Cost-Oblivious Storage Reallocation

Michael A. Bender
Computer Science,
Stony Brook University
and Tokutek, Inc.
USA
bender@cs.stonybrook.edu

Martin Farach-Colton
Computer Science,
Rutgers University
and Tokutek, Inc.
USA
farach@cs.rutgers.edu

Sándor P. Fekete
Computer Science
TU Braunschweig
Germany
s.fekete@tu-bs.de

Jeremy T. Fineman
Computer Science
Georgetown University
USA
jfineman@cs.georgetown.edu

Seth Gilbert
Computer Science
National University of Singapore
Singapore
seth.gilbert@comp.nus.edu.sg

ABSTRACT

Databases allocate and free blocks of storage on disk. Freed blocks introduce holes where no data is stored. Allocation systems attempt to reuse such deallocated regions in order to minimize the footprint on disk. When previously allocated blocks cannot be moved, this problem is called the ***memory allocation*** problem. It is known to have a logarithmic overhead in the footprint size.

This paper defines the ***storage reallocation*** problem, where previously allocated blocks can be moved, or ***reallocated***, but at some cost. This cost is determined by the allocation/reallocation ***cost function***. The algorithms presented here are ***cost oblivious***, in that they work for a broad and reasonable class of cost functions, even when they do not know what the cost function actually is.

The objective is to minimize the storage footprint, that is, the largest memory address containing an allocated object, while simultaneously minimizing the reallocation costs. This paper gives asymptotically optimal algorithms for storage reallocation, in which the storage footprint is at most $(1+\varepsilon)$ times optimal, and the reallocation cost is at most $O((1/\varepsilon)\log(1/\varepsilon))$ times the original allocation cost, which is asymptotically optimal for constant ε. The algorithms are ***cost oblivious***, which means they achieve these bounds with no knowledge of the allocation/reallocation cost function, as long as the cost function is subadditive.

This research was supported in part by NSF grants IIS 1247726, CCF 1217708, CCF 1114809, CCF 0937822, IIS 1247750, CCF 1114930, CCF 1218188, by DFG grant FE407/17-1, and by Singapore NUS FRC R-252-000-443-133.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PODS'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2375-8/14/06 ...$15.00.
http://dx.doi.org/10.1145/2594538.2594548.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sequencing and scheduling*

Keywords

Reallocation, storage allocation, scheduling, physical layout, cost oblivious

1. INTRODUCTION

Databases, and more generally storage systems, need to allocate and free blocks of storage on disk. Freed data introduces holes where no data is stored. Allocation systems attempt to reuse such deallocated regions in order to minimize the footprint on disk.

The problem of allocating and freeing storage is well studied as the ***memory allocation*** problem. In this formulation, allocated objects cannot be moved. The ***competitive ratio*** is defined as the maximum possible ratio of the allocated memory (largest allocated memory address) to the sum of the sizes of allocated segments [34–36]. The lower bound on the competitive ratio is roughly logarithmic in the number of requests and in the ratio of the largest to smallest request [35].

The logarithmic lower bound renders traditional memory allocation too blunt a theoretical tool for understanding storage in many settings. Furthermore, as we show, this lower bound is a consequence of the requirement that allocated storage cannot be moved. But many actual systems have no such restriction. Storage systems typically define a mapping from a logical address space to a physical address space, and this layer of indirection allows storage to be reallocated, at a cost.

Storage reallocation. This paper generalizes memory allocation by allowing the allocator to move previously allocated objects. We call this generalization ***storage reallocation***. Storage reallocation can take place on any physical medium for allocating objects, e.g., main memory, rotating disks, or flash memory.

Thus, garbage collection [31] is a type of in-core storage reallocation. Our own interest in memory reallocation stems for our experience in building the TokuDB [45] and TokuMX [46] databases, in which memory segments are accessed via a so-called "block translation layer," which translates between the block name, which is immutable, and the block address in storage, which may change.

Cost-oblivious storage reallocation. An algorithm for storage reallocation must contend with the tradeoff between storage footprint size and the amount (and cost) of reallocation. It should come as no surprise that a storage reallocator that is designed for main memory is unlikely to work well if the objects are allocated on a rotating device instead—and vice versa. This is because the cost model depends on where the objects are stored.

The question is therefore how to model the cost of reallocating memory objects. Faithful cost models are hard to come by because the memory hierarchy has a hard-to-quantify impact on run time. In RAM, moving an object is roughly proportional to w, the object size. On disk, moving a small object may be dominated by the seek time, while moving a large object may be dominated by the disk bandwidth. In both cases, there are cache effects, both in memory and in storage and in their interaction. The performance characteristics for each aspect of memory vary by brand and model.

Rather than model these complex interactions, this paper specifies a class of cost functions that subsumes them. We give universal reallocators, independent of the particulars of the reallocation cost. We say that a universal reallocator is ***cost oblivious*** with respect to a class of cost functions if it is optimal or near optimal for all cost functions in that class. Our reallocation algorithms are cost oblivious with respect to the class of cost functions that are ***subadditive***, monotonically increasing functions of the object size.[1]

To summarize, in storage reallocation, there is an online sequence of insert (malloc) and delete (free) requests. Objects are allocated to locations in an arbitrarily large array (address space). The cost of allocating or moving (reallocating) a size-w object is some unknown (monotonically increasing) subadditive function $f(w)$.

Storage reallocation is thus a bicriteria optimization problem. The first objective is to store objects in an array so that the largest allocated memory address—-which we call the ***footprint***—is approximately minimized. The second objective is to minimize the amortized reallocation cost per new request. In this paper, we consider the problem of minimizing the amortized reallocation cost, while using a memory footprint that is at most a constant factor larger than optimal.

Storage reallocation in a database. Databases have many moving parts, and any system that changes the way that storage is allocated needs to interact gracefully with the other requirements of the storage system.

A common constraint in storage (re)allocation is that updates be ***nonoverlapping***, i.e., when an object is moved, its new location must be disjoint from its old location. In databases, object writes are not atomic, so nonoverlapping reallocation is necessary for durability[2].

The nonoverlapping constraint is only part of the mechanism for durability. Another consideration is that when an object is moved, the translation table between logical and physical addresses is updated. It is then written to disk during a ***checkpoint***. Only then are blocks that have been freed since the last checkpoint available for reuse. Therefore, the allocator may not write to a location that has been freed since the last checkpoint.

Finally, new memory requests arrive at unpredictable times. It is undesirable for an allocation request to block on a long sequence of reallocations, even if the average throughput is high. A good reallocation algorithm provides some guarantee on the worst-case cost of individual operations, while still maintaining (near) optimal throughput.

Formalization. An ***online execution*** is a sequence of requests of the form $\langle \textsc{InsertObject}, name, length\rangle$ and $\langle \textsc{DeleteObject}, name\rangle$. After each request, the reallocator outputs an allocation for the objects in the system. We say that an object is ***active*** at time t if it has been inserted by one of the first t requests, but not deleted by the end of request t. (Note that an object being deleted remains active until the reallocator completes the delete request.)

If $\mathcal{S}$ and $\mathcal{S}'$ are the allocations immediately before and after request p, then the ***reallocation cost of p*** is the sum of the reallocation costs of all objects moved between $\mathcal{S}$ and $\mathcal{S}'$.

A reallocator A is ***(f, a, b)-competitive*** for cost function $f()$, if (1) the footprint size is always optimized within an a-factor of optimal, and (2) the reallocation cost is at most b times the sum of the allocation costs of every object inserted thus far (including those that have subsequently been deleted). Since every object must be allocated at least once, the cost of such a reallocator is clearly within a factor of b of optimal.

Let $\mathcal{C}$ be a set of cost functions. A reallocation algorithm A is ***cost oblivious*** if it does not depend on $f()$.[3] A cost-oblivious reallocator A is ***$(\mathcal{C}, a, b)$-competitive*** if it is (f, a, b)-competitive for every $f \in \mathcal{C}$; we abbreviate to ***(a, b)-competitive*** if the set $\mathcal{C}$ is unambiguous. We say that A is ***optimally cost-oblivious*** if it is ***(a, b)-competitive*** for some $a, b \in O(1)$. Let $\mathcal{F}_{sa}$ be the class of subadditive functions.

Results. Our reallocation algorithms are tunable to achieve an arbitrarily good competitive ratio $1 + \varepsilon$, $(0 < \varepsilon \leq 1/2)$ with respect to the footprint size. All objects have integral length, and Δ denotes the length of the longest object. We establish the following:

- We give a cost-oblivious algorithm for storage reallocation that is $(\mathcal{F}_{sa}, 1 + \varepsilon, O((1/\varepsilon)\log(1/\varepsilon))$-competitive.

[1] A (monotonically increasing) function $f(x)$ is ***subadditive***, if $f(x + y) \leq f(x) + f(y)$ for any positive x and y. (All monotonically increasing concave functions are subadditive.) The restriction to subadditivity is not severe. While there exist corner cases where a storage system is temporarily superadditive, most mechanisms employed by operating systems, such as prefetching for latency hiding, rely on the subadditivity of costs.

[2] This is also relevant in other contexts: In SSDs, the nonoverlapping constraint is enforced by the hardware, because memory locations must be erased between writes. In FPGAs, satisfying this constraint allows interruption-free reallocations of modules [23].

[3] This means not only that $f()$ is not a parameter to algorithm A, but also A learns nothing about $f()$ as A executes.

This allocator is amortized in the sense that it might reallocate every existing object between servicing two requests.

- As a corollary, we give a defragmenter that is cost oblivious with respect to $\mathcal{F}_{sa}$. The defragmenter takes as input a comparison function, a set of objects having total length V and consuming space $(1+\varepsilon)V$. The defragmenter sorts the objects using $(1+\varepsilon)V+\Delta$ working space, moving each object $O((1/\varepsilon)\log(1/\varepsilon))$ times.
- We extend the storage reallocator to support checkpointing. With an additional $O(\Delta)$ space, we guarantee that each operation completes within $O(1/\varepsilon)$ checkpoints. (An immediate consequence is that all reallocations are non-overlapping.)
- We also partially deamortize the storage reallocator so that the worst-case reallocation cost (and therefore the worst-case time blocking for a new size-w allocation) is reduced to $O((1/\varepsilon)wf(1)+f(\Delta))$.

Related work. We now briefly review the related work.
Dynamic memory allocation. There is an extensive literature on memory allocation [34–36, 38–40, 52] where object reallocation is disallowed. There are upper and lower bounds on the competitive ratio of the memory footprint that are roughly logarithmic in the number of requests and in the ratio of the largest to smallest request. These papers generally analyze traditional strategies such as Best Fit, First Fit, and the Buddy System [33], but also propose alternatives. Traditional memory-allocation strategies often have analogs in bin-packing [17–20, 25], but an enumeration of such results lies beyond the scope of this paper.

Memory allocation where reallocation *is allowed* appears often in the literature on garbage collection [31]. There is a long and important line of literature studying dynamic memory allocation with differing compaction mechanisms, exploring the time/space trade-off between the amount of compaction performed and the total memory used. Ting [44] develops a mathematical model for examining this trade-off for different compaction algorithms; Błażewicz et al. [14] develop a "partial" compaction algorithm for segments of two different sizes that reallocates only a limited number of segments per compactions. More recently Bendersky and Petrank [13] and Cohen and Petrank [21] have more fully explored the trade-offs inherent in partial compaction.

These papers on dynamic memory allocation with compaction are instances of storage reallocation, as addressed in this paper, where the reallocation cost is (typically) linear: the cost of compaction is directly proportional to the amount of memory that is moved. (These papers often address other problems that arise in garbage collection, such as how to update pointers to memory that has moved.) For example, Bendersky and Petrank [13] show that when the cost function is linear, one can achieve constant amortized reallocation cost with memory size that is within a constant-factor of optimal.

In this paper, by contrast, we focus on cost-oblivious algorithms that tolerate the range of cost functions found in external storage systems. Cost obliviousness bears a passing resemblance to similar notions in the memory hierarchy, particularly the cache-oblivious/ideal-cache [24, 37], hierarchical memory [1], and cache-adaptive [8] models. With the exception of the underlying paging [43], work in these models is about writing algorithms that are memory-hierarchy universal rather than analyzing resource allocation.
Other related work. We note that storage reallocation has other applications besides databases. For example, Fekete et al. [23] address the storage reallocation problem in the context of FPGAs, and Bender et al. [10] give (not cost-oblivious) algorithms for constant reallocation cost.

Sparse table data structures [6,7,11,12,16,28,29,32,49–51] also solve the storage reallocation problem and are easily adapted to deal with different-sized objects and linear reallocation cost. But they do so while maintaining the constraint that the object order does not change, which makes the problem harder and the reallocation cost correspondingly larger.
Scheduling/planning interpretation. The storage reallocation problem can be viewed as a reallocation problem in scheduling/planning. In this interpretation, we have an online sequence of requests to insert a new job j into the schedule or to delete an exiting job j. Each job has a length w_j and the rescheduling cost is $f(w_j)$. The goal is to maintain a uniprocessor schedule that (approximately) minimizes the makespan (latest completion time of any job), while simultaneously guaranteeing the overall reallocation cost is approximated minimized. We can abbreviate this scheduling problem as $1|f(w)$ *realloc* $|C_{\max}$, generalizing standard scheduling notation [26]. The goal is actually not to *run* the schedule, but rather to plan a schedule subject to an online sequence of requests.

We thus (briefly) review related work in scheduling and combinatorial optimization. Several papers explore related notions of scheduling reallocation (although to the best of our knowledge, not cost-universal scheduling reallocation). Bender et al. [9] study reallocation scheduling with unit-length jobs having release times and deadlines. Their reallocator maintains a feasible multiprocessor schedule while servicing inserts and deletes.

In the area of robust optimization, the goal is to develop solutions for combinatorial optimization problems that are (near) optimal, and that can be readily updated if the instance changes. In this context, many papers have looked at the problem of minimizing reallocation costs for specific optimization problems (e.g., [23,27,47]). For example, Davis et al. [22] study a reallocation problem, where an allocator divides resources among a set of users, updating the allocation as the users' constraints change. The goal is to minimize the number of changes to the allocation. As another example, Sanders et al. [41] look at the problem of assigning jobs to processors, minimizing the reallocation as new jobs arrive. Jansen et al. [30] look at robust algorithms for online bin packing that minimize migration costs. See Verschae [48] for more details on robust optimization.

Shachnai et al. [42] explore a slightly different notion of reallocation for combinatorial problems. Given an input, an optimal solution for that input, and a modified version of the input, they develop algorithms that find the minimum-cost modification of the optimal solution to the modified input. A difference between their setting and ours is that we measure the ratio of reallocation cost to allocation cost, whereas they measure the ratio of the actual transition cost to the optimal transition cost resulting in a good solution. Also, we focus on a sequence of changes, which means we amortize the expensive changes against a sequence of updates.

There also exist reoptimization problems, which address the goal of minimizing the computational cost for incrementally updating the schedule [2–5, 15]. By contrast, in reallocation, we focus on the cost of reallocating resources rather than the computational cost of generating the allocation.

2. FOOTPRINT MINIMIZATION

In this section we give a cost-oblivious algorithm for footprint minimization in storage reallocation. The footprint always has size at most $(1+\varepsilon)V_t$, where V_t denotes the ***volume***, or total size, of all allocated objects at time t, i.e., of the active objects after the t^{th} operation completes. A size-w object has an amortized reallocation cost of $O\left(f(w)\cdot(1/\varepsilon)\log(1/\varepsilon)\right)$, where $f(w)$ is the (unknown) cost for allocating an object of size w.

THEOREM 1. *For any constant ε with $0<\varepsilon\le 1/2$, there exists a cost-oblivious storage-reallocation algorithm that is $(1+\varepsilon, O\left((1/\varepsilon)\log(1/\varepsilon)\right))$-competitive with respect to $\mathcal{F}_{sa}$, the class of subadditive cost functions.*

Thus, the storage reallocation algorithm is asymptotically optimal for any constant $\varepsilon < 1/2$.

Intuition and cost-function-specific algorithms. We begin by considering some simple cases where the cost function is known in advance. First suppose that the reallocation cost is linear in the object size, i.e., $f(w) = w$. A simple logging-and-compressing strategy attains a $(2,2)$-competitive algorithm for linear cost functions. Specifically, allocate objects from left to right. Upon a deletion, leave a hole where the object used to be. Whenever a deallocation causes the footprint to reach $2V_t$, remove all holes by compacting. The cost to reallocate the entire volume V_t is paid for by the V_t's worth of elements that were deallocated since the last compaction.

Logging and compressing does not work well for constant reallocation cost, i.e., $f(w) = 1$. To see why, suppose the deleted objects have size Δ, and the reallocated elements have size 1. We may need to spend amortized $\Theta(\Delta)$ reallocation cost per deletion.

There do exist good reallocators for constant reallocation cost [10]. Conceptually, round the sizes of object up to the next power of 2, to form ***size classes***, where objects have size 2^i for $i = 1,\ldots,\log\Delta$. Now group the objects by increasing size. Between the ith and $(i+1)$st size class, is either a gap of size 2^i or no gap. To insert an object of size 2^i, put the object into the gap after the ith size class, if one exists, or displace a larger object to make space. Then recursively reinsert the larger object. The amortized reallocation cost is $O(1)$, because the costs per unit volume to displace the recursively larger objects form a geometric series.

It can be shown, however, that with linear reallocation cost this strategy is only $(2, O(\log\Delta))$-competitive.

This section gives a single algorithm that works for $f(w) = w$, $f(w) = 1$, and all other subadditive cost functions. The algorithm keeps the objects partially sorted by size. Since the cost function is subadditive, small objects are the most expensive to move per unit size. We therefore want to guarantee that when an object is inserted or deleted, it can only trigger the movement of larger (less expensive per unit size) objects. Specifically, small objects with total volume W will be able to cause the movements of big objects with total volume $O(W)$, but not the other way around. At the same time, we need to avoid cascading reinserts, which can happen with the algorithm for unit cost described above.

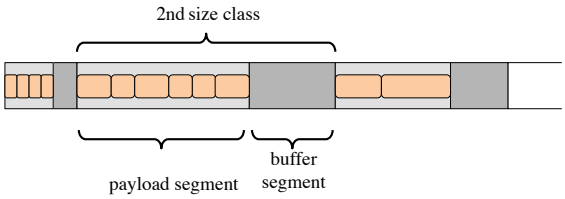

Figure 1: The layout of the data structure when the buffer segments are empty, with $\varepsilon' = 1/2$. The light-gray are the payload segments, and the dark-gray areas are the buffer segments. The orange rectangles are objects currently in the data structure.

Overview and invariants. Objects are categorized into $\lfloor\lg\Delta\rfloor+1$ ***size classes***, where objects have size at least 1 and at most Δ. The value of Δ need not be set in advance. The ith size class contains objects of size w where $2^{i-1}\le w < 2^i$. For size class i, $V_t(i)$ denotes the ***volume*** (total size) of all objects active at time t in size class i. If t is understood, we use $V(i)$.

The array (address space) is divided into $\lfloor\lg\Delta\rfloor+1$ regions, as illustrated in Figure 1. The ith region is dedicated to the ith size class and comprises two subregions, a ***payload segment*** followed by a ***buffer segment***. The ith payload segment contains only objects belonging to the ith size class, whereas the ith buffer segment may contain objects that are in the ith size class or earlier (smaller) size classes.

Whenever (potentially large) reallocations are taking place, we utilize one last region at the end of the array, the ***overflow segment***. The overflow segment is used for temporarily rearranging the objects, as described later.

INVARIANT 2. *The following properties are maintained throughout the execution of the algorithm:*

1. *The ith region ($i = 1,\ldots,\lfloor\lg\Delta\rfloor+1$) comprises the ith payload and ith buffer segment.*
2. *The $(\lfloor\lg\Delta\rfloor+2)$nd region, the overflow segment, stores elements temporarily during reallocation.*
3. *The ith payload segment only stores elements from the ith size class.*
4. *The ith buffer segment only stores elements from size classes $\ell\le i$.*

Allocating and deallocating. When a new size-w object that belongs to an existing size class i is allocated, it is stored at the end of the earliest buffer $j\ge i$ that has sufficient unoccupied space. (Recall that this object cannot be inserted into any buffer in a segment $< i$.)

When there is not enough available space in any of these buffers, a ***buffer flush*** operation is triggered, after which the object is inserted.[4] During a buffer flush, all objects in

[4]As described in this section, $V_t(i)$ immediately increases to count the new object, but the object is not yet placed in the array. Next, the flush occurs, and finally the new object is placed in the array. Our extension in Section 3 places the

some suffix of buffers get moved to their proper payload segments and the segment and region boundaries get redefined.

If the new size-w object belongs to a larger size class than any other active object, then we instead create a new payload segment and buffer segment for the new size class located immediately after the last size class's segment, increasing the total space used by at most an additive $w + \varepsilon' w$, for some constant ε'. (The overflow segment is empty, as it is only used during a buffer flush, and hence is implicitly moved to after the new size class.)

When an active size-w object is deleted, it leaves a hole until the next buffer flush occurs. A dummy deletion request is added to the buffer and forced to consume w space. This buffered dummy request is not freed until the next buffer flush. Since both inserting and deleting a job of size w reduces the space in the buffer by w, we can analyze insertions and deletions as a single case.

INVARIANT 3. *The overflow segment is empty except during buffer flush operations.*

INVARIANT 4. *When a flush of the ith buffer segment occurs at time t, the object and segment boundaries move in such a way as to guarantee that:*

1. *the space occupied by the ith payload segment after the buffer flush completes is exactly $V_t(i)$, and*
2. *the space occupied by the ith buffer segment after the buffer flush completes is $\lfloor \varepsilon' V_t(i) \rfloor$, for $\varepsilon' = \Theta(\varepsilon)$.*

Immediately following this flush, the size-$\lfloor \varepsilon' V_t(i) \rfloor$ buffer contains no objects. (This space is reserved to accommodate future insertions and deletions.)

Buffer flush. A buffer flush updates the segment boundaries in a suffix of regions, moving all objects to their proper payload segments, and leaving all buffer segments empty to accommodate future insertions.

To execute a buffer flush, first determine the ***boundary size class*** b and then flush all buffers for size classes $i \geq b$. The value b is defined as the maximum value such that all objects in buffers $i \geq b$ and the object being inserted/deleted belong to size classes $\geq b$. To determine b, iterate from the largest to the smallest region, examining every object in the region's buffer. If any object belongs to a size class $s < b$, then update b with the size class s. This continues until reaching size class b, where no object from a smaller size class has been encountered.

To flush the size classes $i \geq b$ at time t, first calculate $V_t(i)$ for all $i \geq b$. The goal is to redistribute these size classes to take space at most $S = (1 + \varepsilon') \sum_{i \geq b} V_t(i)$, i.e., space $V_t(i)$ for the ith payload segment and $\lfloor \varepsilon' V_t(i) \rfloor$ for the ith buffer, while moving all objects from buffers into payload segments.

A flush can be implemented to include at most two moves per object in the flushed size classes.

1. First identify the new array suffix of size S to accommodate payload and buffer segments. Temporarily move all objects from buffer segments to empty space immediately after this suffix (or after the current suffix, if the current suffix is longer due to deletes), removing any dummy delete records from buffers. These objects make up the overflow segment. This first step increases space usage by at most $\sum_{i \geq b} \varepsilon' V_t(i)$.

object before performing the flush; this extension requires an additive Δ working space during the flush procedure.

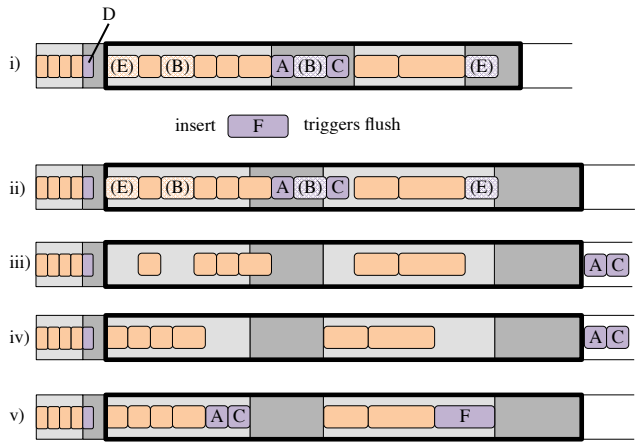

Figure 2: Example of a flush, starting from Figure 1. The lavender rectangles are updates to the data structure, with parentheses and light shading denoting a delete or delete record. (i) The state after insert A, delete B, insert C, insert D, and delete E in that order. (ii)–(v) show a flush that occurs when inserting F. The heavy outline shows the region affected by the flush, i.e., size classes 2 and 3. (ii) The new boundaries for the 2nd and 3rd size class. (iii) The state after moving buffered objects out of the way and dropping deleted objects. (iv) The state after rearranging the payload segments. (v) The state after putting all buffered objects to their proper locations. Observe that for the flushed classes, the buffers are now empty.

2. Next, iterate over payload segments from smallest to largest, moving objects as early as possible, thus removing any gaps left by deleted objects or emptied buffers. At the end of this step, all the objects are packed as far left as possible with no gaps, beginning at the start of region b.
3. Then, iterate over payload segments from largest to smallest, moving each object to its final destination in the redistributed array (which is no earlier than its current location). The final destination can be determined by looking at the values $\{V_t(i)\}$; this step reintroduces gaps to accommodate any not-yet-placed objects in the overflow segment and the empty size-$\lfloor \varepsilon' V_t(i) \rfloor$ buffers.
4. Finally, iterate over all objects in the overflow segment, placing them in their final destination at the end of the appropriate payload segments.

Analysis. The proof of Theorem 1 follows from Lemmas 5 and 6 given below, by fixing $\varepsilon' = \Theta(\varepsilon)$ appropriately. Lemma 5 states that the space used is $1 + O(\varepsilon')$ times the optimal space usage. Lemma 6 states that the reallocation cost is no worse than $O((1/\varepsilon') \lg(1/\varepsilon'))$ times the optimal reallocation cost.

LEMMA 5. *After processing allocation/deallocation requests $1, 2, \ldots, t$, the space used by the storage-reallocation algorithm is $(1 + O(\varepsilon'))V$, where $V = \sum_i V_t(i)$.*

PROOF. Let $f_i \le t$ be the previous time the ith buffer was flushed. The space used by the buffers and payload segments is at most $(1+\varepsilon')\sum_i V_{f_i}(i)$ by construction, and it may grow to $(1+O(\varepsilon'))\sum_i V_{f_i}(i)$ during the present buffer flush.

To prove the lemma, we need only to show that $\sum_i V_{f_i}(i)$ is not much different from $\sum_i V_t(i)$. The difference is reflected precisely by those objects in buffers (including delete records), which correspond to at most an $\varepsilon'\sum_i V_{f_i}(i)$ total volume of objects. Thus, we have $\left|\sum_i V_t(i) - \sum_i V_{f_i}(i)\right| \le \varepsilon'\sum_i V_{f_i}(i)$.

The worst case ratio overhead occurs when all buffered objects are deletions, in which case we have $\sum_i V_t(i) \ge (1-\varepsilon')\sum_i V_{f_i}(i)$.

We thus have at most $(1+O(\varepsilon'))\sum_i V_{f_i}(i)$ space storing at least $(1-\varepsilon')\sum_i V_{f_i}(i)$ active objects. Observing that $(1+O(\varepsilon'))/(1-\varepsilon') = 1+O(\varepsilon')$ for $\varepsilon' \le 1/2$ completes the proof. □

LEMMA 6. *For subadditive cost functions f, the amortized cost of inserting or deleting an object of size w is $O\left(f(w)\cdot(1/\varepsilon)\lg(1/\varepsilon)\right)$.*

PROOF. Consider a buffer-flush operation, and let b be the boundary size class (i.e., all size classes $i \ge b$ have their buffers flushed). There are two cases:

Case 1: The ith buffer contains $\Omega(\varepsilon' V(i))$ volume of objects, for concreteness, say at least $\varepsilon' V(i)/2$ volume.

Case 2: The ith buffer is ***underfull***, i.e., contains less than $\varepsilon' V(i)/2$ volume. Case 2 occurs because of various kinds of roundoff. Specifically, $\varepsilon' V(i)$ may not be large enough to accommodate even one object in size-class i.

We first deal with Case 1. We need to show that the initial allocation cost of objects in the buffer is sufficient to pay for the reallocation cost of objects in the payload segment. Since the objects in the buffer belong to the ith or earlier size classes, they can each have size at most 2^i. For monotonic subadditive functions, $f(x)/x$ (the cost per unit) is nonincreasing, so the cost of allocating the objects in the buffer is at least $\Omega((f(2^i)/2^i)(\varepsilon' V(i)))$. Since f is subadditive, we have $f(2^i) = O(f(2^{i-1}))$, which implies that this buffer cost is at least $\Omega((f(2^{i-1})/2^{i-1})(\varepsilon' V(i)))$. If we charge each buffered object for $\Theta(1/\varepsilon')$ reallocations, it follows that we can afford the total cost of at most $(f(2^{i-1})/2^{i-1})V(i)$ to reallocate all objects in the payload segment. Observing that each object is only flushed once (after an object moves to the payload segment, it stays there until deallocated) completes this case.

We next deal with Case 2, where buffer i is underfull. Buffer i participates in the buffer-flush operation because some object belonging to size class $i' \le i$ is placed in some buffer for size class $j > i$. We charge that object for flushing any underfull buffers between size class i' and size class j. (There may be many such objects, which only decreases the cost per object—we pessimistically charge only a single object.)

The main question, then, is: what is the maximum reallocation cost due to underfull buffers that can be charged against an object in size-class i'? Size-class i may only be charged against the object if $2^{i'} > \varepsilon' V(i)/2$. This implies that $V(i) = O(2^{i'}/\varepsilon')$, and hence the cost of moving every object in size-class i is at most $O(1/\varepsilon')$ times the cost of allocating a single object in size-class i'. Because each successive size class doubles in size, and a size class only has a payload segment (and buffer segment) if there is at least one object in the size class, only the $O(\lg(1/\varepsilon'))$ nearest size classes may satisfy $2^{i'} > \varepsilon' V(i)/2$—in particular, $\varepsilon' 2^{i'+\lceil\lg(1/\varepsilon')\rceil+1}/2 \ge 2^{i'}$, and hence if any larger size-class is underfull, it will not be "skipped over" by an object in size-class i'.

To conclude, buffered objects in size-class i' may be charged for $O(1/\varepsilon')$ reallocations in $O(\lg(1/\varepsilon'))$ different size classes, for a total cost of $O((1/\varepsilon')\lg(1/\varepsilon'))$ allocations. □

Corollary: Defragmenting/Sorting

A corollary of cost-oblivious storage reallocation is a cost-oblivious defragmentation algorithm, i.e., a cost-oblivious algorithm for sorting the objects while simultaneously respecting constraints on the space usage.

We first compare with naïve defragmentation. If $2V$ working space is allowed, then defragmentation is trivial with two movements per object. First crunch the objects into the rightmost V space, using one move per object. Then place each object directly in its final destination within the leftmost V region of space.

The following theorem shows that defragmentation is possible even using $(1+\varepsilon)V+\Delta$ space by applying cost-oblivious storage reallocation as a black box.

THEOREM 7. *For any $0 < \varepsilon \le 1/2$ there exists a cost-oblivious defragmentation algorithm that takes as input (1) an arbitrary comparison function, (2) a set of objects with volume V, and (3) a current allocation of the objects using space at most $(1+\varepsilon)V$. The algorithm sorts the objects according to the comparison function, subject to:*

- *the total space usage at any time never exceeds $(1+\varepsilon)V+\Delta$ space, and*
- *the total cost is at most $O((1/\varepsilon)\log(1/\varepsilon))$ times the cost to allocate all of the objects.*

PROOF. First crunch the objects into the rightmost V space, leaving a size-$\lfloor\varepsilon V\rfloor$ prefix of the array empty. We reserve this prefix to run the cost-universal storage-reallocation algorithm. Starting with the leftmost object in the suffix, remove it from the suffix, store it temporarily in the Δ additional space, and then insert it into the prefix as per cost-universal storage reallocation. Since the storage reallocation guarantees at most $(1+\varepsilon)W$ space usage, for W total volume of objects in the prefix, at no point does the size-$\le (1+\varepsilon)W$ prefix overlap the size-$(V-W)$ suffix. When this process completes, the suffix is empty and all objects are in the cost-universal-storage data structure.

Next, move elements back to the suffix in reverse sorted order. Specifically, delete each object from the prefix (using the cost-universal storage-reallocation algorithm), which implicitly compacts the space used, and place the object just before its successor in the suffix. Again, at any time if W is the remaining volume of objects in the prefix, the prefix uses at most $(1+\varepsilon)W$ space, and the suffix uses exactly $V-W$ space, so the prefix does not overlap the suffix. □

3. FOOTPRINT MINIMIZATION IN A DATABASE CONTEXT

This section extends the storage-reallocation algorithm to the database setting by addressing two key issues: durability and blocking. To provide durability, we extend the algorithm to work with a checkpointing mechanism. Specifically, we show how to complete a buffer flush in $O(1/\varepsilon)$

checkpoints. During a flush, the memory footprint increases by an additive Δ term, up to $(1+\varepsilon)V + \Delta$, where V is the total length of all active objects, and Δ is the length of the longest object. The additive Δ is unavoidable due to the fact that when a large object is moved, its new location cannot overlap its old location.

To prevent updates from blocking for too long, we present a (partially) deamortized version. The deamortized data structure has the same amortized reallocation cost and memory footprint as the original, but it also has a worst-case reallocation cost of $O((1/\varepsilon)wf(1)+f(\Delta))$ for inserting/deleting a size-w object. That is, on each update, the total length of jobs reallocated is roughly proportional to the size of the object being inserted/deleted. Viewed differently, the deamortized bound says the desired footprint bound can be maintained with nonblocking updates, as long as the rate of updates is sufficiently lower than the speed of the machine.

3.1 Overview of the Checkpointing Model

Recall that moving an object updates the map that is maintained between logical and physical addresses. From time to time, and specifically during a checkpoint, this map is written to disk, so that a database that is recovering from a crash has access to the updated map. Suppose an object is reallocated. Then the map must be updated. But if a crash occurs before the next checkpoint, the updated map will not be available to the database on recovery. Therefore, we must maintain two copies of the data—at the old and new locations—until the next checkpoint has completed. Only then is it safe to assume that the database knows, in a durable fashion, the new location of the data.

The consequence for designing a reallocator is that from time to time, the database will perform a checkpoint and all the space that was freed since the last checkpoint will become available. If our algorithm would like to write to a freed but not checkpointed location it will block. Therefore, a reallocation algorithm is better if it requires fewer checkpoints to compete. For example, if we were to write the data to completely new locations, the algorithm would not block on any checkpoints, because we would not be reusing any space. However, the competitive ratio of the footprint would be at least two. We show below that we can achieve our bound of $(1+\varepsilon)$ competitive ratio while blocking on at most $O(1/\varepsilon)$ checkpoints.

The frequency of checkpointing is dependent on many considerations beyond the needs for reallocation, so we assume that checkpoints are initiated by the system, rather than our algorithm. There are other models of checkpointing, such as log-trimming through incremental checkpointing. A complete treatment of checkpointing is beyond the scope of this paper, though it would be interesting to see how different types of checkpointing interact with reallocation.

3.2 Flushing with Checkpoints

The goal of the flush here is identical to in Section 2, but the implementation details differ to accommodate the checkpointing model. Namely, the space used increases by an additive Δ, and the flush itself proceeds in several rounds with checkpoints in between. Another improvement here is that an inserted element gets inserted *before* the flush completes, whereas in Section 2 we assumed for simplicity that the insert blocked until the flush completed. The memory footprint at the end of the flush is identical to that of the previous algorithm.

Note that the additional Δ working space is unavoidable when reallocating large objects. To see this fact, consider a single size-Δ object. This object cannot be moved unless the target location is not overlapping with the original location. That is, if we have less than 2Δ space to work with, the object can never be moved as every target location overlaps its current location.

Inserting (allocating) and deleting (deallocating). Since objects only move during a buffer flush, the insert and delete procedure is almost identical to Section 2. The only difference here is that we insert the object *before* triggering a flush.

To insert an object, place it in the appropriate buffer segment as before. If there is insufficient space to place the object in any following buffer segment, place it at the end of the last buffer segment (filling and exceeding the buffer capacity) and trigger a flush. When deleting an object, a dummy delete request is inserted as in Section 2. If this delete request would overflow the last buffer, then trigger the flush without using space for the dummy delete request.

Buffer flush. A flush proceeds as follows. First identify the boundary size class b as before. Recall that the flush proceeds on size classes $i \geq b$. Let L denote the endpoint of the last object *before* the insert/delete that triggers the flush, i.e., if the total space is S including a newly inserted size-w object, then $L = S - w$.[5] Let L' be the desired memory footprint after the flush, but subtracting off the size of any flush-triggering insert.[6] That is, if the final data structure should take S' space after the flush, then $L' = S' - w$, where w is the size of the last insert if the flush was triggered by an insert. Let B be the total space occupied by the buffers involved in the flush. Move all objects from buffer segments $i \geq b$ to the end of the array, starting from location $(\max\{L, L'\} + B + \Delta)$. The important observation here is that $L + \Delta$ exceeds the location of the newly inserted object, so none of the target locations overlap any of the current objects. And hence all of these movements can be performed within a single checkpoint. The order in which the buffered objects are moved does not matter. This step of the flush is similar to Section 2, except the starting location is up to $B + \Delta$ slots later in the array.

Next, iterate over payload segments from largest to smallest, moving objects as late as possible in the array ending at location $(\max\{L, L'\} + B + \Delta)$. After this step, flushed payload segments are packed as late as possible before location $(\max\{L, L'\} + B + \Delta)$, and flushed buffer segments (including the newly inserted object) are packed as early as possible after $(\max\{L, L'\} + B + \Delta)$.

This payload-packing step, however, moves objects to locations in the array that may have previously been occupied, which would violate the checkpointing model. Instead, break these movements into phases with checkpoints between each phase. Move at least $B + 1$ and at most $B + \Delta$ volume of objects in each phase. (That is, move as many objects as

[5]This detail of subtracting off the newly inserted object is important to obtain a space usage of $(1+\varepsilon)V+\Delta$ throughout the flush rather than $(1+\varepsilon)V + O(\Delta)$.

[6]Similar to the procedure for "S" discussed in Section 2, L' can be calculated by first computing $\sum_{i \geq b}(V_t(i)+\lfloor \varepsilon' V_t(i) \rfloor)$.

possible before exceeding $B + \Delta$ volume. Since the largest object has size Δ, the minimum amount moved is $B + 1$.) As we shall prove, the movements within a phase do not overlap, and the total number of phases is $O(1/\varepsilon')$. Aside from checkpointing, this step differs from the version in Section 2 in that objects are packed later in the array rather than earlier, and hence the movements iterate from largest-to-smallest size class rather than smallest-to-largest. The reason for this change is to take advantage of the $B + \Delta$ working space available at the end of the region.

Next, iterate over payload segments from smallest to largest, moving the objects exactly where they should go in the array. This step, again, may move objects to space that was previously occupied, so we again break it into phases consisting of the next $B + 1$ to $B + \Delta$ target locations with a checkpoint following each phase.

Finally, move the buffered elements to their target locations. Since all buffered elements are currently located after $(\max\{L, L'\} + B + \Delta)$, and all target locations are before $L' + \Delta$, none of these movements overlap, and they can be performed within a single checkpoint.

Analysis. Note that the number of reallocations is similar to that in Section 2, with the only difference being one reallocation for the flush-triggering item. Hence the reallocation cost of Lemma 6 holds for this version of the algorithm. The space used after a flush completes is also identical to Section 2. It remains to prove three facts: 1) the space used during a flush is $(1+O(\varepsilon'))V+\Delta$ where V is the total volume of active jobs, 2) the object movements between checkpoints only move objects to nonoverlapping locations, and 3) the number of checkpoints is $O(1/\varepsilon')$ per flush.

LEMMA 8. *While processing any allocation/deallocation request, the total footprint used by the algorithm is at most $(1 + O(\varepsilon'))V + \Delta$, where V denotes the total volume of all currently active objects.*

PROOF. Let V_{before} and V_{after} denote the total volume of objects before and after the operation, respectively. Let S_{before} and S_{after} denote the total space of the data structure before and after the operation, respectively. According to Lemma 5, we have $S_{before} \le (1 + O(\varepsilon'))V_{before}$, and $S_{after} \le (1 + O(\varepsilon'))V_{after}$. The question is what happens during the operation, notably during a flush operation.

Suppose the flush is triggered by a size-w insertion. The volume during the flush is thus $V = V_{before} + w = V_{after}$. The space used to store all buffered objects, including the newly inserted object, is at most $w+B$, where B is the total amount of space devoted to buffers before the flush. Note that since the buffers are sized to less than an ε' fraction of the total space, we have $B \le \varepsilon' S_{before}$.

Case 1: $S_{before} \ge S_{after}$. Then these objects are written at an offset of $(S_{before} + B + \Delta)$, meaning that the total space during the flush is at most

$$\begin{aligned}
& (S_{before} + B + \Delta) + (w + B) \\
\le\ & (1 + 2\varepsilon')S_{before} + w + \Delta && \text{// upper bound on } B \\
\le\ & (1 + 2\varepsilon')\left[(1 + O(\varepsilon'))V_{before}\right] + w + \Delta && \text{// Lemma 5} \\
\le\ & (1 + O(\varepsilon'))V_{before} + w + \Delta && \text{// larger const in big-O} \\
\le\ & (1 + O(\varepsilon'))(V_{before} + w) + \Delta \\
=\ & (1 + O(\varepsilon'))V + \Delta\ .
\end{aligned}$$

Case 2: $S_{before} < S_{after}$. Then these objects are written at an offset of $(S_{after} - w) + B + \Delta$. And the total space during the flush is at most $S_{after} + 2B + \Delta \le (1+O(\varepsilon'))V_{after} + \Delta = (1+O(\varepsilon'))V + \Delta$, where the steps follow from analogous steps in Case 1.

In the case of a deletion, the argument is similar, except w becomes 0 in all the expressions, and $V = V_{before}$ throughout the flush. That is, the deleted object is considered active until the flush completes. □

LEMMA 9. *During a single phase of object movements between two checkpoints, all object starting locations are disjoint from all object ending locations.*

PROOF. First, consider the payload-packing step, where payload segments are packed to the right. At the start of the jth phase, let ℓ_j denote the last cell occupied by the payload segments that have yet to be packed, and let r_j denote the first occupied cell later than ℓ_j. We claim that at the start of each phase $r_j \ge \ell_j + B + \Delta$, which we shall prove by induction. If true, the claim implies disjointness: if the space between r_j and ℓ_j is at least $B+\Delta$, then we can pack up to $B + \Delta$ volume of jobs in front of ℓ_j during the jth phase before overlapping the ending position of jobs at ℓ_j.

We prove the claim by induction. The claim holds initially because $\ell_0 \le L$, and $r_0 \ge L+B+\Delta$. For the inductive step, observe that if X volume of objects are moved in phase j, then $\ell_{j+1} \le \ell_j - X$, and $r_{j+1} = r_j - X$. Combined with the inductive assumption that $r_j \ge \ell_j + B + \Delta$, we get $r_{j+1} \ge (\ell_j + B + \Delta) - X \ge ((\ell_{j+1} + X) + B + \Delta) - X = \ell_{j+1} + B + \Delta$.

We next consider the unpacking step, where the payload segments are moved to their final positions. Let ℓ_j denote the last cell occupied by unpacked payload segments at the start of the jth phase of movements, and let r_j denote the first cell occupied by the yet-to-be unpacked payload objects. We claim that $\ell_j + B + \Delta \le r_j$ (but this time we shall prove it by contradiction). If the claim holds, then we can afford to increase ℓ_j by $B + \Delta$ in each phase without violating the disjointness.

To prove the claim, suppose for the sake of contradiction that $\ell_j > r_j - B - \Delta$, and let X be the total volume remaining in the packed region. Then the final position of the last payload segment can end no earlier than $\ell_j + X > r_j + X - B - \Delta$ after the unpacking, and hence the space desired by these payload segments is at least $L' > (r_j + X) - B - \Delta$. We also have $r_j + X = \max\{L', L\} + B + \Delta$ is the offset at which the buffered objects were moved, which we simplify to $r_j + X \ge L' + B + \Delta$. Combining these two facts, we get $L' > (r_j + X) - B - \Delta \ge (L' + B + \Delta) - B - \Delta = L'$, i.e., $L' > L'$, which is a contradiction. □

LEMMA 10. *The number of checkpoints occurring during a flush is $O(1/\varepsilon')$.*

PROOF. The checkpoints are dominated by the packing and unpacking steps. Let $P(i)$ denote the total space of the ith payload segment at the time of the flush, i.e., the volume of jobs that were in this size class the last time a flush occurred. Then the total size of flushed buffers is $B = \sum_{i \ge b} \lfloor \varepsilon' P(i) \rfloor$, and the total space of the region being flushed is $S = \sum_{i \ge b} (P(i) + \lfloor \varepsilon' P(i) \rfloor)$. Since each movement phase does more than B work, showing that $B = \Omega(\varepsilon' S)$ would be sufficient. The only difficulty is the floor in the

expression, so we shall consider the case of large $P(i)$ and small $P(i)$ separately.

Case 1: sufficiently large P. More precisely, suppose $B = \sum_{i \geq b} \lfloor \varepsilon' P(i) \rfloor \geq \sum_{i \geq b} \varepsilon' P(i)/2$. Then $B = \Omega(\varepsilon' S)$, since $\sum_{i \geq b} P(i) \geq S/2$ for $\varepsilon < 1$.

Case 2: small P. Suppose $B < \sum_{i \geq b} \varepsilon' P(i)/2$. Note that $B = \sum_{i \geq b} \lfloor \varepsilon' P(i) \rfloor \geq \sum_{i \geq b} \varepsilon' P(i) - \Theta(\lg \Delta)$, since there are only $\Theta(\lg \Delta)$ size classes. It follows that $B < \sum_{i \geq b} \varepsilon' P(i)/2$ implies $\sum_{i \geq b} \varepsilon' P(i) = O(\lg \Delta)$, and hence $S = O((1/\varepsilon') \lg \Delta)$. The algorithm tries to move as many objects as it can until exceeding $B + \Delta$ volume, and hence every consecutive pair of phases moves at least $\Delta/2 = \Omega(\lg \Delta) = \Omega(\varepsilon' S)$ volume. □

3.3 Deamortizing the Data Structure

As described so far, the data structure is amortized—the average reallocation cost per update is low, but on some updates *every* active object may need to be reallocated (i.e., when all size classes are involved in a flush). This section improves the worst-case reallocation cost of a size-w update to $O((1/\varepsilon) w f(1) + f(\Delta))$, without hurting the amortized update cost or the maximum footprint.

Note that the deamortization described here builds on the checkpointing modification, yielding a worst-case $O(1/\varepsilon)$ checkpoints per operation.

Modifications to the algorithm. The main idea of our deamortization is that if a buffer flush performs a total of X reallocations by volume, then this work is spread across the subsequent $\varepsilon' X$ updates by volume. The question, however, is where to place new objects that are inserted during a flush. If, for example, an insert could trigger a smaller flush while a larger flush is still ongoing, that would present even more challenges. We tackle these problems by adding two more buffers to the data structure and modifying the flush, which serve to avoid the issue of nested flushes.

Augment the data structure to include one size-$\lfloor \varepsilon' V_f \rfloor$ buffer, called the ***tail buffer***, following all the size-class segments, where V_f is the total volume of all jobs active at the start of the previous buffer flush. The tail buffer is like any other buffer: objects are only placed in the tail buffer if all earlier buffers are too full, and a buffer flush is only triggered once the tail buffer becomes full. The point of the large tail buffer is to enable the flush to complete before triggering another flush.

When a flush is triggered, calculate the desired space and the temporary working space as before.[7] We treat all space immediately following the temporary working space as another buffer called the ***log***.

The flush process resembles the previous flush process (with or without checkpointing), except that:

1. Objects may be inserted/deleted during a flush. These updates are placed at the end of the log.
2. The work of the flush is spread across these subsequent updates. Specifically, on an insertion/deletion of a size-w object, perform (just over) the next $(4/\varepsilon')w$ steps of the flush by volume. Since a fractional object cannot be moved, the amount of volume processed may be as high $(4/\varepsilon')w + \Delta$.
3. There is an extra phase at the end. During this phase, all objects in the log are moved to their appropriate buffers, i.e., they are re-inserted/re-deleted. This phase proceeds in order from the beginning of the log. Updates may continue to be recorded at the end of the log during this phase. Since the volume moved is significantly larger than the size of the update, the re-insertion/re-deletion will eventually "catch up" to the end of the log, at which point the flush terminates and the log disappears.

[7] But the space is slightly larger now due to the $\lfloor \varepsilon' V \rfloor$ space necessary for the tail buffer.

Analysis. To show correctness of the new flush protocol, we argue that the log is drained completely before another flush gets triggered, i.e., before the tail buffer fills. Note that if the last update during a flush involves a large object, that update may finish the previous flush and trigger the next one. The point is only that the tail buffer cannot overflow before that time.

LEMMA 11. *Let V_f be the total volume of active objects at the time a flush is triggered. For any $\varepsilon' < 1$, the flush completes by the time the subsequent volume of updates first exceeds $\varepsilon' V_f$.*

PROOF. In the worst case, a flush may move every object twice. Specifically, the buffered elements are moved out of the buffers temporarily, then to their final location. Similarly, the payload segments are packed once and then unpacked to their final location. The total volume of reallocations of preexisting elements is thus at most $2V_f$. (Any delete records do not have to be reallocated; these are just destroyed.)

It follows that by the time $(\varepsilon'/2)V_f$ volume of updates are logged, all preexisting elements have been moved to their final locations. But this analysis does not take into account the elements logged during the flush. The next $(\varepsilon'/2)V_f$ volume of updates more than suffice to move all objects from the log to a buffer. □

The following lemmas bound the space and reallocation costs of the updated algorithm.

LEMMA 12. *After each allocation/deallocation request is processed, the total space used by the data structure is at most $(1 + O(\varepsilon'))V + \Delta$, where V denotes the total volume of all currently active objects. If a flush is not in progress, this space improves to $(1 + O(\varepsilon'))V$.*

PROOF. The only significant difference in space between this algorithm and the amortized one is the tail buffer and the log. The tail buffer has size at most $\varepsilon' V_f$, where V_f was the volume at the last flush. According to Lemma 11, the log also has size at most $\varepsilon' V_f$. Combined, the total increase to space is an additive $O(\varepsilon')V_f$. To complete the argument, we need only argue that $V_f = \Theta(V)$, where V is the current volume of active jobs, which is done in the proof of Lemma 5. □

LEMMA 13. *For subadditive cost function f, the amortized cost of inserting or deleting a size-w object is $O(f(w) \cdot (1/\varepsilon') \lg(1/\varepsilon'))$. Moreover, the worst-case cost of an insert or delete is $O((1/\varepsilon') w f(1) + f(\Delta))$.*

PROOF. The worst-case upper bound follows by construction. The algorithm only reallocates $(4/\varepsilon')w$ volume of objects per update, plus up to one last object to exceed

this volume. In the worst case, these objects are size-1 objects except the last which is size-Δ, for a total cost of $O((1/\varepsilon)wf(1) + f(\Delta))$.

As for the amortized bound, adding a larger buffer to the data structure only improves the amortized cost. Specifically, proof of Lemma 6 relied on *lower bounding* the volume of buffered objects, so the same analysis applies once an object is placed in a buffer. The deamortized data structure has an additional reallocation for each object that is placed in the log, moving it from the log to a buffer, but this only occurs once per object. □

Lower bound on worst-case cost. Note that $\Omega(f(\Delta))$ is a lower bound on the worst-case reallocation cost when maintaining a $(1+\varepsilon)V$ footprint size, as exhibited by the following lemma. It is not obvious whether $\Omega(wf(1))$ is also a lower bound on the worst-case reallocation cost of any algorithm. If so, then our deamortized structure's worst-case cost would be asymptotically optimal for constant ε. Although not a general lower bound, an $\Omega(wf(1))$ worst-case cost appears to be unavoidable for any algorithm that stores "enough" small objects after large objects. (And storing objects out of order in this way seems crucial for obtaining a *cost-oblivious* algorithm.) Informally, deleting a size-w object leaves a large hole in the array. To maintain the desired footprint, this hole must be filled by later objects. If all later objects are small (size-1), then a size-w delete may cause $\Omega(w)$ size-1 objects to move.

LEMMA 14. *For any reallocation algorithm that maintains a footprint of $(1+1/2)V$ and subadditive cost function f, there exists an update sequence such that at least one update has a reallocation cost of $\Omega(f(\Delta))$. This lower bound applies even if the reallocation algorithm knows f and the full update sequence.*

PROOF. Here is the sequence. First insert one size-Δ object. Then insert Δ size-1 objects. Then delete the size-Δ object. There are two cases to show the lower bound.
Case 1: some small-object insertion causes the large object to be reallocated. Then that insert has a reallocation cost of at least $f(\Delta)$.
Case 2: the large object does not get reallocated. Then the large object must end before position $(3/2)\Delta$ to achieve the footprint bound, and hence there must be at least $\Delta/2$ small objects appearing after the large one. When deleting the large object, those small objects must move in order to restore the $(3/2)\Delta$ footprint bound. Hence the cost of deleting the large objects is $\Omega(\Delta \cdot f(1)) = \Omega(f(\Delta))$ for subadditive f. □

4. REFERENCES

[1] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proc. 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 305–314, 1987.

[2] C. Archetti, L. Bertazzi, and M. G. Speranza. Reoptimizing the Traveling Salesman Problem. *Networks*, 42(3):154–159, 2003.

[3] C. Archetti, L. Bertazzi, and M. G. Speranza. Reoptimizing the 0-1 knapsack problem. *Disc. Appl. Math.*, 158(17):1879–1887, 2010.

[4] G. Ausiello, V. Bonifaci, and B. Escoffier. Complexity and approximation in reoptimization. In *Proc. CiE*, 2007.

[5] G. Ausiello, B. Escoffier, J. Monnot, and V. T. Paschos. Reoptimization of minimum and maximum traveling salesman's tours. *J. Disc. Alg.*, 7(4):453–463, 2009.

[6] M. A. Bender, R. Cole, E. D. Demaine, and M. Farach-Colton. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *Proc. 10th European Symposium on Algorithms (ESA)*, pages 139–151, 2002.

[7] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM J. Comp.*, 35(2):341–358, 2005.

[8] M. A. Bender, R. Ebrahimi, J. T. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In *Proc. 25th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 958–971, 2014.

[9] M. A. Bender, M. Farach-Colton, S. P. Fekete, J. T. Fineman, and S. Gilbert. Reallocation problems in scheduling. In *Proc. 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 271–279, 2013.

[10] M. A. Bender, S. P. Fekete, T. Kamphans, and N. Schweer. Maintaining arrays of contiguous objects. In *Proc. 17th International Symposium on Fundamentals of Computation Theory (FCT)*, pages 14–25, 2009.

[11] M. A. Bender and H. Hu. An adaptive packed-memory array. In *Proc. 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 20–29, 2006.

[12] M. A. Bender and H. Hu. An adaptive packed-memory array. *Trans. Datab. Syst.*, 32(4), 2007.

[13] A. Bendersky and E. Petrank. Space overhead bounds for dynamic memory management with partial compaction. *ACM Trans. Program. Lang. Syst.*, 34(3):13, 2012.

[14] J. Błażewicz and J. Nawrocki. Dynamic storage allocation with limited compaction - complexity and some practical implications. *Discrete Applied Mathematics*, 10(3):241 – 253, 1985.

[15] H.-J. Böckenhauer, L. Forlizzi, J. Hromkovic, J. Kneis, J. Kupke, G. Proietti, and P. Widmayer. Reusing optimal TSP solutions for locally modified input instances. In *Proc. Fourth IFIP International Conference on Theoretical Computer Science (TCS)*, pages 251–270, 2006.

[16] J. Bulánek, M. Koucký, and M. Saks. Tight lower bounds for the online labeling problem. In *Proc. 44th Symposium on Theory of Computing (STOC)*, pages 1185–1198, 2012.

[17] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Dynamic bin packing. *SIAM J. Comput.*, 12(2):227–258, 1983.

[18] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for np-hard problems. chapter Approximation Algorithms for Bin Packing: A Survey, pages 46–93. PWS Publishing Co., 1997.

[19] E. G. Coffman, Jr., D. S. Johnson, P. W. Shor, and R. R. Weber. Markov chains, computer proofs, and average-case analysis of best fit bin packing. In *Proc. Twenty-Fifth Annual ACM Symposium on Theory of Computing (STOC)*, pages 412–421, 1993.

[20] E. G. Coffman, Jr., D. S. Johnson, P. W. Shor, and R. R. Weber. Bin packing with discrete item sizes, part ii: Tight bounds on first fit. *Random Struct. Algorithms*, 10(1-2):69–101, 1997.

[21] N. Cohen and E. Petrank. Limitations of partial compaction: towards practical bounds. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 309–320, 2013.

[22] S. Davis, J. Edmonds, and R. Impagliazzo. Online algorithms to minimize resource reallocations and network communication. In *Proc. APPROX-RANDOM*, pages 104–115, 2006.

[23] S. P. Fekete, T. Kamphans, N. Schweer, C. Tessars, J. C. van der Veen, J. Angermeier, D. Koch, and J. Teich. Dynamic defragmentation of reconfigurable devices. *ACM Trans. Reconf. Technol. Syst.*, 5(2):8:1–8:20, June 2012.

[24] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 285–297, 1999.

[25] G. Galambos and G. J. Woeginger. On-line bin packing - a restricted survey. *Math. Meth. of OR*, 42(1):25–45, 1995.

[26] R. Graham, E. Lawler, J. Lenstra, and A. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Ann. Disc. Math.*, 5:287 – 326, 1979.

[27] N. G. Hall and C. N. Potts. Rescheduling for new orders. *Op. Res.*, 52(3), 2004.

[28] A. Itai and I. Katriel. Canonical density control. *Inf. Process. Lett.*, 104(6):200–204, 2007.

[29] A. Itai, A. G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *Proc. International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 417–431, 1981.

[30] K. Jansen and K.-M. Klein. A robust afptas for online bin packing with polynomial migration,. In *Proc. 40th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 589–600, 2013.

[31] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Applied Algorithms and Data Structures Series. Chapman and Hall/CRC, 2011.

[32] I. Katriel. Implicit data structures based on local reorganizations. Master's thesis, Technion – Isreal Inst. of Tech., Haifa, May 2002.

[33] K. C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, Oct. 1965.

[34] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, third edition, 1997.

[35] M. G. Luby, J. Naor, and A. Orda. Tight bounds for dynamic storage allocation. *SIAM J. Disc. Math.*, 9:155–166, 1996.

[36] J. Naor, A. Orda, and Y. Petruschka. Dynamic storage allocation with known durations. *Disc. Appl. Math.*, 3:203–213, 2000.

[37] H. Prokop. Cache-oblivious algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[38] J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *J. ACM*, 18(3):416–423, July 1971.

[39] J. M. Robson. Bounds for some functions concerning dynamic storage allocation. *J. ACM*, 21(3):491–499, July 1974.

[40] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *Comput. J.*, 20(3):242–244, 1977.

[41] P. Sanders, N. Sivadasan, and M. Skutella. Online scheduling with bounded migration. *Math. Oper. Res.*, 34(2):481–498, 2009.

[42] H. Shachnai, G. Tamir, and T. Tamir. A theory and algorithms for combinatorial reoptimization. In *Proc. 10th Latin American Theoretical INformatics Symposium (LATIN)*, pages 618–630, 2012.

[43] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, February 1985.

[44] D. W. Ting. Allocation and compaction - a mathematical model for memory management. In *Proc. ACM SIGMETRICS Conference on Computer Performance Modeling Measurement and Evaluation (SIGMETRICS)*, pages 311–317, 1976.

[45] Tokutek, Inc. TokuDB: MySQL Performance, MariaDB Performance. `http://www.tokutek.com/products/tokudb-for-mysql/`.

[46] Tokutek, Inc. TokuMX—MongoDB Performance Engine. `http://www.tokutek.com/products/tokumx-for-mongodb/`.

[47] A. T. Unal, R. Uzsoy, and A. S. Kiran. Rescheduling on a single machine with part-type dependent setup times and deadlines. *Ann. Op. Res.*, 70, 1997.

[48] J. C. Verschae. *The Power of Recourse in Online Optimization Robust Solutions for Scheduling, Matroid and MST Problems The Power of Recourse in Online Optimization: Robust Solutions for Scheduling, Matroid and MST Problems*. PhD thesis, Technischen Universität Berlin, June 2012.

[49] D. Willard. Maintaining dense sequential files in a dynamic environment (extended abstract). In *Proc. 14th ACM Symposium on Theory of Computing (STOC)*, pages 114–121, 1982.

[50] D. E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *Proc. SIGMOD*, pages 251–260, 1986.

[51] D. E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Inf. and Comput.*, 97(2):150–204, 1992.

[52] D. Woodall. The bay restaurant–a linear storage problem. *The American Mathematical Monthly*, 81(3):240–246, 1974.