4.2 used the observed theoretical probabilities by considering the local probability of a data item in combination with the number of requests, \( \frac{O_k \times p_r(M, \theta)}{O \times \sum_{j=1}^{N} R_j(M, \theta)} \). 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 not impossible) to generate references that produce a Zipfian distribution. An implementation of D-Zipfian with actual request generation is analyzed in Figure 5. The y-axis of this figure shows \( \chi^2 \) 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 4.1, one may apply the concepts of Section 4.2 to reduce the observed \( \chi^2 \) 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:

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

While this enhances the $\chi^2$ 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 [7, 18, 15]. 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 3 in Section 4.1.

6. CONCLUSIONS

This paper presents D-Zipfian, a parallel algorithm that executes on $N$ nodes that reference a mutually exclusive subset of data items and produce a Zipfian distribution that is independent of $N$. D-Zipfian considers the rate at which nodes issue requests in order to produce a distribution comparable to one node generating the distribution. This technique is an essential component of a scalable benchmarking framework (e.g., BG [3] or YCSB++) to evaluate the performance of a scalable data store.

7. ACKNOWLEDGMENTS

We thank Jason Yap and our anonymous reviewers for their insights and valuable comments.

8. REFERENCES