Competitive Data-Structure Dynamization*

Claire Mathieu† Rajmohan Rajaraman‡ Neal E. Young§ Arman Yousefi¶

December 13, 2021

Abstract

*Data-structure dynamization*is a general approach for making static data structures dynamic. It is used extensively in geometric settings and in the guise of so-called *merge (or compaction)* policies in big-data databases such as LevelDB and Google Bigtable (our focus). Previous theoretical work is based on worst-case analyses for uniform inputs — insertions of one item at a time and constant read rate. In practice, merge policies must not only handle batch insertions and varying read/write ratios, they can take advantage of such non-uniformity to reduce cost on a per-input basis.

To model this, we initiate the study of data-structure dynamization through the lens of competitive analysis, via two new online set-cover problems. For each, the input is a sequence of disjoint sets of weighted items. The sets are revealed one at a time. The algorithm must respond to each with a set cover that covers all items revealed so far. It obtains the cover incrementally from the previous cover by adding one or more sets and optionally removing existing sets. For each new set the algorithm incurs *build cost* equal to the weight of the items in the set. In the first problem the objective is to minimize total build cost plus total *query cost*, where the algorithm incurs a query cost at each time t equal to the current cover size. In the second problem, the objective is to minimize the build cost while keeping the query cost from exceeding k (a given parameter) at any time. We give deterministic online algorithms for both variants, with competitive ratios of $\Theta(\log^* n)$ and k, respectively. The latter ratio is optimal for the second variant.

1 Introduction and statement of results

1.1 Background

A *static* data structure is built once to hold a fixed set of items, queried any number of times, and then destroyed, without changing throughout its lifespan. *Dynamization* is a generic technique for transforming any static container data structure into a *dynamic* one that supports insertions and queries intermixed arbitrarily. The dynamic structure stores the items inserted so far in static containers called *components*, supporting each insertion by destroying some components and building new ones in toto, and supporting each query by querying each component independently. Dynamization has been applied in computational geometry [38, 21, 1, 2, 17], in geometric streaming algorithms [32, 7, 29, 33], and to design external-memory dictionaries [6, 52, 3, 11].

*A preliminary version of this paper appeared in SODA 2021 [42].

†CNRS Paris

‡Supported by NSF grants 1535929 and 1909363; Northeastern University

§Supported by Google Research Award & NSF grant 1619463; University of California Riverside and Northeastern University

¶Google

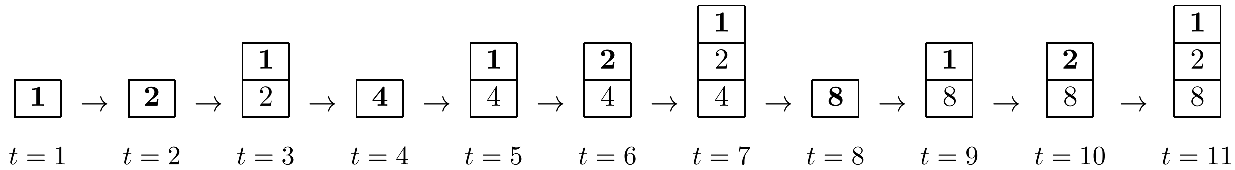

Figure 1: Steps 1–11 of the binary transform [12, 13]. Each cell $\boxed{i}$ is a component holding i items, where i is a distinct power of two. In each step one item is inserted and held in the new (top, bolded) component.

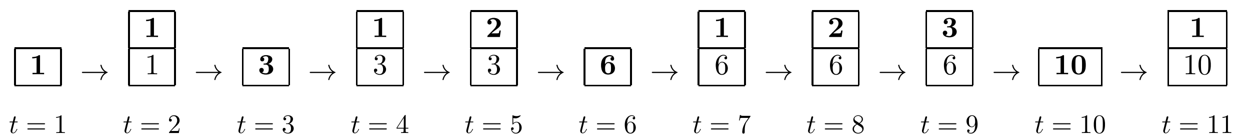

Figure 2: Steps 1–11 of the 2-binomial transform [13]. At time t the top and bottom components hold $\binom{i_1}{1}$ and $\binom{i_2}{2}$ items where $0 \leq i_1 < i_2$ and $\binom{i_1}{1} + \binom{i_2}{2} = t$. For example at time t = 8, $i_1 = 2$ and $i_2 = 4$. If $i_1 = 0$ there is only one component, the bottom component.

Bentley's *binary transform* [12, 13], later called the *logarithmic method* [51, 45], is a widely used example. It maintains its components so that the number of items in each component is a distinct power of two. Each insert operation mimics a binary increment: it destroys the components of size $2^0, 2^1, 2^2, \ldots, 2^{j-1}$, where $j \geq 0$ is the minimum such that there is no component of size 2^j, and builds one new component of size 2^j, holding the contents of the destroyed components and the inserted item. (See Figure 1.) Meanwhile, each query operation queries all current components, combining the results appropriately for the data type. During n insertions, whenever an item is incorporated into a new component, the item's new component is at least twice as large as its previous component, so the item is in at most $\log_2 n$ component builds. That is, the worst-case *write amplification* is at most $\log_2 n$. Meanwhile, the number of components never exceeds $\log_2 n$, so each query examines at most $\log_2 n$ components. That is, the worst-case *read amplification* is at most $\log_2 n$.

Bentley and Saxe's k-***binomial*** transform is a variant of the binary transform [13]. It maintains k components at all times, of respective sizes $\binom{i_1}{1}, \binom{i_2}{2}, \ldots, \binom{i_k}{k}$ such that $0 \leq i_1 < i_2 < \cdots < i_k$. (This decomposition is guaranteed to exist and be unique. See Figure 2.) It thus ensures read amplification at most k, independent of n, but its write amplification is at most $(k!\, n)^{1/k}$, about $\frac{k}{e} n^{1/k}$ for large k. This tradeoff between worst-case read amplification and worst-case write amplification is optimal up to lower-order terms, as is the tradeoff achieved by the binary transform.

Dynamization underlies applied work on external-memory (i.e., big-data) ordered dictionaries, most famously O'Neil et al's *log-structured merge* (LSM) architecture [44] (building on [48, 47]). The dynamization scheme it uses can be viewed as a generalization of the binary transform. The worst-case tradeoff it achieves is optimal, in some parameter regimes, among all external-memory structures [5, 15, 53]. Many current industrial storage systems — NoSQL or NewSQL databases — use such an LSM architecture. These include Google's Bigtable [20] (and Spanner [24]), Amazon's Dynamo [26], Accumulo (by the NSA) [35], AsterixDB [4], Facebook's Cassandra [37], HBase and Accordion (used by Yahoo! and others) [30, 14], LevelDB [27], and RocksDB [28].

In this context, dynamization algorithms are called *merge* (or *compaction*) policies [41]. Recently inserted items are cached in RAM, while all other items are stored in immutable (static) on-disk files, called *components*. Each query (if not resolved in cache) searches the current components for the queried item, using one disk access[1] per component. The components are managed using the merge policy: the items in cache are periodically flushed to disk in one batch, where they are incorporated by destroying and building components[2] according to the policy. Any dynamization algorithm yields such a merge policy in a naive way, just by treating each inserted batch of items as a single unified item of unit size. The read and write amplifications of the resulting "naive" merge policy will be the same as those of the underlying dynamization algorithm.

But this naive approach leaves room for improvement. In production LSM systems the sizes of inserted batches can vary by orders of magnitude [14, §2] (see also [16, 10, 9]). The relative query and insertion rates also vary with time. Such non-uniform workloads can be substantially *easier* in that they admit a solution with average write amplification (over all inserted items) and average read amplification (over all queries) well below worst case, achieving lower total cost. Theoretical dynamization models to date do not address this. Further, merge policies obtained by naively adapting theoretical algorithms don't adapt to non-uniformity, so their average read and write amplifications are close to worst case on most inputs.

In contrast, practical compaction policies do adapt to non-uniformity, but only heuristically. For example, Bigtable's default compaction policy (which, like the k-binomial transform, is configured by a single parameter k and maintains at most k components) is as follows: in response to each insert (cache flush), create a new component holding the inserted items; then, if there are more than k components, merge the i most-recently created components into one, where $i \geq 2$ is chosen minimally so that, for each remaining component S, the size of S in bytes exceeds the total size of all components newer than S [50].

This paper begins to bridge this gap between theory and practice. It proposes new dynamization problems —*Min-Sum Dynamization* and *k-Component Dynamization*— that model non-uniform insert/query rates and insertions of non-uniform size, and it brings competitive analysis to bear to measure how well algorithms take advantage of this non-uniformity. It introduces new algorithms that have substantially better theoretical competitive ratios than existing algorithms, and that outperform some currently used algorithms on typical inputs.

1.2 Min-Sum Dynamization

Definition 1. *The input is a sequence $I = (I_1, I_2, \ldots, I_n)$ of disjoint sets of items, where each item $x \in I_t$ is "inserted at time t" and has a fixed, non-negative weight, $\mathsf{wt}(x)$. A solution is a sequence $\mathcal{C} = (\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n)$, where each $\mathcal{C}_t$ is a collection of sets (called* components*) satisfying $\bigcup_{S\in\mathcal{C}_t} S = U_t$, where $U_t = \bigcup_{i=1}^{t} I_i$. That is, $\mathcal{C}_t$ is a* set cover *for the items inserted by time t.*

For each time $t \in \{1, 2, \ldots, n\}$, the build cost *at time t is the total weight in new sets: $\sum_{S\in\mathcal{C}_t\setminus\mathcal{C}_{t-1}} \mathsf{wt}(S)$, where $\mathsf{wt}(S)$ denotes $\sum_{x\in S} \mathsf{wt}(x)$ and $\mathcal{C}_0$ denotes the empty set. The* query *cost at time t is $|\mathcal{C}_t|$ — the number of components in the current cover, $\mathcal{C}_t$. The objective is to minimize the cost of the solution, defined as the sum of all build costs and query costs over time.*

[1]Database servers are typically configured so that RAM size is 1–3% of disk size, even as RAM and disk sizes grow according to Moore's law [31, p. 227]. A disk block typically holds at least thousands of items. Hence, an index for every disk component, storing the minimum item in each disk block in the component, fits easily in RAM. Then querying any component (a file storing its items in sorted order) for a given item requires accessing just one disk block, determined by checking the index [31, p. 232].

[2]Crucially, builds use sequential (as opposed to random) disk access. This is why LSM systems outperform B^+ trees on write-heavy workloads. See [41, § 2.2.1–2.2.2] for details.

algorithm Adaptive-Binary$(I_1, I_2, \ldots, I_n)$ — *for Min-Sum Dynamization*

1. maintain a cover (collection of components), initially empty
2. for each time $t = 1, 2, \ldots, n$:

2.1. if $I_t \neq \emptyset$: add I_t as a new component

2.2. let $j \geq 0$ be the maximum integer such that t is an integer multiple of 2^j

2.3. if there are multiple components S such that $\mathsf{wt}(S) \leq 2^j$: merge them into one new component

Figure 3: A $\Theta(\log^* m)$-competitive algorithm for Min-Sum Dynamization (Theorem 1).

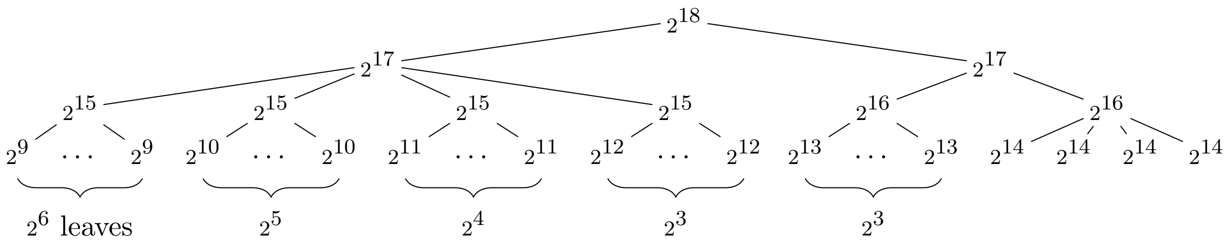

Figure 4: The "merge tree" for an execution of the Adaptive-Binary algorithm (Figure 3). The input sequence starts with $m = 132$ inserts $I_1, I_2, \ldots, I_{132}$ — one for each leaf, of weight equal to leaf's label. It continues with $2^{16} - 132$ empty inserts ($I_t = \emptyset$). At each time $t = 2^9, 2^{10}, 2^{11}, \ldots, 2^{17}$ (during the empty inserts) the algorithm merges all components of weight t to form a single new component, their parent. In this way, the algorithm builds a component for each node, with weight equal to the node's label. At time $t = 2^{17}$ the final component is built — the root, of weight 2^{18}, containing all items. The algorithm merges each item four times, so pays build cost 4×2^{18}.

Remarks. A-priori, the definition of total read cost as $\sum_{t=1}^{n} |\mathcal{C}_t|$ assumes one query per insert, but non-uniform query rates can be modeled by reduction: to model consecutive queries with no intervening insertions, separate the consecutive queries by artificial insertions with $I_t = \emptyset$ (inserting an empty set); to model consecutive insertions with no intervening queries, aggregate the consecutive insertions into a single insertion.

In LSM applications, each unit of query cost represents the time for one random disk access, whereas each unit of build cost represents the (much smaller) time to read and write a byte during sequential disk access. For Min-Sum Dynamization, to normalize these relative costs, take the weight of each item x to be the time to read and write x to disk (within a batch read or write of many items, where disk access is sequential and disk-access time is amortized across many items) normalized by dividing by the disk-access time.

Results on Min-Sum Dynamization

Theorem 1 (Section 2)**.** *For Min-Sum Dynamization, the online algorithm Adaptive-Binary (Figure 3) has competitive ratio $\Theta(\log^* m)$, where $m \leq n$ is the number of non-empty insertions.*

The iterated logarithm is defined by $\log^* m = 1 + \log^* \log_2 m$, except $\log^* m = 0$ for $m \leq 1$.

Roughly speaking, every 2^j time steps ($j \in \{0, 1, 2, \ldots\}$), the algorithm merges all components of weight 2^j or less into one. Figure 4 illustrates one execution of the algorithm. The bound in the theorem is tight for the algorithm.

In contrast, consider the naive adaptation of Bentley's binary transform (i.e., treat each insertion I_t as a size-1 item, then apply the transform). On inputs with $\mathsf{wt}(I_t) = 1$ for all t the algorithms produce the same (optimal) solution. But the competitive ratio of the naive adaptation is $\Theta(\log n)$. (To see the lower bound, consider an input that inserts an item of weight n^2, then $n-1$ single new items of infinitesimal weight. The naive adaptation pays build cost $\Omega(n^2 \log n)$, whereas the optimum and the algorithm of Figure 3 both pay build cost n^2 plus query cost $2n$.)

Min-Sum Dynamization is a special case of *Set Cover with Service Costs*, for which Buchbinder et al. give a randomized online algorithm [18]. For Min-Sum Dynamization, their bound on the algorithm's competitive ratio simplifies to $O(\log^2 n)$.

1.3 K-Component Dynamization

Definition 2. *The input is the same as for Min-Sum Dynamization, but solutions are restricted to those having query cost at most k at each time t (that is, $\max_t |\mathcal{C}_t| \leq k$). The objective is to minimize the total build cost.*

Remark. Perhaps the most closely related well-studied problem is dynamic TCP acknowledgment, a generalization of the classic ski-rental problem [34, 19]. TCP acknowledgement can be viewed as a variant of 2-Component Dynamization, in which time is continuous and building a new component that contains all items inserted so far (corresponding to a "TCP-ack") has cost 1 regardless of the component weight.

Deletions, updates, and expiration. The problem definitions above model queries and insertions. We next consider updates, deletions, and item expiration. We model items in LSM dictionaries as key/value pairs, timestamped by insertion time, and with an optional expiration time. Updates and deletions are lazy ("out of place" [41, §2], [40]): update just inserts an item with the given key/value pair (as usual), while delete inserts an item for the given key with a so-called *tombstone* (a.k.a. *antimatter*) value. Multiple items with the same key may be stored, but only the newest matters: a query, given a key, returns the newest item inserted for that key, or "none" if that item is a tombstone or has expired. As a component S is built, it is "garbage collected": for each key, among the items in S with that key, only the newest is written to disk — all others are discarded.

To model this, we define three generalizations of the problems. To keep the definitions clean, in each variant the input sets must still be disjoint and the current cover must still contain all items inserted so far. To model aspects such as updates, deletions, and expirations, we only redefine the build cost.

Decreasing Weights. Each item $x \in I_t$ has weights $\mathsf{wt}_t(x) \geq \mathsf{wt}_{t+1}(x) \geq \cdots \geq \mathsf{wt}_n(x)$. The cost of building a component $S \subseteq U_t$ at time t is redefined as $\mathsf{wt}_t(S) = \sum_{x \in S} \mathsf{wt}_t(x)$. This variant is useful for technical reasons.

LSM. Each item is a timestamped key/value pair with an expiration time. Given a subset S of items, the set of *non-redundant* items in S, denoted $\mathsf{nonred}(S)$, consists of those that have no newer item in S with the same key. The cost of building a component S at time t, denoted $\mathsf{wt}_t(S)$, is redefined as the sum, over all non-redundant items x in S, of the item weight $\mathsf{wt}(x)$, or the weight of the tombstone item for x if x has expired. The latter weight must be at most $\mathsf{wt}(x)$. Items with the same key may have different weights, and must have distinct timestamps. For any two items $x \in I_t$ and $x' \in I_{t'}$ with $t < t'$, the timestamp of x must be

algorithm Greedy-Dual($I_1, I_2, \ldots, I_n$) — *for k-Component Dynamization with decreasing weights*
1. maintain a cover (collection of components), initially empty
2. for each time $t = 1, 2, \ldots, n$ such that $I_t \neq \emptyset$:
2.1. if there are k current components:
2.1.1. increase all components' credits continuously until some component S has credit$[S] \geq \mathsf{wt}_t(S)$
2.1.2. let S_0 be the oldest component such that credit$[S_0] \geq \mathsf{wt}_t(S_0)$
2.1.3. merge I_t, S_0 and all components newer than S_0 into one new component S'
2.1.4. initialize credit$[S']$ to 0
2.2. else:
2.2.1. create a new component from I_t, with zero credit

Figure 5: A "newest-first" k-competitive algorithm for k-Component Dynamization with decreasing weights (Theorem 3). To obtain a k-competitive algorithm for the LSM variant (Theorem 4, Corollary 5), replace $\mathsf{wt}_t(S_0)$ throughout by $\mathsf{wt}'_t(S_0) = \mathsf{wt}_t(S') - \mathsf{wt}_t(S' \setminus S_0)$, for S' as defined in Line 2.1.3 ($S' = \bigcup_{h=i}^{t} I_h$, for i s.t. $(\exists j)\, S_0 = \bigcup_{h=i}^{j} I_h$).

less than the timestamp of x'. This variant applies to LSM systems.[3]

General. Instead of weighting the items, build costs are specified directly for sets. At each time t a *build-cost function* $\mathsf{wt}_t : 2^{U_t} \to \mathbb{R}_+$ is revealed (along with I_t), directly specifying the build cost $\mathsf{wt}_t(S)$ for every possible component $S \subseteq U_t$. The build-cost function must obey the following restrictions. For all times $i \leq t$ and sets $S, S' \subseteq U_t$,

(R1) sub-additivity: $\mathsf{wt}_t(S \cup S') \leq \mathsf{wt}_t(S) + \mathsf{wt}_t(S')$

(R2) suffix monotonicity: if[4] $S \neq U_t$, then $\mathsf{wt}_t(S \setminus U_i) \leq \mathsf{wt}_t(S)$,

(R3) temporal monotonicity: $\mathsf{wt}_i(S) \geq \mathsf{wt}_t(S)$

The build costs implicit in the other defined variants do obey Restrictions (R1)–(R3).[5] The restrictions also hold, for example, if each item has a weight and $\mathsf{wt}_t(S) = \max_{x \in S} \mathsf{wt}(x)$.

Definition 3 (competitive ratio)**.** *An algorithm is* online *if for every input I it outputs a solution $\mathcal{C}$ such that at each time t its cover $\mathcal{C}_t$ is independent of $I_{t+1}, I_{t+2}, \ldots, I_n$, all build costs $\mathsf{wt}_{t'}(S)$ at times $t' > t$, and n. The* competitive ratio *is the supremum, over all inputs with m non-empty insertions, of the cost of the algorithm's solution divided by the optimum cost for the input. An algorithm is $c(m)$-*competitive *if its competitive ratio is at most $c(m)$.*

Results on k-Component Dynamization

Theorem 2 (Section 3.1)**.** *For k-Component Dynamization (and consequently for its generalizations) no deterministic online algorithm has ratio competitive ratio less than k.*

[3] LSM systems delete tombstone items during full merges (i.e., when building a component $S = U_t$ at time t). This is not captured by the LSM model here, but is captured by the general model that follows. See Section 5.2.

[4] The exception for $S = U_t$ allows modeling deletion of tombstone items during full merges.

[5] The LSM build cost obeys (R1) because $\mathsf{nonred}(S \cup S') \subseteq \mathsf{nonred}(S) \cup \mathsf{nonred}(S)'$. It obeys (R2) because $\mathsf{nonred}(S \setminus U_i) \subseteq \mathsf{nonred}(S)$. It obeys (R3) because the tombstone weight for each item x is at most $\mathsf{wt}(x)$.

algorithm $\mathsf{B}_1(I_1, I_2, \ldots, I_n)$	— *for* $k = 1$
1. for $t = 1, 2, \ldots, n$: use cover $\mathcal{C}_t = \{U_t\}$ where $U_t = \bigcup_{i=1}^t I_i$	— *one component holding all items*
algorithm $\mathsf{B}_k(I_1, I_2, \ldots, I_n)$	— *for* $k \geq 2$
1. initialize $t' = 1$	— *variable* t' *holds the start time of the current phase*
2. for $t = 1, 2, \ldots, n$:	
2.1. let $\mathcal{C}' = \mathsf{B}_{k-1}(I_{t'}, I_{t'+1}, \ldots, I_t)$	— *the solution generated by* B_{k-1} *for the current phase so far*
2.2. if the total cost of $\mathcal{C}'$ exceeds $(k-1)\,\mathsf{wt}_t(U_t)$: take $\mathcal{C}_t = \{U_t\}$ and let $t' = t+1$	— *end the phase*
2.3. else: use cover $\mathcal{C}_t = \{U_{t'}\} \cup \mathcal{C}'_t$, where $\mathcal{C}'_t$ is the last cover in $\mathcal{C}'$	— $\mathcal{C}'_t$ *has at most* $k-1$ *components*

Figure 6: Recursive algorithm for general k-Component Dynamization (Theorem 6).

Theorem 3 (Section 3.2)**.** *For k-Component Dynamization with decreasing weights (and plain k-Component Dynamization) the deterministic online algorithm in Figure 5 is k-competitive.*

For comparison, consider the naive generalization of Bentley and Saxe's k-binomial transform to k-Component Dynamization (treat each insertion I_t as one size-1 item, then apply the transform). On inputs with $\mathsf{wt}(I_t) = 1$ for all t, the two algorithms produce essentially the same optimal solution. But the competitive ratio of the naive algorithm is $\Omega(kn^{1/k})$ for any $k \geq 2$. (Consider inserting a single item of weight 1, then $n-1$ single items of weight 0. The naive algorithm pays $\Omega(kn^{1/k})$. The optimum pays $O(1)$, as do the algorithms in Figures 5 and 6.)

Bigtable's default algorithm (Section 1.1) solves k-Component Dynamization, but its competitive ratio is $\Omega(n)$. For example, with $k = 2$, given an instance with $\mathsf{wt}(I_1) = 3$, $\mathsf{wt}(I_2) = 1$, and $\mathsf{wt}(I_t) = 0$ for $t \geq 3$, it pays $n + 2$, while the optimum is 4. (In fact, the algorithm is memoryless — each $\mathcal{C}_t$ is determined by $\mathcal{C}_{t-1}$ and I_t. No deterministic memoryless algorithm has competitive ratio independent of n.) Even for uniform instances ($\mathsf{wt}(I_t) = 1$ for all t), Bigtable's default incurs cost quadratic in n, whereas the optimum is $\Theta(kn^{1+1/k})$.

Bentley and Saxe showed that their solutions were optimal (for uniform inputs) among a restricted class of solutions that they called *arboreal transforms* [13]. Here we call such solutions *newest-first*:

Definition 4. *A solution $\mathcal{C}$ is* newest-first *if at each time t, if $I_t = \emptyset$ it creates no new components, and otherwise it creates one new component, by merging I_t with some $i \geq 0$ newest components into a single component (destroying the merged components). Likewise, $\mathcal{C}$ is* lightest-first *if, at each time t with $I_t \neq \emptyset$, it merges I_t with some $i \geq 0$ lightest components. An* algorithm *is newest-first (lightest-first) if it produces only newest-first (lightest-first) solutions.*

The Min-Sum Dynamization algorithm Adaptive-Binary (Figure 3) is lightest-first. The k-Component Dynamization algorithm Greedy-Dual (Figure 5) is newest-first. In a newest-first solution, every cover $\mathcal{C}_t$ partitions the set U_t of current items into components of the form $\bigcup_{t=i}^j I_t$ for some i, j.

Any newest-first algorithm for the decreasing-weights variant of either problem can be "bootstrapped" into an equally good algorithm for the LSM variant:

Theorem 4 (Section 3.3)**.** *Any newest-first online algorithm for k-Component (or Min-Sum) dynamization with decreasing weights can be converted into an equally competitive algorithm for the LSM variant.*

Combined with the newest-first algorithm in Figure 5, Theorems 3 and 4 yield a k-competitive algorithm for LSM k-Component Dynamization:

Corollary 5 (Section 3.3). *The online algorithm for LSM k-Component Dynamization described in the caption of Figure 5 has competitive ratio k.*

Our final algorithm is for the general variant:

Theorem 6 (Section 3.4)**.** *For general k-Component Dynamization, the deterministic online algorithm B_k in Figure 6 is k-competitive.*

The algorithm, B_k, partitions the input sequence into phases. Before the start of each phase, it has just one component in its cover, called the current "root", containing all items inserted before the start of the phase. During the phase, B_k recursively simulates B_{k-1} to handle the insertions occurring during the phase, and uses the cover that consists of the root component together with the (at most $k-1$) components currently used by B_{k-1}. At the end of the phase, B_k does a *full merge* — it merges all components into one new component, which becomes the new root. It extends the phase maximally subject to the constraint that the cost incurred by B_{k-1} during the phase does not exceed $k-1$ times the cost of the full merge that ends the phase.

1.4 Properties of Optimal Offline Solutions

Bentley and Saxe showed that, among newest-first solutions (which they called *arboreal*), their various transforms were near-optimal for uniform inputs [12, 13]. Mehlhorn showed (also for uniform inputs) that the best newest-first solutions have cost at most a constant times optimum [43]. We generalize and strengthen Mehlhorn's result:

Theorem 7 (Section 4)**.** *Every instance of k-Component or Min-Sum Dynamization has an optimal solution that is newest-first and lightest-first.*

One consequence is that Bentley and Saxe's transforms give optimal solutions (up to lower-order terms) for uniform inputs. Another is that, for Min-Sum and k-Component Dynamization, optimal solutions can be computed in time $O(n^3)$ and $O(kn^3)$, respectively, because optimal newest-first solutions can be computed in these time bounds via natural dynamic programs.

The body of the paper gives the proofs of Theorems 1–7.

2 Min-Sum Dynamization (Theorem 1)

Theorem 1. *For Min-Sum Dynamization, the online algorithm Adaptive-Binary (Figure 3) has competitive ratio $\Theta(\log^* m)$, where $m \leq n$ is the number of non-empty insertions.*

We prove the theorem in two parts:

(i) The competitive ratio is $O(\log^* m)$ (proof in Section 2.1).

(ii) The competitive ratio is $\Omega(\log^* m)$ (proof in Section 2.2).

2.1 Part (i): the competitive ratio is $O(\log^* m)$

Fix an input $I = (I_1, I_2, \ldots, I_n)$ with $m \le n$ non-empty sets. Let $\mathcal{C}$ be the algorithm's solution. Let $\mathcal{C}^*$ be an optimal solution, of cost OPT. For any time t, call the 2^j chosen in Line 2.2 the *capacity* $\mu(t)$ of time t, and let S_t be the newly created component (if any) in Line 2.3.

It is convenient to over-count the algorithm's build cost as follows. In Line 2.3, if there is exactly one component S with $\mathsf{wt}(S) \le 2^j$, the algorithm as stated doesn't change the current cover, but we pretend for the analysis that it does — specifically, that it destroys and rebuilds S, paying its build cost $\mathsf{wt}(S)$ again at time t. This allows a clean statement of the next lemma. In the remainder of the proof, the "build cost" of the algorithm refers to this over-counted build cost.

We first bound the total query cost, $\sum_t |\mathcal{C}_t|$, of $\mathcal{C}$.

Lemma 1.1. *The total query cost of $\mathcal{C}$ is at most twice the (over-counted) build cost of $\mathcal{C}$, plus* OPT.

Proof. Let S be any component in $\mathcal{C}$ of weight $\mathsf{wt}(S) \ge 1$. Each new occurrence of S in $\mathcal{C}$ contributes at most $2\,\mathsf{wt}(S)$ to $\mathcal{C}$'s query cost. Indeed, let $2^j \ge \mathsf{wt}(S)$ be the next larger power of 2. Times with capacity 2^j or more occur every 2^j time steps. So, after $\mathcal{C}$ creates S, $\mathcal{C}$ destroys S within $2^j \le 2\,\mathsf{wt}(S)$ time steps; note that we are using here the over-counted build cost. So $\mathcal{C}$'s query cost from such components is at most twice the build cost of $\mathcal{C}$.

The query cost from the remaining components (with $\mathsf{wt}(S) < 1$) is at most n, because by inspection of the algorithm each cover $\mathcal{C}_t$ has at most one such component — the component S_t created at time t. The query cost of $\mathcal{C}^*$ is at least n, so $n \le \mathsf{OPT}$, proving the lemma. ☐

Define Δ to be the maximum number of components merged by the algorithm in response to any query. Note that $\Delta \le m$ simply because there are at most m components at any given time in $\mathcal{C}$. (Only Line 2.1 increases the number of components, and it does so only if I_t is non-empty.) The remainder of the section bounds the build cost of $\mathcal{C}$ by $O(\log^*(\Delta)\,\mathsf{OPT})$. By Lemma 1.1, this will imply prove Part (i) of the theorem.

The total weight of all components I_t that the algorithm creates in Line 2.1 is $\sum_t \mathsf{wt}(I_t)$, which is at most OPT because every $x \in I_t$ is in at least one new component in $\mathcal{C}^*$ (at time t). To finish, we bound the (over-counted) build cost of the components that the algorithm builds in Line 2.3, i.e., $\sum_t \mathsf{wt}(S_t)$.

Observation 1.2. *The difference between any two distinct times t and t' is at least* $\min\{\mu(t), \mu(t')\}$.

(This holds because t and t' are distinct integer multiples of $\min\{\mu(t), \mu(t')\}$. See Figure 7.)

Charging scheme. For each time t at which Line 2.3 creates a new component S_t, have S_t *charge* to each item $x \in S_t$ the weight $\mathsf{wt}(x)$ of x. Have x in turn charge $\mathsf{wt}(x)$ to each optimal component $S^* \in \mathcal{C}^*_t$ that contains x at time t. The entire build cost $\sum_t \mathsf{wt}(S_t)$ is charged to components in $\mathcal{C}^*$. To finish, we show that each component S^* in $\mathcal{C}^*$ is charged $O(\log^* \Delta)$ times S^*'s contribution (via its build and query costs) to OPT. (Recall that Δ is the maximum number of components the algorithm merges in response to any query.)

Fix any such S^*. Let $[t_1, t_2]$ be the *interval* of S^* in $\mathcal{C}^*$. That is, $\mathcal{C}^*$ adds S^* to its cover at time t_1, where it remains through time t_2, so its contribution to OPT is $t_2 - t_1 + 1 + \mathsf{wt}(S^*)$. At each (integer) time $t \in [t_1, t_2]$, component S^* is charged $\mathsf{wt}(S^* \cap S_t)$. To finish, we show $\sum_{t=t_1}^{t_2} \mathsf{wt}(S^* \cap S_t) = O(t_2 - t_1 + \log^*(\Delta)\,\mathsf{wt}(S^*))$.

By Observation 1.2, there can be at most one time $t' \in [t_1, t_2]$ with capacity $\mu(t') > t_2 - t_1 + 1$. If there is such a time t', the charge received then, i.e. $\mathsf{wt}(S^* \cap S_{t'})$, is at most $\mathsf{wt}(S^*)$. To finish, we bound the charges at the times $t \in [t_1, t_2] \setminus \{t'\}$, with $\mu(t) \le t_2 - t_1 + 1$.

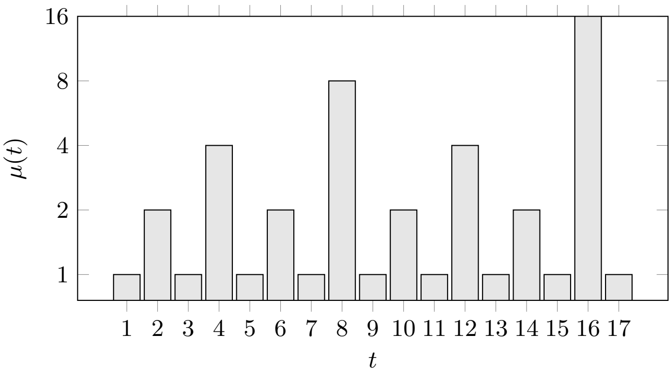

Figure 7: The capacities $\mu(t)$ as a function of t.

Definition 5 (dominant)**.** *Classify each such time t and $\mathcal{C}$'s component S_t as* dominant *if the capacity $\mu(t)$ strictly exceeds the capacity $\mu(i)$ of every earlier time $i \in [t_1, t-1]$ ($\mu(t) > \max_{i=t_1}^{t-1} \mu(i)$) in S^*'s interval $[t_1, t_2]$. Otherwise t and S_t are* non-dominant*.*

Lemma 1.3 (non-dominant times)**.** *The net charge to S^* at non-dominant times is at most $t_2 - t_1$.*

Proof. Let τ_1 be any dominant time. Let $\tau_2 > \tau_1$ be the next larger dominant time step, if any, else $t_2 + 1$. Consider the charge to S^* during the open interval (τ_1, τ_2). We show that this charge is at most $\tau_2 - \tau_1 - 1$.

Component S^* is built at time $t_1 \le \tau_1$, so $S^* \subseteq U_{\tau_1}$. At time τ_1, every item x that can charge S^* (that is, $x \in S^*$) is in some component S in $\mathcal{C}_{\tau_1}$. By the definition of dominant, each time in $t \in (\tau_1, \tau_2)$ has capacity $\mu(t) \le \mu(\tau_1)$, so the components S in $\mathcal{C}_{\tau_1}$ that have weight $\mathsf{wt}(S) > \mu(\tau_1)$ remain unchanged in $\mathcal{C}$ throughout (τ_1, τ_2), and the items in them do not charge S^* during (τ_1, τ_2). So we need only consider items in components S in $\mathcal{C}_{\tau_1}$ with $\mathsf{wt}(S) \le \mu(\tau_1)$. Assume there are such components. By inspection of the algorithm, there can only be one: the component S_{τ_1} built at time τ_1. All charges in (τ_1, τ_2) come from items $x \in S_{\tau_1} \cap S^*$.

Let $\tau_1 = t'_1 < t'_2 < \cdots < t'_\ell$ be the times in $[\tau_1, \tau_2)$ when these items are put in a new component. These are the times in (τ_1, τ_2) when S^* is charged, and, at each, the charge is $\mathsf{wt}(S^* \cap S_{\tau_1}) \le \mathsf{wt}(S_{\tau_1})$, so the total charge to S^* during (τ_1, τ_2) is at most $(\ell - 1)\,\mathsf{wt}(S_{\tau_1})$.

At each time t'_i with $i \ge 2$ the previous component $S_{t'_{i-1}}$, of weight at least $\mathsf{wt}(S_{\tau_1})$, is merged. So each time t'_i has capacity $\mu(t'_i) \ge \mathsf{wt}(S_{\tau_1})$. By Observation 1.2, the difference between each time t'_i and the next t'_{i+1} is at least $\mathsf{wt}(S_{\tau_1})$. So $(\ell - 1)\,\mathsf{wt}(S_{\tau_1}) \le t'_\ell - t'_1 \le \tau_2 - \tau_1 - 1$.

By the two previous paragraphs the charge to S^* during (τ_1, τ_2) is at most $\tau_2 - \tau_1 - 1$. Summing over the dominant times τ_1 in $[t_1, t_2]$ proves the lemma. ☐

Let D be the set of dominant times. For the rest of the proof all times that we consider are dominant. Note that all times that are congested or uncongested (as defined next) are dominant.

Definition 6 (congestion)**.** *For any time $t \in D$ and component S_t, define the* congestion *of t and S_t to be $\mathsf{wt}(S_t \cap S^*)/\mu(t)$, the amount S_t charges S^*, divided by the capacity $\mu(t)$. Call t and S_t* congested *if this congestion exceeds 64, and* uncongested *otherwise.*

Lemma 1.4 (uncongested times)**.** *The total charge to S^* at uncongested times is $O(t_2 - t_1)$.*

Proof. The charge to S^* at any uncongested time t is at most $64\mu(t)$, so the total charge to $\mathcal{C}^*$ during such times is at most $64 \sum_{t \in D} \mu(t)$. By definition of *dominant*, the capacity $\mu(t)$ for each

$t \in D$ is a distinct power of 2 no larger than $t_2 - t_1 + 1$. So $\sum_{t \in D} \mu(t)$ is at most $2(t_2 - t_1 + 1)$, and the total charge to $\mathcal{C}^*$ during uncongested times is $O(t_2 - t_1)$. ☐

Lemma 1.5 (congested times)**.** *The total charge to S^* at congested times is $O(\mathsf{wt}(S^*) \log^* \Delta)$.*

Proof. Let Z denote the set of congested times. For each item $x \in S^*$, let $W(x)$ be the collection of congested components that contain x and charge S^*. The total charge to S^* at congested times is $\sum_{x \in S^*} |W(x)| \, \mathsf{wt}(x)$.

To bound this, we use a random experiment that starts by choosing a random item X in S^*, where each item x has probability proportional to $\mathsf{wt}(x)$ of being chosen: $\Pr[X = x] = \mathsf{wt}(x)/\mathsf{wt}(S^*)$.

We will show that $\mathrm{E}_X[|W(X)|]$ is $O(\log^* \Delta)$. Since $\mathrm{E}_X[|W(X)|] = \sum_{x \in S^*} |W(x)| \, \mathsf{wt}(x)/\mathsf{wt}(S^*)$, this will imply that the total charge is $O(\log^* \Delta) \, \mathsf{wt}(S^*)$, proving the lemma.

The merge forest for S^*. Define the following *merge forest*. There is a leaf $\{x\}$ for each item $x \in S^*$. There is a non-leaf node S_t for each congested component S_t. The parent of each leaf $\{x\}$ is the first congested component S_t that contains x (that is, $t = \min\{i \in Z : x \in S_i\}$), if any. The parent of each node S_t is the next congested component $S_{t'}$ that contains all items in S_t (that is, $t' = \min\{i \in Z : i > t,\, S_t \subseteq S_i\}$), if any. Parentless nodes are roots.

The random walk starts at the root of the tree that holds leaf $\{X\}$, then steps along the path to that leaf in the tree. In this way it traces (in reverse) the sequence $W(X) = \{S_i : X \in S_i\}$ of congested components that X entered during $[t_1, t_2]$. The number of steps is $|W(X)|$. To finish, we show that the expected number of steps is $O(\log^* \Delta)$.

Each non-leaf node S_t in the tree has congestion $\mathsf{wt}(S_t \cap S^*)/\mu(t)$, which is at least 64 and at most Δ. For the proof, define the congestion of each leaf x to be 2^{Δ}. To finish, we argue that *with each step of the random walk, the iterated logarithm of the current node's congestion increases in expectation by at least* $1/5$.

A step in the random walk. Fix any non-leaf node S_t. Let $\alpha_t = \mathsf{wt}(S_t \cap S^*)/\mu(t)$ be its congestion. The walk visits S_t with probability $\mathsf{wt}(S^* \cap S_t)/\mathsf{wt}(S^*)$. Condition on this event (that is, $X \in S_t$). Let random variable α' be the congestion of the child of S_t next visited.

Sublemma 1.5.1. *For any $\beta \in [\alpha_t, 2^{\Delta})$, $\Pr[\alpha' > \beta \,|\, X \in S_t]$ is at least $1 - \alpha_t^{-1}(2 + \log_2 \beta)$.*

Proof. Consider any child $S_{t'}$ of S_t with $\alpha_{t'} \le \beta$. We will bound the probability that $S_{t'}$ is visited next (i.e., $X \in S_{t'}$). Node $S_{t'}$ is not a leaf, as $\alpha_{t'} < 2^{\Delta}$. Define $j(t')$ so that its capacity $\mu(t')$ equals $\mu(t)/2^{j(t')}$. (That is, $j(t') = \log_2(\mu(t)/\mu(t'))$.) The definitions and $\alpha_{t'} \le \beta$ imply

$$\Pr[X \in S_{t'} \,|\, X \in S_t] \;=\; \frac{\mathsf{wt}(S_{t'} \cap S^*)}{\mathsf{wt}(S_t \cap S^*)} \;=\; \frac{\alpha_{t'}\, \mu(t')}{\alpha_t\, \mu(t)} \;\le\; \frac{\beta\, \mu(t)/2^{j(t')}}{\alpha_t\, \mu(t)} \;=\; \frac{\beta}{\alpha_t\, 2^{j(t')}}. \tag{1}$$

Also, the algorithm merged a component containing $S_{t'}$ at time t, so $\mathsf{wt}(S_{t'}) \le \mu(t)$, so

$$\Pr[X \in S_{t'} \,|\, X \in S_t] \;=\; \frac{\mathsf{wt}(S_{t'} \cap S^*)}{\mathsf{wt}(S_t \cap S^*)} \;=\; \frac{\mathsf{wt}(S_{t'} \cap S^*)}{\alpha_t\, \mu(t)} \;\le\; \frac{\mathsf{wt}(S_{t'})}{\alpha_t\, \mu(t)} \;\le\; \frac{1}{\alpha_t}. \tag{2}$$

Combining Bounds (1) and (2), $\Pr[X \in S_{t'} | X \in S_t]$ is at most $\alpha_t^{-1} \min(1, \beta\, 2^{-j(t')})$. Summing this bound over all children $S_{t'}$ of S_t with congestion $\alpha_{t'} \le \beta$, and using that each $j(t')$ is a distinct positive integer, the probability that $\alpha' \le \beta$ is at most

$$\alpha_t^{-1} \sum_{j=1}^{\infty} \min(1, \beta\, 2^{-j}) \le \alpha_t^{-1} \int_0^{\infty} \min(1, \beta\, 2^{-j})\, dj = \alpha_t^{-1}(\log_2(\beta) + 1/\ln 2)$$

(splitting the integral at $j = \log_2 \beta$). The sublemma follows from $1/\ln 2 \le 2$. $\square$

Next we lower-bound the expected increase in the $\log^*$ of the congestion in this step. We use $\sqrt{2}$ as the base of the iterated log.[6] Then $\log^*(2^{\alpha_t/2}) = 1 + \log^* \alpha_t$, so, conditioned on $X \in S_t$,

$$\mathrm{E}[\log^* \alpha'] \;\ge\; \Pr[\alpha' \ge \alpha_t] \log^* \alpha_t \;+\; \Pr[\alpha' \ge 2^{\alpha_t/2}].$$

Bounding the two probabilities above via Sublemma 1.5.1 with $\beta = \alpha_t$ and $\beta = 2^{\alpha_t/2}$, the right-hand side above is

$$\begin{aligned}
&\ge [1 - \alpha_t^{-1}(2 + \log_2 \alpha_t)] \log^* \alpha_t \;+\; [1 - \alpha_t^{-1}(2 + \alpha_t/2)] \\
&= \log^*(\alpha_t) + 1/2 - [2 + (2 + \log_2 \alpha_t) \log^* \alpha_t]/\alpha_t \\
&\ge \log^*(\alpha_t) + 1/2 - 3/10 \;=\; \log^*(\alpha_t) + 1/5,
\end{aligned}$$

using in the last inequality that $\alpha_t \ge 64$ (t is congested). It follows that $E[\log^* \alpha' - \log^* \alpha_t \,|\, X \in S_t] \ge 1/5$. That is, in each step, the expected increase in the iterated logarithm of the congestion is at least $1/5$.

Let random variable $L = |W(X)|$ be the length of the random walk. Let random variable α'_i be the congestion of the ith node on the walk. By the previous section, for each i, given that $i < L$, $\mathrm{E}[\log^* \alpha'_{i+1} - \log^* \alpha'_i \,|\, \alpha'_i] \ge 1/5$. It follows by Wald's equation that $\mathrm{E}[\log^* \alpha'_L - \log^* \alpha'_1] \ge \mathrm{E}[L]/5$. Since $\alpha'_L = 2^\Delta$ and $\log^* \alpha'_1 \ge 0$, we have $\mathrm{E}[\log^* \alpha'_L - \log^* \alpha'_1] \le \log^* 2^\Delta$. It follows that $\mathrm{E}[L] \le 5 \log^* 2^\Delta \le 10 + 5 \log^* \Delta$. That is, the expected length of the random walk is $O(\log^* \Delta)$. By the discussion at the start of the proof, this implies the lemma. $\square$

To recap, for each component S_t built by the algorithm, the (over-counted) build cost is charged item by item to those components in the optimal solution $\mathcal{C}^*$ that currently contain the item. In this way, the algorithm's total over-counted build cost $\sum_t \mathsf{wt}(S_t)$ is charged to components in $\mathcal{C}^*$. By Lemmas 1.3–1.5, each component S^* in the optimal solution $\mathcal{C}^*$ is charged $O(1)$ times its contribution $t_2 - t_1$ to the query cost of $\mathcal{C}^*$ plus (in expectation) $O(\log^* m)$ times its contribution $\mathsf{wt}(S^*)$ to the build cost of $\mathcal{C}^*$. It follows that the expected build cost incurred by the algorithm is $O(\log^* m)$ times the cost of $\mathcal{C}^*$.

By Lemma 1.1, the total *query* cost incurred by the algorithm is at most twice the algorithm's over-counted build cost plus the cost of $\mathcal{C}^*$. It follows that the total (build and query) cost incurred by the algorithm is $O(\log^*(m))$ times the cost of $\mathcal{C}^*$. That is, the competitive ratio is $O(\log^* m)$, proving Part (i) of Theorem 1.[7]

2.2 Part (ii): the competitive ratio is $\Omega(\log^* m)$

Lemma 1.6. *The competitive ratio of the Adaptive-Binary algorithm (Figure 3) is* $\Omega(\log^* m)$.

Proof. We will show a ratio of $\Omega(\log^* m)$ on a particular class of inputs, one for each integer $D \ge 0$. (Figure 4 describes the input I for $D = 2$ and the resulting merge tree, of depth $D + 1$.)

The desired merge tree. For reference, define an infinite rooted tree T_∞ with node set $\{1, 2, 3, \ldots\}$ by the iterative process shown in Figure 9. Each iteration i defines the children of node i. Node i has $2^{i-p(i)}$ children, allocated greedily from the "next available" nodes, so that each node $i \ge 2$ is

[6] Defined by $\log^*_{\sqrt{2}} \alpha_t = 0$ if $\alpha_t \le 8$, else $1 + \log^*_{\sqrt{2}}(\log_{\sqrt{2}} \alpha_t)$. Note that $\log^*_{\sqrt{2}} \alpha_t = \Theta(\log^*_e \alpha_t)$.

[7] Curiously, the algorithm's cost is in fact $O(1)$ times the query cost of $\mathcal{C}^*$ plus $O(\log^* m)$ times its build cost.

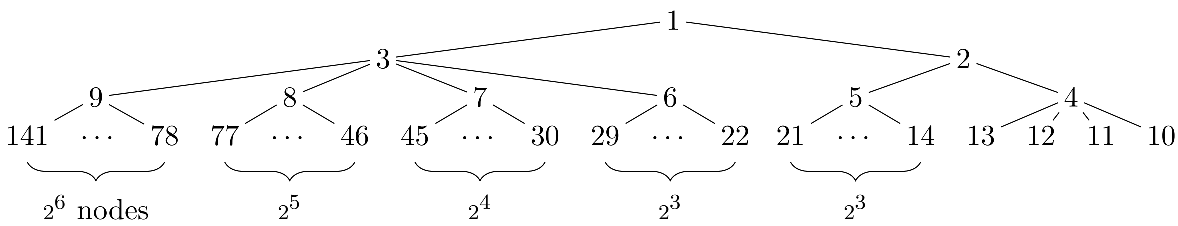

Figure 8: The top three levels of T_∞. Each node i has $2^{i-p(i)}$ children, where $p(i)$ is the parent of i (exc. $p(1)=0$). The merge tree T_2^N (Figure 4) consists of these three levels, with each node i given weight $2^{N-p(i)}$, so the nodes with weight 2^{N-i} are the $2^{i-p(i)}$ children of node i, and their total weight equals the weight of node i. Note that the merge tree of Figure 4 is T_2^{18}.

1. make node 1 the root
2. for $i \leftarrow 1, 2, 3, \ldots$ do:
2.1. let $p(i)$ be the parent of i, except $p(1) = 0$
2.2. give node i the $2^{i-p(i)}$ children

$$\big\{c(i-1) + j \,:\, 1 \le j \le 2^{i-p(i)}\big\},$$

where $c(i-1)$ is the max child of $i-1$, except $c(0) = 1$

Figure 9: An algorithm defining the tree T_∞, with nodes $\{1, 2, 3, \ldots\}$.

given exactly one parent. The depth of i is non-decreasing with i.[8] Figure 8 shows the top three levels of T_∞.

Let n_d be the number of nodes of depth d or less in T_∞. Each such node i satisfies $i \le n_d$ (as depth is non-decreasing with i), so, inspecting Line 2.2, node i has at most $2^i \le 2^{n_d}$ children. Each node of depth $d+1$ or less is either the root or a child of a node of depth d or less, so $n_{d+1} \le 1 + n_d 2^{n_d} \le 2^{2^{n_d}}$. Taking the $\log^*$ of both sides gives $\log^* n_{d+1} \le 2 + \log^* n_d$. Inductively, $\log^* n_d \le 2d$ for each d.

Now fix an integer $D \ge 0$. Define the desired merge tree, T_D^N, to be the subtree of T_∞ induced by the nodes of depth at most $D+1$. Let m be the number of leaves in T_D^N. By the previous paragraph (and $m \le n_{D+1}$), *every leaf in T_D^N has depth $\Omega(\log^* m)$.*

Assign weights to the nodes in T_D^N as follows. Fix $N = 2\,n_D$. Give each node i weight $2^{N-p(i)}$, where $p(i)$ is the parent of i (except $p(1) = 0$). Each weight is a power of two, and the nodes of any given weight 2^{N-i} are exactly the $2^{i-p(i)}$ children of node i. The weight of each parent i equals the total weight of its children.

The input. Define the input I as follows. For each time $t \in \{1, 2, \ldots, m\}$, insert a set I_t containing just one item whose weight equals the weight of the tth leaf of T_D^N. Then, at each time $t \in \{m+1, m+2, \ldots, 2^{N-1}\}$, insert an empty set $I_t = \emptyset$.

No merges until last non-empty insertion. The algorithm does no merges before time $\min_{i=1}^m \mathsf{wt}(I_i)$, which is the minimum leaf weight in T_D^N. The lightest leaves are the children of

[8] This follows by induction: Line 2.2 ensures that $p(i') \le p(i)$ for $i' < i$, so inductively $\mathrm{depth}(i') = 1 + \mathrm{depth}(p(i')) \le 1 + \mathrm{depth}(p(i)) = \mathrm{depth}(i)$.

node n_D, of weight 2^{N-n_D}. Since the total leaf weight is the weight of the root, 2^N, it follows that $m2^{N-n_D} \le 2^N$, that is, $m \le 2^{n_D} = 2^{N-n_D}$ (using $N = 2\,n_D$). So, *the algorithm does no merges until time $t(n_D) = 2^{N-n_D}$ (after all non-empty insertions).*

The algorithm's merge tree matches T_D^N. By the previous two paragraphs, just before time $t(n_D) = 2^{N-n_D}$ the algorithm's cover *matches* the leaves of T_D^N, meaning that the cover's components correspond to the leaves, with each component weighing the same as its corresponding node. The leaves are $\{j : p(j) \le n_d < j\}$. So the following invariant holds initially, for $i = n_D$:

For each $i \in \{n_D, n_D - 1, \ldots, 2, 1\}$, just before time $t(i) = 2^{N-i}$, the algorithm's cover $\mathcal{C}_{t(i)}$ matches the nodes in Q_i, defined as

$$Q_i \doteq \{j : 2^{N-j} < t(i) \le 2^{N-p(j)}\} = \{j : p(j) \le i < j\}.$$

Informally, these are the nodes j that have not yet been merged by time $t(i)$, because their weight $2^{N-p(j)}$ is at least $t(i)$, but whose children (the nodes of weight 2^{N-j}) if any, have already been merged.

Assume the invariant holds for a given i. We show it holds for $i-1$. At time $t(i)$, the algorithm merges the components of weight at most $\mu(t(i)) = t(i) = 2^{N-i}$ in its cover. By the invariant, these are the components of weight $t(i) = 2^{N-i}$, corresponding to the children of node i (which are all in Q_i). They leave the cover and are replaced by their union, whose weight equals $2^{N-p(i)}$. Likewise, by the definition (and $p(j) < j$)

$$Q_{i-1} = \{i\} \cup Q_i \setminus \{j : p(j) = i\},$$

so the resulting cover matches Q_{i-1}, with the new component corresponding to node i. The minimum-weight nodes in Q_{i-1} are then $\{j : p(j) = i-1\}$, the children of node $i-1$. These have weight $2^{N-(i-1)} = t(i-1)$, so the algorithm keeps this cover until just before time $t(i-1)$, so that the invariant is maintained for $i-1$.

Inductively the invariant holds for $i = 1$: just before time $t(1) = 2^{N-1} = n$, the algorithm's cover contains the components corresponding to $\{j : p(j) = 1 < j\}$, with weight $2^{N-p(j)} = 2^{N-1} = n$. At time n they are merged form the final component of weight 2^N, corresponding to the root node 1. So the algorithm's merge tree matches T_D^N.

Competitive ratio. Each leaf in the merge tree has depth $\Omega(\log^* m)$, so every item is merged $\Omega(\log^* m)$ times, and the algorithm's build cost is $\Omega(\mathsf{wt}(1)\log^* m) = \Omega(n \log^* m)$ (using $\mathsf{wt}(1) = 2n$).

But the optimal cost is $\Theta(n)$. (Consider the solution that merges all input sets into one component at time m, just after all non-empty insertions. Its query cost is $\sum_{t=1}^{m-1} t + \sum_{t=m}^{n} 1 = O(m^2 + n)$. Its merge cost is $2\,\mathsf{wt}(1) = O(n)$. Recalling that $m \le 2^{n_D} = 2^{N/2} = O(\sqrt{n})$, the optimal cost is $O(n)$.)

So the competitive ratio is $\Omega(\log^* m)$. $\square$

Note that in Lemma 1.6, $n \approx m^2$, so $\log^* m = \Omega(\log^* n)$. The upper bound in Section 2.1 and the lower bound in Lemma 1.6 prove Theorem 1.

3 K-Component Dynamization and variants (Theorems 2–6)

3.1 Lower bound on optimal competitive ratio

Theorem 2. *For k-Component Dynamization (and consequently for its generalizations) no deterministic online algorithm has ratio competitive ratio less than k.*

Before we give the proof, here is a proof sketch for $k = 2$. The adversary begins by inserting one item of weight 1 and one item of infinitesimal weight $\varepsilon > 0$, followed by a sequence of $n - 2$ weight-zero items just until the algorithm's cover has just one component. (This must happen, or the competitive ratio is unbounded — OPT pays only at time 1, while the algorithm continues to pay at least ε each time step.) By calculation the algorithm pays at least $2 + (n - 1)\varepsilon$, while OPT pays $\min(2 + \varepsilon, 1 + (n - 1)\varepsilon)$, giving a ratio of $1.5 - O(\varepsilon)$.

This lower bound does not reach 2 (in contrast to the standard "rent-or-buy" lower bound) because the algorithm and OPT both pay a "setup cost" of 1 at time 1. However, at the end of sequence, the algorithm and OPT are left with a component of weight ~ 1 in place. The adversary can now continue, doing a second phase without the setup cost, by inserting an item of weight $\sqrt{\varepsilon}$, then zeros just until the algorithm's cover has just one component (again this must happen or the ratio is unbounded). Let m be the length of this second phase. By calculation, for *this* phase, the algorithm pays at least $(m - 1)\sqrt{\varepsilon} + 1$ while OPT pays at most $\min(1 + \sqrt{\varepsilon} + \varepsilon, (m - 1)(\sqrt{\varepsilon} + \varepsilon))$, giving a ratio of $2 - O(\sqrt{\varepsilon})$ for just the phase.

The ratio of the whole sequence (both phases together) is now $1.75 - O(\sqrt{\varepsilon})$. By doing additional phases (using infinitesimal $\varepsilon^{1/i}$ in the ith phase), the adversary can drive the ratio arbitrarily close to 2. The proof generalizes this idea.

Proof of Theorem 2. Fix an arbitrarily small $\varepsilon > 0$. Define $k + 1$ sequences of items (weights) as follows. Sequence $\sigma(k + 1)$ has just one item, $\sigma_1(k + 1) = \varepsilon$. For $j \in \{k, k - 1, \ldots, 1\}$, in decreasing order, define sequence $\sigma(j)$ to have $n_j = \lceil k/\sigma_1(j + 1)\rceil$ items, with the ith item being $\sigma_i(j) = \varepsilon^{n_k + n_{k-1} + \cdots + n_j - i + 2}$. Each sequence $\sigma(j)$ is strictly increasing, and all items in $\sigma(j)$ are smaller than all items in $\sigma(j + 1)$. Every two items differ by a factor of at least $1/\varepsilon$, so the cost to build any component will be at most $1/(1 - \varepsilon)$ times the largest item in the component.

Adversarial input sequence I. Fix any deterministic online algorithm A. Define the input sequence I to interleave the $k + 1$ sequences in $\{\sigma(j) : 1 \leq j \leq k + 1\}$ as follows. Start by inserting the only item from sequence $\sigma(k + 1)$: take $I_1 = \{\sigma_1(k + 1)\} = \{\varepsilon\}$. For each time $t \geq 1$, after A responds to the insertion at time t, determine the next insertion $I_{t+1} = \{x\}$ as follows. For each sequence $\sigma(j)$, call the most recent (and largest) item inserted so far from $\sigma(j)$, if any, the *representative* of the sequence. Define index $\ell(t)$ so that the largest representative in any new component at time t is the representative of $\sigma(\ell(t))$. (The item inserted at time t is necessarily a representative and in at least one new component, so $\ell(t)$ is well-defined.) At time $t + 1$ choose the inserted item x to be the next unused item from sequence $\sigma(\ell(t) - 1)$. Define the *parent* of x, denoted $p(x)$, to be the representative of $\sigma(\ell(t))$ at time t. (Note: A's build cost at time t was at least $p(x) \gg x$.) Stop when the cumulative cost paid by A reaches k. This defines the input sequence I.

The input I is well-defined. Next we verify that I is well-defined, that is, that *(a)* $\ell(t) \neq 1$ for all t (so x's specified sequence $\sigma(\ell(t) - 1)$ exists) and *(b)* each sequence $\sigma(j)$ is chosen at most n_j times. First we verify *(a)*. Choosing x as described above forces the algorithm to maintain the following invariants at each time t: *(i) each of the sequences in $\{\sigma(j) : \ell(t) \leq j \leq k + 1\}$ has a representative, and (ii) no two of these $k - \ell(t) + 2$ representatives are in any one component.* (Indeed, the invariants hold at time $t = 1$ when $\ell(t) = k + 1$. Assume they hold at some time t. At time $t + 1$ the newly inserted element x is the new representative of $\sigma(\ell(t) - 1)$ and is in some new component, so $\ell(t + 1) \geq \ell(t) - 1$. These facts imply that Invariant (i) is maintained. By the definition of $\ell(t + 1)$, the component(s) built at time $t + 1$ contain the representative from $\sigma(\ell(t + 1))$ but no representative from any $\sigma(j)$ with $j > \ell(t + 1)$. This and $\ell(t + 1) \geq \ell(t) - 1$

imply that Invariant (ii) is maintained.) By inspection, Invariants (i) and (ii) imply that A has at least $k - \ell(t) + 2$ components at time t. But A has at most k components, so $\ell(t) \geq 2$.

Next we verify *(b)*, that I takes at most n_j items from each sequence $\sigma(j)$. This holds for $\sigma(k+1)$ just because, by definition, after time 1, I cannot insert an item from $\sigma(k+1)$. Consider any $\sigma(j)$ with $j \leq k$. For each item $\sigma_i(j)$ in $\sigma(j)$, when I inserted $\sigma_i(j)$, algorithm A paid at least $p(\sigma_i(j)) \geq \sigma_1(j+1)$ at the previous time step. So, before all n_j items from $\sigma(j)$ are inserted, A must pay at least $n_j\,\sigma_1(j+1) \geq k$ (by the definition of n_j), and the input stops. It follows that I is well-defined.

Upper-bound on optimum cost. Next we upper-bound the optimum cost for I. For each $j \in \{1, \ldots, k\}$, define $\mathcal{C}(j)$ to be the solution for I that partitions the items inserted so far into the following k components: one component containing items from $\sigma(j)$ and $\sigma(j+1)$, and, for each $h \in \{1, \ldots, k+1\} \setminus \{j, j+1\}$, one containing items from $\sigma(h)$.

To bound $\mathsf{cost}(\mathcal{C}(j))$, i.e., the total cost of new components in $\mathcal{C}(j)$, first consider the new components such that the largest item in the new component is the just-inserted item, say, x. The cost of such a component is at most $x/(1-\varepsilon)$. Each item x is inserted at most once, so the total cost of all such components is at most $1/(1-\varepsilon)$ times the sum of all defined items, and therefore at most $\sum_{i=1}^{\infty} \varepsilon^i/(1-\varepsilon) = \varepsilon/(1-\varepsilon)^2$. For every other new component, the just-inserted item x must be from sequence $\sigma(j+1)$, so the largest item in the component is the parent $p(x)$ (in $\sigma(j)$) and the build cost is at most $p(x)/(1-\varepsilon)$. Defining $m_j \leq n_j$ to be the number of items inserted from $\sigma(j)$, the total cost of building all such components is at most $\sum_{i=1}^{m_j} p(\sigma_i(j))/(1-\varepsilon)$. So $\mathsf{cost}(\mathcal{C}(j))$ is at most $\varepsilon/(1-\varepsilon)^2 + \sum_{i=1}^{m_j} p(\sigma_i(j))/(1-\varepsilon)$.

The cost of OPT is at most $\min_j \mathsf{cost}(\mathcal{C}(j))$. The minimum is at most the average, so

$$(1-\varepsilon)^2\,\mathsf{cost}(\mathsf{OPT}) \;\leq\; \min_{j=1,\ldots,k} \varepsilon + \sum_{i=1}^{m_j} p(\sigma_i(j)) \;\leq\; \varepsilon + \frac{1}{k}\sum_{j=1}^{k}\sum_{i=1}^{m_j} p(\sigma_i(j)).$$

Lower bound on algorithm cost. The right-hand side of the above inequality is at most $(\varepsilon/k + 1/k)\,\mathsf{cost}(\mathsf{A})$, because $\mathsf{cost}(\mathsf{A}) \geq k$ (by the stopping condition) and $\sum_{j=1}^{k}\sum_{i=1}^{m_j} p(\sigma_i(j)) \leq \mathsf{cost}(\mathsf{A})$. (Indeed, for each $j \in \{1, \ldots, k\}$ and $i \in \{1, \ldots, m_j\}$, the item $\sigma_i(j)$ was inserted at some time $t \geq 2$, and A paid at least $p(\sigma_i(j))$ at the previous time $t-1$.) So the competitive ratio is at least $(1-\varepsilon)^2/(\varepsilon/k + 1/k) \geq (1-3\varepsilon)k$. This holds for all $\varepsilon > 0$, so the ratio is at least k. □

3.2 Upper bound for decreasing weights

Theorem 3. *For k-Component Dynamization with decreasing weights (and plain k-Component Dynamization) the deterministic online algorithm in Figure 5 is k-competitive.*

Proof. Consider any execution of the algorithm on any input $I_1, I_2, \ldots, I_n$. Let δ_t be such that each component's credit increases by δ_t at time t. (If Block 2.2 is executed, $\delta_t = 0$.) To prove the theorem we show the following lemmas.

Lemma 3.1. *The cost incurred by the algorithm is at most* $k\sum_{t=1}^{n} \mathsf{wt}_t(I_t) + \delta_t$.

Lemma 3.2. *The cost incurred by the optimal solution is at least* $\sum_{t=1}^{n} \mathsf{wt}_t(I_t) + \delta_t$.

Proof of Lemma 3.1. As the algorithm executes, keep the components ordered by age, oldest first. Assign each component a *rank* equal to its rank in this ordering. Say that the rank of any *item* is the rank of its current component, or $k+1$ if the item is not yet in any component. At each time

t, when a new component is created in Line 2.1.3, the ranks of the items in S_0 stay the same, but the ranks of all other items decrease by at least 1. Divide the cost of the new component into two parts: the contribution from the items that decrease in rank, and the remaining cost.

Throughout the execution of the algorithm, each item's rank can decrease at most k times, so the total contribution from items as their ranks decrease is at most $k\sum_{t=1}^{n} \mathsf{wt}_t(I_t)$ (using here that the weights are non-increasing with time). To complete the proof of the lemma, observe that the remaining cost is the sum, over times t when Line 2.1.3 is executed, of the weight $\mathsf{wt}_t(S_0)$ of the component S_0 at time t. This sum is at most the total credit created, because, when a component S_0 is destroyed in Line 2.1.3, at least the same amount of credit (on S_0) is also destroyed. But the total credit created is $k\sum_{t=1}^{n} \delta_t$, because when Line 2.1.1 executes it increases the total component credit by $k\delta_t$. □

Proof of Lemma 3.2. Let $\mathcal{C}^*$ be an optimal solution. Let $\mathcal{C}$ denote the algorithm's solution. At each time t, when the algorithm executes Line 2.1.1, it increases the credit of each of its k components in $\mathcal{C}_{t-1}$ by δ_t. So *the total credit the algorithm gives is* $k\sum_t \delta_t$.

For each component $S \in \mathcal{C}_{t-1}$, think of the credit given to S as being distributed over the component's items $x \in S$ in proportion to their weights, $\mathsf{wt}_t(x)$: at time t, each item $x \in S$ receives credit $\delta_t \, \mathsf{wt}_t(x)/\,\mathsf{wt}_t(S)$. Have each x, in turn, charge this amount to one component in OPT's current cover $\mathcal{C}^*_t$ that contains x. In this way, *the entire credit* $k\sum_{t=1}^{n} \delta_t$ *is charged to components in* $\mathcal{C}^*$.

Sublemma 3.2.1. *Let x be any item. Let $[t, t']$ be any time interval throughout which x remains in the same component in $\mathcal{C}$. The cumulative credit given to x during $[t, t']$ is at most $\mathsf{wt}_t(x)$.*

Proof. Let S be the component in $\mathcal{C}$ that contains x throughout $[t, t']$. Assume that $\delta_{t'} > 0$ (otherwise reduce t' by one). Let $\mathsf{credit}_{t'}[S]$ denote $\mathsf{credit}[S]$ at the end of iteration t'. Weights are non-increasing with time, so the credit that x receives during $[t, t']$ is

$$\sum_{i=t}^{t'} \frac{\mathsf{wt}_i(x)}{\mathsf{wt}_i(S)}\delta_i \;\le\; \frac{\mathsf{wt}_t(x)}{\mathsf{wt}_{t'}(S)} \sum_{i=t}^{t'} \delta_i \;\le\; \frac{\mathsf{wt}_t(x)}{\mathsf{wt}_{t'}(S)}\, \mathsf{credit}_{t'}[S].$$

The right-hand side is at most $\mathsf{wt}_t(x)$, because $\delta_{t'} > 0$ so by inspection of Block 2.1 $\mathsf{credit}_{t'}[S] \le \mathsf{wt}_{t'}(S)$. □

Next we bound how much charge OPT's components (in $\mathcal{C}^*$) receive. For any time t, let $\mathcal{N}^*_t = \mathcal{C}^*_t \setminus \mathcal{C}^*_{t-1}$ contain the components that OPT creates at time t, and let $N^*_t = \bigcup_{S \in \mathcal{N}^*_t} S$ contain the items in these components. Call the charges received by components in $\mathcal{N}^*_t$ from components created by the algorithm before time t *forward charges*. Call the remaining charges (from components created by the algorithm at time t or after) *backward charges*.

Consider first the *backward* charges to components in $\mathcal{N}^*_t$. These charges come from components in $\mathcal{C}_{t-1}$, via items x in $N^*_t \cap U_{t-1}$, from time t until the algorithm destroys the component in $\mathcal{C}_{t-1}$ that contains x. By Lemma 3.2.1, the total charge via a given x from time t until its component is destroyed is at most $\mathsf{wt}_t(x)$, so the cumulative charge to components in $\mathcal{N}^*_t$ from older components is at most $\mathsf{wt}_t(N^*_t \cap U_{t-1}) = \mathsf{wt}_t(N^*_t) - \mathsf{wt}_t(I_t)$ (using that $N^*_t \setminus U_{t-1} = I_t$). Using that OPT pays at least $\mathsf{wt}_t(N^*_t)$ at time t, and summing over t, *the sum of all backward charges is at most* $\mathsf{cost}(\mathsf{OPT}) - \sum_t \mathsf{wt}_t(I_t)$.

Next consider the *forward* charges, from components created at time t or later, to any component S^* in $\mathcal{N}^*_t$. Component S^* receives no forward charges at time t, because components created by the algorithm at time t receive no credit at time t. Consider the forward charges S^* receives at any time $t' \geq t+1$. At most one component (in $\mathcal{C}_{t'-1}$) can contain items in N_t^*, namely, the component in $\mathcal{C}_{t'-1}$ that contains I_t. (Indeed, the algorithm merges components "newest first", so any other component in $\mathcal{C}_{t'-1}$ created after time t only contains items inserted after time t, none of which are in N_t^*.) At time t', the credit given to that component is $\delta_{t'}$, so the components created by the algorithm at time t' charge a total of at most $\delta_{t'}$ to S^*. Let $m(t,t') = |\mathcal{N}_t^* \cap \mathcal{C}_{t'}^*|$ be the number of components S^* that OPT created at time t that remain at time t'. Summing over $t' \geq t+1$ and $S^* \in \mathcal{N}_t^*$, the forward charges to components in $\mathcal{N}_t^*$ total at most $\sum_{t'=t+1}^{n} m(t,t')\delta_{t'}$. Summing over t, *the sum of all forward charges is at most*

$$\sum_{t=1}^{n} \sum_{t'=t+1}^{n} m(t,t')\delta_{t'} \;=\; \sum_{t'=2}^{n} \delta_{t'} \sum_{t=1}^{t'-1} m(t,t') \;\leq\; \sum_{t'=1}^{n} \delta_{t'}(k-1)$$

(using that $\sum_{t=1}^{t'-1} m(t,t') \leq k-1$ for all t, because OPT has at most k components at time t', at least one of which is created at time t').

Recall that the entire credit $k\sum_{t=1}^{n} \delta_t$ is charged to components in $\mathcal{C}^*$. Summing the bounds from the two previous paragraphs on the (forward and backward) charges, this implies that

$$k\sum_{t=1}^{n} \delta_t \leq \mathsf{cost}(\mathsf{OPT}) - \sum_{t=1}^{n} \mathsf{wt}_t(I_t) + (k-1)\sum_{t=1}^{n} \delta_t.$$

This proves the lemma, as it is equivalent to the desired bound $\mathsf{cost}(\mathsf{OPT}) \geq \sum_{t=1}^{n} \mathsf{wt}_t(I_t) + \delta_t$. ☐

This proves Theorem 3. ☐

3.3 Bootstrapping newest-first algorithms

Theorem 4. *Any newest-first online algorithm for k-Component (or Min-Sum) dynamization with decreasing weights can be converted into an equally competitive algorithm for the LSM variant.*

Proof. Fix an instance (I, wt) of LSM k-Component (or Min-Sum) Dynamization. For any solution $\mathcal{C}$ to this instance, let $\mathsf{wt}(\mathcal{C})$ denote its build cost using build-cost function wt. For any set S of items and any item $x \in S$, let $\mathsf{nr}(x,S)$ be 0 if x is redundant in S (that is, there exists a newer item in S with the same key) and 1 otherwise. Then $\mathsf{wt}_t(S) = \sum_{x\in S} \mathsf{nr}(x,S)\,\mathsf{wt}_t(\{x\})$. (Recall that $\mathsf{wt}_t(\{x\})$ is $\mathsf{wt}(x)$ unless x is expired, in which case $\mathsf{wt}_t(x)$ is the tombstone weight of x.)

For any time t and item $x \in U_t$, define $\mathsf{wt}'_t(x) = \mathsf{nr}(x,U_t)\,\mathsf{wt}_t(\{x\})$. For any item x, $\mathsf{wt}'_t(x)$ is non-increasing with t, so (I, wt') is an instance of k-Component Dynamization with decreasing weights. For any solution $\mathcal{C}$ for this instance, let $\mathsf{wt}'(\mathcal{C})$ denote its build cost using build-cost function wt'.

Lemma 4.1. *For any time t and set $S \subseteq U_t$, we have $\mathsf{wt}'_t(S) \leq \mathsf{wt}_t(S)$.*

Proof. Redundant items in S are redundant in U_t, so

$$\mathsf{wt}'_t(S) = \sum_{x\in S} \mathsf{wt}'_t(x) = \sum_{x\in S} \mathsf{nr}(x,U_t)\,\mathsf{wt}_t(\{x\}) \leq \sum_{x\in S} \mathsf{nr}(x,S)\,\mathsf{wt}_t(\{x\}) = \mathsf{wt}_t(S). \tag{3}$$

☐

Lemma 4.2. *Let $\mathcal{C}$ be any newest-first solution for (I, wt') and (I, wt). Then $\mathsf{wt}'(\mathcal{C}) = \mathsf{wt}(\mathcal{C})$.*

Proof. Consider any time t with $I_t \neq \emptyset$. Let S be $\mathcal{C}$'s new component at time t (so $\mathcal{C}_t \setminus \mathcal{C}_{t-1} = \{S\}$). Consider any item $x \in S$. Because $\mathcal{C}$ is newest-first, S includes all items inserted with or after x. So x is redundant in U_t iff x is redundant in S, that is, $\mathsf{nr}(x, U_t) = \mathsf{nr}(x, S)$, so $\mathsf{wt}'_t(S) = \mathsf{wt}_t(S)$ (because Bound (3) above holds with equality). Summing over all t gives $\mathsf{wt}'(\mathcal{C}) = \mathsf{wt}(\mathcal{C})$. ◻

Given an instance (I, wt) of LSM k-Component Dynamization, the algorithm A' simulates A on the instance (I, wt') defined above. Using Lemma 4.2, that A is c-competitive, and $\mathsf{wt}'(\mathsf{OPT}(I, \mathsf{wt}')) \leq \mathsf{wt}(\mathsf{OPT}(I, \mathsf{wt}))$ (by Lemma 4.1), we get

$$\mathsf{wt}(\mathsf{A}'(I, \mathsf{wt})) = \mathsf{wt}'(\mathsf{A}(I, \mathsf{wt}')) \leq c\,\mathsf{wt}'(\mathsf{OPT}(I, \mathsf{wt}')) \leq c\,\mathsf{wt}(\mathsf{OPT}(I, \mathsf{wt})).$$

So A is c-competitive. ◻

When applying Theorem 4, we can use that, for any time t and $S \subseteq U_t$, $\mathsf{wt}'_t(S) = \mathsf{wt}_t(S') - \mathsf{wt}_t(S' \setminus S)$ for any $S' \subseteq U_t$ such that, for all $x \in S$, item x and every newer item in U_t are in S'.

Combined with the newest-first algorithm Greedy-Dual (Figure 5), Theorems 3 and 4 yield a k-competitive algorithm for LSM k-Component Dynamization:

Corollary 5. *The online algorithm for LSM k-Component Dynamization described in the caption of Figure 5 has competitive ratio k.*

3.4 Upper bound for general variant

Theorem 6. *For general k-Component Dynamization, the deterministic online algorithm B_k in Figure 6 is k-competitive.*

Proof. The proof is by induction on k. For $k = 1$, Algorithm B_1 is 1-competitive (optimal) because there is only one solution for any instance. Consider any $k \geq 2$, and assume inductively that B_{k-1} is $(k-1)$-competitive. Fix any input (I, w) with $I = (I_1, \ldots, I_n)$. Let OPT_k denote the optimal (offline) algorithm, and let $\mathcal{C}^* = \mathsf{OPT}_k(I_1, \ldots, I_n)$ be an optimal solution for I.

Let $\mathcal{N}^*_t = \mathcal{C}^*_t \setminus \mathcal{C}^*_{t-1}$ denote OPT's new components at time t. Let $\Delta_a^b\,\mathsf{OPT}_k$ denote the cost incurred by OPT_k during $[a, b]$, that is, $\sum_{i=a}^{b} \sum_{S \in \mathcal{N}^*_i} w_i(S)$. Likewise, let $\Delta_a^b\,\mathsf{B}_k$ denote the cost incurred by B_k during $[a, b]$. Let $\mathcal{I}_a^b = (I_a, I_{a+1}, \ldots, I_b)$ denote the subproblem formed by the insertions during $[a, b]$, with build-costs inherited from w.

Recall that B_k partitions the input sequence into *phases*, each of which (except possibly the last) ends with B_k doing a *full merge* (i.e., at a time t with $|\mathcal{C}_t| = 1$). Assume without loss of generality that B_k does end the last phase with a full merge. (Otherwise, append a final empty insertion at time $n+1$ and define $w_{n+1}(U_{n+1}) = 0$. This does not increase the optimal cost, and causes the algorithm to do a full merge at time $n+1$ unless its total cost in the phase is zero.) Consider any phase. Now fix a and b to be the first and last time steps during the phase. To prove the theorem, we show $\Delta_a^b\,\mathsf{B}_k \leq k\,\Delta_a^b\,\mathsf{OPT}_k$. (The theorem follows by summing over the phases.)

The proof is via a series of lemmas. Recall that U_t denotes $\bigcup_{i=1}^{t} I_i$.

Lemma 6.1. *For any $j \in [a, b]$,* $\mathsf{cost}(\mathsf{B}_{k-1}(\mathcal{I}_a^j)) \leq (k-1)\,\mathsf{cost}(\mathsf{OPT}_{k-1}(\mathcal{I}_a^j))$.

Proof. By the inductive assumption, B_{k-1} is $(k-1)$-competitive for $\mathcal{I}_a^j$. (We use here that the instance (I, w) obeys Restrictions **(R1)**–**(R3)**, so $\mathcal{I}_a^j$ does also, so $\mathcal{I}_a^j$ is also a valid instance of general $(k-1)$-component Dynamization.) ◻

For $j \in [a, b]$, say that OPT *rebuilds by time j* if $U_{a-1} \subseteq \bigcup_{i=a}^{j} \bigcup_{S \in \mathcal{N}_j} S$. That is, every element inserted before time a is in some new component during $[a, j]$. (Equivalently, $\bigcup_{i=a}^{j} \bigcup_{S \in \mathcal{N}_j} S = U_j$.)

Lemma 6.2. *Suppose* OPT *rebuilds by time* j*. Then* $\Delta_a^j \mathsf{OPT}_k \geq w_j(U_j)$*.*

Proof.

$$\begin{aligned}
w_j(U_j) &= w_j\big(\textstyle\bigcup_{i=a}^{j} \bigcup_{S \in \mathcal{N}_i} S\big) && \textit{(}\mathsf{OPT} \textit{ rebuilds by time } j\textit{)} \\
&\leq \textstyle\sum_{i=a}^{j} \sum_{S \in \mathcal{N}_i} w_j(S) && \textit{(by sub-additivity } \textbf{\textit{(R1)}}\textit{)} \\
&\leq \textstyle\sum_{i=a}^{j} \sum_{S \in \mathcal{N}_i} w_i(S) && \textit{(by temporal monotonicity } \textbf{\textit{(R3)}}\textit{)} \\
&= \Delta_a^j \mathsf{OPT}_k\,. && \textit{(by definition)}
\end{aligned}$$

□

Lemma 6.3. *Suppose* OPT *doesn't rebuild by time* $j \in [a, b]$*. Then* $\Delta_a^j \mathsf{OPT}_k \geq \mathsf{cost}(\mathsf{OPT}_{k-1}(\mathcal{I}_a^j))$*.*

Proof. Because OPT doesn't rebuild by time j, some element x in U_{a-1} is not in any new component during $[a, j]$. Let component $S^* \in C_j^*$ contain x. Since S^* is not new during $[a, j]$, it must be that S^* is in C_i^* for every $i \in [a-1, j]$, and $S^* \subseteq U_{a-1}$.

For $i \in [a, j]$, define $\mathcal{C}_i' = \{S \setminus U_{a-1} : S \in \mathcal{C}_i^*\} \setminus \{\emptyset\}$. Then C' is a solution for $\mathcal{I}_a^j$. Because each $\mathcal{C}_i^*$ has at most k components, one of which is S^*, and $S^* \subseteq U_{a-1}$, it follows that each $\mathcal{C}_i$ has at most $k-1$ components. So $\mathsf{cost}(\mathsf{OPT}_{k-1}(\mathcal{I}_a^j)) \leq \mathsf{cost}(C')$.

If a given component $S \setminus U_{a-1}$ is new in C' at time $i \in [a, j]$, then the corresponding component S is new in $\mathcal{C}^*$ at time i. Further, by suffix monotonicity **(R2)**, the cost $w_i\big(S \setminus U_{a-1}\big)$ paid by C' for $S \setminus U_{a-1}$ is at most the cost $w_i(S)$ paid by $\mathcal{C}^*$ for S. (Inspecting the definition of **(R2)**, we require that $S \neq U_i$, which holds because OPT hasn't rebuilt by time j.) So $\mathsf{cost}(C') \leq \Delta_a^j \mathsf{OPT}_k$. □

Lemma 6.4. $\mathsf{cost}(\mathsf{B}_{k-1}(\mathcal{I}_a^{b-1})) \leq (k-1)\Delta_a^{b-1} \mathsf{OPT}_k$

Proof. If OPT rebuilds by time $b-1$, then

$$\begin{aligned}
\mathsf{cost}(\mathsf{B}_{k-1}(\mathcal{I}_a^{b-1})) &\leq (k-1) w_{b-1}(U_{b-1}) && \textit{(} B_k \textit{ doesn't end the phase at time } b-1\textit{)} \\
&\leq (k-1)\Delta_a^{b-1} \mathsf{OPT}_k && \textit{(Lemma 6.2 with } j = b-1\textit{).}
\end{aligned}$$

Otherwise OPT doesn't rebuild by time $b-1$, so

$$\begin{aligned}
\mathsf{cost}(\mathsf{B}_{k-1}(\mathcal{I}_a^{b-1})) &\leq (k-1)\,\mathsf{cost}(\mathsf{OPT}_{k-1}(\mathcal{I}_a^{b-1})) && \textit{(Lemma 6.1 with } j = b-1\textit{)} \\
&\leq (k-1)\Delta_a^{b-1} \mathsf{OPT}_k && \textit{(Lemma 6.3 with } j = b-1\textit{).}
\end{aligned}$$

□

Lemma 6.5. $w_b(U_b) \leq \Delta_a^b \mathsf{OPT}_k$

Proof. If OPT rebuilds by time b, then

$$w_b(U_b) \leq \Delta_a^b \mathsf{OPT}_k \qquad \textit{(Lemma 6.2 with } j = b\textit{).}$$

Otherwise OPT doesn't rebuild by time b, so

$$\begin{aligned}
w_b(U_b) &< \mathsf{cost}(\mathsf{B}_{k-1}(\mathcal{I}_a^b))/(k-1) && \textit{(because } B_k \textit{ ends the phase at time } b\textit{)} \\
&\leq \mathsf{cost}(\mathsf{OPT}_{k-1}(\mathcal{I}_a^b)) && \textit{(Lemma 6.1 with } j = b\textit{)} \\
&\leq \Delta_a^b \mathsf{OPT}_k && \textit{(Lemma 6.3 with } j = b\textit{).}
\end{aligned}$$

□

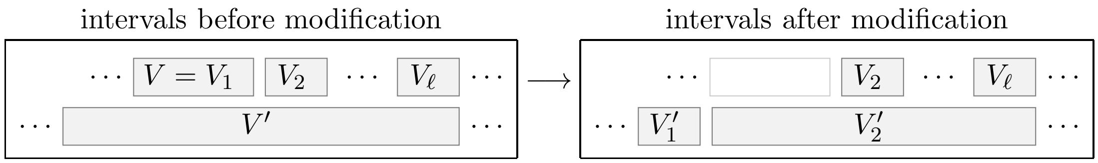

Figure 10: Replacing intervals V and V' by V_1' and V_2' (proof of Theorem 7).

We now finish the proof of Theorem 6.
The definition of B_k gives $\Delta_a^b \mathsf{B}_k = \mathsf{cost}(\mathsf{B}_{k-1}(\mathcal{I}_a^{b-1})) + w_b(U_b)$.
If $a = b$ we have $\mathsf{cost}(\mathsf{B}_{k-1}(\mathcal{I}_a^{b-1})) = 0$.
Otherwise ($a < b$), Lemma 6.4 with $j = b - 1$ gives $\mathsf{cost}(\mathsf{B}_{k-1}(\mathcal{I}_a^{b-1})) \leq (k-1)\Delta_a^{b-1} \mathsf{OPT}_k$.
Lemma 6.5 gives $w_b(U_b) \leq \Delta_a^b \mathsf{OPT}_k$.
The preceding four paragraphs imply $\Delta_a^b \mathsf{B}_k \leq (k-1)\Delta_a^{b-1} \mathsf{OPT}_k + \Delta_a^b \mathsf{OPT}_k \leq k\, \Delta_a^b \mathsf{OPT}_k$. □

4 Properties of optimal offline solutions

Theorem 7. *Every instance of k-Component or Min-Sum Dynamization has an optimal solution that is newest-first and lightest-first.*

Proof. Fix an instance $I = (I_1, \ldots, I_n)$. Abusing notation, let $[i, j]$ denote $\{i, i+1, \ldots, j\}$. For any component S that is new at some time t of a given solution $\mathcal{C}$, we say that S *uses (time) interval* $[t, t']$, where $t' = \max\{j \in [t, n] : (\forall i \in [t, j])\ S \in \mathcal{C}_i\}$ is the time that (this occurence of) S is destroyed. We refer to $[t, t']$ as the *interval* of (this occurence of) S. For the proof we think of any solution $\mathcal{C}$ as being constructed in two steps: (i) choose the set T of time intervals that the components of $\mathcal{C}$ will use, then (ii) given T, for each interval $[t, t'] \in T$, choose a set S of items for $[t, t']$, then form a component S in $\mathcal{C}$ with interval $[t, t']$ (that is, add S to $\mathcal{C}_i$ for $i \in [t, t']$). We shall see that the second step (ii) decomposes by item: an optimal solution can be found by greedily choosing the intervals for each item $x \in U_n$ independently. The resulting solution has the desired properties. Here are the details.

Fix an optimal solution $\mathcal{C}^*$ for the given instance, breaking ties by choosing $\mathcal{C}^*$ to minimize the total query cost $\sum_{[t,t'] \in T^*} t' - t + 1$ where T^* is the set of intervals of components in $\mathcal{C}^*$. Assume without loss of generality that, for each $t \in [1, n]$, if $I_t = \emptyset$, then $\mathcal{C}_t^* = \mathcal{C}_{t-1}^*$ (interpreting $\mathcal{C}_0^*$ as $\emptyset$). (If not, replace $\mathcal{C}_t^*$ by $\mathcal{C}_{t-1}^*$.) For each item $x \in U_n$, let $\alpha^*(x)$ denote the set of intervals in T^* of components that contain x. The build cost of $\mathcal{C}^*$ equals $\sum_{x \in U_n} \mathsf{wt}(x)\, |\alpha^*(x)|$. For each time t and item $x \in I_t$, the intervals $\alpha^*(x)$ of x *cover* $[t, n]$, meaning that the union of the intervals in $\alpha^*(x)$ is $[t, n]$.

Next construct the desired solution $\mathcal{C}'$ from T^*. For each time t and item $x \in I_t$, let $\alpha(x) = \{V_1, \ldots, V_\ell\}$ be a sequence of intervals chosen greedily from T^* as follows. Interval V_1 is the latest-ending interval starting at time t. For $i \geq 2$, interval V_i is the latest-ending interval starting at time $t'_{i-1} + 1$ or earlier, where t'_{i-1} is the end-time of V_{i-1}. The final interval has end-time $t'_\ell = n$. By a standard argument, this greedy algorithm chooses from T^* a minimum-size interval cover of $[t, n]$, so $|\alpha(x)| \leq |\alpha^*(x)|$.

Obtain $\mathcal{C}'$ as follows. For each interval $[i, j] \in T^*$, add a component in $\mathcal{C}'$ with time interval $[i, j]$ containing the items x such that $[i, j] \in \alpha(x)$. This is a valid solution because, for each time t and $x \in I_t$, $\alpha(x)$ covers $[t, n]$. Its build cost is at most the build cost of $\mathcal{C}^*$, because

$\sum_{x\in U_n} \mathsf{wt}(x)|\alpha(x)| \le \sum_{x\in U_n} \mathsf{wt}(x)|\alpha^*(x)|$. At each time t, its query cost is at most the query cost of $\mathcal{C}^*$, because it uses the same set T^* of intervals. So $\mathcal{C}'$ is an optimal solution.

$\mathcal{C}'$ is newest-first. The following properties hold:

1. *α uses (assigns at least one item to) each interval $V \in T^*$.* Otherwise removing V from T^* (and using the same α) would give a solution with the same build cost but lower query cost, contradicting the definition of $\mathcal{C}^*$.

2. *For all $t \in [1, n]$, the number of intervals in T^* starting at time t is 1 if $I_t \neq \emptyset$ and 0 otherwise.* Among intervals in T^* that start at t, only one — the latest ending — can be used in any $\alpha(x)$. So by Property 1 above, T^* has at most interval starting at t. If $I_t \neq \emptyset$, $\mathcal{C}^*$ must have a new component at time t, so there is such an interval. If $I_t = \emptyset$ there isn't (by the initial choice of $\mathcal{C}^*$ it has no new component at time t).

3. *For every two consecutive intervals V_i, V_{i+1} in any $\alpha(x)$, V_{i+1} is the interval in T^* that starts just after V_i ends.* Fix any such V_i, V_{i+1}. For every other item y with $V_i \in \alpha(y)$, the interval following V_i in $\alpha(y)$ must also (by the greedy choice) be V_{i+1}. That is, *every* item assigned to V_i is also assigned to V_{i+1}. If V_{i+1} were to overlap V_i, replacing V_i by the interval $V_i \setminus V_{i+1}$ (within T^* and every $\alpha(x)$) would give a valid solution with the same build cost but smaller total query cost, contradicting the choice of $\mathcal{C}^*$. So V_{i+1} starts just after V_i ends. By Property 2 above, V_{i+1} is the only interval starting then.

4. *For every pair of intervals V and V' in T^*, either $V \cap V' = \emptyset$, or one contains the other.* Assume otherwise for contradiction, that is, two intervals *cross*: $V \cap V' \neq \emptyset$ and neither contains the other. Let $[a, a']$ and $[b, b']$ be a rightmost crossing pair in T^*, that is, such that $a < b < a' < b'$ and no crossing pair lies in $[a + 1, n]$. By Property 1 above, $[a, a']$ is in some $\alpha(x)$. Also $a' < n$. Let $[a' + 1, c]$ be the interval added greedily to $\alpha(x)$ following $[a, a']$. (It starts at time $a' + 1$ by Property 3 above.) The start-time of $[b, b']$ is in $[a, a' + 1]$ (as $a < b < a'$), so by the greedy choice (for $[a, a']$) $[b, b']$ ends no later than $[a' + 1, c]$. Further, by the tie-breaking in the greedy choice, $c > b'$. So $[a' + 1, c]$ crosses $[b, b']$, contradicting that no crossing pair lies in $[a + 1, n]$.

By inspection of the definition of newest-first, Properties 2 and 4 imply that $\mathcal{C}'$ is newest-first.

$\mathcal{C}'$ is lightest-first. To finish we show that $\mathcal{C}'$ is lightest-first. For any time $t \in [1, n]$, consider any intervals $V, V' \in T^*$ where V ends at time t while V' includes t but doesn't end then. To prove that $\mathcal{C}'$ is lightest-first, we show $\mathsf{wt}(V) < \mathsf{wt}(V')$.

The intervals of $\mathcal{C}'$ are nested (Property 4 above), so $V \subset V'$ and the items assigned to $V = V_1$ are subsequently assigned (by Property 3 above) to intervals $V_2, \ldots, V_\ell$ within V' as shown in Figure 10, with V_ℓ and V' ending at the same time. Since V' doesn't end when V does, $\ell \ge 2$. Consider modifying the solution $\mathcal{C}'$ as follows. Remove intervals V and V' from T^*, and replace them by intervals V_1' and V_2' obtained by splitting V' so that V_2' starts when V started. (See the right side of Figure 10.)

Reassign all of V''s items to V_1' and V_2'. Reassign all of V's items to V_2' and unassign those items from each interval V_i. This gives another valid solution. It has lower query cost (as V is gone), so by the choice of $\mathcal{C}^*$ (including the tie-breaking) the new solution must have strictly larger build cost. That is, the change in the build cost, $\mathsf{wt}(V)(1 - \ell) + \mathsf{wt}(V')$, must be positive, implying that $\mathsf{wt}(V') > \mathsf{wt}(V)(\ell - 1) \ge \mathsf{wt}(V)$ (using $\ell \ge 2$). Hence $\mathsf{wt}(V') > \mathsf{wt}(V)$. $\square$

5 Open problems and miscellaneous practical considerations

5.1 Open problems

For k-Component Dynamization:

- Is there an online algorithm with competive ratio $O(\min(k, \log^* m))$?
- Is there an algorithm with ratio $O(k/(k-h+1))$ versus OPT_h (the optimal solution with maximum query cost $h \leq k$)?
- Is there a randomized algorithm with ratio $o(k)$?
- A *memoryless* randomized algorithm with ratio k, or $O(k)$?

For Min-Sum Dynamization:

- Is there an $O(1)$-competitive algorithm?
- Is there a *newest-first* algorithm with competitive ratio $O(\log^* m)$? (Some LSM architectures only support newest-first algorithms.)
- What are the best ratios for the LSM and general variants?

For both problems:

- If we assume that $\max_{t,t'} \mathsf{wt}(I_t)/\mathsf{wt}(I_{t'})$ (for t' such that $\mathsf{wt}(I_{t'}) > 0$) is bounded, as may occur in practice, can we prove a better ratio?
- For the decreasing-weights and LSM variants, is there always an optimal newest-first solution?

5.2 Variations on the model

Tombstones deleted during major compactions. Times when the cover $\mathcal{C}_t$ has just one component (containing all inserted items) are called *full merges* or *major compactions*. At these times, LSM systems delete all tombstone items (even non-redundant tombstones). Our definition of LSM k-Component Dynamization doesn't capture this, but our definition of General k-Component Dynamization does, so the algorithm B_k in Figure 6 is k-competitive in this case.

Monolithic builds. Our model underestimates query costs because it assumes that new components can be built in response to each query, before responding to the query. In reality, builds take time. Can this be modelled cleanly, perhaps via a problem that constrains the build cost at each time t (and $\mathsf{wt}(I_t)$) to be at most 1, with the objective of minimizing the total query cost?

Splitting the key space. To avoid monolithic builds, when the data size reaches some threshold (e.g., when the available RAM can hold 1% of the stored data) some LSM systems "split": they divide the workload into two parts — the keys above and below some threshold — then restart, handling each part on separate servers. This requires a mechanism for routing insertions and queries by key to the appropriate server. Can this (including a routing layer supporting multiple splits) be cleanly modeled?

Other LSM systems (LevelDB and its derivatives) instead use many small (disk-block size) components, storing in the (cached) indices each component's key interval (its minimum and maximum key). A query for a given key accesses only the components whose intervals contain the key. This suggests a natural modification of our model: redefine the query cost at time t to be the maximum number of such components for any key.

Bloom filters. Most practical LSM systems are configurable to use a Bloom filter for each component, so as to avoid (with some probability) accessing component that don't hold the queried key. However, Bloom filters are only cost-effective when they are small enough to be cached. They require about a byte per key, so are effective only for the smallest components (with a total number of keys no more than the bytes available in RAM). Used effectively, they can save a few disk accesses per query (see [25]). They do not speed up range queries (that is, efficient searches for all keys in a given interval, which LSM systems support but hash-based external-memory dictionaries do not).

External-memory. More generally, to what extent can we apply competitive analysis to the standard I/O (external-memory) model? Given an input sequence (rather than being constrained to maintain a cover) the algorithm would be free to use the cache and disk as it pleases, subjective only to the constraints of the I/O model, with the objective of minimizing the number of disk I/O's, divided by the minimum possible number of disk I/O's for that particular input. This setting may be too general to work with. Is there a clean compromise?

The results below don't address this per se, but they do analyze external-memory algorithms using metrics other than standard worst-case analysis, with a somewhat similar flavor:

[8] Studies competitive algorithms for allocating cache space to competing processes.

[10] Analyzes external-memory algorithms while available RAM varies with time, seeking an algorithm such that, no matter how RAM availability varies, the worst-case performance is as good as that of any other algorithm.

[16] Presents external-memory sorting algorithms that have per-input guarantees — they use fewer I/O's for inputs that are "close" to sorted.

[22, 36] Present external-memory dictionaries with a kind of *static-optimality* property: for any sequence of queries, they incur cost bounded in terms of the minimum achievable by any static tree of a certain kind. (This is analogous to the static optimality of splay trees [49, 39].)

5.3 Practical considerations

Heuristics for newest-first solutions. Some LSM systems *require* newest-first solutions. The Min-Sum Dynamization algorithm Adaptive-Binary (Figure 3) can produce solutions that are not newest-first. Here is one naive heuristic to make it newest-first: at time t, do the minimal newest-first merge that includes all of the components that the algorithm would otherwise have selected to merge. This might result in only a small cost increase on some workloads.

Major compactions. For various reasons, it can be useful to force major compactions at specified times. An easy way to model this is to treat each interval between forced major compactions as a separate problem instance, starting each instance by inserting all items from the major compaction.

Estimating the build cost $\mathsf{wt}_t(S)$. Our algorithms for the decreasing-weights, LSM, and general variants depend on the build costs $\mathsf{wt}_t(S)$ of components S that are not yet built. These can be hard to know exactly in practice. However, the algorithms only depend on the build costs of components S that are unions of the current components. For the LSM variant, it may be possible to construct, along with each component S, a small signature that can be used to estimate the build costs of unions of such components (at later times t), using techniques for estimating intersections of large sets (e.g. [23, 46]). It would be desirable to show that dynamization algorithms are robust in this context — that their competitive ratios are approximately preserved if they use approximate build costs.

Exploiting slack in the Greedy-Dual algorithm. For paging, Least-Recently-Used (LRU) is preferred in practice to Flush-When-Full (FWF), although their competitive ratios are equal. In practice, it can be useful to tune an algorithm while preserving its theoretical performance guarantee. In this spirit, consider the following variant of the Greedy-Dual algorithm in Figure 5. As the algorithm runs, maintain a "spare credit" ϕ. Initially $\phi = 0$. When the algorithm does a merge in Line 2.1.3, increase ϕ by the total credit of the components newer than S_0, which the algorithm destroys. Then, at any time, optionally, reduce ϕ by some amount $\delta \leq \phi$, and increase the credit of any component in the cover by ϕ. The proof of Theorem 3, essentially unchanged, shows that the modified algorithm is still k-competitive. This kind of additional flexibility may be useful in tuning the algorithm. As an example, consider classifying the spare credit by the rank of the component that contributes it, and, when a new component S' of some rank r is created, transferring all spare credit associated with rank r to $\mathsf{credit}[S']$ (after Line 2.1.4 initializes $\mathsf{credit}[S']$ to 0). This natural Balance algorithm balances the work done for each of the k ranks.

6 Acknowledgements

Thanks to Carl Staelin for bringing the problem to our attention and for informative discussions about Bigtable.

References

[1] P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. A framework for index bulk loading and dynamization. In F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors, *Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 115–127. Springer Berlin Heidelberg, 2001.

[2] P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan. Approximating extent measures of points. *Journal of the ACM (JACM)*, 51(4):606–635, 2004.

[3] A. Aggarwal, A. K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *28th Annual Symposium on Foundations of Computer Science (FOCS 1987)*, pages 204–216. IEEE, Oct. 1987.

[4] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, and K. Faraaz. AsterixDB: A scalable, open source BDMS. *Proceedings of the VLDB Endowment*, 7(14):1905–1916, 2014.

[5] L. Arge. External Memory Data Structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, Massive Computing, pages 313–357. Springer US, Boston, MA, 2002.

[6] L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. *Computational Geometry*, 29(2):147–162, Oct. 2004.

[7] A. Bagchi, A. Chaudhary, D. Eppstein, and M. T. Goodrich. Deterministic sampling and range counting in geometric data streams. *ACM Trans. Algorithms*, 3(2), May 2007.

[8] R. D. Barve, E. F. Grove, and J. S. Vitter. Application-controlled paging for a shared cache. *SIAM Journal on Computing*, 29(4):1290–1303, Jan. 2000.

[9] M. A. Bender, R. A. Chowdhury, R. Das, R. Johnson, W. Kuszmaul, A. Lincoln, Q. C. Liu, J. Lynch, and H. Xu. Closing the Gap Between Cache-oblivious and Cache-adaptive Analysis. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 63–73, Virtual Event USA, July 2020. ACM.

[10] M. A. Bender, R. Ebrahimi, J. T. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley. Cache-adaptive Algorithms. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 958–971, Philadelphia, PA, USA, 2014. Society for Industrial and Applied Mathematics.

[11] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming b-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 81–92, New York, NY, USA, 2007. ACM.

[12] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, June 1979.

[13] J. L. Bentley and J. B. Saxe. Decomposable searching problems I. Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, Dec. 1980.

[14] E. Bortnikov, A. Braginsky, E. Hillel, I. Keidar, and G. Sheffi. Accordion: Better memory organization for LSM key-value stores. *Proceedings of the VLDB Endowment*, 11(12):1863–1875, Aug. 2018.

[15] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 546–554, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[16] G. S. l. Brodal, R. Fagerberg, and G. Moruz. Cache-aware and cache-oblivious adaptive sorting. In *Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 576–588. Springer, Berlin, Heidelberg, July 2005.

[17] H. Brönnimann, T. M. Chan, and E. Y. Chen. Towards In-place Geometric Algorithms and Data Structures. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, pages 239–246, New York, NY, USA, 2004. ACM.

[18] N. Buchbinder, S. Chen, and J. Naor. Competitive analysis via regularization. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 436–444. SIAM, 2014.

[19] N. Buchbinder and J. (Seffi) Naor. The design of competitive online algorithms via a primal—dual approach. *Foundations and Trends® in Theoretical Computer Science*, 3(2–3):93–263, 2009.

[20] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. BigTable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[21] Y. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80(9):1412–1434, Sept. 1992.

[22] V. Ciriani, P. Ferragina, F. Luccio, and S. Muthukrishnan. Static optimality theorem for external memory string access. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 219–227, Nov. 2002.

[23] R. Cohen, L. Katzir, and A. Yehezkel. A minimal variance estimator for the cardinality of big data set intersection. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 95–103, Halifax NS Canada, Aug. 2017. ACM.

[24] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, Aug. 2013.

[25] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 79–94, New York, NY, USA, 2017. ACM.

[26] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[27] A. Dent. *Getting Started with LevelDB*. Packt Publishing Ltd, Nov. 2013.

[28] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Stumm. Optimizing space amplification in RocksDB. *CIDR*, pages 3–12, 2017.

[29] D. Feldman, M. Schmidt, and C. Sohler. Turning big data into tiny data: Constant-size coresets for k-means, PCA and projective clustering. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, pages 1434–1453, Philadelphia, PA, USA, 2013. Society for Industrial and Applied Mathematics.

[30] L. George. *HBase: The Definitive Guide: Random Access to Your Planet-Size Data*. "O'Reilly Media, Inc.", Aug. 2011.

[31] G. Graefe. Modern B-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2010.

[32] S. Har-Peled and S. Mazumdar. On coresets for k-means and k-median clustering. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '04, pages 291–300, New York, NY, USA, 2004. ACM.

[33] S. Har-Peled and S. Mazumdar. Coresets for k-means and k-median clustering and their applications. *arXiv:1810.12826 [cs]*, Oct. 2018.

[34] A. R. Karlin, C. Kenyon, and D. Randall. Dynamic TCP acknowledgment and other stories about e/(e - 1). *Algorithmica*, 36(3):209–224, July 2003.

[35] J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Hubbell, P. Michaleas, J. Mullen, A. Prout, A. Reuther, A. Rosa, and C. Yee. Achieving 100,000,000 database inserts per second using Accumulo and D4M. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sept. 2014.

[36] P. Ko and S. Aluru. Optimal self-adjusting trees for dynamic string data in secondary storage. In N. Ziviani and R. Baeza-Yates, editors, *String Processing and Information Retrieval*, volume 4726, pages 184–194. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[37] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[38] Lee and Preparata. Computational Geometry—A Survey. *IEEE Transactions on Computers*, C-33(12):1072–1101, Dec. 1984.

[39] C. Levy and R. Tarjan. A new path from splay to dynamic optimality. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '19, pages 1311–1330, Philadelphia, PA, USA, 2019. Society for Industrial and Applied Mathematics.

[40] H. Lim, D. G. Andersen, and M. Kaminsky. Towards accurate and fast evaluation of multi-stage log-structured designs. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 149–166, Berkeley, CA, USA, 2016. USENIX Association.

[41] C. Luo and M. J. Carey. LSM-based storage techniques: A survey. Technical Report arXiv: 1812.07527, (to appear in VLDB), Dec. 2018.

[42] C. Mathieu, R. Rajaraman, N. E. Young, and A. Yousefi. Competitive data-structure dynamization. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2269–2287. SIAM, 2021.

[43] K. Mehlhorn. Lower bounds on the efficiency of transforming static data structures into dynamic structures. *Mathematical systems theory*, 15(1):1–16, Dec. 1981.

[44] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, June 1996.

[45] M. H. Overmars. *The Design of Dynamic Data Structures*. Number 156 in Lecture Notes in Computer Science. Springer, Berlin, 1. ed., 2. print edition, 1987.

[46] R. Pagh, M. Stöckel, and D. P. Woodruff. Is min-wise hashing optimal for summarizing set intersection? In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems - PODS '14*, pages 109–120, Snowbird, Utah, USA, 2014. ACM Press.

[47] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 1–15, New York, NY, USA, 1991. ACM.

[48] D. G. Severance and G. M. Lohman. Differential files: Their application to the maintenance of large databases. *ACM Trans. Database Syst.*, 1(3):256–267, Sept. 1976.

[49] D. D. Sleator and R. E. Tarjan. Self-adjusting binary trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 235–245, New York, NY, USA, 1983. ACM.

[50] C. Staelin. Personal communication, 201 3.

[51] J. van Leeuwen and M. H. Overmars. The art of dynamizing. In J. Gruska and M. Chytil, editors, *Mathematical Foundations of Computer Science 1981*, Lecture Notes in Computer Science, pages 121–131. Springer Berlin Heidelberg, 1981.

[52] J. S. Vitter. *Algorithms and Data Structures for External Memory.* Number 2,4 in Foundations and Trends in Theoretical Computer Science. Now Publ, Boston, 2008.

[53] K. Yi. Dynamic indexability and the optimality of B-trees. *Journal of the ACM*, 59(4):1–19, Aug. 2012.