Consistent, Order-Preserving Data Management in Distributed Storage Systems

Baruch Awerbuch
Department of Computer Science
Johns Hopkins University
3400 N. Charles Street
Baltimore, MD 21218, USA
baruch@cs.jhu.edu

Christian Scheideler
Department of Computer Science
Johns Hopkins University
3400 N. Charles Street
Baltimore, MD 21218, USA
scheideler@cs.jhu.edu

February 5, 2004

Abstract

In this paper we consider the problem of maintaining a mapping of data objects to memory modules so that the mapping is order preserving, i.e. objects closely together in the sorted set of current objects are also closely together in the mapping. Keeping close objects closely together is important for many applications such as efficiently executing programs using a large amount of space or complex search queries such as semi-group range queries. Our main result assumes a static set of memory modules of uniform capacity, but we also show how to extend this to a dynamic set of memory modules of non-uniform capacity in a decentralized environment.

We assume that insert and delete requests of objects are continuously injected into the system, and the injection is under adversarial control. Data objects are of uniform size and every module can send or receive at most one object in each unit of time. We prove asymptotically tight upper and lower bounds for the maximum rate at which the adversary can inject update requests into the modules so that an ordered placement can be preserved without exceeding the capacity of a module at any time. Specifically, we show that in a $(1-\epsilon)$-utilized system (i.e. the available space is used up to an ϵ fraction) the maximum injection rate that can be sustained is $\Theta(\epsilon)$. While this does not seem to be particularly surprising, it is actually hard to prove in a rigorous way.

Besides having interesting consequences for distributed storage and information systems, we also feel that our approach offers interesting future problems for theoretical research.

1 Introduction

1.1 Motivation

The concept of *virtualization* or *indirection* is a basic concept in computer science. Perhaps one of its most basic manifestations is the invention of *virtual memory*. That is, program variables (array entries, file blocks) are referred to by their *virtual name*, rather then by their physical address. Consider a program that performs a loop over an array. From the point of view of *correctness* of this program, it does not matter where the array elements are stored. However, from the point of view of *performance*, it is desirable to map entries of an array into the same page in the memory, or at least to try to minimize the number of pages that store the array's entries. Thus, for certain applications it is important to consider locality when mapping data to pages.

Locality also plays an important role in information retrieval. Imagine a geographic range or a time range, and values being assigned to certain points in this range, e.g. the temperature, stock quotes, etc. Consider the problem of performing, upon request, range operations like computing the average temperature in a geographic region or determining the maximum weekly fluctuation of a stock. This is known as *semi-group range queries*. Semi-group range queries are useful for web search engines, geographic information systems, inventory control, and consistency checks of file systems. To achieve a high efficiency, it is important to keep elements that are close to each other in the object space also close to each other when mapping them to pages, i.e. to store them in a de-fragmented way. The reason for this is that the work for locally processing semi-group range queries usually scales logarithmic with the number of objects (e.g., [5, 1, 37, 38]), whereas the work for aggregating the results from different pages scales linearly with the number of pages involved in a range query. Thus, uniform hashing, i.e. globally scattering data objects over the pages, makes range queries *very expensive*. This is particularly painful in large, distributed information systems, where the data is distributed over multiple sites, because apart from the communication overhead, hashing may require a range query to be processed at a large number of sites instead of just a single site.

Since hashing has many benefits in a distributed setting such as evenly distributing data and simple lookup requests among sites, one is tempted to use locality-preserving hash functions as a compromise. Such a function maps objects from some underlying universe of object names (e.g., file blocks, array entries, etc.) into pages of memory such that a consecutive range of objects is stored in a small number of pages. The question is, *can such hashing possibly work*? The paper by Linial and Sasson [29] seems to indicate that this is possible. (See also follow-up work in [28, 22, 21]). However, it should be clear that no static mapping can preserve locality for a dynamically changing set of data objects without causing the load at some site to exceed its capacity. Hence, dynamic mapping schemes have to be used. Whereas dynamic hashing schemes that do not preserve locality are known [25, 6, 7], it is questionable whether dynamic locality-preserving hashing can be done and actually makes sense (since the benefit of hashing over deterministic strategies is unclear in this case).

Instead of using hashing, this paper accomplishes the goal of de-fragmented data storage using a *deterministic, adaptive* memory assignment strategy that keeps the data objects in order and that is efficiently manageable under adversarial inputs in concurrent environments.

1.2 Our approach: models and results

Order-preserving data management. For our algorithmic approach, we focus on a natural model of storage, where each memory module (or node) is responsible for a consecutive range of object names. This will also be called *order-preserving* data management. In formal terms, consider a universe P^* of all objects with a linear order defined on these objects so that w.l.o.g. we can assume that objects are represented by real numbers in $[0, 1)$. Consider also a set of memory modules or nodes V with naming

function $\mathsf{Name} : V \to P^*$ so that each node $v \in V$ has a name $\mathsf{Name}(v)$ in $P^* = [0, 1)$. Given an object o, we expect to find this object in memory module v whose name is the closest predecessor of $\mathsf{Name}(o)$, i.e.

$$v = \mathsf{argmin}\{\mathsf{Name}(v) \mid v \in V \text{ and } \mathsf{Name}(v) \leq \mathsf{Name}(o)\} \ . \qquad (*)$$

Similar to [25] and follow-up work in the area of peer-to-peer systems [36, 33, 39, 11], we say that a set of objects is stored *consistently* among a set of nodes if each object is stored at the node specified in $(*)$. However, our approach significantly differs from these approaches:

- *object names:* in [25, 36, 33, 39, 11], the real numbers describing the names of the objects are *static hash values* of their true names. In the current paper, the real numbers represent the *original* names in order to avoid fragmentation and enable (one-dimensional) range queries.
- *node names:* in [25, 36, 33, 39, 11], the real numbers describing the names of the nodes are *static hash values* of their IP addresses or *random values*. In the current paper, the real numbers of the nodes are dynamically changed to adapt the mapping to a changing set of currently used objects.

Operational and transient consistency. To adapt to a changing set of objects, we only allow two operations: *renaming* of nodes, and *migrating* objects. Clearly, the object assignment must return to a consistent state after any insertion and deletion of an object because otherwise objects cannot be found efficiently any more. This property is called *operational consistency*. In the transient state, i.e., during the execution of insert/delete operations, the storage system may be inconsistent. A stronger (more restrictive) model of *transient consistency* requires consistency even in the transient state. That is, after every elementary operation (such as renaming a node or migrating an object), the system must be back in a consistent state. Transient consistency is important to make sure that the system can process read requests at any time and recover easily from a crash.

Concurrency. Information systems based on a large number of processing units (such as storage area networks and peer-to-peer systems) are becoming increasingly important. Hence, our focus will be on a multiple processing unit environment, i.e. each node represents its own processing unit.

Generation of update requests. We are interested in the *dynamic* setting where continuously new objects are inserted and old objects are deleted. We assume that the generation of insert/delete requests is under adversarial control. A λ-bounded adversary is allowed to generate an arbitrary set of object updates in each time step as long as the average number of object updates that have to be handled by a node is λ or less. If this is always true when averaging over time windows of size T, we call such an adversary a (λ, T)-bounded adversary. Given such an adversary, an algorithm is called *stable* if it can preserve an order-preserving mapping without exceeding the capacity of a module at any time. Since we assume that every module can receive or send at most one object per time unit, λ can be at most 1 for an algorithm to be stable.

Memory utilization vs. update rate. The memory utilization of a system denotes the degree to which it is filled, i.e. for $\gamma \in [0, 1]$, a *γ-utilized* system needs a γ fraction of its resources to store the data objects currently in the system.

We assume that we have a static set of memory modules of uniform capacity, and we are interested in investigating the maximum rate of update requests that can be sustained given a certain memory utilization to keep the assignment of data to memory modules consistent and order-preserving without exceeding the capacity of a module. The main contributions of this paper are matching upper and lower bounds on the rate of insert and delete requests that can be sustained for a given memory utilization. Specifically, in Section 2 we show

Theorem 1.1 *There is a $\Theta(\epsilon)$-bounded adversary so that any operationally consistent, order-preserving online algorithm is unstable in a $(1-\epsilon)$-utilized system.*

For our model, that demands memory consistency even in transient state, we show:

Theorem 1.2 *For any $0 < \epsilon < 1$, the online algorithm in Figure 1 is maintaining an order-preserving mapping under any $\alpha \cdot \epsilon$-bounded adversary (for some constant $\alpha > 0$) as long as the system is at most $(1-\epsilon)$-utilized at any point in time.*

Thus, our upper and lower bounds for stability are essentially tight.

Fragmentation. Our online algorithm achieves actually more than just being stable. For a data management algorithm to be work-competitive for range queries, it has to be ensured that a set of consecutive objects is not distributed among too many nodes. Given nodes with capacity C, we define the placement of objects to nodes to be *$(1+\alpha)$-fragmented* if for any set of s consecutive objects currently in the system, the number of nodes storing the objects is at most $\lceil (1+\alpha) \cdot \max[s/C, 1] \rceil$. Our algorithm maintains a $(1+O(\epsilon))$-fragmented mapping in an at most $(1-\epsilon)$-utilized system and is therefore $(1+O(\epsilon))$-competitive concerning the number of nodes to be contacted for range queries.

Dynamic servers or dynamic capacity model. We also consider the case that memory modules continuously join and leave the system and that the memory modules have arbitrary, non-uniform capacities. In fact, by viewing the object movements necessary to cope with a dynamic set of memory modules or capacity changes as object injections, we can reduce these cases to the case of insertions and deletions of objects in a static system, allowing us to carry over our stability result for a static system to dynamic, non-uniform systems. Specifically, we prove (see Sections 3.2 and 3.1):

Theorem 1.3 *The online algorithm in Figure 1 is maintaining an order-preserving mapping under any $\alpha \cdot \epsilon$-bounded adversary (for some constant $\alpha > 0$, considering both object updates and capacity changes) as long as the system is at most $(1-\epsilon)$-utilized at any point in time.*

Certainly, considering $1+\epsilon$-fragmentation does not make sense in a heterogeneous environment because the number of nodes storing a range of objects depends on their capacities.

Decentralized storage systems. Finally, we will address the issue of how to turn our online algorithm into a decentralized storage system. We will distinguish between "busy" nodes, i.e. nodes storing information, and "idle" nodes, i.e. empty nodes that will be taken whenever nodes are needed to help out busy nodes. The busy nodes and the idle nodes are organized in suitable overlay networks, with links from busy nodes to random idle nodes so that nodes can be transferred between the busy and the idle structures as necessary (see Section 4 for details).

1.3 Previous work

We first discuss prior work in the centralized setting, i.e. there is a central dispatcher that inserts and deletes data at the memory modules and moves data between the memory modules to prevent the object load at a module from exceeding its capacity. For this model, order-preserving search structures have been presented in the context of cache-oblivious B-trees and monotonic list labeling. Bender et al. [4] present a cache-oblivious B-tree that, in our context, needs an amortized work of $O(\log^2 n)$ per insert and delete operation to keep objects distributed among the memory modules in an order-preserving way without exceeding the

capacity of a memory module. This bound holds as long as the system is at most $1/2$-utilized at any time. The same result has also been shown independently by Itai et al. [23]. Brodal et al. [8] generalized the bound to $O((\log^2 n)/\epsilon)$ as long as the system is at most $(1-\epsilon)$-utilized at any time.

Also lower bounds have been investigated. Dietz and Zhang [13] showed that for the case that the order of the (names of the) memory modules cannot be changed, any smooth, order-preserving data management algorithm has an amortized cost of $\Omega(\log^2 n)$ per insertion. In words, an algorithm is called *smooth* if the items moved before each insertion form a contiguous neighborhood of the position for the new item, and the new labels are as widely and equally spaced as possible. The algorithms in [4, 8, 23] fulfill these properties, and their upper bound is therefore best possible. Later, Dietz et al. [12] showed that any online order-preserving placement strategy that preserves the order of the memory modules needs an amortized work of $\Omega(\log n)$. Our algorithm can break through this lower bound because it allows the order of the memory modules to be *changed*, i.e. memory modules can be renamed to help out overloaded memory modules in another area.

Also concurrent forms of search trees have been reported [26, 31, 24] though they seem to be more suitable for a parallel processing environment than a distributed environment. Our algorithm can easily be adapted to a completely decentralized environment, and due to the lower bounds above it can achieve a better work overhead than achievable by these trees

The issue of memory allocation preserving object locality has also been extensively investigated in the systems community, even though we are unable to pinpoint clear algorithmic statements. The open problem of finding locality preserving hashing in the context of range queries is stated in [16]. A recent work [14] attempts to guarantee locality properties for SHA-1 hashing used in Chord [36]. Data clustering on a single disk is best described in the "Fast-file" system which keeps all of the data in a single file contiguous (up to 64K at a time) [30]. More recent work at AT&T deals with clustering of all the data in the same web page together on a disk [35]. Other relevant work includes striping file systems [9, 19]. Zebra [19], for example, is a log-structured file system that stripes data across multiple disks for efficient parallel writes. GPFS [34] is a high-performance file system that stripes data across multiple disk servers each of which is a RAID array. Other work includes [32, 18, 10, 17, 3, 3, 27, 15].

2 Lower bounds and instability

This section states a number of results proved in the Appendix. The first result gives a lower bound on the work overhead of arbitrary order-preserving placement methods when using operational consistency.

Theorem 2.1 *For any operationally consistent, order-preserving online algorithm there is an (adaptive) $\Theta(\epsilon)$-bounded adversary so that the algorithm is unstable in a $(1-\epsilon)$-utilized system.*

The question is, what kind of properties an algorithm has to fulfill to be stable under a $\Theta(\epsilon)$-bounded adversary. A placement algorithm is called $(1+\epsilon)$-*faithful* if every node has a number of objects that is at least $\ell/(1+\epsilon)$ and at most $(1+\epsilon)\ell$ where ℓ is the average load in the system. Furthermore, a placement algorithm is called *node-order preserving* if it imposes a fixed order on the nodes, or equivalently, a fixed numbering from 1 to n, so that for all $i \in \{1,\ldots,n-1\}$, the ranges of the nodes fulfill $R_i \leq R_{i+1}$ at any time. Notice that a faithful algorithm is always node-order preserving since nodes can never be empty and therefore can never change their position in the node ordering. However, node-order preserving algorithms may not be faithful.

Theorem 2.2 *For any constant $0 < \epsilon < 1$, any (offline or online) operationally consistent and faithful placement algorithm in a $(1-\epsilon)$-utilized system is already unstable under $\Theta(1/n)$-bounded adversaries.*

Hence, faithful algorithms perform poorly. What about node-order preserving algorithms? The results in the previous work section imply the following theorem.

Theorem 2.3 *There is an operationally consistent node-order preserving online algorithm so that for any $0 < \epsilon < 1$ it holds that as long as the system is at most $(1-\epsilon)$-utilized, the algorithm is stable under any $\alpha/\log^2 n$-bounded adversary for some constant $\alpha > 0$.*

Thus, the ability to make nodes empty is crucial for a high efficiency. However, due to the lower bounds in [13, 12], these algorithms cannot be stable under any $\Theta(1/\log n)$-bounded adversary because of a work overhead of $\Omega(\log n)$, and it is conjectured in [4] that the correct lower bound is actually $\Omega(\log^2 n)$. In any way, it appears to be necessary to reorder nodes in order to obtain Theorem 2.1. This is exactly our approach.

3 An asymptotically optimal placement algorithm

We assume that we have a static set of n nodes of uniform capacity and that object updates are generated by a (λ, T)-bounded adversary. Later, we show how to extend the algorithm to dynamically changing sets of nodes and nodes of non-uniform capacity.

For simplicity, we assume in the following that $1/\epsilon$ is an integer. We partition the n nodes into two sets B and I. B represents the set of *busy* nodes, and I represents the set of *idle* nodes. Objects are only stored in the busy nodes. Busy nodes are organized in *clusters* of between $s(\epsilon)/2$ and $2s(\epsilon)$ nodes, where $s(\epsilon)$ is specified later. Each cluster consists of nodes with consecutive ranges. The range in $[0,1)$ a node v is responsible for is denoted by R_v, and the range of a cluster C is defined as $R_C = \bigcup_{v \in C} R_v$. $|C|$ denotes the number of nodes in C, and the nodes in C are denoted by $v_1^C, v_2^C, \ldots, v_{|C|}^C$. Let P_t denote the set of objects in the system at time t and let $\ell_t = |P_t|/n$ denote the average load of the system. We will use an integral version $\bar{\ell}_t$ of ℓ_t in a sense that $\bar{\ell}_t$ is equal to the last integer crossed by $\ell_{t'}$ with $t' \leq t$. This will prevent fast oscillations in the algorithm.

For any node v and cluster C, let $\ell_t(v)$ denote the load (i.e. the number of objects) of v and define the load of C, $\ell_t(C)$ and a special parameter $\hat{\ell}_t$ as

$$\ell_t(C) = \sum_{v \in C} \ell_t(v) \qquad \hat{\ell}_t = \lceil (1+\epsilon/2)\bar{\ell}_t \rceil + \Delta$$

for some fixed integer Δ to be specified later. As we will see later, Δ is important to cope with bursty injections of object updates.

The basic strategy of the algorithm is to move objects between nodes of a cluster so that at any step t, in every cluster C all but the node of highest range in C have close to $(1+\epsilon/2)\ell_t$ objects (see Fig. 1).

In the following, we assume that $0 < \epsilon < 1$, $s(\epsilon) = 20/\epsilon$, $\Delta \geq T$, and $\lambda \leq \epsilon/750$. It is possible to get much better constants, but we did not try to optimize them here to keep the proof at a reasonable length. The following fact is easy to check.

Fact 3.1 *The smoothing algorithm achieves transient consistency.*

Next we prove a lemma about the number of objects in the nodes that holds as long as the system is at most $(1-\epsilon)$-utilized. For each node v_k^C, let

$$\ell_t^*(v_k^C) = \begin{cases} \hat{\ell}_t & \text{if } k < |C| \\ \ell_t(C) - (|C|-1)\hat{\ell}_t & \text{otherwise} \end{cases} \tag{1}$$

1. Each node v_k^C computes $\delta_\ell = (k-1)\hat{\ell}_t - \sum_{i=1}^{k-1} \ell_t(v_i^C)$ and $\delta_r = \sum_{i=1}^{k} \ell_t(v_i^C) - k \cdot \hat{\ell}_t$.
   (a) If $k > 1$ and $\delta_\ell \geq 1$, then $\{$*too few objects in* $v_1^C, \ldots, v_{k-1}^C\}$
      i. the object with lowest number in v_k^C is moved to v_{k-1}^C.
      ii. If $k = |C|$ and v_k is empty, v_k^C is taken out of C and inserted as an idle node in I.
   (b) If $k < |C|$ and $\delta_r \geq 1$, then $\{$*too many objects in* $v_1^C, \ldots, v_k^C\}$
      i. the object with highest number in v_k^C is moved to v_{k+1}^C.
   (c) If $k = |C|$ and $\delta_r \geq 1$, then
      i. an idle node is fetched from I and integrated into C with number $k+1$.
      ii. the object with highest number in v_k^C is moved to v_{k+1}^C.
2. Each node in C receives the objects moved to it and updates its range accordingly.
3. Each node processes the newly injected object updates belonging to its range.
4. If $|C|$ exceeds or equal to $2s(\epsilon)$, then
   (a) C is split into two clusters of size $s(\epsilon)$.
5. If $|C|$ goes below or equal to $s(\epsilon)/2$, apply Procedure MERGE to C (Fig. 2).

Figure 1: The smoothing algorithm, performed in every cluster C in each time step t.

The proofs of the statements in this section are given in Section B in the Appendix. The next lemma is the most difficult one to prove. We use a potential method for it. The complexity of applying this method comes from the fact that one cannot just look at individual nodes but has to look at clusters, but clusters may split and merge.

Lemma 3.2 *At any time t, it holds for every node v_k^C that $\ell_t(v_k^C) \in [\ell_t^*(v_k^C) - \epsilon\Delta, \ell_t^*(v_k^C) + \epsilon\Delta]$.*

Next we show that there are always sufficiently many idle nodes in I.

Lemma 3.3 *For all $\epsilon \leq 1$ it holds: If $s(\epsilon) \geq 20/\epsilon$, then at any point in time, $|I|$ is at least the current number of clusters in the system.*

Combining the lemmata yields the following theorem.

1. Procedure MERGE: cover all merge candidates in disjoint groups of 2 or 3 consecutive clusters.
   (a) for all C in a group (C, C') s.t. either only C or both C and C' are merge candidates:
      i. If $|C| + |C'| \leq 3s(\epsilon)/2$, then C and C' are merged into a single cluster.
      ii. else nodes of lowest range are moved from C' to C so that both have the same # of nodes.
   (b) for all C in a group (C, C', C'') that are all merge candidates: merge group into a single cluster.

Figure 2: The cluster merging procedure applied to every cluster C in each time step t.

Theorem 3.4 *As long as the system is at most $(1-\epsilon)$-utilized, it holds: For any (λ, T)-bounded adversary with $\lambda \leq \epsilon/750$ and $T \leq \Delta$ the smoothing algorithm is stable. Furthermore, the algorithm guarantees a $1+\epsilon$-fragmentation as long as the system is at least $2(1+\epsilon)(T+2)/(\epsilon c)$ utilized, where c is the capacity of a node.*

Next, we discuss some extensions of the algorithm.

3.1 Handling arrivals and departures of nodes

Suppose that we allow nodes to join and gracefully leave the system. If a node joins, it is initially declared an idle node. If a node v wants to leave, the following strategy is used.

If v is an idle node, it can just leave. Otherwise, suppose v is a busy node in some cluster C. Then v fetches an idle node w and moves all of its objects in decreasing order to w. While these movements are happening, v will still accept all objects from its predecessor in C, but w will receive all objects from v's successor. Once v is empty, it can leave the system.

¿From Lemma 3.3 and Theorem 3.4 it follows:

Theorem 3.5 *As long as the system is at most $(1-\epsilon)$-utilized and the rate of node departures in a cluster is at most $\rho \cdot \epsilon/C$ for some constant $\rho > 0$, where C is the capacity of a node, the smoothing algorithm achieves the same performance as in Theorem 3.4.*

3.2 Nodes with non-uniform capacities

Suppose we have a system of non-uniform nodes, i.e. each node has a different capacity, and this capacity may even change over time. Given some time point t, we define the capacity of a node v by $C_t(v)$ and the average capacity of the system as $C_t = n^{-1} \sum_v C_t(v)$. Furthermore, let $\bar{C}_t(v)$ be the discretized version of $C_t(v)/C_t$. Instead of a common $\bar{\ell}_t$, we will then use $\bar{\ell}_t \cdot \bar{C}_t(v)$ in the algorithm. This will give the following result:

Theorem 3.6 *As long as the system is at most $(1-\epsilon)$-utilized, $C_t(v)$ changes by at most λT in T steps for any v and $\lambda = O(\epsilon)$ is sufficiently small, the smoothing algorithm is stable against arbitrary (λ, T)-bounded adversaries.*

Notice that considering $1+\epsilon$-fragmentation does not make sense in a heterogeneous environment because the number of nodes storing a range of objects depends on their capacities.

4 A decentralized storage system

The algorithm in Figure 1 has the advantage that it can be turned into an algorithm for the distributed setting. For this, we basically use three overlay networks – one for computing the average load, one for managing the busy nodes, and one for managing the idle nodes –, and each node in the busy network has a random link to the network of idle nodes to fetch idle nodes or to move to the idle nodes if necessary. This represents the first alternative to the DHT-based peer-to-peer systems that can use the *real* names of the data objects instead of their hashed names for consistent mapping while preserving a load balanced distribution.

In order to convert the smoothing algorithm into a local control algorithm for decentralized storage systems such as peer-to-peer systems, several issues have to be addressed:

- *How to break symmetry?* Merge candidates need a mechanism to decide with whom to merge. Clusters have to coordinate their selection of idle nodes.

- *How to organize busy and idle nodes in a distributed setting?* Busy and idle nodes have to be interconnected to allow the access to busy and idle nodes and to move nodes between the two sets.
- *How to determine the average load?* We need a distributed mechanism that can quickly and accurately determine the average load of the system.

4.1 The basic structure

Recall that the algorithm in Figure 1 partitions the nodes into two classes: *busy* nodes and *idle* nodes. The busy nodes represent the group of nodes responsible for storing the objects, and the idle nodes do not store any objects and will be used as floating resources.

Let V be the set of all nodes, B be the set of busy nodes, I be the set of idle nodes, and F be the set of busy nodes representing the last node of a cluster. We will use the following graphs to interconnect the nodes:

- $G_A = (V, E_A)$: This graph interconnects all nodes with the sole purpose of determining the average load.
- $G_B = (B, E_B)$: This graph consists of a cycle in which the busy nodes are ordered according to their names (resp. the ranges that they represent). Also, every cluster of busy nodes is completely interconnected.
- $G_F = (F, E_F)$: This graph interconnects the final nodes of each cluster in a way that allows to break the symmetry for merge operations.
- $G_I = (I, E_I)$: This graph contains all idle nodes.
- $G_{BI} = (B, I, E_{BI})$: This is a bipartite graph assigning a random idle node to each busy node.

First, we describe how new nodes join the system and old nodes leave the system, and then we describe in detail how each of the graphs works our system is composed of.

4.2 Joining and leaving the system

If a new node u joins the system by contacting some node v, then v calls the join operation of G_A to integrate u into that graph, and v calls the join operation of G_I, i.e. u is initially an idle node.

If a node u wants to leave the system and u is currently an idle node, then u starts the leave operation in G_I. Otherwise, u fetches a node in G_I via G_{BI} to exchange their roles and then starts the leave operation in G_I.

4.3 The graph G_A

Here, we use a so-called *deterministic skip graph* [20, 2]. In this skip graph, the nodes are connected in a doubly linked cycle. This cycle will be called *base cycle* and denoted by C_-, where "$-$" denotes the empty string. On top of this base cycle, a hierarchy of cycles is maintained so that the following invariants are fulfilled:

1. Each cycle C_b with binary string b that is of size at least 8 has two cycles, C_{b0} and C_{b1}, in an intertwined fashion on top of it so that $V(C_{b0}) \cup V(C_{b1}) = V(C_b)$ and $V(C_{b0}) \cap V(C_{b1}) = \emptyset$.
2. For any binary string b and any $x \in \{0, 1\}$, every edge in C_{bx} bridges at most 3 edges in C_b.

We also assume that the cycles have a common orientation so that the predecessor and successor of a node in a cycle is well defined. If these invariants are true, the following result is easy to show (see also [20, 2]):

Lemma 4.1 *The deterministic skip graph has a diameter of* $O(\log n)$.

Join

Next we explain how to join the skip graph. Suppose that an idle node u has been taken to be inserted between two nodes v and w. After this is completed for C_-, u checks the edges of C_0 and C_1 bridging it. If there is an edge $\{v', w'\}$ bridging it that violates Invariant 2, then u is integrated into the corresponding cycle by replacing $\{v', w'\}$ by $\{v', u\}$ and $\{u, w'\}$. Otherwise, any edge $\{v', w'\}$ from C_0 or C_1 bridging u is taken and replaced in the above way. In general, if u has already been integrated into C_b, it checks whether an edge in C_{b0} or C_{b1} bridging it is now violating the invariant. If so, this edge is handled in the way described above. Otherwise, any edge in C_{b0} or C_{b1} is taken and handled in the way above. It is easy to see that this strategy preserves the invariants. Also, we get:

Lemma 4.2 *Inserting a new node into the deterministic skip graph takes* $O(\log n)$ *time and work.*

Leave

Leaving the skip graph is similar to joining the skip graph. See [20, 2] for details. If a message can carry up to $\log n$ pointers, then the following result can be shown:

Lemma 4.3 *Removing a node from the deterministic skip graph takes* $O(\log n)$ *time and work.*

Computing the average load

The average load can be computed in the following way:

Initially, each node u forwards the tuple $(1, \ell_u)$ to all predecessors of u (including u) in C_- with an edge in C_0 or C_1 bridging u. Given an edge $e = \{v', w'\}$, we say that e *bridges* u if u is between v' and w', including v' but excluding w'. Each node v with edges in C_0 (resp. C_1) sums up all tuples (x_w, ℓ_w) it receives to $(x, \ell) = (\sum_w x_w, \sum_w \ell_w)$ and sends (x, ℓ) along edges in C_0 (resp. C_1) to all predecessors of v (including v) in C_0 (resp. C_1) with an edge in C_{00} or C_{01} (resp. C_{10} or C_{11}) bridging v. The summation and forwarding is continued for higher layers until all nodes at the highest layer have a tuple (x, ℓ). For each cycle C in the highest layer, the tuples are summed up to some tuple (n_v, L_v) for each node v in C, which has the following property:

Lemma 4.4 *At the end of the computation above, every node* v *in the deterministic skip graph has the same* (n_v, L_v)*, where* $n_v = n$*, the current number of nodes in the skip graph, and* L_v *is the current load in the skip graph. The computation takes* $O(\log n)$ *time.*

Hence, every node can easily compute the average load.

4.4 The graph G_B

Recall that $G_B = (B, E_B)$ consists of a cycle in which the busy nodes are ordered according to their names, and every cluster of busy nodes is completely interconnected. Hence, if (due to the smoothing algorithm) some cluster integrates a new node u, u will be inserted into the cycle and will be given edges to all other nodes in that cluster. This can certainly be done with constant time and work.

4.5 The graph G_F

In order to break the symmetry of merge operations, we form groups of 2 or 3 final nodes in the following way. Each node in G_F has a color specified by the mapping $c : V_F \to \{black, white\}$. We want to maintain the following invariant for c:

G_F contains at most two consecutive nodes of the same color.

In order to maintain this invariant, we use the following rules when a new node u joins G_F: If u's neighbors have the same color, or one of u's neighbors has a neighbor with the same color, then u uses the other color. Otherwise, u chooses any color. If a node u leaves, then its right neighbor, v, checks whether afterwards the invariant is violated. If so, it changes its color.

The colors allow the clusters to be organized in groups of size 2 or 3:

- For every white node $v \in \mathcal{V}_F$ where the predecessor u and successor w of distance 2 fulfill $c(u) = c(v) = c(w)$, v and its successor form a group.
- For every white node $v \in \mathcal{V}_F$ that has a white successor w, v, w, and the successor of w form a group.
- For every white node $v \in \mathcal{V}_F$ that has two black successors, v and its two successors form a group.

The following lemma is not hard to check:

Lemma 4.5 *The coloring scheme breaks symmetry in a unique way so that every node belongs to exactly one group.*

Thus, merging can be done in the same way as described in Figure 2.

4.6 The graph G_I

For G_I we can use any DHT-based overlay network, such as Chord [36], i.e. every node receives a random number in $[0, 1)$ as its ID. The nodes are ordered on a cycle according to their IDs and every node with ID x has shortcut pointers to the closest successors of $x + 1/2^i$ for every $i \in \mathbb{N}$. Joining and leaving G_I can be done as in Chord.

4.7 The graph G_{BI}

Every busy node v selects a random number x_v and maintains a pointer to the closest successor in G_I to x_v. This makes sure that the edges are randomly distributed among the idle nodes. The edges are used whenever a final node of a cluster becomes idle and therefore wants to join G_I, or a final node of a cluster wants to integrate a new idle node into it, or a busy node simply wants to leave the system. Since there are more idle nodes than final nodes and busy nodes rarely leave, it is not difficult to see that it only takes $O(\log n)$ time and work for any one of the cases above to be processed, w.h.p. Details will be given in a final paper.

5 Conclusions

In this paper, we only looked at the problem of maintaining a low fragmentation for one-dimensional data with a small work overhead. An interesting future problem would be to look also at problems for higher-dimensional data, since they have many interesting applications in data bases and geographic information systems. Furthermore, we only looked at worst case scenarios concerning the injection of update requests.

In scenarios in which the distribution of update requests is highly concentrated on certain areas in the name space, it should be possible to obtain stability for much higher injection rates than just $O(\epsilon)$. Exploring these issues would be particularly interesting for single-disk systems because a work overhead of $O(1/\epsilon)$ as implied by our results is unacceptable for single disk management.

References

[1] S. Alstrup, G. Stolting Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *Proc. of the 41st IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 198–207, 2000.

[2] B. Awerbuch and C. Scheideler. The Hyperring: A low-congestion deterministic data structure for distributed environments. In *Proc. of the 15th ACM/SIAM Symp. on Discrete Algorithms (SODA)*, 2004.

[3] R.A. Baeza-Yates and H. Soza-Pollman. Analysis of linear hashing revisited. *Nordic Journal of Computing*, 5(1):70–85, 1998.

[4] M. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. of the 41st*.

[5] A. Bolour. Optimal retrieval algorithms for small region queries. *SIAM Journal on Computing*, 10(4):721–741, 1981.

[6] 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.

[7] 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.

[8] G. Brodal, R. Fagerberg, and R. Jacob. Cache-oblivious search trees via trees of small height. In *Proc. of the 13th ACM/SIAM Symp. on Discrete Algorithms (SODA)*, pages 39–48, 2002.

[9] L.-F. Cabrera and D.D.E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computer Systems*, 4(4):405–436, 1991.

[10] C.Y. Chen, C.C. Chang, and R.C.T. Lee. Optimal mmi file systems for orthogonal range queries. *Information Systems*, 18(1):37–54, 1993.

[11] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Symposium on Operating Systems Principles*, pages 202–215, 2001.

[12] P.F. Dietz, J.I. Seiferas, and J. Zhang. A tight lower bound for on-line monotonic list labeling. In *Proc. of the 6th Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 131–142, 1994.

[13] P.F. Dietz and J. Zhang. Lower bounds for monotonic list labeling. In *Proc. of the 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 173–180, 1990.

[14] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proc. of the First Biennial Conference on Innovative Data Systems Research*, 2003.

[15] S. Hanke, T. Ottmann, and E. Soisalon-Soininen. Relaxed balanced red-black trees. In *CIAC*, pages 193–204, 1997.

[16] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based peer-to-peer networks. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[17] E.P. Harris. *Towards optimal storage design for efficient query processing in relational database systems*. PhD thesis, The University of Melbourne, Parkville, Victoria 3052, Australia, December 1994.

[18] E.P. Harris and K. Ramamohanarao. Using optimized multiattribute hash indexes for hash joins. In *Proc. of the 5th Australasian Database Conference*, pages 92–111, 1994.

[19] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 309–329. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[20] N. J. Harvey and I. Munro. Brief announcement: Deterministic skipnet. In *Proc. of the 22nd IEEE Symp. on Principles of Distributed Computing (PODC)*, 2003.

[21] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. pages 604–613, 1998.

[22] P. Indyk, R. Motwani, P. Raghavan, and S. Vempala. Locality-preserving hashing in multidimensional spaces. In *Proc. of the 29th ACM Symp. on Theory of Computing (STOC)*.

[23] A. Itai, A.G. Konheim, and M. Rodeh. A sparse table implementation of sorted sets. Technical Report Research Report RC 9146, IBM T.J. Watson Research Center, Yorktown Heights, New York, November 1981.

[24] T. Johnson and D. Shasha. The performance of concurrent data structure algorithms. *Transactions on Database Systems*, pages 51–101, 1993.

[25] 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.

[26] P. Krishna and T. Johnson. Highly scalable data balanced distributed B-trees, 1995.

[27] H.T. Kung and P. Lehman. A concurrent database manipulation problem: binary search trees. *ACM Transactions on Database Systems*, 5(3):339–353, 1980.

[28] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proc. of the 30th ACM Symp. on Theory of Computing (STOC)*, pages 614–623, 1998.

[29] N. Linial and O. Sasson. Non-expansive hashing. In *Proc. of the 28th ACM Symp. on Theory of Computing (STOC)*, pages 509–518, 1996.

[30] M.K. McKusick, W.N. Joy, S.J. Leffler, and R.S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[31] X. Messeguer. Skip trees, an alternative data structure to skip lists in a concurrent approach. *Informatique Theorique et Applications*, 31(3):251–269, 1997.

[32] K. Ramamohanarao and E.P. Harris. Effective clustering of records for fast query processing. In *CODAS*, pages 516–525, 1996.

[33] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM '01*, 2001.

[34] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. of the 1st Conference on File and Storage Technologies (FAST)*, 2002.

[35] E. Shriver, E. Gabber, L. Huang, and C.A. Stein. Storage management for web proxies. pages 203–216, 2001.

[36] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *SIGCOMM '01*, pages 149–160, 2001.

[37] S. Subramanian and S. Ramaswamy. The P-range tree: A new data structure for range searching in secondary memory. In *Proc. of the 6th ACM/SIAM Symp. on Discrete Algorithms (SODA)*, 1995.

[38] D.E. Willard. New data structures for orthogonal range queries. *SIAM Journal on Computing*, 14:232–253, 1985.

[39] B.Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. In *UCB Technical Report UCB/CSD-01-1141*, 2001.

A Proofs in Section 2

A.1 Proof of Theorem 2.1

We only sketch the proof. Consider any fixed $\epsilon > 0$ with $\epsilon = o(n)$. Let C be the *capacity* of the nodes, i.e. the number of objects that can be stored in a node. The adversarial strategy works in rounds. At the beginning of each round, we have a $(1-\epsilon)$-utilized system, i.e. there are $(1-\epsilon)n \cdot C$ objects in the system.

Suppose that we are at the beginning of some round. Since the system is $(1-\epsilon)$-utilized, we must have at least $\epsilon n/6$ groups $G_1, \ldots, G_m$ of $1/\epsilon$ nodes of consecutive names where each C_i is at least $(1-2\epsilon)$-utilized. For each of these groups, the adversary injects insert requests so that without any change in range, each node in the groups would be $(1+\epsilon)$-utilized. Certainly, at most $n/6 \cdot 3\epsilon = \epsilon n/2$ requests suffice for this. Thus, afterwards, we have a system that is at most $(1-\epsilon) + \epsilon/2 = (1-\epsilon/2)$-utilized. Finally, any objects are deleted to get back to a $(1-\epsilon)$-utilized system for the next round.

We are interested in proving a lower bound for the number of object movements necessary to handle the insertions of objects. Let $H = \bigcup_{j=1}^{m} G_j$ denote the set of heavy nodes and $L = V \setminus H$ denote the set of light nodes. To simplify the proof, we consider only so-called *lazy* reassignment algorithms that first place all the inserted objects before moving any objects to other nodes. Since an optimal algorithm may have moved the newly inserted objects immediately to different nodes, an optimal lazy algorithm needs at most $C \cdot \epsilon n/2$ more object movements than an optimal algorithm.

Let $R_H \subset [0,1)$ denote the ranges covered by H and R_L denote the ranges covered by L. Consider the assignment of ranges to nodes in H at the beginning and the end of the object movement phase (i.e. after inserting the new objects and before deleting any old objects). We do some modifications to the object movements to make our life easier.

First of all, at the end there must be nodes in L that take over ranges inside R_H to cope with the overload in H. Suppose that there are also nodes in H that take over a range inside R_L. Then any such node can replace its role with a node moved from R_L into R_H without increasing the number of object replacements. Hence, we only have to consider object movements where at the end every node in H has a range that is at least partly in R_H.

Next, consider the set S of all nodes with ranges intersecting with R_H. For every range R covered by a node in S, let the node $v \in H$ that at the beginning had the most objects in R be called its *owner*. Then we will rearrange the nodes so that every range is taken by is owner, if possible. Since every node in H had originally $(1+\epsilon)C$ objects and every node can store at most C objects, there must always be a range at the end for which it is the owner. If there is more than one, then we assign the owner to the range with the most objects of which it is the owner. For the nodes in H, this can only lower the number of objects that had to be moved out of them. But for the nodes in L that now have ranges partly in R_H, this may increase the number of object movements. This increase, however, is bounded by $m \cdot 2C$ (C objects to the lower and higher end of every R_{G_i}), which is at most $C \cdot \epsilon n/3$.

Hence, we only have to focus on final range assignments where each node in H has a range overlapping with its old range. Next, we bound $|S \cap L|$.

The nodes in H were originally at most 1-utilized. Hence, the average utilization of nodes in L must be at least $((1-\epsilon)n - n/6)/(5n/6) = 1 - 6\epsilon/5$. Therefore, the average utilization of those nodes in L that are more than C objects away from R_H must be at least

$$\frac{(5/6-\epsilon)n - 2m}{(5/6)n} = 1 - 8\epsilon/5 \; .$$

Hence, at most $2m + (8/5)\epsilon n < 2\epsilon n$ nodes in L can be in S without overloading the remaining nodes in L. Hence, $|S \cap L| \leq 2\epsilon n$.

Since there are $\epsilon n/6$ groups G_i, there must be at least $\epsilon n/12$ groups with at most 24 nodes v in L with $R_v \cap R_{G_i} \neq \emptyset$ because otherwise we have a contradiction to the upper bound on $|S \cap L|$.

Let G_i be any one of these groups. At the end, G_i consists of consecutive sequences of nodes in H, each covering part of its original range, interrupted by nodes in L. Consider any consecutive sequence of H-nodes, and number them from 1 to k. Let x_i denote the number of objects remaining at node i and let $y_{i,j}$ be the number of objects moved from node i to node $j \in \{i-1, i+1\}$. It holds that $x_i + y_{i,i-1} + y_{i,i+1} = (1+\epsilon)C$ for every i. No matter how $y_{1,0}$ is chosen, since we must have $x_j \leq C$ for all j and the end, the number of objects that have to be moved between the nodes 1 to k is at least

$$O\left(\sum_{i=1}^{k} i \cdot \epsilon C\right) = O(k^2 \cdot \epsilon C)$$

Let $k_1, \ldots, k_s$ be the lengths of all sequences of consecutive H-nodes in G_i. Since $s \leq 24$ and G_i has $1/\epsilon$ nodes, it holds that the number of object movements is at least

$$O\left(\sum_{j=1}^{s} k_j^2 \cdot \epsilon C\right) = O(C/\epsilon)$$

Hence, summing up over all groups, it follows that over all nodes in the system, any strategy preserving an ordered object placement that does not violate the capacity constraints has to move $\Omega(n \cdot C)$ objects, whereas the number of object updates is only $O(\epsilon n \cdot C)$. Hence, there is a $(\Theta(\epsilon), T)$-bounded adversary causing more object movements than the system can handle, and therefore causing instability.

A.2 Proof of Theorem 2.2

We only sketch the proof. Also here, the adversary works in rounds. Suppose that initially we have $(1-\epsilon)n \cdot C$ objects in the system. In each round, the adversary cuts the current set of objects into three sets S_1, S_2, S_3 where S_1 contains the lowest third, S_2 contains the middle third, and S_3 contains the highest third of the names. Now, the adversary removes ϵC objects every C consecutive objects in S_1 and inserts ϵC objects every C consecutive objects in S_3.

Recall that the nodes are numbered from 1 to n with node i always having a lower range than node $i+1$. Consider any round r and any object o that is in set S_2 at round r. Suppose that o is the kth smallest object in the system. Then, the lowest node at which o can be at round r is k/C. On the other hand, the highest node at which o can be at round $r+j$ is $n-((1-\epsilon)n \cdot C-k+jC \cdot \epsilon n/3)/C = \epsilon n+k/C-j \cdot \epsilon n/3$. It holds that $k/C \geq \epsilon n+k/C-j \cdot \epsilon n/3$ if and only if $j \geq 3(1+d/(\epsilon n))$. Hence, after 9 rounds, each object in S_2 must have moved to a node of distance at least $d = 2\epsilon n$ of its original node. Thus, the total work for moving the objects in S_2 over 9 rounds is at least $((1-\epsilon)n/3)C \cdot 2\epsilon n$. Therefore, on average we need $\Theta((1-\epsilon)\epsilon n)$ object movements per node for these update requests to maintain an ordered object placement. On the other hand, every node only receives $O(\epsilon C)$ update requests within 9 rounds. Thus, the maximum rate that can be sustained is $\Theta(\epsilon/((1-\epsilon)\epsilon n)) = \Theta(1/((1-\epsilon)n))$.

B Proofs in Section 3

B.1 Proof of Lemma 3.2

We first show some basic facts about the behavior of the algorithm.

Claim B.1 *For every node v_k^C it holds that if $\ell_t(v_k^C) \geq \ell_t^*(v_k^C)$, then movements of objects do not increase $\ell_t(v_k^C)$, and if $\ell_t(v_k^C) \leq \ell_t^*(v_k^C)$, then movements of objects do not decrease $\ell_t(v_k^C)$.*

Proof. First, consider the case that $\ell_t(v_k^C) \geq \ell_t^*(v_k^C)$. If $k = |C|$, then it follows that

$$\sum_{i=1}^{|C|-1} \ell_t(v_i^C) \leq (|C|-1)\hat{\ell}_t$$

and therefore no object is moved to v_k^C. So suppose that $k < |C|$. If an object is moved from v_{k-1}^C to v_k^C, then also an object is moved from v_k^C to v_{k+1}^C because

$$\sum_{i=1}^{k-1} \ell_t(v_i^C) \geq (k-1)\hat{\ell}_t + 1 \quad \text{and therefore} \quad \sum_{i=1}^{k} \ell_t(v_i^C) \geq k \cdot \hat{\ell}_t + 1\,.$$

If an object is moved from v_{k+1}^C to v_k^C, then also an object is moved from v_k^C to v_{k-1}^C because

$$\sum_{i=1}^{k} \ell_t(v_i^C) \leq k \cdot \hat{\ell}_t - 1 \quad \text{and therefore} \quad \sum_{i=1}^{k-1} \ell_t(v_i^C) \leq (k-1)\hat{\ell}_t - 1\,.$$

The proof for $\ell_t(v_k^C) \leq \hat{\ell}_t$ is similar. □

Claim B.2 *At any time t in which there is at least one node in a cluster C which deviates by at least one from its ideal value, there is a node v_k^C with $k < |C|$ whose deviation from its ideal value is reduced by 1.*

Proof. Let v_k^C be the node of lowest $k < |C|$ with $\ell_t(v_i^C) \neq \hat{\ell}_t$. (If there is no such node, the distribution is already perfect.) It follows from our balancing rules that v_k^C will lose an object if $\ell(v_k^C) < \hat{\ell}_t$ and will gain an object if $\ell(v_k^C) > \hat{\ell}_t$. Hence, its deviation is reduced by 1. □

This motivates the following definition.

Definition B.3 *For every cluster C, consider the potential*

$$\phi_t(C) = \sum_{i=1}^{|C|-1} |\ell_t(v_i^C) - \ell_t^*(v_i^C)|$$

Claim B.1 and —B.2 immediately imply the following result.

Claim B.4 *If at the beginning of step t, $\phi_t(C) \geq 1$, then stages 1 and 2 of the algorithm decrease $\phi_t(C)$ by at least 1.*

Now, consider the time to be partitioned into consecutive time frames of length T. Our aim is to show that if λ is sufficiently small, then for every time frame F and every node v_k^C, there is a time step $t \in F$ at which $\ell_t(v_k^C) = \ell_t^*(v_k^C)$. Suppose for now that this is correct. Then these time steps can be at most $2T$ steps apart from each other. Furthermore, it follows from Claim B.1 that the deviation of v_k^C from $\ell^*(v_k^C)$ can only be increased if $\hat{\ell}_t$ changes or object updates cause the number of objects in v_k^C to change. Since at most $2\lambda T$ object updates can be injected per node in $2T$ steps, the number of objects in v_k^C changes by at most $2\lambda T$ and $\hat{\ell}_t$ changes by at most $\lceil 1 + \epsilon/2 \rceil \lambda T$ in $2T$ steps. Hence, it follows that at any time t,

$$\ell(v_k^C) \in [\ell_t^*(v_k^C) - \lceil (2 + \epsilon/2)\lambda T \rceil, \ell_t^*(v_k^C) + \lceil (2 + \epsilon/2)\lambda T \rceil] \,, \tag{2}$$

which results in a deviation from $\ell_t^*(v_k^C)$ of at most $\epsilon\Delta$ if $\Delta \geq T$ and $\lambda \leq \epsilon/3$. Thus, it remains to prove the following claim.

Claim B.5 *If $\lambda \leq \alpha\epsilon$ for some sufficiently small constant $\alpha > 0$, then it holds: For every time frame F and every node v_k^C, there is a time step $t \in F$ at which $\ell_t(v_k^C) = \ell_t^*(v_k^C)$.*

Proof. We will prove the lemma by induction, using a stronger property than in the claim above, namely that for every time frame F and every cluster C there is a time step $t \in F$ with $\phi(C) = 0$. For this statement to make sense, we have to specify how to adapt C to split and merge events. Consider any time interval I and some fixed cluster C existing at the beginning of I. If C merges with another cluster during I, we identify C with the resulting cluster. Also, if C passes nodes to its predecessor or receives nodes from its successor during I, we identify C with the resulting cluster. If C splits into two clusters C_1 and C_2, we will still view for the analysis the two clusters as a single cluster C in I, but redefine in this case $\phi(C)$ as $\phi(C) = \phi(C_1) + \phi(C_2)$. Hence, if $\phi(C) = 0$ for some time in I, then it is also true that $\phi(C_1) = \phi(C_2) = 0$ for that time. Thus, all clusters that were newly created in I are still covered so that a proof by induction can be constructed for our claim.

Since at the beginning we start out with an empty system (e.g., B just consists of a single empty node), the induction hypothesis is correct for the first time frame. So let us assume that the hypothesis is true for some time frame F. Then we will show that it is also true for its successor F'.

Consider some fixed cluster C with $\phi(C) = 0$ at some time point $t \in F$. (Note that by our arguments above, C is a real cluster, and every cluster in F' will be covered by one of these clusters.) Let I be the time interval from t till the end of F'.

First, we bound the maximum size C can have during I using our manipulations. Since in $2T$ steps every node receives at most $2\lambda T$ object updates, a cluster of size s can receive at most $s \cdot 2\lambda T$ updates. Initially, C has a size of at most $2s(\epsilon)$. Hence, in order to have a size of s at the end, it must hold that $2s(\epsilon) + (s \cdot 2\lambda T)/\Delta \geq s$. If $\Delta \geq 8\lambda T$, then s can be at most $3s(\epsilon)$ to fulfill this inequality. Thus, C splits into at most 2 clusters in I, say C_1 and C_2, that are initially of size $s(\epsilon)$. Each time C_1 or C_2 merges, this creates a cluster of size at most $3s(\epsilon/2)$. Hence, altogether the size of C during I can be at most $3s(\epsilon)$.

Next, we go through all possible events in F and F' that can possibly increase the potential of C.

- Injections of object updates: Since $|C| \leq 3s(\epsilon)$, the number of objects in C can change by at most $2\lambda T \cdot 3s(\epsilon)$, and therefore $\phi(C)$ can increase by at most

$$6\lambda T \cdot s(\epsilon) \tag{3}$$

- Changes in $\hat{\ell}_t$: Each change in $\hat{\ell}_t$ requires at least n object updates. Since there are at most $2\lambda T \cdot n$ object updates in I, $\hat{\ell}_t$ can change at most $2\lambda T$ times. Each time $\hat{\ell}_t$ changes, $\phi(C)$ increases by at most $3s(\epsilon) \cdot \lceil 1+\epsilon/2 \rceil \leq 6s(\epsilon)$. Hence, overall, changes in $\hat{\ell}_t$ increase $\phi(C)$ by at most

$$12\lambda T \cdot s(\epsilon) \tag{4}$$

- Split events: A split event does not increase $\phi(C)$, because it will never increase the set of nodes under consideration in $\phi(C)$.

- Merge events or events in which nodes are moved to C: Each time C merges with another cluster, its size is increased by at least $s(\epsilon)/2$. If C is involved in a split event, then it must initially have a size of more than $3s(\epsilon)/2$. Since it can only shrink or grow by at most $s(\epsilon)/2$ nodes in I, it cannot be involved in a merge event or an event in which nodes are moved to it before the split event. The resulting clusters C_1 and C_2 are initially of size $s(\epsilon)$. Since C_1 cannot become a merge candidate, C_2 will not be involved in a merging throughout I. However, it may happen that C_1 merges with its predecessor. In this case, C_1 gets additional $s(\epsilon)/2$ nodes. Since C_1 can only grow or shrink by less than $s(\epsilon)/2$ nodes, C_1 cannot be part of a merge operation any more.

  So suppose that C is not involved in a split event. Since C's initial size is at least $s(\epsilon)/2$, two merge events would give a total size of at least $3s(\epsilon)/2$ under the assumption that no node leaves. Since C does not split, C would have to shrink to $s(\epsilon)$ nodes to be part of another merge event, which is not possible.

  Hence, C can participate in at most two merging events. Since a merging cluster can have a size of at most $s(\epsilon)$ and from equation (2) it follows that the load of every node deviates from its ideal load by at most $\lceil 2+\epsilon/2 \rceil \lambda T$, each merging increases $\phi(C)$ by at most $s(\epsilon) \cdot \lceil 2+\epsilon/2 \rceil \lambda T$. Hence, merging events can increase $\phi(C)$ by at most

$$4\lceil (2+\epsilon/2)\lambda T \rceil \cdot s(\epsilon) \leq 12\lambda T \cdot s(\epsilon) \tag{5}$$

- Events in which nodes are moved to C: This only happens if C is a merge candidate, i.e. it is of size $s(\epsilon)/2$. Since $|C|$ will be increased by at least $s(\epsilon)/4$ nodes each time nodes are moved to C, C can only participate twice in such an event. Each event can increase $\phi(C)$ by at most $s(\epsilon) \cdot \lceil 2+\epsilon/2 \rceil \lambda T$. Hence, the total increase of $\phi(C)$ due to these events is at most

$$2\lceil (2+\epsilon/2)\lambda T \rceil \cdot s(\epsilon) \leq 6\lambda T \cdot s(\epsilon) \tag{6}$$

- Events in which nodes are taken from C: This can only decrease $\phi(C)$.

Combining (3) to (6), the total increase of $\phi(C)$ in I can be at most

$$6\lambda T \cdot s(\epsilon) + 12\lambda T \cdot s(\epsilon) + 12\lambda T \cdot s(\epsilon) + 6\lambda T \cdot s(\epsilon) \leq 36\lambda T \cdot s(\epsilon)$$

This is less than T if $\lambda < 1/(36s(\epsilon))$ for some sufficiently small constant $\alpha > 0$. Since according to Claim B.2 $\phi(C)$ decreases in stage 1 and 2 of each step as long as $\phi(C) > 0$, there must be a time point in F' where $\phi(C) = 0$, completing the proof. □

The proof of the claim ends the proof of Lemma 3.2.

B.2 Proof of Lemma 3.3

First, we show that at any point in time it holds for every cluster C that $|C| \leq \lceil \ell_t(C)/\hat{\ell}_t \rceil + 1$.

Let $\mathcal{C}_t$ be the set of clusters in B at step t. Consider any cluster $C \in \mathcal{C}_t$. From the proof of the previous lemma we know that there was a step t' at most $2T$ steps before t where $\phi(C) = 0$, which implies that $|C| \leq \lceil \ell_t(C)/\hat{\ell}_t \rceil$.

During these $2T$ steps, $\ell(C)$ can change by at most $2s(\epsilon) \cdot 2\lambda T$. Since $2s(\epsilon) \cdot 2\lambda T \leq \Delta$ for our choices of $s(\epsilon)$ and λ, this means that $|C|$ can change by at most 1. Hence, in the worst case, $|C| \leq \lceil \ell_t(C)/\hat{\ell}_t \rceil + 1$. Thus,

$$\begin{aligned} |B| &= \sum_{C \in \mathcal{C}_t} |C| \leq \sum_{C \in \mathcal{C}_t} \left\lceil \frac{\ell_t(C)}{\hat{\ell}_t} \right\rceil + 1 \leq \frac{1}{\hat{\ell}_t} \cdot n \cdot \ell_t + 2|\mathcal{C}_t| \\ &\leq \frac{n \cdot \ell_t}{(1+\epsilon/2)\bar{\ell}_t + \Delta} + \frac{2n}{s(\epsilon)/2} \leq \frac{n}{1+\epsilon/2} + \frac{\epsilon n}{5} \leq \epsilon n \left(1 - \frac{1}{8}\right) . \end{aligned}$$

Hence, $|I| \geq \epsilon n/8$. Since $|\mathcal{C}_t| \leq 2n/s(\epsilon) = \epsilon n/10$, the lemma follows.

B.3 Proof of Theorem 3.4

First, we show that the algorithm is $1+\epsilon$-balanced. For any node v we have $\ell_t(v) \leq \hat{\ell}_t + \epsilon\Delta$. Thus, with $\ell_t \geq 2(1+\epsilon)(\Delta+1)/\epsilon$ we get

$$\begin{aligned} \ell_t(v) &\leq \lceil (1+\epsilon/2)\bar{\ell}_t \rceil + (1+\epsilon)\Delta \\ &\leq (1+\epsilon/2)(\ell_t+1) + 1 + (1+\epsilon)\left(\ell_t \cdot \frac{\epsilon}{2(1+\epsilon)} - 2\right) \\ &\leq (1+\epsilon)\ell_t . \end{aligned}$$

Next, we consider fragmentation. We know that for every cluster C, $|C| \geq s(\epsilon)/2$. Furthermore, we know from Lemma 3.2 that every busy node v that is not the last one in its cluster,

$$\ell_t(v) \geq \hat{\ell}_v - \epsilon\Delta = \lceil (1+\epsilon/2)\bar{\ell}_t \rceil + (1-\epsilon)\Delta \geq (1+\epsilon/2)\ell_t .$$

Hence, for every range R containing at least $1/\epsilon$ objects, the number of nodes containing objects of R is at most

$$\frac{\ell_t(R)}{(s(\epsilon)/2-1)\cdot\ell_t} + \frac{\ell_t(R)}{(1+\epsilon/2)\ell_t} \leq (1+\epsilon) \cdot \frac{\ell_t(R)}{\ell_t} .$$

Finally, we bound the work overhead. The potential of a cluster is only changed by object updates or changes in $\bar{\ell}_t$. Every object update changes $\phi(C)$ of the corresponding cluster C by at most 1, and the work necessary to reduce $\phi(C)$ by 1 is at most $|C| \leq 2s(\epsilon)$. Furthermore, at least n object updates are necessary to change $\bar{\ell}_t$ by 1, and each change in $\bar{\ell}_t$ changes the potentials of each cluster C by at most $2|C|$. Hence, overall the work overhead is $O(1/\epsilon)$.