Round-Hashing for Data Storage: Distributed Servers and External-Memory Tables

Roberto Grossi
Dipartimento di Informatica, Università di Pisa, Italy
grossi@di.unipi.it

Luca Versari
Dipartimento di Informatica, Università di Pisa, Italy
luca.versari@di.unipi.it

Abstract

This paper proposes round-hashing, which is suitable for data storage on distributed servers and for implementing external-memory tables in which each lookup retrieves at most one single block of external memory, using a stash. For data storage, round-hashing is like consistent hashing as it avoids a full rehashing of the keys when new servers are added. Experiments show that the speed to serve requests is tenfold or more than the state of the art. In distributed data storage, this guarantees better throughput for serving requests and, moreover, greatly reduces decision times for which data should move to new servers as rescanning data is much faster.

2012 ACM Subject Classification Theory of computation → Sorting and searching, Information systems → Data dictionaries

Keywords and phrases consistent hashing, external memory, hash tables

Digital Object Identifier 10.4230/LIPIcs.ESA.2018.43

Acknowledgements Work supported by MIUR. We thank Rasmus Pagh for pointing us the reference on AC^0-RAM dictionaries, and the anonymous referees for their comments.

1 Introduction

We study the problem of consistent hashing for data storage, where the keys are web pages or data items to be dynamically mapped to a set of m buckets, uniquely identified by integers in $[0 \dots m-1]$. At any time, we want to support the following operations (including `init`() for the initialization) to increase or decrease the number of buckets (i.e. change mapping).

- `numBuckets`(): Return the current number m of buckets.
- `findBucket`(u): Given a key u, find its corresponding bucket identifier in $[0 \dots m-1]$.
- `newBucket`(): Add a new bucket having identifier m, thus making the range $[0 \dots m]$, and return the identifiers of the buckets whose keys should be redistributed.
- `freeBucket`(): Release the last bucket $m-1$, thus making range $[0 \dots m-2]$, and return the identifiers of the buckets whose keys should be redistributed.

Armed with the above operations, we can implement hashing by storing the keys in the buckets indicated by `findBucket`(), deciding when it is necessary to increase or decrease the number of buckets provided by `numBuckets`(): in the latter case, we have to redistribute the keys in the buckets indicated by `newBucket`() and `freeBucket`().

Table 1 Performance of the hashing methods for m buckets (servers). Here $s_0 \ll m$ is a constant slack parameter (typically $s = 64$ or 128), and $\alpha =$ (number of stored keys) $/\, m$ is the load factor. Although creating a new bucket moves $O(\alpha)$ keys on the average, each hashing method can take different time to decide which keys should be moved: "local" means that few other buckets scan their keys, while "distributed" means that all buckets scan their keys in parallel to decide which ones have to move to the new bucket. The $\tilde{O}()$ notation indicates an expected cost.

	find bucket	space	new bucket	notes
consistent hashing [13]	$O(\log m)$	$O(m)$	$\tilde{O}(\alpha + \log m)$	local
rendezvous hashing [24]	$O(\log m)$	$O(m)$	$O(m\alpha)$	distributed
jump consistent hash [14]	$\tilde{O}(\log m)$	$O(1)$	$O(m\alpha)$	distributed
linear hashing [15, 16]	$O(\log(m/s_0))$	$O(1)$	$\tilde{O}(s_0\alpha)$	local
round-hashing (ours)	$O(1)$	$O(1)$	$\tilde{O}(s_0\alpha)$	local, no division

History and motivation

Consistent hashing was invented by Karger et al. [13] for shared web caching, and highest random weight hashing (also known as rendezvous hashing) was invented by Thaler and Ravishankar [24] for web proxy servers. Both hashing methods were conceived independently around the mid 90s, and shared similar goals with different implementations: cached web pages are assigned to servers, so that when a server goes down, its cached web pages are reassigned to the other servers so as to preserve their load balancing; similarly, when a new server is added, some cached web pages are moved to it from the others. In contrast, a classical randomized load balancing scheme that uses hash mapping with modular operations on m is more expensive, as it requires to reallocate most of the keys when m changes.

Consistent hashing, in its basic version, maps both web pages and servers to the circular universe $[0 \dots 2^w - 1]$, where each hash value requires w bits: each web page starts from its hash value in the circular universe and is assigned to the server whose hash value is clockwise met first; this can be done in $O(\log m)$ time using a search data structure of size $O(m)$ for m servers. Rendezvous hashing, for a given web page p, applies hashing to the pairs $\langle p, i\rangle$ for each server i, and then assigns p to the server $i = i_0$ that gives the maximum hash value among these pairs; this is computed in $O(\log m)$ time using a tree of size $O(m)$ as discussed by Wang and Ravishankar [26]. The first two rows of Table 1 report a summary of these bounds. Both methods apply their rule above when a server is deleted or added. They have been successfully exploited in the industry, e.g. Akamai, Microsoft's CARP, Chord [23], and Amazon's Dynamo [8] to name a few.

Recently, Lamping and Veach presented jump consistent hashing [14] at Google, observing that it can be tailored for data centers and data storage applications in general. In this scenario, servers cannot disappear, as this would mean loss of valuable data; rather, they can be added to increase storage capacity.[1] As a result, the hash values "jump" to higher values for the keys moved to a new bucket; moreover, the hash values are a contiguous range $[0 \dots m - 1]$ for m servers, rather than a subset of m integers from $[0 \dots 2^w - 1]$. This has a dramatic impact on the performance of the jump consistent hash, as illustrated in [14], observing that only balance and monotonicity should be guaranteed from the original proposal in [13]. The auxiliary storage is just $O(1)$, as shown in the third row of Table 1; average query cost is the m-th harmonic number, so $O(\log m)$, with no worst case guarantee.

[1] Data is split into shards, where each shard is handled by a cluster of machines with replication, thus it is not acceptable for shards to disappear [14].

We observe that linear hashing, introduced by Litwin [16] and Larson [15] at the beginning of the 80s, can also be successfully employed in this scenario: as reported in the fourth row of Table 1, the resulting cost is $O(\log m)$ time with $O(1)$ space, where $s_0 \leq m$ is a user-selectable parameter that can be conveniently fixed to be $s_0 = O(1)$.

Looking at the first four rows in Table 1, when a new bucket is created, $O(\alpha)$ keys on average are moved from the other buckets, where α is the load factor, namely, the number of stored keys divided by the number m of buckets.[2] However, the hashing methods take different time to decide which keys should be move. Specifically, consistent hashing has to examine the $O(\alpha)$ keys in the two neighbor servers in the worst case, and update the data structure in $O(\log m)$ time.[3] Rendezvous hashing requires that each bucket scans its keys and test whether the new bucket is now the maximum for some of them. Hence all the keys are scanned, $O(m\alpha)$, but only $O(\alpha)$ of them are moved in total. Jump consistent hashing needs to perform a similar task, to see which keys "jump" to the new bucket. Linear hashing requires to scan the keys in $s_0 = O(1)$ buckets to find the $O(\alpha)$ ones to move.

Our hashing scheme

In the scenario of consistent hashing for data storage, we present a new mapping scheme, called *round-mapping*, to implement the operations `init`(), `numBuckets`(), `findBucket`(u), `newBucket`(), and `freeBucket`() mentioned before. Based on this, we obtain *round-hashing*, which computes the hash value of the given key and invokes round-mapping for this value achieving $O(1)$ time and space in the worst case, as shown in the last row of Table 1. This is a desirable feature, as otherwise hashing with no worst-case guarantee can pose security threats, such as algorithmic complexity attacks [3, 7] for low-bandwidth denial of service exploiting its worst-case behavior. Our scheme adds new buckets in a round-robin fashion by interleaving them with the existing buffers, so as to grow stepwise. For a constant slack parameter $s_0 \ll m$ (typically $s_0 = 64$ or 128), round-hashing can guarantee that the number of keys in the most populated bucket is at most $1 + 1/s_0$ times the number of keys in the least populated bucket.

Compared to the other schemes in Table 1, round-hashing is much simpler and faster due to the fixed arithmetic scheme of round-mapping that avoids division. This brings us in the realm of the cost of instructions on a commodity processor. To concretely illustrate our points, we refer to Intel processors [11]. Here Euclidean division is not our friend: integer division and modulo operations on 64-bit integers take 85–100 cycles, whereas addition takes 1 cycle (and can be easily pipelined). Interestingly, this goes in the direction of the so-called AC^0-RAM dictionaries (e.g. see Andersson et al. [2]) and Practical RAM (e.g. see Brodnik et al. [4] and Miltersen [18]), where integer division and multiplication are not permitted, among others. However, multiplication should be taken with a grain of salt as, surprisingly, it takes 3–4 cycles (which becomes 1 cycle when it can be pipelined). Also, the modulo operation for powers of two or for small constants proportional to s_0, can be replaced with a few shift and multiplication operations [10] as available, for instance, in the `gcc` compiler from version 2.6. Our implementation of round-hashing avoids general integer division and modulo operations because they are almost two orders of magnitude slower than the other operations: using them could nullify the advantage of the $O(1)$ time complexity. Furthermore, adding buckets is also fast and a straight-forward modification of the scheme.

[2] We are assuming, wlog, that the buckets have all the same size.
[3] For the sake of discussion, we consider the basic version of consistent hashing, and refer the reader to [13, 14] for the version with multiple hash values per server.

Distributed servers

Motivated by the application to distributed servers, we performed an experimental study of the above hashing methods, applying our tuning wherever possible. The code is publicly available at `https://github.com/veluca93/round_hashing` to replicate the experiments.

Our first observation addresses how balanced are the buckets filled with the hashing methods in Table 1. By uniformly sampling all the possible keys, their hash values can be used to estimate how far the number of keys in buckets are from the ideal load factor α, reporting the least and the most populated buckets after the experiments. We observed that jump consistent hashing is very close to α, ranging from $0.988\,\alpha$ to $1.012\,\alpha$; the experimental study in [14] shows that it compares favorably with consistent hashing (rendezvous hashing is not directly compared). We can match this performance by setting $s_0 = 128$ for linear hashing and $s_0 = 64$ for round-hashing.

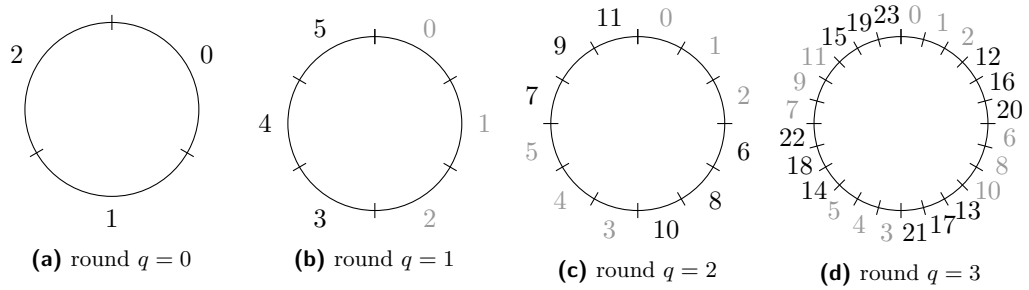
As a result of our tuning, to find the bucket number for a key, round-hashing is almost an order of magnitude faster than jump consistent hashing, and even much faster than the other hashing methods in Table 1. This is crucial for the system throughput: first, round-hashing can *serve* tenfold or more requests; second, when a new bucket number is added, it improves the performance of *rescanning* the keys to decide which ones move to the new bucket. We refer the reader to Section 3 for further details on our experimental study.

External-memory tables

It is interesting to apply round-hashing to high-throughput servers with many lookup requests, relatively few updates, and where some keys can be kept in a `stash` in main memory. We obtain a variant of dynamic hash tables, called *round-table*, and adopt the the EM model [1] to evaluate the complexity. Let n be the number of keys currently stored in the table, and B be the maximum number of keys that fit inside one block transfer, where a `stash` of k keys can be kept in main memory. We measure space occupancy using the *space utilization* $1 - \epsilon$, where $0 \leq \epsilon < 1$, defined as the ratio of the number n of keys divided by the number of external-memory blocks times B, hence the number of blocks is $\lceil \frac{n}{B(1-\epsilon)} \rceil$. In other words, ϵ represents the "waste" of space in external memory, so the lower ϵ, the better.

Round-table achieves the following bounds. Each lookup reads just 1 block from external memory in the worst case, taking $O(1)$ CPU time and thus requiring only $O(1)$ words from main memory. Each update (insertion or deletion) requires to access at most $4s_0$ blocks in external memory, in the worst case, taking $O(s_0(B + \log n / \log\log n))$ CPU time w.h.p. (expected time is $O(s_0 B)$) and using $O(B)$ memory cells. The number of keys in the `stash` is $k \approx n/\exp(B)$. Experiments in Section 4 confirm our estimation.

In the literature for external-memory hashing, Mirrokni et al. [19] provide a version that keeps bucket load within a factor of $1 + \epsilon$, but cannot guarantee at most one memory access. The optimal bounds in Jensen and Pagh [12] and Conway et al. [6] do not require the stash, with no guarantee of at most one memory access. As for the work on tables with one external-memory access, some results [17, 9] rely on perfect hashing, but are either not dynamic or cannot reach arbitrarily high utilization. A recent cuckoo hashing based approach [21], combined with in-memory Bloom filters to ensure that lookups access the correct position, is not simple to dynamize. A general scheme [22] relies on perfect hashing to store the `stash` on external memory, thus having higher worst-case cost for insertions. The result in [5] achieves single-access lookups, but at the cost of $O\left(\frac{n}{B(1-\epsilon)}\right)$ internal memory. A solution based on predecessor search needs $O\left(\frac{n}{B}\right)$ internal memory, as discussed in [20].



(a) round $q = 0$ **(b)** round $q = 1$ **(c)** round $q = 2$ **(d)** round $q = 3$

Figure 1 Example of round-mapping with $s_0 = 3$, where the sequences of bucket numbers are not actually materialized by our algorithm. Black colored bucket numbers represent those buckets that have been added during round q.

2 Round-Mapping and Round-Hashing

Conceptually we map the range of our hash function onto a circle of unitary circumference, starting from a fixed point 0. For a given integer $s_0 > 1$, the circumference is then split into arcs of length proportional to either $1/s$ or $1/(s+1)$ for some integer s ($s_0 \le s \le 2s_0 - 1$). We refer to arcs of length proportional to $1/s$ (resp. $1/(s+1)$) as *long* (resp. *short*) arcs. At any time *all the short arcs, if any, appear consecutively* along the circumference, starting from point 0 and proceeding clockwise (hence, also the long arcs appear consecutively).

Each arc has a corresponding *arc number*, which is simply its position along the circumference, and *bucket number*, which is assigned at the moment of the creation of the arc and maintained implicitly as the algorithm progresses. All elements whose hash value fall inside a given arc are assigned to the corresponding bucket.

At the beginning, we set $s = s_0$ and start with s_0 long arcs. We also keep a counter for the number of buckets, initially zero.

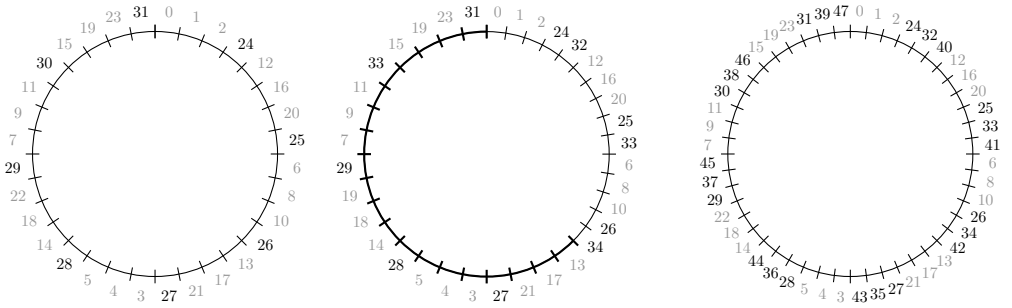
Operation `numBuckets`() simply returns the value of the above counter in $O(1)$ time.

Operation `findBucket`(u) is more involved, and is discussed in Section 2.1.

Operation `newBucket`() is implemented in $O(s_0)$ time by looking at long arcs, as follows. First, we check the border condition "all arcs are short": in that case, if $s < 2s_0 - 1$ then we set $s := s + 1$; else, we set $s := s_0$; either way, all the arcs become long. Second, we run *step* s to allocate a new bucket by taking the first s long arcs that are encountered clockwise along the circumference, and by replacing them with new $s + 1$ short arcs, say, $a_0, a_1, \ldots, a_s$. Their associated bucket numbers are mapped in the following way: the bucket numbers for $a_0, a_1, \ldots, a_{s-1}$ are inherited from the s long arcs that created them; the bucket number of a_s is the value of the counter, which is then increased by 1.

▶ **Lemma 1.** *At any time, the number of long arcs is always a multiple of s.*

Figures 1 and 2 show an example when a sequence of calls to `newBucket`() is performed (only some snapshots of the computation are presented). To understand the example, it helps to introduce a couple of concepts. We say that a *round* starts when the condition $s = s_0$ holds. Let the rounds be numbered as $q = 0, 1, 2, \ldots$, and let the length $len(q) = s_0\, 2^q$ of a round q represent the number of buckets at the end of its step $s = 2s_0 - 1$. For example, choosing $s_0 = 3$, the first rounds $q = 0, 1, 2, 3$ are shown in Figure 1. At step $s = 4$ of round $q = 4$, shown in Figure 2, each call to `newBucket`() takes s consecutive bucket numbers and inserts a new bucket number: after 0 1 2 24 it inserts 32, after 12 16 20 25 it inserts 33, after 6 8 10 26 it inserts 34, and so on. Note that 32, 33, 34, etc., are *native* of round $q = 4$ as



(a) end of step $s = s_0 = 3$ **(b)** between step $s = 3$ and 4 **(c)** end of step $s = 2s_0 - 1 = 5$

Figure 2 More detailed examples with $s_0 = 3$ during round $q = 4$. Long arcs are thicker.

Algorithm 1: Mapping from arcs to buckets.

Function `findBucket`(u)
  $j \leftarrow$ arc hit by u
  if $j < s_0$ **then return** j
  if $j > p$ **then** $j' \leftarrow j - \frac{p+1}{s+1}$, $s' = s$
  else $j' \leftarrow j$, $s' = s + 1$
  $x \leftarrow (j' \,\%\, s') \,\%\, s_0$
  $q' \leftarrow q + \left\lfloor \frac{s'-1}{s_0} \right\rfloor$
  $i = \left(1 + \left\lfloor \frac{s'-1}{s_0} \right\rfloor\right) \cdot \left\lfloor \frac{j'}{s'} \right\rfloor + \left\lfloor \frac{j' \% s'}{s_0} \right\rfloor$
  return $pos(i, x, q')$

Function $pos(i, x, q)$
  $e \leftarrow$ position of the least significant bit 1 in i
  return $\lfloor \frac{(s_0+x)2^q+i}{2^{e+1}} \rfloor$

they are created there. Black numbers in Figures 1 and 2 indicate which bucket numbers are native for the round. After step $s = 2s_0 - 1$, each round contains twice the bucket numbers than the previous round. Also, the concatenation of every other chunk of s_0 non-native bucket numbers, produces exactly the outcome of the previous round. We will exploit this regular pattern in the rest of the section.

Operation `freeBucket`() is simply the unrolling of the last `newBucket`() operation performed, hence, it also requires $O(s_0)$ time.

2.1 Implementation of `findBucket`(u)

We exploit the invariant property that short arcs are numbered from 0 to p, and thus $p+1$ is a multiple of $s+1$, where p is maintained as the last added short arc. We also use $pow(a)$, where $a > 0$, to denote the largest integer exponent $e \geq 0$ such that 2^e divides a (a.k.a. 2-adic order). Equivalently, $pow(a)$ is the position of the least significant bit 1 in the binary representation of the unsigned integer $a > 0$.

First, consider the ideal situation: after the step $s = 2s_0 - 1$ of round q, we have $len(q)$ buckets, numbered consecutively from 0 to $len(q) - 1$. We also have $len(q)$ arcs on the circle, numbered consecutively from 0 to $len(q) - 1$. As arc j is mapped to bucket number $b(j)$ using our scheme, we give a closed formula for $b(j)$ that can be computed in $O(1)$ time in the word RAM model, where divisions and modulo operations involve just powers of two or constants in the range $[s_0 \dots 2s_0]$ (see Algorithm 1).

Let $j = s_0\, i + x$ where $x \in \{0, 1, \ldots, s_0 - 1\}$. If $i = 0$, then $b(j) = b(x) = x$. Thus $b(j) = j$ for $0 \leq j < s_0$. Hence, let assume $i > 0$ in the rest of the section, and thus we need to compute $b(j)$ for $j \geq s_0$.

We say that the bucket number in position j belongs to *chunk* i (hence, a chunk is of length s_0). For odd values of i, the bucket number is native for round q. For even values of i, the bucket number is native for round $q - pow(i)$, as it can be checked in Figure 1: for example, in round q after the last step, bucket number 9 is in position $j = 37 = 3 \cdot 12 + 1$, so $i = 12$ and 9 is native for round $q - pow(i) = 4 - 2 = 2$. In general, as $pow(i) = 0$ when i is odd, we can always say that the bucket number is native for round $q - pow(i)$ for $i > 0$. Another useful observation is that the smallest native number in round q is $len(q - 1)$ by construction (e.g. 24 in round $q = 4$).

In the ideal situation, we find the native round for the bucket number at position j: as its chunk is preserved in the native round, we can use its offset x inside the chunk to recover the value of that bucket number. In the native round q, each chunk i starts with bucket number $len(q - 1)$ as previously observed, increased by one for each such chunk, thus the first bucket number in chunk i is $len(q - 1) + \lfloor i/2 \rfloor$. Also, any two adjacent numbers in the chunk, differ by 2^{q-1} by construction. Summing up, there are two cases for the bucket number for j:

- i odd and thus native for round q: the bucket number is $\left\lfloor \frac{(s_0+x)2^q+i}{2} \right\rfloor$
- i even and thus native for round $q - pow(i)$: the bucket number is $\left\lfloor \frac{(s_0+x)2^q+i}{2^{pow(i)+1}} \right\rfloor$

As $pow(a) = 0$ when a is odd, we can compactly write these positions in the ideal situation as

$$pos(i, x, q) = \left\lfloor \frac{(s_0 + x)2^q + i}{2^{pow(i)+1}} \right\rfloor$$

Second, consider the general situation, with an intermediate step $s_0 \leq s \leq 2s_0 - 1$ in round q. Recall that we know the position p of the last created arc. This gives the following picture. The first $p + 1$ short arcs in clockwise order can be seen as $\frac{p+1}{s+1}$ consecutive groups, each of $s + 1$ arcs, and the remaining arcs are long and form groups of s arcs each. Let us set $s' = s + 1$ in the former groups, and $s' = s$ in the latter groups. In the following, we equally say that each group contains s' arcs or that each group contains s' bucket numbers. In general, we say s' entries (arcs or bucket numbers) when it is clear from the context.

A common feature is that the first s_0 entries of each group are inherited from the previous round, and the last $s' - s_0$ entries in each group are those added in the current round: each new entry is *appended* at the end of each group, so the entry in position p is the last in its group.

Now, given a position j, we want to compute $b(j)$, the corresponding bucket number. The idea is to reduce this computation to the ideal situation analyzed before.

If $j > p$, we conceptually remove one entry for each group such that $s' = s + 1$. This is equivalent to set $j := j - \frac{p+1}{s+1}$ and, consequently, $p := p - \frac{p+1}{s+1}$. Now, we have all the groups of the same size s', which are sequentially numbered starting form 0.

Let $i' = \lfloor j/s' \rfloor$ be the number of the group that contains the entry corresponding to j. We now decide whether j is one of the first s_0 entries of its group or not. We have two cases, according to the value of $r = j \,\%\, s'$.

If $r < s_0$, the wanted entry is one of the first s_0 entries of its group. If we concatenate those entries over all groups, we obtain the ideal situation of the previous round $q - 1$. There, the wanted entry occupies position $j' = s_0 i' + r$. Hence, $b(j) = pos(i', r, q - 1)$ in the ideal situation.

If $r \geq s_0$, the wanted entry is one of the last s' entries of its group. Analogously, if we concatenate those entries over all groups, the position of the wanted entry becomes

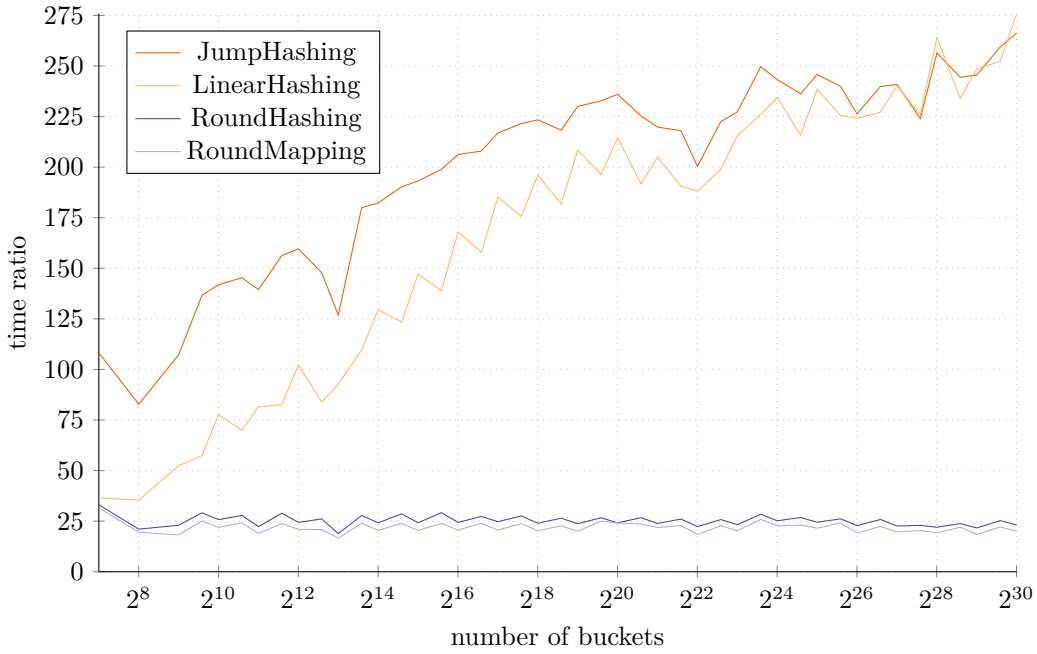


Figure 3 Time needed to compute a single hash as the number of buckets varies.

$j'' = (s' - s_0)i' + r - s_0$, where $x = r - s_0$ is the internal offset. However, we cannot solve this directly. We use instead the observation that the futures entries that will contribute to get the ideal situation for round q, will be *appended* at the end of each group. In this ideal situation, the wanted entry correspond to arc $2i'+1$ and is at position $j' = s_0(2i'+1)+r-s_0$ for round q. Thus, $b(j) = pos(2i'+1, r-s_0, q)$ in the ideal situation.

We can summarize the entire computation of $b(j)$ in an equivalent formula computed by Algorithm 1 that can be computed in $O(1)$ time.

▶ **Lemma 2.** `findBucket`() *can be implemented in* $O(1)$ *time using bitwise operations.*

Interestingly, `findBucket`() is much faster than other approaches known in the literature for consistent hashing, as we will see in the experiments.

▶ **Theorem 3.** *Round-mapping with integer parameter* $s_0 > 1$ *can be implemented using* $O(1)$ *words, so that* `init`(), `numBuckets`() *and* `findBucket`() *take* $O(1)$ *time, and* `newBucket`() *and* `freeBucket`() *take* $O(s_0)$ *time.*

Round-hashing computes the hash value of the given key and invokes round-mapping to obtain its bucket number. Whenever it decides to increase or decrease the number m of buckets, it invokes again round-mapping to know the $\Theta(s_0)$ buckets whose keys must be redistributed. Letting α be the load factor, namely, the overall number of stored keys divided m, we obtain the result in the last row of Table 1.

▶ **Theorem 4.** *Round-hashing with integer parameter* $s_0 > 1$ *requires* $O(1)$ *working space and takes* $O(1)$ *time to find the bucket for the key to be searched, inserted or deleted, and* $\tilde{O}(s_0\alpha)$ *average time to add or remove a bucket whenever needed.*

3 Distributed Servers

We experimentally evaluated round-hashing and our C implementation of Algortihm 1, on a commodity hardware based on Intel Xeon E3-1545M v5 CPU and 32Gb RAM, running Linux 4.14.34, and using `gcc` 7.3.1 compiler. We give some implementation details on the experimented algorithms, observing that we decided not to run consistent hashing [13] and rendezvous hashing [24] as they are outperformed by jump consistent hashing as discussed in detail in [14]. Specifically, we ran the following code.

- Jump consistent hashing [14]: we employed the implementation provided by the authors' optimized code.
- Linear hashing [15, 16]: the pseudocode is provided but not the code, which we wrote in C. As for the $O(\log m)$ hash functions, we followed the approach suggested in [15]: we employed the fast and high-quality pseudo-random number generator in [25] using the key to hash as a seed and the jth output as the outcome of the jth hash function. This takes constant time per hash function. Moreover, we replaced all modulo operations with the equivalent faster operations, as we did for round-mapping.
- Round-hashing (this paper): we employed the first output from the pseudo-random number generator in [25] as hash value. We chose the size of our hash range to be a power of two, so that mapping a hash value to an arc number can be done without divisions: we computed the product between the number of buckets and the hash value, divided by the maximum possible hash value. Note that some care is required to compute the product correctly as it may overflow.

It is worth noting that replacing the expensive division was very effective in our measurements. In particular, we replaced the division by s' in Algorithm 1 with the precomputed equivalent combination of multiplication and shift: as $s_0 \leq s' \leq 2s_0$, this can be done at initialization time with a constant amount of work. This reduced the time per round-hashing call from 14.02ns to 8.71ns, a 60% decrease, which is an interesting lesson that we learned.

Figure 3 shows the running times for the above implementations, when computing ten million hash values, as the number of buckets varies on the x-axis. On the y-axis, the running times are reported for jump consistent hashing, linear hashing, round-hashing and round-mapping alone (i.e. given a position u in the circumference, return its bucket number). As it can be seen, the costs of round-hashing and round-mapping are very close and *constant* along the x-axis, outperforming the non-constant costs of jump consistent hashing and linear hashing, which behave similarly when the number of buckets is large. Note that round-hashing has at least an order of magnitude improvement at around 2^{16} buckets and on, which indicates that it scales well.

All the running times in Figure 3 were normalized by the time needed to compute the sum of all the values. Looking at the absolute figures, the running time for the sum is about 0.4ns per element, and that of round mapping is 8–10ns per element (and the pseudo-random number generator in [25] takes twice the cost of the sum).

Speed is not the whole story as it is important also how the hash values in the range are distributed in the buckets. To this end, we show in Table 2 the results using 64-bit hash values: as it was infeasible to compute the bucket for every possible hash value, we chose 10^9 values at regular intervals in the hash range of 2^{64} values, and computed the bucket size distribution for them.

The columns in the table report the parameters for 10^4 buckets, where the actual bucket sizes are obtained by multiplying parameters in {min,max,1%,99%} by the load factor $\alpha = 10^9/10^4$. Specifically, s_0 useful for linear hashing and round-hashing, the standard

Table 2 Statistics on how much hash space is assigned to a given bucket, with a total of 10000 buckets. Note that the actual bucket sizes are obtained by multiplying the numbers in columns min, max, 1%, 99& by the load factor α. Extremal values and percentiles are a ratio from the ideal value.

	s_0	$\frac{\sigma}{\mu}$	min	max	1%	99%	percentile ratio
jump consistent h.		0.316	0.988	1.012	0.993	1.007	1.014
round-hashing	1	29.325	0.610	1.221	0.610	1.221	2.001
	2	20.272	0.814	1.221	0.814	1.221	1.500
	4	7.192	0.977	1.221	0.977	1.221	1.250
	8	4.465	0.976	1.085	0.976	1.085	1.112
	16	2.560	0.976	1.028	0.976	1.028	1.053
	32	0.613	0.976	1.002	0.976	1.002	1.027
	64	0.421	0.989	1.002	0.989	1.002	1.013
	128	0.277	0.995	1.002	0.995	1.002	1.007
linear hashing	1	29.329	0.602	1.232	0.605	1.228	2.030
	2	20.274	0.803	1.234	0.808	1.228	1.520
	4	7.203	0.964	1.232	0.969	1.225	1.264
	8	4.476	0.965	1.095	0.970	1.090	1.124
	16	2.583	0.965	1.041	0.970	1.034	1.066
	32	0.685	0.968	1.014	0.973	1.009	1.037
	64	0.527	0.980	1.014	0.984	1.009	1.025
	128	0.417	0.985	1.014	0.990	1.009	1.019

error $\frac{\sigma}{\mu}$ where σ is the variance and μ is the average of the bucket sizes, the minimum and maximum bucket size, the 1% and 99% percentiles of the size, and the ratio between the latter two. This ratio is the most important parameter in the table as it shows how well-balanced are buckets. It can be easily seen that both round-hashing and linear-hashing can match almost perfectly, with round-hashing having a slightly better distribution. Based on this table, we can see that round-hashing and linear-hashing have distribution properties that are similar to jump consistent hashing, as long as we choose suitable values: $s_0 = 64$ for round-hashing and $s_0 = 128$ for linear hashing. Figure 3 has been plotted using these values of s_0.

4 External-Memory Tables

Given a universe U of keys, and a random hashing function $h : U \to I$, where $I = \{0, 1, \ldots, |I| - 1\}$, we build a hash table that keeps a `stash` of keys in main memory. Armed with the round-hashing, we obtain a hash table called *round-table* that uses $O(k+1)$ words in main memory, where k denotes the number of stash keys. We consider the `stash` to be a set of k keys, where notation `stash`$[b]$ indicates the set $\{x \in$ `stash` : `findBucket`$(h(x)/|I|) = b\}$ (e.g. a hash table in main memory with maximum size $O(B + \log n/\log\log n)$ w.h.p. via a classical load balancing argument). To check if $x \in$ `stash`, we check if $x \in$ `stash`$[b]$ where $b =$ `findBucket`(x). Also, for a user given parameter ϵ, the guaranteed space utilization in external memory is $1 - \epsilon$.

The lookup algorithm is straightforward while the insertion algorithm is a bit more complex. After checking that the key is not in the table, it proceeds with the insertion. For this, we need to maintain the claimed space utilization of $(1 - \epsilon)$. That is, if $\lceil \frac{n}{B(1-\epsilon)} \rceil >$

Table 3 Percentage of elements on the `stash` as s_0 and ϵ change, with $B = 1024$.

s_0	ϵ											
	0		0.001		0.01		0.03		0.05		0.1	
	real	est.	real	est.	real	est.	real	est.	real	est.	real	est.
1	17.2%	18.4%	17.1%	18.3%	16.7%	17.4%	15.9%	15.7%	15.1%	14.5%	13%	12%
4	5.6%	6.8%	5.5%	6.7%	5.1%	5.9%	4.2%	4.4%	3.4%	3.4%	1.7%	1.7%
16	1.8%	2.8%	1.7%	2.7%	1.3%	1.9%	0.7%	0.7%	0.3%	0.1%	**0.01%**	**0.1%**
32	1.4%	2%	1.3%	1.9%	0.9%	1.2%	0.4%	0.3%	**0.1%**	**0.5%**	**0.003%**	**0.009%**
64	1.3%	1.6%	1.2%	1.5%	0.8%	0.9%	**0.3%**	**0.6%**	**0.1%**	**0.2%**	**0.003%**	**0.002%**
256	1.3%	1.3%	1.2%	1.3%	**0.8%**	**1%**	**0.3%**	**0.3%**	**0.08%**	**0.09%**	**0.003%**	**0.0005%**
ideal	-	1.2%	-	1.2%	-	0.8%	-	0.3%	-	0.07%	-	0.0003%

`numBuckets`(), we need one more block. We invoke `newBucket`(), and receive a list of $z < 2s_0$ block numbers. We have to distribute the keys stored in these z blocks over $z + 1$ blocks, where the extra block has number `numBuckets`() as it is the latest allocated block number by round-mapping. In the distribution, the keys from the `stash` are also involved, as described below in the function `distribute`. After that, `findBucket`() finds the external-memory block `block`() that should contain the key: if it is full, the key is added to the `stash`.

Function `distribute`$(b_0, b_1, \ldots, b_{z-1})$ takes these z block numbers from `newBucket`(), knowing that $b_z =$ `numBuckets`() is the new allocated block number, and thus allocates `block`(b_z). Then it loads `block`(b_{z-1}) and moves to `block`(b_z) all keys $x \in$ `block`(b_{z-1}) such that `findBucket`$(x) = b_z$. Also, for each $x \in$ `stash`$[b_{z-1}]$ such that `findBucket`$(x) = b_z$, it moves x to `block`(b_z), if there is room, or to `stash`$[b_z]$ otherwise. Next, we repeat this task for b_{z-2} and b_{z-1} while also taking care of moving keys from `stash`$[b_{z-1}]$ to `block`(b_{z-1}) if there is room, and so on. In this way, the cost of `distribute` is $2z + 1$ block transfers, using $O(B)$ space in main memory, taking $O(s_0(B + \log n/\log\log n))$ CPU time w.h.p., and $O(s_0 B)$ expected time.

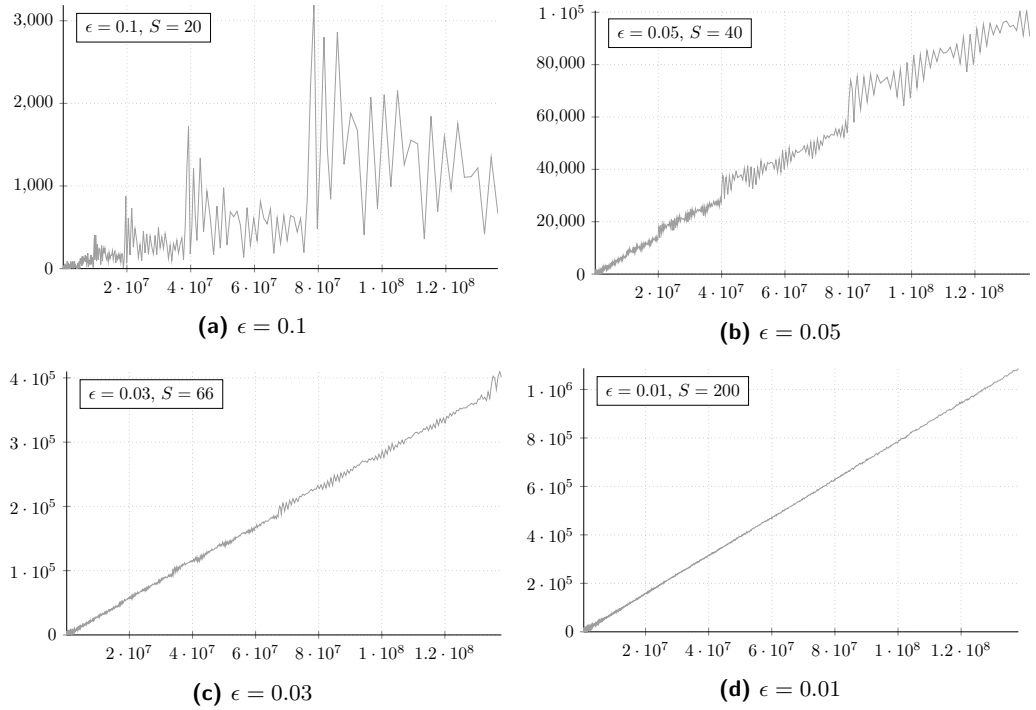
The deletion algorithm is similar to the insertion one, and its performance can be bound in the same way as above. We check the condition $\lceil \frac{n}{B(1-\epsilon)} \rceil <$ `numBuckets`$() - 1$ for $n > 0$ to run `freeBucket`() using a slightly different `distribute` that proceeds in reverse. Note that the rhs of the condition is `numBuckets`$() - 1$ to avoid `newBucket`() being called too soon.

We can show that as long as we choose $s_0 > \frac{2}{\epsilon}$ we have that the stash size of a hash table implemented with round-hashing is similar to the behaviour we would get with an uniform hash function (that would require rehashing). Thus, we recommend choosing $s_0\epsilon > 2$, as confirmed by the experiments below. Moreover, we can show how to keep a copy of the stash in external memory, without increasing space usage but increasing the number of block operations per update to $O(1 + \epsilon s_0)$.

To evaluate our approach, we consider the worst-case stash size (over the number of keys) across multiple values of n (going from $2^{10}B$ to $2^{13}B$) for $B = 512, 1024, 2048$ as ϵ and s_0 vary. The results are reported in Tables 3, where the left side of every column reports the ratio predicted by the analysis and the right side shows the effective maximum ratio reported during the experiment. As our analysis is substantially different when $\epsilon s_0 > 1$, we reported those values in bold to highlight them. Finally, the last row reports the best values one can hope to achieve for that value of ϵ, that is, the values that our analysis predicts for a uniform hash function.

Looking at these results, we can make some observations. First, the values predicted by the analysis match the results fairly well, especially when $s_0\epsilon \gg 1$ or $s_0\epsilon \ll 1$. In particular, it almost never happens that the analysis is wrong by more than a factor of 3. Second, when

Figure 4 Stash size (on the y-axis) as n grows (on the x-axis) for $s_0 = \frac{\epsilon}{2}$ and different values of ϵ.



(a) $\epsilon = 0.1$ **(b)** $\epsilon = 0.05$ **(c)** $\epsilon = 0.03$ **(d)** $\epsilon = 0.01$

s_0 is small, stash size is fairly high, even for low space utilization. This is to be expected, as in this case different buckets may have very different assignment probabilities. Third, as s_0 grows, stash size quickly approaches the one that we would expect from the ideal case. Nonetheless, the improvement is fairly small when s_0 goes over 32, even at low utilization. We thus recommend s_0 to be chosen near 32 for practical usage.

We also considered how stash size varies over time, as more elements are inserted. To study that, we fixed $s_0 = \frac{2}{\epsilon}$, as recommended in the analysis section, and plotted the size of the `stash` against the number of elements in the table. The plots can be found in Figure 4. These plots clearly show the "cyclic" behavior of round-table: when a new round begins, the distribution of keys in buckets is further away from being uniform and, as a result, the stash size increases. As more steps of the round are completed, the spikes in stash size get progressively smaller as round-table balances keys in a better way, until a new round starts again and the table reverts to its previous behavior.

5 Conclusions

We discussed a version of consistent hashing, called round-hashing, that scales well for large data sets in distributed servers. A key tool is round-mapping, and it would be interesting to see if it can have other applications, and if the number of changed buckets at each step can be reduced while keeping the same guarantees. As an example, we discussed how to obtain a dynamic hash table for external memory that guarantees at most one access to the external memory in the worst case, $(1 - \epsilon)$ space utilization, efficient updates, and small stash size in main memory.

References

1 Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, sep 1988.

2 A. Andersson, P. B. Miltersen, S. Riis, and M. Thorup. Static dictionaries on AC^0 RAMs: query time $\theta(\sqrt{\log n / \log\log n})$ is necessary and sufficient. In *Proceedings of 37th Conference on Foundations of Computer Science*, pages 441–450, Oct 1996. doi:10.1109/SFCS.1996.548503.

3 Noa Bar-Yosef and Avishai Wool. Remote algorithmic complexity attacks against randomized hash tables. In *SECRYPT 2007, Proceedings of the International Conference on Security and Cryptography, Barcelona, Spain, July 28-13, 2007, SECRYPT is part of ICETE - The International Joint Conference on e-Business and Telecommunications*, pages 117–124, 2007.

4 Andrej Brodnik, Peter Bro Miltersen, and J. Ian Munro. Trans-dichotomous algorithms without multiplication—some upper and lower bounds. In *Algorithms and Data Structures, 5th International Workshop, WADS '97, Halifax, Nova Scotia, Canada, August 6-8, 1997, Proceedings*, pages 426–439, 1997.

5 F. Cesarini and G. Soda. Single access hashing with overflow separators for dynamic files. *BIT Numerical Mathematics*, 33(1):15–28, Mar 1993. doi:10.1007/BF01990340.

6 Alexander Conway, Martin Farach-Colton, and Philip Shilane. Optimal hashing in external memory. In *Proc. Automata, Languages and Programming, 45th International Colloquium (ICALP18)*, 2018.

7 Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *USENIX Security Symposium*. USENIX Association, 2003.

8 Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220, 2007.

9 Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing - a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979. doi:10.1145/320083.320092.

10 T. Granlund and P. L. Montgomery. Division by invariant integers using multiplication. *ACM Sigplan Notices*, 29:61–72, 1994.

11 Intel Co. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel, order 248966-040, April 2018.

12 Morten Skaarup Jensen and Rasmus Pagh. Optimality in external memory hashing. *Algorithmica*, 52(3):403–411, 2008.

13 David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM. doi:10.1145/258533.258660.

14 John Lamping and Eric Veach. A fast, minimal memory, consistent hash algorithm. *CoRR*, abs/1406.2294, 2014. arXiv:1406.2294.

15 Per-Åke Larson. Linear hashing with partial expansions. In *VLDB*, volume 6, pages 224–232, 1980.

16 Witold Litwin. Linear hashing: a new tool for file and table addressing. In *VLDB*, volume 80, pages 1–3, 1980.

17 Harry G Mairson. The program complexity of searching a table. In *Foundations of Computer Science, 1983., 24th Annual Symposium on*, pages 40–47. IEEE, 1983.

18 Peter Bro Miltersen. Lower bounds for static dictionaries on RAMs with bit operations but no multiplication. In *Automata, Languages and Programming, 23rd International Colloquium, ICALP96, Paderborn, Germany, 8-12 July 1996, Proceedings*, pages 442–453, 1996.

19 Vahab Mirrokni, Mikkel Thorup, and Morteza Zadimoghaddam. Consistent hashing with bounded loads. *CoRR*, 2016. URL: `https://arxiv.org/pdf/1608.01350v1`.

20 Rasmus Pagh. Basic external memory data structures. *Algorithms for Memory Hierarchies*, pages 14–35, 2003.

21 Salvatore Pontarelli, Pedro Reviriego, and Michael Mitzenmacher. EMOMA: exact match in one memory access. *CoRR*, abs/1709.04711, 2017. `arXiv:1709.04711`.

22 M. V. Ramakrishna and Walid R. Tout. *Dynamic external hashing with guaranteed single access retrieval*, pages 187–201. Springer Berlin Heidelberg, Berlin, Heidelberg, 1989. `doi:10.1007/3-540-51295-0_127`.

23 Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

24 David Thaler and Chinya V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Trans. Netw*, 6(1):1–14, 1998.

25 Sebastiano Vigna. `xoroshiro128+`: an extremely fast and well-distributed pseudo random number generator. `http://xoroshiro.di.unimi.it/`, 2018.

26 Wei Wang and Chinya V. Ravishankar. Hash-based virtual hierarchies for scalable location service in mobile ad-hoc networks. *Mobile Networks and Applications*, 14(5):625–637, 2009.