Load balancing with dynamic set of balls and bins

Aamand, Anders; Knudsen, Jakob Bæk Tejs; Thorup, Mikkel

Published in:
STOC 2021 - Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing

DOI:
10.1145/3406325.3451107

Publication date:
2021

Document version
Publisher's PDF, also known as Version of record

Document license:
Other

Citation for published version (APA):
Load Balancing with Dynamic Set of Balls and Bins

Anders Aamand
aa@di.ku.dk
BARC, University of Copenhagen
Denmark

Jakob Bæk Tejs Knudsen
jakn@di.ku.dk
BARC, University of Copenhagen
Denmark

Mikkel Thorup
mikkel2thorup@gmail.com
BARC, University of Copenhagen
Denmark

ABSTRACT
In dynamic load balancing, we wish to distribute balls into bins in an environment where both balls and bins can be added and removed. We want to minimize the maximum load of any bin but we also want to minimize the number of balls and bins that are affected when adding or removing a ball or a bin. We want a hashing-style solution where we given the ID of a ball can find its bin efficiently.

We are given a user-specified balancing parameter \( \varepsilon = 1 + \varepsilon \), where \( \varepsilon \in (0, 1) \). Let \( n \) and \( m \) be the current number of balls and bins. Then we want no bin with load above \( C = \lfloor cn/m \rfloor \), referred to as the capacity of the bins.

We present a scheme where we can locate a ball checking \( 1 + O(\log(1/\varepsilon)) \) bins in expectation. When inserting or deleting a ball, we expect to move \( O(1/\varepsilon) \) balls, and when inserting or deleting a bin, we expect to move \( O(C/\varepsilon) \) balls. Previous bounds were off by a factor \( 1/\varepsilon \).

The above bounds are best possible when \( C = O(1) \) but for larger \( C \), we can do much better: Let

\[
 f = \begin{cases} 
   \varepsilon C & \text{if } C \leq \log 1/\varepsilon \\
   \varepsilon C \cdot \sqrt{\log(1/(\varepsilon \sqrt{C})} & \text{if } \log 1/\varepsilon \leq C < \frac{1}{2\varepsilon^2} \\
   1 & \text{if } C \geq \frac{1}{2\varepsilon^2} 
\end{cases}
\]

We show that we expect to move \( O(1/f) \) balls when inserting or deleting a ball, and \( O(C/f) \) balls when inserting or deleting a bin. Moreover, when \( C \geq \log 1/\varepsilon \), we can search a ball checking only \( O(1) \) bins in expectation.

For the bounds with larger \( C \), we first have to resolve a much simpler probabilistic problem. Place \( n \) balls in \( m \) bins of capacity \( C \), one ball at the time. Each ball picks a uniformly random non-full bin. We show that in expectation and with high probability, the fraction of non-full bins is \( \Theta(f) \). Then the expected number of bins that a new ball would have to visit to find one that is not full is \( \Theta(1/f) \). As it turns out, this is also the complexity of an insertion in our more complicated scheme where both balls and bins can be added and removed.

CCS CONCEPTS
• Mathematics of computing → Probabilistic algorithms.
• Theory of computation → Data structures design and analysis.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

STOC '21, June 21–25, 2021, Virtual, Italy
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8053-9/21/06... $15.00
https://doi.org/10.1145/3406325.3451107

1 INTRODUCTION
Load balancing in dynamic environments is a central problem in designing several networking systems and web services [12, 25]. We wish to allocate clients (also referred to as balls) to servers (also referred to as bins) in such a way that none of the servers gets overloaded. Here, the load of a server is the number of clients allocated to it. We want a hashing-style solution where we given the ID of a client can efficiently find its server. Both clients and servers may be added or removed in any order, and with such changes, we do not want to move too many clients. Thus, while the dynamic allocation algorithm has to always ensure a proper load balancing, it should aim to minimize the number of clients moved after each change to the system. For every update in the system, we need to change the allocation of clients to servers. For simplicity, we assume that the updates (ball and bin insertions and removals) do not happen simultaneously and will be operated one at a time, so that we have time to finish changing the allocation before we get another update. Such allocation problems become even more challenging when we face hard constraints in the capacity of each server, that is, each server has a capacity and the load may not exceed this capacity. Typically, we want capacities close to the average loads.

There is a vast literature on solutions in the much simpler case where the set of servers is fixed and only the client set is updated. For now, we focus on solutions that are known to work in our fully-dynamic case where both clients and servers can be added and removed in an arbitrary order. This rules out solutions where only the last added server may be removed. The above problem formulation is very general, and does not assume anything about the ratio between the number of clients \( n \), and the number of servers \( m \). Processors are cheap, so one could for instance imagine systems with a large number of servers. However, it is also conceivable having a system with many clients or a balanced system with \( n = m \).

The classic solution to the scenario where both clients and servers can be added and removed is Consistent Hashing [12, 25] where the current clients are assigned in a random way to the

\[ \text{In particular, this rules out the external memory techniques [16] where blocks (playing the role of fixed capacity servers) can only be added to and removed from the top of the current memory.} \]
current servers. While consistent hashing schemes minimize the expected number of movements, they may result in hugely over-loaded servers, and they do not allow for explicit capacity constraints on the servers. The basic point is that the load balancing of consistent hashing [12, 25] is no better than a random assignment of clients to servers. The same issue holds for Highest Random Weight Hashing (popularly known as Rendezvous Hashing) [26]. Hence, with $n$ clients and $m$ servers, we expect good load balancing if $n/m = \omega(\log m)$, but the balance is lost with smaller loads, e.g., with $n \approx m$, we expect many servers to be overloaded with $\Theta(\log m / \log \log m)$ clients.

More recently, Mirroknii et al. [19] presented an algorithm that works with arbitrary capacity constraints on the servers. For the purpose of load balancing, the system designer can specify a balancing parameter $c = 1 + \epsilon$, guaranteeing that the maximum load is at most $\lceil cn/m \rceil$. While maintaining this hard balancing constraint, they limit the expected number of clients to be moved when clients or servers are inserted or removed. From a more practical perspective, we think of the load balancing parameter $c = 1 + \epsilon$ as a simple knob which captures the tradeoff between load balancing and stability upon changes in the system. This gives a more direct control to the system designer in meeting explicit balancing constraints.

Even without capacity constraints, the obvious general lower bounds for moves are as follows. When a client is added or removed, at least we have to move that client. When a server is added or removed, at least we have to move the clients belonging to it. On the average, we therefore have to move least $\frac{n}{m}$ clients when a server is added or removed.

With the algorithm from [19], while guaranteeing a balancing parameter $c = 1 + \epsilon \leq 2$, when a client is added or removed, the expected number of clients moved is $O(\frac{n}{f})$. When a server is added or removed, the expected number of clients moved is $O(\frac{n}{f^2 m})$. These numbers are only a factor $O(\frac{1}{f})$ worse than the general lower bounds without capacity constraints. For balancing parameter $c \geq 2$, the expected number of moves is increased by a factor $1 + O(\frac{\log c}{c})$ over the lower bounds. This implies that for superconstant $c$, we only expect to pay a negligible cost in extra moves.

Focusing on the challenging case where $c = 1 + \epsilon \leq 2$, we present an algorithm which reduces the number of moves by a factor $1/\epsilon$. When inserting or deleting a ball, we expect to move $O(1/\epsilon)$ balls, and when inserting or deleting a bin, we expect to move $O(C/\epsilon)$ balls. To search a ball we only need to consider $O(\log(1 + 1/\epsilon))$ “consecutive” bin

With $C := cn/m$, these bounds are essentially best possible when $C = O(1)$ is a constant. However, for larger $C$, we can do even better. In order to explain, this we first have to consider the following much simpler probabilistic problem: Consider placing $n$ balls in $m$ bins, each of capacity $C = (1 + \epsilon)n/m$, one ball at the time, where each ball picks a uniformly random non-full bin. We are interested in the number of non-full bins. To our surprise, this relatively simple statistic does not seem to have been analyzed before, and so, we believe our bounds to be of independent interest. To state our bounds, we define

$$f = \begin{cases} \epsilon C \sqrt{\log(1/\epsilon)} & \text{if } C \leq \log 1/\epsilon \\ \epsilon C / \sqrt{\log(1/\epsilon)} & \text{if } \log 1/\epsilon \leq C < \frac{1}{2\epsilon^2} \\ 1 & \text{if } C \geq \frac{1}{2\epsilon^2} \end{cases}$$

whenever $0 < \epsilon \leq 1$ and $C \geq 1$ is integral. We are going to prove the following result.

**Theorem 1.1.** Let $n, m \in \mathbb{N}$ and $0 < \epsilon < 1$ be such that $C = (1 + \epsilon)n/m$ is integral. Moreover assume that $1/\epsilon = m^o(1)$. Suppose we distribute $n$ balls sequentially into $m$ each of capacity $C$, for each ball choosing a uniformly random non-full bin. The expected fraction of non-full bins is $\Theta(f)$.

How does this result relate to our dynamic load allocation problem? We can think of the distribution scheme in the theorem as the algorithmically weakest way to assign the balls to the capacitated bins. Here, by algorithmically weak, we mean that it cannot be implemented in the dynamic setting where balls and bins can come and go. However, it is still helpful to think of it as the mathematically ideal way of solving dynamic load allocation with bounded loads in the following sense. Imagine that an insertion of a ball is carried out by repeatedly choosing a random bin until we find a non-full one where we place the ball. Then we avoid all the unpleasant dependencies between the loads of the bins visited during the insertion that arise in algorithmically stronger schemes. For example, one can compare to a scheme like linear probing where the cascading effect of balls causes heavy dependencies between the loads of bins visited during a search or an insertion. It follows from Theorem 1.1 that in the simple scheme above, the expected number of bins visited when making an insertion is $O(1/f)$. The main contribution of this paper is to present a much stronger scheme which supports general insertions and deletions of both balls and bins, and which, nonetheless, achieves complexity bounds that are analogous to those in the mathematically ideal scheme above. To be precise, with our scheme, we expect to move $O(1/f)$ balls when inserting or deleting a ball, and $O(C/f)$ balls when inserting or deleting a bin and this is tight. Similar bounds hold on the number of bins visited when performing any of these updates. Our main technical challenge is handling all the intricate dependencies that arise in the much more complicated probabilistic setting in our scheme.

**Applications.** Consistent hashing has found numerous applications [11, 20] and early work in this area [12, 24, 25] has been cited more than ten thousand times. To highlight the wide variety of areas in which similar allocation problems might arise, we mention a few more important references to applications: content-addressable networks [21], peer-to-peer systems and their associated multicast applications [7, 23]. Our algorithm and that from [19] are very similar to consistent hashing, and should work for most of the same applications, bounding the loads whenever this is desired. In fact, the algorithm from [19] already found two quite different industrial applications; namely Google’s cloud system [18] and Vimeo’s video streaming [22]. Both systems had to handle the lightly loaded case. Also, in both cases, load balancing was not an objective to maximize, but rather a hard constraint, e.g., in the Vimeo blog post [22], Rodland describes how no server is allowed to be overloaded, and how he found a load balancing parameter $c = 1.25$ to be satisfactory.
for Vimeo’s video streaming. We shall return to this later. With our algorithm, we get the same load balancing but with much fewer reallocations.

1.1 Background: Consistent Hashing

The standard solution to our fully-dynamic allocation problem is consistent hashing [12, 25]. We shall use it as a starting point for our own solution, so we review it below.

Simple Consistent Hashing. In the simplest version of consistent hashing, we hash the active balls and bins onto a unit circle, that is, we hash to the unit interval, using the hash values to create a circular order of balls and bins. Assuming no collisions, a ball is placed in the bin succeeding it in the clockwise order around the circle. One of the nice features of consistent hashing is that it is history-independent, that is, we only need to know the IDs of the balls and the bins and the hash functions, to compute the distribution of balls in bins. If a bin is closed, we just move its balls to the succeeding bin. Similarly, when we open a new bin, we only have to consider the balls from the succeeding bin to see which ones belong in the new bin.

With \( n \) balls, \( m \) bins, and a fully random hash function \( h \), each bin is expected to have \( n/m \) balls. This is also the number of balls we expect to move when a bin is opened or closed.

One problem with simple consistent hashing as described above is that the maximum load is likely to be \( \Theta(\log m) \) times bigger than the average. This has to do with a big variation in the coverage of the bins. We say that bin \( b \) covers the interval of the cycle from the preceding bin \( b' \) to \( b \) because all balls hashing to this interval land in \( b \). When \( m \) bins are placed randomly on the unit cycle, on the average, each bin covers an interval of size \( 1/m \), but we expect some bins to cover intervals of size \( \log m \). Thus the maximum load is thus expected to be a factor \( \Theta(\log m) \) above the average.

A related issue is that the expected number of balls landing in the same bin as any given ball is almost twice the average. More precisely, consider a particular ball \( x \). Its expected distance to the neighboring bin on either side is exactly \( 1/(m + 1) \), so the expected size of the interval between these two neighbors is \( 2/(m + 1) \). All balls landing in this interval will end in the same bin as \( x \); namely the bin \( b \) succeeding \( x \). Therefore we expect \( 2(n − 1)/(m + 1) \approx 2n/m \) other balls to land with \( x \) in \( b \). Thus each ball is expected to land in a bin with load almost twice the average. If the load determines how efficiently a server can serve a client, the expected performance is then only half what it should be.

In [12] they addressed the above issue using so called virtual bins. We will also employ these virtual bins in our solution and describe them below.

Consistent Hashing with Virtual Bins. To get a more uniform bin cover, [12] suggests the use of virtual bins. The virtual bin trick is that the ball contents of \( k = \Theta(\log m) \) virtual bins is united in a single super bin. The super bins are the \( m \) bins seen by the user of the system. Internally it is the \( km \) virtual bins we place on the cycle together with the \( n \) balls. Each virtual bin has a pointer to its super bin. To place a ball, we go along the cycle to the first virtual bin, and then we follow the pointer to its super bin.

A super bin covers the union of the intervals covered by its \( k \) virtual bins. The point is that for any constant \( \epsilon > 0 \), if we pick a large enough \( k = \Theta(\log m) \), then with high probability, each super bin covers a fraction \( (1 + \epsilon)/m \) of the unit cycle.

We note that many other methods have been proposed to maintain such a uniform bin cover as bins are added and removed (see, e.g., [6, 9, 13, 14, 17, 26]), and in our algorithms, we shall also employ such virtual bins.

With a uniform bin cover, balls distribute uniformly between bins. On the positive side, in the heavily loaded case when \( n/m \) is large, e.g., \( n/m = \Theta(\log m) \), all loads are \( (1 + o(1))n/m \), w.h.p. However, with \( n = m \), we still expect many bins with \( \Theta((\log m)/(\log \log m)) \) balls even though the average is 1. In this paper, we aim for good load balancing for all possible load levels.

1.2 Simple Consistent Hashing with Bounded Loads

As we mentioned earlier, Mirrokni et al. [19] presented an algorithm that works with arbitrary capacity constraints on the bins. For the purpose of load balancing, the system designer can specify a balancing parameter \( \epsilon = 1 + \epsilon \), guaranteeing that the maximum load is at most \( C = \lceil cn/m \rceil \).

Their idea is very simple. As in simple consistent hashing, we place balls and bins randomly on a cycle, but instead of placing balls in the first bin along the cycle, we place them in the first non-full bin. Thus we can think of the distribution as first placing all the bins on the cycle, and then placing the balls one-by-one, putting each in the first non-full bin found by going in clockwise around the cycle. If we have hash functions for placing arbitrary balls and bins along the cycle, and if we have a priority order on all balls, telling us the order in which we insert balls, then this completely determines the placement of any set of the balls in any set of capacitated bins. This also includes rolling adjustment of the capacities relative to average load \( n/m \). Instead of giving all bins the maximal capacity \( C = \lceil cn/m \rceil \), they always have \( \lceil cn/m \rceil - m \lceil cn/m \rceil \) bins with capacity \( \lceil cn/m \rceil \). The only exception is that we never drop any capacity below 1. A hash function choose which bins have which capacities, and this ensures that only few capacities have to be changed with each system update. In Mirrokni et al. [19] they show that their results hold, both when capacities are adjusted to \( \pm \epsilon \), and when a joint capacity \( C \) is given, defining \( \epsilon = Cm/n - 1 \). In this
paper, for simplicity, we will focus on the latter model with fixed capacities.

Mirrokni et al. [19] also provided an analysis of their system. With \( \varepsilon \leq 1 \), they showed that starting from the hash location of any ball, the expected number of full bins passed on the way to the first non-full bin is \( O(1/\varepsilon^2) \). From this they get that the expected number of balls that has to be moved when a ball is inserted or deleted is \( O(1/\varepsilon^2) \). Likewise, the expected number of balls that has to be moved when a bin is inserted or deleted is \( O(C/\varepsilon^2) \). These bounds are all tight for simple consistent hashing with bounded loads.

Finally, Mirrokni et al. [19] also discussed many potentially relevant techniques that could possibly be made to work for fully-dynamic load balancing where both balls and bins can be added and removed, and with strict requirements on the maximal load for each bin. In these comparisons, their scheme was the one with the best proven bounds on the number of moves needed in connection with the updates.

**Faster Searches.** Mirrokni et al. [19] states that to search a ball, they have to consider \( O(1/\varepsilon^2) \) bins, but using an old trick [3, 15], this is easily improved to \( O(1/\varepsilon) \). The idea is that when we search for a ball, we can stop as soon as we reach a bin that is not filled with balls of higher priority. This helps the searches if the priorities are random. We shall use the idea later, so let’s elaborate. The bins considered in the search are exactly the bins from the bin hashed to and till the first non-full bin if only the balls of higher priority was inserted. Let \( r(q, m, C) \) be expected number of bins considered if there are \( q \) balls of higher priority, and \( m \) bins of capacity \( C \). Then with \( n \) balls in total, the expected cost with random priorities is \( \sum_{q=0}^{n} r(q, m, C)/(n + 1) \). The analysis in [19] implies \( r(q, m, C) = O(1/\varepsilon^2) \) where \( \varepsilon_q = C/q^2 \), implying an expected cost of \( O(1/\varepsilon) \) with random priorities.

We note that random priorities do not help with updates, for if we, say, want to insert a ball, and meet a bin that is full including balls of lower priority, then we have to place the lowest priority ball in a later bin. However, finding the established server of a client if any, is often the most frequent operation in the system, so a faster search is very important in practice. As stated, a similar analysis gives that for our system, we have to consider fewer bins when searching than when inserting a ball. In particular, we only need to consider \( O(1) \) bins in expectation when \( C \geq \log 1/\varepsilon \).

### 1.3 Our Scheme: Consistent Hashing with Virtual Bins and Bounded Loads

Our algorithm basically just combines the bounded loads with virtual bins. When a ball is placed in a virtual bin, it is also placed in its super bin which has a limited capacity. In the following, we describe two different versions of our scheme. The first one, described in Section 1.3.1, is conceptually the simplest to understand and easier to analyze mathematically. It is this version that we will analyze in the main body of the paper. The second one, described in Section 1.3.2, is the version most suitable to be implemented in practice for several reasons to be described. Our results hold for both implementations we will sketch how to derive the results for the second more practical version in the full version of the paper [2].

Common to both versions is that we fix some natural number \( k \), which is the number of virtual bins for each super bin.

#### 1.3.1 Mathematically Clean Version: Many Independent Cycles

For this version, we hash each super bin to \( k \) different cycles or levels using \( k \) independent hash functions\(^2\). The \( k \) hash values on the \( k \) cycles will be the associated virtual bins of the given super bin. We also hash the balls to the cycles, but contrary to the bins, each ball gets just a single random hash value on a single random cycle.

The static placement of the balls can be described as follows: We start by placing all balls which hash to the first cycle using standard consistent hashing with bounded loads as described in Section 1.2. We assume that we have priorities on the balls and we will simulate that they are inserted in priority order. After the first level, the balls hashing to this level have thus been distributed into the virtual bins and we put them in the corresponding super bins. Initially, each super bin had capacity \( C \). If the virtual bin of such a super bin received a balls at the first level, its new capacity is then reduced accordingly to \( C – a \). We continue this process on level \( i = 2, \ldots, k \).

At level \( i \), each super bin has a certain remaining capacity and we use standard consistent hashing with bounded loads (with these capacities) to place the balls at level \( i \) into the virtual bins and thus, into the corresponding super bins. If a super bin had capacity \( C_0 \) before the hashing to level \( i \), and it received \( a \) balls at level \( i \), its remaining capacity for the next levels is \( C_0 – a \). Traversing the levels one at a time like described, corresponds to enforcing that regardless of the initial priorities of the balls, if two balls hash to different levels, the ball hashing to the lower level will have the highest priority of the two. With these modified priorities, the static image at a given point can be obtained by simply inserting the balls one by one in priority order, placing each ball in the first virtual bin whose super bin is not full. This completely describes the placement of balls in bins if we know the hash functions and the priority order, so the system is history-independent as described in [5].

Searching for a ball \( x \) is almost the same as for normal consistent hashing. We calculate the hash value of \( x \) and visit the virtual bins starting from that hash value in cyclic order until we either find \( x \) in a corresponding super bin or we meet a ball of lower priority hashing to the same level.

Insertions are a bit more complicated. For inserting a ball \( x \) we calculate \( h(x) \) which in particular indicates the level, \( i \), that \( x \) hashes to. We traverse level \( i \) starting at \( h(x) \) until we meet a bin, \( b \), which either (a) is not full or (b) contains a ball of lower priority than \( x \) (all balls hashing to levels \( j > i \) have lower priority than \( x \) by convention). We insert \( x \) in \( b \). In case (a), the insertion is complete, but in case (b) we pop \( y \) from \( b \) and recurse the insertion starting with \( y \) (which happens at some level \( j \geq i \)).

Ball deletions are symmetric to ball insertions in the sense that the hash functions tells us exactly the placement of all balls in bins, both before and after the ball which we are to insert or delete is inserted or deleted. Deleting a bin is the same as re-inserting all balls in it, and inserting a bin is symmetric to deleting a bin. Therefore we get that the number of balls to be moved is essentially

\[^2\text{For simplicity, we advice the reader to think of all our hash functions as fully random. However, our results hold even when the hashing is implemented with the practical mixed tabulation from [5]. We will explain how to modify our proofs to show this in the full version of the paper[2].}\]
determined by the number that has to be moved in connection with an insertion (we shall discuss this in more detail later).

For most of our results, we will assume that the hashing of balls to the different levels is uniform, but in Section 2 we will see an applications where the probability of hashing to level \( i \) is \( 1/2^i \) for \( 1 \leq i \leq k-1 \) and \( 2^{-k+1} \) for \( i = k \). In this setting we already obtain a big improvement over standard consistent hashing using just \( \log 1/\epsilon \) levels.

1.3.2 Practical Version: A Single Linear Order. We next describe the more practical implementation of our algorithm and here we will also give more details on the concrete ranges of the hash functions. As will be seen, it is very similar to the version above having some minor alterations. For this implementation all balls and all virtual bins are hashed to a single range, which we think of not as a cyclic order but rather as a linear order. In order to describe the static image at given point, we would again consider the balls one by one in priority order, placing each ball in the first virtual bin whose super bin is not full. Again, this ensures that the system is history-independent.

We now provide some more details on the hash functions and the priority order. Generally the hash values are in some universe \([u] = \{0, \ldots, u-1\}\). We imagine \( u \) to be so large that we expect no collisions between hash values (if there are ties, we can break them in favour of the ID’s of the balls, but we will ignore this detail). We also think of both balls and bins having ID’s in \([u]\).

We have a single hash \( h: [u] \rightarrow [u] \) describing the location of the balls. We also use \( h \) to give the random priority order of the balls, inserting those with smallest hash values first.

For the super bins, and for some parameter \( k \), each bin has \( k+1 \) associated virtual bins. Their hash locations are described via \( k+1 \) hash functions \( h_i: [u] \rightarrow [u], i \in [k] = \{0, \ldots, k\} \). We assume that \( k \) divides \( u \), e.g., that both are powers of two, and we restrict \( h_i \) to map uniformly into \([iu/k, (i+1)u/k]\). This way each super bin gets exactly one virtual bin in each of the \( k+1 \) intervals \([iu/k, (i+1)u/k]\). Having this spread is important because of the priority order of the balls, which implies that virtual bins with larger hash values are more likely to be full.

The last interval \([u, u+u/k)\) is outside the normal hash range \([u]\). These last virtual bins will pick up any key that did not end in a bin in the normal range \([u]\). Since every super bin is represented in \([u, u+u/k),\) all balls are picked up unless there are more balls than the total capacity. As a result, we do no longer think of balls and bins as hashing to a cycle, but just to a linearly ordered universe with an extra set of representative virtual bins by the end making sure that all balls get placed.

We briefly explain why this system is preferable in practice. The first reason is that when using the hash values of the balls as their priorities we obtain a very simple description of the static distribution of balls in the bins: We may simply insert the balls in order from lowest to highest hash value, always placing the ball in the first non-full bins. A way of picturing this is to imagine that the balls of lower hash values are “pushing” balls of higher hash values ahead of them. On a line, it is very easy to implement this comparison as a standard comparison between hash values. In fact, it is possible to obtain a similar image for cycles, but for this one needs to impose a cyclic priority order of the balls hashing to a given level, and performing comparisons for such a cyclic order is a bit more technical to implement\(^3\). If on the other hand, we decided to stick with the linear priority order on each cycle, thus giving up on the nice image from above, we still encounter some technical issues with the implementation. With searches and insertions, everything works fine, but the issues come up when deleting balls and inserting bins. For instance, when deleting a ball which is placed in the “last” bin on the cycle, we may have to pull back balls that have been forwarded from this bin to the “first” bins in the cycle, and for deciding if such balls are to be pulled back, we have to use a different comparison of hash values. Thus, even with linear priorities the cyclic probing still muddies the implementation and makes it less efficient.

Again, we shall play a bit with the ranges of the hash functions for the virtual bins. However, they will always partition \([u]\) consecutively with the range of \( h_{i-1}\), following the range of \( h_i\) with \( i \in [k-1]\). With the exponentially decreasing hash ranges described by the end of Section 1.3.1, \( h_i\), maps uniformly to \([u-u/2^i, u-u/2^{i+1})\) for \( i \in [k-1]\) and \( h_{k-1}\) maps uniformly to \([u-u/2^{k-1}, u)\). As above \( h_k\) is special, mapping to \([u, u+u/k]\).

Searches and insertions have similar descriptions to the ones given in Section 1.3.1. Moreover, the history independence again implies that deletions are symmetric to insertions. Finally, deleting a bin corresponds to inserting the ball in the bin, and inserting a bin is symmetric to the deletion of the bin.

1.4 Main Results on Consistent Hashing

We now state our main results on consistent hashing with bounded loads and virtual bins.

1.4.1 \( O(1/\epsilon) \) Reallocated Balls, with \( \log 1/\epsilon \) Levels. Our first result, to be proved in Section 2, uses a logarithmic number of virtual bins to achieve that the number of bins visited during an insertion (and thus the number of reallocated balls) is \( O(1/\epsilon) \). It uses a non-uniform distribution of the balls to the different levels, with the probability of a ball hashing to level \( i \) being \( 2^{-i} \) for \( 1 \leq i \leq k-1 \) and \( 2^{-k+1} \) for \( i = k \).

Theorem 1.2. Let \( 0 < \epsilon < 1 \) and suppose that we distribute \( n \) balls into \( m \) bins each of capacity \( C = (1+\epsilon)n/m \) using consistent hashing with bounded loads and \( k = \lceil \log(1/\epsilon) \rceil \) levels, where the probability, \( p_i \), that a ball hashes to level \( i \) is

\[
p_i = \begin{cases} 2^{-i}, & 1 \leq i \leq k-1 \\ 2^{-k+1}, & i = k. \end{cases}
\]

Assume that \( 1/\epsilon = n^{\Omega(1)} \). When inserting or deleting a ball, we expect to visit (and hence move) \( O(1/\epsilon) \) balls, and when inserting or deleting a bin, we expect to move \( O(C/\epsilon) \) balls. Finally, when searching a ball, we expect to visit \( O(\log 1/\epsilon) \) bins.

In the previous system of simple consistent hashing with bounded loads, but no virtual bins, Mirrokni et al. [19] proved that ball insertions and deletions are expected to move \( O(1/\epsilon^2) \) balls while bin insertions and deletions are expected to move \( O(C/\epsilon^2) \) balls. Those bounds are a factor \( 1/\epsilon \) worse than ours. Mirrokni et al. [19] would

\(^3\)For example, for just two balls, the notion of one hashing before the other is not well defined.
also perform searches considering \( O(1/\varepsilon^2) \) bins in expectation, but using the trick of assigning random priorities to the balls, one can get down to \( O(1/\varepsilon) \) bins in expectation, still without the use of virtual bins. Combining our scheme using virtual bins, with the trick of random priorities the expected number of bins visited during a search drops exponentially to \( O(\log 1/\varepsilon) \), as stated in the theorem.

When proving Theorem 1.2, the main technical challenge is bounding the expected number of bins visited during an insertion of a ball. In fact, the remaining parts of the theorem follow once we have this bound. Indeed, the result on searches follows by using the trick described by the end of Section 1.2. Moreover, the result on deletions follows since the history independence implies that a deletion is symmetric to an insertion. A deletion of a bin corresponds to the insertion of all the balls in that bin and one can use linearity of expectation to argue about the desired result. Finally, the insertion of a bin is symmetric to its deletion by the history independence. There are some details, and we refer the reader to the full version [2] for these details.

1.4.2 Better Bounds when the Capacities Are Large. In classic consistent hashing without virtual bins, we obtain no advantage when the number of balls \( n \) are much larger than the number of bins \( m \), or in other words, when the capacity of a bin, \( C \), is large. The basic issue is that most of the uncertainty in the system without virtual bins stems from the uncertainty in the distance between a bin and its predecessor, which determines the expected number of balls hashing directly to the bin.

However, the use of virtual bins improves the concentration of the number of balls hashing directly to a super bin, and we obtain an advantage of this improved concentration. This was in fact the whole point of introducing virtual bins in classic consistent hashing without load bounds [25]. To be precise, fix \( k = A(\log n)/\varepsilon^2 \) for some appropriately large constant \( A \). Then standard Chernoff bounds show that each bin cover a fraction \( (1 + \lambda)\varepsilon/m \) of the combined hash range, where \( \lambda \) can be made arbitrarily small (by increasing \( A \)). If further the average load \( m/n \) is above \( k \), then with high probability, no bin gets load above \( C = (1 + \varepsilon)mn/n \) by balls hashing directly to them. In particular, all load bounds are satisfied without the having to forward a single ball. The result below (which is the main result of our paper) asymptotically settles the expected insertion time for general \( C \), in particular for any \( C \leq (\log n)/\varepsilon^2 \). Before stating the theorem, we encourage the reader to recall the definition of \( f \) in Eq. (1).

**Theorem 1.3.** Let \( 0 < \varepsilon < 1 \) and suppose that we distribute \( n \) balls into \( m \) bins each of capacity \( C = \frac{(1 + \varepsilon)n}{m} \) using consistent hashing with bounded loads and \( k = c/\varepsilon^2 \) uniform levels for a sufficiently large constant \( c \). Assume that \( 1/\varepsilon = n^{O(1)} \). In expectation we move \( O(1/f) \) balls when inserting or deleting a ball, and \( O(C/f) \) balls when inserting or deleting a bin. Finally, when searching a ball, we expect to visit \( O(1) \) bins when \( C \geq \log 1/\varepsilon \) and \( O(\log 1/\varepsilon) \) bins when \( C < \log 1/\varepsilon \).

Our bounds in Theorem 1.3 show that we do get an advantage from bigger capacities even when \( C \) is smaller than \( k = \Theta((\log n)/\varepsilon^2) \). In fact, already for \( C = 1/\varepsilon^2 \), the expected insertion time drops to \( O(1) \).

Again, the hardest part of proving Theorem 1.3, is bounding the expected number of bins visited during an insertion by \( O(1/f) \), and the remaining parts will follow using similar arguments as for Theorem 1.2. We provide the details in the full version [2].

**High Probability Bounds.** Theorems 1.2 and 1.3 only bound the expected number of balls moved during the insertions and deletions of balls and bins. However, it is also possible to obtain high probability bounds. We will provide such high probability bounds in a later full version of the paper.

1.4.3 Distributing Balls Randomly into Capacitated Bins. To understand the strength of our bounds, we consider a much simpler problem where we place \( n \) balls in \( m \) bins, each of capacity \( C = (1 + \varepsilon)n/m \), one ball at the time. Each ball picks a uniformly random non-full bin. Letting \( X \) denote the fraction of non-full bins, we show in Section 3 that \( \mathbb{E}[X] = \Theta(f) \) and \( X = \Theta(f) \) with high probability. Surprisingly, this relatively simple question has not been studied before.

What is the idea of considering this simpler distribution scheme? With a fraction of \( X \) non-full bins, the expected number of random bins visited in order to find one of the non-full ones is \( 1/X \). This is reminiscent to searching for a non-full bin using (any variation of) consistent hashing with bounded loads, except that we get rid of the intricate dependencies which arise in the more complicated schemes that can handle both insertions and deletions. In this way, the scheme above can be thought of as the simplest way of achieving the desired load balancing, but of course it has no chance of working in a fully dynamic setting. We thus obtain, the same complexity bounds as the weakest system imaginable, at the same time being able to handle both insertions and deletions of balls and bins.

1.4.4 The Practical Implementation with Mixed Tabulation. When proving Theorems 1.2 and 1.3, we will assume that our scheme is implemented as described in Section 1.3.1 and, moreover, using fully random hash functions. In the full version [2], we will explain why our results hold even with the more practical implementation from Section 1.3.2. We will also sketch how one can obtain the same results with the practical mixed tabulation scheme from [8]. In the implementation with mixed tabulation, we would use \( k \) independent mixed tabulation hash functions for the hashing of virtual bins, and a single independent mixed tabulation hash function for the hashing of balls.

1.5 The Model and its Applicability

Consistent hashing with or without virtual bins is a simple versatile scheme that has been implemented in many different systems with different constraints and performance measures [11, 20]. The most classic implementation of consistent hashing is the distributed system Chord [24, 25] which has more than ten thousand citations. The Chord papers [24, 25] give a thorough description of the many issues affecting the design. On the high level, they have a system of pointers so that given an arbitrary hash location, they can find the next bin in the clockwise order using \( O(\log n) \) messages. This is how they find the (virtual) bin a ball hashes to. In simple consistent hashing, this is where the ball is to be found. With virtual bins, there are additional pointers between virtual bins and their super bins that we can follow using \( O(1) \) messages. In fact, Chord does
maintain explicit successor pointers between neighboring (virtual) bins, so we only have to pay $O(1)$ extra messages to find a next bin along the cycle.

As described by Mirrokni et al. [19], the successor pointers give immediate support for forwarding in case of capacitated bins. Mirrokni et al. only used this forwarding for simple consistent hashing without virtual bins, and this has been adopted both by Google’s Cloud Pub/Sub [18] and Vimeo [22]. Both systems had to handle the lightly loaded case. Also, in both cases, load balancing was not an objective to maximize, but rather a hard constraint, e.g., in the Vimeo blog post [22], Rodland describes how no server is allowed to be overloaded, and how he found a load balancing parameter $c = 1 + \varepsilon = 1.25$ to be satisfactory for Vimeo’s video streaming.

The successor pointers in Chord work equally well for moving between virtual bins. In fact, Rodland from Vimeo has told (personal communication) the last author, Thorup, that their system does allow a combination of virtual bins and bounded loads, like what we suggest in this paper, so a system similar to ours is already running. Thorup had the general idea from much earlier (around the time of the first versions of [19]), but deriving the mathematical understanding, presented here in Theorem 1.3 took several years.

Let us now consider the time to search a ball in a Chord-like setting. By Theorem 1.3, we expect to consider $O(\log(1/f))$ consecutive virtual bins with associated super bins. Finding the virtual bin succeeding the hash location uses $O(\log n)$ messages while each other bin is found with $O(1)$ messages. Then our message bottleneck is actually to find the first virtual bin.

Now it could be the case that balls/clients themselves remembered if they are in the system, and if so, what bin/server they belonged to. The latter requires that they are notified if they get moved due to other updates in the system, e.g., if their bin/server was removed.

Another way to circumvent the $O(\log n)$ messages for placing the hash location would be if we for some $m = \Theta(m)$, placed the reference points $p_i = ui/m$, $i \in \hat{m}$, in the doubly-linked list of virtual bins. For a ball $x$ its hash reference point is $P_{[h(x)\mod/m]}$. Regardless of system updates, it could remember its reference point, and from there follow in expectation $O(1)$ successor pointers to get the current virtual bin succeeding its real hash location. The reference points could be updated by background rebuilding to be ready every time $m$ is halved or doubled, thus maintaining an $\hat{m}$ approximating $m$ within a factor of 2.

In fact, our scheme is equally relevant for less distributed systems than Chord. In Google’s Cloud Pub/Sub [18], the most important aspects of the system was (1) that it has good load balance (2) that only few clients/balls have to be moved in connection with update, that is, a ball or bin insertion or deletion, and (3) history independence so that the placement of balls in bins can be computed by anyone knowing the hash functions and the current set of balls and bins. The fact that each system update only leads to few moves implies that even if we have a few mistakes in the set of balls and bins, then this only implies a few mistakes in the placement of balls in bins.

System updates, inserting or deleting a ball or a bin are hopefully not too frequent. As mentioned in [18], the dominant concern is the actual reallocation of balls between bins; for in the real world, this means moving clients between servers disrupting service etc. Theorems 1.2 and 1.3 give us concrete bounds on how many balls we expect to move.

The computation of which balls are to be moved in connection with updates depends very much on the situation. As in [18], thanks to history independence, we can compute the balls to be moved from scratch. We know the update to the set of balls and bins, and the hash functions tell us exactly which balls are placed in which bins before and after update. The difference tells us exactly which balls have to be moved. This solution if fine if the computation cost is small compared with the cost of actually moving the clients.

Alternatively, we may want a more distributed local identifia-
tion of the moves as in in the Chord system. This is fairly straight-
forward for insertions, and we already described it earlier. It does,
however, get a bit more complicated for the other updates, and we
shall return to such a distributed implementation in Section 1.6.

Stepping back, we offer a generic scheme for a load balanced distribution of balls in bins when both can be added and removed. We are not claiming to have a theoretical model that captures all the important aspects of performance since this depends very much on the concrete implementation context. Our main contribution is a theoretical analysis of combinatorial parameters described in Theorems 1.2 and 1.3.

### 1.6 Computing Moves Locally in a Distributed Environment

We will now discuss how we could compute which balls have to be moved in connection with system updates in a distributed Chord-type system. Recall that sometimes it may be fast enough to identify the moves more centrally, simply by computing the placement of the balls in the bins before and after the update, and just identify the difference. However, in this subsection, we will discuss how to identify the moves locally, not spending much more time than the number of moves specified in Theorems 1.2 and 1.3.

We already discussed how to insert balls, but we want to do it in a way that also makes it fast and easy to delete balls. The basic idea to make deletions efficient is that we for every virtual bin store the number of balls that have passed it. More precisely, each bin has a pass count that starts at zero when there are no balls. We now consider the process where balls are inserted in priority order, each just placed in the first virtual bin with a non-empty super bin. This increases the count on all the virtual bins between the hash location and the virtual bin the ball ends in. Each super bin will also store which of its virtual bins that have a positive pass count.

The above pass counts are quite easy to maintain when balls arrive to the real system, that is, not in priority order. To see this, we review the insertion of a ball, adding when pass counts should be incremented. To insert a new ball, we first hash it to some location which also determines its priority. Starting from the hash location, we visit the virtual bins following, each time looking in the corresponding super bin. If the super bin is not full, we simply place the ball in it and terminate the insertion. If the super bin is filled with balls of higher priority, we increment the pass count of the virtual bin, and continue to the next virtual bin. However, if the super bin is filled and contains a ball of lower priority, we insert the new ball and pop the ball of lowest priority. The popped ball
belongs to some virtual bin, which could be the same, but could also be only much later in the linear order than the virtual bin we just came from. The pass count is incremented from whichever virtual bin we pop the ball from, and then we recursively insert the popped ball, continuing from the next virtual bin. The $O(1/f)$ bound from Theorem 1.3 actually bounds not only the number of moves, but also the number of bins considered during the above insertion.

Next we consider the deletion of a ball. Essentially, we just want to reverse the above process, systematically finding the balls the ball to be deleted have displaced. We think of deletions as first removing a ball, and then recursively, filling a hole. Finding the ball to be removed is easy, as described before, and when we remove it, we will have to decrement the pass count on all the virtual bins between its hash location and up to the virtual bin before the one it landed in. Next we want to see if we can refill the whole. Assuming that the bin we removed was in the level $i$ virtual bin of a super bin. We now check corresponding super bin $b$ to see if any ball has been displaced by the ball we deleted. This is the case if and only if at least one of its virtual bins has a positive pass count. Let $j$ be the lowest level of a virtual bin with a positive pass count. It is not hard to see that we must have $j \geq i$. We now consider the virtual bins following the level $j$ virtual bin until we find a ball with hash location before $h_j(b)$. The virtual bins passed decrease their counts, and then we recursively delete the ball. As described above, our total work is within a constant factor of the symmetric insertion, that is, we consider $O(1/f)$ bins and spend $O(1/f)$ time in total.

We now consider the insertion of deletion of super bins. We think of these super bin or server updates as more rare than the ball or client updates.

Deleting a super bin $b$ is relatively easy. Essentially, we just reinsert all the balls in it. A small detail is that if a ball $x$ was in the level $j$ virtual bin, then we insert it starting from $h_j(b)$ rather than from $h(x)$. This can only save work over the regular insertion of $x$ and in particular, this means that we do not increase the pass count for virtual bins between $h(x)$ and $h_j(b)$. By Theorem 1.3, the expected number of balls that has to be moved when deleting a super bin is $O(C/f)$. However, on top of that, we do have to spend at least $O(k)$ time on removing the $k$ virtual bins from the system.

Inserting a super bin $b$ is a bit more complicated. We would like to just fill it as we filled the holes arising when deleting a ball, but we have the issue that we do not know the pass counts for the $k$ virtual bins representing the new super bin. To handle this, for $i = 1, \ldots, k$, we first find the hash location $h_i(b)$ of its virtual bin $b_i$, which takes $O(\log n)$ messages, including inserting it in the linked list of virtual bins. Next consider the virtual bin $u$ following $b_i$. If bin $u$ has no ball and pass count zero, then we can just set the pass count of $h_i(b)$ to zero. Otherwise, we continue along the virtual bins, counting the balls in them, until we find a ball that has hash after $h_i(b)$. All but the last ball are the balls that have passed the level $i$ virtual bin $b_i$, which now gets a pass count. Now that we have the pass count, we can move those balls to $b_i$, as long as super bin $b$ has space for them, using the same procedure as described under deletions of balls.

We now first analyze the number of bins considered to compute the pass counts of the virtual bins $b_i$. We note that the bins considered are exactly the same as if we searched for a ball that hashed to $h_i(b)$. Now consider instead the case where we first generate a random $i \in [k]$, and then generate $h_i(b)$. With $i$ random, $h_i(b)$ is uniformly random in $[u]$, and then the expected number of bins considered is exactly the same as those considered in the search of a ball with hash value uniformly random in $[u]$. We conclude that the expected total number of bins considered over all $i \in [k]$ is exactly $k$ times bigger. Thus, by Theorem 1.3, we expect to consider at most $O(k^{\log(1/\epsilon)}(1/\epsilon))$ bins when $C \leq \log 1/\epsilon$, and only $O(k)$ bins when $C \geq \log 1/\epsilon$. Now that the pass counts are fixed, inserting a bin is symmetric to deleting it and has the same cost, yielding a bound of $O(C/f)$.

### 1.7 Dynamic Load Capacities

We now also consider what happens when we use self-adjusting capacities like Mirrokni et al. [19]. Below, the capacitated bins correspond to our super bins. Rather than fixed capacities, the user of the system specifies a balancing parameter $c = (1 + \epsilon)$ and then the maximal capacity is $C = \lfloor cn/m \rfloor$. We do not want all bins to change capacity each time $cn/m$ passes an integer.

Instead, as in Mirrokni et al. [19], assuming an arbitrary fixed ordering of the super bins, we let the lowest $q = \lfloor cn/m \rfloor - m - q$ have capacity $C = \lfloor cn/m \rfloor$ while the remaining $r = m - q$ have capacity $C - 1$. We refer to the former bins as big bins and the latter bins as small bins, though the difference is only 1. Moreover, as an exception to the above rule, we will never let the capacity drop below 1, that is, if $cn < m$, then all bins have capacity 1.

The basic point in the above system is that a ball update changes at most $[c] = O(1)$ bin capacities while a bin update changes at most $O(C)$ capacities. Switching the capacity from large to small has the same effect as inserting an extra high priority ball in the super bin while leaving the capacity at $C$. In the other direction, switching the capacity from small to large corresponds to a deletion of an extra high priority ball.

From an analysis perspective, this means that we are essentially studying a system with $n' = r + n$ balls in bins of capacity $C$ where $Cm = (cn'/m)$. In our analysis, this corresponds to having a $1$-th level which puts exactly one ball in each of $r$ bins, $1$-th level to the rest. Such a perfect level poses no issues for the analysis. Thus the cost per capacity change is the same as that of regular insertions/deletions, and therefore have no effect on our overall bounds.

A small point, elaborated in Mirrokni [19], is that for all the bounds to hold, we may always do things in the order that maximizes capacity in every step, so that we always have a total capacity of $Cm = [c(n + g)/m]$. For example, when inserting a ball, we increase capacities before inserting, while deleting a ball, we decrease capacities last. Likewise for a bin insertion, we insert it before decreasing capacities, while when deleting a bin, we start by increasing the capacities.

### 1.8 Roadmap of the Paper

In Section 2, we prove the part of Theorem 1.2 concerning insertions of balls. That the statements about ball deletions and bin insertions and deletions follow is covered in the full version [2]. The proof is not too involved and as such it serves as a nice warm-up for understanding and analysing our consistent hashing with bounded
loads and virtual bins. We encourage the reader to study this proof before diving into the much more complicated proof of Theorem 1.3.

Next, we switch our attention to the improved bounds that can be obtained with larger values of the quantity $C$, starting with the proof of Theorem 1.1 in Section 3. In fact, we will not need Theorem 1.1 for the proofs of the later results, but we have three other reasons to include it. First, as we described, it is useful to think of the simple distribution scheme in the proof as the mathematical ideal way of solving the load allocation problem with bounded loads, in the sense that it removes all the intricate dependencies that arise in the more powerful schemes. Without it, Theorem 1.3 would not be nearly as interesting. Second, it again serves as a good warm-up for Theorem 1.3. Indeed, the main ideas and steps when bounding the number of non-full bins by $O(f)$ in Theorem 1.3 are the same, but the more complicated distribution on the number number of balls landing in a given bin has been poses many more technical difficulties. Third, we believe the distribution scheme in Theorem 1.1 to be a natural one and that understanding it is interesting in its own right.

Due to space limitations, we have to defer the complete proof of Theorem 1.3 to the full version of the paper [2]. However, we will discuss the main ideas and techniques of the proof. The proof is split in two parts. First, we obtain a bound of $O(f)$ on the number of non-full bins which holds with high probability. Second, we analyse the number of bins visited during an insertion by employing this bound. We discuss the main ideas for performing these steps in Section 4.

The statements for ball deletions, bin insertions, bin deletions, and search times in Theorems 1.2 and 1.3 are deferred to the full version [2]. The same is the case for the analysis of the practical implementation described in Section 1.3.2 and the arguments why our results continue to hold when the hashing is implemented with mixed tabulation.

2 EXPECTED $O(1/\epsilon)$ INSERTION TIME WITH $[\log(1/\epsilon)]$ LEVELS

In this section we prove the part of Theorem 1.2 concerning insertions, restated below. We will assume that we use the implementation described in Section 1.3.1 but the result also holds with the other implementation in Section 1.3.2 (see the full version [2]).

**Theorem 2.1.** Suppose that we distribute $n$ balls into $m$ bins each of capacity $C = (1 + \epsilon)n/m$ using consistent hashing with bounded loads and $k = [\log(1/\epsilon)] + 2$ levels, where the probability, $p_i$, that a ball hashes to level $i$ is

$$p_i = \begin{cases} 2^{-i}, & 1 \leq i \leq k - 1 \\ 2^{-k+1}, & i = k. \end{cases}$$

Assume that $1/\epsilon = n^{o(1)}$. The expected number of bins visited when inserting a ball is then $O(1/\epsilon)$.

We remark that one way to implement the above hashing is by using an auxiliary hash function $s : U \to [2^{k+1}]$. Letting $h_1, \ldots, h_k$ denote the hash functions distributing balls at level 1, $\ldots, k$, the hash value of a key $x$ is then given by $h_i(x)$, where $i$ is the number of leading 0’s of $s(x)$.

Proof. Let $Z$ denote the number of virtual bins visited in total and $Z_i$ denote the number of virtual bins visited at level $i \in \{k\}$. Then $Z = \sum_{i=1}^{k} Z_i$. We will show that $\mathbb{E}[Z_i] = O(2^i)$ from which it follows that $\mathbb{E}[Z] = O(2^k) = O(1/\epsilon)$.

First, it follows from a standard Chernoff bound that if $X_i$ is the number of balls hashing to level $i$ and $\mu_i = \mathbb{E}[X_i] = pn$, then for $\delta \leq 1$,

$$\Pr[|X_i - \mu_i| \geq \delta \mu_i] \leq \exp(-\delta^2 \mu_i/3)$$

Thus, it holds that $|X_i - \mu_i| = O(\sqrt{\mu_i \log n})$ with probability at least $1 - n^{-3}$. Similarly, if $X_{<i} = \sum_{j<i} X_j$ and $\mu_{<i} = \sum_{j<i} \mu_j$, it holds that $|X_{<i} - \mu_{<i}| = O(\sqrt{\mu_i \log n})$ with the same high probability.

For each $j \in \{m\}$, we define $C_j$ to be the remaining capacity of bin $j$ after the distribution of balls to levels $1, \ldots, i-1$. Then $\sum_{j \in \{m\}} C_j = (1 + \epsilon)n - X_{<i}$, so it follows from the above that with probability $1 - O(n^{-2})$,

$$\sum_{j \in \{m\}} C_j \geq (1 + \epsilon)n - \mu_{<i} - O\left(\sqrt{\mu_{<i} \log n}\right)$$

$$= (1 + \epsilon)n - O(\sqrt{n \log n}).$$

For $i < k$ we have that $\mu_i = 2^{-i}n$, so it follows that, $\sum_{j \in \{m\}} C_j \geq 2X_i$ with probability $1 - O(n^{-2})$, where we used the assumption that $1/\epsilon = n^{o(1)}$. In the case $i = k$, we instead have that

$$X_k \leq 2^{-k+1}n + O(\sqrt{n \log n}) \leq \epsilon n/2 + (\sqrt{n \log n})$$

with probability at least $1 - O(n^{-2})$, so again $\sum_{j \in \{m\}} C_j = X_k + \epsilon n \geq 2X_k$, again using that $1/\epsilon = n^{o(1)}$.

Now fix $i \in \{k\}$, and write $C_j = C_j^{(l)}$ for simplicity. Let $\mathcal{E}$ denote the event that $pn/2 \leq X_i \leq 2pn$ and that $\sum_{j \in \{m\}} C_j \geq 2X_k$. Then $\Pr[\mathcal{E}^c] = O(n^{-2})$, so that

$$\mathbb{E}[Z_i] \leq \mathbb{E}[|Z_i| \mid \mathcal{E}] + \mathbb{E}[|Z_i| \mid \mathcal{E}^c] \Pr[\mathcal{E}^c] \leq \mathbb{E}[Z_i] \mid \mathcal{E} + O(1).$$

Thus, it will suffice to show that $\mathbb{E}[Z_i] \mid \mathcal{E} = O(2^i)$. Let $b$ be the first bin visited at level $i$, i.e., during the insertion we set some level $j \leq i$ arrived at bin $b$ and $b$ is not full after the hashing of bins to level $1, \ldots, i-1$. Let $I$ be a maximal interval at level $i$ containing $b$ and satisfying that all bins lying in $I$ are full at level $i$. Let $R$ denote the number of bins in $I$ excluding $b$. Then $Z_i \leq R + 1$. We will show that $\mathbb{E}[R] = O(2^i)$ (for notational convenience the conditioning on $\mathcal{E}$ has been left out). Let $s \in \mathcal{H}$ be given and let $A_s$ denote the event that $s = 1 \leq R \leq 2s$. We are now going to provide an upper bound on $\Pr[A_{s}]$. Let $I_s$ and $I_s^+$ be the intervals respectively ending and starting at $b$ and of lengths $s/2m$. Similarly, let $I_s^-$ and $I_s^-$ be the intervals respectively ending and starting at $b$ and of lengths $s/2m$. Let $I_1 = I_s^- \cup I_s$ and $I_2 = I_s^+ \cup I_s$. Finally, partition $I_2$ into 54 intervals of equal lengths, $J_1, \ldots, J_{54}$. Let $a$ be such that $1-a/(1+a) = 5/6$ (or $a = 1/11$) and $C = \frac{1}{54} \sum_{j \in \{m\}} C_j$. We claim that if $A^h$ holds then either of the following events must be true

$B_1$: $I_s^-$ contains at most $2s$ virtual bins different from $b$.

$B_2$: $I_s^+$ contains at least $s/2$ virtual bins different from $b$. 

1270
B3: The total capacity of bins different than $b$ hashing to $J_f$ is at most $(1-a)sC/9$ for some $1 \leq \ell \leq 54$.

B4: The total number of balls hashing to $J_f$ is at least $(1+\delta)sC/18$ for some $1 \leq \ell \leq 54$.

To see this, suppose that $A_1$ occurs but neither of $B_1, B_2, B_3$ occurs. We show that then $B_4$ must occur. As $R \leq 2s$ and $B_1$ did not occur, $I_1 \leq I_2$. As $R \geq s + 1$ and $B_2$ did not occur, either $I_1^c \subseteq I$ or $I_1^+ \subseteq I$. Letting $\ell$ denote the number of $j$ such that $J_j \subseteq I$ it therefore follows that $\ell \geq 3$. Since $B_3$ did not occur, the total capacity of bins hashing to $I$ is at least $\ell C(1-a)\Sigma$. Finally, since all balls which ends up in a bin in $I$ must have hashed to $I$ it follows that the total number of balls hashing to $I$ is at least $\ell C(1-a)s/9\ell$. In particular, for some $1 \leq \ell \leq 54$, at least $\ell C(1-a)s/(\ell + 2)$ balls must hash to $J_f$. But $\ell C(1-a)s/(\ell + 2) \geq \ell C(1-a)s/15 \geq \ell C(1+a)s/18$, so we conclude that $B_4$ holds.

Simple Chernoff bounds gives that inequality give that $\Pr[I_1^c] = \exp(-\Omega(\ell))$ and $\Pr[I_1^c] = \exp(-\Omega(\ell))$. To bound $\Pr[B_3]$, let $\ell \in [54]$ be fixed and define $Y_j$ to be the indicator for bin $j$ hashing to $J_f$. Further, define $Y = \sum_{i\in[m]} Y_i; \text{ then } \mathbb{E}[Y] = \frac{C}{9}$. This time however, we only have that $|Y_j| \leq C$, so applying Chernoff we obtain that

$$\Pr[B_3] = \Pr[Y \leq (1-a)\mathbb{E}[Y]] \leq \exp(-\Omega(\mathbb{E}[Y]/C)) = \exp(-\Omega(s/2^i))$$

For $B_4$, note that since we conditioned on $E$, the expected number of balls hashing to an interval $J_f$ is $\frac{\ell C(1-a)s}{9\ell} \leq \frac{\ell C}{18}$. Thus, another Chernoff bound yields that $\Pr[B_4] = \exp(-\Omega(sC))$. Note that $C \geq 1/2^i$, so that we in particular have that $\Pr[B_4] = \exp(-\Omega(s/2^i))$. Combining our bounds, it follows that for $s \geq 2^i$,

$$\Pr[A_1] = \exp(-\Omega(s/2^i)).$$

Now we can upper bound

$$\mathbb{E}[R] \leq 2^i + \sum_{j=0}^{\infty} \Pr[A_{2^i+j}]2^{2^i+j+1} \leq 2^i + 2^{i+1}\sum_{j=1}^{\infty} \exp(-\Omega(2^i))2^j = O(2^i),$$

as desired. This completes the proof. □

3 BALLS INTO CAPACITATED BINS

In this section we prove Theorem 1.1. Let us start by recalling the setting of the theorem. We let $n, m \in \mathbb{N}$ and $\varepsilon$ be given with $0 < \varepsilon < 1$ and suppose that we sequentially distribute $n$ balls into $m$ bins, each of capacity $C = (1 + \varepsilon)n/m$. For simplicity, we assume that $n, m$ and $\varepsilon$ are such that $C$ is a positive integer. Each ball is placed in a uniformly random non-full bin, where a bin is full if it contains precisely $C$ balls. The theorem claims that if $1/\varepsilon = m^{o(1)}$, then the expected fraction of non-full bins is $\Theta(1)$, where

$$f = \left\{ \begin{array}{ll} \ell C, & \text{if } C \leq \log(1/\varepsilon), \\ \ell C \log(1/\varepsilon)/\varepsilon^{1/2}, & \text{if } \log(1/\varepsilon) < C \leq 1/2^{2\varepsilon}, \\ 1, & \text{if } 1/2^{2\varepsilon} \leq C. \end{array} \right.$$
with high probability. Let $\gamma > 1$ be a constant to be fixed. We are going to split the argument into three cases.

**Case 1:** $C \leq \gamma \log(1/e)$. We will show that in this case, the expected fraction of non-full bins is $\Theta(cC)$. To do this, we first show the following technical claim.

**Claim 1.** If $C \leq \gamma \log(1/e)$, then $\mathbb{E}[T] = (1 + \Omega(1))n$.

**Proof of Claim.** Start by fixing a bin $j \in [m]$ and consider throwing $m \log(1/e)/2$ balls into $m$ bins. The probability that bin $j$ is empty is

$$\left(1 - \frac{1}{m}\right)^{m \log(1/e)/2} = \Omega(\sqrt{e}).$$

As we now argue, it follows that when throwing $N \geq m \log(1/e)/2$ balls into $m$ bins uniformly at random, a given bin receives at most $N/m - \log(1/e)/2$ balls with probability $\Omega(\sqrt{e})$. For this, we use the results of [10], stating that if $X \sim B(k, p)$ is binomially distributed with $p < 1 - 1/k$, then $\mathbb{P}[X \leq \mathbb{E}[X]] > 1/4$. Combining this result with the above, we obtain that the given bin receives none of the first $m \log(1/e)/2$ balls with probability $\Omega(\sqrt{e})$ and at most $(N - m \log(1/e)/m) = N/m - \log(1/e)/2$ of the remaining balls with probability at least $1/4$. Moreover, these events are independent, happening simultaneously with probability $\Omega(\sqrt{e})$, which gives the desired.

Now let $N = n \log(1/e)m/4$ and define $Z = \sum_{j \in [m]} Z_j^{(N)}$. From the above observation, it follows that

$$\mathbb{E}[Z] \leq Cm - \Omega(\sqrt{e} \log(1/e)m),$$

and by applying Lemma 3.3 it follows that it similarly hold with high probability that $Z \leq Cm - \Omega(\sqrt{e} \log(1/e)m)$, with a potentially larger implicit constant in the $\Omega$-notation. Assuming that $\epsilon$ is smaller than a sufficiently small constant we therefore have that with high probability,

$$Z \leq (C - \gamma \epsilon \log(1/e))m \leq C(1 - \epsilon)m = (1 + \epsilon)(1 - \epsilon)n < n.$$

Thus $T > N$ with high probability, but this also means that

$$\mathbb{E}[T] \geq n + \frac{\log(1/e)m}{4} \geq n + \frac{Cm}{4\gamma} = n\left(1 + \frac{1 + \epsilon}{4\gamma}\right) = n(1 + \Omega(1)),$$

as desired. □

Using the claim and Corollary 3.2 it follows that also $T = (1 + \Omega(1))n$ with probability $1 - n^{-\gamma}$ for any constant $\gamma$ and that $|T - \mathbb{E}[T]| = O\left(\frac{\sqrt{m} \log n}{\epsilon^2}\right)$ with the same high probability.

We now choose $N = \mathbb{E}[T] + O\left(\frac{\sqrt{m} \log n}{\epsilon^2}\right)$ so large that $Pr[T \geq N] \leq n^{-2}$. Then $N = (1 + \Omega(1))n$ as well. Consider a bin $j \in [m]$ and let $A_k = [X_j^{(N)} = k]$ for each $k \geq 0$. Then

$$Pr[A_k] = \binom{N}{k} \frac{1}{m^k} \left(1 - \frac{1}{m}\right)^{N-k}.$$ 

If $k = N^{1/2 - \Omega(1)}$, then simple calculus yields that $Pr[A_k]$ can be approximated with the Poisson distribution with mean $\mu = N/m$ as follows,

$$Pr[A_k] = (1 + o(1)) \left(1 - \frac{1}{m}\right)^{N/k} = (1 + o(1)) \frac{\mu^k}{k!} e^{-\mu}.$$ 

In particular, this holds when $k \leq C$. Thus, for any $k \leq C$ it holds that

$$Pr[A_k] \leq (1 + o(1)) \frac{\mu^k}{k!} e^{-\mu} \leq \left(1 + \Omega(1)\right) \frac{n}{Cm} \leq \frac{1 + \Omega(1)}{1 + \epsilon} = 1 + \Omega(1),$$

where the last inequality requires that $\epsilon$ is smaller than a sufficiently small constant which we may assume. Let $\alpha = \Omega(1)$ be the implicit constant in the $\Omega$-notation above, such that for $k \leq C$ (and $n, m, 1/\epsilon$ sufficiently large), we have that $Pr[A_k] / Pr[A_{k-1}] \geq 1 + \alpha$. It follows that,

$$Pr[Y_j^{(N)} < C] = \sum_{k=1}^{C} Pr[A_{C-k}] \leq \sum_{k=1}^{C} (1 + \alpha)k^{-1} Pr[A_{C-1}] = O(Pr[A_{C-1}]).$$

and

$$\mathbb{E}[C - Y_j^{(N)}] = \sum_{k=1}^{C} k Pr[A_{C-k}] \leq \sum_{k=1}^{C} (1 + \alpha)^{k-1} Pr[A_{C-1}] = O(Pr[A_{C-1}]).$$

It trivially holds that $Pr[Y_j^{(N)} < C] \geq Pr[A_{C-1}]$ and $\mathbb{E}[C - Y_j^{(N)}] \geq Pr[A_{C-1}]$, so we have proved that $Pr[Y_j^{(N)} < C] = \Omega(Pr[A_{C-1}])$ and $\mathbb{E}[C - Y_j^{(N)}] = \Omega(Pr[A_{C-1}])$. By linearity of expectation,

$$\mathbb{E}\left[\sum_{j \in [m]} C - Y_j^{(N)}\right] = \Omega(m Pr[A_{C-1}]) = \Omega(m Pr[Y_j^{(N)} < C]).$$

Now with probability at least $1 - n^{-2}$, it holds that $N - O\left(\frac{\sqrt{m} \log n}{\epsilon^2}\right) \leq T \leq N$. Since $T$ is chosen such that $\sum_{j \in [m]} C - Y_j^{(T)} = \epsilon n$, it follows that

$$\mathbb{E}\left[\sum_{j \in [m]} C - Y_j^{(T)}\right] = \Theta(\epsilon n).$$

Thus, combining (2) and (3), we obtain that $Pr[Y_j^{(N)} < C] = \Theta(\epsilon C)$. Finally,

$$Pr[Y_j^{(T)} < C] \geq Pr[Y_j^{(N)} < C] - Pr[N < T] = \Omega(\epsilon C) - n^{-2} = \Omega(\epsilon C).$$

Using the exact same argument but instead choosing $N = \mathbb{E}[T] + O\left(\frac{\sqrt{m} \log n}{\epsilon^2}\right)$ so small that $Pr[T \leq N] \leq n^{-2}$, we obtain that $Pr[Y_j^{(T)} < C] = \Omega(\epsilon C)$, so in fact $Pr[Y_j^{(T)} < C] = \Theta(\epsilon C)$. But $Pr[Y_j^{(T)} < C]$ is independent of $j$ and exactly the expected fraction of non-full bins. Thus the proof is complete in the case $C \geq \gamma \log(1/e)$.

**Case 2:** $\gamma \log(1/e) < C \leq \frac{1}{\sqrt{e}}$. To make the argument work, we will assume that $\gamma = O(1)$ is sufficiently large. We can make this assumption since the argument from case 1 holds for any $\gamma = O(1)$. In general, the argument from case 1 serves as a nice warm up but for the present case we have to be more careful in our estimates. Again, we choose $N = \mathbb{E}[T] + O\left(\frac{\sqrt{m} \log n}{\epsilon^2}\right)$ so large that
Pr[T ≥ N] ≤ n⁻² and put µ = N/m. Let us state by proving some crude bounds on N as stated in the following claim.

**Claim 2.** If γ = O(1) is sufficiently large, then C + √C ≤ N ≤ m ≤ 3C/2.

**Proof.** We first prove the lower bound. Suppose for contradiction that N/m < C + √C. Then

\[ \mathbb{E} \left[ \sum_{j \in [m]} C - Y_j^{(N)} \right] = \Omega(\sqrt{C}m) = \Omega(n/\sqrt{C}) ≥ 2m, \]

using that γ is sufficiently large. This contradicts the fact that with high probability T ≤ N. For the upper bound, note that if N/m ≥ 3C/2, then for any j \in [m],

\[ Pr[X_j^{(N)} ≤ C] ≤ \exp \left( -\frac{N}{18m} \right) \leq \exp \left( -\frac{C}{12} \right) \leq \exp \left( -\frac{Y \log(1/\epsilon^2)}{12} \right) = \epsilon^{\sqrt{C}} \]

by a Chernoff bound and assuming γ ≥ 24. Thus,

\[ \mathbb{E} \left[ \sum_{j \in [m]} C - Y_j^{(N)} \right] ≤ \epsilon^2 Cm ≤ \epsilon n/2, \]

where the last inequality used that ε is sufficiently small. This contradicts with high probability T ≥ N - O(√m/ε²).

As before, we consider a bin j \in [m] and define Pr[A_k] = Pr[X_j^{(N)} = k]. Then for k ≤ C,

\[ \frac{Pr[A_k]}{Pr[A_{k-1}]} = \frac{\binom{N}{k}}{\binom{N}{k-1}} \frac{1}{m-1} = \frac{N-k+1}{k} \frac{1}{m-1} \]

\[ ≤ \frac{\mu}{k} \frac{m}{k-1} = \frac{\mu}{k} (1 + O(1/m)). \]

It follows from the claim that µ/k ≥ 1 + 1/√C for k ≤ C. By our assumptions C ≤ 1/ε² = m^{o(1)} and thus Pr[A_k]/Pr[A_{k-1}] = (µ/k)^{1/ε²}. Let α \in \mathbb{N} be minimal such that Pr[A_{C-1}/Pr[A_{C-α}] ≥ 2. Using the crude bounds in the claim and simple calculations we obtain that α = Θ(1/Log(µ/C)). Now,

\[ Pr[Y_j^{(N)} < C] = O(α Pr[A_{C-1}]) = O(α Pr[AC]), \]

and

\[ \mathbb{E}[C - Y_j^{(N)}] = O(α² Pr[A_{C-1}]) = O(α² Pr[AC]). \]

As in case 1, Pr[Y_j^{(T)} < C] = Θ(Pr[Y_j^{(N)} < C]). Thus, if we can find the value of α, Eq. (4) will give us the result we are looking for. The problem is that α depends on N and hence of \mathbb{E}[T] which we as of now do not know the value of. However, we know that \mathbb{E}[C - Y_j^{(N)}] is close to en, so on a high level we can plug this into Eq. (5) and solve for α.

Let us make the above argument precise. First, we write µ = C + β noting that by the claim, √C ≤ β ≤ C/2. For note later use that

\[ α = O\left( \frac{1}{\log(µ/C)} \right) = O\left( \frac{C}{β} \right). \]

Using the Poisson approximation,

\[ Pr[AC] = (1 + o(1)) \frac{e^C}{C} e^{-µ} = \Theta\left( \left( \frac{β}{C} \right)^{C} \frac{1}{\sqrt{C} e^β} \right) = \Theta\left( \left( 1 + \frac{β}{C} \right)^{C} \frac{1}{\sqrt{C} e^β} \right). \]

Write f(x) = log (1 + x), so that \exp(f(β/C)) = 1 + β/C. As β/C ≤ 1/2, we can use a Taylor expansion to conclude that

\[ f \left( \frac{β}{C} \right) = f(0) + f’(0)\left( \frac{β}{C} \right) = \Theta(β/C^2). \]

Write Δ = β - C f(β/C), so that Δ = Θ(β²/C) = Θ(C/α²). Then

\[ Pr[AC] = Θ\left( \frac{1}{\sqrt{C} α} \right). \]

On the other hand, it follows from Corollary 3.2 that with high probability

\[ Pr[AC] = Θ\left( \frac{1}{√C} α \right). \]

so that, \mathbb{E}[C - Y_j^{(N)}] = Θ(εn). Plugging all this into Eq. (5), we find that

\[ \frac{α²}{√C} = Θ(εn). \]

Using that α² = Θ(C/Δ), this reduces to Δε^Δ = Θ(1/ε√C). So that

\[ Δ = Θ\left( \frac{1}{\sqrt{C}} \right) \]

and thus,

\[ α = Θ\left( \sqrt{C/\log(1/εn)} \right). \]

Combining Eq. (4) and Eq. (5), we find that

\[ Pr[Y_j^{(N)} < C] = Θ(Pr[Y_j^{(N)} < C]) = O(1), \]

A similar argument to that used in the first case shows that also

\[ Pr[Y_j^{(T)} < C] = Θ(Pr[Y_j^{(N)} < C]) = O(1), \]

which completes the proof.

\[ \square \]

**4 HIGH LEVEL IDEAS FOR THE PROOF OF THEOREM 1.3**

In this section, we describe (in broad terms) the main ideas needed to obtain the bound on the insertion time as stated in Theorem 1.3. Again, analysing the insertion time is the most difficult part, and in fact the remaining statements of the theorem follows by rather simple arguments (see the full version [2]). The proof proceeds in two steps as described in the next two sections.
4.1 Non-Full Bins: In Expectation and with Concentration

The first big step in the proof is understanding the fraction of non-full bins in the setting of Theorem 1.3, i.e., proving that this fraction is $\Omega(1/\ell)$ with high probability. In fact, we will require a slightly more general result which further shows that for each level $i$ and each capacity $j \leq C$, the number of bins which after level $i$ contains at most $j$ balls is concentrated around its mean. This is captured in the theorem below which is to be seen as an analogue to Theorem 1.1 in the more complicated setting of consistent hashing with bounded loads and virtual bins.

**Theorem 4.1.** Let $n, m \in \mathbb{N}$ and $0 < \epsilon < 1$. Suppose we insert $n$ balls into $m$ bins, each of capacity $C = (1 + \epsilon)n/m$, using consistent hashing with bounded loads and virtual bins and $k$ levels. For $(i, j) \in [k] \times [C + 1]$, let $X_{i,j}$ denote the number of bins with at most $j$ balls after the hashing of balls to levels $0, \ldots, i - 1$. For any $y = O(1)$ and $(i, j) \in [k] \times [C + 1]$, it holds that $|X_{i,j} - \mu_{i,j}| \leq m^{1/2+\epsilon y}$ with probability $1 - n^{-y}$. If moreover $k \geq c/\epsilon^2$ for a sufficiently large universal constant $c$, it holds that $\mu_{k-1,C-1} = \Omega(m^{1/2})$.

The proof of this theorem follows similar ideas to those used for Theorem 1.1, but the new setting adds several new challenges. Recall, that when we proved Theorem 1.1, we observed that the number of balls in a given bin essentially has the same distribution as $Z = \min(X, C)$, where $X \sim B(N, 1/m)$ is a binomial random variable and the parameter $N \geq n$ depends on the values of $C$ and $\epsilon$. The proof proceeded by analysing how fast the point probabilities $\Pr[Z = t]$ decayed when decreasing $t$ starting at $t = C$. With a minimal such that $\Pr[Z = C]/\Pr[Z = C - \alpha]$, we then got that

$$\mathbb{E}[C - Z] = \Theta(\alpha^2 \Pr[Z = C]) = \Theta(\alpha \Pr[Z < C]).$$

(6)

Since when placing $n$ balls into $m$ bins of capacity $C = (1 + \epsilon)n/m$, there are on average $\Theta(\epsilon C)$ unfilled places in a bin, we could conclude that $\mathbb{E}[C - Z] = \Theta(\epsilon C)$. Moreover, both $\Pr[Z = C]$ and $\alpha$ depends directly on $N$, so solving the first equation lead to the value of $N$ which in turn gave the value of $\alpha$. Then, we could finally estimate $\Pr[Z < C] = \Theta(\mathbb{E}[C - Z]/\alpha) = \Theta(\epsilon C/\alpha)$.

We approach the problem similarly when proving Theorem 4.1. However, the analogue variable which describes the number of balls landing in a bin, is more difficult to analyze. Our first important observation is that given the hashing of balls to previous levels, the number of balls landing in or being forwarded from a bin at a given level essentially follows a geometric distribution. The mean of this distribution depends on the hashing to the previous levels. After careful considerations, we then obtain an equation similar to Eq. (6), but where now $Z$ is the sum of geometric variables with some (yet to be determined means). The problem is that with the binomial distribution, $\Pr[Z = C]$ and $\alpha$ were relatively simple functions of $N$, but for a sum of geometric variables, they are much harder to analyze. Of particular interest is the bound on the point probability $\Pr[Z = C]$. For this, we apply the results of Amand et al. [1]. They provide sharp estimates for the point probabilities for sums of random integer variables, when the variables satisfy a property they call strong monotonicity relating to the characteristic functions of the variables. Fortunately, geometric variables are strongly monotone. Using the bounds of [1], we can similarly obtain bounds on $\alpha$ which is also defined using the point probabilities of $Z$. Now combined with a careful inductive proof, we obtain the asymptotic estimates of $\mathbb{E}[X_{i,c-1}]$ for all $i$. In particular, this gives the estimate $\mathbb{E}[X_{k-1,C-1}] = \Theta(m)$. We highlight that obtaining the high probability bounds constitutes the least difficult part of the proof. In fact, such bounds hold regardless of the number of levels and can also easily be proved in the setting of Theorem 1.1.

The main challenge is settling the value of the means $\mathbb{E}[X_{i,C-1}]$.

We remark that Bernoulli variables are strongly monotone, so that the bound in [1] can also be applied when some of the variables in the sum are Bernoulli. In the above, we assumed that the capacities of the bins were all equal to $C$, but combining with this observation, we obtain similar results when we use the dynamic capacities described in Section 1.7. For the dynamic capacities, we had a total capacity of $Cm$, with $m_1$ bins of capacity $[C]$ and $m_2$ bins of capacity $[C]$. As described in Section 1.7, this corresponds to all bins having capacity $[C]$, but including a $1$-th level, where $m_1$ bins receive a single ball. In particular, the number of balls landing in a random bin at the new lowest level is Bernoulli. Then the contribution to a random bin is essentially a sum of geometric variables and a single Bernoulli variable and since the bounds in [1] holds for such a sum, the proof carries through almost unchanged.

4.2 Analysing an Insertion

With Theorem 4.1, we can perform the analysis of an insertion for which we now provide the high level idea. First of all, it will be helpful to recall in details how an insertion of a ball is handled. When inserting a ball, we uniformly hash $x$ to a random point at a random level $i$. Starting at $h(x)$ we travel clockwise along level $i$ until we arrive at a virtual bin. If the virtual bin is filled to its capacity with balls hashing to level $1, \ldots, i$, we forward a ball from that bin at level $i$ (it could be $x$ but it could also be another ball that hashed to level $i$ of lower priority than $x$). We repeat the step, continuing to walk along level $i$ until we meet a new virtual bin. The first time we meet a virtual bin, $b$, which was not filled to its capacity with balls hashing to level $1, \ldots, i$, we insert the forwarded ball and find the smallest level $j > i$ such that the virtual bin of $b$ at level $j$ received a ball at level $j$. If no such level exists, the insertion is completed. Otherwise $b$ has an overflow of one ball at level $j$, and we continue the insertion walking along level $j$ starting at $b$.

The idea in the proof of Theorem 1.2 is splitting the the insertion into epochs. An epoch starts by visiting $[1/f]$ virtual bins of the insertion (unless of course the insertion is completed before that many bins have been seen). The last of these $[1/f]$ virtual bins lies at some level $i$ and we finish the epoch by completing the part of the insertion taking place at level $i$. At this point, we are either done with the insertion or we need to forward a ball from some virtual bin at some level $j > i$. The next epochs are similar; having finished epoch $a - 1$, in epoch $a$, we visit $[1/f]$ virtual bins. If we are not done with the insertion, we are at some level $\ell$ and we again finish the part of the insertion taking place at level $\ell$. Importantly, at the beginning of each epoch, we have just arrived at a virtual bin at a completely fresh level.

The proof shows that during the first $[1/f]$ steps of an epoch, the probability of finishing the insertion in each step is $\Omega(f)$. The
intuition for this, is that when we reach a bin \( b \) at some level, \( i \), the probability that \( b \) received more than \( C(1 - \varepsilon^2) \) balls from levels different than \( i \) is \( 1 - \Omega(1) \). This follows by at application of Theorem 4.1. Here, it should be noted that Theorem 4.1 states that the number of non-full bins is \( \Omega(fm) \) with high probability, but it is easy to provide a reduction showing that also the number of bins with at most \( C(1 - \varepsilon^2) \) balls is \( \Omega(fm) \) with the same high probability. Moreover, since the number of levels \( k = O(1/e^2) \) is large, it can be shown that the probability that level \( i \) contributes with at least \( C/e^2 \) balls to \( b \) is \( 1 - \Omega(1) \). These two events are not quite independent, but it is still possible to combine the bounds to show that the probability that \( b \) gets filled is \( 1 - \Omega(1) \).

Thus, the probability of not finishing the insertion during the first \( \lceil 1/f \rceil \) steps of an epoch is \( (1 - \Omega(f))^{1/f} = e^{-\Omega(1)} = 1 - \Omega(1) \). Now conditioning on not finishing the insertion during the first \( \lceil 1/f \rceil \) steps of an epoch, we can still show that the expected number of bins visited during the rest of the epoch is \( O(1/f) \). Here we importantly use that for finishing an epoch, we are just finishing the insertion at the current level and then we can apply an argument similar to the one used to prove Theorem 1.2 to bound the expected number of bins visited by \( O(1/f) \). Letting \( \mathcal{E} \) denote the event of finishing the insertion during the first \( \lceil 1/f \rceil \) steps of an epoch and \( T \), the total number of bins visited during the insertion, we have (at a very high level) that

\[
\mathbb{E}[T] \leq \Pr[\mathcal{E}] (1/f) + \Pr[\mathcal{E}^c] O(1/f) + \mathbb{E}[T] = O(1/f) + \mathbb{E}[\mathcal{E}] \mathbb{E}[T] = O(1/f) + p \mathbb{E}[T],
\]

where \( p = \Pr[\mathcal{E}^c] = 1 - \Omega(1) \). Solving this equation, we find that \( \mathbb{E}[T] = O(1/f) \). We refer the reader to the full version of the paper [2] for more details regarding this analysis.

ACKNOWLEDGMENTS

The authors wish to thank Noga Alon and Nick Wormald for helpful discussions. With Noga Alon, we studied sums of integer variables [1], including bounds needed for the analysis of this paper. In unpublished work on a the random graph \( d \)-process, Nick Wormald and Andrzej Rucinski also used the idea of analyzing balls in capacitated bins by throwing an appropriately larger number of balls into uncapacitated bins. They did not present an estimate on the number of non-full bins, as needed for this paper.

Research supported by grant 16582, Basic Algorithms Research Copenhagen (BARC), from the VILLUM Foundation.

REFERENCES