4 Distributed data management I – Hashing

There are two major approaches for the management of data in distributed systems: hashing and caching. The *hashing* approach tries to minimize the use of communication hardware by distributing data randomly among the given processors, which helps to avoid hot spots, and the *caching* approach tries to minimize the use of communication hardware by keeping data as close to the requesting processors as possible. In this section, we will concentrate on the hashing approach.

4.1 Static hashing

The basic idea behind hashing is to use a compact function (the *hash function*) in order to map some space U onto some space V. Hashing can be used very effectively in the context of resource management, as we will see. Consider the problem of distributing data with addresses in some space $U = \{1, \ldots, M\}$ evenly among n storage units (also called *nodes* in the following). In the static case, we assume that n is fixed and the nodes are numbered from 1 to n. If the whole address space U is occupied by data items, it is easy to achieve an even distribution of the data among the nodes: node i gets all data j with $(j \bmod n) + 1 = i$. However, if U is sparsely populated, and in addition the data allocation in U changes over time, it is more difficult to keep the data evenly distributed among the nodes. In this case, (pseudo-)random hash functions can help.

Suppose that we have a random hash function that assigns every element in U to a node in $\{1, \ldots, n\}$ chosen uniformly at random, i.e., for every $x \in U$, every node is chosen with probability $1/n$. Then for any set $S \subseteq U$, the expected number of elements in S that are assigned to node i is $|S|/n$ for every $i \in \{1, \ldots, n\}$. In addition to this, the following result can be shown:

Theorem 4.1 *For any set $S \subseteq U$ of size m, the maximum number of elements in S placed in a single node when using a random hash function is at most*

$$\frac{m}{n} + O\left(\sqrt{(m/n)\log n} + \frac{\log n}{\log\log n}\right)$$

with high probability.

One can significantly lower the deviation from an optimal distribution of m/n data items per node by using a simple trick:

Suppose the data items arrive one by one and that instead of using a single random hash function, we use two independent random hash functions h_1 and h_2. For each data item x, we check the current number of data items in the nodes $h_1(x)$ and $h_2(x)$ and place x in the least loaded of them. (Ties are broken arbitrarily.) This rule is also called *minimum rule*. It has the following performance.

Theorem 4.2 ([5]) *For any set $S \subseteq U$ of size m, the maximum number of elements in S placed in a single node when using the minimum rule with two independent, random hash functions is at most*

$$\frac{m}{n} + O(\log\log n)\ .$$

Hence, random hashing techniques allow us to distribute any subset in U extremely evenly among the nodes. If data items can be moved after they have been placed, an even better distribution can be achieved when using the strategy in Figure 1.

```
pick two random nodes v_1 and v_2
if there is a data item placed in v_1 with alternative location in v_2 then
   pick any data item x that is placed in v_1 with alternative location v_2
   place x into the least loaded node (among v_1 and v_2)
   if there is a tie (i.e. v_1 without x has the same load as v_2) then
      place x into a randomly chosen of the two nodes
```

Figure 1: The self-balancing scheme.

Theorem 4.3 ([3]) *For any set $S \subseteq U$ of size m it holds that if the self-balancing scheme is run sufficiently long, then the maximum load of a node will eventually converge to at most $\lceil m/n \rceil + 1$, with high probability.*

Hence, two random hash functions can in principle distribute any subset of U almost perfectly among the nodes. Hash functions that have near-random qualities in practice are, for example, cryptographic hash functions such as SHA-1. So static hashing (i.e. the number of nodes is fixed) works fine. But what can we do if the number of nodes changes over the time? In this case we need *dynamic* hashing techniques.

4.2 Dynamic hashing

In a distributed system the set of available nodes may change over time. New nodes may join or old nodes may leave the system. Thus, a hashing strategy is needed that can adjust efficiently to a changing set of nodes. The naive approach to simply use a new random hash function each time the set of nodes changes is certainly not an efficient approach, because it would mean to replace virtually all the data. We will show that much better strategies exist for this, but first we have to specify the parameters we want to use to measure the quality of dynamic hashing approaches.

Let $\{1, \ldots, N\}$ be the set of all identification numbers a node may have and let $\{1, \ldots, M\}$ be the set of all possible addresses a data item can occupy. Suppose that the current number of data items in the system is $m \leq M$ and that the current number of nodes in the system is $n \leq N$. We will often assume for simplicity that the data and the nodes are numbered in a consecutive way starting with 1 (but any numbering that gives a unique number to each datum and node would work for our strategies). Let c_i be the current *capacity of node* i. Then $(c_1, \ldots, c_n)$ is the current *capacity distribution of the system*. We demand that $c_i \in [0, 1]$ for every i and that $\sum_{i=1}^{n} c_i = 1$. That is, each c_i represents the capacity of node i relative to the whole capacity of the system. (In reality, the capacity c_i of a node i may be based on its storage capacity, its bandwidth, its computational power, or a mixture of theses parameters.) Our goal is to make sure that every node i with capacity c_i has $c_i \cdot m$ of the data.

The system may change now in a way that the set of data items, the set of available nodes, or the capacities of the nodes change. In this case, a dynamic hashing scheme is needed that fulfills several criteria:

- **Fairness:** A scheme is called *fair* if the expected number of data items it places at node i is between $\lfloor (1-\epsilon)c_i \cdot m \rfloor$ and $\lceil (1+\epsilon)c_i \cdot m \rceil$ for all i, where $\epsilon > 0$ can be made arbitrarily small.

- **Time Efficiency:** A scheme is called *time-efficient* if it can compute the position of a data item in a short amount of time.

- **Compactness:** We call a scheme *compact* if the amount of information the scheme requires to compute the position of a data item is small (in particular, it should only depend on N and m in a logarithmic way).

- **Adaptivity:** We call a fair scheme *adaptive* if in the case that there is any change in the set of data items, nodes, or the capacities of the system, it can redistribute data items to get back to a fair distribution. To measure the adaptivity of a placement scheme, we use competitive analysis. For any sequence of operations σ that represent changes in the system, we intend to compare the number of (re-)placements of data performed by the given scheme with the number of (re-)placements of data performed by an optimal strategy that ensures that, after every operation, the data distribution among the nodes is perfectly fair (i.e. node i has exactly $c_i m$ data items, up to ± 1). A placement strategy will be called *c-competitive* if for any sequence of changes σ it requires the (re-)placement of (an expected number of) at most c times the number of data items an optimal adaptive and perfectly fair strategy would need.

To clarify the last definition, notice that when the capacity distribution in the system changes from $(c_1, ..., c_n)$ to $(c'_1, ..., c'_n)$, an optimal, perfectly fair strategy would need

$$\sum_{i:c_i > c'_i} (c_i - c'_i) \cdot m$$

replacements of data items. Thus, for example, if the capacity distribution changes from $(1/2, 1/2, 0)$ to $(0, 1/2, 1/2)$ (node 1 leaves and node 3 enters the system), ideally only a fraction of $1/2$ of the data would have to be moved.

4.3 Dynamic hashing for uniform systems

In this section we present two strategies that work well for nodes of uniform capacity (i.e., every node has the same capacity): the consistent hashing strategy and the cut-and-paste strategy. In the uniform case, the only changes in the system we have to consider are that a data item or node leaves or enters the system.

The consistent hashing strategy

The consistent hashing strategy was proposed by Karger et al. [4] and works as follows:

Suppose that we have a random hash function f and a set of independent, random hash functions $g_1, \ldots, g_k$, where k may depend on n. The function $f : \{1, \ldots, M\} \to [0, 1)$ maps the data uniformly at random to real numbers in the interval $[0, 1)$ and each function $g_i : \{1, \ldots, N\} \to [0, 1)$ maps the nodes uniformly at random to real numbers in the interval $[0, 1)$. Item i is assigned to the node j representing its closest successor in $[0, 1)$, i.e. item i is mapped to the node j with minimum $g_k(j)$ so that $f(i) \leq g_k(j)$, treating $[0, 1)$ here as a ring (see also Figure 2).

Theorem 4.4 ([4]) *Consistent hashing*

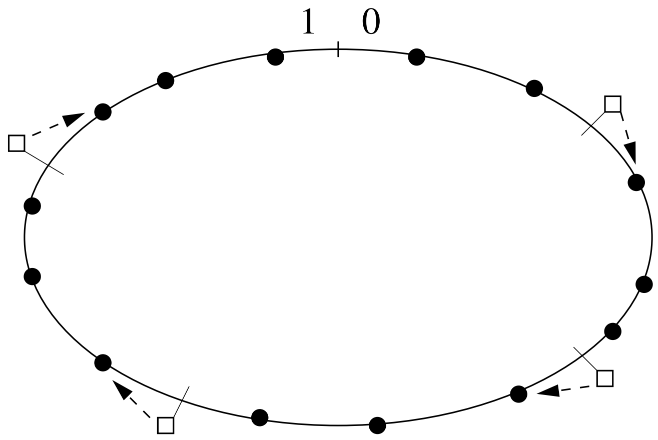

Figure 2: The consistent hashing strategy.

1. *is perfectly fair,*
2. *only requires a constant expected number of time steps (and $O(\log n)$ in the worst case) to compute the location of a data item when using a suitable algorithm,*
3. *needs $\Theta(n \log^2 N)$ memory (i.e., $k = \Theta(\log N)$) to make sure that the number of data items stored in a node deviates only by a constant factor from the ideal distribution with high probability, and*
4. *is 2-competitive concerning the amount of data that has to be moved if the set of nodes changes. Note that no movements (apart from insertions of new or deletions of old data) are required if the set of data changes.*

One might think that this strategy can be easily extended to cover the heterogeneous case by allowing nodes with more capacity to have more random points in $[0, 1)$. However, this would require $\Omega(\min[c_{\max}/c_{\min}, m])$ points to be fair, where $c_{\max}$ is the maximum capacity and $c_{\min}$ is the minimum capacity of a node. Thus, in the worst case the number of points a single node may have to use could be as much as $\Theta(m)$, violating severely our conditions on the space complexity. In fact, restricting the total number of points to something strictly below m cannot guarantee fairness in general (just consider two bins with capacities c/m and $(m-c)/m$ for some constant $c > 1$).

The cut-and-paste strategy

The cut-and-paste strategy consists of two stages and is based on a fixed, random hash function $f : \{1, \ldots, M\} \to [0, 1)$. Since f is a random function, it guarantees that for any subset of $[0, 1)$ of size s, the fraction of the data that is assigned to a number in this subset is equal to s. Thus, it only has to be ensured that the interval $[0, 1)$ is evenly distributed among the nodes, i.e. every node gets a part of $[0, 1)$ of size $1/n$ (see Figure 3).

For the mapping of the $[0, 1)$ range to the nodes, a so-called *cut-and-paste function* is used. This function will make sure that every node has the same share of the $[0, 1)$-interval. To simplify the description, given n nodes, we will denote the set of ranges assigned to node i by $[0, 1/n)_i$. In the

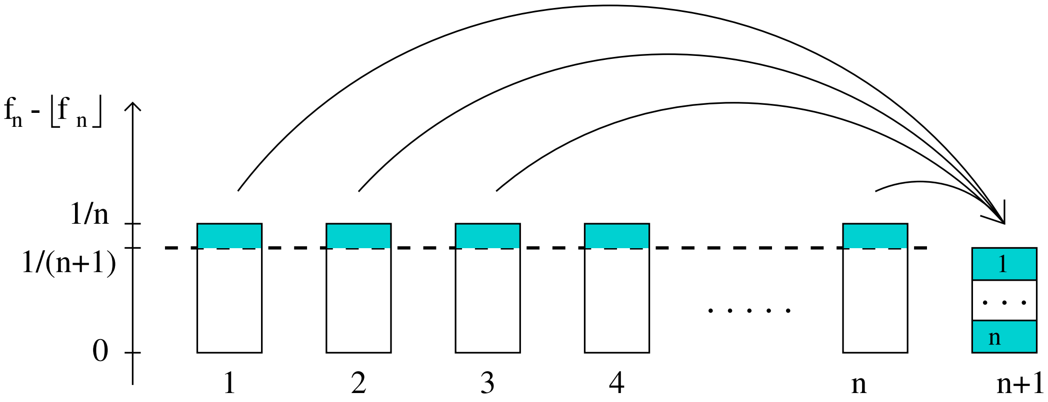

Figure 3: The cut-and-paste strategy.

case of a step-wise increase in the number of nodes from 1 to N, the cut-and-paste function works as follows:

At the beginning, we assign the whole range $[0, 1)$ to node 1. The *height* of a data item x in this case is defined as $f(x)$. For the change from n to $n+1$ processors, $n \in \{1, \ldots, N-1\}$, we cut off the range $[1/(n+1), 1/n)_i$ from every node i and concatenate these intervals to a range $[0, 1/(n+1))_{n+1}$ for the new node $n+1$. What this actually means for the movement of the data is described in Figure 4.

Replacement from n to $n+1$ nodes:
for every node $i \in \{1, \ldots, n\}$:
  for all data items x at i with current height $h \geq 1/(n+1)$:
    move x to node $n+1$
    the new height of x is $h - \frac{1}{n+1} + \frac{n-i}{n(n+1)}$

Figure 4: The replacement scheme for a new node.

If one node leaves, say node i, then we reverse the replacement scheme in a way that first node n moves all of its data back to the nodes 1 to $n-1$ and then takes over the role of node i (i.e., it obtains the identification number i and gets all data node i is required to have). This ensures the following result.

Theorem 4.5 ([1]) *The cut-and-paste strategy is perfectly fair and 2-competitive concerning the amount of data that have to be moved if the node set changes.*

Furthermore, the cut-and-paste strategy guarantees the following invariant:

Invariant: In *any* situation in which we have n nodes, the data items are distributed among them as this would be if we had a step-wise extension from 1 to n nodes.

In order to compute the actual position of a data item, it therefore suffices to replay the cut-and-paste scheme for a step-wise increase from 1 to n nodes. Fortunately, we do not have to go through all

n steps to compute the current position of a data item, but we only have to consider those steps that require the data item to be replaced. Using this strategy, we obtain the algorithm given in Figure 5 to compute the position of a data item.

Input: address $c \in \{1, \dots, M\}$
Output: node number $d \in \{1, \dots, n\}$
Algorithm:
set $d = 1$ and $x = f(c)$
while $x \geq 1/n$ do
    set $y = \lceil 1/x \rceil$
    set $x = x - 1/y + (y - 1 - d)/(y(y-1))$
    set $d = y$

Figure 5: The computation of the position of a data item.

It is not difficult to show that for any number d of a node in which a data item is currently stored, the second next node at which it will be stored is at least $2d$. Hence, the number of rounds needed for the computation of the position of a data item is $O(\log n)$. Thus, we obtain the following result.

Theorem 4.6 ([1]) *The cut-and-paste strategy*

- *can compute the location of a data item in $O(\log n)$ time steps and*
- *needs $O(n \log N)$ memory to store the numbering for the nodes.*

4.4 Dynamic hashing for non-uniform systems

Finally, we consider the case that we have an arbitrary capacity distribution. Also here we present two alternative strategies: SHARE and SIEVE. The results in this section are based on work in [2].

The SHARE strategy

SHARE uses as a subroutine the consistent hashing strategy presented earlier. It is based on two hash functions (in addition to the hash functions that are used by the consistent hashing strategy): a hash function $h : \{1, \dots, M\} \to [0, 1)$ that maps the data items uniformly at random to real numbers in the interval $[0, 1)$, and a hash function $g : \{1, \dots, N\} \to [0, 1)$ that maps starting points of intervals for the nodes uniformly at random to real numbers in $[0, 1)$. SHARE works in the following way:

Suppose that the capacities for the n given nodes are represented by $(c_1, \dots, c_n) \in [0, 1)^n$. Every node i is given an interval I_i of length $s \cdot c_i$, for some fixed *stretch factor* s, that reaches from $g(i)$ to $(g(i) + s \cdot c_i)$ mod 1, where $[0, 1)$ is viewed as a ring. If $s \cdot c_i \geq 1$, then this means that the interval is wrapped $\lceil s \cdot c_i \rceil$ times around $[0, 1)$. To simplify the presentation, we assume that each such interval consists of $\lceil s \cdot c_i \rceil$ intervals $I_{i'}$ with different numbers i' (that are identified with i), where $\lfloor s \cdot c_i \rfloor$ of them are of length 1.

For every $x \in [0, 1)$, let $C_x = \{i : \; x \in I_i\}$ and $c_x = |C_x|$, which is called the *contention* at point x. Since the total number of endpoints of all intervals I_i is at most $2(n + s)$, $[0, 1)$ has to be cut into at most $2(n+s)$ *frames* $F_j \subseteq [0,1)$ so that for each frame F_j, C_x is the same for every $x \in F_j$. This is important to ensure that the data structures for SHARE have a low space complexity. The computation of the position of a data item d is now simply done by calling the consistent hashing strategy with item d and node set $C_{h(d)}$ (see Figure 6).

Algorithm SHARE(b)**:**
Input: address d of a data item and a data structure containing all intervals I_i
Output: number of node that stores d
Phase 1: query data structure for point $h(d)$ to derive the node set $C_{h(d)}$
Phase 2: x = CONSISTENT-HASHING(d, $C_{h(d)}$)
return x

Figure 6: The SHARE algorithm.

For this strategy to work correctly, we require that every point $x \in [0,1)$ be covered by at least one interval I_i with high probability. This can be ensured if $s = \Theta(\log N)$. Under the assumption that the consistent hashing strategy uses k hash functions for the nodes, we arrive at the following result:

Theorem 4.7 ([2]) *If* $s = \Omega(\log N)$ *and* $k = \Omega(\log N)$, *the SHARE strategy*

- *is fair,*
- *only requires a constant expected number of time steps (and* $O(\log n)$ *in the worst case) to compute the location of a data item,*
- *needs* $\Theta(n \log^2 N)$ *memory to make sure that the number of data items stored in a node deviates only by a constant factor from the ideal distribution, with high probability, and*
- *is 2-competitive concerning the amount of data items that have to be moved if the node set changes. As in the previous strategies, no movements (apart from insertions of new or deletions of old data items) are required if the data set changes.*

A very important property of SHARE is that it is *oblivious*, i.e., the distribution of the data items among the nodes *only* depends on the current set of nodes and not on the history. This is not true, for example, for the cut-and-paste strategy. The drawback of SHARE is that the fraction of data items in a node is not highly concentrated around its capacity (unless s is very large) and that its space complexity depends on N and not just on n. The next, more complicated scheme will remove these drawbacks, but at the cost of being non-oblivious.

The SIEVE strategy

The SIEVE strategy is also based on random hash functions that assign to each data item a real number chosen independently and uniformly at random out of the range $[0,1)$. Suppose that initially the number of nodes is equal to n. Let $n' = 2^{\lceil \log n \rceil + 1}$. We cut $[0,1)$ into n' *ranges* of size $1/n'$, and we

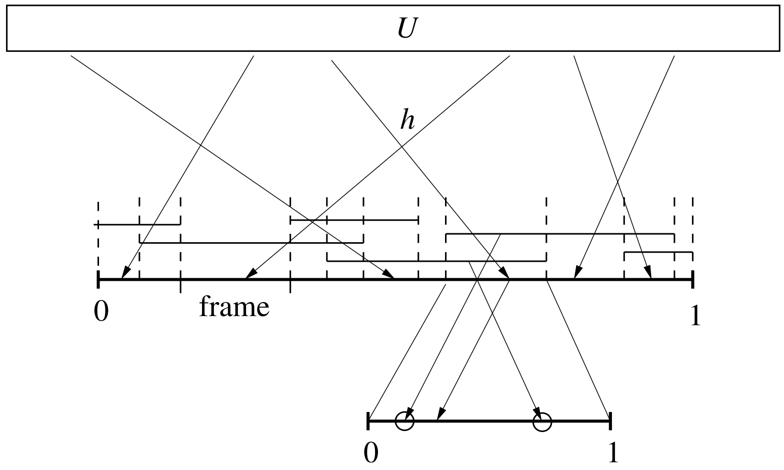

Figure 7: Illustration of the SHARE algorithm.

demand that every range be used by at most one node. If range I has been assigned to node i, then i is allowed to select any interval in I that starts at the lower end of I. The intervals will be used in a way (described in more detail below) that any data item mapped to a point in that interval will be assigned to the node owning it. We say that a range is *completely occupied* by a node if its interval covers the whole range. A node can own several ranges, but it is only allowed to own at most one range that is not completely occupied by it. Furthermore, we demand from every node i that the total amount of the $[0, 1)$ interval covered by its intervals is equal to $c_i/2$ (it will actually slightly deviate from that, but for now we assume it is $c_i/2$). These demands ensure the following property.

Lemma 4.8 *For any capacity distribution, it is possible to assign ranges to the nodes in a one-to-one fashion so that each node can select intervals in* $[0, 1)$ *of total size equal to* $c_i/2$.

Proof. Since every node is allowed to have only one partly occupied range, at most n of the n' ranges will be partly occupied. The remaining $\geq n$ ranges cover a range of at least $1/2$, which is sufficient to accommodate all ranges that are completely occupied by the nodes. □

So suppose we have an assignment of nodes to intervals in their ranges such that the lemma is fulfilled. Then we propose the strategy described in Figure 8 to distribute the data items among the nodes (the fall-back node will be specified later). It is based on L random hash functions $h_1, \ldots, h_L : \{1, \ldots, M\} \rightarrow [0, 1)$, where initially $L = \log n' + f$. The parameter f will be specified later. Figure 8 implies the following result:

Theorem 4.9 *SIEVE can be implemented so that the position of a data item can be determined in expected time* $O(1)$ *using a space of* $O(n \log n)$.

Proof. Since the nodes occupy exactly half of the interval $[0, 1)$, the probability that a data item succeeds to be placed in a round is $1/2$. Hence, the expected time to compute the position of a data item is $O(1)$. □

Let a data item that has not been assigned to a node in the for-loop of the algorithm above be called a *failed* data item. Obviously, the expected fraction of data items that fail is equal to $1/2^L$. Thus,

Algorithm SIEVE(d):
Input: number d of a data item
Output: node number that stores d
for $i = 1$ **to** L **do**
  set $x = h_i(d)$
  if x is in some interval of node s **then return** s
return number of fall-back node

Figure 8: The SIEVE algorithm.

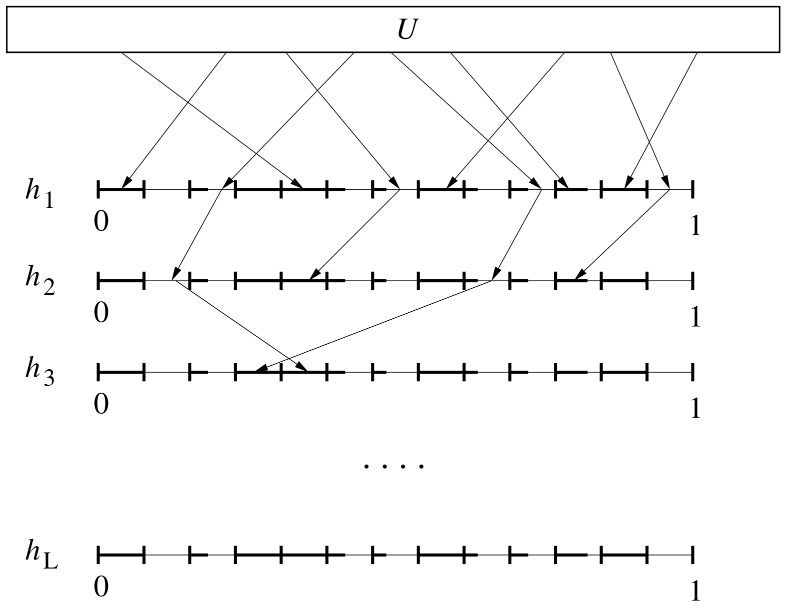

Figure 9: Illustration of the SIEVE algorithm.

the expected share of the data items any node i (apart from the fall-back node) will get is equal to $c_i(1-1/2^L)$. However, we want to ensure that every node gets an expected share of c_i. To ensure this, we first specify how to select the fall-back node.

Initially, the node with the largest share is the fall-back node. If it happens at some time step that the share of the largest node exceeds the share of the fall-back node by a factor of 2, then the role is passed on to that node.

Next we ensure that every node i gets an expected share of c_i. Let every non-fall-back node choose an *adjusted share* of $c_i' = c_i/(1-1/2^L)$, and the fall-back node chooses an adjusted share of $c_i' = (c_i - 1/2^L)/(1-1/2^L)$. First of all, the adjusted shares still represent a valid share distribution because

$$\sum_i c_i' = \frac{1-c_i}{1-1/2^L} + \frac{c_i - 1/2^L}{1-1/2^L} = 1 \; .$$

When using these adjusted shares for the selection of the intervals, now every non-fall-back node i gets a true share of $(c_i/(1-1/2^L)) \cdot (1-1/2^L) = c_i$ and the fall-back node gets a true share of $((c_i - 1/2^L)/(1-1/2^L)) \cdot (1-1/2^L) + 1/2^L = c_i$. Hence, the adjusted shares will ensure that the expected share of every node is precisely equal to its capacity. Thus, we arrive at the following conclusion.

Theorem 4.10 *SIEVE is perfectly fair.*

In addition, it can be shown (similar to the static hashing case) that node i gets at most $c_i m + O(\sqrt{c_i m \log m})$ data items with high probability.

In order to show that SIEVE also has a very good adaptivity, we have to consider the following cases:

1. the capacities change
2. the number n' of ranges has to increase to accommodate new nodes
3. the role of the fall-back node has to change
4. the number L of levels has to increase to ensure that $1/2^L$ is below the share of the fall-back node

As for the SHARE strategy, changes in the number of data items do not require SIEVE to replace data items in order to remain fair, since SIEVE is based on random hashing.

We begin with considering the situation that the capacities of the system change from $(c_1, c_2, \ldots)$ to $(d_1, d_2, \ldots)$. Then we use the following strategy: every node i with $d_i < c_i$ reduces its intervals in a way that afterwards it again partly occupies at most one range, and then every node i with $d_i > c_i$ extends its share so that it also partly occupies afterwards at most one range.

It is easy to check that there will always be ranges available for those nodes that increase their share so that every range is used by at most one node. It remains to bound the expected fraction of the data items that have to be replaced.

Lemma 4.11 *For any change from one capacity distribution to another that does not involve the change of the fall-back node, SIEVE has a competitive ratio of 2.*

Proof. For each non-fall-back node i, the expected fraction of the data items that needs to be replaced when changing from c_i to d_i is

$$\left(1 - \frac{1}{2^L}\right)|c_i' - d_i'| = \left(1 - \frac{1}{2^L}\right)\frac{|c_i - d_i|}{1 - 1/2^L} = |c_i - d_i|$$

and for the fall-back node i, $\left(1 - \frac{1}{2^L}\right)|c_i' - d_i'| \leq |c_i - d_i|$. Hence, the expected fraction of the data items that needs to be replaced is at most $\sum_i |c_i - d_i|$ which is at most twice the minimum number of data movements to get back to a fair placement. $\square$

Next we consider the situation that the number n' of ranges has to increase. This happens if a new node is introduced which requires $n' = 2^{\lceil \log n \rceil + 1}$ to grow. In this case, we simply subdivide each old range into two new ranges. Since afterwards the property is still kept that every node partly occupies at most one range, nothing has to be replaced.

Consider now the situation that the role of the fall-back node has to change. Recall that this happens if the node with the maximum share has at least twice the share of the fall-back node. Let i_1 be the old and i_2 be the new fall-back node. When i_1 became the fall-back node, we had $c_{i_1}^{old} \geq c_{i_2}^{old}$, and when i_2 becomes the fall-back node, it holds that $c_{i_1}^{new} \leq (1/2)c_{i_2}^{new}$. Since $c_{i_2}^{new} \geq 1/n$ (it is the node with maximum capacity), there must have been a capacity change of at least $(1/2)c_{i_2}^{new} \geq 1/(2n)$ in the system. When i_1 passes on the role of the fall-back node to i_2, then

$$\left|\frac{c_1 - 1/2^L}{1-1/2^L} - \frac{c_1}{1-1/2^L}\right| + \left|\frac{c_2}{1-1/2^L} - \frac{c_2 - 1/2^L}{1-1/2^L}\right| + \frac{1}{2^L}$$

of the data items have to be moved, which is at most $\frac{3}{2^L-1}$. If the f in the formula $L = \log n' + f$ is sufficiently large, then $\frac{3}{2^L-1} \ll \frac{1}{2n}$, and therefore the amount of work for the replacement can be "hidden" in the replacements necessary to react to changes in the capacity distribution of the system.

Next consider the situation that the number of levels L has to grow. Once in a while this is necessary, since for the case that many new nodes are introduced the fall-back node may not be able or willing to store a fraction of $1/2^L$ of the data items. We ensure that this will never happen with the following strategy:

Whenever the share of the fall-back node is less than $1/2^{L-t}$ for some integer t, we increase the number of levels from L to $L+1$.

This strategy will cause data items to be replaced. We will show, however, that also here the fraction of data items that have to be replaced can be "hidden" in the amount of data items that had to be replaced due to changes in the capacity distribution.

Let i_1 be the fall-back node that required an increase from $L-1$ to L (resp. the initial fall-back node if no such node exists), and let i_2 be the current fall-back node that requires now an increase from L to $L+1$. Then we know that the size of i_1 must have been at least $1/2^{(L-1)-t}$ when it became a fall-back node, and when i_2 must have dropped below $1/2^{L-t}$ so that there is a change from L to $L+1$. Due to the way the role of a fall-back node is passed on, this implies that there must have been a capacity change of at least $1/2^{(L-1)-t} - 1/2^{L-t} = 1/2^{L-t}$. Changing from L to $L+1$ involves the movement of an expected fraction of at most

$$\left(\sum_{j \neq i_2} \left|\frac{c_j}{1-1/2^L} - \frac{c_j}{1-1/2^{L+1}}\right|\right)$$
$$+ \left|\frac{c_{i_2} - 1/2^L}{1-1/2^L} - \frac{c_{i_2} - 1/2^{L+1}}{1-1/2^{L+1}}\right|$$

of the data items, which is at most $\frac{2}{2^L-3}$. If t and $f \geq t$ are sufficiently large, then $\frac{2}{2^L-3} \ll 1/2^{L-t}$, and therefore also here the amount of work for the replacement can be "hidden" in the replacements necessary to accommodate changes in the distribution of shares.

Hence, we arrive at the following result.

Theorem 4.12 *SIEVE is $(2+\epsilon)$-competitive, where $\epsilon > 0$ can be made arbitrarily small.*

References

[1] A. Brinkmann, K. Salzwedel, and C. Scheideler. Efficient, distributed data placement strategies for storage area networks. In *Proc. of the 12th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 119–128, 2000.

[2] A. Brinkmann, K. Salzwedel, and C. Scheideler. Compact, adaptive placement schemes for non-uniform capacities. In *Proc. of the 14th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 53–62, 2002.

[3] A. Czumaj, C. Riley, and C. Scheideler. Perfectly balanced allocation. In *7th Intl. Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)*, pages 240–251, 2004.

[4] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. of the 29th ACM Symp. on Theory of Computing (STOC)*, pages 654–663, 1997.

[5] A. S. P. Berenbrink, A. Czumaj and B. Vöcking. Balanced allocations: The heavily loaded case. In *Proc. of the 32nd ACM Symp. on Theory of Computing (STOC)*, pages 745–754, 2000.