Competitive Data-Structure Dynamization*

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

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 Google Bigtable and LevelDB (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 set by adding one or more sets and optionally removing existing sets. For each new set the algorithm incurs \( \text{build cost} \) 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

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. Inserted items are accommodated by destroying and rebuilding components. Dynamization has been applied in computational geometry [38, 21, 1, 2, 18], in geometric streaming algorithms [32, 8, 29, 33], and to design external-memory dictionaries [7, 51, 3, 12].

†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

Bentley’s binary transform [13, 14], later called the logarithmic method [50, 44], 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 [14]. It maintains \( k \) components at all times, of respective sizes \( (\frac{n}{k})^1, (\frac{n}{k})^2, \ldots, (\frac{n}{k})^k \) such that \( 0 < r_1 < r_2 < \cdots < r_k \). (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 \( \frac{k^k}{e^k} n^{1/k} \) 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 [43] (building on [47, 46]). The dynamization scheme it uses can be viewed as a generalization of the binary transform. The tradeoff it achieves is optimal, in some parameter regimes, among all external-memory structures [6, 16, 52]. 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 [5], Facebook’s Cassandra [37], HBase and Accordion (used by Yahoo! and others) [30, 15], LevelDB [27], and RocksDB [28]. In
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 on-disk files, that is, static 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 a 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 [15, §2] (see also [17, 11, 10]). The rate of query operations also varies with time. Non-uniform workloads (whose insert sizes and query rates vary) 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.

--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].

--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.

---

**Figure 1**: Steps 6–11 of the binary transform [13, 14]. Each cell\(\lfloor t \rfloor\) 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 6–11 of the 2-binomial transform [14]. At time \(t\) the top and bottom components hold \((i_1^t)\) and \((i_2^t)\) items where \(0 \leq i_1 < i_2\) and \((i_1^t) + (i_2^t) = t\). 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.

---

In contrast, practical compaction policies do adapt (albeit heuristically) to non-uniformity. 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 \(S\) [49].

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, including batch insertions of non-uniform size, and brings competitive analysis to bear to measure how well algorithms take advantage of this non-uniformity. It introduces new algorithms that have substantially better competitive ratios than existing algorithms.

### 1.2 Min-Sum Dynamization

**Definition 1.1.** 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, \(wt(x)\). A solution is a sequence \(C = (C_1, C_2, \ldots, C_n)\), where each \(C_t\) is a collection of sets (called components) satisfying \(\bigcup_{S \in C_t} S = I_t^*\), where \(I_t^* = \bigcup_{i=1}^t I_i\). That is, \(C_t\) 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: \(\sum_{S \in C_t \setminus C_{t-1}} wt(S)\), where \(wt(S)\) denotes \(\sum_{x \in S} wt(x)\) and \(C_0\) denotes the empty set. The query cost at time \(t\) is \(|C_t| - \text{the number of components in the current cover, } C_t\). The objective is to minimize the cost of the solution, defined as the sum of all build costs and query costs over time.
Remarks. A-priori, the definition of total read cost as \(\sum_{i=1}^{n} |C_i|\) 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 (divided) by the disk-access time.

Results.

Theorem 2.1. (Section 2) For Min-Sum Dynamization, the online algorithm in Figure 3 has competitive ratio \(\Theta(\log^* m)\).

Here \(m\) is the number of non-empty insertions. The iterated logarithm is defined by \(\log^0 m = 0\) for \(m \leq 1\) and \(\log^* m = 1 + \log^* \log_2 m\) for \(m > 1\).

Roughly speaking, every \(2^j\) time steps, 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 \(\text{wt}(I_t) = 1\) for all \(t\) the algorithms produce the same (optimal) solution. But the competitive ratio of the naive adaptation is \(\Omega(\log n)\). (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 \(2n\).)

### 1.3 \(K\)-Component Dynamization

Definition 1.2. 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, \(\max_i |C_i| \leq k\)). The objective is to minimize the total build cost.

Remarks. Previously studied online covering problems (e.g. [4, 19]) do not capture the particular notions of query cost and build cost (e.g. the role of components being destroyed). Perhaps the most closely related well-studied problem is dynamic TCP acknowledgment, a generalization of the classic ski-rental problem [34, 19]. TCP acknowledgment 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. Items in LSM dictionaries are timestamped key/value pairs 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 \(\text{wt}_t(x) \geq \text{wt}_{t+1}(x) \geq \cdots \geq \text{wt}_n(x)\). The cost of
They also hold, for example, if each item has a weight and \( \text{wt}(S) = \max_{x \in S} \text{wt}(x) \).

For LSM, (P1) holds because nonred(\( S \cup S' \)) \( \subseteq \) nonred(\( S \) \( \cup \) nonred(\( S' \)), (P2) because nonred(\( S \setminus I_t^* \)) \( \subseteq \) nonred(\( S \)), and (P3) because the tombstone weight for each item \( x \) is at most \( \text{wt}(x) \).


definition 1.3. (competitive ratio) An algorithm is online if for every input \( I \) it outputs a solution \( C \) such that at each time \( t \) its cover \( C_t \) is independent of \( I_{t+1}, I_{t+2}, \ldots, I_n \). For example, with \( n \) non-empty insertions, 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

Theorem 3.1. (section 3.1) For \( k \)-Component Dynamization (and consequently for its generalizations) no deterministic online algorithm has ratio less than \( k \).

Theorem 3.2. (section 3.2) For \( k \)-Component Dynamization with decreasing weights (and plain \( k \)-Component Dynamization) the deterministic online algorithm in Figure 5 has competitive ratio \( k \).

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 \( \text{wt}(I_t) = 1 \) 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 \( \text{wt}(I_1) = 3 \), \( \text{wt}(I_2) = 1 \), and \( \text{wt}(I_t) = 0 \) for \( t \geq 3 \), it pays \( n + 2 \), while the optimum is 4. (In fact, the algorithm is memoryless — each \( C_t \) is determined by \( C_{t-1} \) and \( I_t \). No deterministic memoryless algorithm has competitive
algorithm greedy-dual\((I_1, I_2, \ldots, I_n)\)

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\([S]\) \(\geq wt_t(S)\)  
      2.1.2. let \(S_0\) be the oldest component such that credit\([S_0]\) \(\geq wt_t(S_0)\)  
      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 \(k\)-competitive algorithm for \(k\)-Component Dynamization with decreasing weights (Theorem 3.2). To obtain a \(k\)-competitive algorithm for the LSM variant (Theorem 3.3, Corollary 3.1), replace \(wt_t(S_0)\) throughout by \(wt'_t(S_0) = wt_t(S'') - wt_t(S'' \setminus S_0)\), for \(S''\) as defined in Line 2.1.3 (\(S'' = \bigcup_{h=i}^j I_h\) for \(i\) s.t. \((\exists j) S_0 = \bigcup_{h=i}^j I_h\)).

ratio independent of \(n\).) Even for uniform instances (\(wt(I_t) = 1\) 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 [14]. Here we call such solutions newest-first:

**Definition 1.4.** A solution \(C\) 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, \(C\) 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 in Figure 3 is lightest-first. The \(k\)-Component Dynamization algorithm in Figure 5 is newest-first. In a newest-first solution, every cover \(C_t\) partitions the set \(I_t^*\) of current items into components of the form \(\bigcup_{i \leq j} I_t\) 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 3.3.** (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 3.2 and 3.3 yield a \(k\)-competitive algorithm for LSM \(k\)-Component Dynamization:

**Corollary 3.1.** (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 3.4.** (Section 3.4) The deterministic online algorithm in Figure 6 is \(k\)-competitive for general \(k\)-Component Dynamization.

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 (which they called arboreal) solutions, their various transforms were near-optimal for uniform inputs [13, 14]. Mehlhorn showed (for uniform inputs) that the best newest-first solution has cost at most a constant times the optimum [42]. We strengthen and generalize this:

**Theorem 4.1.** (Section 4) Every instance \(I\) of \(k\)-Component or Min-Sum Dynamization has an optimal solution that is newest-first and lightest-first.
algorithm $B_1(I_1, I_2, \ldots, I_n)$
1. for $t = 1, 2, \ldots, n$: use cover $C_t = \{I_t^j\}$ where $I_t^j = \bigcup_{i=1}^t I_i$ — the only possible solution: all items in one component

algorithm $B_k(I_1, I_2, \ldots, I_n)$
1. initialize $t' = 1$
2. for $t = 1, 2, \ldots, n$:
   2.1. let $C' = B_{k-1}(I'_v, I'_{v+1}, \ldots, I_t)$ — the solution generated by $B_{k-1}$ for the current phase so far
   2.2. if the total cost of $C'$ exceeds $(k-1)\text{wt}(I_t^j)$: take $C_t = \{I_t^j\}$ and let $t' = t + 1$ — end the current phase
   2.3. else: use cover $C_t = \{I_t^j\} \cup C'_t$, where $C'_t$ is the last cover in $C'$ — $C'_t$ has at most $k - 1$ components

Figure 6: Recursive algorithm for general $k$-Component Dynamization (Theorem 3.4).

One corollary 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.

2 Min-Sum Dynamization (Theorem 2.1)

THEOREM 2.1. For Min-Sum Dynamization, the online algorithm in Figure 3 has competitive ratio $\Theta(\log^* m)$.

Recall that $m$ is the number of non-empty insertions. Section 2.1 proves the upper bound, $O(\log^* m)$. Section 2.2 proves the lower bound, $\Omega(\log^* m)$.

2.1 Algorithm is $O(\log^* m)$-competitive Fix an input $I = (I_1, I_2, \ldots, I_n)$ with $m \leq n$ non-empty sets. Let $C$ be the algorithm’s solution. Let $C^*$ be an optimal solution, of cost $\text{OPT}$. 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 $\text{wt}(S) \leq 2^j$, 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 $\text{wt}(S)$ again at time $t$. This allows a clean statement of the next lemma. In the remainder of the proof, “build cost” of the algorithm refers to this over-counted build cost.

We first bound the total query cost, $\sum_t |C_t|$, of $C$. 

LEMMA 2.1. The total query cost of $C$ is at most twice the (over-counted) build cost of $C$, plus $\text{OPT}$.

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

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

Define $\Delta$ to be the maximum number of components merged by the algorithm in response to any query. Note that $\Delta \leq m$ simply because there are at most $m$ components at any given time in $C$. (Only Line 2.1 increases the number of components, and it does so only if $I_t$ is non-empty.) To finish, we bound the build cost of $C$ by $O(\log^*(\Delta) \cdot \text{OPT})$.

The total weight of all components $I_t$ that the algorithm creates in Line 2.1 is $\sum_t \text{wt}(I_t)$, which is at most $\text{OPT}$ because every $x \in I_t$ is in at least one new component in $C^*$ (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., $\sum_t \text{wt}(S_t)$.

OBSERVATION 2.2. The difference between any two distinct times $t$ and $t'$ is at least $\min\{\mu(t), \mu(t')\}$.

(The observation holds because $t$ and $t'$ are 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 $\text{wt}(x)$ of $x$. Have $x$ in turn charge $\text{wt}(x)$ to each optimal component $S^* \in C^*$ that contains $x$ at time $t$. The entire build cost $\sum_t \text{wt}(S_t)$ is charged to components in $C^*$. To finish, we show that each component $S^*$ in $C^*$ is charged $O(\log^* \Delta)$ times $S^*$’s contribution (via its build and query costs) to $\text{OPT}$.
are the times in \((\tau_1, \tau_2)\) when \(S^*\) is charged, and, at each, the charge is \(wt(S^* \cap S_{\tau_1}) \leq wt(S_{\tau_1})\), so the total charge to \(S^*\) during \((\tau_1, \tau_2)\) is at most \((\ell - 1)wt(S_{\tau_1})\).

At each time \(t_i^j\) with \(i \geq 2\) the previous component \(S_{t_i-1}\), of weight at least \(wt(S_{\tau_1})\), is merged. So each time \(t_i^j\) has capacity \(\mu(t_i^j) \geq wt(S_{\tau_1})\). By Observation 2.2, the difference between each time \(t_i^j\) and the next \(t_i^j+1\) is at least \(wt(S_{\tau_1})\). So \((\ell - 1)wt(S_{\tau_1}) \leq t_i^j - t_i^j \leq \tau_2 - \tau_1 - 1\).

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

Let \(D\) be the set of dominant times. For the rest of the proof the only times we consider are those in \(D\).

**Definition 2.2. (congestion)** For any time \(t \in D\) and component \(S_t\), define the congestion of \(t\) and \(S_t\) to be \(wt(S_t \cap S^*)/\mu(t)\), 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 2.4. (dominant 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 \(C^*\) 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 \(C^*\) during uncongested times is \(O(t_2 - t_1)\). \(\square\)

**Lemma 2.5. (dominant congested times)** The total charge to \(S^*\) at congested times is \(O(wt(S^*) \log^* \Delta)\).

**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 \(\sum_{x \in S^*} |W(x)| wt(x)\).

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 \(wt(x)\) of being chosen: \(Pr[X = x] = wt(x)/wt(S^*)\).

We will show that \(E_X[|W(X)|]\) is \(O(\log^* \Delta)\). Since \(E_X[|W(X)|] = \sum_{x \in S^*} |W(x)| wt(x)/wt(S^*)\), this will imply that the total charge is \(O(\log^* \Delta) wt(S^*)\), 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, \(t = \min\{t \in Z : x \in S_t\}\), 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 $W(X)$ 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 $\text{wt}(S_t \cap S^+) / \mu(t)$, which is at least 64 and at most $\Delta$. For the proof, define the congestion of each leaf $x$ to be $2^\Delta$. 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 $\alpha_t = \text{wt}(S_t \cap S^+) / \mu(t)$ be its congestion. The walk visits $S_t$ with probability $\text{wt}(S^+ \cap S_t) / \text{wt}(S^+)$. Condition on this event. We use two bounds on the conditional probability that a given child $S$ of $S_t$ is the next node in the walk (i.e., $X \in S$).

Here is the first bound:

$$\Pr[X \in S \mid X \in S_t] = \frac{\text{wt}(S \cap S^+)}{\alpha_t \mu(t)} \leq \frac{\text{wt}(S)}{\alpha_t \mu(t)} \leq \frac{1}{\alpha_t},$$

(2.1) using in the last inequality that the algorithm merged a component containing $S$ at time $t$, so $\text{wt}(S) \leq \mu(t)$.

The next bound requires a definition. For each non-leaf child $S_{t'}$ of $S_t$, define $j(t')$ so that its capacity $\mu(t')$ equals $\mu(t) / 2^{j(t')}$. (That is, $j(t') = \log_2(\mu(t) / \mu(t'))$.)

By definition of dominant, each non-leaf child $S_{t'}$ has a distinct, integer $j(t') \geq 1$ (as the capacities of dominant times are powers of two, increasing with $t$).

Now let $S_{t'}$ be any non-leaf child of $S_t$ with congestion $\alpha_{t'} \leq \beta \alpha_t$ for some $\beta \geq 1$. By the definitions

$$\Pr[X \in S \mid X \in S_t] = \frac{\text{wt}(S_{t'} \cap S^+)}{\alpha_{t'} \mu(t')} \leq \frac{\beta \alpha_t \mu(t) / 2^{j(t')}}{\alpha_t \mu(t)} = \frac{\beta}{2^{j(t')}}.$$

(2.2)

Define random variable $\alpha'$ to be the congestion of the next node in the walk, that is, the congestion of the child of $S_t$ that contains $X$.

We first show that the event $\alpha' \leq \alpha_t$ is unlikely. That is, the next node is unlikely to be a child with congestion at most $\alpha_t$. Each such child is a non-leaf $S_{t'}$ with $\alpha_{t'} \leq \alpha_t$. Summing Bounds (2.1) and (2.2) (with $\beta = 1$) over these children, and using that each $j(t')$ is a distinct positive integer, the probability of $\alpha' \leq \alpha_t$ is at most $\sum_{j=1}^{\infty} \min(1/\alpha_t, 1/2^j) \leq \int_0^{\infty} \min(1/\alpha_t, 1/2^j) dj$.

By calculation (splitting the integral at $j = \log_2 \alpha_t$) the probability $\Pr[\alpha' \leq \alpha_t]$ is at most $\log_2 \alpha_t / \alpha_t + 2/\alpha_t$.

Following similar reasoning (but with $\beta = 2^{\alpha_t/2} / \alpha_t$ and splitting the integral at $j = \alpha_t/2$, and using that leaves have congestion $2^{\Delta} > 2^{\Delta/2} \geq 2^{\alpha_t/2}$, the probability $\Pr[\alpha' \leq 2^{\alpha_t/2}]$ is at most $1/2 + 2/\alpha_t$.

Finally we show that the expected increase in the log* of the congestion in this step, that is, $E[\log^* \alpha'] - \log^* \alpha_t$, is at least 1/5. We use $\sqrt{2}$ as the base of the iterated log. Then $\log^*(2^{\alpha_t/2}) = 1 + \log^* \alpha_t$ and by the conclusions of the two previous paragraphs

$$E[\log^* \alpha'] \geq \Pr[\alpha' \geq \alpha_t] \log^* \alpha_t + \Pr[\alpha' \geq 2^{\alpha_t/2}]$$

$$\geq [1 - (2 + \log_2 \alpha_t) / \alpha_t] \log^* \alpha_t + 1/2 - 2/\alpha_t$$

$$= \log^*(\alpha_t) + 1/2 - (2 + 2 \log_2 \alpha_t) \log^* \alpha_t / \alpha_t$$

$$\geq \log^*(\alpha_t) + 1/2 - 3/10 = \log^*(\alpha_t) + 1/5,$$

using in the last inequality that $\alpha_t \geq 64$ (it is congested).

The expected number of steps is $O(\log^* \Delta)$. Let r.v. $L$ be the number of nodes on the walk. Let r.v. $\phi_i$ be the iterated logarithm of the congestion of the $i$th node on the walk. By the bound above, for each $i$, given that $i < L$, $E[\phi_{i+1} - \phi_i | \phi_i] \geq 1/5$. By Wald’s equation, $E[\phi_L - \phi_1] \geq E[L] / 5$. Since $\phi_1 \geq 0$ and $\phi_L = \log^* \Delta$, we have $E[\phi_L - \phi_1] \leq \log^* \sqrt{2} \Delta$. So $E[L] \leq 5 \log^* \Delta \leq 10 + 5 \log^* \Delta$. \[\square\]

The upper bound of $O(\log^* m)$ in Theorem 2.1 follows from Lemmas 2.1—2.5 and $\Delta \leq m$.

2.2 Competitive ratio is $\Omega(\log^* m)$

Lemma 2.6. The competitive ratio of the algorithm in Figure 3 is $\Omega(\log^* m)$.

Proof. We will define an input $I$ parameterized by an arbitrary integer $D \geq 0$. For $D = 2$, Figure 4 describes the input $I$ and the merge tree (of depth $D + 1$).

The desired merge tree. Define an infinite rooted tree $T_{\infty}$ with node set $\{1, 2, 3, \ldots\}$ as follows.

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

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

where $c(i-1)$ is the max child of $i - 1$ (exc. $c(0) = 1$)

\[\text{Note that } \log^*_2 \alpha_t = \Theta(\log^*_2 \alpha_t).\]
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 $i \geq 2$ is given exactly one parent. The depth of $i$ is non-decreasing with $i$. Figure 8 shows the top three levels of $T_{\infty}$.

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

Define the desired merge tree, $T_D^n$, to be the subtree of $T_{\infty}$ induced by the nodes of depth at most $D$. Let $m$ be the number of leaves in $T_D^n$. By the previous paragraph (and $m \leq n_{D+1}$), every leaf in $T_D^n$ has depth $\Omega(\log^* m)$.

Assign weights to the nodes in $T_D^n$ as follows. Fix $N = 2n_D$. Give each node $i$ weight $2^{i-p(i)}$, 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 $2^N$ 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 $t$th leaf of $T_D^n$. Then, at each time $t \in \{m+1, m+2, \ldots, 2N-1\}$, insert an empty set $I_t = \emptyset$.

**No merges until last non-empty insertion.** The algorithm does no merges before time $\min_{i=1}^{m} \text{wt}(I_i)$, which is the minimum leaf weight in $T_D^n$. The lightest leaves are the children of 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 $m 2^{N-n_D} \leq 2^N$, that is, $m \leq 2^{2n_D} = 2^{N-n_D}$ (using $N = 2n_D$). 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 $\{j : p(j) \leq n_d < j\}$. 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 $C_{i(i)}$ matches the nodes in $Q_i$, defined as

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

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 $(p(j) < j)$

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

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 $\Omega(\log^* n)$, so every item is merged $\Omega(\log^* m)$ times, and the algorithm’s build cost is $\Omega(\text{wt}(1) \log^* m) = \Omega(n \log^* m)$ (using $\text{wt}(1) = 2n$).

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 $\sum_{i=1}^{m} t + \sum_{i=m}^{n} 1 = O(m^2 + n)$. Its merge cost is $2 \text{wt}(1) = O(n)$. Recalling that $m \leq 2^{nD} = 2^{N/2} = O(\sqrt{n})$, the optimal cost is $O(n)$.)

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

The upper bound in Section 2.1 and the lower bound in Lemma 2.6 prove Theorem 2.1. Note that in Lemma 2.6, $n \approx m^2$, so $\log^* m = \Omega(\log^* n)$.

3  $K$-Component variants (Theorems 3.1–3.4)

3.1 Lower bound on optimal competitive ratio

**Theorem 3.1.** For $k$-Component Dynamization (and consequently for its generalizations) no deterministic online algorithm has 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 $\varepsilon$ 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 $\sim 1$ 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 the just phase.

The ratio of the whole sequence (both phases together) is now $1.75 - O(\sqrt{\varepsilon})$. By doing additional phases (using infinitesimal $\varepsilon^{1/4}$ in the $i$th phase), the adversary can drive the ratio arbitrarily close to 2. The proof generalizes this idea.

**Proof of Theorem 3.1.** 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 $i$th item being $\sigma_1(j) = \varepsilon^{n_k + n_{k-1} + \cdots + n_{j-i}}$. 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 $\ell(t + 1) \geq \ell(t)$.)

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 $t$. 

10
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 \( p(\sigma_i(j)) \geq \sigma_j(j + 1) \) at the previous time step. So, before all \( n_j \) items from \( \sigma(j) \) are inserted, A must pay at least \( n_j \sigma_j(j + 1) \geq k \) (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 \( C(j) \) 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 \( \text{cost}(C(j)) \), i.e., the total cost of new components in \( C(j) \), 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 \( n_j \leq n_j \) to be the number of items inserted from \( \sigma(j) \), the total cost of building all such components is at most \( \sum_{j=1}^{m_j} \text{cost}(C(j))/\varepsilon/(1 - \varepsilon) \). So \( \text{cost}(C(j)) \) is at most \( \varepsilon/(1 - \varepsilon)^2 + \sum_{j=1}^{m_j} \text{cost}(C(j))/\varepsilon/(1 - \varepsilon) \).

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

\[
(1 - \varepsilon)^2 \text{cost}(\text{OPT}) \leq \varepsilon + \frac{m_j}{k} \sum_{j=1}^{m_j} \text{cost}(C(j)) \leq \varepsilon + \frac{1}{k} \sum_{j=1}^{m_j} \text{cost}(C(j)).
\]

**Lower bound on algorithm cost.** The right-hand side of the above inequality is at most \( \varepsilon/k + 1/k \) \( \text{cost}(A) \), because \( \text{cost}(A) \geq k \) (by the stopping condition) and \( \sum_{j=1}^{k} \sum_{i=1}^{m_j} \text{cost}(C(j)) \leq \text{cost}(A) \). (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 \( k \). \( \square \)

### 3.2 Upper bound for decreasing weights

**Theorem 3.2.** For \( k \)-Component Dynamicization with decreasing weights (and plain \( k \)-Component Dynamicization) the deterministic online algorithm in Figure 5 has competitive ratio \( k \).

**Proof.** Consider any execution of the algorithm on any input \( I_1, I_2, \ldots, I_n \). Let \( \delta_i \) be such that each component’s credit increases by \( \delta_i \) at time \( t_i \). (If Block 2.2 is executed, \( \delta_i = 0 \)). To prove the theorem we show the following lemmas.

**Lemma 3.2.1.** The cost incurred by the algorithm is at most \( k \sum_{i=1}^{n} \text{wt}_t(I_i) + \delta_t \).

**Lemma 3.2.2.** The cost incurred by the optimal solution is at least \( \sum_{i=1}^{n} \text{wt}_t(I_i) + \delta_t \).

**Proof of Lemma 3.2.1.** 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 \( k \sum_{i=1}^{n} \text{wt}_t(I_i) \) (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 \( \text{wt}_t(S_0) \) 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 \( k \sum_{i=1}^{n} \delta_i \), because when Line 2.1.1 executes it increases the total component credit by \( k\delta_i \). \( \square \)

**Proof of Lemma 3.2.2.** Let \( C^* \) be an optimal solution. Let \( C \) 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 \( C_{t-1} \) by \( \delta_t \). So the total credit the algorithm gives is \( k \sum_{i=1}^{n} \delta_i \).

For each component \( S \in C_{t-1} \), think of the credit given to \( S \) as being distributed over the component’s items \( x \in S \) in proportion to their weights, \( \text{wt}_t(x) \): at time \( t \), each item \( x \in S \) receives credit \( \delta_t \text{wt}_t(x)/\text{wt}_t(S) \). Have each \( x \), in turn, charge this amount to one component in \( \text{OPT} \)’s current cover \( C_i^* \) that contains \( x \). In this way, the entire credit \( k \sum_{i=1}^{n} \delta_i \) is charged to components in \( C^* \).
Sublemma 3.2.2.1. Let x be any item. Let [t, t'] be any time interval throughout which x remains in the same component in C. The cumulative credit given to x during [t, t'] is at most wt_t(x).

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

\[ \sum_{i=t}^{t'} \frac{wt_t(x)}{wt_t(S)} \delta_i \leq \frac{wt_t(x)}{wt_{t'}(S)} \sum_{i=t}^{t'} \delta_i \leq \frac{wt_t(x)}{wt_{t'}(S)} \text{credit}_{t'}[S]. \]

The right-hand side is at most wt_t(x), because δ_{t'} > 0 so by inspection of Block 2.1 credit_{t'}[S] \leq wt_{t'}(S).

Next we bound how much charge OPT's components (in C*) receive. For any time t, let N'_t = C'_t \ \cap C_{t-1} contain the components that OPT creates at time t, and let N'_t = \bigcup_{S \in N'_t} S contain the items in these components. Call the charges received by components in 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 N'_t. These charges come from components in C_{t-1}, via items x in N'_t \ \cap I_{t-1}, from time t until the algorithm destroys the component in C_{t-1} that contains x. By Sublemma 3.2.2.1, the total charge via a given x from time t until its component is destroyed is at most wt_t(x), so the cumulative charge to components in N'_t from older components is at most wt_t(N'_t \ \cap I_{t-1}) = wt_t(N'_t) - wt_t(I_t) (using that N'_t \ \cap I_{t-1} = I_t). Using that OPT pays at least wt_t(N'_t) at time t, and summing over t, the sum of all backward charges is at most cost(OPT) - \sum_{t} wt_t(I_t).

Next consider the forward charges, from components created at time t or later, to any component S* in 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 C_{t-1}) can contain items in N'_t, namely, the component in C_{t-1} that contains I_t. (Indeed, the algorithm merges components “newest first”, so any other component in 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 δ_{t'}, so the components created by the algorithm at time t' charge a total of at most δ_{t'} to S*. Let m(t, t') = |N'_t \ \cap 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 N'_t, the forward charges to components in N'_t total at most \sum_{t'=t+1}^{n} m(t, t')δ_{t'}. Summing over t, the sum of all forward charges is at most

\[ \sum_{t'=t+1}^{n} \sum_{t=1}^{n} m(t, t')δ_{t'} = \sum_{t'=2}^{n} \sum_{t'=1}^{t'-1} m(t, t') \leq \sum_{t'=1}^{n} δ_{t'}(k-1) \]

(assuming 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} δ_t is charged to components in C*. Summing the bounds from the two previous paragraphs on the (forward and backward) charges, this implies that

k \sum_{t=1}^{n} δ_t \leq \text{cost(OPT)} - \sum_{t=1}^{n} wt_t(I_t) + (k-1) \sum_{t=1}^{n} δ_t.

This proves the lemma, as it is equivalent to the desired bound \text{cost(OPT)} \geq \sum_{t=1}^{n} wt_t(I_t) + δ_t.

This proves Theorem 3.2.

3.3 Bootstrapping newest-first algorithms

Theorem 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.

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

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

Lemma 3.3.1. For any time t and set S \subseteq I_t, we have wt'_t(S) \leq wt_t(S).

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

\[ wt'_t(S) = \sum_{x \in S} wt'_t(x) = \sum_{x \in S} nr(x, I_t)wt_t({x}) \leq \sum_{x \in S} nr(x, S)wt_t({x}) = wt_t(S). \]
LEMMA 3.3.2. Let $C$ be any newest-first solution for $(I, wt')$ and $(I, wt)$. Then $wt'(C) = wt(C)$.

Proof. Consider any time $t$ with $I_t \neq \emptyset$. Let $S$ be $C$’s new component at time $t$ (so $C_t \setminus C_{t-1} = \{S\}$). Consider any item $x \in S$. Because $C$ is newest-first, $S$ includes all items inserted with or after $x$. So $x$ is redundant in $I_t^*$ iff $x$ is redundant in $S$, that is, $nr(x; I_t^*) = nr(x; S)$, so $wt'(S) = wt_t(S)$ (because Bound (3.3) above holds with equality). Summing over all $t$ gives $wt'(C) = wt(C)$.

Given an instance $(I, wt)$ of LSM $k$-Component Dynamization, the algorithm $A'$ simulates $A$ on the instance $(I, wt')$ defined above. Using Lemma 3.3.2, that $A$ is $c$-competitive, and $wt'(OPT(I, wt')) \leq wt(OPT(I, wt))$ (by Lemma 3.3.1), we get

$$wt(A'(I, wt)) = wt(A(I, wt')) \leq cw'(OPT(I, wt')) \leq cw(OPT(I, wt)).$$

So $A$ is $c$-competitive.

When applying Theorem 3.3, we can use that, for any time $t$ and $S \subseteq I_t^*$, $wt'_t(S) = wt_t(S') - wt_t(S' \setminus S)$ for any $S' \subseteq I_t^*$ such that, for all $x \in S$, item $x$ and every newer item in $I_t^*$ are in $S'$.

Combined with the newest-first algorithm in Figure 5, Theorems 3.2 and 3.3 yield a $k$-competitive algorithm for LSM $k$-Component Dynamization:

COROLLARY 3.1. 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 3.4. The deterministic online algorithm in Figure 6 is $k$-competitive for general $k$-Component Dynamization.

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 $k \geq 2$. Let $OPT_k$ denote the optimal (offline) algorithm. Fix any input $(I, w)$ with $I = (I_1, \ldots, I_n)$. Let $C^*$ be $OPT_k(I_1, \ldots, I_n)$ be an optimal solution.

Recall that $B_k$ partitions the input sequence into phases, each of which (except possibly the last) ends with a full merge (i.e., at a time $t$ with $|C_t| = 1$). Consider any phase. Let $t$ and $t'$ be the first and last time steps during the phase. Let $OPT_k$ denote the cost incurred by $OPT_k$ during this phase. That is, $\Delta OPT_k = \sum_{i=t}^{t'} \sum_{S \in C^*_{t-1}} w_i(S)$. Likewise, let $\Delta B_k$ denote the cost incurred by $B_k$ during this phase. To prove the lemma, we show $\Delta B_k \leq k \Delta OPT_k$.

**Proof idea.** Here is the rough idea. The two parts of $B_k$’s solution for the phase (the recursive part and the full merge) are balanced so that the recursive part’s cost is $(k - 1)$ times the full-merge cost. In the case that the optimal schedule does a full merge during the phase, its cost is at least $B_k$’s full merge cost, implying $k$-competitiveness for the phase. Otherwise, in the optimal schedule, some component resides untouched throughout the phase, so the $k - 1$ remaining components must provide a solution for the recursive part. By induction, $B_{k-1}$ is $(k-1)$-competitive, so that part of the optimal schedule must cost at least $1/(k - 1)$ times the algorithm’s cost for the recursive part. This is enough to show $k$-competitiveness for the phase. This is just the rough idea. In the full proof that follows, various technicalities are necessary to work with the somewhat abstract properties of the build-cost function.

**Details.** We will use the following inequality. Let $\pi' = (I_t, I_{t+1}, \ldots, I_{t'})$ denote the subproblem for the phase and let $\pi = (I_t, I_{t+1}, \ldots, I_{t-1})$ be $\pi'$ with the last input set removed. Then

$$cost(B_{k-1}(\pi)) \leq cost(B_{k-1}(\pi')) \leq (k - 1) cost(OPT_{k-1}(\pi')).$$

The second inequality holds because (by induction) $B_{k-1}$ is $(k - 1)$-competitive. (We use here that Properties (P1)-(P3) for $(I, w)$ imply the same properties for the subproblem.)

We also use the following utility lemma.

LEMA 3.4.1. Let $i \in [t, t']$ be any time during the phase such that, for $OPT_k$’s cover $C^*_i$ at time $i$, every component $S \in C^*_i$ was new in $C^*$ sometime during the interval $[t, i]$. Then $w_i(I^*_i) \leq \Delta OPT_k$.

Proof. We have

$$\Delta OPT_k \geq \sum_{S \in C^*_i} w_i(S) \geq w_i(\bigcup_{S \in C^*_i} S) = w_i(I^*_i).$$

The first inequality holds because, by (P3) temporal monotonicity, when a component $S \in C^*_i$ is built during $[t, i]$, $OPT_k$ incurs build cost at least $w_i(S)$. The second inequality holds by (P1) sub-additivity. The final equality holds because $C^*_i$ is a cover at time $i$. This proves Lemma 3.4.1.

We continue with the proof of Theorem 3.4.

**Case 1.** Suppose the phase ends with a full merge. (This holds in all phases except possibly the last). Then

$$\Delta B_k = cost(B_{k-1}(\pi)) + w_{t'}(I^*_t).$$

There are three subcases, depending on the structure of $OPT_k$’s cover $C^*$ during the phase. For Subcases 1.2 and 1.3, note (by inspection of $B_k$ and (P2) suffix monotonicity) that $t' \geq t + 1$.
Subcase 1.1. Suppose that some component \( S^* \in C_i^\tau \) is not new in \( C^* \) any time during the phase. That is, \( S^* \) is in \( C_i^\tau \) for every \( i \in [t-1, t'] \). So \( S^* \in C_t^{i-1} \). For \( i \in [t, t'] \), define \( C_i^\tau = (S \setminus I_{i-1}^\tau : S \in C_i^\tau) \setminus \{\emptyset\} \). Then \( C' \) is a solution for \( \pi' \) that has most \( k-1 \) components in each cover, so \( \text{cost}(\text{OPT}_{k-1}(\pi')) \leq \text{cost}(C') \).

For each \( i \in [t, t'] \), if a given component \( S \setminus I_{i-1}^\tau \) is new in \( C' \) at time \( i \), then the corresponding component \( S \) is new in \( C^* \) at time \( i \). Further, by (P2) suffix monotonicity, the cost \( w_i(S \setminus I_{i-1}^\tau) \) paid by \( C' \) for \( S \setminus I_{i-1}^\tau \) is at most the cost \( w_i(S) \) paid by \( C^* \) for \( S \). Hence, \( \text{cost}(C') \leq \Delta \text{OPT}_k \). By this and the previous paragraph,

\[
\text{cost}(\text{OPT}_{k-1}(\pi')) \leq \Delta \text{OPT}_k.
\]

The condition for \( B_k \) to end the phase is met at time \( t' \). That is,

\[
w_i(I_t^\tau) < \text{cost}(\text{OPT}_{k-1}(\pi'))/(k-1).
\]

By inspection, Bounds (3.4)–(3.7) imply \( \Delta B_k \leq k \Delta \text{OPT}_k \) as desired.

Subcase 1.2. Suppose that each component in \( C_{i-1}^\tau \) is new sometime during \( [t, t'-1] \). Then every component in \( C_i^\tau \) is also new sometime during \( [t, t'] \). Applying Lemma 3.4.1 with \( i = t' - 1 \) and again with \( i = t' \),

\[
\text{max}\{w_{t-1}(I_{t-1}^\tau), w_{t'}(I_t^\tau)\} \leq \Delta \text{OPT}_k.
\]

But the condition for \( B_k \) to end the phase is not met at time \( t'-1 \), that is,

\[
\text{cost}(\text{OPT}_{k-1}(\pi')) \leq (k-1)w_{t-1}(I_{t-1}^\tau).
\]

By inspection, Bounds (3.5), (3.8), and (3.9) imply that \( \Delta B_k \leq k \Delta \text{OPT}_k \).

Subcase 1.3. In the remaining subcase, every component in \( C_{i-1}^\tau \) is new in \( C^* \) during the phase, but some component \( S^* \in C_{i-1}^\tau \) is not. Applying the reasoning in Subcase 1.1. Bounds (3.6) and (3.12) imply \( \Delta B_k \leq (k-1) \Delta \text{OPT}_k \).

Suppose, as in Subcase 1.1, that some component \( S^* \in C_i^\tau \) is never new during \([t, t']\). Bound (3.6) holds by the reasoning in Subcase 1.1. Bounds (3.6) and (3.12) imply \( \Delta B_k \leq (k-1) \Delta \text{OPT}_k \).

Otherwise every component in \( C_i^\tau \) is new in \( C^* \) at some point during \([t, t']\). Lemma 3.4.1 with \( i = t' \) implies \( w_{t'}(I_t^\tau) \leq \Delta \text{OPT}_k \). This and Bound (3.12) imply \( \Delta B_k \leq (k-1) \Delta \text{OPT}_k \).

4 Properties of optimal offline solutions

Theorem 4.1. Every instance \( I \) 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 \( C \), we say that \( S \) uses (time) interval \([t, t']\), where \( t' = \max\{j \in [t, n] : (\forall i \in [t, j]) S \in C_i\} \) is the time that (this occurrence of) \( S \) is destroyed. We refer to \([t, t']\) as the interval of (this occurrence of) \( S \). For the proof we think of any solution \( C \) as being constructed in two steps: (i) choose the set \( T \) of time intervals that the components of \( C \) will use, then (ii) given \( T \), for each interval \([t, t'] \in T \), choose a set \( S \) for \([t, t'] \), then form a component \( S \in C \) with interval \([t, t'] \) (that is, add \( S \) to \( C_i \) 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 \( x \in I_i \) independently. The resulting solution has the desired properties. Here are the details.

Fix an optimal solution \( C^* \) for the given instance, breaking ties by choosing \( C^* \) to minimize the total query cost \( \sum_{(t, t') \in T^*} t' - t + 1 \) where \( T^* \) is the set of intervals of components in \( C^* \). Assume without loss of generality that, for each \( t \in [1, n] \), if \( I_t = \emptyset \), then \( C_t^* = C_{t-1}^* \) (interpreting \( C_0^* \) as \( \emptyset \)). (If not, replace \( C_t^* \) by \( C_{t-1}^* \).) For each item \( x \in I_i^\tau \), let \( \alpha^*(x) \) denote the set of intervals in \( T^* \) of components that contain \( x \). The build cost of \( C^* \) equals \( \sum_{x \in I_i^\tau} \text{wt}(x) |\alpha^*(x)| \). 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 \( C' \) from \( T^* \). For each time \( t \) and item \( x \in I_t \), let \( \alpha(x) = \{V_1, \ldots, V_{l}\} \) 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 \( t' = n \). 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 $C'$ as follows. For each interval $[i, j] \in T^*$, add a component in $C'$ 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 $C^*$, because $\sum_{x \in I_t} wt(x) |\alpha(x)| \leq \sum_{x \in I_t} wt(x) |\alpha^*(x)|$. At each time $t$, its query cost is at most the query cost of $C^*$, because it uses the same set $T^*$ of intervals. So $C'$ is an optimal solution.

$C'$ is newest-first. The following properties hold:

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

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 one interval starting at $t$. If $I_t \neq \emptyset$, $C^*$ 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 $C^*$ 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 $C^*$. 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 above imply that $C'$ is newest-first.

$C'$ is lightest-first. To finish we show that $C'$ 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 $C'$ is lightest-first, we show $\text{wt}(V) < \text{wt}(V')$.

The intervals of $C'$ 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 $V_2, \ldots, V_k$ within $V'$ as shown in Figure 9, with $V_k$ and $V'$ ending at the same time. Since $V'$ doesn’t end when $V$ does, $\ell \geq 2$. Consider modifying the solution $C'$ as follows. Remove intervals $V$ and $V'$ from $T^*$, and replace them by intervals $V_2'$ and $V_3'$ obtained by splitting $V'$ so that $V_2'$ starts when $V$ started. (See the right side of Figure 9.)

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 $C^*$ (including the tie-breaking) the new solution must have strictly larger build cost. That is, the change in the build cost, $\text{wt}(V)/(1 - \ell) + \text{wt}(V')$, must be positive, implying that $\text{wt}(V') > \text{wt}(V)/(\ell - 1) \geq \text{wt}(V)$ (using $\ell \geq 2$). Hence $\text{wt}(V') > \text{wt}(V)$.

5. Conclusion

5.1. Open problems

For $k$-Component Dynamization:
- Is there an online algorithm with competitive ratio $O(\min(k, \log^* m))$?
- Is there an algorithm with ratio $O(k/(k - h + 1))$ versus $\text{OPT}_h$ (the optimal solution with maximum query cost $h \leq k$)?
- Is there a randomized algorithm with ratio $o(k)$?\textsuperscript{6}
- A memoryless randomized algorithm with ratio $k$?

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?\textsuperscript{7}

For both problems:
- If we assume $\max_{t,t'} \wt(I_t)/\wt(I_{t'})$ (for $t'$ such that $\wt(I_{t'}) > 0$) is bounded, as may occur in practice, can we prove a better ratio?\textsuperscript{8}
- 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 $C_t$ 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 problems as defined don’t model this. E.g., for the general model, it violates suffix monotonicity.\textsuperscript{9}

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 $\wt(I_t)$) to be at most 1, with the objective of minimizing the total query cost?

\textsuperscript{9}One can show, using the online randomized primal-dual paradigm of Buchbinder and Naor [19], that there is a randomized algorithm that is $(k - \epsilon)$-competitive for some constant $\epsilon > 0$. The core result is a $(2 - \epsilon)$-competitive algorithm for a continuous-time variant of 2-Component Dynamization. A technical obstacle is that, in contrast to TCP Acknowledgement and Rent-or-Buy [34, 19], the continuous-time variant is strictly harder than the discrete-time problem — one can show that the optimal competitive ratio for deterministic algorithms is strictly above 2.

\textsuperscript{7}With minor modifications to the proof of Theorem 2.1, one may be able to show that the algorithm in Figure 3 is $O(\log^* m)$-competitive for this variant. Similarly one may be able to show that the memoryless randomized variant that, at each time step, uses capacity 2 with probability 1/2, has ratio $O(\log^* m)$.

\textsuperscript{8}Assuming this holds and the input sets $I_t$ are randomly permuted, there is an online algorithm for Min-Sum Dynamization that is asymptotically $(1 + o(1))$-competitive.

\textsuperscript{9}However, with a slight modification to the proof of Theorem 3.4, one may be able to show that the algorithm in Figure 6 remains $k$-competitive in this setting.

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:

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

[11] 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.
2.1.3 initializes the credit of any component in the cover by \( \delta \). The proof of Theorem 3.2, 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 \( \text{credit}(S') \) (after Line 2.1.4 initializes \( \text{credit}(S') \) 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


[33] A. R. Karlin, C. Kenyon, and D. Randall. Dynamic TCP acknowledgment and other stories about e/(-e -


