8 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.

8.1 Static hashing

We start with a basic introduction to hashing. In general, a hash function may be any function from a space of size M to a space of size n. Hashing is mostly used in the context of resource management. Consider, for example, the problem of mapping an object space $U = \{1, \ldots, M\}$ to n units (often simply called *buckets* or *bins*). In the static case we assume that n is fixed and the units are numbered from 1 to n. If all of the object space U is used, it is easy to achieve an even distribution of the objects among the units: unit i gets all objects in the range $[(i-1) \cdot M/n + 1, i \cdot M/n]$. However, if U is sparsely populated, and in addition the object allocation in U changes over time, it is more difficult to keep the objects evenly distributed among the units. In this case, random hash functions can help.

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

Theorem 8.1 *For any set $S \subseteq U$ of size m, the maximum number of elements in S placed in a single unit 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 objects per unit by using a simple trick:

Suppose the objects 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 object x, we check the current number of objects in the units $h_1(x)$ 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 8.2 *For any set $S \subseteq U$ of size m, the maximum number of objects in S placed in a single unit 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 to distribute arbitrary sets of objects extremely evenly among the units. Hash functions that have near-random qualities in practice are, for example, secure hash functions such as SHA-1. So static hashing (i.e. the number of units is fixed) works fine. But what can we do if the number of units changes over the time? In this case we need *dynamic* hashing techniques.

8.2 Dynamic hashing

In a distributed system the number of available units may change over time. New units may join or old units may leave the system, which may be due to a switch-off or failure of the unit or the link it is connected to. Thus, a hashing strategy is needed that can adjust efficiently to a changing set of units. The naive approach to simply use a new random hash function each time the set of units changes is certainly not an efficient approach, because it would mean to replace virtually all objects. 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 possible units and $\{1, \ldots, M\}$ be the set of all possible objects that can be in the system at any time. Suppose that the current number of objects in the system is $m \leq M$ and that the current number of units in the system is $n \leq N$. We will often assume for simplicity that the objects and units are numbered in a consecutive way starting with 1 (but any numbering that gives unique numbers to each object and unit would work for our strategies). Let the current *capacity of unit* i be given by a parameter c_i and the current *capacity distribution of the system* be defined as $(c_1, \ldots, c_n)$. 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 unit i relative to the whole capacity of the system. (In reality, the capacity c_i of a unit i may be based on its storage capacity, its bandwidth, its computational power, or a mixture of theses parameters.) Our goal is to achieve that every unit i with capacity c_i obtains $c_i \cdot m$ of the objects.

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

- **Faithfulness:** A scheme is called *faithful* if the expected number of objects it places at unit 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 allows a fast computation of the position of an object.

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

- **Adaptivity:** We call a faithful scheme *adaptive* if in the case that there is any change in the number of objects, units, or the capacities of the system, it allows to redistribute objects to get back to a faithful 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 objects performed by the given scheme with the number of (re-)placements of objects performed by an optimal strategy that ensures that, after every operation, the distribution of objects among the units is perfectly faithful (i.e. bin i has exactly $c_i m$ balls, 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 objects an optimal adaptive and perfectly faithful 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 faithful strategy would need

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

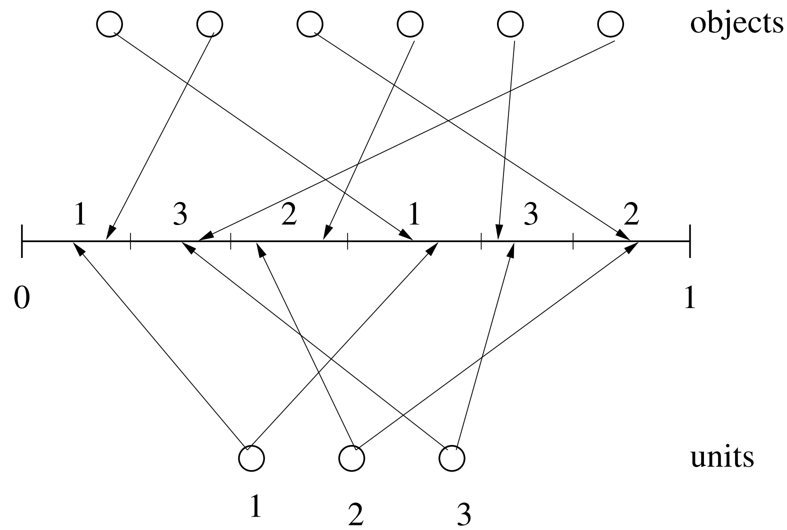

Figure 1: The nearest-neighbor strategy.

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

8.3 Dynamic hashing for uniform systems

In this section we present two strategies that work well for units of uniform capacity (i.e. every unit has the same capacity): the nearest-neighbor strategy and the cut-and-paste strategy. In this case, the only changes in the system we have to consider are that an object or unit leaves or enters the system.

The nearest-neighbor strategy

Suppose that every unit that can be in the system at any time has an identification number in the range $\{1, \ldots, N\}$. Then the nearest-neighbor strategy 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 units uniformly at random to real numbers in the interval $[0, 1)$. Item i is assigned to the unit closest to it with regard to this mapping when viewing $[0, 1)$ as a ring, i.e. it is assigned to the unit x that minimizes $\min_j \min[|f(i) - g_j(x)|, 1 - |f(i) - g_j(x)|]$ (see also Figure 1).

This strategy was suggested and analyzed in [3]. It has the following properties:

Theorem 8.3 ([3]) *The nearest-neighbor strategy*

1. *is perfectly faithful,*
2. *only requires a constant expected number of time steps (and $O(\log n)$ in the worst case) to compute the location of an object,*

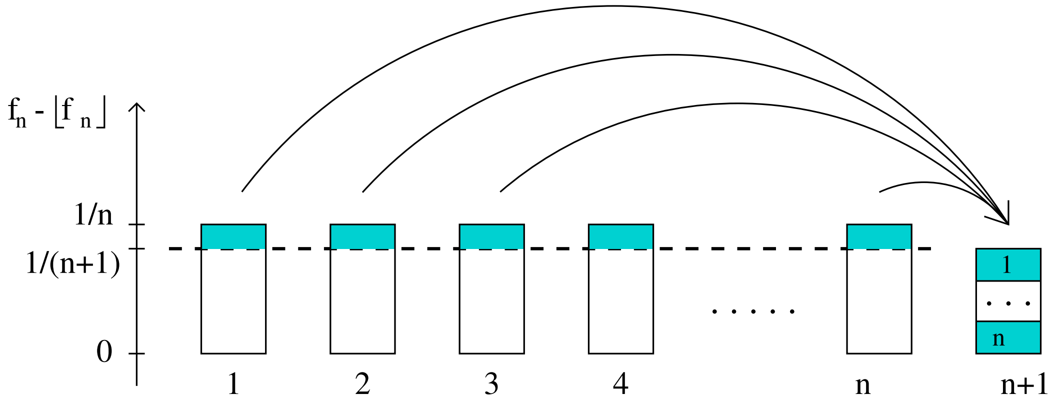

Figure 2: The object mapping strategy.

3. *needs* $\Theta(n \log^2 N)$ *memory (i.e.* $k = \Theta(\log N)$*) to make sure that the number of objects stored in a unit deviates only by a constant factor from the ideal distribution with high probability, and*

4. *is 2-competitive concerning the amount of objects that have to be moved if the set of units changes. Note that no movements (apart from insertions of new or deletions of old objects) are required if the set of objects changes.*

One might think that this strategy can be easily extended to cover the heterogeneous case by allowing units 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 faithful, where $c_{\max}$ is the maximum capacity and $c_{\min}$ is the minimum capacity of a bin. Thus, in the worst case the number of points 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 faithfulness 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 active objects that is assigned to a number in this subset is equal to s. Thus, it only has to be ensured that the range $[0, 1)$ is distributed among the storage devices in such a way that every storage device gets a part of this range of size $1/n$ (see Figure 2).

For the mapping of the $[0, 1)$ range to the units, a so-called *cut-and-paste function* is used. This function will take care that every unit has the same share of the $[0, 1]$ interval. To simplify the description, given n units, we will denote the set of ranges assigned to unit i simply by $[0, 1/n]_i$. In the case of a step-wise increase in the number of units from 1 to N, the cut-and-paste function works as follows:

At the beginning, we assign the whole range $[0, 1)$ to unit 1. The *height* of an object 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 unit i and concatenate these intervals to a range $[0, 1/(n + 1)]_{n+1}$ for the new unit $n + 1$. What this actually means for the movement of objects is described in Figure 3.

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

Figure 3: The replacement scheme for a new unit.

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

Theorem 8.4 ([1]) *The cut-and-paste strategy is perfectly faithful and 2-competitive concerning the amount of objects that have to move if the set of units changes.*

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

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

In order to compute the actual position of an object, it therefore suffices to replay the cut-and-paste scheme for a step-wise increase from 1 to n units. Fortunately, we do not have to go through all steps for an object, but only have to consider those steps that require the object to be replaced. In this case, we obtain the algorithm given in Figure 4 to compute the position of an object.

Input: object number $c \in \{1, \ldots, M\}$
Output: unit number $d \in \{1, \ldots, n\}$
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 4: The computation of the position of an object.

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

Theorem 8.5 ([1]) *The cut-and-paste strategy*

- *can compute the location of an object in $O(\log n)$ time steps and*
- *needs $O(n \log N)$ memory to store the numbering for the units.*

8.4 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 nearest neighbor strategy presented earlier. It is based on two hash functions (in addition to the hash functions that are used by the nearest neighbor strategy): a hash function $h : \{1, \ldots, M\} \to [0, 1)$ that maps the objects uniformly at random to real numbers in the interval $[0, 1)$, and a hash function $g : \{1, \ldots, N\} \to [0, 1)$ that maps starting points of intervals for the units uniformly at random to real numbers in $[0, 1]$. SHARE works in the following way:

Suppose that the capacities for the n given units are represented by $(c_1, \ldots, c_n) \in [0, 1)^n$. Every unit 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 $(g(i) + s \cdot c_i) \bmod 1$, where $[0, 1)$ is viewed as a ring. If $s \cdot c_i \geq 1$, then this means that the interval is wrapped around $\lceil s \cdot c_i \rceil$ 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 each $x \in F_j$. This is important to ensure that the data structures for SHARE have a low space complexity. The computation of the position of an object b is now simply done by calling the nearest-neighbor strategy with object b and unit set $C_{h(b)}$ (see Figure 5).

> **Algorithm** SHARE(b)**:**
> **Input:** number b of an object and a data structure
>   containing all intervals I_i
> **Output:** unit number that stores b
> **Phase 1:** query data structure for point $h(b)$
> to derive the unit set $C_{h(b)}$
> **Phase 2:** x = NEAREST-NEIGHBOR(b, $C_{h(b)}$)
> **return** x

Figure 5: The SHARE algorithm.

For this strategy to work correctly, we require that every point $x \in [0, 1)$ is 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 nearest-neighbor strategy uses k hash functions for the units, we arrive at the following result:

Theorem 8.6 *If $s = \Omega(\log N)$ and $k = \Omega(\log N)$, the SHARE strategy*

- *is faithful,*
- *only requires a constant expected number of time steps (and $O(\log n)$ in the worst case) to compute the location of an object,*

- *needs $\Theta(n \log^2 N)$ memory to make sure that the number of objects stored in a unit deviates only by a constant factor from the ideal distribution with high probability, and*
- *is 2-competitive concerning the amount of objects that have to be moved if the set of units changes. As in the previous strategies no movements (apart from insertions of new or deletions of old objects) are required if the set of objects changes.*

A very important property of SHARE is that it is *oblivious*, i.e. the distribution of the objects among the units *only* depends on the current set of units and objects 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 objects in a unit 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 object a real number chosen independently and uniformly at random out of the range $[0, 1)$. Suppose that initially the number of bins 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 demand that every range is used by at most one unit. If range I has been assigned to unit 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 object mapped to a point in that interval will be assigned to the unit owning it. We say that a range is *completely occupied* by a unit if its interval covers the whole range. A unit 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 unit i that the total amount of the $[0, 1)$ interval covered by its intervals is equal to $d_i/2$ (it will actually slightly deviate from that, but for now we assume it is $d_i/2$). This ensures the following property.

Lemma 8.7 *For any capacity distribution it is possible to assign ranges to the units in a one-to-one fashion so that each unit can select intervals in $[0, 1)$ of total size equal to $c_i/2$.*

Proof. Since every bin 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 bins. □

So suppose we have an assignment of unit to intervals in their ranges such that the lemma is fulfilled. Then we propose the strategy described in Figure 6 to distribute the objects among the unit (the fall-back unit will be specified later). It is based on L random hash functions $h_1, \ldots, h_L : \{1, \ldots, M\} \to [0, 1)$, where initially $L = \log n' + f$. The parameter f will be specified later. Figure 6 implies the following result:

Theorem 8.8 *SIEVE can be implemented so that the position of an object can be determined in expected time $O(1)$ using a space of $O(n \log n)$.*

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

Algorithm SIEVE(b)**:**
Input: number b of a ball
Output: bin number that stores b

for $i = 1$ **to** L **do**
  set $x = h_i(b)$
  if x is in some interval of bin s **then return** s
return number of fall-back bin

Figure 6: The SIEVE algorithm.

Let an object that has not been assigned to a unit in the for-loop of the algorithm above be called a *failed* object. Obviously, the expected fraction of objects that fail is equal to $1/2^L$. Thus, the expected share of the objects any unit i (apart from the fall-back unit) will get is equal to $c_i(1-1/2^L)$. However, we want to ensure that every unit gets an expected share of c_i. To ensure this, we first specify how to select the fall-back unit.

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

Next we ensure that every unit i gets an expected share of c_i. Let every non-fall-back unit choose an *adjusted share* of $c'_i = c_i/(1 - 1/2^L)$, and the fall-back unit 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

$$\sum c'_i = \frac{1 - c_i}{1 - 1/2^L} + \frac{c_i - 1/2^L}{1 - 1/2^L} = 1 \ .$$

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

Theorem 8.9 *SIEVE is perfectly faithful.*

In addition, it can be shown (similar to the static hashing case) that unit i gets at most $c_i m + O(\sqrt{c_i m \log m})$ objects 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 units
3. the role of the fall-back unit 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 unit

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

We begin with considering the situation that the capacities of the system change from $P = (p_1, p_2, \ldots)$ to $Q = (q_1, q_2, \ldots)$. Then we use the following strategy: every unit i with $q_i < p_i$ reduces its intervals in a way that afterwards it again partly occupies at most one range, and then every unit i with $q_i > p_i$ 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 units that increase their share so that every range is used by at most one unit. It remains to bound the expected fraction of the objects that have to be replaced.

Lemma 8.10 *For any change from one capacity distribution to another that does not involve the change of the fall-back unit, SIEVE has a competitive ratio of 2.*

Next we consider the situation that the number n' of ranges has to increase. This happens if a new unit 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 unit partly occupies at most one range, nothing has to be replaced.

Consider now the situation that the role of the fall-back unit has to change. Recall that this happens if the unit with the maximum share has at least twice the share of the fall-back unit. Let s_1 be the old and s_2 be the new fall-back unit. Suppose that the number of units in the system is n. Then s_2 has a share of at least $1/n$. At the time when s_1 was selected, the share of s_1 was at least as large as the share of s_2. Hence, the total amount of changes in the shares of s_1 and s_2 since then must have been at least $1/(2n)$. Changing from s_1 to s_2 involves the movement of an expected fraction of

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

of the objects, 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 units are introduced the fall-back unit may not be able or willing to store a fraction of $1/2^L$ of the objects. We ensure that this will never happen with the following strategy:

Whenever the share of the fall-back unit 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 objects to be replaced. We will show, however, that also here the fraction of objects that have to be replaced can be “hidden” in the amount of objects that had to be replaced due to changes in the capacity distribution.

Let s_j be the fall-back unit that required an increase from $L - 1$ to L (resp. the initial fall-back unit if no such unit exists), and let s_k be the current fall-back unit that requires now an increase from L to $L + 1$. Then we know that the size of s must have been at least $1/2^{L-(t+1)}$ when it became a fall-back unit. Suppose that s_k took over the role of a fall-back unit from s_j. Then its share must have been twice as large then the share of s_j. Since its share was at most the share of s_j when s_j got the role as fall-back unit, the total amount of changes in the shares of s_j and s_k since then must have been at least

$1/2^{L-t}$. This can also shown to be true for a longer history of fall-back units from s_j to s_k. Changing from L to $L+1$ involves the movement of an expected fraction of at most

$$\left(\sum_{i \neq k} \left| \frac{d_i}{1 - 1/2^L} - \frac{d_i}{1 - 1/2^{L+1}} \right|\right)$$

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

of the objects, 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 8.11 *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] 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.