Dynamic Set Cover: Improved Amortized and Worst-Case Update Time

Sayan Bhattacharya[*]   Monika Henzinger[†]   Danupon Nanongkai[‡]   Xiaowei Wu[§]

Abstract

In the dynamic minimum set cover problem, a challenge is to minimize the update time while guaranteeing close to the optimal $\min(O(\log n), f)$ approximation factor. (Throughout, m, n, f, and C are parameters denoting the maximum number of sets, number of elements, frequency, and the cost range.) In the *high-frequency* range, when $f = \Omega(\log n)$, this was achieved by a deterministic $O(\log n)$-approximation algorithm with $O(f \log n)$ amortized update time [Gupta et al. STOC'17]. In the *low-frequency* range, the line of work by Gupta et al. [STOC'17], Abboud et al. [STOC'19], and Bhattacharya et al. [ICALP'15, IPCO'17, FOCS'19] led to a deterministic $(1 + \epsilon)f$-approximation algorithm with $O(f \log(Cn)/\epsilon^2)$ amortized update time. In this paper we improve the latter update time and provide the first bounds that subsume (and sometimes improve) the state-of-the-art dynamic vertex cover algorithms. We obtain:

1. $(1+\epsilon)f$**-approximation ratio in** $O(f \log^2(Cn)/\epsilon^3)$ **worst-case update time:** No non-trivial worst-case update time was previously known for dynamic set cover. Our bound subsumes and improves by a logarithmic factor the $O(\log^3 n / \operatorname{poly}(\epsilon))$ worst-case update time for unweighted dynamic vertex cover (i.e., when $f = 2$ and $C = 1$) by Bhattacharya et al. [SODA'17].

2. $(1+\epsilon)f$**-approximation ratio in** $O\left((f^2/\epsilon^3) + (f/\epsilon^2)\log C\right)$ **amortized update time:** This result improves the previous $O(f \log(Cn)/\epsilon^2)$ update time bound for most values of f in the low-frequency range, i.e. whenever $f = o(\log n)$. It is the first that is independent of m and n. It subsumes the constant amortized update time of Bhattacharya and Kulkarni [SODA'19] for unweighted dynamic vertex cover (i.e., when $f = 2$ and $C = 1$).

These results are achieved by leveraging the *approximate complementary slackness* and *background schedulers* techniques. These techniques were used in the *local* update scheme for dynamic vertex cover. Our main technical contribution is to adapt these techniques within the *global* update scheme of Bhattacharya et al. [FOCS'19] for the dynamic set cover problem.

---

[*]University of Warwick, UK. Email: `S.Bhattacharya@warwick.ac.uk`

[†]University of Vienna, Austria. Email: `monika.henzinger@univie.ac.at`

[‡]KTH, Stockholm, Sweden. Email: `danupon@kth.se`

[§]IOTSC, University of Macau, China. Email: `xiaoweiwu@um.edu.mo`. This work was done in part while the author was a postdoc at the University of Vienna.

Contents

Part I
EXTENDED ABSTRACT

1 Introduction

In the *minimum set cover* problem, we get a universe of elements $\mathcal{U}$ and a collection of sets $\mathcal{S} \subseteq 2^{\mathcal{U}}$ as input, where $\bigcup_{s \in \mathcal{S}} s = \mathcal{U}$ and each set $s \in \mathcal{S}$ has a *cost* $c_s > 0$ associated with it. A collection of sets $\mathcal{S}' \subseteq \mathcal{S}$ forms a *set-cover* of $\mathcal{U}$ iff $\bigcup_{s \in \mathcal{S}'} s = \mathcal{U}$. The goal is to compute a set cover $\mathcal{S}'$ of $\mathcal{U}$ with minimum total cost $c(\mathcal{S}') = \sum_{s \in \mathcal{S}'} c_s$. This is one of the most fundamental problems in approximation algorithms. In recent years, this problem has also received significant attention in the *dynamic setting*, where the input keeps changing over time. Specifically, here we want to design a *dynamic algorithm* for minimum set cover that can handle the following operations:

Preprocessing: Initially, the algorithm receives as input a universe of elements $\mathcal{U}$, a collection of sets $\mathcal{S} \subseteq 2^{\mathcal{U}}$ with $\bigcup_{s \in \mathcal{S}} s = \mathcal{U}$, and a cost $c_s \geq 0$ for each set $s \in \mathcal{S}$.

Updates: Subsequently, the input keeps changing via a sequence of updates, where each update either (1) deletes an element e from the universe $\mathcal{U}$ and from every set $s \in \mathcal{S}$ that contains e, or (2) inserts an element e into the universe $\mathcal{U}$ and specifies the sets in $\mathcal{S}$ that the element e belongs to.

After each update, we would like to maintain an approximate cost of the optimal set cover of the updated set system. (Some algorithms also allow accessing a solution with such cost. See the remark after Theorem 1.1.) A dynamic algorithm has an *amortized update time* of $O(t)$ iff it takes $O((\alpha + \beta) \cdot t)$ total time (including the time spent on preprocessing) to handle any sequence of $\beta \geq 1$ updates, where α is the number of elements being preprocessed. We want to design a dynamic algorithm with small approximation ratio and update time. We get two main results:

Theorem 1.1 *There are deterministic dynamic algorithms for the minimum set cover problem with $(1 + \epsilon)f$-approximation ratio and*

1. *a worst-case update time of $O(f \log^2(Cn)/\epsilon^3)$, and*

2. *an amortized update time of $O\left(\frac{f^2}{\epsilon^3} + \frac{f}{\epsilon^2} \log C\right)$.*

Here, the symbol f denotes an upper bound on the maximum frequency of any element across all the updates,[1] $C \geq 1$ is a parameter such that $1/C \leq c_s \leq 1$ for all sets $s \in \mathcal{S}$, m is the number of sets in $\mathcal{S}$, and n is the maximum number of elements in the universe $\mathcal{U}$ across all the updates.

Remark: Both our algorithms maintain an $(1+\epsilon)f$-approximation to the *cost* of the minimum set cover after every update and can return this value in constant time. In addition, the algorithm for amortized update time maintains a *solution* of such value throughout the updates (i.e. it outputs the change of the maintained solution after every update), while the algorithm for worst-case update time does not and instead outputs the whole solution in time linear to the solution size whenever the solution is asked for (similar to the dynamic matching algorithm in [BFH19a]).

Perspective: The minimum set cover problem is very well understood in the static setting. There is a simple primal-dual algorithm that gives an f-approximation in $\Theta(fn)$ time, whereas a simple greedy algorithm gives a $\Theta(\log n)$-approximation in $\Theta(fn)$ time. Furthermore, there are strong inapproximability results which imply that the approximation guarantees achieved by these simple primal-dual and greedy algorithms are essentially the best possible [DS14, DGKR05, KR03].

In the dynamic setting, an important challenge is to match the approximation ratio of the (static) greedy and primal-dual algorithms, while minimizing the update time. In recent years, a series of papers on dynamic algorithms have been devoted to this topic. See Table 1 for a concise description of the results obtained in these papers. To summarize, we currently know how to get a

---

[1]Frequency of an element $e \in \mathcal{U}$ is defined as the number of sets in $\mathcal{S}$ that contain e.

Reference	Approximation	Update Time	Deterministic?	Weighted?
[GKKP17]	$O(\log n)$	$O(f \log n)$	yes	yes
[GKKP17, BCH17]	$O(f^3)$	$O(f^2)$	yes	yes
[BHI15a]	$O(f^2)$	$O(f \log(m + n))$	yes	yes
[AAG$^+$19]	$(1 + \epsilon)f$	$O\left(\frac{f^2}{\epsilon} \log n\right)$	no	no
[BHN19]	$(1 + \epsilon)f$	$O\left(\frac{f}{\epsilon^2} \log(Cn)\right)$	yes	yes
Our result (amortized)	$(1 + \epsilon)f$	$O\left(\frac{f^2}{\epsilon^3} + \frac{f}{\epsilon^2} \log C\right)$	yes	yes
Our result (worst case)	$(1 + \epsilon)f$	$O(f \log^2(Cn)/\epsilon^3)$	yes	yes

Table 1: Summary of known results on dynamic set cover. All the previous update times are amortized. The last column indicates if the result holds when different sets have different costs.

$\Theta(\log n)$-approximation in $O(f \log n)$ update time, and a $(1+\epsilon)f$-approximation in $O(f \log(Cn)/\epsilon^2)$ update time. In addition, there is a strong conditional lower bound [AAG$^+$19] which states that any dynamic set cover algorithm with nontrivial approximation ratio must have an update time of $\Omega(f^{1-\delta})$, for any constant $\delta > 0$. This explains the $O(\text{poly}(f))$ factor inherent in all the update time bounds of Table 1, but leaves open the following question.

(Question 1) Must we necessarily incur a polylog(m, n) factor in the update time if we want to aim for near-optimal approximation ratio?

The above question falls within the study of *constant update time* (see below). Besides helping us understand the best possible update time for a dynamic problem to its limit, this study is useful in ruling out non-trivial cell-probe lower bounds [PD06, Lar12, LWY18]. Another important line of work in dynamic graph algorithms is achieving *worst-case* update time. All previous dynamic set cover algorithms can guarantee only *amortized* update time, leaving it widely open the following.

(Question 2): Is there a dynamic algorithm with non-trivial worst-case update time?

When $f = 2$ and $C = 1$, the above questions are equivalent to asking whether there are 2-approximation algorithms for *dynamic (unweighted) vertex cover* with (i) constant update time and (ii) non-trivial worst-case update time. There exists a long line of work on this dynamic (unweighted) vertex cover problem [OR10, BGS11, GP13, NS13, BHI15b, PS16, Sol16, BHN16, BK19]. Currently, the state of the art results on this problem are as follows.

- The deterministic algorithm of [BK19] achieves $(2 + \epsilon)$-approximation in $O(1/\epsilon^2)$ amortized update time for unweighted vertex cover, and the randomized algorithm of [Sol16] achieves 2-approximation in $O(1)$ amortized update time for unweighted vertex cover.
- The deterministic algorithm of [BHN17] achieves $(2 + \epsilon)$-approximation in $O(\log^3 n / \text{poly}(\epsilon))$ worst-case update time for unweighted vertex cover (also see [CS18a, ACC$^+$18, BFH19b]).

Our $O(f \log^2(Cn))$ worst-case bound in Theorem 1.1, when restricted to unweighted vertex cover, improves the $O(\log^3 n)$ bound of [BHN17] by a logarithmic factor. Moreover, ours is the first non-trivial worst-case update time that holds for $f > 2$. On the other hand, our amortized bound in Theorem 1.1 is the first generalization of the vertex cover results from [Sol16, BK19]: When $f > 2$ and $C > 1$, a possible generalization of the constant amortized update time obtained in [BK19, Sol16] is the one guaranteeing $(1 + \epsilon)f$-approximation ratio and $O(\text{poly}(f, C))$ update time. The only previous result of this kind is the $O(f^2)$ update time achieved by [GKKP17, BCH17]; however this comes with a higher approximation ratio of $O(f^3)$. Our amortized update time is the first to achieve the target $O(\text{poly}(f, C))$ bound *simultaneously* with a $(1 + \epsilon)f$-approximation ratio.

2

Finally, note that our amortized update time improves the previous one in [BHN19] in almost the whole range of parameters that we should be interested in: For a fixed $\epsilon > 0$, we get an update time of $O(f^2 + f \log C)$, whereas [BHN19] obtained an update time of $O(f \log(Cn))$. Note that in the *high-frequency range*, when $f = \omega(\log n)$, the $\Theta(\log n)$-approximation ratio obtained by [GKKP17] is already better than a $(1 + \epsilon)f$-approximation. In other words, we are typically interested in getting an $(1 + \epsilon)f$-approximation only in the *low-frequency range*, when $f = O(\log n)$. In this regime, our $O(f^2 + f \log C)$ update time strictly improves upon the update time of [BHN19] for most values of f, i.e. whenever $f = o(\log n)$.

1.1 Techniques

Both our results build on the recent algorithm of Bhattacharya, Henzinger and Nanongkai [BHN19]. This algorithm and most previous deterministic algorithms for dynamic set cover and vertex cover (e.g. [BK19, BHI15b, BHN16, BHN17]) are based on the following static primal-dual algorithm. (For ease of exposition, in this section we assume that $C = 1$; i.e., every set has the same cost.)

The static primal-dual algorithm assigns a fractional *weight* $w_e \geq 0$ to every element $e \in \mathcal{U}$, as follows. Initially, we set $w_e \leftarrow 0$ for all elements $e \in \mathcal{U}$ and $F \leftarrow \mathcal{U}$. Subsequently, the algorithm proceeds in rounds. In each round, we continuously raise the weights of all the elements in F until some set $s \in \mathcal{S}$ becomes *tight* (a set s becomes tight when its total weight $w_s = \sum_{e \in s} w_e$ becomes equal to 1). At this point, we delete the elements contained in the newly tight sets from F, and after that we proceed to the next round. The process stops when F becomes empty. At that point, we return the collection of tight sets as a valid set cover and the weights $\{w_e\}$ as the dual certificate. Specifically, it turns out that the weights $\{w_e\}$ returned at the end of the algorithm form a valid solution to the dual *fractional packing problem*, which asks us to assign a weight $w_e \geq 0$ to each element in $\mathcal{U}$ so as to maximize the objective $\sum_{e \in \mathcal{U}} w_e$, subject to the constraint that $\sum_{e \in s} w_e \leq 1$ for all sets $s \in \mathcal{S}$. We can also show that the collection of tight sets returned at the end of the static algorithm forms a valid set cover, whose cost is at most f times the cost of the dual objective $\sum_{e \in \mathcal{U}} w_e$. This leads to an approximation guarantee of f. In the dynamic setting, the main challenge now is to maintain the (approximate) output of the static algorithm described above in small update time. This is where [BHN19] and previous deterministic algorithms use radically different approaches.

More specifically, previous deterministic algorithms (e.g. [BK19, BHI15b, BHN16, BHN17]) follow some *local update rules* and maintain the approximate complementary slackness conditions. Thus, whenever the weight w_s of a tight set $s \in \mathcal{S}$ becomes too large (resp. too small) compared to 1, these algorithm decrease (resp. increase) the weights of some of the elements contained in s. This step affects the weights of some other sets that share these elements with s, and hence it might lead to a chain of cascading effects. Using very carefully chosen potential functions, these algorithms are able to bound these cascading effects over any sufficiently long sequence of updates. For technical reasons, however, this approach seems to work only when $f = 2$. Thus, although previous works could get constant and worst-case update time for maintaining a $(2 + \epsilon)$-approximate vertex cover, it seems very difficult to extend their potential function analysis to the more general minimum set cover problem (or, equivalently, to minimum vertex cover on hypergraphs).

In contrast, [BHN19] makes no attempt at maintaining the approximate complementary slackness conditions. It simply waits until the overall cost of the dual solution changes by a significant amount (compared to the cost of the set cover maintained by the algorithm). (This approach shares some similarities with the earlier randomized algorithm by [AAG+19], although [AAG+19] is not based on the static algorithm described above.) At that point, the algorithm identifies a critical collection of affected elements and recomputes their weights from scratch using a *global rebuilding*

subroutine. The time taken for this recomputation step is, roughly speaking, proportional to the number of critically affected elements, which leads to a bound on the amortized update time. The strength of this framework is that this global rebuilding strategy extends seamlessly to the general set cover problem (where $f > 2$). Unfortunately this strategy incurs an additional $\Theta(\log n)$ factor in the update time that seems impossible to overcome.

Our algorithm with amortized update time results from carefully combining the these two sharply different approaches, namely the algorithm of [BK19] that uses some local update rules to obtain an $O(1/\epsilon^2)$ amortized update time, and the new approach of [BHN19]. In our *hybrid* approach, whenever the weight w_s of a tight set s becomes too large compared to 1, we decrease the weights of some of the elements contained in s using the same local rule as in [BK19]. In contrast, whenever the weight w_s of a tight set s becomes too small compared to 1, we follow a lazy strategy and try to wait it out. After some period of time, when the total cost of the dual solution becomes significantly small compared to the size of the set cover maintained by the algorithm, we apply a *global rebuilding* subroutine as in [BHN19] to fix the weights of some critical elements. This hybrid approach allows us to combine the best of both worlds, leading to a dynamic algorithm that has $(1 + \epsilon)f$-approximation ratio for any $f \geq 2$ and an amortized update time of $O(f^2/\epsilon^3)$.

Our algorithm with worst-case update time extends the approach of [BHN19] by having many *schedulers* working in parallel. This general idea has been used in many dynamic algorithms with worst-case update time (e.g. [CPR11, GP13, CS18b, NSW17, NS17, Wul17]). The main challenge is typically how to make the schedulers *consistent* in what they maintain, especially if they maintain an overlapping part of the solution. More specifically, we have $k = O(\log n)$ schedulers, where the ith scheduler is associated with an integer r_i such that $r_1 \geq r_2 \geq \ldots \geq r_k$. The ith scheduler is responsible for running the global rebuilding subroutine of [BHN19] on sets that get tight *at and after round* r_i in the static algorithm described above. Thus, the sets that the ith scheduler is responsible for are also under the responsibilities of the jth schedulers for all $j > i$. A complication arises when these schedulers want to rebuild these sets at the same time, since it is not clear which solution of which scheduler we should use as a final solution. Typically, this can be resolved by forcing all schedulers to be *consistent*; i.e. the ith and j^{th} schedulers agree on what happens to each set that they are both responsible for. This seems very hard to achieve in our case. At a high level, we get around this issue by requiring the schedulers to be only *loosely consistent*: Schedulers may maintain drastically different *local views* on the sets they are responsible for, except that there are some mild consistency conditions that tie their behaviors together. This way, each scheduler can work independently while our conditions guarantee that we can still combine results from the schedulers when needed. More specifically, we use the solution that the ith scheduler maintains for level r_i. Due to the consistency conditions this results in an approximately minimum set cover.

2 Preliminaries: A Static Primal-Dual Algorithm

Uniform-cost case: Recall the notations defined in the beginning of Section 1. In order to highlight the main ideas behind our algorithms, in this extended abstract we only consider the special case where every set has the same cost ($c_s = c_{s'}$ for all $s, s' \in \mathcal{S}$) and our goal is to compute a set cover in $(\mathcal{U}, \mathcal{S})$ of minimum size. The full version of the paper is presented in the appendix.

The dual: In the *maximum fractional packing* problem, we get a set system $(\mathcal{U}, \mathcal{S})$ as input. We have to assign a weight $w_e \in [0, 1]$ to every element $e \in \mathcal{U}$, subject to the constraint that $\sum_{e \in s} w_e \leq 1$ for all sets $s \in \mathcal{S}$. The goal is to maximize $\sum_{e \in \mathcal{U}} w_e$. We let $w_s = \sum_{e \in s} w_e$ denote the total weight received by a set $s \in \mathcal{S}$. LP-duality implies the following lemma.

Lemma 2.1 *Consider any instance $(\mathcal{U}, \mathcal{S})$ of the set cover problem. Let OPT denote the size of the minimum set cover on this instance, and let $\{w_e\}_{e \in \mathcal{U}}$ denote any feasible fractional packing solution on the same input instance. Then we have $\sum_{e \in \mathcal{U}} w_e \leq \mathsf{OPT}$.*

We now describe an f-approximation algorithm for minimum set cover in the static setting. The algorithm works as follows. There is a *time-variable t* that is initially set to $-\infty$, and it keeps increasing continuously throughout the duration of the algorithm. At every time t, the algorithm maintains a partition of the universe of elements $\mathcal{U}$ into two subsets: $\mathcal{A}(t) \subseteq \mathcal{U}$ and $\mathcal{F}(t) = \mathcal{U} \setminus \mathcal{A}(t)$. The elements in $\mathcal{A}(t)$ and $\mathcal{F}(t)$ are respectively called *alive* and *frozen* at time t. In the beginning, we have $\mathcal{A}(-\infty) = \mathcal{U}$ and $\mathcal{F}(-\infty) = \emptyset$. As t increases starting from $-\infty$, alive elements become frozen one after the other. Specifically, we have $\mathcal{A}(t) \supseteq \mathcal{A}(t')$ and $\mathcal{F}(t) \subseteq \mathcal{F}(t')$ for all $t \leq t'$. Let t_e be the time at which an element $e \in \mathcal{U}$ becomes frozen (i.e., moves from $\mathcal{A}$ to $\mathcal{F}$). We refer to t_e as the *freezing time* of e. Note that $t_e \leq t$ for all $e \in \mathcal{F}(t)$. At any time t, the weight of an element $e \in \mathcal{U}$ is determined as follows: If $e \in \mathcal{A}(t)$, then $w_e(t) = (1+\epsilon)^t$. Otherwise, $e \in \mathcal{F}(t)$ and $w_e(t) = (1+\epsilon)^{t_e}$.

Let $w_s(t) = \sum_{e \in s} w_e(t)$ denote the weight of a set $s \in \mathcal{S}$ at time t. We say that the set s is *tight* (resp. *slack*) at time t if $w_s(t) = 1$ (resp. $w_s(t) < 1$). Let $\mathcal{T}(t) \subseteq \mathcal{S}$ denote the collection of tight sets at time t. When $t = -\infty$, we have $w_e(t) = (1+\epsilon)^t = 0$ for all elements $e \in \mathcal{A}(t) = \mathcal{U}$, and hence $w_s(t) = 0$ for all sets $s \in \mathcal{S}$. This implies that $\mathcal{T}(-\infty) = \emptyset$. Now, the following invariant completes the description of the algorithm: At any time t, we have $\mathcal{F}(t) = \bigcup_{s \in \mathcal{T}(t)} s$.

To summarize, the algorithm starts at time $t = -\infty$. At that point every element is alive and has weight 0, and all the sets are slack with weight 0. As t starts increasing continuously, the weights of the alive elements keep increasing according to the equation $w_e(t) = (1+\epsilon)^t$. Whenever a set s becomes tight during this process, every alive element $e \in s$ becomes frozen at the same time-instant, which ensures that the weights of all the elements $e \in s$ (and that of the set s) do not increase any further as the value of t keeps increasing. The process stops at time $t = 0$. Note that at time $t = 0$, if an element e is alive, then all sets containing e must be tight. This means that any element e has freezing time $t_e \leq 0$, which leads to the following claim.

Claim 2.1 *At time $t = 0$, we have $\mathcal{A}(0) = \emptyset$ and $\mathcal{F}(0) = \mathcal{U}$.*

Levels of elements and sets: Claim 2.1 implies that the continuous process describing the static algorithm ends at time $t = 0$. At that point, every element $e \in \mathcal{U} = \mathcal{F}(0)$ has a well-defined freezing time $t_e \leq 0$. We define the *level* of an element $e \in \mathcal{U}$ to be $\ell(e) = -t_e$. The *level* of a set $s \in \mathcal{T}(0)$ is defined as $\ell(s) = -t_s$, where t_s is the time at which the set s became tight. The levels of the remaining sets $s \in \mathcal{S} \setminus \mathcal{T}(0)$ are defined to be $\ell(s) = 0$.

Henceforth, we use the symbol w_e to denote the weight of an element e at time $t = 0$ (i.e., $w_e = w_e(0)$). Similarly, we use the symbol w_s to denote $w_s(0)$, and the symbol $\mathcal{T}$ to denote $\mathcal{T}(0)$. Finally, when we say that a set s is tight, we mean that it is tight at time $t = 0$.

Property 2.1 *We have $\ell(e) = \max_{s \in \mathcal{S}: e \in s} \ell(s)$ and $w_e = (1+\epsilon)^{-\ell(e)}$ for all elements $e \in \mathcal{U}$.*

Property 2.2 *We have $w_s \leq 1$ for all $s \in \mathcal{S}$. Further, every set $s \in \mathcal{S}$ at level $\ell(s) > 0$ is tight.*

Lemma 2.2 *The weights $\{w_e\}_{e \in \mathcal{U}}$ form a fractional packing and the collection of sets $\mathcal{T}$ forms an f-approximate minimum set cover in $(\mathcal{U}, \mathcal{S})$.*

Proof: Since $w_s \leq 1$ for all $s \in \mathcal{S}$, the weights $\{w_e\}_{e \in \mathcal{U}}$ form a fractional packing. Consider any element $e \in \mathcal{U}$. If at least one set $s \in \mathcal{S}$ containing e lies at level $\ell(s) > 0$, then $s \in \mathcal{T}$ and hence the element e is covered by $\mathcal{T}$. Otherwise, every set $s \in \mathcal{S}$ containing the element e lies at level $\ell(s) = 0$. So Property 2.1 implies that $\ell(e) = 0$, and hence $w_e = (1+\epsilon)^{-0} = 1$. Thus, every set s

5

containing e has weight $w_s \geq w_e = 1$. In other words, every set s containing e is tight, and so the element e is again covered by $\mathcal{T}$. This implies that $\mathcal{T}$ forms a valid set cover.

Since each element e contributes to the weight w_s of only the (at most f) sets that contain it, we have $\sum_{e \in \mathcal{U}} f \cdot w_e \geq \sum_{s \in \mathcal{S}} w_s \geq \sum_{s \in \mathcal{T}} w_s = |\mathcal{T}|$. The last equality holds since $w_s = 1$ for all sets $s \in \mathcal{T}$. Now, Lemma 2.1 implies that $f \cdot \mathsf{OPT} \geq f \cdot \sum_{e \in \mathcal{U}} w_e \geq |\mathcal{T}|$. ∎

3 Our Algorithm for Amortized Update time: An Overview

Preprocessing: We start by computing the solution returned by the static algorithm from Section 2. Let $\ell(s)$ be the level of a set $s \in \mathcal{S}$ in the output of this static algorithm. The level $\ell(e)$ and the weight w_e of every element $e \in \mathcal{U}$ are determined by the levels of the sets containing it, in accordance with Property 2.1. The weight of a set $s \in \mathcal{S}$ is defined as $w_s = \sum_{e \in s} w_e$. In addition, we associate a variable ϕ_s with every set $s \in \mathcal{S}$. The value of ϕ_s is called the *dead-weight* of s. In contrast, the value of w_s denotes the *real-weight* of s. The *total-weight* of a set $s \in \mathcal{S}$ is given by $w_s^* = w_s + \phi_s$. Just after preprocessing, we have $\phi_s = 0$ for all $s \in \mathcal{S}$, so that the total-weight of every set is equal to its dead-weight. Throughout Section 3, we will say that a set $s \in \mathcal{S}$ is *tight* if $w_s^* = 1$ and *slack* if $w_s^* < 1$. Accordingly, the total-weights $\{w_s^*\}_{s \in \mathcal{S}}$ satisfy Property 2.2 just after preprocessing. The significance of the notion of dead-weights will become clear shortly.

3.1 Handling deletions of elements

When an element e gets deleted, the real-weight w_s of every set s containing e decreases by w_e. To compensate for this loss, we set $\phi_s \leftarrow \phi_s + w_e$ for all sets $s \in \mathcal{S}$ that contained e. Thus, the total-weight of every set remains unchanged due to an element-deletion. It should now be apparent that our algorithm satisfies Property 2.2 if we replace the real-weights w_s by the total-weights w_s^*.

Invariant 3.1 *We have $w_s^* \leq 1$ for all $s \in \mathcal{S}$. Further, every set $s \in \mathcal{S}$ at level $\ell(s) > 0$ is tight.*

As the elements keep getting deleted the size of minimum set cover keeps decreasing. But the set cover maintained by the algorithm we have described so far remains unchanged. Hence, after sufficiently many deletions, the approximation ratio of our algorithm will degrade by a significant amount. To address this concern, our algorithm *rebuilds* part of the solution once the sum of the dead-weights of the sets becomes too large. Specifically, we maintain the following invariant.

Invariant 3.2 *We have $\sum_{s \in \mathcal{S}} \phi_s \leq \epsilon \cdot f \cdot \sum_{e \in \mathcal{U}} w_e$.*

After preprocessing, the above invariant holds since $\phi_s = 0$ for all $s \in \mathcal{S}$. Subsequently, after handling each element-deletion in the manner described above, we perform the following operations.

- WHILE Invariant 3.2 is violated:
    - Identify the smallest level $k \geq 0$ such that $\sum_{s \in \mathcal{S}: \ell(s) \leq k} \phi_s > \epsilon \cdot f \cdot \sum_{e \in \mathcal{U}: \ell(e) \leq k} w_e$.
    - Call the subroutine REBUILD($\leq k$) as described below.

The subroutine REBUILD($\leq k$)**:** Let $\mathcal{S}'_k$ (resp. $\mathcal{U}'_k$) be the collection of sets $s \in \mathcal{S}$ at levels $\ell(s) \leq k$ (resp. the collection of elements $e \in \mathcal{U}$ at levels $\ell(e) \leq k$) just before the call to REBUILD($\leq k$). The subroutine works in two steps: Step I (clean-up) and Step II (rebuild). To simplify the analysis, we make the following crucial assumption in this extended abstract.

Assumption 3.1 *Every set $s \in \mathcal{S}'_k$ contains at least one element from $\mathcal{U}'_k$.*

Step I (clean-up): We set $\phi_s \leftarrow 0$, $\ell(s) \leftarrow k$ for all $s \in \mathcal{S}'_k$. This resets $\ell(e) \leftarrow k$ and $w_e \leftarrow (1+\epsilon)^{-k}$ for all $e \in \mathcal{U}'_k$, as per Property 2.1. The real-weights $\{w_s\}_{s \in \mathcal{S}'_k}$ get updated accordingly.

The clean-up step as described above can only decrease the weight w_e of an element $e \in \mathcal{U}'_k$, since it moves up from its earlier level (which was $\leq k$) to level k. Hence, the real-weights w_s of the sets $s \in \mathcal{S}'_k$ can also only decrease due to this step. Furthermore, since $\ell(e) = \max_{s \in \mathcal{S}:e \in s} \ell(s)$ for every element e (see Property 2.1), all the sets containing an element $e \in \mathcal{U}'_k$ belong to $\mathcal{S}'_k$. Accordingly, we do not change the real-weight w_s of any set $s \in \mathcal{S}$ at level $\ell(s) > k$ during the clean-up step. Neither do we change the dead-weight ϕ_s of any set s or the level/weight of any element e at level $> k$. Since Invariant 3.1 was satisfied just before the clean-up step, we get:

Observation 3.1 *Just after the clean-up step, every set $s \in \mathcal{S}$ at level $\ell(s) > k$ is tight. All the remaining sets $s \in \mathcal{S}'_k$ are at level $\ell(s) = k$ with real-weights $w_s \leq 1$ and dead-weights $\phi_s = 0$.*

Step II (rebuild): Recall that the static algorithm from Section 2 starts at time $t = -\infty$ and stops when t becomes equal to 0. Observation 3.1 implies that after the clean-up step, we are in exactly the same state as the static algorithm from Section 2 at time $t = -k$ (provided we replace the real-weights w_s by the total-weights w_s^* for all sets s at level $\ell(s) > k$). At this point, we perform the remaining steps prescribed by the static algorithm from Section 2 as its time-variable moves from $t = -k$ to $t = 0$. We emphasize that while executing these remaining steps, we do not change the dead-weights of the sets in $\mathcal{S}'_k$ (these dead-weights were reset to zero during the clean-up step, and they continue to remain zero). This leads us to the following observation.

Observation 3.2 *At the end of the call to the subroutine* REBUILD$(\leq k)$, *Invariant 3.1 is satisfied. Furthermore, we also have $\phi_s = 0$ for all sets $s \in \mathcal{S}$ at levels $\ell(s) \leq k$.*

We note that using appropriate data structures this subroutine can be implemented efficiently.

Lemma 3.1 *Under Assumption 3.1, the subroutine* REBUILD$(\leq k)$ *runs in $O(f \cdot |\mathcal{U}'_k|)$ time.*

3.2 Handling insertions of elements

We handle the insertion of an element e' by calling the procedure in Figure 3.1, where $\mathcal{S}_{e'} = \{s \in \mathcal{S} : e' \in s\}$ denotes the collection of sets containing e'. From the outset, we often do not explicitly specify how the real-weight and dead-weight of a set s changes due to the execution of the procedure in Figure 3.1. Instead, they will be implicitly determined as: $w_s = \sum_{e \in s} w_e$ and $w_s^* = w_s + \phi_s$.

Step (01) assigns the element e' a level and a weight in accordance with Property 2.1. This increases the real-weight and the total-weight of every set $s \in \mathcal{S}_{e'}$ by $w_{e'}$. So the sets in $\mathcal{S}_{e'}$ can now potentially violate Invariant 3.1. We say that a set s is *dirty* if it violates Invariant 3.1, and *clean* otherwise. Note that Observation 3.3 is satisfied at this juncture. The FOR loop in Step (02) takes care of these dirty sets. Before proceeding further, we need to define a few important notations.

Notations: For any set $s \in \mathcal{S}$, we let $\mathcal{N}(s) = \{s' \in \mathcal{S} \setminus \{s\} : s \cap s' \neq \emptyset\}$ denote the *neighbors* of s. Next, fix any set $s \in \mathcal{S}$ and consider the following thought experiment. Suppose that we move the set s from its current level to some other level j, while keeping the levels of all the remaining sets $s' \in \mathcal{S} \setminus \{s\}$ unchanged. This potentially changes the levels and weights of some of the elements $e \in s$ in accordance with Property 2.1, and hence the real-weights $w_{s'}$ of some of the sets $s' \in \mathcal{N}(s)$ also get changed. Let $w_{s'}(s \to j)$ denote the resulting real-weight of a set s' after s has moved to level j. It is easy to check that $w_{s'}(s \to j)$ is a continuous non-increasing function of j for all $s', s \in \mathcal{S}$, and that $w_s(s \to \infty) = 0$ for all $s \in \mathcal{S}$. This leads us to the concept of the *target level* $\ell^*(s)$ of a set $s \in \mathcal{S}$ with real-weight $w_s > 1$: If a set $s \in \mathcal{S}$ has real-weight $w_s > 1$, then $\ell^*(s) = \min\{j : w_s(s \to j) = 1\}$. Note that $\ell^*(s) > \ell(s)$.

7

Observation 3.3 *A set s' is dirty only if $s' \in \mathcal{S}_{e'}$ and $w_{s'}^* > 1$.*

---

01. Assign the element e' a level $\ell(e') \leftarrow \max_{s \in \mathcal{S}_{e'}} \ell(s)$ and weight $w_{e'} \leftarrow (1 + \epsilon)^{-\ell(e')}$.
02. FOR every set $s \in \mathcal{S}_{e'}$: Call the subroutine FIX(s).
03. WHILE Invariant 3.2 is violated:
04.   Identify the smallest level $k \geq 0$ such that $\sum_{s \in \mathcal{S}:\ell(s) \leq k} \phi_s > \epsilon \cdot f \cdot \sum_{e \in \mathcal{U}:\ell(e) \leq k} w_e$.
05.   Call the subroutine REBUILD($\leq k$) as described in Section 3.1.

Figure 3.1: Handling the insertion of an element e'.

The subroutine FIX(s): By induction, suppose that Observation 3.3 holds at the start of a given call to FIX(s). The subroutine first checks if $w_s^* > 1$. If not, then Observation 3.3 implies that the set s is already clean and hence the subroutine finishes execution and returns the call. From now onward, we assume that $w_s^* = 1 + \mu_s$ for some $\mu_s > 0$ at the beginning of the call. If $\phi_s \geq \mu_s$, then we set $\phi_s \leftarrow \phi_s - \mu_s$. This makes the set s clean, and again the subroutine finishes execution. Hence, from now onward, we assume that $\mu_s > \phi_s$ at the beginning of the call. We first set $\phi_s \leftarrow 0$, in order to reduce the total-weight of s as much as possible. At this stage, we have $w_s^* = w_s > 1$ and $\phi_s = 0$. The subroutine now moves the set s up to its target-level $\ell^*(s) = j$ (say). This upward movement is achieved via a continuous process. Informally, as the set s keeps moving up, some of its neighbors $s' \in \mathcal{N}(s)$ keep losing their real-weights (because the weights of the some of the elements $e \in s' \cap s$ keep decreasing). In order to compensate for this loss, the affected neighbors $s' \in \mathcal{N}(s)$ keep increasing their dead-weights $\phi_{s'}$ in a continuous manner, whenever possible.

To be more specific, consider an infinitesimal time-interval during this continuous process when the set s moves up from level λ to $\lambda + d\lambda$. As a result, some of the elements $e \in s$ have their weights decreased. This in turn change the real-weights $w_{s'}$ of some of the neighbors $s' \in \mathcal{N}(s)$ by $dw_{s'}(s \rightarrow \lambda)$. Note that $dw_{s'}(s \rightarrow \lambda) \leq 0$. If $w_{s'}(s \rightarrow \lambda) \leq 1$, then we set $\phi_{s'} \leftarrow \phi_{s'} - dw_{s'}(s \rightarrow \lambda)$, in order to compensate for the loss of real-weight of s' during this infinitesimally small time-interval.

Thus, from the perspective of a neighbor $s' \in \mathcal{N}(s)$, the process looks like this: As the set s keeps moving up, the real-weight of s' keep decreasing in a continuous manner, until $w_{s'}$ becomes ≤ 1. From this point onward, the dead-weight $\phi_{s'}$ keeps increasing at the same rate at which the real-weight $w_{s'}$ decreases (thereby keeping the total-weight $w_{s'}^*$ constant).

Observation 3.4 *A call to* FIX(s) *never leads to an already clean set becoming dirty. Furthermore, if Observation 3.3 holds in the beginning of the call, then it continues to hold at the end of the call.*

At the end of the FOR loop in Figure 3.1, every set is clean and hence Invariant 3.1 is satisfied. However, the dead-weights of some of the sets are increased due to the calls to FIX(s). This might lead to a violation of Invariant 3.2. This is addressed by the WHILE loop in steps (03)-(05). Observation 3.2 implies that both the invariants hold at the end of procedure in Figure 3.1.

3.3 Bounding the approximation ratio and amortized update time

The following theorem upper bounds the approximation ratio of our dynamic algorithm.

Theorem 3.1 *The collection of tight sets $\mathcal{T}^* = \{s \in \mathcal{S} : w_s^* = 1\}$ forms a $(1 + \epsilon)f$-approximate minimum set cover in $(\mathcal{U}, \mathcal{S})$.*

Proof: Following the argument in the proof of Lemma 2.2, Invariant 3.1 implies that the collection of tight sets $\mathcal{T}^*$ forms a set cover in $(\mathcal{U}, \mathcal{S})$, and the element-weights $\{w_e\}_{e \in \mathcal{U}}$ form a fractional

8

packing in $(\mathcal{U}, \mathcal{S})$. Next, as in the proof of Lemma 2.2, we first derive that $\sum_{e \in \mathcal{U}} f \cdot w_e \geq \sum_{s \in \mathcal{T}^*} w_s$. Adding the term $\sum_{s \in \mathcal{T}^*} \phi_s$ to both sides of this inequality, we get: $\sum_{e \in \mathcal{U}} f \cdot w_e + \sum_{s \in \mathcal{T}^*} \phi_s \geq \sum_{s \in \mathcal{T}^*}(w_s + \phi_s) = \sum_{s \in \mathcal{T}^*} w_s^* = |\mathcal{T}^*|$. Next, from Invariant 3.2 we derive that: $(1 + \epsilon)f \cdot \sum_{e \in \mathcal{U}} w_e \geq \sum_{e \in \mathcal{U}} f \cdot w_e + \sum_{s \in \mathcal{T}^*} \phi_s \geq |\mathcal{T}^*|$. In other words, there is a fractional packing $\{w_e\}_{e \in \mathcal{U}}$ whose value is within a multiplicative $(1 + \epsilon)f$ factor of the size of a valid set cover $\mathcal{T}^*$. Hence, $\mathcal{T}^*$ is a $(1 + \epsilon)f$-approximate minimum set cover in $(\mathcal{U}, \mathcal{S})$ according to Lemma 2.1. ∎

We spend the rest of this section explaining the main ideas behind the analysis of the amortized update time of our algorithm. We start with an assumption that helps simplify this analysis.

Assumption 3.2 *Suppose that an element e' getting inserted is assigned to a level $\ell(e') = j'$ in step (01) of Figure 3.1. After that, the level of e' does not change during the* FOR *loop in step (02).*

The update time of our algorithm is dominated by two main types of operations: (1) an iteration of the FOR loop in Figure 3.1 where a set s potentially moves up to its target-level, and (2) a call to the subroutine REBUILD($\leq k$). For an operation of type (1), in this section we bound the *fractional work* done by our algorithm instead of the actual time taken to implement it. We give an intuitive justification as to why fractional work is a useful proxy for the actual running time that is analyzed in the full version. In order to bound the time spent on operations of type (2), we introduce the notion of *down-tokens*. We now explain each of these concepts in more details.

Fractional work: Consider an event where a set $s \in \mathcal{S}$ moves up from level j_0 to level j_1, and the level of every other set remains unchanged. This event can change the level of an element $e \in \mathcal{U}$ only if $e \in s$. For all $e \in s$, let $\ell_0(e)$ and $\ell_1(e)$ respectively denote the level of e just before and just after the event. Then the total *fractional work* done during this event $= \sum_{e \in s} f \cdot (\ell_1(e) - \ell_0(e))$.

Justification for fractional work: In the full version our starting point will be a *discretized variant* of the static algorithm from Section 2, where in each round the weights of of the alive elements increase by a multiplicative factor of $(1 + \epsilon)$ and the level of every set and element is an integer in the range $\{0, 1, \ldots, \lceil \log_{(1+\epsilon)} n \rceil\}$. Using appropriate data structures, we can ensure that our algorithm spends $O(f)$ time for each element increasing its level by one unit. This precisely corresponds to the notion of fractional work defined above (when the levels are integers).

Down-tokens: We associate $(1 + \epsilon)^{\ell(s)} \cdot \phi_s$ amount of *down-tokens* with each set $s \in \mathcal{S}$. The *total volume* of down-tokens is given by $\sum_{s \in \mathcal{S}}(1 + \epsilon)^{\ell(s)} \cdot \phi_s$.

The parameter γ_ϵ: In the rest of this section, to ease notations we define $\gamma_\epsilon = (\ln(1+\epsilon))^{-1} = O(\frac{1}{\epsilon})$.

Overview of our analysis: By Lemma 3.3, the total fractional work done per update due to operations of type (1) is at most $O(f^2 \gamma_\epsilon) = O(f^2/\epsilon)$. It now remains to bound the total time spent on operations of type (2). Towards this end, we make the following important observations: (a) Excluding the calls to REBUILD($\leq k$), the procedure for handling the insertion of an element increases the total volume of down-tokens by at most $O(f^2)$ (see Corollary 3.1). (b) Excluding the calls to REBUILD($\leq k$), the procedure for handling the deletion of an element increases the total-volume of down-tokens by at most $O(f)$. This holds because when an element e gets deleted, the dead-weight ϕ_s associated with each set s containing it increases by $w_e = (1 + \epsilon)^{-\ell(e)}$ (see the first paragraph in Section 3.1). Hence, the total volume of down-tokens increases by $\sum_{s \in \mathcal{S}: e \in s}(1 + \epsilon)^{\ell(s)} \cdot (1 + \epsilon)^{-\ell(e)} \leq \sum_{s \in \mathcal{S}: e \in s}(1 + \epsilon)^{\ell(e)} \cdot (1 + \epsilon)^{-\ell(e)} \leq f$. (c) The total time spent on all the calls to REBUILD($\leq k$) is at most $O(1/\epsilon)$ times the decrease in the total volume of down-tokens (see Lemma 3.4). Since the total volume of down-tokens is always nonnegative, all these observations taken together imply an amortized update time of $O(f^2/\epsilon)$. This is slightly better than the bound in Theorem 1.1, because in the full version we do not have the luxury of analyzing *fractional work*.

9

Lemma 3.2 *Consider a call to the subroutine* FIX(s) *in Figure 3.1, which moves the set s up to its target-level. Consider an infinitesimally small time-interval during this iteration when the set s moves up from level λ to level $\lambda + d\lambda$. During this infinitesimally small time-interval, (a) the fractional work done $= -f\gamma_\epsilon(1+\epsilon)^\lambda \cdot dw_s(s \to \lambda)$, and (b) the total volume of down-tokens increases by $\leq -f(1+\epsilon)^\lambda \cdot dw_s(s \to \lambda)$. Here, we have $dw_s(s \to \lambda) = w_s(s \to \lambda + d\lambda) - w_s(s \to \lambda)$.*

Proof: (Sketch) Consider the collection of elements $X_s(\lambda) = \{e \in \mathcal{U} : e \in s$, and $\ell(s') \leq \lambda$ for all $s' \in \mathcal{S} \setminus \{s\}$ with $e \in s'\}$. As the set s moves up from level λ to level $\lambda + d\lambda$, each element $e \in X_s(\lambda)$ changes its level by $d\lambda$ and no other element changes its level.[2] Hence, we get:

$$\text{The fractional work done during this interval } = f|X_s(\lambda)| \cdot d\lambda. \tag{3.1}$$

When the set s is at level λ, each element $e \in X_s(\lambda)$ has weight $(1+\epsilon)^{-\lambda}$. Starting from λ, if we increase the level of s by an infinitesimal amount, then only the elements $e \in X_s(\lambda)$ change their weights, while the weights of every other element in s remains unchanged. Hence, we derive that:

$$\frac{dw_s(s \to \lambda)}{d\lambda} = \frac{d}{d\lambda}\left(|X_s(\lambda)| \cdot (1+\epsilon)^{-\lambda}\right) = -\gamma_\epsilon^{-1} \cdot (1+\epsilon)^{-\lambda} \cdot |X_s(\lambda)|.$$

Thus, we get: $|X_s(\lambda)| \cdot d\lambda = -\gamma_\epsilon \cdot (1+\epsilon)^\lambda \cdot dw_s(s \to \lambda)$. Part (a) of the lemma now follows from (3.1).

Let dw be the change in the weight of an element $e \in X_s(\lambda)$ as the set s moves up from level λ to level $\lambda + d\lambda$. Note that $dw < 0$. From the preceding discussion, it follows that:

$$dw_s(s \to \lambda) = |X_s(\lambda)| \cdot dw. \tag{3.2}$$

Consider the collection of sets $\mathcal{S}^*(\lambda) = \{s' \in \mathcal{S} \setminus \{s\} : s' \cap X_s(\lambda) \neq \emptyset$ and $\ell(s') \leq \lambda\}$. As the set s moves up from level λ to level $\lambda + d\lambda$, each set $s' \in \mathcal{S}^*(\lambda)$ decreases its real-weight $w_{s'}$ by $|s' \cap X_s(\lambda)| \cdot (-dw) \geq 0$, and the real-weight of every other set $s' \notin \mathcal{S}^*(\lambda) \cup \{s\}$ remains unchanged. For each of these sets $s' \in \mathcal{S}^*(\lambda)$, the increase in its dead-weight $\phi_{s'}$ is upper bounded by the decrease in its real-weight (see the description of FIX(s) in Section 3.2). Hence, for each set $s' \in \mathcal{S}^*(\lambda)$, we get $0 \leq d\phi_{s'} \leq |s' \cap X_s(\lambda)| \cdot (-dw)$. None of the other sets change their dead-weights as s moves up from level λ to level $\lambda + d\lambda$. Thus, total volume of the down-tokens increases by:

$$\sum_{s' \in \mathcal{S}^*(\lambda)} (1+\epsilon)^{\ell(s')} \cdot d\phi_{s'} \leq (1+\epsilon)^\lambda \cdot \sum_{s' \in \mathcal{S}^*(\lambda)} d\phi_{s'} \leq (1+\epsilon)^\lambda \cdot (-dw) \cdot \sum_{s' \in \mathcal{S}^*(\lambda)} |s' \cap X_s(\lambda)|$$

$$\leq (1+\epsilon)^\lambda \cdot (-dw) \cdot f \cdot |X_s(\lambda)|.$$

The last inequality holds since each element in $X_s(\lambda)$ is contained in at most f sets from $\mathcal{S}^*(\lambda)$. By (3.2), the increase in the total volume of down-tokens is $\leq -f(1+\epsilon)^\lambda \cdot dw_s(s \to \lambda)$. ∎

Lemma 3.3 *During steps (01)-(02) in Figure 3.1, the total fractional work done is $O(f^2\gamma_\epsilon)$.*

Proof: Suppose that the FOR loop in step (02) runs for r iterations, where in each iteration $i \in \{1, \ldots, r\}$ it deals with a distinct set $s_i \in \mathcal{S}_{e'}$. We will show that the fractional work done during each iteration is $O(f\gamma_\epsilon)$. Since $r = |\mathcal{S}_{e'}| \leq f$, this will imply the lemma.

For the rest of the proof, focus on any iteration $i \in \{1, \ldots, r\}$, and the call to FIX(s_i). At the start of this call, we have $w_s^* = w_s = 1 + \delta_i$, for some $\delta_i > 0$, even after resetting the dead-weight $\phi_{s_i} \leftarrow 0$. (Otherwise, the set s_i does not change its level and the fractional work done $= 0$). At the end of this iteration, the set s_i has moved up to its target-level $\ell^*(s) \leq \ell(e')$ (this inequality

---

[2] Since we consider an infinitesimally small interval and the collection $\{\ell(e) : e \in \mathcal{U}\}$ is finite, $X_s(\lambda) = X_s(\lambda + d\lambda)$.

10

follows from Assumption 3.2), and it is clean with weights $w_s^* = w_s = 1$. Part (a) of Lemma 3.2 now implies that the fractional work done during this iteration is:

$$= \int_{1+\delta_i}^{1} -f\gamma_\epsilon(1+\epsilon)^\lambda dw_{s_i}(s_i \to \lambda) \leq f\gamma_\epsilon(1+\epsilon)^{\ell^*(s)} \int_{1+\delta_i}^{1} -dw_{s_i}(s_i \to \lambda) = \delta_i f\gamma_\epsilon(1+\epsilon)^{\ell^*(s)}$$

$$\leq \delta_i f\gamma_\epsilon(1+\epsilon)^{\ell(e')}. \quad (3.3)$$

Let $w'_{s_i}, w''_{s_i}, w'''_{s_i}, w''''_{s_i}$ respectively denote the real-weight of the set s_i just before the insertion of the element e', just after step (01), just before it starts moving up towards its target-level during the call to FIX(s_i), and just after the call to FIX(s_i). Thus, we have $w'''_{s_i} = 1 + \delta_i$ and $w''''_{s_i} = 1$. Since a call to FIX$(.)$ never increases the real-weight of any set, it follows that $w'''_{s_i} \leq w''_{s_i}$. Since the set s_i was clean just before the insertion of the element e', we get $w'_{s_i} \leq 1$. Finally, step (01) in Figure 3.1 implies that $w''_{s_i} = w'_{s_i} + (1+\epsilon)^{-\ell(e')}$. To summarize, we have:

$$w'_{s_i} \leq 1 = w''''_{s_i} < w'''_{s_i} = 1 + \delta_i \leq w''_{s_i} = w'_{s_i} + (1+\epsilon)^{-\ell(e')}. \quad (3.4)$$

From (3.4) we derive that $1+\delta_i \leq w'_{s_i} + (1+\epsilon)^{-\ell(e')} \leq 1 + (1+\epsilon)^{-\ell(e')}$, which gives us: $\delta_i \leq (1+\epsilon)^{-\ell(e')}$. This observation, along with (3.3), concludes the proof of the lemma. ∎

Corollary 3.1 *Steps (01)-(02) in Figure 3.1 increase the total volume of down-tokens by $O(f^2)$.*

Proof: By Lemma 3.2, the increase in the total volume of down-tokens during steps (01)-(02) is at most γ_ϵ^{-1} times the total fractional work done. The corollary now follows from Lemma 3.3. ∎

Claim 3.1 *Let $\alpha_1 \ldots \alpha_j$ and $\beta_1 \ldots \beta_j$ be nonnegative real numbers satisfying the following property: j is the smallest index $j' \in \{1, \ldots, j\}$ such that $\sum_{i=1}^{j'} \alpha_i > \sum_{i=1}^{j'} \beta_i$. Then for all $0 \leq \lambda_1 \leq \cdots \leq \lambda_j$, we have $\sum_{i=1}^{j}(1+\epsilon)^{\lambda_i} \cdot \alpha_i \geq \sum_{i=1}^{j}(1+\epsilon)^{\lambda_i} \cdot \beta_i$.*

Lemma 3.4 *The time spent to implement a call to the subroutine REBUILD$(\leq k)$ is at most $O(1/\epsilon)$ times the decrease in the total volume of down-tokens during the same call.*

Proof: Unless specified otherwise, throughout this proof we focus on the time-instant just before the call to REBUILD$(\leq k)$. At that time, from Section 3.1 and Figure 3.1 we infer that:

$$k \text{ is the smallest level such that } \sum_{s \in \mathcal{S}: \ell(s) \leq k} \phi_s > \sum_{e \in \mathcal{U}: \ell(e) \leq k} \epsilon f \cdot w_e. \quad (3.5)$$

Define $\mathcal{L}_k = \{\lambda : \lambda \leq k \text{ and } \lambda = \ell(s) \text{ for some set } s \in \mathcal{S}\}$. Since each set gets assigned to exactly one level, we have $|\mathcal{L}_k| \leq |\mathcal{S}|$. In particular, the collection $\mathcal{L}_k$ is finite. Let $\mathcal{L}_k = \{\lambda_1, \ldots, \lambda_j\}$ where $\lambda_1 < \lambda_2 < \cdots < \lambda_j = k$. For any $\lambda_i \in \mathcal{L}_k$, let $\phi_i = \sum_{s \in \mathcal{S}: \ell(s) = \lambda_i} \phi_s$ and $w_i = \sum_{e \in \mathcal{U}: \ell(e) = \lambda_i} w_e$ respectively denote the total dead-weight and element-weight stored at level λ_i. Note that $\sum_{s \in \mathcal{S}: \ell(s) \leq k} \phi_s = \sum_{i=1}^{j} \phi_i$ and $\sum_{e \in \mathcal{U}: \ell(e) \leq k} w_e = \sum_{i=1}^{j} w_i$. From (3.5) we get: j is the smallest index $j' \in \{1, \ldots, j\}$ s.t. $\sum_{i=1}^{j'} \phi_i > \sum_{i=1}^{j'} \epsilon f \cdot w_i$. Now, Claim 3.1 gives us:

$$\sum_{i=1}^{j}(1+\epsilon)^{\lambda_i} \cdot \phi_i \geq \sum_{i=1}^{j'}(1+\epsilon)^{\lambda_i} \cdot \epsilon f \cdot w_i \quad (3.6)$$

Each element $e \in \mathcal{U}$ at level $\ell(e) = \lambda_i$ has weight $w_e = (1+\epsilon)^{-\lambda_i}$. Hence, the quantity $(1+\epsilon)^{\lambda_i} w_i$ equals the number of elements at levels λ_i. Summing over all the levels in $\mathcal{L}_k$, we infer that the

11

right hand side (RHS) of (3.6) equals $\epsilon f \cdot \mathcal{U}'_{\leq k}$, where $\mathcal{U}'_{\leq k}$ is the collection of elements at levels $\leq k$ just before the call to REBUILD($\leq k$). In contrast, the left hand side (LHS) of (3.6) equals the total volume of down-tokens at level $\leq k$ just before the call to REBUILD($\leq k$). The call to REBUILD($\leq k$) does not change the dead-weight of any set $s \in \mathcal{S}$ at level $\ell(s) > k$, and Observation 3.2 states that at the end of the call to this subroutine every set at level $\leq k$ has zero dead-weight. To summarize, the LHS of (3.6) equals the amount by which the total volume of down-tokens decreases during the call to REBUILD($\leq k$), whereas the RHS of (3.6) equals ϵ times the total time spent by our algorithm to implement this call (see Lemma 3.1). This completes the proof of the lemma. ■

4 Our Algorithm for Worst Case Update Time: An Overview

Our complete algorithm needs to deal with a lot of subtle issues, and is presented in the full version. Here, to highlight the main ideas, we only focus on the decremental (deletions only) setting.

Observation 4.1 *The static algorithm in Section 2 never assigns a set $s \in \mathcal{S}$ to a level $\ell(s) \geq L = 1 + \lceil \log_{(1+\epsilon)} n \rceil$, where n is the total number of elements.*

Proof: Recall the continuous process from Section 2 that starts at time $t = -\infty$. When $t = L$, every element $e \in s$ has weight $w_e \leq (1 + \epsilon)^{-L} < 1/n$, and hence $w_s = \sum_{e \in s} w_e < |s| \cdot (1/n) \leq 1$. Since s is not yet tight at time $t = L$, it gets assigned to a level $< L$ at the end of the algorithm. ■

Schedulers: Our dynamic algorithm uses L subroutines called SCHEDULERS – one for each level in $[L] = \{1, \ldots, L\}$. Informally, for each $k \in [L]$ the subroutine SCHEDULER(k) is responsible for all the sets and elements at levels $\leq k$. Each scheduler works on its own local memory that is disjoint from the memory locations used by the other schedulers. Another key feature of our algorithm is that we allow different schedulers to hold mutually inconsistent views regarding the level of an individual set or element. Before proceeding any further, we introduce some important concepts and notations. Most of the concepts defined below – such as the notions of real-weights, dead-weights and total-weights – closely mirror their counterparts from Section 3.

Let $\mathcal{S}^{(k)} \subseteq \mathcal{S}$ and $\mathcal{U}^{(k)} \subseteq \mathcal{U}$ respectively denote the collection of sets and elements SCHEDULER(k) is responsible for. Let $\ell^{(k)}(e)$ and $\ell^{(k)}(s)$ respectively denote the level of an element $e \in \mathcal{U}^{(k)}$ and a set $s \in \mathcal{S}^{(k)}$ from the perspective of SCHEDULER(k). As usual, the level of an element $e \in \mathcal{U}^{(k)}$ according to SCHEDULER(k) is completely determined by the levels of the sets in $\mathcal{S}^{(k)}$ that contain it: We have $\ell^{(k)}(e) = \max_{s \in \mathcal{S}^{(k)}: e \in s} \ell^{(k)}(s) \leq k$ for all $e \in \mathcal{U}^{(k)}$. Let $w_e^{(k)} = (1 + \epsilon)^{-\ell^{(k)}(e)}$ be the weight of an element $e \in \mathcal{U}^{(k)}$ according to SCHEDULER(k). The *real-weight* of a set $s \in \mathcal{S}^{(k)}$ according to SCHEDULER(k) equals $w_s^{(k)} = \sum_{e \in s \cap \mathcal{U}^{(k)}} w_e^{(k)} + \delta_s^{(k)}$, where $\delta_s^{(k)}$ is the *extra-weight* of s. We will shortly see that the concept of extra-weight has a natural explanation. Intuitively, the quantity $\delta_s^{(k)}$ measures the weight received by a set $s \in \mathcal{S}^{(k)}$ from elements that are at level $> k$. Each set $s \in \mathcal{S}^{(k)}$ has a *dead-weight* $\phi_s^{(k)}$, and its *total-weight* is given by $w_s^{*(k)} = w_s^{(k)} + \phi_s^{(k)}$. Finally, SCHEDULER($k$) maintains a collection $D^{(k)}$ of some elements that got deleted from $\mathcal{U}^{(k)}$ in the past due to an external update operation. The elements in $D = \bigcup_{k=1}^{L} D^{(k)}$ are called *dead elements*.

Preprocessing: We first run the static algorithm from Section 2. Let $\ell(s), \ell(e), w_s, w_e$ denote the levels and weights of elements and sets returned by this static algorithm. At this stage, all the different schedulers completely agree with each other regarding the level of each element and set. Specifically, consider any level $k \in [L]$. At this point in time, we have $\mathcal{S}^{(k)} = \{s \in \mathcal{S} : \ell(s) \leq k\}$, $\mathcal{U}^{(k)} = \{e \in \mathcal{U} : \ell(e) \leq k\}$ and $\mathsf{D}^{(k)} = \emptyset$. For all elements $e \in \mathcal{U}^{(k)}$ and sets $s \in \mathcal{S}^{(k)}$, we have

12

$\ell^{(k)}(e) = \ell(e)$, $w_e^{(k)} = w_e$ and $\ell^{(k)}(s) = \ell(s)$. For all $s \in \mathcal{S}^{(k)}$, we set $\delta_s^{(k)} = \sum_{e \in s : \ell(e) > k} w_e$ and $\phi_s^{(k)} = 0$. Hence, we have $w_s^{*(k)} = w_s^{(k)} = w_s$ for all $s \in \mathcal{S}^{(k)}$ just after the end of preprocessing.

Invariants: We now describe three important invariants that are satisfied by our dynamic algorithm. Invariant 4.1 closely mirrors Invariant 3.1 from Section 3.1, and it clearly holds at the end of preprocessing. Invariant 4.2 dictates that the number of dead elements in the control of a SCHEDULER(k) is really small compared to the number of elements in $\mathcal{U}^{(k)}$. At the end of preprocessing, this invariant trivially holds because $D^{(k)} = \emptyset$. Invariant 4.3 says that the sets $\mathcal{S}^{(k)}, \mathcal{U}^{(k)}$ and $D^{(k)}$ form a very nice laminar structure as k ranges from L to 1. Specifically, the sets/elements a SCHEDULER($k-1$) is responsible for are precisely the ones that lie at level $\leq k-1$ according to the next SCHEDULER(k). Furthermore, SCHEDULER(L) is responsible for all the elements and sets. Again, it is easy to check that this invariant holds at the end of preprocessing.

Invariant 4.1 *Consider any $k \in [L]$. Every set $s \in \mathcal{S}^{(k)}$ at level $\ell^{(k)}(s) > 0$ has total-weight $w^{*(k)} = 1$. Furthermore, every set $s \in \mathcal{S}^{(k)}$ at level $\ell^{(k)}(s) = 0$ has total-weight $w^{*(k)} \leq 1$.*

Invariant 4.2 *Consider any $k \in [L]$. We have $\left| D^{(k)} \right| \leq 2\epsilon \cdot \left| \mathcal{U}^{(k)} \right|$.*

Invariant 4.3 *For every $k \in [2, L]$, we have $\mathcal{S}^{(k-1)} = \{s \in \mathcal{S}^{(k)} : \ell^{(k)}(s) \leq k-1\}$, $\mathcal{U}^{(k-1)} = \{e \in \mathcal{U}^{(k)} : \ell^{(k)}(e) \leq k-1\}$ and $D^{(k-1)} = \{e \in D^{(k)} : \ell^{(k)}(e) \leq k-1\}$. Furthermore, we have $\mathcal{S}^{(L)} = \mathcal{S}$ and $\mathcal{U}^{(L)} = \mathcal{U}$.*

Ownerships: We say that an element $e \in \mathcal{U}$ (resp. $e \in D$) is *owned* by a level $k \in [L]$ iff $e \in \mathcal{U}^{(k)}$ (resp. $e \in D^{(k)}$) and $e \notin \mathcal{U}^{(j)}$ (resp. $e \in D^{(j)}$) for all $j \in [k-1]$. A set $s \in \mathcal{S}$ is *owned* by a level $k \in [L]$ iff $s \in \mathcal{S}^{(k)}$ and $s \notin \mathcal{S}^{(j)}$ for all $j \in [k-1]$. Note that each element/set is owned by a unique level. We now describe how to handle a sequence of element deletions after preprocessing.

Handling the deletion of an element: Suppose that an element e, which was owned by level j, gets deleted. Accordingly, we *feed* this deletion to SCHEDULER(j), ..., SCHEDULER(L). Note that these are precisely the schedulers that are responsible for this element and are affected by the concerned deletion. Each of these affected schedulers works within its own local memory, independently of others. We describe the actions taken by SCHEDULER(k), for any $k \in \{j, \ldots, L\}$.

Suppose that the element e was at level $\ell^{(k)}(e) = i$ just before its deletion. SCHEDULER(k) moves the element e from $\mathcal{U}^{(k)}$ to $D^{(k)}$, without changing the its level $\ell^{(k)}(e) = i$ or weight $w_e^{(k)} = (1+\epsilon)^{-i}$. For each set $s \in \mathcal{S}^{(k)}$ containing the element e, this reduces its real-weight $w_s^{(k)}$ by $(1+\epsilon)^{-i}$. To compensate for this loss, we increase its dead-weight $\phi_s^{(k)}$ by the same amount $(1+\epsilon)^{-i}$. We do not change the extra-weights of the sets. Thus, the total-weight of every set also remains unchanged.

Triggering a rebuild: It is easy to check that the above actions do not violate Invariant 4.1 and Invariant 4.3. However, if we keep acting in this lazy manner, then the number of dead elements keep growing with time, and so after some number of updates Invariant 4.2 will get violated. Furthermore, unlike in Section 3 here we cannot even afford to wait until the moment Invariant 4.2 is violated before triggering a rebuild, because we are shooting for worst-case update time and the rebuild needs to occur *in the background* – a few steps at a time. Accordingly, for each $k \in [L]$, whenever SCHEDULER(k) finds that $|D^{(k)}| \geq \epsilon \cdot |\mathcal{U}^{(k)}|$ (note that this is still ϵ-*far* from violating Invariant 4.2), it starts *rebuilding* its part of the input $(\mathcal{U}^{(k)}, \mathcal{S}^{(k)}, D^{(k)})$ *in the background* in a separate memory location that is not affected by the happenings elsewhere. We now explain this rebuilding procedure in a bit more details.

Rebuilding done by SCHEDULER(k)**:** Suppose that SCHEDULER(k) has triggered a rebuild at the present moment because $|D^{(k)}| = \epsilon \cdot |\mathcal{U}^{(k)}|$. In order to see the high level idea, for now assume

13

that SCHEDULER(k) does not need to handle an element deletion due to any external update operation while it is performing the rebuild.[3] Then the rebuild subroutine will *clean-up* all the dead elements in $D^{(k)}$, by resetting $D^{(k)} = \emptyset$ and $\phi_s^{(k)} = 0$ for all $s \in \mathcal{S}^{(k)}$, and then run the static algorithm (see Section 3) on input $(\mathcal{S}^{(k)}, \mathcal{U}^{(k)})$ starting from time $t = -k$ onward. During this run of the static algorithm, the extra-weights $\delta_s^{(k)}$ of the sets $s \in \mathcal{S}^{(k)}$ will not change, because these extra-weights are coming from elements that are at levels $> k$. Note that this is exactly the same principle that underpins the rebuild subroutine described in Section 3.1. At the end of this static algorithm, we will get the following guarantees: Invariant 4.1 is satisfied by SCHEDULER(k), and $D^{(k)} = \emptyset$. At this point, SCHEDULER(k) will *order* all the schedulers corresponding to levels $j \in [k-1] = \{1, \ldots, k-1\}$ to abort whatever they are doing and synchronize their perspectives with the perspective of SCHEDULER(k). We refer to this as the *synchronization* event. At the end of this, each SCHEDULER(j), $j \in [k-1]$, will satisfy: $\mathcal{S}^{(j)} = \{s \in \mathcal{S}^{(k)} : \ell^{(k)}(s) \le j\}$, $\mathcal{U}^{(j)} = \{e \in \mathcal{U}^{(k)} : \ell^{(k)}(s) \le j\}$, $D^{(j)} = \emptyset$, $\ell^{(j)}(e) = \ell^{(k)}(e)$ for all $e \in \mathcal{U}^{(j)}$, and $w_s^{*(j)} = w_s^{*(k)}$ for all $s \in \mathcal{S}^{(j)}$. Our algorithm ensures that this synchronization is achieved in $O(k)$ time *on the fly*, as follows. The rebuild subroutine of SCHEDULER(k) will run k different *threads* – one for each level $j \in \{1, \ldots, k\}$. Each of these threads will run in a designated separate region of memory, independent of others. It will be the responsibility of thread j to prepare everything in its allocated memory region, which can then be handed over to SCHEDULER(j) at the time of synchronization by simply passing a pointer to the beginning of the concerned memory region. Thus, synchronization involves the passing of k pointers, and hence takes $O(k) = O(L) = O(\log n/\epsilon)$ time. It is easy to check that Invariants 4.1 and 4.3 continue to remain satisfied at the end of synchronization. Furthermore, the remaining Invariant 4.2 is trivially satisfied under our working assumption that no element deletion occurs in $\mathcal{U}^{(k)}$ as SCHEDULER(k) is rebuilding in the background. Next, we analyze the worst-case update time of this algorithm, and outline what happens when this working assumption does not hold.

Worst-case update time: We measure the time taken to implement a procedure in terms of *units of work*. The update time of our algorithm is dominated by the time spent on the rebuild subroutines of the individual schedulers. There are L schedulers running in the background, and we ensure that each of these individual schedulers perform $O(fL/\epsilon)$ units of work after every external update (element deletion). This implies a worst-case update time of $O(L \cdot fL/\epsilon) = O(f \log^2 n/\epsilon^3)$.

We now explain why each SCHEDULER(k) needs to perform $O(fL/\epsilon)$ units of work per update. The factor $O(L)$ comes from the fact that the rebuild subroutine of SCHEDULER(k) needs to run $k = O(L)$ threads in the background so as to ensure that it can synchronize on the fly when it finishes execution. It now remains to explain the rationale behind the remaining $O(f/\epsilon)$ factor. Suppose that $d = |D^{(k)}|$ and $u = |\mathcal{U}^{(k)}|$ when the rebuild subroutine of SCHEDULER(k) gets triggered. We have $d = \epsilon u$, and hence the subroutine needs to perform $O(f(u+d)) = O(fu)$ units of work overall (see Lemma 3.1). According to our scheme, the subroutine splits this work across a sequence of external updates (element deletions), performing cf/ϵ units of work per update for some large constant $c > 1$. This ensures that in the general setting (when the working assumption in the previous paragraph does not hold), at most $fu/(cf/\epsilon) = \epsilon u/c$ many elements get deleted from $\mathcal{U}^{(k)}$ when the rebuild subroutine is in progress. In the full version, we show that there is a way to handle these incoming deletions *on the fly* during the rebuild subroutine. However, this comes at a cost: When an element e gets deleted that has already been processed by the rebuild subroutine, it gets classified as a dead element. But since the subroutine is working at a sufficient fast rate, at most $\epsilon u/c$ many new dead elements might get created in this manner at the time of synchronization, whereas the old dead elements that were present at the time the rebuild subroutine was triggered

---

[3]According to this assumption, during the same time-interval when SCHEDULER(k) is performing a rebuild, some other SCHEDULER(j) with $j > k$ might still have to handle element deletions due to external updates.

are removed anyway. Taking everything together, just after synchronization we end up having $|D^{(j)}| \leq (\epsilon/c) \cdot |\mathcal{U}^{(j)}|$ for all $j \in [k]$ and hence Invariant 4.2 continues to remain satisfied.

Approximation ratio: The main challenge here is to show that all the same element/set might get assigned to different levels by different schedulers, there is a way to come up with a *consistent* assignment of levels to all the elements in $\mathcal{U} \cup D$ and all the sets in $\mathcal{S}$. Furthermore, the sets with weight $= 1$ in this consistent assignment form a $(1 + \epsilon)f$-approximate minimum set cover in $(\mathcal{U}, \mathcal{S})$.

The consistent assignment is as follows: Each set $s \in \mathcal{S}$ gets assigned to the level that owns it. Let $\ell(s)$ denote the level of a set $s \in \mathcal{S}$ in the consistent assignment. This automatically defines the level $\ell(e)$ and weight w_e of every element $e \in \mathcal{U} \cup D$, since $\ell(e) = \max_{s \in \mathcal{S}:e \in s} \ell(s)$ and $w_e = (1 + \epsilon)^{-\ell(s)}$. For all $s \in \mathcal{S}$, we define its weight to be $w_s^{(\mathcal{U} \cup D)} = \sum_{e \in \mathcal{U} \cup D:e \in s} w_e$.

For each $k \in [L]$, define $\mathcal{S}(\leq k) = \{s \in \mathcal{S} : \ell(s) \leq k\}$, $D(\leq k) = \{e \in D : \ell(e) \leq k\}$ and $\mathcal{U}(\leq k) = \{e \in \mathcal{U} : \ell(e) \leq k\}$. The lemma below shows that the levels of elements/sets in the consistent assignment are nicely aligned with the levels of the same elements/sets according to the individual schedulers. Lemma 4.1 and Invariant 4.2 together imply Corollary 4.1.

Lemma 4.1 *For each $k \in [L]$, we have $\mathcal{S}(\leq k) = \mathcal{S}^{(k)}$, $\mathcal{U}(\leq k) = \mathcal{U}^{(k)}$, and $D(\leq k) = D^{(k)}$.*

Proof: (Sketch) The lemma follows from induction: It is easy to check that the lemma holds just after preprocessing. When an element e gets deleted, each SCHEDULER(k) responsible for e simply moves it from $\mathcal{U}^{(k)}$ to $D^{(k)}$ without changing its weight or level. Hence, the lemma continues to remain satisfied. Finally, a moment's thought reveals that the lemma continues to hold after a rebuild subroutine of some SCHEDULER(k) executes its synchronization step. ∎

Corollary 4.1 *For each $k \in [L]$, we have $|D(\leq k)| \leq 2\epsilon \cdot |\mathcal{U}(\leq k)|$.*

Corollary 4.2 *We have $\sum_{e \in D} w_e \leq 2\epsilon(1 + \epsilon) \cdot \sum_{e \in \mathcal{U}} w_e$.*

Proof: (Sketch) The proof is almost the same as the proof of Lemma 4.8 in the Arxiv version of [BHN19]. Basically, consider any dead-element $e' \in D$ and any actual element $e \in \mathcal{U}$ such that both their levels lie within the interval $[k - 1, k]$, i.e., $k - 1 \leq \ell(e), \ell(e') \leq k$. Then it follows that their weights $w_{e'}$ and w_e are within a $(1 + \epsilon)$ multiplicative factor of each other. This observation, along with Corollary 4.1, is sufficient to ensure that $\sum_{e \in D} w_e \leq 2\epsilon(1 + \epsilon) \cdot \sum_{e \in \mathcal{U}} w_e$. ∎

Lemma 4.2 *For every set $s \in \mathcal{S}$ at level $\ell(s) > 0$, we have $w_s^{(\mathcal{U} \cup D)} = 1$. Furthermore, for every set $s \in \mathcal{S}$ at level $\ell(s) = 0$, we have $w_s^{(\mathcal{U} \cup D)} \leq 1$.*

Lemma 4.2 (whose proof follows from induction) closely mirrors Lemma 2.2 from Section 2. Consider the collection of sets $\mathcal{T} = \{s \in \mathcal{S} : w_s^{(\mathcal{U} \cup D)} = 1\}$. Lemma 4.2 implies that (see the proof of Lemma 2.2) the element-weights $\{w_e\}$ form a valid fractional packing in $(\mathcal{U} \cup D, \mathcal{S})$ and $\mathcal{T}$ forms a valid set cover in $(\mathcal{U} \cup D, \mathcal{S})$. Furthermore, following the proof of Lemma 2.2, we get: $f \cdot \sum_{e \in \mathcal{U} \cup D} w_e \geq |\mathcal{T}|$. Applying Corollary 4.2, we now derive that:

$$f \cdot \sum_{e \in \mathcal{U}} w_e \geq (1 + 2\epsilon(1 + \epsilon))^{-1} \cdot f \cdot \sum_{e \in \mathcal{U} \cup D} w_e \geq (1 + 2\epsilon(1 + \epsilon))^{-1} \cdot |\mathcal{T}|. \tag{4.1}$$

Since $\mathcal{T}$ is a set cover in $(\mathcal{U} \cup D, \mathcal{S})$, it also forms a set cover in $(\mathcal{U}, \mathcal{S})$. Similarly, since the element-weights $\{w_e\}_{e \in \mathcal{U} \cup D}$ form a fractional packing in $(\mathcal{U} \cup D, \mathcal{S})$, the weights $\{w_e\}_{e \in \mathcal{U}}$ form a fractional packing in $(\mathcal{U}, \mathcal{S})$. Thus, we have a set cover $\mathcal{T}$ and a fractional packing $\{w_e\}$ in $(\mathcal{U}, \mathcal{S})$ whose sizes are within a $(1 + 2\epsilon(1 + \epsilon))f$ multiplicative factor of each other, according to (4.1). Hence, Lemma 2.1 implies that $\mathcal{T}$ forms a $(1 + 2\epsilon(1 + \epsilon))f$-approximate minimum set cover in $(\mathcal{U}, \mathcal{S})$.

Part II
FULL VERSION

Remark: There are a few minor notational inconsistencies between the extended abstract and this part of the paper. However, we emphasize that *this part consists of a self-contained full version of our two dynamic algorithms*, with every necessary notation and concept defined from scratch.

Organization: In Section A we describe the preliminary concepts and notations, along with a static primal-dual algorithm we will be building upon in subsequent sections. We present our dynamic algorithm for amortized update time in Section B, where Section B.2 contains the algorithm description and Section B.3 analyzes its amortized update time. Our dynamic algorithm for worst case update time is presented in Section C.

A   Preliminaries

In the *minimum set cover* problem, we get a universe of elements $\mathcal{U}$ and a collection of sets $\mathcal{S} \subseteq 2^{\mathcal{U}}$ as input, where $\bigcup_{s \in \mathcal{S}} s = \mathcal{U}$ and each set $s \in \mathcal{S}$ has a *cost* $c_s > 0$ associated with it. Without loss of generality, we assume there exists a parameter $C > 1$ such that $1/C < c_s < 1$ for all sets $s \in \mathcal{S}$, A collection of sets $\mathcal{S}' \subseteq \mathcal{S}$ forms a *set-cover* of $\mathcal{U}$ iff $\bigcup_{s \in \mathcal{S}'} s = \mathcal{U}$. The goal is to compute a set cover $\mathcal{S}'$ of $\mathcal{U}$ with minimum total cost $c(\mathcal{S}') = \sum_{s \in \mathcal{S}'} c_s$.

Dynamic Set Cover. Initially, the algorithm receives as input a universe of elements $\mathcal{U}$, a collection of sets $\mathcal{S} \subseteq 2^{\mathcal{U}}$ with $\bigcup_{s \in \mathcal{S}} s = \mathcal{U}$, and a cost $c_s \geq 0$ for each set $s \in \mathcal{S}$. Subsequently, the input keeps changing via a sequence of updates, where each update either (1) deletes an element e from the universe $\mathcal{U}$ and from every set $s \in \mathcal{S}$ that contains e, or (2) inserts an element e into the universe $\mathcal{U}$ and specifies the sets in $\mathcal{S}$ that the element e belongs to. After each update, we would like to maintain an approximate cost of the optimal set cover of the updated set system. The time taken by a dynamic algorithm to handle the preprocessing step is referred to as its *preprocessing time*. We say that a dynamic algorithm has an *amortized update time* of $O(\lambda)$ iff the algorithm takes $O(\Gamma + t \cdot \lambda)$ total time to handle any sequence of $t \geq 1$ updates after preprocessing, where $O(\Gamma)$ is preprocessing time. We want to design a dynamic algorithm whose approximation ratio and amortized update time are as small as possible.

Through out the remaining part of this paper, we use m to denote the number of sets in $\mathcal{S}$, and n to denote the maximum number of elements in the universe $\mathcal{U}$ across all the updates. We use f to denote an upper bound on the maximum frequency of any element across all the updates, where the frequency of an element $e \in \mathcal{U}$ is defined as the number of sets in $\mathcal{S}$ that contain e.

Fix $\epsilon > 0$ to be a sufficiently small constant, e.g., $\epsilon < 0.1$.

Hierarchical Decomposition. Let $L = \lceil \log_{1+\epsilon}(Cn) \rceil + 1$ and $[L] = \{0, 1, \ldots, L\}$. We call L the *highest* level and 0 the *lowest*. In the following we describe a static primal-dual algorithm assigns a level $l(s) \in [L]$ to every set $s \in \mathcal{S}$. We also define the level $l(e)$ of element $e \in \mathcal{U}$ as $l(e) = \max_{s \in \mathcal{S}: e \in S}\{l(s)\}$. The algorithm guarantees that each element e at level $l(e) = k$ has weight $(1 + \epsilon)^{-k}$. We use $w(s) = \sum_{e \in s} w(e)$ to denote the weight of $s \in \mathcal{S}$, which is the total weight of elements it contains. We call a set s *tight* if $w(s) > \frac{c_s}{1+\epsilon}$.

Initially, we set $l(e) \leftarrow L$ and $w(e) \leftarrow (1 + \epsilon)^{-L}$ for all elements $e \in \mathcal{U}$ and let $S \leftarrow \mathcal{S}$ and $E \leftarrow \mathcal{U}$. Note that at this point any set $s \in \mathcal{S}$ has weight

$$w(s) = \sum_{e \in s} w(e) \leq n \cdot (1 + \epsilon)^{-L} \leq \frac{1}{C} \leq c_s.$$

Subsequently, the algorithm proceeds in rounds $i = L, L - 1, \ldots, 0$. In round i, we identify sets in S that become tight, and move them from S to S_i. We set $l(s) \leftarrow i$ for all $s \in S_i$. Then we move the elements contained in the newly tight sets from E to E_i. Let $l(e) \leftarrow i$ for each $e \in E_i$. Then we raise the weights of all the remaining elements in E by a factor of $(1 + \epsilon)$, and proceed to the next round. The process stops when E becomes empty. Observe that in round 0, all elements in E has weight $(1 + \epsilon)^0 = 1 > c_s$. Hence every element will eventually be assigned some level in $[L]$. We claim that the collection of tight sets is a valid set cover and is a $(1 + \epsilon)f$-approximation.

Primal Dual Analysis. We abuse the notation slightly and use $w(X)$ to denote the total weight for any collection X of sets or elements. For any collection X of sets, we use $c(X)$ to denote their total cost. Let OPT be the cost of the minimum set cover. We have the following lemma.

Lemma A.1 *If we have $w(s) \leq (1 + \epsilon) \cdot c_s$ for every set $s \in \mathcal{S}$, then $w(\mathcal{U}) \leq (1 + \epsilon) \cdot \mathsf{OPT}$.*

Proof: Let $\mathcal{S}^*$ be the minimum set cover, i.e., $\mathsf{OPT} = c(\mathcal{S}^*)$. Then we have

$$w(\mathcal{U}) = \sum_{e \in \mathcal{U}} w(e) \leq \sum_{s \in \mathcal{S}^*} \sum_{e \in s} w(e) \leq (1 + \epsilon) \cdot c(\mathcal{S}^*) = (1 + \epsilon) \cdot \mathsf{OPT}.$$

The first inequality comes from the fact that $\bigcup_{s \in \mathcal{S}^*} s = \mathcal{U}$, and the second inequality comes from the assumption of the lemma. ∎

Hence any set cover S with $c(S) \leq \alpha \cdot w(\mathcal{U})$ is an $\alpha(1 + \epsilon)$-approximation. In particular, in the above hierarchical decomposition, the collection of tight sets $T \subseteq \mathcal{S}$ is a valid set cover since each element is contained in at least one tight set. Moreover, for every $s \in T$, we have $w(s) > \frac{c_s}{1+\epsilon}$. Hence T is a $(1 + \epsilon)^2 f$-approximate set cover since

$$c(T) \leq (1 + \epsilon) \sum_{s \in T} w(s) \leq (1 + \epsilon) f \cdot \sum_{e \in \mathcal{U}} w(e) \leq (1 + \epsilon)^2 f \cdot \mathsf{OPT}.$$

We will show that our dynamic algorithms maintains a similar hierarchical decomposition described above, and the collection of tight sets is a $(1 + O(\epsilon))f$-approximate set cover.

B   Our Algorithm for Amortized Update Time

In this section we present the data structure and the algorithm with amortized $O\left(\frac{f^2}{\epsilon^3} + \frac{f}{\epsilon^2} \log C\right)$ update time. Specifically, we prove the following theorem.

Theorem B.1 *There are deterministic dynamic algorithms for the minimum set cover problem with $(1 + \epsilon)f$-approximation ratio and an amortized update time of $O\left(\frac{f^2}{\epsilon^3} + \frac{f}{\epsilon^2} \log C\right)$.*

Our algorithm maintains the $(1 + \epsilon)f$-approximate set cover *value* after every update and can return it in constant time. The algorithm also maintains a *solution* of such value throughout the updates, and can output the change of the maintained solution after every update.

Notations. In addition to the *real weight* $w(s) = \sum_{e \in s} w(e)$, we also maintain for every set s a *dead weight* $\phi(s)$. We use $w^*(s) = w(s) + \phi(s)$ to denote its total weight. The idea of introducing a dead weight for each set s is to keep track of the weight that set s has lost due to decreases of element weight and element deletions. More specifically, when $w(s)$ decreases, we increase $\phi(s)$ such that $w(s) + \phi(s)$ does not decrease too much (so that we do not need to update our data structure immediately). As long as the total dead weight of sets is small, e.g., at most an ϵ fraction of the total weight of sets, then our data structure maintains a solution with bounded approximation ratio. We rebuild part of the data structure only when the fraction of dead weight gets too large.

B.1 Invariants

We define the weight of set s *at level i* as

$$w(s, i) = \sum_{e \in s} (1 + \epsilon)^{-\max\{i, \max_{s' \neq s: e \in s'} l(s')\}}.$$

In other words, $w(s, i)$ is the weight of s if we place s at level i. Note that $w(s) = w(s, l(s))$.

We maintain the following invariants.

Invariant B.1 (Bounded Weight Invariant) $\forall s \in \mathcal{S}$, $w(s, l(s) + 1) < c_s$.

For convenience we extend the definition of $w(s, i)$ to all positive integers i. Hence the invariant holds for all sets s at level L since $w(s, L + 1) \leq n \cdot (1 + \epsilon)^{-(L+1)} < \frac{1}{C} < c_s$.

Definition B.1 (Tight Sets) *We call a set s* tight *if $w(s) + \phi(s) > \frac{c_s}{1+\epsilon}$;* slack *otherwise.*

Invariant B.2 (Tightness Invariant) *All sets at level at least 1 are tight.*

Note that a tight set can be at level 0, but slack sets cannot be at level other than 0. With Invariant B.2 maintained, the collection of tight sets is a feasible set cover.

Corollary B.1 *Every element is contained in at least one tight set.*

Proof: Suppose there exits an element e such that all sets containing it are slack, then by Invariant B.2, we have $l(e) = 0$ and $w(e) = 1$. Hence each set s containing e has weight at least $1 \geq c_s$, which contradicts the definition of slack sets. ∎

Invariant B.3 (Local ϕ Invariant) *If $w(s) + \phi(s) > c_s$ then $\phi(s) = 0$.*

Invariant B.1 and B.3 imply the following immediately.

Corollary B.2 *We have $w(s) + \phi(s) < (1 + \epsilon) \cdot c_s$ for all sets $s \in \mathcal{S}$.*

Proof: If $w(s) + \phi(s) \leq c_s$ then the corollary trivially holds. Otherwise by Invariant B.3 we have $\phi(s) = 0$. By definition it is easy to prove that $w(s, i) \leq (1 + \epsilon) \cdot w(s, i+1)$ for all $i \in [L-1]$. Hence by Invariant B.1 we have $w(s) = w(s, l(s)) \leq (1 + \epsilon) \cdot w(s, l(s) + 1) < (1 + \epsilon) \cdot c_s$. ∎

Let T_i be the collection of tight sets at level i. Let $\Phi_i = \sum_{s:l(s)=i} \phi(s)$ be the total dead weight of sets at level i. Let $\Phi_{\leq k} = \sum_{i=0}^{k} \Phi_i$ and $T_{\leq k} = \sum_{i=0}^{k} T_i$. Let $\Phi = \Phi_{\leq L}$ and $T = T_{\leq L}$. We maintain the following invariant, which guarantees that the total dead weight is bounded.

Invariant B.4 (Global ϕ Invariant) *We have $\Phi \leq \epsilon \cdot \big(c(T) + f \cdot w(\mathcal{U})\big)$.*

With all the invariants maintained, we have the following approximation ratio guarantee.

Lemma B.1 *The collection of tight sets T is a $(1 + 5\epsilon)f$-approximate set cover.*

Proof: By definition of tight sets, we have $w(s) + \phi(s) > \frac{c_s}{1+\epsilon}$ for all $s \in T$. Hence the cost of the set cover is (where the third inequality comes from Invariant B.4)

$$c(T) < (1 + \epsilon) \cdot \sum_{s \in T} (w(s) + \phi(s)) \leq (1 + \epsilon) \cdot w(\mathcal{S}) + (1 + \epsilon) \cdot \Phi$$
$$\leq (1 + \epsilon)f \cdot w(\mathcal{U}) + \epsilon(1 + \epsilon) \cdot c(T) + \epsilon(1 + \epsilon)f \cdot w(\mathcal{U})$$
$$= (1 + \epsilon)^2 f \cdot w(\mathcal{U}) + \epsilon(1 + \epsilon) \cdot c(T).$$

Hence we have (for every $\epsilon \in (0, 0.1)$)

$$c(T) < \frac{(1 + \epsilon)^2 f}{1 - \epsilon(1 + \epsilon)} \cdot w(\mathcal{U}) \leq \frac{(1 + \epsilon)^3 f}{1 - \epsilon(1 + \epsilon)} \cdot \mathsf{OPT} \leq (1 + 5\epsilon)f \cdot \mathsf{OPT},$$

where the second inequality comes from the Lemma A.1 ∎

Recall that $E_i = \{e \in \mathcal{U} : l(e) = i\}$ contains the elements at level i. Let $E_i(s) = s \cap E_i$ be elements in s that are at level i. We maintain for every s the sets $E_i(s)$ for all $i \in [L]$. Recall that each element appears in at most f sets, and our data structure maintains for each element a pointer to each set containing it. It is easy to check throughout the algorithm that the data structure can be maintained efficiently when elements change their levels. Note that for all $i < l(s)$, $E_i(s) = \emptyset$.

Finally, we introduce the notion of base level.

Definition B.2 (Base Level) *For each set s, let $b(s) = \lfloor \log_{1+\epsilon} \frac{1}{c_s} \rfloor$ be the* base level *of s.*

Note that all base levels are at most $\log_{1+\epsilon} C = O(\frac{1}{\epsilon} \log C)$, since $c_s \in (\frac{1}{C}, 1)$ for all $s \in \mathcal{S}$.

Lemma B.2 *Invariant B.1 implies $E_i(s) = \emptyset$ for all $s \in \mathcal{S}$ and $i < b(s)$.*

Proof: Fix any set s. Each element at level $i \leq b(s) - 1$ has weight at least

$$(1 + \epsilon)^{-(b(s)-1)} \geq (1 + \epsilon) \cdot c_s.$$

Hence if $E_i(s)$ is not empty, then $w(s) \geq (1 + \epsilon) \cdot c_s$, which violates Corollary B.2. ∎

As we will present in the next section, our main algorithm consists of four subroutines, namely Insert, Delete, Promote and Rebuild. We summarize the invariants in the following table.

Invariant	Property	Affected by	Fixed (by)
Bounded Weight (Inv. B.1)	$\forall s: w(s, l(s) + 1) < c_s$	Insert	Promote
Tightness (Inv. B.2)	$\forall u: l(s) \geq 1 \to w(s) + \phi(s) > \frac{c_s}{1+\epsilon}$	All subroutines	immediately
Local ϕ (Inv. B.3)	$\forall s: w(s) + \phi(s) > c_s \to \phi(s) = 0$	All subroutines	immediately
Global ϕ (Inv. B.4)	$\Phi \leq \epsilon \cdot \big(c(T) + f \cdot w(\mathcal{U})\big)$	Promote, Delete	Rebuild

B.2 Algorithm

In this section we describe the algorithms to handle the updates. In addition to $\mathsf{Insert}(e)$ and $\mathsf{Delete}(e)$ that handle element insertions and deletions, we introduce two subroutines, $\mathsf{Promote}(s)$ and $\mathsf{Rebuild}(k)$, the former of which promotes a set s when its total weight gets too large, and the later rebuilds all levels on or below k when there are too much dead weight at levels at most k. Roughly speaking, the promotions of sets maintain the bounded weight invariant (Invariant B.1) and the rebuilds of levels maintain the global ϕ invariant (Invariant B.4). We maintain the tightness invariant (Invariant B.2) and local ϕ invariant (Invariant B.3) in all subroutines (namely, Delete, Insert, $\mathsf{Promote}$ and $\mathsf{Rebuild}$).

The main algorithm is shown as follows (refer to Algorithm 1).

Algorithm 1 DynamicsetCover

1: initialize $l(s) = 0$ for all $s \in \mathcal{S}$
2: **for** each update **do**
3:     **if** element e is deleted **then**
4:         $\mathsf{Delete}(e)$.
5:     **if** element e is inserted **then**
6:         $\mathsf{Insert}(e)$.
7:         **while** exits set s for which Invariant B.1 is violated **do**
8:             $\mathsf{Promote}(s)$.
9:     **while** Invariant B.4 is violated **do**
10:         find the smallest k for which $\Phi_{\leq k} > \epsilon \cdot (c(T_{\leq k}) + f \cdot w(E_{\leq k}))$.
11:         $\mathsf{Rebuild}(k)$.

As we will show later, the bounded weight invariant (Invariant B.1) can be violated for a set s only after an element contained in s is inserted. In other words, line 8 of Algorithm 1 will be executed on set s only if s contains the element e that is inserted.

In the following we describe the four subroutines one by one.

B.2.1 Deletions

Suppose element e is deleted. By definition, each s containing e has its real weight $w(s)$ decreases by $w(e)$. On the other hand, we increase its dead weight $\phi(s)$ accordingly. We also exclude e from the list of elements $E_{l(e)}(s)$ maintained by each set s containing e[4]. Since each element is contained in at most f sets and it takes $O(1)$ time to handle each set containing e, the following algorithm runs in $O(f)$ time.

When an element e is deleted, for each s containing e our algorithm changes $w(e)$ unit of real weight of s to dead weight. If local ϕ invariant (Invariant B.3) is violated then we decreases $\phi(s)$: we have either $w(s) + \phi(s) \leq c_s$ or $\phi(s) = 0$ after line 6 of Algorithm 2. Note that if a set is at level 0, then we do not increase its dead weight (which stays 0) since we do not need to maintain its tightness. Consequently each set at level at least 1 remains tight after the deletion. That is, the tightness invariant (Invariant B.2) is maintained. Note that the bounded weight invariant (Invariant B.1) also holds since $w(s, l(s) + 1)$ does not increase after the deletion. However, since the total dead weight is increased, the global ϕ invariant (Invariant B.4) can possibly be violated. As we will show later, the invariant will be maintain by rebuilding some levels.

---

[4]Every time when an element changes its level, or gets inserted or deleted, we change these element collections (and necessary pointers) accordingly. For ease of presentation we omit these steps in the description of all our algorithms.

Algorithm 2 Delete(e)

1: **for** each s containing e **do**
2:     $w(s) \leftarrow w(s) - w(e)$.
3:     **if** $l(s) > 0$ **then**
4:         $\phi(s) \leftarrow \phi(s) + w(e)$.
5:         **if** $w(s) + \phi(s) > c_s$ **then**
6:             $\phi(s) \leftarrow \max\{0, c_s - w(s)\}$.

B.2.2   Insertions

Suppose element e is inserted. We consider any s containing e. If we have $w(s) \leq c_s$ after the insertion then we can easily fix all the invariants (regarding set s) by decreasing $\phi(s)$ (if necessary). Suppose that after the insertion $w(s) > c_s$. As we will show in our potential analysis, we can upper bound the increase in the potential of s by $O(\frac{f^2}{\epsilon^3} + \frac{f}{\epsilon^2} \log C)$ if $E_{l(s)}(s) \neq \emptyset$ *before* the insertion.

We call a set *good* if one of the following properties holds, and *bad* otherwise.

Definition B.3 (Good Sets) *Suppose we insert a new element e at level k. We call set s containing e good before the insertion if (1) $w(s) + (1 + \epsilon)^{-k} \leq c_s$; or (2) $E_{l(s)}(s) \neq \emptyset$.*

Before inserting a new element e, we would like to make sure that all sets containing e are good. Note that the definition of "good" depends on the level the new element is inserted into. The higher level the element e is inserted into, the more sets containing e will be good. We can insert e at a high level only by increasing the levels of sets containing e, before the element is inserted.

Our algorithm increases the level of each bad set s to the lowest level i on which it becomes good. As s does not contain any element at the level on which it is bad, increasing $l(s)$ does not change the weight of any element or set, and thus can be done in $O(1)$ time.

We call such an increase of $l(s)$ a *lift-up* of s, and we charge s one unit of *lift-up cost*. Note that if $l(s) < b(s)$, then we can directly increase $l(s)$ to $b(s)$, since by Lemma B.2 $E_i(s) = \emptyset$ for all $i < b(s)$, and $w(s) + (1 + \epsilon)^{-i} > c_s$ for all $i < b(s)$.

We denote the operation (which increases $l(s)$ by at least one) by Lift-Up(s).

Algorithm 3 Lift-Up(s)

1: **if** $l(s) < b(s)$ **then**
2:     $l(s) \leftarrow b(s)$.
3: **else**
4:     $l(s) \leftarrow l(s) + 1$.

A lift-up of s will be called only when $E_{l(s)}(s) = \emptyset$. In addition, for potential analysis purpose (which will be clear later), we lift a set s up only when $\phi(s) = 0$. Thus before lifting the set s up, we need to remove its dead weight. On the other hand, since $w(s) + \phi(s)$ is decreased when we set $\phi(s)$ to 0, we need to ensure that s is tight after the insertion of the element, as we have $l(s) > 0$ after the lift-up. More specifically, suppose we remove the dead weight of s and its real weight before the insertion is $w(s)$. Then our algorithm guarantees that the element is inserted at a level k such that $w(s) + (1 + \epsilon)^{-k} > \frac{c_s}{1+\epsilon}$, i.e., s is tight after the insertion.

The detailed description can be found in Algorithm 4. We lift sets up in a carefully chosen order, such that all sets are good before the element is inserted.

More specifically, let k be the tentative level for the new element. As the first step, we identify the collection of bad sets B containing e. Note that since our algorithm does not decrease the

tentative level k, all sets containing e that are not in B will be good when k settles at its final level. We consider the sets in B one by one in non-decreasing order of $c_s - w(s)$. Intuitively, the set s with the smallest $c_s - w(s)$ is most sensitive to the increase in weight due to the element insertion, and hence should be considered first.

Suppose a set $s \in B$ is considered. We lift s up until it becomes good, i.e., when either (1) $E_{l(s)}(s) \neq \emptyset$; or (2) the lift-up of s increases the tentative level k of e such that $w(s) + (1+\epsilon)^{-k} \leq c_s$. In the first case s stops at level $l(s)$ and will be good no matter what the final value of k will be. In the later case all sets contained in B are good, as s is considered as the bad set with the minimum $c_s - w(s)$, and with the current tentative level k it holds that $w(s) + (1+\epsilon)^{-k} \leq c_s$. In other words, k is so large that each set $s' \in B$ not considered yet has $w(s') + (1+\epsilon)^{-k} \leq c_{s'}$, i.e., s' is good. Hence no more lift-up will happen. This is the reason why we consider sets in the order specified in line 3 of Algorithm 4.

Algorithm 4 Insert(e)

1: $k \leftarrow \max_{s:e \in s}\{l(s)\}$.
2: Let $B = \{s_1, s_2, \ldots, s_b\}$ be the collection of bad sets containing e, where $b = |B| \leq f$.
3: Sort the sets in B and assume w.l.o.g. that $c_{s_1} - w(s_1) \leq c_{s_2} - w(s_2) \leq \ldots \leq c_{s_b} - w(s_b)$.
4: **for** $i = 1, 2, \ldots, b$ **do**
5:     **while** s_i is bad **do**
6:         $\phi(s_i) \leftarrow 0$, Lift-Up(s_i).
7:         $k \leftarrow \max\{k, l(s_i)\}$.
8: $l(e) \leftarrow k, w(e) \leftarrow (1+\epsilon)^{-k}$.
9: **for** each s containing e **do**
10:     $w(s) \leftarrow w(s) + w(e)$.
11:     **if** $w(s) + \phi(s) > c_s$ **then**
12:         $\phi(s) \leftarrow \max\{0, c_s - w(s)\}$.

Note that it takes $O(f \log f)$ time to sort the sets in B in line 3 of Algorithm 4. Hence excluding the lift-up costs (which are charged to the sets), Algorithm 4 runs in $O(f \log f)$ time. Note that to ease the analysis we also consider line 7 to be part of the list-up cost.

Since the insertion does not create any dead weight or change the level of any existing element, the global ϕ invariant (Invariant B.4) remains valid. The local ϕ invariant (Invariant B.3) is maintained because if $w(s) + \phi(s) > c_s$ after the insertion then we decrease $\phi(s)$ such that either $\phi(s) = 0$ or $w(s) + \phi(s) = c_s$.

Next we show that the tightness invariant (Invariant B.2) is maintained, and all sets are good before the element is inserted. Recall that we remove the dead weight of each set that is lifted up, which might make a set slack. We show in Lemma B.3 that each set whose dead weight is removed will be tight after the insertion of the new element (which maintains the tightness invariant).

Lemma B.3 *Before the new element e is inserted (in line 8 of Algorithm 4), all sets containing e are good. Moreover, the tightness invariant is maintained after the insertion.*

Proof: As we have argued in the above discussion, since the tentative level k does not decrease, if a set is good at some point, then it remains good afterwards. Hence our algorithm ensures that all sets containing e are good before the element is inserted.

Next we prove that the tightness invariant is maintained by showing that all sets lifted up are tight after the element insertion. More specifically, let k^* be the final level of e, i.e., $w(e) = (1+\epsilon)^{-k^*}$. We show that if we lift set s up, then $w(s) + w(e) > \frac{c_s}{1+\epsilon}$ at the end of the algorithm. For sets not

lifted up, the tightness invariant trivially holds since if a set is at level at least 1 then it is tight before the element is inserted, and the algorithm does not decrease its real weight or dead weight.

Recall that in the while loop in which s is chosen, our algorithm increases the level $l(s)$ of s (which may also increase k) until either (1) $E_{l(s)}(s) \neq \emptyset$, or (2) $w(s) + (1 + \epsilon)^{-k} \leq c_s$.

If the algorithm never increases k, then the lemma easily follows since each set $s_i \in B$ is bad initially, which implies $w(s_i) + (1 + \epsilon)^{-k^*} > c_{s_i}$. Otherwise we consider the last while loop in which k is increased (to k^*) because some set s_i is lifted up.

If s_i is lifted up from level 0 to its base level, then we have

$$w(s_i) + (1 + \epsilon)^{-k^*} = w(s_i) + (1 + \epsilon)^{-b(s)} \geq c_{s_i} > \frac{c_{s_i}}{1 + \epsilon}. \tag{B.1}$$

Otherwise s_i is lifted up by exactly one level, and we have

$$w(s_i) + (1 + \epsilon)^{-(k^*-1)} > c_{s_i},$$

which implies

$$w(s_i) + (1 + \epsilon)^{-k^*} \geq \frac{w(s_i)}{1 + \epsilon} + (1 + \epsilon)^{-k^*} > \frac{c_{s_i}}{1 + \epsilon}.$$

In both case we have $w(s_i) + (1 + \epsilon)^{-k^*} > \frac{c_{s_i}}{1+\epsilon}$. That is, s_i is tight after the insertion. Next we show that every $s_j \neq s_i$ that is lifted up is also tight after the insertion.

Consider any s_j with $j > i$. That is, s_j is considered after s_i. Since $k = k^*$ when s_j is considered in the while loop, if s_j is lifted up, then s_j is bad and we have $w(s_j) + (1 + \epsilon)^{-k^*} > c_{s_j}$, as required.

Now consider any s_j with $j < i$, i.e., s_j is considered before s_i. Then we have $c_{s_j} - w(s_j) \leq c_{s_i} - w(s_i)$, which implies

$$w(s_j) + (1 + \epsilon)^{-k^*} \geq \frac{w(s_j)}{1 + \epsilon} + (1 + \epsilon)^{-k^*} \geq \frac{c_{s_j} - c_{s_i} + w(s_i)}{1 + \epsilon} + (1 + \epsilon)^{-k^*} > \frac{c_{s_j}}{1 + \epsilon},$$

where the last inequality comes from Inequality (B.1). $\blacksquare$

The bounded weight invariant (Invariant B.1) might be violated due to the increase of set weight, in which case we invoke the promotion subroutine as follows.

B.2.3 Promotion

Recall that if the bounded weight invariant does not hold for set s, then we have $w(s, l(s) + 1) \geq c_s$ and $\phi(s) = 0$ (by the local ϕ invariant). In this case we call Promote(s), which increases the level of s, and decreases $w(e)$ for all $e \in E_{l(s)}(s)$. In the meanwhile, the subroutine increases the dead weight $\phi(s')$ of each $s' \neq s$ containing e accordingly to compensate for the decrease in $w(s')$, which maintains the tightness invariant (Invariant B.2).

If $E_{l(s)}(s) = \emptyset$ then we first lift s up to a level i such that $E_i(s) \neq \emptyset$. As before, we charge the lift-up cost to set s.

If $E_k(s) \neq \emptyset$, the update time for promoting set s from level k to $k + 1$ is $O(f \cdot |E_k(s)|)$.

As before, the local ϕ invariant (Invariant B.3) is maintained since if $w(s') + \phi(s') > c_{s'}$ then we decrease $\phi(s')$ such that either $\phi(s') = 0$ or $w(s') + \phi(s') = c_{s'}$ (in line 13 of Algorithm 5). As the promotion does not decrease $w(x) + \phi(x)$ to below c_x for $x = s$ or any set at level at least 1 intersecting s, the tightness invariant (Invariant B.2) remains valid. By repeatedly calling the promotion subroutine, we can also maintain the bounded weight invariant (Invariant B.1).

The global ϕ invariant (Invariant B.4) might be violated due to the increase of dead weight, in which case we invoke the rebuild subroutine as follows.

Algorithm 5 Promote(s)

1: **while** $E_{l(s)}(s) = \emptyset$ **do**
2:     Lift-Up(s).
3: $k \leftarrow l(s)$.
4: **while** $w(s, k+1) \geq c_s$ **do**
5:     **for** each $e \in E_k(s)$ **do**
6:         $l(e) \leftarrow k+1$, $w(e) \leftarrow (1+\epsilon)^{-(k+1)}$.
7:         $w(s) \leftarrow w(s) - \epsilon(1+\epsilon)^{-(k+1)}$.
8:         **for** each $s' \neq s$ containing e **do**
9:             $w(s') \leftarrow w(s') - \epsilon(1+\epsilon)^{-(k+1)}$
10:             **if** $l(s') > 0$ **then**
11:                 $\phi(s') \leftarrow \phi(s') + \epsilon(1+\epsilon)^{-(k+1)}$.
12:                 **if** $w(s') + \phi(s') > c_{s'}$ **then**
13:                     $\phi(s') \leftarrow \max\{0, c_{s'} - w(s')\}$.
14:     $l(s) \leftarrow k+1$, $k \leftarrow k+1$.

B.2.4 Rebuild

Our algorithm maintains Φ_i, T_i, E_i and $c(T_i)$ explicitly for all $i \in [L]$. In addition, we maintain $\Phi = \Phi_{\leq L}$, $c(T) = c(T_{\leq L})$ and $w(\mathcal{U}) = \sum_{e \in \mathcal{U}} w(e)$. Recall that we perform a rebuild only when the global ϕ invariant (Invariant B.4) does not hold, i.e., $\Phi > \epsilon \cdot (c(T) + f \cdot w(\mathcal{U}))$, which can be checked in $O(1)$ time. When $\Phi > \epsilon \cdot (c(T) + f \cdot w(\mathcal{U}))$, we find the smallest k for which $\Phi_{\leq k} > \epsilon \cdot (c(T_{\leq k}) + f \cdot w(E_{\leq k}))$, and rebuild all levels $i \leq k$. Trivially such a level can be found in $O(k)$ time (by scanning the levels in a bottom-up manner). In the following, we show that such a level can be found in $O(|T_{\leq k}|)$ time (which is smaller than $O(k)$ when most of the T_i's are empty). Note that $O(|T_{\leq k}|) = O(f \cdot |E_{\leq k}|)$.

As we will show later, Rebuild(k) takes time $O(\frac{f}{\epsilon^2} \cdot |E_{\leq k}| + \frac{1}{\epsilon} \log C)$, and we consider the time spent on finding the level k to rebuild part of the rebuild cost.

Skipping Empty Levels. We keep the pointers to $T_0, T_1, \ldots, T_L$ as an array of size $L+1$. Additionally we organize the pointers that point to non-empty sets in a doubly linked list. Note that if there exists an element at level i then there must also exists a set at level i. Hence for all $i \geq 1$ if E_i is non-empty then T_i is also non-empty. In other words, we only need to look at the levels i for which T_i are non-empty (following the linked list), and compute $\Phi_{\leq k}$, $c(T_{\leq k})$ and $w(E_{\leq k}) = \sum_{i=1}^{k} |E_i| \cdot (1+\epsilon)^{-i}$ in $O(|T_{\leq k}|)$ time.

Maintaining the Linked List. Note that deletions can change a set from tight to slack only if the set is at level 0. Hence it suffices to check whether T_0 is empty after each deletion, and update the linked list in $O(1)$ time. After Rebuild(k), we can reconstruct T_i for all $i \in [k]$, and update the linked list in $O(f \cdot |E_{\leq k}|)$ time (which becomes part of the rebuild cost). Similar to deletions, the insertion of an element can only change T_0, and the linked list can be updated in $O(1)$ time. However, the insertion may trigger lift-ups or promotions of sets, which may change the collections of tight sets and the linked list.

The case when a set changes its level by one is easy. Suppose a set s is promoted or lifted up from level k to $k+1$. Then only s changes its level, and only tight sets in level 0 can change to slack. Moreover, the collections of tight sets at each level $i \geq 1$ remain unchanged. Thus it suffices

to check T_0, T_k and T_{k+1}, and update the linked list in $O(1)$ time.

Now suppose a set s at level 0 is lifted up to its base level $b(s)$, which happens when a new element e contained in s is inserted, and $w(s) + w(e) > 1$. If s was tight at level 0 then we first check whether T_0 is empty after the lift-up, and update the linked list. If $T_{b(s)}$ was non-empty before s is lifted up then we can simply include s in $T_{b(s)}$ without modifying the linked list further. Otherwise we find the largest $i \leq b(s)$ for which T_i is non-empty (which takes $O(b(s))$ time), and insert the pointer to $T_{b(s)}$ to the linked list in $O(1)$ time. Recall that $O(b(s)) = O(\frac{1}{\epsilon} \log C)$, and we charge this update cost to the insertion. Since each insertion causes $O(f)$ sets at level 0 being lifted up, each insertion is charged at most $O(\frac{f}{\epsilon} \log C)$ due to the maintenance of the linked list.

Properties after Rebuild. Let S be the sets at level at most k before $\mathsf{Rebuild}(k)$ is triggered. We show that after $\mathsf{Rebuild}(k)$, we have the following properties, where $l(s)$ denotes the level of set s after the rebuild.

- For all $s \in S$, $l(s) \leq k$ and $\phi(s) = 0$.

- For all $s \in S$ and $l(s) = k$, $w(s) > \frac{c_s}{1+\epsilon}$ and $w(s, k+1) < c_s$.

- For all $s \in S$ and $l(s) < k$, $w(s) < c_s$. Moreover, if $l(s) > 0$ then $w(s) > \frac{c_s}{1+\epsilon}$.

Consequently, the bounded weight invariant, tightness invariant and local ϕ invariant are maintained. Moreover, since the subroutine removes all dead weight at level at most k, by repeatedly calling the subroutine (at higher and higher levels), the global ϕ invariant can also be maintained.

Next, we describe the details of the rebuild subroutine, which is very similar to the rebuild subroutine of [BHN19]. On a high level, the subroutine has two steps. In the first step, we put all elements and sets at level k, and remove all dead weight, which decrease the total weight of some sets. Then in the second step we move the slack sets to lower levels (which increases the weight of elements and sets) until they become tight, or reach level 0. The second step is done by calling the subroutine $\mathsf{Fix\text{-}Level}$, which we borrow from [BHN19].

Lemma B.4 *[BHN19, Section B.1] There exists a subroutine $\mathsf{Fix\text{-}Level}$ with the following properties. $\mathsf{Fix\text{-}Level}(k, S', E')$ takes as input a collection of sets S' and a collection of elements E', both are at level k. Moreover, each set $s \in S'$ has $w(s) < c_s$ and $\phi(s) = 0$. The subroutine places each set $s \in S'$ at level $l(s)$ such that (1) $w(s) < c_s$ and (2) if $l(s) > 0$ then $w(s) > \frac{c_s}{1+\epsilon}$. The subroutine runs in time $O(f \cdot |E'| + k)$.*

We call an element *slack* if all sets containing it are slack.

A key difference between our algorithm and that of [BHN19] is, before calling the subroutine $\mathsf{Fix\text{-}Level}$ on the collection of slack sets S' and the slack elements E', we first move all elements in E' and sets in S' to level $k' = \min\{k, \log_{1+\epsilon} \frac{2C \cdot |E'|}{\epsilon}\}$. As we will show in Lemma B.5, the operation does not break any invariants, but helps improve the update time for $\mathsf{Fix\text{-}Level}$.

Note that since each set $s \in S'$ has no dead weight, its total weight equals to its real weight.

Lemma B.5 *Before calling $\mathsf{Fix\text{-}Level}(k', S', E')$ in line 13, each $s \in S'$ has weight less than c_s.*

Proof: In line 6 of Algorithm 6, we identify the slack sets, i.e., sets s with $w(s) \leq \frac{c_s}{1+\epsilon}$. We then move all slack sets and elements containing only slack sets to level k'. If $k' = k$ then no movement is made and the lemma trivially holds.

Suppose $k' = \log_{1+\epsilon} \frac{2C \cdot |E'|}{\epsilon} < k$. Note that each $e \in E'$ has weight $(1 + \epsilon)^{-k'} = \frac{\epsilon}{2C \cdot |E'|}$ on level k'. Hence the movement increases the weight of each $s \in S'$ by at most $\frac{\epsilon}{2C} \leq \frac{\epsilon \cdot c_s}{2}$, as there

25

Algorithm 6 Rebuild(k)

1: let $\hat{S} = \{s : l(s) \leq k\}$ be the sets at levels at most k. $\triangleright |\hat{S}| = O(f \cdot |E_{\leq k}|)$
2: **for** each $s \in \hat{S}$ **do**
3:      $\phi(s) \leftarrow 0$, $l(s) \leftarrow k$.
4: **for** each $e \in E_{\leq k}$ **do**
5:      $l(e) \leftarrow k$, $w(e) \leftarrow (1+\epsilon)^{-k}$ (and decrease $w(s)$ by the same amount for all s containing e).
6: let $S' \leftarrow \{s \in \hat{S} : w(s) \leq \frac{c_s}{1+\epsilon}\}$ be the collection of slack sets.
7: let $E' \subseteq E_{\leq k}$ be slack elements.
8: let $k' = \min\{k, \log_{1+\epsilon} \frac{2C \cdot |E'|}{\epsilon}\}$.
9: **for** each $e \in S'$ **do**
10:      $l(s) \leftarrow k'$.
11: **for** each $e \in E'$ **do**
12:      $l(e) \leftarrow k'$, $w(e) \leftarrow (1+\epsilon)^{-k'}$ (and decrease $w(s)$ by the same amount for all s containing e).
13: Fix-Level(k', S', E'). $\triangleright O(f \cdot |E'| + k')$ time
---

are at most $|E'|$ elements contained in s that are moved to level k'. Since $w(s) \leq \frac{c_s}{1+\epsilon}$ before the movement, set s has weight $\leq c_s\left(\frac{1}{1+\epsilon} + \frac{\epsilon}{2}\right) < c_s$ after the movement of elements, as claimed. $\blacksquare$

By definition $k' \leq \log_{1+\epsilon} \frac{2C|E'|}{\epsilon} = O(\frac{|E'|}{\epsilon^2} + \frac{1}{\epsilon}\log C)$, and thus Fix-Level($k', S', E'$) runs in time $O(f \cdot |E'| + k') = O(\frac{f}{\epsilon^2} \cdot |E'| + \frac{1}{\epsilon}\log C)$. In summary, Rebuild($k$) runs in $O(\frac{f}{\epsilon^2} \cdot |E_{\leq k}| + \frac{1}{\epsilon}\log C)$ time, and cleans up all dead weight at levels at most k. Moreover, the weight invariant and the tightness invariant are maintained. By repeatedly invoking the subroutine, we also maintain the global ϕ invariant.

B.3 Potential Analysis and Amortized Update Time

We use a potential analysis to bound the total update time. On a high level, we have two types of potentials. Each set s has potential if $w(s) > c_s$, which we call *up potential*. Each unit of dead weight also has some potential, which we call *down potential*. We show that

- one unit of down potential will be used to pay for $O(1)$ update time (in Section B.3.2);

- one unit of up potential will be transformed into at most one unit of down potential and in the meantime pay for $O(1)$ update time (in Section B.3.3).

Specifically, the up potential is released to pay for the update cost of promotions of sets and the increase in down potential due to the promotions. The down potential is released to pay for update time of the rebuilds of levels and the lift-up costs. In other words, the up and down potential are consumed during promotions and rebuilds. Hence it suffices to upper bound the amount of potential we gain from element insertions and deletions. We show in Section B.3.4 that

- each insertion does not increase the total down potential, and increases the total up potential by $O(\frac{f^2}{\epsilon^3} + \frac{f}{\epsilon^2}\log C)$;

- each deletion does not increase the total up potential, and increases the total down potential by $O(\frac{f}{\epsilon^3} + \frac{1}{\epsilon^2}\log C)$.

Since each unit of potential is used to pay for $O(1)$ units of update time, the amortized update time is bounded by $O(\frac{f^2}{\epsilon^3} + \frac{f}{\epsilon^2}\log C)$.

B.3.1 Potential Function

We first define the up potential, i.e., potentials of sets s with weight $w(s) > c_s$. Recall that the local ϕ invariant guarantees $\phi(s) = 0$ for each set s with $w(s) > c_s$.

If set s has weight $w(s) \le c_s$ then it has 0 potential; otherwise its potential is given as follows. Recall that $w(s, i)$ is the weight of s if we set $l(s) = i$. Hence $w(s, i)$ is a non-increasing function of i. Suppose $w(s) = w(s, l(s)) > c_s$. Let $\hat{l}(s)$ be the highest level such that $w(s, \hat{l}(s)) \ge c_s$. Let

$$\alpha_i = \left(\frac{f}{\epsilon^3} + \frac{1}{\epsilon^2} \log C \right) (1 + \epsilon)^{i+1}$$

be the *conversion ratio* from set weight to up potential at level i. We define the potential of s as

$$\sum_{i=l(s)}^{\hat{l}(s)-1} \big(w(s, i) - w(s, i+1) \big) \cdot \alpha_i + \big(w(s, \hat{l}(s)) - c_s \big) \cdot \alpha_{\hat{l}(s)}.$$

As it will be clear from our later analysis, the up potential of set s is defined such that the potential released by $\mathsf{Promote}(s)$ is at least the update time and the increase in total down potential due to the promotion.

The down potential is defined as follows. Let

$$\beta_i = \frac{1}{f} \cdot \alpha_i = \left(\frac{1}{\epsilon^3} + \frac{1}{\epsilon^2 f} \log C \right) (1 + \epsilon)^{i+1}$$

be the *conversion ratio* from dead weight to down potential at level i. The dead weight $\phi(s)$ of set s has potential $\phi(s) \cdot \beta_{l(s)}$. In other words, each unit of dead weight at level i has potential β_i.

B.3.2 Rebuilds

Recall that $\mathsf{Rebuild}(k)$ takes time $O(\frac{f}{\epsilon^2} \cdot |E_{\le k}| + \frac{1}{\epsilon} \log C)$, and cleans up the dead weights at level at most k. Also recall that a set s is charged one unit of lift-up cost each time $\mathsf{Lift\text{-}Up}(s)$ is called, which increases $l(s)$ by at least one. Moreover, if $l(s) < b(s)$, then $\mathsf{Lift\text{-}Up}(s)$ increases $l(s)$ to $b(s)$ immediately. Hence from the last time set s is involved in some rebuild, at most $\max\{l(s) - b(s) + 1, 0\}$ units of lift-up costs are charged to s.

We show that (1) $\mathsf{Rebuild}(k)$ does not increase the up potential of any set; (2) the down potential released by dead weight $\Phi_{\le k}$ is sufficient to pay for the rebuild cost and the lift-up cost charged to sets in $T_{\le k}$.

Lemma B.6 *$\mathsf{Rebuild}(k)$ does not increase the up potential of any set.*

Proof: Observe that $\mathsf{Rebuild}(k)$ does not change the up potential of sets at levels higher than k. Moreover, after the rebuild, each set s at level at most $k - 1$ has weight at most c_s, and has 0 up potential. Consider any set s that is at level k after the rebuild:

- if set s was at level $i \le k - 1$ before the rebuild, then we show that we have $w(s) < c_s$ after the rebuild. Recall that in the first step of rebuild (line 2 of Algorithm 6), we put all elements and sets at level k. By the bounded weight invariant, we have $w(s, k) \le w(s, i+1) < c_s$. Since s stays at level k afterwards, its weight does not increase in the later steps of $\mathsf{Rebuild}(k)$. Hence after the rebuild we have $w(s) < c_s$, which implies that s has 0 potential.

- if set s was at level k before the rebuild, then the first step of rebuild does not change $w(s)$ or the level of any element contained in s, since they were at level k or higher. Hence s have the same potential after the rebuild.

In summary, $\mathsf{Rebuild}(k)$ does not increase the up potential of any set. ∎

Lemma B.7 *The down potential of $\Phi_{\leq k}$ is at least*

$$\frac{f}{\epsilon^2} \cdot |E_{\leq k}| + \frac{1}{\epsilon} \log C + \sum_{s \in T_{\leq k}} \max\{l(s) - b(s) + 1, 0\}.$$

Proof: Recall that the conversion ratio of dead weight to down potential is smaller at lower levels. Hence given $\Phi_{\leq k} > \epsilon \cdot (c(T_{\leq k}) + f \cdot w(E_{\leq k}))$ and the fact that k is the smallest level for which the inequality holds, the down potential of $\bar{\Phi}_{\leq k}$ is minimized when $\Phi_i = \epsilon \cdot (c(T_i) + f \cdot w(E_i))$ for all $i \in [k-1]$. Besides, we have $\Phi_k > \epsilon \cdot (c(T_k) + f \cdot w(E_k))$.

Let $t \in [k]$ be an arbitrary level for which $E_t \neq \emptyset$. The down potential of $\Phi_{\leq k}$ is at least (recall that $\beta_i = \left(\frac{1}{\epsilon^3} + \frac{1}{\epsilon^2 f} \log C\right)(1+\epsilon)^{i+1}$)

$$\epsilon \cdot \sum_{i=0}^{k} \sum_{s \in T_i} c_s \cdot \beta_i + \epsilon f \cdot \sum_{i=0}^{k} \sum_{e \in E_i} w(e) \cdot \beta_i$$

$$\geq \epsilon \cdot \sum_{i=0}^{k} \sum_{s \in T_i} c_s \cdot \frac{(1+\epsilon)^{i+1}}{\epsilon^3} + \epsilon f \cdot \sum_{i=0}^{k} \sum_{e \in E_i} w(e) \cdot \frac{(1+\epsilon)^{i+1}}{\epsilon^3} + \epsilon f \cdot (1+\epsilon)^{-t} \cdot \frac{(1+\epsilon)^{t+1}}{\epsilon^2 f} \log C$$

$$\geq \sum_{i=0}^{k} \sum_{s \in T_i} \frac{(1+\epsilon)^{i-b(s)}}{\epsilon^2} + f \cdot \sum_{i=0}^{k} \sum_{e \in E_i} (1+\epsilon)^{-i} \cdot \frac{(1+\epsilon)^{i+1}}{\epsilon^2} + \frac{1+\epsilon}{\epsilon} \log C$$

$$\geq \sum_{i=0}^{k} \sum_{s \in T_i} \cdot \frac{\epsilon \cdot \max\{i - b(s) + 1, 0\}}{\epsilon^2} + \frac{f}{\epsilon^2} |E_{\leq k}| + \frac{1}{\epsilon} \log C$$

$$\geq \sum_{s \in T_{\leq k}} \max\{l(s) - b(s) + 1, 0\} + \frac{f}{\epsilon^2} |E_{\leq k}| + \frac{1}{\epsilon} \log C,$$

where the third inequality comes from $(1+\epsilon)^x \geq 1 + \epsilon \cdot x \geq \epsilon \cdot (x+1)$ for all $x \geq 0$. ∎

In summary, $\mathsf{Rebuild}(k)$ does not increase the up potential, and releases an amount of down potential that is sufficient to pay for the rebuild cost and the lift-up cost charged to sets in $T_{\leq k}$.

B.3.3 Promotions

Suppose we promote s from level k to $k+1$. Recall that the update time is $O(f \cdot |E_k(s)|)$. Also recall that the promotion of s from level k to $k+1$ increases the total dead weight by at most $f \cdot |E_k(s)| \cdot \epsilon(1+\epsilon)^{-(k+1)}$. Since all these increased dead weights are at level at most k, the total down potential is increased by at most

$$f \cdot |E_k(s)| \cdot \epsilon(1+\epsilon)^{-(k+1)} \cdot \left(\frac{1}{\epsilon^3} + \frac{1}{\epsilon^2 f} \log C\right)(1+\epsilon)^{k+1} = \left(\frac{f}{\epsilon^2} + \frac{1}{\epsilon} \log C\right) \cdot |E_k(s)|.$$

In the following, we show that promoting s from level k to $k+1$ does not increase the up potential of any set intersecting s, and releases at least $\left(\frac{f}{\epsilon^2} + \frac{1}{\epsilon} \log C\right) \cdot |E_k(s)|$ up potential of s.

28

Lemma B.8 *Promoting s from level k to $k+1$ releases $\left(\frac{f}{\epsilon^2} + \frac{1}{\epsilon} \log C\right) \cdot |E_k(s)|$ up potential of s.*

Proof: Note that promoting s from level k to $k+1$ does not change $\hat{l}(s)$ or $w(s, i)$ for all $i \geq k+1$. Hence by definition, the up potential released from set s is

$$\left(w(s,k) - w(s,k+1)\right) \cdot \left(\frac{f}{\epsilon^3} + \frac{1}{\epsilon^2} \log C\right)(1+\epsilon)^{k+1}$$

$$= \epsilon(1+\epsilon)^{-(k+1)} \cdot |E_k(s)| \cdot \left(\frac{f}{\epsilon^3} + \frac{1}{\epsilon^2} \log C\right)(1+\epsilon)^{k+1} = \left(\frac{f}{\epsilon^2} + \frac{1}{\epsilon} \log C\right) \cdot |E_k(s)|,$$

which concludes the lemma. ∎

It remains to show that the up potentials of other sets are not increased due to the promotion. Obviously the promotion does not change the up potential of any set that is disjoint from s, or any set at level higher than k.

Lemma B.9 *Promoting s from level k to $k+1$ does not increase the potential of set at levels at most k intersecting s.*

Proof: Fix any set s' at level $l(s') = i \leq k$ that intersects with s. Recall that promoting s from level k to $k+1$ decreases $w(s')$ and may possibly increase $\phi(s')$. If $\phi(s') > 0$ after the promotion then s' has 0 potential, and the lemma trivially holds.

Otherwise the promotion moves some elements in $E_k(s')$ to $E_{k+1}(s')$, and does not change the level of any other element contained in s'. Hence $w(s', j)$ remains unchanged for all $j \geq k+1$, and $w(s', j)$ decreases by the same amount, for all $j \leq k$. Consequently, $w(s', j) - w(s', j+1)$ remains unchanged for all $j \neq k$, and $w(s', k) - w(s', k+1)$ decreases. Moreover, $\hat{l}(s')$ does not increase. Hence the up potential of s' does not increase due to the promotion. ∎

B.3.4 Insertions and Deletions

Recall that Delete(e) handles each deletion in $O(f)$ time and Insert(e) handles each insertion in $O(f \log f)$ time. Additionally, $O(\frac{f}{\epsilon} \log C)$ time is charged to each insertion for maintaining the linked list of pointers to non-empty collections of tight sets.

Next, we bound the total up and down potential increased due to element insertions and deletion.

Deletion of e. In Algorithm 2, we increase $\phi(s)$ by at most $w(e)$ for each s containing e. Hence the total down potential increases by at most

$$\sum_{s: e \in s} w(e) \left(\frac{1}{\epsilon^3} + \frac{1}{\epsilon^2 f} \log C\right)(1+\epsilon)^{l(s)+1} \leq \sum_{s: e \in s} w(e) \left(\frac{1}{\epsilon^3} + \frac{1}{\epsilon^2 f} \log C\right)(1+\epsilon)^{l(e)+1} = O(\frac{f}{\epsilon^3} + \frac{1}{\epsilon^2} \log C).$$

Insertion of e. Recall that in Algorithm 4, we lift sets containing e up until all sets are good, and then insert element e. For all s containing e, we increase $w(s)$ by $w(e) \leq (1+\epsilon)^{-l(s)}$.

In the following, we show that e increases the up potential of s by $O(\frac{f}{\epsilon^3} + \frac{1}{\epsilon^2} \log C)$. Since each element is contained in at most f sets, the $O(\frac{f^2}{\epsilon^3} + \frac{f}{\epsilon^2} \log C)$ upper bound on the increase of total up potential follows. A similar upper bound on the increase of potential appeared in [BK19].

Lemma B.10 *An insertion of e increases the potential of each s containing e by $O(\frac{f}{\epsilon^3} + \frac{1}{\epsilon^2} \log C)$.*

29

Proof: Let $w(s)$ be the weight of s and $t = l(s)$, right before e is inserted. If $w(s) + w(e) \leq c_s$ then s has 0 up potential after the insertion, and the lemma trivially holds. Otherwise (since the algorithm lifts s up until it becomes good) we know that before the insertion $E_t(s) \neq \emptyset$. Moreover, by the bounded weight invariant, we have $w(s, t+1) < c_s$ before the insertion.

Recall that the conversion ratio of set weight to up potential at level i is given by $\alpha_i = \left(\frac{f}{\epsilon^3} + \frac{1}{\epsilon^2} \log C \right) (1 + \epsilon)^{i+1}$, and the up potential of s is given by

$$\sum_{i=l(s)}^{\hat{l}(s)-1} \left(w(s, i) - w(s, i+1) \right) \cdot \alpha_i + \left(w(s, \hat{l}(s)) - c_s \right) \cdot \alpha_{\hat{l}(s)}.$$

The insertion of e increases $w(s)$ by $w(e)$, and can possibly increase $\hat{l}(s)$. Let t' be the value of value of $\hat{l}(s)$ *after* the insertion. By definition of $w(s, i)$, the insertion does not decrease $w(s, i) - w(s, i+1)$ for any i. Since the insertion increases $w(s) - c_s$ by $w(e)$, the up potential of s is increased by at most $w(e) \cdot \alpha_{t'}$. (Recall that the conversion ratio is larger at higher level.)

Since $w(e) \leq (1 + \epsilon)^{-t}$, the increment in up potential of s is at most

$$(1 + \epsilon)^{-t} \cdot \left(\frac{f}{\epsilon^3} + \frac{1}{\epsilon^2} \log C \right) (1 + \epsilon)^{t'+1} = \left(\frac{f}{\epsilon^3} + \frac{1}{\epsilon^2} \log C \right) (1 + \epsilon)^{t'-t+1}.$$

It remains to show that $(1 + \epsilon)^{t'-t} = O(1)$.

Since $E_t(s) \neq \emptyset$ and $w(s, t+1) < c_s$ before the insertion, after the insertion, the weight of s at level t' is at most

$$c_s - |E_t(s)| \left((1 + \epsilon)^{-(t+1)} - (1 + \epsilon)^{-t'} \right) + (1 + \epsilon)^{-t'} \leq c_s - (1 + \epsilon)^{-(t+1)} + 2(1 + \epsilon)^{-t'}.$$

On the other hand, the weight of s at level t' after the insertion is at least c_s. Hence we have $c_s - (1 + \epsilon)^{-(t+1)} + 2(1 + \epsilon)^{-t'} \geq c_s$, which is equivalent to $(1 + \epsilon)^{t'-t} \leq 2(1 + \epsilon)$, as required. ∎

B.3.5 Total lift-up cost

Since $\mathsf{Rebuild}(k)$ removes all lift-up costs charged to the sets in $T_{\leq k}$, each set s at level i is charged at most $\max\{i - b(s) + 1, 0\}$ units of lift-up cost. Assume that the final instance contains no element. Then all sets are slack and are at level 0 in the final instance. Hence at the end of update sequence there is no lift-up cost charged to any set.

Next we show that the assumption is without loss of generality. Given any sequence of t updates, we append to the end of the sequence a deletion for each element that exists after the t updates. Hence there are in total at most $\Gamma + 2t$ updates, where Γ is the number of elements preprocessed. The total update time of the new sequence of updates, given that the final graph is empty, is at most $O((\Gamma + t) \cdot \lambda)$, where $\lambda = O(\frac{f^2}{\epsilon^3} + \frac{f}{\epsilon^2} \log C)$ is the amortized update time. Thus the total update time to process the t updates is $O((\Gamma + t) \cdot \lambda)$.

C   Our Algorithm for Worst Case Update Time

In this section we present the algorithm with a worst case $O(f \log^2(Cn)/\epsilon^3)$ update time. Specifically, we prove the following theorem.

Theorem C.1 *There are deterministic dynamic algorithms for the minimum set cover problem with $(1 + \epsilon)f$-approximation ratio and a worst-case update time of $O(f \log^2(Cn)/\epsilon^3)$.*

Our algorithm maintains the $(1 + \epsilon)f$-approximate set cover *value* after every update and can return it in constant time. The algorithm can also output the solution in time linear to the solution size whenever the solution is asked for (similar to the dynamic matching algorithm in [BFH19a]).

Notations. We use $[i, j]$ to denote $\{i, i + 1, \ldots, j\}$ for any integers $j \geq i \geq 0$. Our algorithm guarantees that every element e is at level $l(e) = \max_{s \in \mathcal{S}: e \in s}\{l(s)\}$. Similar to [BHN19], we classify the elements into three types, namely active, passive and dead. Each active element e has weight exactly $(1 + \epsilon)^{-l(e)}$; each passive element e has weight at most $(1 + \epsilon)^{-l(e)}$. An element becomes dead at the moment when it gets deleted. The weight of a dead element is decided by its weight before the deletion. As it will be clear from the description of the algorithm, each newly inserted element will be passive, which may turn into active only if it is involved in a rebuild of levels.

Worst Case Update Time. In standard worst case update time algorithms, we assume that a new update arrives when the previous one is properly handled. The worst case update time is then measured by the maximum time between two consecutive updates. In this paper we assume that updates and queries arrive in a fixed rate. That is, the time between any two consecutive updates is fixed, say, by some parameter λ. We show that our algorithm updates the data structure or answers the query before the next update/query arrives. Our algorithm maintains L parallel data structures, and does $O(L \cdot \lambda)$ units of work between two consecutive updates/queries. Consequently we have a worst case update time of $O(L \cdot \lambda)$. Our goal is the design algorithms that work with updates with a high arrival rate, i.e., small λ.

We show that the main technical challenge is to do the rebuild efficiently while committing newly arrived updates to the data structure (hierarchy) that is being rebuild. Specifically, to guarantee that the number of passive elements and dead elements are bounded, we need to assume that a bounded number of updates arrive when the rebuild is being executed in the background.

We first present in Section C.3 a rebuild algorithm that works with $\lambda = c \cdot \frac{1}{\epsilon} \cdot f \cdot L^2$, for some sufficiently large constant c, which implies an $O(\frac{f}{\epsilon^4} \log^3(Cm))$ worst case update time. We improve the algorithm and obtain an $O(\frac{f}{\epsilon^3} \log^2(Cm))$ worst case update time in Section C.4.

Data Structure. As before (in the amortized algorithm), for every set we maintain a partition of the elements it contains according to the levels of elements. Additionally, every element maintains a pointer to every set containing it.

C.1 Algorithm Framework

We first describe a data structure that consists of L independent hierarchies as follows.

For each $k \in [1, L]$, we maintain a hierarchy H_k that has levels $[0, k]$, which we refer to as the *local view* of level k (local view(k)). We use $A_k(i)$ to denote the collection of active elements at level i, from the local view(k), and $A_k(\leq i)$ to denote the collection of active elements on or below level i, from local view(k). We define the sets $P_k(i)$, $P_k(\leq i)$ for passive elements, and $D_k(i)$, $D_k(\leq i)$ for dead elements similarly. Let $E_k(i) = A_k(i) \cup P_k(i) \cup D_k(i)$, and $E_k(\leq i) = \bigcup_{j \leq i} E_k(j)$. Let $S_k(i)$ (resp. $S_k(\leq i)$) denote the collection of sets at level i (resp. at most i), from local view(k). We also maintain a counter for the size of each collection (of elements or sets) we maintain. Unless otherwise specified, we use a doubly linked list for each of these collections. Thus given the pointer to an element or a set, the insertion and deletion of the element or set can be done in $O(1)$ time. Moreover, we can also merge two collections in $O(1)$ time.

Let $l_k(e)$ be the level of element e, and $l_k(s)$ be the level of set s, in local view(k).

Additionally, every set in local view(k) has an *extra weight*, which represents the total weight the set receives from levels $[k+1, L]$ (which are not included in H_k). Let $\delta_k(s)$ denote the extra weight of s in local view(k). The weight of set s in local view(k) is then defined as

$$w_k(s) = \delta_k(s) + \sum_{e \in s \cap E_k(\leq k)} w_k(e),$$

where $w_k(e)$ denotes the weight of element e in local view(k).

We call a set s *tight* (with respect to local view(k)) if $w_k(s) > \frac{c_s}{1+\epsilon}$. We maintain the invariant that each set at level $[1, k]$ of H_k is tight.

Invariant C.1 (Local Tightness) *For all $i \in [1, k]$, each $s \in S_k(i)$ has weight $w_k(s) > \frac{c_s}{1+\epsilon}$.*

Every update (that inserts or deletes an element) becomes L updates, one for each hierarchy.

Handling An Update in Local View(k). Suppose an element e is inserted or deleted. We first check if all sets containing e exist in $S_k(\leq k)$. If no, then we ignore the update. Otherwise
- if the element is deleted, we convert e into a dead element in H_k;
- if the element is inserted, we insert e as a passive element at level $l_k(e) = \max_{s:e \in s}\{l_k(s)\}$. If $l_k(e) > 0$ then we set $w_k(e) = 0$; otherwise $l(e) = 0$, and we set $w_k(e)$ to be $\min_{s:e \in s}\{c_s - w_k(s)\}$, which is at most 1. Note that by Invariant C.1 at least one set containing e is tight.

We also update the relevant counters for local view(k) accordingly.

We maintain the following local invariant regarding the total number of passive elements and dead elements in each local view.

Invariant C.2 (Local Element) *For every $k \in [L]$, we have $|P_k(\leq k) \cup D_k(\leq k)| < 2\epsilon \cdot |A_k(\leq k)|$.*

The local element invariant (Invariant C.2) guarantees that the number of dead elements is bounded, which is crucial to bound the approximation ratio of the set cover.

Additionally, we define the *rebuild-triggering event* for local view(k) as $|P_k(\leq k) \cup D_k(\leq k)| \geq \epsilon \cdot |A_k(\leq k)|$. Whenever the event is triggered, Scheduler(k) gets into action and the hierarchy H_k will be completely rebuilt. We show (in Section C.1.3) that when Scheduler(k) is running (in the background), Invariant C.2 remains valid. The triggering event defines a stronger version of Invariant C.2. It is crucial that we rebuild the data structure earlier (before Invariant C.2 is violated), since it might take a long time before the rebuild finishes, and we need to guarantee that the invariant is maintained before the rebuild finishes.

Scheduler(k). We first describe the scheduler for level k, which rebuilds H_k in the background, in a high level. The rebuild (which we refer to as Rebuild(k)) takes the elements in $E_k(\leq k)$ as input and outputs a new hierarchy H_k^*, which has levels $[0, k+1]$. We show that we are able to commit the newly arrived updates to the hierarchy H_k^* while it is being constructed. Hence when Rebuild(k) finishes, the output hierarchy H_k^* is up-to-date. Scheduler(k) then replaces H_i for all $i \leq k$ based on H_k^*. In addition, Scheduler(k) updates local view($k+1$) based on H_k^*.

C.1.1 Consistency Invariants

Initially, in the preprocessing we construct a hierarchy for the input graph, based on which we construct L consistent hierarchies such that $E_k(i)$ is the same for all $k \geq i$. In general (when there are updates), the local views can be inconsistent. For example, when levels on or below k are rebuilt

32

in Scheduler(k), local view($k-1$) will be completely replaced, while local view($k+2$) is oblivious to the rebuild. In this case $A_{k+2}(i)$ might be different from $A_{k-1}(i)$ for some $i \leq k$.

Nevertheless, our algorithm maintains the following invariants that guarantee some consistency between the parallel hierarchies. We show that the invariants are maintained in Section C.1.3.

Invariant C.3 (Lower View Consistency) *We have $S_{i+1}(\leq i) = S_i(\leq i)$ for all $i \in [1, L-1]$.*

Invariant C.4 (Upper View Consistency) *Suppose an element e is passive (resp. dead) w.r.t. local view(i). Then e is also passive (resp. dead) w.r.t. local view(j) for all $j > i$.*

By invariant C.3, if a set exists in local view(k) then it also exists in local view($k+1$). Hence every update not ignored by local view(k) will not be ignored by local view(i) if $i > k$. Consequently, every passive/dead element in H_k also appears in H_i, as stated in Invariant C.4.

Next we present the details of Scheduler(k).

C.1.2 Rebuild and Synchronization

Recall that Rebuild(k) takes the elements in $E_k(\leq k)$ as input and outputs a new hierarchy H_k^*. We will present two algorithms for the construction in Section C.3 and Section C.4, respectively.

Property C.1 *We show that Rebuild(k) constructs a hierarchy H_k^* with the following properties.*

(a) *The hierarchy H_k^* has levels $[0, k+1]$, and contains every set in $S_k(\leq k)$. If an element e in $A_k(\leq k) \cup P_k(\leq k)$ is not deleted by an update that arrives when Scheduler(k) is running, then it will be contained in H_k^*. Moreover, if $e \in A_k(\leq k)$ then it is active in H_k^*; if $e \in P_k(\leq k)$ then it either becomes active, or will be passive at level $k+1$ of H_k^*.*

(b) *Each set s in H_k^* has extra weight $\delta_k^*(s) = \delta_k(s)$. Each set at level $[1, k+1]$ of H_k^* is tight.*

(c) *When Rebuild(k) finishes, the hierarchy H_k^* is up-to-date, i.e., all updates that have arrived are committed to the hierarchy.*

(d) *For each $i \leq k$, $|P_k^*(\leq i) \cup D_k^*(\leq i)| < \epsilon \cdot |A_k^*(\leq i)|$, where $P_k^*(\leq i)$, $D_k^*(\leq i)$, $A_k^*(\leq i)$ denote the collection of passive, dead, and active elements, respectively, at levels $[0, i]$ of H_k^*.*

(e) *The construction of H_k^* takes time at most $\frac{\epsilon}{2L} \cdot |E_k(\leq k)| \cdot \lambda$.*

We run $k+1$ copies of Rebuild(k) in Scheduler(k), and construct $k+1$ identical hierarchies H_k^*. We name the hierarchies as $H_{k\to 1}^*, H_{k\to 2}^*, \ldots, H_{k\to k+1}^*$. As the name indicates, the hierarchy $H_{k\to i}^*$ is constructed for the purpose of replacing/updating local view(i) when the rebuild finishes.

For each $i \leq k$, we take the hierarchy $H_{k\to i}^*$, remove levels $[i+1, k+1]$ and update the extra weights as follows. The extra weight $\delta_{k\to i}^*(s)$ of each set s in $H_{k\to i}^*$ is defined to be its extra weight $\delta_k(s)$ in local view(k) plus the total weight s receives from (elements in) levels $[i+1, k+1]$ in $H_{k\to i}^*$. Then we replace the old hierarchy H_i with the new hierarchy $H_{k\to i}^*$, and free the space taken by the old hierarchy. If Scheduler(i) is currently running in the background, we terminate it immediately (in $O(1)$ time).

We also update local view($k+1$) as follows. Take the hierarchy $H_{k\to k+1}^*$, and include all sets and elements at level $k+1$ of H_{k+1} into level $k+1$ of $H_{k\to k+1}^*$. By storing the elements in $A_{k+1}(k+1)$ (resp. $P_{k+1}(k+1)$, $D_{k+1}(k+1)$ and $S_{k+1}(k+1)$) as a linked list, the merging of data structure can be done in $O(1)$ time. The extra weight $\delta_{k\to k+1}^*(s)$ of every set s in $H_{k\to k+1}^*$ is set to be $\delta_{k+1}(s)$.

33

Let the resulting hierarchy be the new local view of $k + 1$. Consequently, in local view$(k + 1)$, the extra weights of sets do not change after the update.

If Scheduler$(k+1)$ is currently running in the background, we do not terminate it. Additionally, if Scheduler$(k + 1)$ is currently reading/duplicating elements[5] in the old hierarchy H_{k+1}, then we do not free the space taken by the old hierarchy H_{k+1}. Instead, we keep committing newly arrived updates to the data structure until the duplicating finishes. As soon as Scheduler$(k + 1)$ finishes reading the old hierarchy H_{k+1}, we free the space taken by the old hierarchy.

Data Structure for the Hierarchies. We assume that each local view(i), where $i \in [1, L]$, has a pointer that points to the hierarchy H_i. The hierarchy H_i maintains a pointer for each collection of the elements and sets at each level. Hence when we need to replace the H_i in local view(i), it suffices to change the pointer from the old hierarchy to the new hierarchy in $O(1)$ time.

C.1.3 Maintenance of Invariants

In this section we show that Invariants C.1, C.2, C.3 and C.4 are maintained.

We first show that Invariants C.3 and C.4 are maintained.

Lemma C.1 *The lower view consistency invariant (Invariant C.3) is maintained.*

Proof: Initially when all local views are consistent, the invariant trivially holds. Note that sets change their levels only when some hierarchy is rebuilt. In other words, insertions and deletions of elements do not change levels of sets. Consider the point in time when some Scheduler(k) finishes, where $k \in [1, L]$.

Recall that Scheduler(k) replaces local view(i) for every $i \leq k$, and update local view$(k + 1)$ based on the identical hierarchies output by Rebuild(k). Hence $S_i(\leq i)$ and $S_i(\leq i - 1)$ does not change for every $i \in [k + 2, L]$. Additionally, $S_{k+1}(\leq k + 1)$ does not change. Thus Invariant C.3 holds for every $i \in [k + 1, L - 1]$. Since after Scheduler(k) finishes, $S_i(j) = S_{k+1}(j)$ for every $i \in [1, k]$ and $j \in [0, k]$, Invariant C.3 is also maintained for every $i \in [1, k]$. ∎

Lemma C.2 *The upper view consistency invariant (Invariant C.4) is maintained.*

Proof: Suppose e is passive w.r.t. local view(i) but not passive w.r.t. local view(j) for some $j > i$. Note that when e is inserted to H_i, it is also inserted to H_k for every $k > i$ (by Invariant C.3), as passive elements. The element e stays passive in H_j until either (1) Scheduler(k) converts e into active, or moves it to level $k + 1$, for some $k \geq j$; or (2) Scheduler$(j - 1)$ converts e into active. In both cases local view(i) will be replaced, and e will no longer be passive in H_i, which is a contradiction. The proof for dead elements is similar. ∎

Next we show that the local tightness invariant (Invariant C.1) is maintained.

Lemma C.3 *The local tightness invariant (Invariant C.1) is maintained.*

Proof: We show that after Scheduler(k) replaces local view(i) for all $i \leq k$ and updates local view$(k + 1)$, all sets at levels $[1, i]$ of local view(i) is tight, for all $i \leq k + 1$.

For local view(i), where $i \leq k$, the invariant follows straightforwardly from the fact that every set at level $[1, k + 1]$ of H_k^* is tight, and the way we update the extra weight (in Section C.1.2). Next we show that every set at level $[1, k + 1]$ from the new local view$(k + 1)$ is tight.

---

[5]As we will show later, the first step of Rebuild$(k + 1)$ is making a copy for every set and element in H_{k+1}.

Consider the point in time right before we update local view$(k + 1)$. Fix any set s at level $[1, k + 1]$ of H_{k+1}. If $s \in S_{k+1}(k + 1)$, then the weight $w_{k+1}(s)$ does not change when Scheduler(k) updates local view$(k + 1)$. If $s \in S_{k+1}(\le k)$ then s also appears in $S_k(\le k)$ (by Invariant C.3) and $H^*_{k \to k+1}$. When Scheduler(k) updates local view$(k + 1)$, $w^*_{k \to k+1}(s)$ is updated as follows.

- It increases by $\sum_{e \in s \cap E_{k+1}(k+1)} w_{k+1}(e)$ when we include elements in $E_{k+1}(k + 1)$ to $H^*_{k \to k+1}$.

- Then it decreases by $\delta_k(s) - \delta_{k+1}(s)$ when we update the extra weight of s in $H^*_{k \to k+1}$.

We show that $\delta_k(s) - \delta_{k+1}(s) = \sum_{e \in s \cap E_{k+1}(k+1)} w_{k+1}(e)$, which implies that modifying $H^*_{k \to k+1}$ does not change $w^*_{k \to k+1}(s)$. Hence s is tight in the new local view$(k + 1)$ if $l^*_{k \to k+1}(s) \ge 1$.

Consider the last point in time when $\delta_k(s)$ (the extra weight of s in local view(k)) is updated[6], which must be the time when Scheduler(j) finishes, for some $j \ge k$. In this case the extra weights are updated (in Section C.1.2) such that $\delta_k(s) - \delta_{k+1}(s) = \sum_{e \in s \cap E_{k+1}(k+1)} w_{k+1}(e)$. Since then, the LHS of the equality does not change. Furthermore, the RHS of the equality does not change either: if an element is inserted to $P_{k+1}(k + 1)$, then it has weight 0; if an element is deleted and moved to $D_{k+1}(k + 1)$, then its contribution to $w_{k+1}(s)$ does not change. Hence we have $\delta_k(s) - \delta_{k+1}(s) = \sum_{e \in s \cap E_{k+1}(k+1)} w_{k+1}(e)$ right before we update local view$(k + 1)$, and s is tight if $l^*_{k \to k+1}(s) \ge 1$ after local view$(k + 1)$ is updated. ∎

Finally, we show that the local element invariant (Invariant C.2) is maintained.

Lemma C.4 *The local element invariant (Invariant C.2) is maintained.*

Proof: By Property C.1(d), when Scheduler(k) replaces the local view(i) for every $i \le k$, Invariant C.2 is maintained. Indeed, right after local view(i) is replaced, we can guarantee that the rebuild-triggering event is not triggered, for all $i \le k$.

Recall that we also update local view$(k + 1)$. However, by Invariant C.3 and C.4, before local view$(k + 1)$ is updated, every passive/dead element contained in $H^*_{k \to k+1}$ is also contained in $P_{k+1}(\le k) \cup D_{k+1}(\le k)$, since every update committed to local view(k) and Scheduler(k) is also committed to local view$(k + 1)$. Hence updating local view$(k + 1)$ does not increase $|P_{k+1}(\le k + 1) \cup D_{k+1}(\le k + 1)|$. Moreover, updating local view$(k + 1)$ does not decrease $|A_{k+1}(\le k + 1)|$, because every active element in $A_{k+1}(\le k)$ is also contained in $H^*_{k \to k+1}$. Hence Invariant C.2 will not be violated when Scheduler(k) updates local view$(k + 1)$.

Finally, we show that Invariant C.2 is maintained in local view(k) before Scheduler(k) finishes. By Property C.1(d), Scheduler(k) takes at most $\frac{\epsilon}{2} \cdot |E_k(\le k)| \cdot \lambda$ time to construct the identical hierarchies, during which $x \le \frac{\epsilon}{2} \cdot |E_k(\le k)|$ updates arrive.

Let a be the size of $P_k(\le k) \cup D_k(\le k)$, and b be the size of $A_k(\le k)$ when Scheduler(k) starts. By the definition of the rebuild triggering event, we have $a = \epsilon \cdot b$. Hence we have $x \le \frac{\epsilon}{2} \cdot (a + b) = \frac{\epsilon + \epsilon^2}{2} \cdot b$. Since each update decreases $|A_k(\le k)|$ by at most one, before Scheduler(k) finishes we have $|A_k(\le k)| \ge b - x \ge (1 - \frac{\epsilon + \epsilon^2}{2}) \cdot b$. Since each update increases $|P_k(\le k) \cup D_k(\le k)|$ by at most one, before Scheduler(k) finishes we have

$$|P_k(\le k) \cup D_k(\le k)| \le a + x \le (\epsilon + \frac{\epsilon + \epsilon^2}{2}) \cdot b \le \frac{\epsilon + \frac{\epsilon + \epsilon^2}{2}}{1 - \frac{\epsilon + \epsilon^2}{2}} \cdot |A_k(\le k)| < 2\epsilon \cdot |A_k(\le k)|.$$

Note that while Scheduler(k) is running in the background, we might update local view(k) (when Scheduler$(k - 1)$ finishes). However, as argued above, updating local view(k) does not increase $|P_k(\le k) \cup D_k(\le k)|$ nor decrease $|A_k(\le k)|$. Hence the above upper bound still holds. ∎

---

[6]Note that if $\delta_{k+1}(s)$ is updated then $\delta_k(s)$ will also be updated.

C.1.4 Answering a Query

We answer the query on the size of set cover as follows. All slack sets in $S_1(0)$, i.e., sets s at level 0 of H_1 with $w_1(s) \leq \frac{c_s}{1+\epsilon}$, are not in the set cover. All other sets are in the set cover. The correctness follows from the approximation ratio analysis in the next section.

C.2 Consistent Hierarchy and Approximation Ratio

We first show that while the local views are inconsistent, by Invariant C.3, there is a natural way of partitioning the sets into $L+1$ levels, which induces a consistent hierarchy.

Lemma C.5 *The collections $\{S_k(k)\}_{k \in [L]}$ form a partition of all sets into $L+1$ levels.*

Proof: We prove by induction on k that $\{S_i(i)\}_{i \in [k]}$ is a partition of the sets $S_k(\leq k)$.

The statement is trivially true for $k = 1$. For the case $k+1$, we have $S_{k+1}(k+1) \cap S_{k+1}(\leq k) = \emptyset$. By Invariant C.3, we have $S_k(\leq k) = S_{k+1}(\leq k)$. By induction hypothesis $\{S_i(i)\}_{i \in [k]}$ is a partition of the sets in $S_{k+1}(\leq k)$. Hence $\{S_i(i)\}_{i \in [k+1]}$ is a partition of the sets in $S_{k+1}(\leq k+1)$. ∎

We show that at any point in time, there exists a hierarchy (with levels $[0, L]$) containing all elements and sets, which we refer to as the *consistent hierarchy*, such that every set at level $[1, L]$ is tight, and the collection of sets at level 0 is $S_1(0)$. Moreover, set $s \in S(0) = S_1(0)$ is tight in the consistent hierarchy if and only if it is tight w.r.t. local view(1). Hence the collection of tight sets is a feasible set cover (which implies the correctness for answering a query in Section C.1.4)

We construct the consistent hierarchy as follows. For all $i \in [1, L]$, let the sets and elements at level i of the hierarchy be defined as follows:

$$S(i) = S_i(i), \quad A(i) = A_i(i), \quad P(i) = P_i(i) \quad \text{and} \quad D(i) = D_i(i).$$

Let $S(0) = S_1(0)$, $A(0) = A_1(0)$, $P(0) = P_1(0)$ and $D(0) = D_1(0)$. Let $E(i) = A(i) \cup P(i) \cup D(i)$.

We first show that the hierarchy is well defined, i.e., every set and element (that is not deleted) appears exactly once in the hierarchy. By Lemma C.5, every set appears exactly once in the consistent hierarchy. Let $l(s)$ (resp. $l(e)$) be the level of set s (resp. element e) in the consistent hierarchy. Note that for each $l(s) = i$, we have $l_i(s) = i$. Consider any element e that exists, e.g., inserted and not deleted. Let $\max_{s:e \in s}\{l(s)\} = k$. Hence e exists in local view(k) since all sets containing in e appear in local view(k). Moreover, we have $e \in E_k(k) = E(k)$, which implies that $l(e) = \max_{s:e \in s}\{l(s)\}$. Moreover, (1) $e \notin E_i(i)$ for all $i > k$ since each element in $E_i(i)$ contains a set in $S_i(i)$, while the maximum level of sets containing e is k; (2) since for all $i < k$, there exists some set containing e that is not in $S_i(\leq i)$, e does not appear in any of $\{E_i(i)\}_{i<k}$. Hence e appears exactly once (at level k) in the consistent hierarchy.

Since there is no extra weight attached to each set, the real weight $w(s)$ of a set s in the consistent hierarchy is defined as the total weight of the elements it contains. In the following, we call a set tight if it is tight w.r.t. the consistent hierarchy.

Lemma C.6 (Global Tightness) *Every set at level $[1, L]$ in the consistent hierarchy is tight. Set s at level 0 in the consistent hierarchy is tight if and only if it is tight w.r.t. local view(1).*

Proof: Consider any $s \in S(i)$, where $i \geq 1$, in the consistent hierarchy. By the local tightness invariant (Invariant C.1), from local view(i), the weight s receives from $E_i(i)$ plus $\delta_i(s)$ is more than $\frac{c_s}{1+\epsilon}$. Since we define $E(i) = E_i(i)$, it suffices to show that the extra weight $\delta_i(s)$ equals to the weight s receives from levels $[i+1, L]$ in the consistent hierarchy.

To prove that, we show that for every $k \in [i, L-1]$ (note that $\delta_L(s) = 0$),

$$\delta_k(s) - \delta_{k+1}(s) = \sum_{e \in s \cap E_{k+1}(k+1)} w(e).$$

Consider the last point in time when $\delta_k(s)$ is changed, which must be the time when Scheduler(j) finishes, for some $j \geq k$. When $\delta_k(s)$ changes, our algorithm guarantees that $\delta_k(s) - \delta_{k+1}(s) = \sum_{e \in s \cap E_{k+1}(k+1)} w_{k+1}(e)$. Since then no Rebuild($j$) finishes, for all $j \geq k$, and both the LHS and RHS of the equality does not change, which concludes the proof.

Using the same argument, for every set $s \in S(0) = S_1(0)$, the weight s receives from levels $[i+1, L]$ in the consistent hierarchy is $\delta_1(s)$. Since $E(0) = E_1(0)$, i.e., every element in level 0 of local view(1) is preserved in the consistent hierarchy, set $s \in S(0)$ is tight in the consistent hierarchy if and only if it is tight w.r.t. local view(1). ∎

Lemma C.6 implies that every element e in the consistent hierarchy contains at least one tight set: if $l(e) \geq 1$ then it contains a set at level ≥ 1, which is tight; if $l(e) = 0$ then it contains a set $s \in S_1(0)$ with weight c_s, which is tight. Hence the collection of tight sets of the consistent hierarchy is a feasible set cover. Next we analyze the approximation ratio.

We show that in the consistent hierarchy, the number of dead elements is bounded. We prove the following stronger statement, which upper bounds the total number of passive/dead elements.

Lemma C.7 *In the consistent hierarchy, for all $i \in [L]$ we have*

$$|P(\leq i) \cup D(\leq i)| \leq 2\epsilon \cdot |A(\leq i)|.$$

Proof: By the local element invariant (Invariant C.2), we have $|P_i(\leq i) \cup D_i(\leq i)| \leq 2\epsilon \cdot |A_i(\leq i)|$. In the following we show that (1) $P(\leq i) \cup D(\leq i) \subseteq P_i(\leq i) \cup D_i(\leq i)$; (2) $A_i(\leq i) \subseteq A(\leq i)$.

We first prove (1). Note that every $e \in P(\leq i) \cup D(\leq i)$ must appear as a passive or dead element in $P_j(j) \cup D_j(j)$ for some $j \leq i$. By Invariant C.4, e is also contained in $P_i(\leq i) \cup D_i(\leq i)$. We remark that $P(\leq i) \cup D(\leq i)$ can be a proper subset of $P_i(\leq i) \cup D_i(\leq i)$. For example, when some dead element in $D_i(\leq i)$ is cleaned up by Scheduler($i-2$), it is still contained in $P_i(\leq i) \cup D_i(\leq i)$, but not in $P(\leq i) \cup D(\leq i)$.

Statement (2) follows immediately from the fact that every active element appears exactly once (as an active element) in the consistent hierarchy. Observe that if $e \in A_i(\leq i)$, then all sets containing e appear in $S(\leq i)$ in the consistent hierarchy, which implies $e \in A(\leq i)$. We remark that $A_i(\leq i)$ can be a proper subset of $A(\leq i)$. For example, when some passive element in $P_{i-2}(\leq i-2)$ is converted to active by Scheduler($i-2$), it will be contained in $A(\leq i)$, but not in $A_i(\leq i)$. ∎

Following the proof of [BHN19, Lemma 4.8], Lemma C.7 implies that the total weight of dead elements is at most $O(\epsilon)$ times the total weight of active elements. This implies the approximation ratio.

C.3 A Simple Rebuild Algorithm

In this section we present a simple rebuild algorithm for constructing the hierarchy H_k^* in Scheduler(k) that works with $\lambda = c \cdot \frac{1}{\epsilon} \cdot f \cdot L^2$, for some sufficiently large constant c.

Recall the static rebuild algorithm from [BHN19] as follows. Roughly speaking, the algorithm puts all elements to level $k+1$ (line 1-7 in Algorithm 7), and then gradually moves the elements to lower levels until every set becomes tight or reaches level 0 (line 8-17 in Algorithm 7). Note that since the rebuild is done in the background, in the first phase (putting elements to level $k+1$), we

37

need to make a copy for every active/passive element in $E_k(\leq k)$. We also make a copy for each set in $S_k(\leq k)$. Note that each set will be copied only once. Moreover, when it is copied, its weight and extra weight are also copied.

In the following, we use $w_k^*(e)$ and $l_k^*(e)$ to denote the weight and level of element e in H_k^*. The other notations, e.g., $w_k^*(s)$, $\delta_k^*(s)$, $l_k^*(s)$ and $A_k^*(i)$, are defined similarly. We call a set s tight if $w_k^*(s) > \frac{c_s}{1+\epsilon}$. We call an element *tight* if it is contained in at least one tight set; *slack* otherwise.

Whenever we change the weight of an element we also update the weight of sets containing it. Specifically, if we change $w_k^*(e)$ from $(1+\epsilon)^{-i}$ to $(1+\epsilon)^{-j}$, then we increase $w_k^*(s)$ by $(1+\epsilon)^{-j} - (1+\epsilon)^{-i}$ for each s containing e. For ease of presentation, we do not state it explicitly in the pseudocode.

Note that every element that is moved to level $k+1$ either has weight $(1+\epsilon)^{-(k+1)}$ (which becomes active), or is tight (and stays passive at level $k+1$). In Round-i, where $i = k+1, k, \ldots, 1$, we identify the sets that become tight, and move all slack sets and elements to level $i-1$. Since each slack set has weight at most $\frac{c_s}{1+\epsilon}$ and slack elements increase their weights by a factor of $(1+\epsilon)$ when moved one level down, we can guarantee that each set has weight at most c_s, and all sets at levels $[1, k+1]$ are tight. Additionally, there is no dead element, and passive elements only appear at level $k+1$.

Algorithm 7 Rebuild(k)

1: Initialize $E_k^* \leftarrow \emptyset$ and $S_k^* \leftarrow \emptyset$.
2: **for** each $e \in A_k(\leq k)$ **do**
3:      $E_k^* \leftarrow E_k^* \cup \{e\}$, $S_k^* \leftarrow S_k^* \cup e$, $w_k^*(e) \leftarrow (1+\epsilon)^{-(k+1)}$.
4: **for** each $e \in P_k(\leq k)$ **do**
5:      $E_k^* \leftarrow E_k^* \cup \{e\}$, $S_k^* \leftarrow S_k^* \cup e$, $w_k^*(e) \leftarrow \min\{(1+\epsilon)^{-(k+1)}, \min_{s:e\in s}\{c_s - w_k^*(s)\}\}$.
6: **for** each $e \in E_k^*$ and $s \in S_k^*$ **do**
7:      $l_k^*(e) \leftarrow \perp$, $l_k^*(s) \leftarrow \perp$.   ▷ their levels are undecided
8: **for** i from $k+1$ to 1 (Round-i) **do**
9:      **for** each $s \in S_k^*$ **do**   ▷ settle the tight sets at level i
10:          **if** $w_k^*(s) > \frac{c_s}{1+\epsilon}$ **then**
11:              $l_k^*(s) \leftarrow i$, $S_k^* \leftarrow S_k^* \setminus \{s\}$.
12:              **for** each $e \in s \cap E_k^*$ **do**
13:                  $l_k^*(e) \leftarrow i$, $E_k^* \leftarrow E_k^* \setminus \{e\}$.
14:      **for** each $e \in E_k^*$ **do**   ▷ move slack elements one level down
15:          $w_k^*(e) \leftarrow (1+\epsilon)^{-(i-1)}$, and update $w_k^*(s)$ accordingly for all s containing e.
16: **for** each $s \in S_k^*$ **do**
17:      $l_k^*(s) \leftarrow 0$.

While H_k^* are being rebuilt by Scheduler(k) in the background, each update that arrives will be handled in both local view(k) and Scheduler(k). Recall that we have already described the way an update is handled in local view(k). Next we describe how to handle an update in Scheduler(k).

Handling An Update in Rebuild(k). Consider any update that arrives in Round-i, for $i \in [k+1]$ (for ease of argument, we consider line 1-7 part of Round-$(k+1)$).

Suppose element e is deleted:

- If $l_k^*(e) = \perp$, i.e., we have not decided the final level of e, then e will simply be removed, which also decreases the weight of the sets containing e.

- If e has already settled its level (at some $j \in [i, k+1]$), then we convert it into a dead element (with the same weight) at level j.

Suppose element e is inserted:

- If $l_k^*(s) = \perp$ for all s containing e, then we set the weight of e as

$$w_k^*(e) \leftarrow \min\left\{(1+\epsilon)^{-i}, \min_{s:e\in s}\{c_s - w_k^*(s)\}\right\}.$$

  That is, imagine that we increase $w_k^*(e)$ gradually until either (1) some set s containing e has weight c_s, or (2) $w_k^*(e) = (1+\epsilon)^{-i}$. In case (1), e will be passive and at least one set containing e becomes tight in Round-i. Hence we set $l_k^*(e) \leftarrow i$ when we identify the tight sets containing e. In case (2), e is active, and its level remains undecided.

- If there exists $l_k^*(s) \neq \perp$ for some s containing e, then we set $l_k^*(e) \leftarrow \max_{s:e\in s \wedge l_k^*(s)\neq\perp}\{l_k^*(s)\}$ and e is passive. If $l_k^*(e) \geq 1$ then we set $w_k^*(e) \leftarrow 0$. Otherwise $l_k^*(e) = 0$, which means that e arrives in Round-0 and all sets containing e are slack. Then we set $w_k^*(e) \leftarrow \min_{s:e\in s}\{c_s - w_k^*(s)\}$, and identify the sets containing e that become tight and mark them as tight.

Note that an update arriving when Scheduler(k) is running does not necessarily introduce a passive or dead element. The element e that is inserted or deleted may become active, or be removed immediately. However, if e arrives in Round-i and becomes passive or dead, then $l_k^*(e) \geq i$.

Properties of H_k^*. It is easy to check that Property C.1(a) (b) and (c) are satisfied by the construction. Moreover, all elements in $D_k^*(\leq k+1)$ and $P_k^*(\leq k)$ are from newly arrived deletions and insertions. Next, we show that Property C.1(d) is satisfied, i.e., $|P_k^*(\leq i) \cup D_k^*(\leq i)| < \epsilon \cdot |A_k^*(\leq i)|$. We remark that the statement also holds for $k+1$ but we do not need it.

Lemma C.8 *When Rebuild(k) finishes, we have $|P_k^*(\leq i) \cup D_k^*(\leq i)| < \epsilon \cdot |A_k^*(\leq i)|$ for all $i \leq k$.*

Proof: Let $E_k^*(i) = A_k^*(i) \cup P_k^*(i) \cup D_k^*(i)$ be the elements at level i of H_k^*. We first show that for every $i \in [k]$, it takes $(i+1) \cdot (|S_k^*(\leq i)| + f \cdot |E_k^*(\leq i)|) = O(f \cdot L \cdot |E_k^*(\leq i)|)$ time to construct the levels $[0, i]$ (recall that every element is contained in at most f sets). To construct each level i, we first scan through the collection of slack sets and elements (to identify the tight ones), and then move every slack element one level down (and update the set weights). Since each slack set/element that is scanned in Round-i eventually settles at level at most i, it takes $O(|S_k^*(\leq i)| + f \cdot |E_k^*(\leq i)|) = O(f \cdot |E_k^*(\leq i)|)$ time to construct level i. Since there are $i+1$ levels on or below i, the claim follows.

Recall that Scheduler(k) runs $k+1 = O(L)$ copies of Rebuild(k) simultaneously. Hence it takes $O(f \cdot L^2 \cdot |E_k^*(\leq i)|)$ total time to construct the levels on or below i.

Observe that every $e \in P_k^*(\leq i) \cup D_k^*(\leq i)$ comes from an update that arrives after Round-$(i+1)$ finishes: suppose otherwise, i.e., e arrives in Round-j for some $j \geq i+1$, then we have $l_k^*(e) \geq i+1$, which is a contradiction.

Claim C.1 *At most $\frac{\epsilon}{3} \cdot |E_k^*(\leq i)|$ updates arrive while Scheduler(k) is constructing levels $[0, i]$.*

Proof: Since it takes $O(f \cdot L^2 \cdot |E_k^*(\leq i)|)$ time to construct levels on or below i and updates arrive every $c \cdot \frac{1}{\epsilon} f \cdot L^2$ time (for some sufficiently large constant c), we can assume that at most $\frac{\epsilon}{3} \cdot |E_k^*(\leq i)|$ updates arrive when levels $[0, i]$ are being constructed. ∎

Since each update increases $|P_k^*(\leq i) \cup D_k^*(\leq i)|$ by at most one. We have

$$|P_k^*(\leq i) \cup D_k^*(\leq i)| \leq \frac{\epsilon}{3} \cdot |E_k^*(\leq i)| = \frac{\epsilon}{3} \cdot |A_k^*(\leq i) \cup P_k^*(\leq i) \cup D_k^*(\leq i)|.$$

Reordering the inequality, we have $|P_k^*(\le i) \cup D_k^*(\le i)| \le \frac{\epsilon}{3-\epsilon} \cdot |A_k^*(\le i)| < \epsilon \cdot |A_k^*(\le i)|$. ■

Note that if x updates arrive while Scheduler(k) is running in the background, then we have $|E_k^*(\le k)| \le |E_k(\le k)| + x$, where $|E_k(\le k)|$ denote the number of elements in H_k when Scheduler(k) starts. Hence Claim C.1 implies that at most

$$\frac{\epsilon}{3} \cdot |E_k^*(\le i)| \le \frac{\epsilon}{3(1 - \frac{\epsilon}{3})} \cdot |E_k(\le k)| \le \frac{\epsilon}{2} \cdot |E_k(\le k)|$$

updates arrive while Scheduler(k) is running in the background, which implies Property C.1(e).

Applying the rebuild algorithm to the general framework, we obtain an algorithm with worst case $O(\frac{f}{\epsilon^4} \log^3(Cm))$ update time.

C.4 Improving the Worst Case Update Time

In this section we present an improved rebuild algorithm that works with $\lambda = c \cdot \frac{1}{\epsilon} \cdot f \cdot L$, for some sufficiently large constant c, which implies a worst case update time of $O(\frac{f}{\epsilon^3} \log^2(Cm))$. Specifically, we present an efficient Rebuild(k) subroutine that runs in $O(f \cdot |E_k(\le k)| + k)$ time. We develop our algorithm based on the Fix-Levels(k) algorithm from [BHN19]. Note that we can assume without loss of generality that $O(f \cdot |E_k(\le k)| + k) = O(f \cdot |E_k(\le k)|)$, since otherwise we can finish the rebuild in $O(L)$ time, before the next update arrives. The main challenge is to commit the updates to Scheduler(k) when the new hierarchy is being rebuilt, while guaranteeing Property C.1(d).

C.4.1 Static Efficient Rebuild(k)

Recall the rebuild algorithm from [BHN19] as follows (refer to Algorithm 8). The first phase of the new algorithm is the same as Algorithm 7: we make a copy of every active and passive element in H_k, and put them at level $k + 1$ (with appropriate weight). Then we identify the tight sets and elements, which stay at level $k + 1$. However, instead of moving the slack elements down level by level (in which case an element might get scanned $O(k)$ times), we scan the elements in a specific order such that every element that gets scanned will have its level decided immediately. By doing so, the update time spent on each element is $O(f)$, which implies an $O(f \cdot |E_k(\le k)|)$ total rebuild time.

Specifically, suppose E_k^* contains the slack elements (each of which has weight $(1+\epsilon)^{-(k+1)}$), we define the target level $l_T(s)$ of every slack set s as follows. If $s \cap E_k^* = \emptyset$ then $l_T(s) = 0$; otherwise $l_T(s)$ is the maximum i such that

$$w_k^*(s) + \left((1 + \epsilon)^{-i} - (1 + \epsilon)^{-(k+1)}\right) \cdot |s \cap E_k^*| > \frac{c_s}{1 + \epsilon}.$$

Note that we have $l_T(s) \le x$ if and only if

$$w_k^*(s) + \left((1 + \epsilon)^{-(x+1)} - (1 + \epsilon)^{-(k+1)}\right) \cdot |s \cap E_k^*| \le \frac{c_s}{1 + \epsilon}.$$

When Round-k begins, we have $l_T(s) \le k$ since each slack set s has weight $\le \frac{c_s}{1+\epsilon}$.

Roughly speaking, $l_T(s)$ is the highest level i at which s is tight if we move s and the elements it contains to level i. Note that the target level of s may change when the collection of slack elements changes. It is shown in [BHN19] that when the number of slack elements decreases, the target level of each slack set does not increase. This is sufficient as [BHN19] shows only an amortized update time guarantee. However, to guarantee a worst case update time we need to commit the updates

40

to the hierarchy that is being rebuild. In this case the above property is not guaranteed. Instead, we show in our algorithm that all slack sets in Round-i have target level at most i.

Algorithm 8 EfficientRebuild(k)

1: Copy elements in $A_k(\leq k) \cup P_k(\leq k)$ to E_k^*, and sets in $S_k(\leq k)$ to S_k^*.
2: Initialize their weights as in line 2-5 of Algorithm 7, and set their levels to $\perp$ (undecided).
3: Identify the tight sets and elements, assign them level $k+1$ and exclude them from S_k^* and E_k^*.
4: **for** i from k to 0 (Round-i) **do** $\triangleright$ settle the level of sets with target level i
5:   **while** exists $s \in S_k^*$ with $l_T(s) = i$ **do**
6:     $l_k^*(s) \leftarrow i$, $S_k^* \leftarrow S_k^* \setminus \{s\}$.
7:     **for** each $e \in s \cap E_k^*$ **do**
8:       $w_k^*(e) \leftarrow (1+\epsilon)^{-i}$, $l_k^*(e) \leftarrow i$.
9:       $E_k^* \leftarrow E_k^* \setminus \{e\}$, and update $l_T(s')$ for all s' containing e.

We maintain an array $\Gamma[0, \ldots, k]$ such that $\Gamma[i]$ points to a linked list of sets with target level i. Hence in Round-i, we can identify a set with target level i (line 5 of Algorithm 8) in $O(1)$ time.

C.4.2 Handling a Newly Arrived Update

Next we describe the algorithm that handles an update arriving in Round-i. For convenience we refer to line 1-3 of Algorithm 8 as Round-$(k+1)$. Suppose element e is deleted:

- If $l_k^*(e) = \perp$, i.e., we have not decided the final level of element e, then e will simply be removed. Note that for each s containing e, the removal decreases the weight $w_k^*(s)$ of s and possibly the target level $l_T(s)$.

- If e has already settled its level (at some $j \in [i, k+1]$), then we convert it into a dead element (with the same weight) at level j.

Suppose element e is inserted:

- If $l_k^*(s) = \perp$ for all s containing e, then we first try to include e to E_k^* with weight $(1+\epsilon)^{-(k+1)}$. Note that e increases $w_k^*(s)$ by $(1+\epsilon)^{-(k+1)}$ and $|s \cap E_k^*|$ by one, for every s containing e. If after inserting e to E_k^* we have $l_T(s) \leq i$ for all s containing e then we include e in E_k^* as stated above, and mark e active. Recall that for each set s we have $l_T(s) \leq i$ if

$$w_k^*(s) + \left((1+\epsilon)^{-(i+1)} - (1+\epsilon)^{-(k+1)}\right) \cdot |s \cap E_k^*| < \frac{c_s}{1+\epsilon}.$$

Otherwise we insert e as a passive element with appropriate weight such that after the insertion all sets containing e have target level at most i, and at least one of them has target level i. In this case, e is not included in E^*, and will be passive at level i.

Specifically, we maintain the invariant that all sets in S_k^* have target level at most i in Round-i. If there exists s containing e with $l_T(s) = i$ before the insertion then we insert e as a passive element at level i with $w_k^*(e) \leftarrow 0$. Otherwise we set $w_k^*(e)$ to be slightly larger than[7]

$$\min_{s:e \in s} \left\{ \frac{c_s}{1+\epsilon} - w_k^*(s) - \left((1+\epsilon)^{-i} - (1+\epsilon)^{-(k+1)}\right) \cdot |s \cap E_k^*| \right\},$$

---

[7]Here "slightly larger than" means larger than but arbitrarily close to

so that the maximum target level of sets containing e becomes i. Set e as a passive element at level i. Note that we can guarantee $w_k^*(e) > 0$ since $w_k^*(s) - \left((1+\epsilon)^{-i} - (1+\epsilon)^{-(k+1)}\right) \cdot |s \cap E_k^*| \le \frac{c_s}{1+\epsilon}$ for all s containing e before the insertion. We can guarantee $w_k^*(e) \le (1+\epsilon)^{-i}$ since otherwise for all s containing e:

$$\left(w_k^*(s) + (1+\epsilon)^{-(k+1)}\right) + \left((1+\epsilon)^{-(i+1)} - (1+\epsilon)^{-(k+1)}\right) \cdot (|s \cap E_k^*| + 1)$$

$$< w_k^*(s) + \left((1+\epsilon)^{-i} - (1+\epsilon)^{-(k+1)}\right) \cdot |s \cap E_k^*| + (1+\epsilon)^{-i} < \frac{c_s}{1+\epsilon},$$

which implies that e should have been included in E_k^* and become active.

- If there exists $l_k^*(s) \ne \perp$ for some s containing e, then we set $l_k^*(e) \leftarrow \max_{s:e \in s \wedge l_k^*(s) \ne \perp} \{l_k^*(s)\}$ and e is passive. If $l_k^*(e) \ge 1$ then we set $w_k^*(e) \leftarrow 0$. Otherwise $l_k^*(e) = 0$, and we set $w_k^*(e) \leftarrow \min_{s:e \in s}\{c_s - w_k^*(s)\} \le 1$, and identify the sets containing e that become tight and mark them as tight.

As before, if e arrives in Round-i and becomes passive or dead, then we have $l_k^*(e) \ge i$.

Correctness. We show that every set (and element) will be assigned a level in $[k+1]$ when Rebuild(k) finishes. Specifically, we show that in Round-i, all slack sets have level at most i. Since we identify the sets with target level i in Round-i and settle their levels, the following lemma implies that all slack sets have target level 0 in Round-0, and will have their level decided after Round-0.

Lemma C.9 *In Round-i, every set in S_k^* has target level at most i.*

Proof: We prove that for every $i \in [0, k]$, if when Round-i begins all sets in S_k^* have target level at most i, then during Round-i all sets in S_k^* have target level at most i. Note that Round-i ends only if there is no set with target level i, we can guarantee that all sets in S_k^* have target level at most $i-1$ when Round-i ends. As we have shown, when Round-k begins we have $l_T(s) \le k$ for all $s \in S_k^*$. Hence the above statement implies the lemma.

Suppose that every set in S_k^* has target level at most i when Round-i begins. We show that there does not exist any set with target level larger than i during Round-i.

Consider any set $s \in S_k^*$ in Round-i. The target level $l_T(s)$ of s changes either (1) when some element in s is moved to level i in Round-i, or (2) when an element in s is deleted and removed in Round-i; or (3) when a new element is inserted and contained in s in Round-i.

- In case (1), $w_k^*(s)$ increases by $(1+\epsilon)^{-i} - (1+\epsilon)^{-(k+1)}$, and $|s \cap E_k^*|$ decreases by one. By definition of target level, $l_T(s)$ does not increase, given that originally $l_T(s) \le i$.

- In case (2), both $w_k^*(s)$ and $|s \cap E_k^*|$ decrease, and thus $l_T(s)$ does not increase.

- In case (3), suppose e is inserted. If e is inserted as an active element in E_k^* then our algorithm guarantees that $l_T(s) \le i$ for all s containing e after the insertion. If e is inserted as a passive element with weight 0 then $l_T(s)$ does not change. Otherwise e is inserted as a passive element with weight larger than (but arbitrarily close to)

$$\min_{s':e \in s'} \left\{ \frac{c_{s'}}{1+\epsilon} - w_k^*(s') - \left((1+\epsilon)^{-i} - (1+\epsilon)^{-(k+1)}\right) \cdot |s' \cap E_k^*| \right\}.$$

Consider any s' (including s) containing e after the insertion. If $|s' \cap E_k^*| = 0$, when by definition we have $l_T(s') = 0 \le i$ (before and after the insertion of e). Otherwise $|s' \cap E_k^*| \ge 1$

and we have

$$w_k^*(e) + w_k^*(s') + ((1+\epsilon)^{-(i+1)} - (1+\epsilon)^{-(k+1)}) \cdot |s' \cap E_k^*|$$
$$\leq w_k^*(e) + w_k^*(s') + ((1+\epsilon)^{-i} - (1+\epsilon)^{-(k+1)}) \cdot |s' \cap E_k^*| - ((1+\epsilon)^{-i} - (1+\epsilon)^{-(i+1)}) < \frac{c_{s'}}{1+\epsilon},$$

which implies $l_T(s') \leq i$.

Hence set s have target level at most i during Round-i, which implies the lemma. ∎

C.4.3   Properties of the Hierarchy

It is easy to check that Property C.1(a) and (c) are satisfied by the construction. Next we show that Property C.1(b) is satisfied. Recall that we set $\delta_k^*(s) = \delta_k(s)$ for all s in H_k^*.

Lemma C.10 *When EfficientRebuild(k) finishes, all sets at levels $[1, k+1]$ are tight.*

Proof: Note that each set s settles its level at i if and only if it has target level i. By definition of target level, when we set $l_k^*(s) = i > 0$, the weight of s is more than $\frac{c_s}{1+\epsilon}$. Moreover, by definition of target level, if we set $l_k^*(s) = i + 1$ then $w_k^*(s) \leq \frac{c_s}{1+\epsilon}$. Hence we have $w_k^*(s) \in (\frac{c_s}{1+\epsilon}, 1]$ when we set $l_k^*(s) = i$. Since the weight of s does not change afterwards, s remains tight when EfficientRebuild(k) finishes. ∎

Next we show that Property C.1 (e) is satisfied. By Lemma C.9, every element will be scanned exactly once in EfficientRebuild(k). Moreover, the algorithm spends $O(f)$ time to settle an element and update the data structure. Hence immediately we have the following.

Claim C.2 *EfficientRebuild(k) spends $O(f \cdot |E_k^*(\leq i)|)$ time to construct levels $[0, i]$.*

Specifically, construction of H_k^* takes total time $O(f \cdot |E_k^*(\leq k+1)|) = O(f \cdot |E_k(\leq k)|) \leq \frac{\epsilon}{2L} \cdot |E_k(\leq k)| \cdot \lambda$, if we assume $\lambda = c \cdot \frac{1}{\epsilon} \cdot f \cdot L$, for some sufficiently large constant c.

Finally, we show that Property C.1(d) is satisfied. Recall that Scheduler(k) runs L parallel copies of EfficientRebuild(k).

Lemma C.11 *When EfficientRebuild(k) finishes, for all $i \leq k$, $|P_k^*(\leq i) \cup D_k^*(\leq i)| < \epsilon \cdot |A_k^*(\leq i)|$.*

Proof: We first show that if an update that arrives in Round-j creates a passive or dead element e, then we have $l_k^*(e) \geq j$. Claim C.2 implies that Scheduler(k) takes $O(f \cdot L \cdot |E_k^*(\leq i)|)$ time to construct levels $[0, i]$. Since updates arrive every $\lambda = O(\frac{f}{\epsilon} \cdot L)$ time, the lemma follows immediately.

Consider any update that arrives in Round-j and inserts or deletes element e.

If e is deleted, then either e will be removed immediately, or it will be assign to the level of some set s containing e whose level is already decided. In the later case we have $l_k^*(e) \geq j$. If e is inserted, then either (1) e becomes active; or (2) e is assign to the level of some set s containing e whose level is already decided, which has level at least j; or (3) e is given an appropriate weight such that at least one set s containing e has target level i. In the last case e will be assigned level j in Round-j, which completes the case analysis. ∎

References

[AAG+19]   Amir Abboud, Raghavendra Addanki, Fabrizio Grandoni, Debmalya Panigrahi, and Barna Saha. Dynamic set cover: improved algorithms and lower bounds. In *STOC*, 2019.

[ACC+18]   Moab Arar, Shiri Chechik, Sarel Cohen, Cliff Stein, and David Wajc. Dynamic matching: Reducing integral algorithms to approximately-maximal fractional algorithms. In *ICALP*, volume 107 of *LIPIcs*, pages 7:1–7:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[BCH17]   Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika Henzinger. Deterministic fully dynamic approximate vertex cover and fractional matching in O(1) amortized update time. In *IPCO*, 2017.

[BFH19a]   Aaron Bernstein, Sebastian Forster, and Monika Henzinger. A deamortization approach for dynamic spanner and dynamic maximal matching. In *SODA*, pages 1899–1918. SIAM, 2019.

[BFH19b]   Aaron Bernstein, Sebastian Forster, and Monika Henzinger. A deamortization approach for dynamic spanner and dynamic maximal matching. In *SODA*, pages 1899–1918. SIAM, 2019.

[BGS11]   Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in $o(\log n)$ update time. In *FOCS*, 2011.

[BHI15a]   Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Design of dynamic algorithms via primal-dual method. In *ICALP*, 2015.

[BHI15b]   Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. In *SODA*, 2015.

[BHN16]   Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. New deterministic approximation algorithms for fully dynamic matching. In *STOC*, pages 398–411. ACM, 2016.

[BHN17]   Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. Fully dynamic approximate maximum matching and minimum vertex cover in $O(\log^3 n)$ worst case update time. In *SODA*, pages 470–489. SIAM, 2017.

[BHN19]   Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. A new deterministic algorithm for dynamic set cover. In *FOCS*, 2019.

[BK19]   Sayan Bhattacharya and Janardhan Kulkarni. Deterministically maintaining a $(2+\epsilon)$-approximate minimum vertex cover in $o(1/\epsilon^2)$ amortized update time. In *SODA*, 2019.

[CPR11]   Timothy M. Chan, Mihai Patrascu, and Liam Roditty. Dynamic connectivity: Connecting to networks and geometry. *SIAM J. Comput.*, 40(2):333–349, 2011. Announced at FOCS'08.

[CS18a]   Moses Charikar and Shay Solomon. Fully dynamic almost-maximal matching: Breaking the polynomial worst-case time barrier. In *ICALP*, volume 107 of *LIPIcs*, pages 33:1–33:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[CS18b]   Moses Charikar and Shay Solomon. Fully dynamic almost-maximal matching: Breaking the polynomial worst-case time barrier. In *ICALP*, volume 107 of *LIPIcs*, pages 33:1–33:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[DGKR05] Irit Dinur, Venkatesan Guruswami, Subhash Khot, and Oded Regev. A new multilayered PCP and the hardness of hypergraph vertex cover. *SIAM J. Comput.*, 34(5):1129–1146, 2005.

[DS14]   Irit Dinur and David Steurer. Analytical approach to parallel repetition. In *STOC*, 2014.

[GKKP17] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Online and dynamic algorithms for set cover. In *STOC*, 2017.

[GP13]   Manoj Gupta and Richard Peng. Fully dynamic $(1+\epsilon)$-approximate matchings. In *FOCS*, 2013.

[KR03]   Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within 2-\varepsilon. In *CCC*, 2003.

[Lar12]   Kasper Green Larsen. The cell probe complexity of dynamic range counting. In *STOC*, pages 85–94. ACM, 2012.

[LWY18]   Kasper Green Larsen, Omri Weinstein, and Huacheng Yu. Crossing the logarithmic barrier for dynamic boolean data structure lower bounds. In *STOC*, pages 978–989. ACM, 2018.

[NS13]   Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. In *STOC*, 2013.

[NS17]   Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and o(n$^{1/2 - \epsilon}$)-time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1122–1129, 2017.

[NSW17]   Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *FOCS*, pages 950–961. IEEE Computer Society, 2017.

[OR10]   Krzysztof Onak and Ronitt Rubinfeld. Maintaining a large matching and a small vertex cover. In *STOC*, 2010.

[PD06]   Mihai Patrascu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006.

[PS16]   David Peleg and Shay Solomon. Dynamic $(1 + \epsilon)$-approximate matchings: A density-sensitive approach. In *SODA*, 2016.

[Sol16]   Shay Solomon. Fully dynamic maximal matching in constant update time. In *FOCS*, 2016.

[Wul17]   Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1130–1143, 2017.