Consistent and Compact Data Management in Distributed Storage Systems

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

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

ABSTRACT

In this paper we consider the problem of maintaining a consistent mapping of a virtual object space to a set of memory modules, i.e. the object space can be decomposed into a set of ranges where every module is responsible for exactly one range. A module owning some range R is responsible for storing all objects in R. Besides consistency, we require the mapping to be *compact*, i.e. any object or consecutive range of objects should be spread out over as few memory modules as possible. A compact mapping 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.

In both settings, new objects may be added, old objects may be deleted, or objects may be modified over time. Each object consists of a set of data blocks of uniform size. So insert, delete, or modify operations on objects can be seen as insert or delete operations of data blocks. Each module can send or receive at most one data block in each unit of time and the injection of insert or delete requests for data blocks is under adversarial control. We prove asymptotically tight upper and lower bounds on the maximum rate at which the adversary can inject requests into the system so that a consistent and compact 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 $\Theta(\epsilon)$.

*Supported by NSF grant ANIR-0240551 and NSF grant CCR-0311795.

†Supported by NSF grant CCR-0311121 and NSF grant CCR-0311795.

SPAA'04, June 27–30, 2004, Barcelona, Spain.

Categories and Subject Descriptors

F.2.8 [**Analysis of Algorithms and Problem Complexity**]: Non-numerical Algorithms and Problems; H.2.4 [**Database Management**]: Systems—*Distributed databases*; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*File organization*

General Terms

Algorithms, Theory

Keywords

distributed data management, peer-to-peer systems, range queries, load balancing

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 (resp. array entries or 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 memory module, or at least to try to minimize the number of memory modules that store the array's entries, so that the entries can be accessed with low overhead.

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 objects that are close to each other in the object space also close to each other when mapping them to memory modules, i.e. to store them in a compact 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., [1, 5, 31, 33]), whereas the work for aggregating the results from different memory modules scales linearly with the number of modules involved in a range query. Thus, uniform hashing, i.e. globally scattering objects over the modules, makes range queries *very expensive*. Instead, this paper accomplishes the goal of compact data storage using a *deterministic, adaptive* memory assignment strategy that keeps the objects in order and that is efficiently manageable under adversarially generated requests in concurrent environments.

1.2 Our approach: models and results

We assume that objects are splittable into data blocks of uniform size and updates are done block-wise. Hence, insert, delete, or update requests of objects can be seen as insert and delete requests of data blocks. Each memory module can send or receive one data block in a time unit and has infinite processing power (i.e. we can neglect internal computation). Memory modules are simply denoted by *nodes* in the following, and instead of objects and object space we are just talking about data and data space, meaning here the individual data blocks the objects are composed of. We assume that all nodes have a *capacity* of L, i.e. they can store up to L data blocks.

Consistency. Let V be a set of memory modules and D be the virtual data space. Consider any one-to-one mapping $\mathsf{ID} : V \cup D \to [0, 1)$ and let $\mathsf{ID}(x)$ denote the *identification number*, or ID, of x. Similar to [19] and follow-up work in the area of peer-to-peer systems [30, 27, 34, 9], we say that a set of data is stored *consistently* among a set of nodes if each data item d is stored at the node v whose ID is the closest successor to the ID of d, i.e.

$$v = \text{argmin}\{\mathsf{ID}(w) \mid w \in V \text{ and } \mathsf{ID}(w) \geq \mathsf{ID}(d)\}$$

However, our approach significantly differs from these approaches in the following way:

- *data IDs:* in [19, 30, 27, 34, 9], the IDs of the data blocks are *static hash values* of their addresses. In the current paper, the IDs represent the *original* addresses in order to avoid fragmentation and enable (one-dimensional) range queries.
- *node IDs:* in [19, 30, 27, 34, 9], the IDs of the nodes are *static hash values* of their IP addresses or *random values*. In the current paper, the IDs of the nodes are dynamically changed to adapt the mapping to a changing set data items.

Compactness. A consistent data placement is called $(1+\gamma)$-*compact* if for any set S of k consecutive data blocks, S is distributed across at most $(1+\gamma)(k/L)+c$ nodes for some constant c. Obviously, $\gamma = 0$ is best possible. Thus, being $(1+\gamma)$-compact means, for example, that semi-group range queries cost only a $1+\gamma$ factor more work than in the best possible case.

Operational and transient consistency. To be able to adapt to a changing set of data items, the following two operations are necessary:

- *renaming* of node IDs and
- *migrating* data, i.e. moving data from one node to another.

We call these operations *critical* because they can get the system into an inconsistent state when not used carefully.

As a minimum requirement, the system should return to a consistent state after the completion of any insert or delete operation of a data item because otherwise data 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 (i.e. more restrictive) model of *transient consistency* requires consistency even in the transient state. That is, after every execution of a critical operation 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.

Notice that transient consistency requires a node to be empty before it can take over a new range that is not in the neighborhood of its old range.

Generation of insert/delete requests. Recall that every node has a capacity of L. We are interested in the *dynamic* setting where continuously new data items are inserted and old data items 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 insert/delete requests in each time step as long as for any range R that currently contains L data items, the average number of insert/delete requests injected by the adversary in a unit of time for blocks in R, i.e. its *injection rate*, is at most λ. If this holds for all time windows of size T, we call such an adversary a (λ, T)-bounded adversary. (λ, T)-bounded adversaries allow us, for example, to insert or delete objects consisting of up to λT data blocks.

Given such an adversary, an algorithm is called *stable* if it can preserve a consistent mapping without exceeding the capacity of a node at any time. Since we assume that every node can receive or send at most one data block per time unit, λ can be at most 1 for a 1-compact 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 $\sigma \in [0, 1]$, a σ-*utilized* system needs a σ fraction of its resources to store the current data.

Given a static set of nodes, we are interested in determining the maximum rate of requests that can be sustained given a certain memory utilization to keep the assignment of data to nodes consistent and compact without exceeding the capacity of a node. 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 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 a consistent and $1 + \epsilon$-compact 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.

Dynamic systems and dynamic capacities. We also consider the case that nodes continuously join and leave the system and that the nodes have arbitrary, non-uniform capacities. Supporting non-uniform capacities has the benefit that also other criteria than just compactness can be considered. For example, if the goal is to balance the data evenly among the nodes, the capacities of the nodes can be lowered so that a compact storage also implies a balanced storage. Furthermore, nodes with a better connection can be given a larger capacity because they can serve more requests, or nodes that want to leave the system can gradually lower their capacity to limit the influence of data movements on the performance of the system. To adjust our adversarial injection model to this setting, we assume that L is the maximum capacity of a node.

By viewing the data movements necessary to cope with a dynamic set of nodes and capacity changes as injections of requests, we can reduce these cases to the case of insertions and deletions of data in a static system, allowing us to carry over our stability result for a static system to a dynamic system of non-uniform capacities. Specifically, we prove (see Sections 3.1 and 3.2):

THEOREM 1.3. *A modification of the online algorithm in Figure 1 can maintain a consistent mapping under any $\alpha \cdot \epsilon$-bounded adversary (for some constant $\alpha > 0$, considering both insert/delete requests and capacity changes) as long as the system is at most $(1 - \epsilon)$-utilized at any point in time.*

Certainly, considering $1 + \epsilon$-compactness does not make sense in a heterogeneous environment because the number of nodes storing a range of data blocks 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 nodes and moves data between the nodes to prevent the data load at a node from exceeding its capacity. That is, all the work is done by the dispatcher. For this model, consistent search structures have been presented in the context of balanced search trees and monotonic list labeling. Several papers independently presented solutions for maintaining embedded balanced search trees so that, in our context, the dispatcher needs an amortized work of $O(\log^2 n)$ per insert and delete operation to keep data distributed among the nodes in a consistent way without exceeding the capacity of a node [2, 4, 17]. This bound holds as long as at most a constant fraction of the storage capacity of the system is utilized at any time. Subsequently, Brodal et al. [6] 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 [11] showed that for the case that the order of the node IDs cannot be changed, any smooth, consistent data management algorithm has an amortized cost of $\Omega(\log^2 n)$ at the dispatcher 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 positions are as equally distributed among the nodes as possible. The algorithms in [4, 6, 17] fulfill these properties, and their upper bound is therefore best possible. Later, Dietz et al. [10] showed that *any* online consistent placement strategy that preserves the order of the node IDs needs an amortized work of $\Omega(\log n)$.

The lower bounds for the case of a single dispatcher also hold for the case that the nodes do the work, i.e. each node can send or receive a data block in a time unit, because the work obviously remains the same irrespective of who is doing it. Hence, the maximum injection rate any smooth, consistent online data management algorithm can sustain is $O(1/\log^2 n)$ and the maximum injection rate any consistent online data management algorithm can sustain is $O(1/\log n)$. Our algorithm can break through these lower bounds because it allows the order of the node IDs to be *changed*, i.e. node IDs can be renamed to help out overloaded nodes in another area.

Also concurrent forms of search trees have been reported [21, 25, 18] 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 data locality has also been extensively investigated in the systems community. A recent work [12] attempts to achieve locality properties for SHA-1 hashing to support range queries in Chord [30]. Work on other variants of local hashing includes Linear Hashing (LH) [32], LH* [23], and hQT* [20]. 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) [24]. More recent work at AT&T deals with clustering of all the data in the same web page together on a disk [29]. Other relevant work includes striping file systems [7, 16]. Zebra [16], for example, is a log-structured file system that stripes data across multiple disks for efficient parallel writes. GPFS [28] is a high-performance file system that stripes data across multiple disk servers each of which is a RAID array. Other work includes [3, 8, 13, 14, 15, 22, 26].

2. INSTABILITY

We start with a result that provides an instability bound for arbitrary consistent placement methods when using operational consistency.

THEOREM 2.1. *For any operationally consistent online algorithm and any $\epsilon \le 1/2$ with $\epsilon = \omega(1/n)$ there is an (adaptive) $\Theta(\epsilon)$-bounded adversary so that the algorithm is unstable in a $(1-\epsilon)$-utilized system.*

PROOF. (Sketch) Let V be the set of nodes in the system, $|V| = n$. Consider any $\epsilon \le 1/2$ with $\epsilon = \omega(1/n)$. Recall that L denotes the capacity of the nodes.

We assume that we have a $(1-\epsilon)$-utilized system. In this case, we must have $m = \epsilon n/3$ clusters $C_1, \dots, C_m$ of $1/(2\epsilon)$ nodes with consecutive IDs where each C_i is at least $(1-\frac{3}{2}\epsilon)$-utilized. This is because even if we remove $n/6$ 1-utilized nodes, the average utilization of the remaining nodes is still at least $1-\frac{6}{5}\epsilon$. Since the total utilized capacity in a cluster is at least $(1-\frac{3}{2}\epsilon)L/(2\epsilon)$ but the available capacity is only $L/(2\epsilon)$, every node in a cluster must have a load of at least $L/4$. So each node represents a well-defined range. For each of these ranges, the adversary injects 3ϵ insert requests, which means that at most $\Theta(\epsilon)$ requests are injected into any range containing L existing data items. Overall, the adversary injects $(n/6)\cdot 3\epsilon = \epsilon n/2$ insert requests, and therefore, afterwards, we have a system that is at most $(1-\epsilon)+\epsilon/2 = (1-\epsilon/2)$-utilized.

We are interested in proving a lower bound for the number of data blocks that need to move to handle the insertions without exceeding the capacity of a node. Let $V_h = \bigcup_{j=1}^{m} C_j$ denote the set of heavy nodes and $V_\ell = V \setminus V_h$ denote the set of light nodes. To simplify the proof, we consider only so-called *lazy* reassignment algorithms that first place all the inserted blocks before moving any blocks to other nodes. Since an optimal algorithm may have moved the newly inserted blocks immediately to different nodes, an optimal lazy algorithm moves at most $L\cdot\epsilon n/2$ more data blocks than an optimal algorithm.

Let $R_h \subset [0,1)$ denote the set of ranges covered by V_h and R_ℓ denote the set of ranges covered by V_ℓ. Consider the assignment of ranges to nodes in V_h at the beginning (i.e. after inserting the new data) and the end of the data movement phase. We do some modifications to the data movements to make our life easier.

First of all, at the end there must be nodes in V_ℓ that take over ranges inside R_h to cope with the overload in V_h. Suppose that there are also nodes in V_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 data blocks that have to move. Hence, we only have to consider data movements where at the end every node in V_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 $v \in V_h$ that before the data movements had the most data blocks in R be called its *owner*. Then we will rearrange the nodes in V_h so that every range R is taken over by its owner, if possible. Since every node can store at most L data blocks, for every node in V_h that was initially more than $(1-3\epsilon)$-utilized, there must 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 data blocks of which it is the owner. However, there may also be nodes in V_h that were initially less than $(1-3\epsilon)$-utilized and that do not own any range. If such a node is currently at a position not intersecting with the range of its original cluster, say C, then it aims to replace its position with any V_ℓ-node (with preference given to V_ℓ-nodes with ranges completely inside R_h) or any other displaced V_h-node with a range currently intersecting with the range of C. Such a node must always exist because otherwise the range of C would be shared by too few nodes. Thus, at the end, every V_h-node has a range that intersects with the range of its original cluster. Also, it can be ensured that for any non-owning V_h-node, there is no V_ℓ-node between its current range and its original range (if so, exchange positions).

For the nodes in V_h, this replacement strategy can only decrease the number of initial blocks that have to be moved out of them and therefore only lowers the total number of blocks that have to be moved w.r.t. nodes in V_h. But for some of the nodes in V_ℓ that are in S, this may increase the number of moved data blocks that were initially stored in them. This increase, however, is bounded by $m\cdot 2L$ (L data blocks to the lower and higher end of every R_{C_i}), which is at most $L\cdot\epsilon n/3$.

Next, we bound $|S\cap V_\ell|$. The nodes in V_h were originally at most 1-utilized. Hence, the average utilization of nodes in V_ℓ must be at least $((1-\epsilon)n - n/6)/(5n/6) = 1-6\epsilon/5$. Therefore, at most $(6\epsilon/5)\cdot(5n/6) = \epsilon n$ nodes can be moved from V_ℓ to a range inside R_h without overloading the remaining nodes in V_ℓ. Hence, at most $2m+\epsilon n < 2\epsilon n$ nodes in V_ℓ can be in S. Thus, $|S\cap V_\ell| \le 2\epsilon n$.

Since there are $\epsilon n/3$ clusters, there must be at least $\epsilon n/6$ clusters C_i with at most 12 nodes $v \in V_\ell$ with $R_v \cap R_{C_i} \neq \emptyset$ because otherwise we have a contradiction to the upper bound on $|S\cap V_\ell|$.

Let C_i be any one of these clusters. C_i can be uniquely decomposed into sequences of V_h-nodes that are separated by V_ℓ-nodes. Consider any such sequence, and suppose that it consists of the V_h-nodes $v_1,\dots,v_k$. For simplicity, we assume that all of these nodes are owners, i.e. they cover a part of their original ranges (the general case is shown in a full paper). Then we perform the following rearrangement again and again until it cannot be used any more:

Suppose that there is a node v_i that is (at most) $1-\delta$-utilized and has δL data blocks at v_{i-1} or v_{i+1} that originally belonged to v_i. Then v_i moves these δL blocks to itself. Certainly, this can only reduce the number of blocks that have to be moved.

At the end, any node v_i that is not fully utilized must store all of its blocks. Also, it may own some blocks from v_{i-1} or v_{i+1}. If so, it transforms these blocks into v_i-blocks. This does not cause the load of any node to violate the load limit of L and can only reduce the number of moved blocks. Thus, the original sequence of k nodes is perfectly partitioned by not fully utilized nodes v_i into subsequences of nodes that are completely utilized. Let $v_j,\dots,v_{j'}$ be any such subsequence in the middle (if such a subsequence exists). Because v_j has no original block in v_{j-1} and $v_{j'}$ has no original block in $v_{j'+1}$, the blocks stored in every node between v_j and $v_{j'}$ can be declared its original blocks without violating the capacity limit. Thus, for any such subsequence no movement has to be performed, and therefore only the first and the last subsequence require movements.

Since any node in a middle subsequence must have had a load of $1-3\epsilon$ before the data insertions, and the average utilization in every cluster is $1-\frac{3}{2}\epsilon$, at most half of the nodes in a cluster can be in such subsequences.

Consider now any of the other subsequences, say again, nodes $v_1,\dots,v_k$. Then we use the following strategy again and again until no improvement is possible:

Go from v_1 to v_k, and while at v_i, convert a maximum number of blocks in v_i into v_i-blocks without violating the original capacity limit of v_i. This only reduces the number of blocks that have to be moved.

At the end, we obtain a consecutive sequence of $d \leq k$ originally (i.e. before the insertion events) 1-utilized nodes, and the remaining nodes were originally $1 - 3\epsilon$-utilized. Thus, the amount of data that has to be moved between the nodes can easily be calculated to be $\Omega(d^2 \cdot \epsilon L)$ for that sequence. Since the number of originally $1-3\epsilon$-utilized nodes is limited to $1/(4\epsilon)$, summing up over all sequences in C_i gives a lower bound of

$$\Omega\left(12 \cdot \left(\frac{1}{12} \cdot \frac{1}{2\epsilon}\right)^2 \cdot \epsilon L\right) = \Omega(L/\epsilon) .$$

Hence, on average, a node in V_h has to spend $\Omega(L)$ time on moving data blocks for the system to be stable, whereas the number of requests injected for each range of size L is only $O(\epsilon L)$. Thus, there is a $\Theta(\epsilon)$-bounded adversary causing more data movements than the system can handle without exceeding the capacity of a node, and therefore causing instability. □

The question is, what kind of properties does an algorithm have to fulfill to be stable under a $\Theta(\epsilon)$-bounded adversary. A placement algorithm is called $(1+\delta)$-*faithful* if every node has a number of data blocks that is at least $\ell/(1+\delta)$ and at most $(1+\delta)\ell$ where ℓ is the average load in the system. Furthermore, a placement algorithm is called *order preserving* if it imposes a fixed order on the node IDs, 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 order preserving since nodes can never be empty and therefore can never change their position in the node ordering. However, 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.*

PROOF. Also here, the adversary works in rounds. Suppose that initially we have $(1-\epsilon)n \cdot L$ data blocks in the system. In each round, the adversary cuts the current set of data blocks 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 IDs. Now, the adversary removes ϵL blocks every L consecutive blocks in S_1 and inserts ϵL blocks every L consecutive blocks 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 data block d that is in set S_2 at round r. Suppose that d is the block with kth smallest ID in the system. Then, the node of smallest ID at which d can be at round r is k/L. On the other hand, the node of largest ID at which d can be at round $r+j$ is $n - ((1-\epsilon)n \cdot L - k + jL \cdot \epsilon n/3)/L = \epsilon n + k/L - j \cdot \epsilon n/3$. It holds that $k/L \geq \epsilon n + k/L - j \cdot \epsilon n/3$ if and only if $j \geq 3(1 + d/(\epsilon n))$. Hence, after 9 rounds, each data block 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 blocks in S_2 over 9 rounds is at least $((1-\epsilon)n/3)L \cdot 2\epsilon n$. Therefore, on average we need $\Theta((1-\epsilon)\epsilon n)$ data movements per node for the requests to maintain a consistent data placement. On the other hand, every node only receives $O(\epsilon L)$ requests within 9 rounds. Thus, the maximum rate that can be sustained is $\Theta(\epsilon/((1-\epsilon)\epsilon n)) = \Theta(1/((1-\epsilon)n))$. □

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

THEOREM 2.3. *There is an operationally consistent 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 $\alpha \cdot \epsilon / \log^2 n$-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 [11, 10], 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 [10, 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 system of n nodes of uniform capacity that is at most $(1-\epsilon)$-utilized at any time and that insert/delete requests are generated by a (λ, T)-bounded adversary. Later, we also show how to extend the algorithm to a dynamically changing set of nodes and nodes of non-uniform capacity.

We propose a protocol called *smoothing algorithm* (see Figure 1) to keep the system consistent and compact. The basic approach of the smoothing algorithm is to partition the nodes into two sets, B and I. B represents the set of *busy* nodes, and I represents the set of *idle* nodes. Data items are only stored in the busy nodes. Busy nodes are organized in *clusters* of size $\Theta(1/\epsilon)$. Each cluster consists of nodes with consecutive ranges.

The smoothing algorithm works in rounds, where each round needs a constant amount of time. In each round, every node v in a cluster C first checks whether a more balanced state of C can be obtained by moving a data block to its predecessor or successor in C. If yes, it does so. In addition, the last node in C checks whether C has too much data, and if so, it pulls a new node from I into C. Afterwards, each node in C receives a pending insert/delete request belonging to its range. Finally, C checks whether it has too many nodes or too few nodes. If it has too many nodes, it splits into two clusters, and if it has too few nodes, it merges with (or obtains nodes from) one of its neighboring clusters.

To understand the parameters in Figure 1, we need some notation. In the smoothing algorithm, the number of nodes in a cluster is kept between $s(\epsilon)/2$ and $2s(\epsilon)$, where $s(\epsilon)$ is specified later. As mentioned above, 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 $v_1^C, v_2^C, \ldots, v_{|C|}^C$.

For any node v and cluster C, let $\ell_t(v)$ denote the load (i.e. the number of data blocks) of v. Furthermore, we define the load of C, $\ell_t(C)$, and the *safe* load $\bar{L}$ as

$$\ell_t(C) = \sum_{v \in C} \ell_t(v) \quad \text{and} \quad \bar{L} = (1 - \tfrac{\epsilon}{4})L .$$

1. Each node v_k^C computes $\delta_\ell = (k-1)\bar{L} - \sum_{i=1}^{k-1} \ell_t(v_i^C)$ and $\delta_r = \sum_{i=1}^{k} \ell_t(v_i^C) - k \cdot \bar{L}$.
   (a) If $k > 1$ and $\delta_\ell \geq 1$ then {*too litte data in* $v_1^C, \ldots, v_{k-1}^C$}
      i. the block with lowest ID 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 much data in* $v_1^C, \ldots, v_k^C$}
      i. the block with highest ID in v_k^C is moved to v_{k+1}^C.
   (c) If $k = |C|$, $\delta_r \geq 1$, and $\ell_t(v_k^C) \geq \bar{L}$ then {*cluster has too much data*}
      i. an idle node is fetched from I and integrated into C with number $k+1$.
      ii. the block with highest ID in v_k^C is moved to v_{k+1}^C.
2. Each node in C receives the data block moved to it and updates its ID accordingly.
3. Each node processes the newly injected data blocks belonging to its range.
4. If $|C| = 2s(\epsilon)$ then {*cluster is too large*}
   (a) C is split into two clusters of size $s(\epsilon)$.
5. If $|C| = s(\epsilon)/2$ then {*cluster is too small*}
   (a) apply Procedure MERGE to C (Fig. 2).

Figure 1: The smoothing algorithm, which is continuously executed in every cluster C.

In the following, we assume that $0 < \epsilon < 1$, $s(\epsilon) = 6/\epsilon$, $L \geq T$, and $\lambda \leq \epsilon/90$. It should be possible to get a much better constant for λ, but we did not try to optimize it 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 data blocks in the nodes that holds as long as the system is at most $(1-\epsilon)$-utilized. For each node v_k^C, let its *ideal* load at time t be defined as

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

The next lemma bounds the maximum deviation of the load of a node from its ideal load and is therefore crucial for

Procedure MERGE:

1. Group the clusters C with $|C| = s(\epsilon)/2$ into groups of 2 or 3 consecutive clusters so that one of the following two cases is fulfilled for each group.
2. For all groups (C, C') in which C or both have a size of $s(\epsilon)/2$:
   (a) If $|C| + |C'| \leq 3s(\epsilon)/2$ then C and C' are combined into a single cluster.
   (b) Else, nodes are moved between C and C' so that both have the same number of nodes.
3. For all groups (C, C', C'') in which all clusters have a size of $s(\epsilon)/2$:
   (a) merge the group into a single cluster.

Figure 2: The cluster merging procedure.

stability. A node v_k^C of some cluster C is called an *inner* node if $k < |C|$.

LEMMA 3.2. *At any time t, it holds for every inner node v that $\ell_t(v) \in [\bar{L} - \lambda T, \bar{L} + \lambda T]$.*

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

CLAIM 3.3. *For every node v_k^C it holds that if $\ell_t(v_k^C) \geq \ell_t^*(v_k^C)$, then data movements do not increase $\ell_t(v_k^C)$, and if $\ell_t(v_k^C) \leq \ell_t^*(v_k^C)$, then data movements 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 $\sum_{i=1}^{|C|-1} \ell_t(v_i^C) \leq (|C|-1)\bar{L}$ and therefore no block is moved to v_k^C. So suppose that $k < |C|$. If a data block is moved from v_{k-1}^C to v_k^C, then also a data block is moved from v_k^C to v_{k+1}^C because $\sum_{i=1}^{k-1} \ell_t(v_i^C) \geq (k-1)\bar{L} + 1$ and therefore $\sum_{i=1}^{k} \ell_t(v_i^C) \geq k \cdot \bar{L} + 1$. If a data block is moved from v_{k+1}^C to v_k^C, then a block is also moved from v_k^C to v_{k-1}^C because $\sum_{i=1}^{k} \ell_t(v_i^C) \leq k \cdot \bar{L} - 1$ and therefore $\sum_{i=1}^{k-1} \ell_t(v_i^C) \leq (k-1)\bar{L} - 1$.

The proof for $\ell_t(v_k^C) \leq \ell_t^*(v_k^C)$ is similar. □

CLAIM 3.4. *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 load is reduced by 1.*

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

This motivates the use of the following potential function for every cluster C:

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

Claims 3.3 and 3.4 immediately imply the following result.

CLAIM 3.5. *If at the beginning of step t, $\phi_t(C) \geq 1$, then stages 1 and 2 of the smoothing algorithm decrease $\phi_t(C)$ by at least 1.*

Our aim is to show that if λ is sufficiently small, then for every time frame F of size T 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 it follows from Claim 3.3 that the deviation of an inner node v_k^C from $\ell_t^*(v_k^C)$ can only be increased by requests to v_k^C. Furthermore, a final node v only becomes an inner node if $\ell_t(v) \geq \bar{L}$ (see stage 1.(c)). Since at most λT requests can be injected per node in T steps, the number of data blocks in an inner node v can be at most λT away from its ideal load. Hence, for these nodes, it follows that at any time t, $\ell_t(v) \in [\bar{L} - \lambda T, \bar{L} + \lambda T]$. Thus, it remains to prove the following claim.

CLAIM 3.6. *If $\lambda \leq \alpha\epsilon$ for some sufficiently small constant $\alpha > 0$, then it holds: For every time frame F of size T 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 of size T 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 frame F of size T and some fixed cluster C existing at the beginning of F. If C merges with another cluster during F, we identify C with the resulting cluster. Also, if C passes nodes to its predecessor or receives nodes from its successor during F, 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 F, 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 F, then it is also true that $\phi(C_1) = \phi(C_2) = 0$ for that time. Thus, all clusters that are newly created in F 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 (i.e., B just consists of a single empty node), the induction hypothesis is correct for all clusters currently in the system when considering the earliest time frame of size T. So let us consider now any cluster C and any time frame F of size T with $\phi(C) = 0$ at the beginning of F. Then we will show that there is also a later time step in F with $\phi(C) = 0$, completing the induction.

We need several facts for the induction step. We define any node v_k^C in some cluster C with $k < |C|$ at the beginning of a round of our protocol to be an *inner* node and otherwise a *final* node.

FACT 3.7. *In the smoothing algorithm, the only case where a final node of a cluster becomes an inner node is in stage 1.(c).*

PROOF. We consider all relevant cases in which a cluster C may change:

- C splits: this will only create a new final node.
- C merges with some other cluster(s): If C is the first in a merge group (C', C'') or the first or second in a merge group (C', C'', C'''), then C must have just shrunk to a size of $s(\epsilon)/2$ (see Figure 2) and therefore the last node in C has been an inner node at the beginning of the round. Therefore, no final node becomes in inner node in this case.
- C gains some nodes from cluster C': C' must have just shrunk to a size of $s(\epsilon)/2$, so its last node has been an inner node at the beginning of the round.
- C loses some nodes to cluster C': See the previous case.

□

Now, recall that we consider a cluster C with $\phi(C) = 0$ at the beginning of F. (Notice that by our arguments above, C is a real cluster, and every cluster emerging from C in F will be covered by C.) Going through all possible cases for C in F, one can show the following fact.

FACT 3.8. *At any time during F, $|C| \leq 3s(\epsilon)$.*

PROOF. If C does not participate in MERGE during F, then C can only reach a size of $2s(\epsilon)$. Otherwise, C may undergo the following events:

- C first splits: Then C decomposes into two clusters, C_1 and C_2, of size $s(\epsilon)$ each. C_1 may participate in MERGE again. If it loses nodes, it will not participate in MERGE again during F. If it merges, its size will grow by $s(\epsilon)/2$, and afterwards, it will also not participate in MERGE again during F, because the smoothing algorithm can only cause it to grow or shrink by one node during F (if $\lambda \leq 1/(2s(\epsilon))$). C_2 cannot participate in MERGE again during F.
- C first merges with other clusters: This increases its size to at least $s(\epsilon)$ and at most $(3/2)s(\epsilon)$, which is also valid if it merges again. Apart from merging, C may only lose nodes in another participation in MERGE.
- C gains nodes from some cluster or loses nodes to some cluster: This can be handled with similar arguments as above.

In any case, $|C| \leq 3s(\epsilon)$. □

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

- Injections of requests: Since $|C| \leq 3s(\epsilon)$, the number of data blocks in C can change by at most $\lambda T \cdot 3s(\epsilon)$, and therefore $\phi(C)$ can increase by at most
$$3\lambda T \cdot s(\epsilon) \tag{2}$$
- Split events: This does not increase the nodes under consideration in $\phi(C)$, and final nodes do not count in $\phi(C)$. Thus, a split event does not increase $\phi(C)$.
- Merge events: Going through all possible cases, C can gain at most $s(\epsilon)$ nodes due to merge events. Using Fact 3.7 and the fact that every node can receive at most λT requests in F, it holds that the potential increase due to merge events is at most
$$\lambda T \cdot s(\epsilon) \tag{3}$$

- Events in which C loses nodes: This does not increase the nodes under consideration in ϕ and hence does not increase $\phi(C)$.
- Events in which C gains nodes: This only happens once in F and only if C is in a merge pair (C, C'). Since $|C'| \leq 2s(\epsilon)$, C gains at most $(3/4)s(\epsilon)$ nodes, causing a potential increase of at most

$$\lambda T \cdot (3/4)s(\epsilon) \leq \lambda T \cdot s(\epsilon) \tag{4}$$

Combining (2) to (4), the total increase of $\phi(C)$ in F can be at most

$$3\lambda T \cdot s(\epsilon) + \lambda T \cdot s(\epsilon) + \lambda T \cdot s(\epsilon) = 5\lambda T \cdot s(\epsilon)$$

This is less than $T/3$ if $\lambda < 1/(15s(\epsilon))$. Since each round of the smoothing algorithm needs (at most) 3 time steps and according to Claim 3.4, $\phi(C)$ decreases in each round as long as $\phi(C) > 0$, there must be a later time point in F where $\phi(C) = 0$, completing the proof. □

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

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

Lemma 3.9. *For all $\epsilon \leq 1$ it holds: If $s(\epsilon) \geq 6/\epsilon$ then at any point in time, $|I|$ is at least the current number of clusters in the system.*

Proof. It follows from Lemma 3.2 that the load of every inner node is at least $\bar{L} - \lambda T$ at any time, and $\bar{L} - \lambda T \geq (1 - \epsilon/3)L$ for our choices of $\bar{L}$, L, and λ. Because every cluster has a size of more than $s(\epsilon)/2$ at any time, the average number of blocks in a node in B is at least

$$\frac{s(\epsilon)/2 \cdot (\bar{L} - \lambda T)}{s(\epsilon)/2 + 1} \geq \left(1 - \frac{\epsilon}{3}\right)^2 L \geq \left(1 - \frac{2\epsilon}{3}\right) L$$

Because the system is at most $(1-\epsilon)$-utilized, it follows that

$$|B| \leq \frac{(1-\epsilon)L \cdot n}{(1 - 2\epsilon/3)L} \leq \left(1 - \frac{\epsilon}{3}\right) n \, .$$

Thus, $|I| \geq (\epsilon/3) \cdot n$. Since the number of clusters is at most $n/(s(\epsilon)/2) \leq (\epsilon/3) \cdot n$ at any time, the lemma follows. □

Combining the lemmata yields the following theorem.

Theorem 3.10. *For any $0 < \epsilon < 1$, the smoothing algorithm is maintaining a consistent and $(1 + \epsilon/2)$-compact mapping under any $\epsilon/90$-bounded adversary as long as the system is at most $(1 - \epsilon)$-utilized at any point in time.*

Proof. The stability follows from the previous lemmata. Hence, it remains to bound the compactness. According to Lemma 3.2 every inner node has a load of at least $\bar{L} - \lambda T \geq (1 - \epsilon/3)L$ at any time. In the worst case, the final node has load 0. Hence, k consecutive blocks are spread out over at most

$$\frac{k}{(1 - \epsilon/3)L} + 3 \leq \left(1 + \frac{\epsilon}{2}\right) \cdot \frac{k}{L} + 3$$

nodes. □

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 data in decreasing order to w. While these movements are happening, v will still accept all data from its predecessor in C, but w will receive all data from v's successor. Once v is empty, it can leave the system.

From Lemma 3.9 and Theorem 3.10 it follows:

Theorem 3.11. *As long as the system is at most $(1 - \epsilon)$-utilized and the rate of node departures in a cluster is at most $\rho \cdot \epsilon/L$ for some constant $\rho > 0$, the smoothing algorithm achieves the same performance as in Theorem 3.10.*

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 change over time. Let $L_t(v)$ be the capacity of v at time t. Then we can show the following result:

Theorem 3.12. *As long as the system is at most $(1 - \epsilon)$-utilized and $C_t(v)$ 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 $1 + \epsilon$-compactness does not make sense in a heterogeneous environment because the number of nodes storing a range of objects depends on their capacities.

4. DECENTRALIZED STORAGE

Notice that the smoothing algorithm does not require global parameters and therefore can be turned into an algorithm for the distributed setting. For this, we combine various overlay networks – one for managing the busy nodes, one for managing the idle nodes, and one for connecting the busy nodes with idle nodes to turn idle nodes into busy node or vice versa. This represents the first alternative to the DHT-based peer-to-peer systems that can use the *real* names of the objects instead of their hashed names for a consistent mapping while preserving a compact 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, such as:

- *How to organize busy and idle nodes in a distributed setting?* Busy and idle nodes have to be interconnected to allow fast access to busy and idle nodes and to move nodes between the two sets.
- *How to break symmetry?* Merge candidates need a mechanism to decide with whom to merge. Clusters have to coordinate their selection of idle nodes.

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:

- $G_B = (B, E_B)$: This graph consists of a cycle in which the busy nodes are ordered according to their IDs (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 our system is composed of works.

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_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_B

Recall that $G_B = (B, E_B)$ consists of a cycle in which the busy nodes are ordered according to their IDs, 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.

4.4 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 $c : F \rightarrow \{black, white\}$. 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 $v \in F$ 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 $v \in F$ that has a white successor w, v, w, and the successor of w form a group.
- For every white node $v \in F$ that has two black successors, v and its two successors form a group.

The following lemma is not hard to check:

LEMMA 4.1. *The coloring scheme breaks symmetry in a unique way so that every node belongs to exactly one group of size 2 or 3.*

Once these groups are established, merge groups can be quickly built by either using these groups directly or (in case of item 2 in Figure 2) by coordinating with the next group to the right.

4.5 The graph G_I

For G_I we can use any DHT-based overlay network, such as Chord [30], 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.

4.6 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 $\Theta(1/\epsilon)$ as implied by our results is unacceptable for single disk management.

Acknowledgements

We would like to thank the anonymous referees for helpful comments.

6. 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] A. Andersson and T.W. Lai. Fast updating of well-balanced trees. In *Proc. of the 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 111–121, 1990.

[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 IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 399–409, 2000.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proc. of the 1st Biennial Conference on Innovative Data Systems Research*, 2003.

[13] S. Hanke, T. Ottmann, and E. Soisalon-Soininen. Relaxed balanced red-black trees. In *3rd Italian Conference on Algorithms and Complexity CIAC*, pages 193–204, 1997.

[14] 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.

[15] 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.

[16] J.H. Hartman and J.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.

[17] A. Itai, A.G. Konheim, and M. Rodeh. A sparse table implementation of sorted sets. In *Proc. of 8th Int. Colloquium on Automata, Languages and Programming (ICALP)*, 1981.

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

[19] 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.

[20] J.S. Karlsson. HQT*: A scalable distributed data structure for high-performance spatial accesses. In *FODO* 1998.

[21] P. Krishna and T. Johnson. Highly scalable data balanced distributed B-trees. Technical Report 95-015, University of Florida, Dept. of CISE, 1995. Available at ftp.cis.ufl.edu:cis/tech-reports.

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

[23] W. Litwin, M. Neimat, G. Levy, S. Ndiaye, and T. Seck. LH*: A high-availability and high-security scalable distributed data structure. In *IEEE Research Issues in Data Engineering (RIDE-97)*, 1997.

[24] 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.

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

[26] K. Ramamohanarao and E.P. Harris. Effective clustering of records for fast query processing. In *1st Int. Symposium on Cooperative Database Systems for Advanced Applications CODAS*, pages 516–525, 1996.

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

[28] 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.

[29] E. Shriver, E. Gabber, L. Huang, and C.A. Stein. Storage management for web proxies. In *USENIX '01*, pages 203–216, 2001.

[30] 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.

[31] 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.

[32] D. Schneider W. Litwin, M. Neimat. Linear hashing for distributed files. In *ACM SIGMOD Conference*, pages 327–335, 1993.

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

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