Fully-Dynamic Coresets

Monika Henzinger [ID]
University of Vienna, Faculty of Computer Science, Austria
https://homepage.univie.ac.at/monika.henzinger/
monika.henzinger@univie.ac.at

Sagar Kale
University of Vienna, Faculty of Computer Science, Austria
https://sagark4.github.io/
sagar.kale@univie.ac.at

──── **Abstract** ────

With input sizes becoming massive, coresets – small yet representative summary of the input – are relevant more than ever. A weighted set C_w that is a subset of the input is an ε-coreset if the cost of any feasible solution S with respect to C_w is within $[1\pm\varepsilon]$ of the cost of S with respect to the original input. We give a very general technique to compute coresets in the fully-dynamic setting where input points can be added or deleted. Given a static (i.e., not dynamic) ε-coreset-construction algorithm that runs in time $t(n, \varepsilon, \lambda)$ and computes a coreset of size $s(n, \varepsilon, \lambda)$, where n is the number of input points and $1-\lambda$ is the success probability, we give a fully-dynamic algorithm that computes an ε-coreset with worst-case update time $O((\log n) \cdot t(s(n, \varepsilon/\log n, \lambda/n), \varepsilon/\log n, \lambda/n))$ (this bound is stated informally), where the success probability is $1-\lambda$. Our technique is a fully-dynamic analog of the merge-and-reduce technique, which is due to Har-Peled and Mazumdar [17] and is based on a technique of Bentley and Saxe [3], that applies to the insertion-only setting where points can only be added. Although, our space usage is $O(n)$, our technique works in the presence of an adaptive adversary, and we show that $\Omega(n)$ space is required when adversary is adaptive.

As a concrete implication of our technique, using the result of Braverman et al. [4], we get fully-dynamic ε-coreset-construction algorithms for k-median and k-means with worst-case update time $O(\varepsilon^{-2}k^2 \log^5 n \log^3 k)$ and coreset size $O(\varepsilon^{-2}k \log n \log^2 k)$ ignoring $\log\log n$ and $\log(1/\varepsilon)$ factors and assuming that $\varepsilon = \Omega(1/\operatorname{poly}(n))$ and $\lambda = \Omega(1/\operatorname{poly}(n))$ (which are very weak assumptions made only to make these bounds easy to parse). This results in the first fully-dynamic constant-approximation algorithms for k-median and k-means with update times $O(\operatorname{poly}(k, \log n, \varepsilon^{-1}))$. Specifically, the dependence on k is only quadratic, and the bounds are worst-case. The best previous bound for both problems was *amortized* $O(n \log n)$ by Cohen-Addad et al. [10] via randomized $O(1)$-coresets in $O(n)$ space.

We also show that under the OMv conjecture [18], a fully-dynamic $(4 - \delta)$-approximation algorithm for k-means must either have an amortized update time of $\Omega(k^{1-\gamma})$ or amortized query time of $\Omega(k^{2-\gamma})$, where $\gamma > 0$ is a constant.

1 Introduction

Clustering is an ubiquitous notion that one encounters in computer-science areas such as data mining, machine learning, image analysis, bioinformatics, data compression, and computer graphics, and also in the fields of medicine, social science, marketing, etc. Today, when the input data has become massive, one would rather run an algorithm on a small but representative summary of the input, and for clustering problems, a *coreset* serves that function perfectly. The concept of a coreset was defined first in computational geometry as a small subset of a point set that approximates the shape of the point set. The word coreset has now evolved to mean an appropriately weighted subset of the input that approximates the original input with respect to solving a computational problem.

Let P be a problem for which the input is a weighted subset[1] $X_w \subseteq U$; think of U as in a metric space (U, d), so U is unweighted and d is the distance function. Let $n := |X_w|$ and $W := \sum_{x \in X_w} w(x)$. We also refer to elements of U as points. The goal in the problem P then is to output S^* that belongs to the feasible-solution space (or *query* space) Q such that the cost $c(S^*, X_w)$ is minimized. For example, in the *k-median* (respectively, *k-means*) problem, Q is the set of all (unweighted) subsets of X_w of cardinality at most k and $c(S, X_w) := \sum_{x \in X_w} w(x) \min_{s \in S} d(x, s)$ (respectively, $\sum_{x \in X_w} w(x) \min_{s \in S} (d(x, s))^2$). Then, for the problem P, a weighted set C_w such that $C_w \subseteq X_w$ is an ε-*coreset* if, for any feasible solution $S \in Q$, we have that $c(S, X_w) \in [1 \pm \varepsilon] c(S, C_w)$; we sometimes say that the *quality* of coreset C_w is ε. For many problems, fast coreset-construction algorithms exist; e.g., for k-median and k-means, $\tilde{O}(nk)$-time[2] algorithms for computing ε-coresets of size $O(\varepsilon^{-2} k \operatorname{polylog}(n))$ exist.

Throughout the paper, we assume that the cost function c for the problem P is *linear*: for any weighted subsets $Y_w^1, Y_w^2 \subseteq U$ with disjoint supports and any $S \in Q$, we have that $c(S, Y_w^1 \cup Y_w^2) = c(S, Y_w^1) + c(S, Y_w^2)$, where the union $Y_w^1 \cup Y_w^2$ is the weighted union. It is easy to see that k-median and k-means cost functions are linear.

Our goal in this paper is to give *dynamic algorithms* for computing a coreset. In the dynamic setting, the input changes over time. A dynamic algorithm is a data structure that supports three types of operations: *Insert*(p, w), which inserts a point p with weight w into X_w; *Delete*(p), which removes point p from X_w; and *Query*$()$, which outputs a coreset of X_w. Weight updates can be simulated by deleting and re-inserting a given point, or the data structure may support a separate weight-changing operation. This is known as the *fully* dynamic model as opposed to the *insertion-only* setting where a point can only be inserted. At any time instant, a coreset is maintained by the algorithm, and the complexity measure of interest is the *update* time, i.e., how fast the solution can be updated after receiving a point update, and also the *size* of the coreset, which determines the *query* time. Suppose there is a dynamic coreset-construction algorithm, say ALG_D, for a problem P. Then a solution for the problem P can be maintained dynamically by running ALG_D, and on query, a solution is computed by querying ALG_D and running a static (i.e., not dynamic) algorithm for P on the returned coreset. In this paper, we give a very general technique on how to maintain a coreset in the fully-dynamic setting: given a *static* coreset-construction algorithm for any problem P, we show how to turn it into a dynamic coreset-construction algorithm for P.

---

[1] When using a set operation such as union or notation such as $\subseteq$ with one or more weighted sets, we mean it for the underlying unweighted sets. Also, all weights are nonnegative.

[2] Logarithmic factors are hidden in the $\tilde{O}$ notation.

Intuitively, our technique is to the fully-dynamic setting as the merge-and-reduce technique is to the insertion-only setting. The *merge-and-reduce* technique, which is based on a technique of Bentley and Saxe [3], is due to Har-Peled and Mazumdar [17] and is a fundamental technique to obtain an insertion-only coreset-construction algorithm using a static coreset-construction algorithm, say ALG_S, as a black box. Loosely speaking, it is as follows. At any time instant, the algorithm maintains up to $\lceil \log n \rceil$ buckets. For $i \in \{1, 2, \ldots, \lceil \log n \rceil\}$, the bucket B_i has capacity 2^{i-1}, each bucket can be either *full*, (i.e. at capacity 2^{i-1}) or *empty*, and each point goes in exactly one bucket. Then at any time-instant, the current number of points uniquely determines the states of the buckets. Whenever a point is inserted, the states of the buckets change like a binary counter. That is, the new point goes into B_i, where B_i is the smallest-index empty bucket, and all the points in $\cup_{j=1}^{i-1} B_j$ are moved to B_i (*merge*). Note that this creates a full bucket B_i. Then a coreset is computed on B_i by running ALG_S on it (*reduce*). The overall coreset is then just union of all non-empty buckets.

We show that a similar result can be achieved in the fully-dynamic setting. Our main result is the following theorem (stated slightly informally).

▶ **Theorem 1.** *Assume that there is a static coreset construction algorithm for a problem P with linear cost function that **a)** runs in time $t_P(n_s, \varepsilon_s, \lambda_s, W_s)$, **b)** always outputs a set of cardinality at most $s_P(\varepsilon_s, \lambda_s, W_s)$ and total weight at most $(1+\delta)W_s$, and **c)** has the guarantee that the output is an ε_s-coreset with probability at least $1-\lambda_s$, where n_s is the number of* integer-*weighted input points and W_s is the total weight of points.*

Then there is a fully-dynamic coreset-construction algorithm for P that, with rational-*weighted input points, **a)** always maintains an output set of cardinality at most $s_P(\varepsilon, \lambda, W)$, **b)** has the guarantee that the output is an ε-coreset with probability at least $1-\lambda$, and **c)** has worst-case update time*

$$O\left((\log n) \cdot t_P \left(s_P^*, \frac{\varepsilon}{\log n}, \frac{\lambda}{n}, W \right) \right) ,$$

where $W = O((1+\delta)^{\lceil \log n \rceil} \operatorname{poly}(n))$, $s_P^* = s_P \left(\frac{\varepsilon}{\log n}, \frac{\lambda}{n}, W \right)$, and n is the current number of points.

We mention below a concrete implication of the above theorem for k-median and k-means using the result of Braverman et al. [4].

▶ **Theorem 2.** *For the k-median and k-means problems, there is a fully-dynamic algorithm that maintains a set of cardinality $O(\varepsilon^{-2}k(\log n \log k \log(k\varepsilon^{-1} \log n) + \log(1/\lambda)))$, that is an ε-coreset with probability at least $1-\lambda$, and has worst-case update time*

$$O\left(\varepsilon^{-2}k^2 \log^5 n \log^3 k \log^2(\varepsilon^{-1})(\log \log n)^3 \right) ,$$

assuming that $\varepsilon = \Omega(1/\operatorname{poly}(n))$ and $\lambda = \Omega(1/\operatorname{poly}(n))$. [3]

Ignoring $\log \log n$ and $\log(1/\varepsilon)$ above, the coreset cardinality is $O(\varepsilon^{-2}k \log n \log^2 k)$ and worst-case update time is $O(\varepsilon^{-2}k^2 \log^5 n \log^3 k)$. It can be easily proved that running an α-approximation algorithm for k-median on an ε-coreset gives a $2\alpha(1+\varepsilon)$-approximation whereas that for k-means gives a $4\alpha(1+\varepsilon)$-approximation. Any such polynomial-time static algorithm – say, e.g., $(5 + \varepsilon')$-approximation algorithm for k-median by Arya et al. [2] and

---

[3] We make these very weak assumptions to simplify some extremely unhandy factors involving ε and λ in the expression for the update time.

16-approximation algorithm for k-means by Gupta and Tangwongsan [15] – can be run on our output coreset in $O(\text{poly}(k, \log n, \varepsilon^{-1}))$ time to obtain a constant approximation. This is the first fully-dynamic constant-approximation algorithm for k-median and k-means whose *worst-case* time per operation is polynomial in k, $\log n$, and ε^{-1}. The best previous result was a randomized algorithm with *amortized* $O(n \log n)$ update time and $O(n)$ space by Cohen-Addad et al. [10].

With a simple reduction, we also show a conditional lower bound on the time per operation for k-means. The following theorem is proved as Theorem 20 in Section 4.

▶ **Theorem 3.** *Let $\gamma > 0$ be a constant. Under the OMv conjecture [18], for any $\delta > 0$, there does not exist a fully-dynamic algorithm that maintains a $(4 - \delta)$-approximation for k-means with amortized update time $O(k^{1-\gamma})$ and query time $O(k^{2-\gamma})$ such that over a polynomial number of updates, the error probability is at most $1/3$.*

Our technique

At the core, our technique is simple. We always maintain a balanced binary tree of depth $\lceil \log n \rceil$ containing exactly n leaf nodes (recall that n is the current number of points). Each node corresponds to a subset of X_w, the current input: each leaf node corresponds to a singleton (hence n leaf nodes), and an internal node corresponds to the weighted union of the sets represented by its children. If the cardinality of the union exceeds a certain threshold, then we use the static coreset-construction algorithm to compute its coreset. The root gives a coreset of the whole input.

We next explain how we handle updates in this data structure. Point insertions are straightforward: create a new leaf node and run all the static-algorithm instances at the nodes on the leaf-to-root path. The way we handle point deletions is similar in spirit to the way delete-min works in a min-heap data structure: whenever a point at leaf-node ℓ_d is deleted, we swap contents of ℓ_d with those of the rightmost leaf-node, say ℓ_r, and delete ℓ_r, thus maintaining the balance of the tree. Then we run all the static-algorithm instances at the nodes on the two affected leaf-to-root paths.

However, there are some complications that require new techniques to make it work in *worst-case* time. To maintain guarantees for the output coreset quality and overall success probability, we need to adapt the parameters ε_s and λ_s used for the static algorithm at the internal nodes. The problem is that both depend on n, which changes over time and thus might become outdated. To show an *amortized* update-time bound, we can simply rerun the static algorithms at all internal nodes whenever n has changed by a constant factor. To achieve our worst-case bound, we use two *refresh pointers* that point at leaf nodes, and after each update operation, we rerun using the new values of ε_s and λ_s all the static-algorithm instances at the nodes on the leaf-to-root path from the leaf nodes pointed to by the refresh pointers. This keeps the outputs of the static-algorithm instances at the internal nodes always fresh. After every update, we move these pointers to the right so that they point to the next leaf nodes.

Further complications are caused by fractional weights at the leaf nodes and fractional intermediate-output weights. A problem arises when the weights in X_w are fractional, and the static algorithm expects integer-weighted input [9]. Even if the static algorithm can handle fractional weights [12, 4], there can be a problem. The output of the static algorithm at an internal node is the input for the static algorithm at its parent. Naïvely feeding these output fractional weights directly to the static algorithm at the parent may result in numbers exponential in n near the root, thus prohibitively increasing the update time. To deal with

these problems, rounding is needed for the input, i.e., at the leaf nodes, as well as for each intermediate-output at an internal node. Thus, we propose a more sophisticated rounding scheme and show that the rounding errors accumulated by our rounding are not too high.

We note that our balanced binary-tree data structure may be used to get dynamic algorithms in the following situations. Let $f : \mathbb{R}^{\dim} \to \mathbb{R}^{\dim}$ be a *multi-valued* function. Suppose for any u and v with disjoint supports and for any $f_u \in f(u)$ and $f_v \in f(v)$, we have $f_u + f_v \in f(u + v)$. Also suppose that $f(f(v)) \subseteq f(v)$ for any v. Now, given input v, we want to compute some vector in $f(v)$. If there is a static algorithm for this, then using our technique, we can maintain some vector in $f(v)$ for a dynamically changing vector v. The allowed dynamic operation on v is "add a to the ith coordinate of v," where $a \in \mathbb{R}$. The resulting dynamic algorithm is fast if the static algorithm always outputs a "small" vector; this is true for coresets because coresets are small by nature. Thinking about coresets in the above language, each point is an identity vector in $\mathbb{R}_+^{|U|}$, and then each weighted set of points naturally identifies with a vector. An ε-coreset reduces the number of points drastically. Union of coresets of two disjoint sets is a coreset of the union of those two sets (see Lemma 5). Although an ε-coreset of an ε-coreset is not an ε-coreset, it is a $(2\varepsilon + \varepsilon^2)$-coreset (see Lemma 6).

Space

In the merge-and-reduce technique, a bucket B_i will not actually contain 2^{i-1} points but just a coreset of 2^{i-1} points that would have been there otherwise at any time instant. Thus, using space just $\lceil \log n \rceil$ times the coreset size for a bucket, one can get a coreset of the whole input [17]. This makes it also applicable in the more restricted streaming model, where the input points arrive in a sequence and the goal is to compute a coreset using sublinear space. In the fully-dynamic setting, deletions also need to be handled, and hence no deterministic or randomized algorithm against an adaptive adversary that stores only a coreset is possible: the adversary generating the input could simply ask a query and then delete all points in the returned coreset. Hence, an algorithm that does not store any information about the non-coreset points would not be able to maintain a valid coreset. Even though we store all the points in our fully-dynamic technique, i.e., its space usage is $O(n)$, it works against an adaptive adversary because we never make any assumption about the next update and perform each update independently of all previous updates. By a straightforward reduction from the communication problem of INDEX, we show that $\Omega(n)$ space is required in the presence of an adaptive adversary. The proof of the following theorem appears in Section 4.

▶ **Theorem 4.** *A fully-dynamic algorithm that obtains any bounded approximation for 1-median or 1-means that works in the presence of an adaptive adversary and has success probability $1 - 1/(8n^2)$ must use $\Omega(n)$ space, where n is the current number of points.*

Comparison with the sparsification technique

Our technique is close to the sparsification technique of Eppstein et al. [11] that is used to speed up dynamic graph algorithms. There, one has to assume that the number of vertices in the input graph, say n_v, does not change, but the edge set changes dynamically, and the bounds are obtained in terms of n_v and m, the current number of edges. Their dynamic edge-tree structure is based on a fixed vertex-partition tree. In the vertex-partition tree, a node at level i corresponds to a vertex-set of cardinality $n_v/2^i$, and a vertex-set at a node is a union of its children's vertex sets (cf. our technique). To start using the edge tree, the vertex-partition tree has to be built first and hence the knowledge of n_v is necessary. Neither

do we need such a fixed structure nor any information about the number of points. Also, in the sparsification technique, there is no analog of weight handling/rounding. Another crucial difference is that they do not use a routine analogous to our refresh-pointers routine because their internal-node guarantees are always fresh. As we discussed before, these refresh pointers are critical for us also in making sure that the error introduced by the unavoidable rounding of output weights of the static-algorithm instances is kept in check.

1.1 Related Work

The most related work is by Cohen-Addad et al. [10] who give an $O(1)$-coreset for k-median and k-means in *amortized* update time of $O(n \log n)$.

For k-median and k-means, the first coreset-construction algorithms were by Har-Peled and Mazumdar [17] for Euclidean metrics and by Chen [9] for general metrics. Improved algorithms computing smaller coresets were later obtained by Har-Peled and Kushal [16] and by Feldman and Langberg [12]. The current known best is by Braverman et al. [4]: $O(\varepsilon^{-2} k \log k \log n)$-size coresets in $\tilde{O}(nk)$ time, who also give an excellent summary of the literature on coresets that we highly recommend. Note that by merge-and-reduce technique, each improvement also gave rise to better (insertion-only) streaming coreset constructions. For k-median and k-means, Frahling and Sohler [14] gave the first coreset-construction algorithm in the dynamic-streaming setting where points can be added or removed. It uses space and update time of $O(\text{poly}(\varepsilon^{-1}, \log m, \log \Delta))$ for constant k and dim when the points lie in the discrete Euclidean metric space $\{1, \ldots, \Delta\}^{\dim}$; for k-median, this was recently improved to $O(\varepsilon^{-2} k \, \text{poly}(\dim, \log \Delta))$ space and update time of $O(\text{poly}(\varepsilon^{-1}, k, \dim, \log \Delta))$ by Braverman et al. [5]. Coreset constructions with improvements in certain parameters in the Euclidean settings have been obtained [13, 24].

The k-median and k-means problems have received significant attention in the algorithms community [8, 20, 19, 7, 2, 23, 21, 15, 22, 1, 6]. The best approximation ratio for k-median is 2.675 by Byrka et al. [6] and that for k-means is $9 + \varepsilon$ by Ahmadian et al. [1].

2 Preliminaries

Let us fix a problem P with the input X_w, the set of feasible solutions Q, and the linear cost function $c : Q \times \mathcal{W} \to \mathbb{R}_+$, where $\mathcal{W}$ is the set of all weighted subsets[4] of X_w. All the numbers encountered are nonnegative.

The computational model

The input set X_w is a weighted set of n points having rational weights whose numerators and denominators are bounded by $O(\text{poly}(n))$. The algorithm works in the random access machine model with word size $O(\log n)$. Each memory word can be accessed in constant time. With each update, a new point is inserted, an existing point is deleted, or the weight of an existing point is modified by adding or subtracting a nonnegative number. The net weight of each point always stays nonnegative with its numerator and denominator always bounded by $O(\text{poly}(n))$.

We will prove some basic lemmas about coresets. Using these, we can take weighted union of two coresets without any loss (Lemma 5) and take a coreset of a coreset without much loss (Lemma 6).

---

[4] To be precise: denote unweighted version of X_w by X', then $\mathcal{W}$ is essentially $\mathbb{R}_+^{X'}$.

▶ **Lemma 5.** *If C_w^1 and C_w^2 are ε-coresets of X_w^1 and X_w^2, respectively, with respect to a linear cost function c such that $X_w^1 \cap X_w^2 = \emptyset$, then $C_w^1 \cup C_w^2$ is an ε-coreset of $X_w^1 \cup X_w^2$.*

Proof. By linearity of c: for any $S \in Q$,

$$c(S, X_w^1 \cup X_w^2) = c(S, X_w^1) + c(S, X_w^2) \in [1 \pm \varepsilon] \left(c(S, C_w^1) + c(S, C_w^2) \right) = [1 \pm \varepsilon] c(S, C_w^1 \cup C_w^2),$$

where, recall that, $C_w^1 \cup C_w^2$ is a weighted union.   ◀

▶ **Lemma 6.** *If C_w' is an ε-coreset of C_w, and C_w'' is a δ-coreset of C_w', both with respect to c, then C_w'' is an $(\varepsilon + \delta + \varepsilon\delta)$-coreset of C_w with respect to c.*

Proof. For any $S \in Q$, we have $c(S, C_w) \in [1 \pm \varepsilon]c(S, C_w')$ and $c(S, C_w') \in [1 \pm \delta]c(S, C_w'')$. So,

$$c(S, C_w) \geqslant (1-\varepsilon)c(S, C_w') \geqslant (1-\varepsilon)(1-\delta)c(S, C_w'') \geqslant (1 - \varepsilon - \delta - \varepsilon\delta)c(S, C_w''),$$

and $c(S, C_w) \leqslant (1+\varepsilon)c(S, C_w') \leqslant (1+\varepsilon)(1+\delta)c(S, C_w'') = (1 + \varepsilon + \delta + \varepsilon\delta)c(S, C_w'')$.   ◀

Let C_w^1 be an ε-coreset of C_w and C_w^2 be an ε-coreset of C_w^1. Then we say that C_w^1 and C_w^2 are, respectively, 1-level and 2-level ε-coresets of C_w. Extending this notion, we define an i-level ε-coreset to be an ε-coreset of an $(i-1)$-level ε-coreset.

▶ **Lemma 7.** *If C_w^ℓ is an ℓ-level ε-coreset of C_w, then C_w^ℓ is a $\left(\sum_{i=1}^{\ell} \binom{\ell}{i} \varepsilon^i \right)$-coreset of C_w.*

Proof. The proof is by induction on ℓ. Base case is when $\ell = 1$, and by definition, a 1-level coreset is an ε-coreset. By induction hypothesis, we have that $C_w^{\ell-1}$ is a $\left(\sum_{i=1}^{\ell-1} \binom{\ell-1}{i} \varepsilon^i \right)$-coreset of C_w. Now, C_w^ℓ is an ε-coreset of $C_w^{\ell-1}$, hence C_w^ℓ is an $\left(\varepsilon + (1+\varepsilon) \sum_{i=1}^{\ell-1} \binom{\ell-1}{i} \varepsilon^i \right)$-coreset of C_w by Lemma 6. Now, use Lemma 8, which appears below, with $\alpha = \varepsilon$ to finish the proof.   ◀

We prove two basic lemmas.

▶ **Lemma 8.** *For any positive integer ℓ and $\alpha \in \mathbb{R}_+$, we have $\alpha + (1+\alpha) \sum_{i=1}^{\ell-1} \binom{\ell-1}{i} \alpha^i = \sum_{i=1}^{\ell} \binom{\ell}{i} \alpha^i$.*

Proof idea. The proof is provided in Appendix A and uses elementary identities involving binomial coefficients and algebraic manipulations.   ◀

▶ **Lemma 9.** *For any positive integer ℓ and $\alpha \in [0, 1]$, we have $\sum_{i=1}^{\ell} \binom{\ell}{i} \left(\frac{\alpha}{2\ell} \right)^i \leqslant \alpha$.*

Proof. $\sum_{i=1}^{\ell} \binom{\ell}{i} \left(\frac{\alpha}{2\ell} \right)^i \leqslant \sum_{i=1}^{\ell} \ell^i \frac{\alpha^i}{2^i \ell^i} = \sum_{i=1}^{\ell} \frac{\alpha^i}{2^i} \leqslant \sum_{i=1}^{\ell} \frac{\alpha}{2^i} \leqslant \alpha$.   ◀

Now, as a corollary to Lemma 7, we get the following using Lemma 9.

▶ **Corollary 10.** *If C_w^ℓ is an ℓ-level $(\varepsilon/(2\ell))$-coreset of C_w, then C_w^ℓ is an ε-coreset of C_w.*

As we discussed earlier, rounding of the weights at internal nodes is needed in our dynamic algorithm to achieve the desired worst-case update time. Towards that, we need two lemmas.

In the next lemma, think of a/b as the original weight of the point, c/d as the weight that we want to approximate a/b with, and D as the cost of this point with respect to a feasible solution in Q. So the lemma says that by rounding, the cost of the point stays within $1 \pm b/d$ of the original cost.

▶ **Lemma 11.** *For positive integers a, b, and d, let $c = \lfloor ad/b \rfloor$. Then $cD/d \in [1 \pm b/d]aD/b$ for any nonnegative real D.*

Proof. By the definition of c, we have that $c/d \leqslant a/b \leqslant c/d + 1/d$, and $1/d \leqslant a/d$ because $a \geqslant 1$; hence $a/b \geqslant c/d \geqslant a/b - a/d$, which implies that $aD/b \geqslant cD/d \geqslant aD/b - aD/d = (1 - b/d)aD/b$.   ◀

The proof of the following lemma is very similar. Here, think that we approximate the weight r of a point by $\lfloor r \rfloor + c/d$ and the cost of the point stays within $1 \pm 1/d$ of the original cost.

▶ **Lemma 12.** *Let $r \geqslant 1$ be a rational number, a and b be positive integers such that $a/b = r - \lfloor r \rfloor$, d be any positive integer, and $c = \lfloor ad/b \rfloor$. Then $(\lfloor r \rfloor + c/d)D \in (1 \pm 1/d)rD$ for any nonnegative real D.*

Proof. By the definition of c and using $r \geqslant 1$, we get that $a/b \geqslant c/d \geqslant a/b - r/d$; adding $\lfloor r \rfloor$ and multiplying by D finishes the proof.   ◀

3   A Dynamic Coreset

We describe our dynamic algorithm for maintaining an ε-coreset for a problem P with query space Q that uses a static coreset algorithm, say, ALG_S.
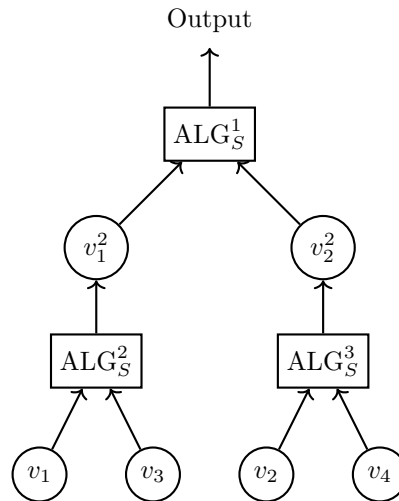

◼ **Figure 1** An ALG_S node takes input from two point-nodes. If the union of the sets has cardinality greater than s', then the ALG_S node computes a coreset of cardinality at most s' and passes it on to the point-node above it (its parent). The number of leaf nodes is always n, and the number of levels is always $O(\log n)$, where n is the current number of points.

The main idea is described in Figure 1 using a tree with a special structure. Each node is of one of the two types: a point-node representing a weighted set of points or an alg-node representing an instance of ALG_S. We sometimes use a point-node to denote the point set it represents and an alg-node to denote the ALG_S instance it represents. Each level contains either only point-nodes or only alg-nodes. All leaf nodes are point-nodes and represent a weighted singleton with an input point. Each alg-node gets as input the weighted union of its children, and its output is represented by its parent node (which is a point-node). When running ALG_S at an alg-node A, if the union of its children has cardinality larger than s', then A would compute a coreset of cardinality at most s' otherwise it would just output the weighted union. We will later fix this threshold s' for computing a coreset. An example of

how insertions and deletions are handled is shown in Figure 2 (where all weights are assumed to be one). For the ease of description, from now onwards, we will think of this tree with alg-nodes being collapsed into their parent nodes. Then each leaf node would contain a weighted singleton and each internal node would contain the output of the ALG_S instance run on the weighted union of its children's sets.
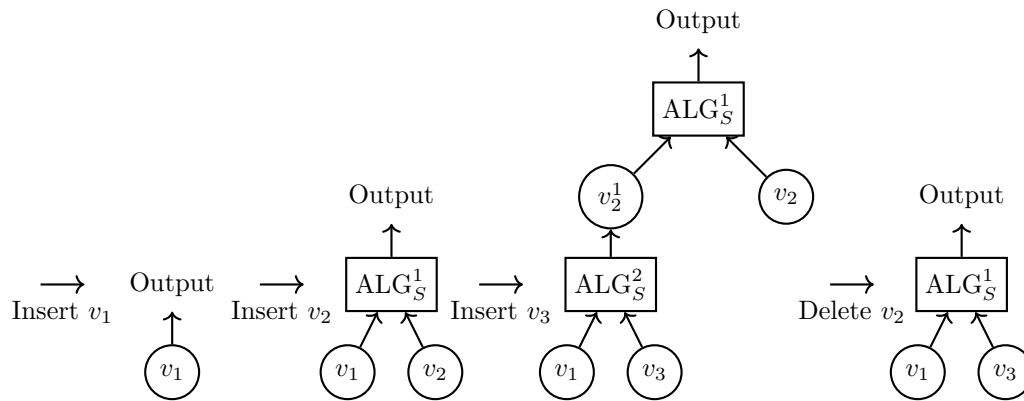
Figure 2 An example of how insertions and deletions are handled. We start with an empty tree. The first point that is inserted is represented by v_1. We use a point and the node that represents it interchangeably. Then v_2 is inserted followed by v_3. Next, if v_4 is inserted, we get exactly the tree shown in Figure 1, and if v_2 is deleted, then we get the last tree.

We guarantee that the resulting tree then will always be a *complete* binary tree, i.e., every level except possibly the lowest is completely filled, and the nodes at the lowest level are packed to the left. To describe the updates briefly, let ℓ_r denote the rightmost leaf node at the lowest level; for simplicity, assume that the lowest level is not full. Insertion is straightforward: the new point goes in a new leaf node to the right of ℓ_r. For deletion of a point at leaf node ℓ_d, if $\ell_d \neq \ell_r$, then we replace contents of ℓ_d with those of ℓ_r and delete ℓ_r. See Section 3.1 for details of these operations. For weight update, the tree does not change.

▶ Remark 13. Since a coreset will not be computed until a node has more than s' points, the tree can be modified so that each leaf node corresponds to a set of $\Theta(s')$ points. Then the number of nodes in the tree is $\Theta(n/s')$. This reduces the additional space used for maintaining this tree. This is important when the number of points is very large. See Section 3.2 for further details. This is essentially the same idea as used for asymmetric sparsification in Section 3.4 in Eppstein et al. [11].

We call the leaf nodes at the same level as that of the leftmost leaf node to be at level 0. We increment these level numbers naturally as we move upwards in the tree. Since we maintain a complete binary tree, the root, which is at the highest level, is on level $\lceil \log n \rceil$. After a point insertion, deletion, or weight update, we recompute all the nodes that are affected by running ALG_S from scratch. Once we update a leaf node, all the nodes on its leaf-to-root path are affected. Since at most two leaf nodes are updated after every point update, we run at most $2\lceil \log n \rceil$ instances of ALG_S. Finally, to reduce the cardinality of our output coreset, we run another *outer* instance of ALG_S with $\varepsilon_s = \varepsilon/3$ and $\lambda_s = \lambda/2$ with input as the output of the root. Here, ε_s and λ_s are parameters for ALG_S as described below, and our goal is to compute an ε-coreset with probability at least $1-\lambda$. The outer instance is run after every update.

The static coreset algorithm ALG_S takes as input an integer weighted set of n_s points with total weight W_s and always returns a weighted set of cardinality at most $s(\varepsilon_s, \lambda_s, W_s)$; this set is an ε_s-coreset with probability at least $1-\lambda_s$. Let the running time of ALG_S be $t(n_s, \varepsilon_s, \lambda_s, W_s)$. We assume that the functions t and s are nondecreasing in W_s and nonincreasing in ε_s and λ_s, and also that t is nondecreasing in n_s. We call such functions t and s *well-behaved*.

We note that t and s implicitly depend on the query space Q as well. In particular, for k-median and k-means, they depend on k and the dimension or the cardinality of the universe from which a solution is allowed to be picked. Also, assume that the total weight of ALG_S's output is at most $1+\delta$ times the total input weight and it outputs a coreset of points with integer weights. For the dynamic algorithm, n denotes the current number of points, and we assume that any input weight is a rational number with numerator and denominator bounded by n^c, for a fixed constant c.

▶ **Theorem 14.** *Assume that there is a static algorithm ALG_S that takes as input an integer-weighted set of n_s points with total weight W_s and always returns an integer-weighted set of cardinality at most $s(\varepsilon_s, \lambda_s, W_s)$ with total weight at most $(1+\delta)W_s$, and this set is an ε_s-coreset with probability at least $1-\lambda_s$. Let the running time of ALG_S be $t(n_s, \varepsilon_s, \lambda_s, W_s)$, and assume that both s and t are well-behaved. Then there is a fully-dynamic algorithm that, on rational-weighted input points, always maintains an $s\left(\frac{\varepsilon}{3}, \frac{\lambda}{2}, W_p\right)$-cardinality weighted set. This set is an ε-coreset with probability at least $1-\lambda$. Its worst-case update time is*

$$O\left(t\left(2s^*, \frac{\varepsilon}{6\lceil \log n_p \rceil}, \frac{\lambda}{2n_p}, W_p\right) \cdot \left(1 + \log(1+\delta) + \frac{\log \varepsilon^{-1}}{\log n}\right) \cdot \log n\right),$$

where $W_p = (1+\delta)^{\lceil \log n_p \rceil} n_p^{c''} \lceil 1/\varepsilon \rceil$, c'' is a constant, $s^* = s\left(\frac{\varepsilon}{6\lceil \log 2n_p \rceil}, \frac{\lambda}{4n_p}, W_p\right)$, and $8n/3 \leqslant n_p \leqslant 8n$.

Proof. We first prove that the output of the algorithm is an ε-coreset if every non-outer ALG_S instance outputs an ε_s-coreset of its input for some $\varepsilon_s \leqslant \varepsilon/(6\lceil \log n \rceil)$ and the outer ALG_S instance outputs an $(\varepsilon/3)$-coreset of its input. We prove the following by induction on level number: every node at level ℓ contains a $(\sum_{i=1}^{\ell} \binom{\ell}{i} \varepsilon_s^i)$-coreset of the leaf nodes in its subtree. In the base case, a node at level 1 contains an ε_s-coreset of its input trivially. An ALG_S instance A at level i gets as input two sets, say C_w' and C_w'', each of which is a $(\sum_{i=1}^{\ell-1} \binom{\ell-1}{i} \varepsilon_s^i)$-coreset for the leaf nodes in their respective nodes' subtrees. Hence, $C_w' \cup C_w''$ is a $(\sum_{i=1}^{\ell-1} \binom{\ell-1}{i} \varepsilon_s^i)$-coreset for leaf nodes in the subtree rooted at A by Lemma 5. Now, A outputs an ε_s-coreset of $C_w' \cup C_w''$, hence by Lemma 6, its output is an $(\varepsilon_s + (1+\varepsilon_s) \sum_{i=1}^{\ell-1} \binom{\ell-1}{i} \varepsilon_s^i)$-coreset of the leaf nodes in its subtree, which, by Lemma 8, means a $(\sum_{i=1}^{\ell} \binom{\ell}{i} \varepsilon_s^i)$-coreset. This completes the induction step. Hence, the root node, which is at level $\lceil \log n \rceil$, contains $(\sum_{i=1}^{\lceil \log n \rceil} \binom{\lceil \log n \rceil}{i} \varepsilon_s^i)$-coreset. Now, since $\varepsilon_s \leqslant \varepsilon/(6\lceil \log n \rceil)$, by Lemma 9, the output at the root is an $(\varepsilon/3)$-coreset. The outer ALG_S instance outputs an $(\varepsilon/3)$-coreset of this, hence, by Lemma 6, the final output is an $(2\varepsilon/3 + \varepsilon^2/9)$-coreset, which is an ε-coreset of all points.

Recall that the running time of ALG_S is $t(n_s, \varepsilon_s, \lambda_s, W_s)$ to compute an ε_s-coreset with probability at least $1-\lambda_s$, where n_s is the number of points in the input. Our output success probability will depend on λ_s, and ε depends on ε_s as proved in the previous paragraph. We will need $\varepsilon_s \leqslant \varepsilon/(6\lceil \log n \rceil)$ and $\lambda_s \leqslant \lambda/(2n)$, so these depend on n, which can change a lot over time. We now show how to maintain these guarantees for ε_s and λ_s after each update.

Towards this, we need a little tweak to our algorithm and an additional maintenance routine that we call the *refresher*. The algorithm works in phases. The refresher routine maintains two *refresh* pointers that always point to consecutive leaf nodes, say r_1 and r_2.

The refresh pointers are reset after the end of a phase as follows. If the number of leaf nodes is a power of 2, then r_1 and r_2 point to the two leftmost leaf nodes, otherwise they point to the two leftmost leaf nodes at the level above the lowest level. Assume, for completeness, that the very first phase ends after receiving two points, so the tree is just two leaf nodes and their parent as the root.

For each subsequent phase, let n_0 be the value of n at the beginning of the phase. Each phase ends after $n_0/2$ updates, and we set $n_p = 4n_0$. This guarantees that n_p is greater than n throughout the whole phase and even the next phase (details appear below). After receiving an update, we rerun all the ALG_S instances on the leaf-to-root path starting at r_1 and r_2 (at most $2\lceil \log n \rceil$ such instances). This is the refresher routine. Then we move the refresh pointers to the next two leaf nodes on the right. If we reach the right end, then we go to the next level if it exists, otherwise we stop. If we stop, then we achieved the goal of (re-)running all the ALG_S instances that are present at the end of the phase at least once in this phase (this will become clearer below). After the refresher routine, we execute the update which affects at most two leaf nodes. We rerun all the ALG_S instances that are affected by this update, again, at most $2\lceil \log n \rceil$ such instances. So in total, at most $4\lceil \log n \rceil$ of non-outer ALG_S instances are run after an update and one outer instance, which explains the $\log n$ factor in the update time. We now explain the parameters used in the ALG_S instances. For all the non-outer ALG_S instances, we use $\varepsilon_s = \varepsilon/(6\lceil \log n_p \rceil)$ and $\lambda_s = \lambda/(2n_p)$. (This explains the ε_s and λ_s parameters of the functions t and s in the theorem statement.) Note here that the running time of the outer instance is going to be less than any non-outer instance because t is non-increasing in ε_s and λ_s.

As we use $n_p = 4n_0$ and there could be at most $n_0/2$ insertions in a phase, the final value of n is at most $3n_0/2$, and, thus, n_p is always greater than n. In fact, crucially, n_p is an upper bound on n for even the next phase; in the next phase, $n \leqslant n_0 + n_0/2 + (n_0 + n_0/2)/2 = 9n_0/4 \leqslant n_p$. Also, in the current phase, $n_0/2 \leqslant n \leqslant 3n_0/2$, hence $8n/3 \leqslant n_p \leqslant 8n$, as required (cf. the theorem statement).

We now prove that any non-outer ALG_S instance uses $\varepsilon_s \leqslant \varepsilon/(6\lceil \log n \rceil)$ and $\lambda_s \leqslant \lambda/(2n)$ at any time instant. Let L be the set of leaf nodes at the beginning of the phase; therefore, $|L| = n_0$. An ALG_S instance that exists at the end of the phase is either on the leaf-to-root path for some leaf in L or it was created/updated in this phase. At the end of the phase, the refresh pointers will hit all surviving leaf nodes in L; the argument is as follows. Each phase lasts for $n_0/2$ updates, $|L| = n_0$, and we move the two refresh pointers to the right on next two leaf nodes after each update. Importantly, new leaf nodes are added only to the right of the rightmost leaf node at the lowest level, and hence, the refresher routine will have hit all surviving leaf nodes in L before hitting a newly created leaf node.

This shows that, in any case (being either hit by a point update or by the refresher routine), each ALG_S instance is run with $n_p = 4n_0$, setting up these instances for the next phase. This means that at any time instant, each ALG_S instance was created/updated in the current phase or created/updated in the previous phase, thus showing that $\varepsilon_s \leqslant \varepsilon/(6\lceil \log n \rceil)$ and $\lambda_s \leqslant \lambda/(2n)$ for all ALG_S instances at all times.

At any time instant, there are at most n non-outer instances of ALG_S, each with success probability at least $1 - \lambda/(2n)$, and the outer ALG_S instance has success probability at least $1 - \lambda/2$. Hence, the final success probability is at least $1-\lambda$ by the union bound over these $n+1$ instances.

How to handle weights

We will need one further tweak to argue that each weight ever encountered by the algorithm can be stored using $O(1 + \log(1+\delta) + \log(1/\varepsilon)/\log n)$ words, which also explains that factor in the update time. By assumption, an insertion or weight update comes with a weight that is a fraction with the numerator and the denominator bounded by n^c for some fixed constant c. After receiving such an update, we approximate the weight by a fraction that has numerator bounded by $n_p^{c'}\lceil 1/\varepsilon \rceil$, where $c' = 2c + 1$ is also a fixed constant, and the denominator is equal to $n_p^{c+1}\lceil 1/\varepsilon \rceil^5$. The change in the cost due to this approximation is at most ε/n_p times the original cost; hence, by the linearity of the cost function, the output coreset quality is affected by at most an additive factor of $O(\varepsilon/n)$. More formally, the following claim holds by Lemma 11 and using $b/d \leqslant \varepsilon/n_p$ below (think of D below as cost).

$\triangleright$ **Claim 15.** Let $d = n_p^{c+1}\lceil 1/\varepsilon \rceil$. Given a rational number a/b, where a and b are integers, $a \leqslant n_p^c$ and $b \leqslant n_p^c$, let $f = \lceil ad/b \rceil$. Then $f \leqslant n_p^{2c+1}\lceil 1/\varepsilon \rceil$ and $(f/d)D \in [1 \pm \varepsilon/n_p](a/b)D$ for any nonnegative real D.

Recall that due to the refresher routine, at any time instant, the denominator of the weight at any leaf node can be one of the two: $n_p^{c+1}\lceil 1/\varepsilon \rceil$ or $n_{pp}^{c+1}\lceil 1/\varepsilon \rceil$, where n_{pp} is the value of n_p for the previous phase. When the two children of an internal node use different denominators, this complicates our rounding scheme. Thus, when taking a union of the children's sets at an internal node, for each weight, we make its numerator an integer and the denominator equal to $(n_p n_{pp})^{c+1}\lceil 1/\varepsilon \rceil$, which is a common multiple of $n_p^{c+1}\lceil 1/\varepsilon \rceil$ and $n_{pp}^{c+1}\lceil 1/\varepsilon \rceil$ – the only possible denominators of an input weight after rounding. Next, we run the ALG_S instance with integer weights as given by the numerator, then (implicitly) dividing the output weights by the denominator $(n_p n_{pp})^{c+1}\lceil 1/\varepsilon \rceil$ afterwards. Since each ALG_S instance can increase the total weight by at most a factor of $1+\delta$, the sum of the numerators of all weights at level i is always bounded by $n(1+\delta)^i(n_p n_{pp})^{c'}\lceil 1/\varepsilon \rceil$. Since $i \leqslant \lceil \log n \rceil$ and $n_{pp} = \Theta(n_p)$, there exists a constant c'', such that the sum of the numerators of all weights at any level i and all the possible numerators and denominators are bounded by $(1+\delta)^{\lceil \log n_p \rceil} n_p^{c''}\lceil 1/\varepsilon \rceil =: W_p$, and hence, can be stored in $O(1 + \log(1+\delta) + \log(1/\varepsilon)/\log n)$ words as desired (see the beginning of the paragraph before Claim 15). This also justifies the W_s parameters of the functions t and s in the theorem statement.

Now we put everything together. The outer ALG_S instance outputs a weighted set of size at most $s\left(\frac{\varepsilon}{3}, \frac{\lambda}{2}, W_p\right)$. This set is an ε-coreset with probability at least $1-\lambda$, which we proved by a union bound over all ALG_S instances. We set $s' = s\left(\frac{\varepsilon}{6\lceil \log n_p \rceil}, \frac{\lambda}{2n_p}, W_p\right)$, which is the threshold for computing a coreset at each internal node, i.e., (recall that) if the number of points at an internal node is greater than s', then we run ALG_S to compute a coreset. An upper bound on the threshold for the current phase and the previous phase is $s^* = s\left(\frac{\varepsilon}{6\lceil \log 2n_p \rceil}, \frac{\lambda}{4n_p}, W_p\right)$ because the n_p value for the previous phase can be at most twice that of the current phase. Then the worst-case update time is dominated by the non-outer

---

[5] The static algorithm ALG_S expects integer-weighted input and outputs integer-weighted points, whereas our dynamic algorithm handles fractional weights. If fractional weights are naïvely stored in our dynamic algorithm, then at internal nodes, combining two fractions may result in larger magnitude numbers. E.g., naïvely handling two points with weights a/b and c/d so as to be used in ALG_S results in weights $ad/(bd)$ and $bc/(bd)$. Thus, at level i, the numerators and denominators may be as large $(\mathrm{poly}(n))^{2^i}$. Note that some rounding would be needed even if ALG_S can handle rational weights, because its output may be points with rational weights having much larger magnitude; e.g., even if the output magnitude is about only quadratic in that of the input, the blowup near the root in our dynamic algorithm would be nth power of the input. In fact, we do this rounding in the proof of Theorem 2.

ALG_S instances, each running in time $t\left(2s^*, \frac{\varepsilon}{2\lceil \log n_p \rceil}, \frac{\lambda}{2n_p}, W_p\right)$, and we run $O(\log n)$ of these after receiving an update. An additional factor of $1 + \log(1+\delta) + \log(1/\varepsilon)/\log n$ appears because each weight may need memory worth $O(1 + \log(1+\delta) + \log(1/\varepsilon)/\log n)$ words, and we need constant time to access each memory word. ◄

Before proving the concrete bounds for k-median and k-means that are stated in Theorem 2, we prove a weaker theorem that is a direct consequence of Theorem 14 using the static algorithm of Chen [9].

▶ **Theorem 16.** *For the k-median and k-means problems, there is a fully-dynamic algorithm that maintains a set of cardinality $O(\varepsilon^{-2}k\log^2(n/\varepsilon)(k\log n + \log(1/\lambda)))$, that is an ε-coreset with probability at least $1-\lambda$, and has worst-case update time*

$$O\left(\varepsilon^{-2}k^2\log^3 n \log^2 \frac{n}{\varepsilon} \log \frac{n}{\lambda}\left(k\log n + \log \frac{n}{\lambda}\right)\log\log \frac{n}{\varepsilon}\left(1 + \frac{\log \varepsilon^{-1}}{\log n}\right)\right).$$

Ignoring the $\log \log n$ factors, for $\lambda = \Omega(1/\operatorname{poly}(n))$ and $\varepsilon = \Omega(1/\operatorname{poly}(n))$, the coreset cardinality is $O(\varepsilon^{-2}k^2\log^3 n)$, and the worst-case update time is $O(\varepsilon^{-2}k^3\log^7 n)$.

Proof. Chen's algorithm takes in an integer weighted set and outputs also an integer weighted set. Its output has the same total weight as the input, so $\delta = 0$ (see Theorem 14). Also, for Chen's algorithm, $s(\varepsilon_s, \lambda_s, W_s) = O(\varepsilon_s^{-2}k(k\log n + \log(1/\lambda_s))\log^2 W_s)$ and the running time $t(n_s, \varepsilon_s, \lambda_s, W_s) = O(n_s k \log(1/\lambda_s) \log \log W_s)$ (see Theorems 3.6 and 5.5 in Chen [9]), which is dominated by the computation of a bicriteria approximation. Note that both s and t are well-behaved. Using $W_p = O(\operatorname{poly}(n)/\varepsilon)$,

$$s^* = O\left(\varepsilon^{-2}k\log^2 n \log^2 \frac{n}{\varepsilon}\left(k\log n + \log \frac{n}{\lambda}\right)\right),$$

and $\delta = 0$ in Theorem 14 gives the desired bounds using the functions t and s above. ◄

Now we use the result of Braverman et al. [4] to get better bounds as stated in Theorem 2 in the introduction section. Unfortunately, we cannot use Theorem 14 as a complete black box for this because in this case, on integer weighted input, ALG_S does not output an integer weighted coreset. The proof of the following theorem is thus an extension of the proof of Theorem 14.

▶ **Theorem 2.** *For the k-median and k-means problems, there is a fully-dynamic algorithm that maintains a set of cardinality $O(\varepsilon^{-2}k(\log n \log k \log(k\varepsilon^{-1}\log n) + \log(1/\lambda)))$, that is an ε-coreset with probability at least $1-\lambda$, and has worst-case update time*

$$O\left(\varepsilon^{-2}k^2\log^5 n \log^3 k \log^2(\varepsilon^{-1})(\log \log n)^3\right),$$

assuming that $\varepsilon = \Omega(1/\operatorname{poly}(n))$ and $\lambda = \Omega(1/\operatorname{poly}(n))$.

Proof. Our dynamic algorithm expects to have at its disposal a static algorithm ALG_S that takes integer-weighted input and outputs an integer-weighted coreset. Since the algorithm of Braverman et al. that we use as ALG_S outputs on *integer* weighted input a coreset with *fractional* weights, we need some modifications. Hence, before ALG_S is ready to be used in the dynamic algorithm, we round its output to turn it into integers.

Weight-Rounding Modifications for ALG_S

- Let input to ALG_S be Y_w which is a set of n_s points with integer weights $w(1), \dots, w(n_s)$.
- We run ALG_S on the same points with scaled weights $s'w(1), \dots, s'w(n_s)$, where s' is the desired cardinality of the output coreset (which is the same as the threshold for computing a coreset at an internal node in this case). We set s' later in a such a way that it can be computed by our dynamic algorithm. This step of multiplying input weights by s' is done to make sure that each of the fractional weights output by ALG_S is at least 1 (see Line 6 of Algorithm 2 in Braverman et al. [4]).
- Let the output C_w of ALG_S be a weighted set of s' points with fractional weights $w_o(1), \dots, w_o(s')$. Using the rounding strategy of Lemma 12, round these fractional weights to have an integer numerator and the denominator equal to $\lceil (\log n_p)/\varepsilon \rceil$ to get weights $\tilde{w}(1), \dots, \tilde{w}(s')$, where n_p is as defined in the proof of Theorem 14. Formally, for $i \in \{1, \dots, s'\}$:

$$
\tilde{w}(i) = \lfloor w_o(i) \rfloor + \frac{\left\lfloor (w_o(i) - \lfloor w_o(i) \rfloor) \left\lceil \frac{\log n_p}{\varepsilon} \right\rceil \right\rfloor}{\left\lceil \frac{\log n_p}{\varepsilon} \right\rceil}.
$$

  Since $w_o(i) \geqslant 1$, by Lemma 12, for any real $D \geqslant 0$, we have $\tilde{w}(i)D \in [1 \pm \varepsilon/\log n_p] w_o(i)D$.
- Hence, by the linearity of the cost function, C_w with weights $\tilde{w}(1)/s', \dots, \tilde{w}(s')/s'$ is an $(\varepsilon_s + 2\varepsilon/\log n_p)$-coreset of Y_w with weights $w(1), \dots, w(n_s)$ if C_w with weights $w_o(1), \dots, w_o(s')$ is an ε_s-coreset of Y_w with weights $s'w(1), \dots, s'w(n_s)$. Note that $\tilde{w}(i)/s'$ can be represented as a fraction with an integer numerator and denominator equal to $s'\lceil (\log n_p)/\varepsilon \rceil$.
- The additive loss of $2\varepsilon/\log n_p$ in the coreset quality due to this rounding is tolerable because every non-outer ALG_S instance will be run with $\varepsilon_s = O(\varepsilon/\log n_p)$[6]. Hence, the coreset quality at internal nodes will always be $O(\varepsilon_s + \varepsilon/\log n_p) = O(\varepsilon/\log n)$, as desired.
- This rounding ensures that on integer-weighted input with total weight W, the output weights of ALG_S are fractions with integer numerator bounded by $(1+\delta)Ws'\lceil (\log n_p)/\varepsilon \rceil$ and integer denominator equal to $s'\lceil (\log n_p)/\varepsilon \rceil$. Here, $1+\delta$ is the factor by which ALG_S can increase the total weight.

To handle rational weights in the dynamic algorithm, we first proceed as described in the paragraph on how to handle weights in the proof of Theorem 14. Recall that we assume that each insertion or weight update by the adversary comes with a weight that is a fraction with the numerator and the denominator bounded by n^c for some fixed constant c, and we set $c' = 2c+1$. Also, each leaf node was created/updated in the current phase or created/updated in the previous phase and thus uses the value either n_p or n_{pp}, where n_{pp} is the value of n_p for the previous phase. We then showed the following. At any time instant, the weight of the point at a leaf node is rounded in such a way that the numerator is bounded by $n_p^{c'}\lceil 1/\varepsilon \rceil$ and the denominator is equal to $n_p^{c+1}\lceil 1/\varepsilon \rceil$, or the numerator is bounded by $n_{pp}^{c'}\lceil 1/\varepsilon \rceil$ and the denominator is equal to $n_{pp}^{c+1}\lceil 1/\varepsilon \rceil$. Due to this rounding, the output coreset quality is affected by at most an additive factor of $\max\{2\varepsilon/n_p, 2\varepsilon/n_{pp}\} = O(\varepsilon/n)$. We now prove the following more general statement towards the current proof.

---

[6] If we go for smaller additive loss, say ε/n_p, the denominators of resulting numbers due to this rounding would become exponential in n_p. And if we go for a larger additive loss, it would worsen the coreset quality at non-outer instances to $\omega(\varepsilon/\log n_p)$ resulting in the quality of the output coreset worse than ε.

▶ **Lemma 17.** *At any time instant, every weight at a node at level i has an integer numerator and a denominator that is a factor of $(n_p n_{pp})^{c+1} \lceil 1/\varepsilon \rceil (s_p' s_{pp}' \lceil (\log n_p)/\varepsilon \rceil \lceil (\log n_{pp})/\varepsilon \rceil)^i =: D(i)$, where s_p' and s_{pp}' are values of the threshold s' in the current and the previous phase, respectively.*

Proof. We prove this statement by induction over the sequence of nodes updated by the algorithm.

In the base case, the first ever node update will be due to creation of a leaf node, and the weight will have denominator $n_p^{c+1} \lceil 1/\varepsilon \rceil$. Next we discuss the induction step. Let the update be on a node at level i, so we run the *modified* ALG_S instance with all weights having a denominator that is a factor of $D(i-1)$, which is true by induction hypothesis. Then, since the modified ALG_S adds a factor of $s_p' \lceil (\log n_p)/\varepsilon \rceil$ to the denominator, all resulting output weights have a denominator that is a factor of $D(i-1) s_p' \lceil (\log n_p)/\varepsilon \rceil$, which is a factor of $D(i)$. This finishes the induction step for the case when the node update is not the last of the phase. When the node being updated is the last of the phase, we have to be careful. In this case, we need to show that for all weights in all nodes, n_{pp} or s_{pp}' do not appear in the denominator, as this will set these denominators for the next phase. Towards this, we need the following claim.

▷ **Claim 18.** Let u be a node at level i. Fix a time instant. Suppose, in the current phase, all nodes in the subtree rooted at u were updated and u was updated after the update of the last-updated leaf node in the subtree. Then the denominator of the weights at u is a factor of $n_p^{c+1} \lceil 1/\varepsilon \rceil (s_p' \lceil (\log n_p)/\varepsilon \rceil)^i$ at the fixed time instant.

We omit the proof of this claim as it can be proved easily by induction on the level number at any fixed time instant.

After the last node update of the phase, every node in the tree has been updated in the current phase and the premise of Claim 18 holds due to the refresher routine. Hence, by Claim 18, after the last node update of the phase, i.e., just before the new phase begins, all denominators at level i are a factor of $n_p^{c+1} \lceil 1/\varepsilon \rceil (s_p' \lceil (\log n_p)/\varepsilon \rceil)^i$. Since n_p and s_p' of this phase will become n_{pp} and s_{pp}' in the next phase, the induction hypothesis stays true for the next phase as well. This finishes the proof of Lemma 17. ◀

Since an ALG_S instance may increase the total weight by at most a factor of $1+\delta$, the sum of the numerators of weights at any level i is at most

$$n_p (1+\delta)^i (n_p n_{pp})^{c'} \left\lceil \frac{1}{\varepsilon} \right\rceil \left(s_p' s_{pp}' \left\lceil \frac{\log n_p}{\varepsilon} \right\rceil \left\lceil \frac{\log n_{pp}}{\varepsilon} \right\rceil \right)^i ;$$

this can be seen by an easy induction on the level number. Using this bound, we set the threshold s' in a way similar to that in the proof of Theorem 14: we set

$$s_p' = s \left(\frac{\varepsilon}{6 \lceil \log n_p \rceil}, \frac{\lambda}{2 n_p}, W_p \right) ,$$

where

$$W_p = (1+\delta)^{\lceil \log n_p \rceil} n_p^{c_1} \left(k \left\lceil \frac{\log n_p}{\varepsilon} \right\rceil \right)^{c_2 \lceil \log n_p \rceil} ,$$

and c_1 and c_2 are chosen to be large enough constants so that W_p upper bounds the sum of the numerators of all weights at any level. From now onwards, we assume that $\lambda = \Omega(1/\operatorname{poly}(n))$. For ALG_S, the function s is $s(\varepsilon_s, \lambda_s, W_s) = O(\varepsilon_s^{-2} k(\log k \log W_s + \log(1/\lambda_s)))$ and $\delta = O(\varepsilon)$.

Then, using $n_{pp} = \Theta(n_p)$, we get that both s'_p and s'_{pp} are $O\left(\left(k\left\lceil\frac{\log n_p}{\varepsilon}\right\rceil\right)^{c_3}\right)$, where c_3 is a fixed constant (so, independent of c_1 and c_2). Observe that W_p and thus s'_p are determined by the phase and hence can be computed by our algorithm. More concretely, we get that both s'_p and s'_{pp} are

$$O\left(\varepsilon^{-2}k\log^3 n\log k\log\frac{k\log n}{\varepsilon}\right).$$

All possible numerators and denominators encountered by the algorithm are bounded by

$$N := O\left(\operatorname{poly}(n)\left(\frac{k\log n}{\varepsilon}\right)^{O(\log n)}\right),$$

so, can be stored in $m := (\log N)/\log n = O(\log((k\log n)/\varepsilon))$ words.

The running time of ALG_S is $t(n_s, \varepsilon_s, \lambda_s, W_s) = O(n_s k\log(1/\lambda_s)\log\log W_s)$, which, similar to Chen's algorithm, is dominated by computation of a bicriteria approximation. At a non-outer ALG_S instance, $n_s = O(s'_p)$, $\varepsilon_s = O(\varepsilon/\log n_p)$, $\lambda_s = O(\lambda/n_p)$, and $W_s \leqslant W_p$. With every update, $O(\log n)$ instances of ALG_S are run, and an additional m factor appears because a weight may need up to m words. Hence, the worst-case update time assuming $\varepsilon = \Omega(1/\operatorname{poly}(n))$ and $\lambda = \Omega(1/\operatorname{poly}(n))$ is

$$O\left(t\left(s'_p, \frac{\varepsilon}{\log n}, \frac{\lambda}{n}, W_p\right)m\log n\right) = O\left(\varepsilon^{-2}k^2\log^5 n\log k\log^2\left(\frac{k\log n}{\varepsilon}\right)\log\log\left(\frac{k\log n}{\varepsilon}\right)\right)$$

and a looser, easier to parse, bound is $O\left(\varepsilon^{-2}k^2\log^5 n\log^3 k\log^2(1/\varepsilon)(\log\log n)^3\right)$. The output coreset cardinality is

$$s\left(\frac{\varepsilon}{3}, \frac{\lambda}{2}, W_p\right) = O\left(\varepsilon^{-2}k\left(\log n\log k\log\left(\frac{k\log n}{\varepsilon}\right) + \log\frac{1}{\lambda}\right)\right).$$

This finishes the proof of Theorem 2. ◀

3.1 The Binary-Tree Structure

We describe the tree structure in more detail, especially, how insertions and deletions are handled. We always maintain a complete binary tree, in which every level except possibly the lowest is completely filled, and the nodes in the lowest level are packed to the left. We also maintain the property that each internal node has exactly two children. Our data structure behaves somewhat like a heap, though a crucial difference is that we do not have keys. This structure supports insertion and deletion of a leaf node. Insertion of a new leaf-node ℓ works as follows.

- If the current number of leaf nodes is a power of 2, then let v be the leftmost leaf node,
- Else let v be the leftmost leaf node in the level above the lowest level.
- Let p be v's parent.
- Create a new node u.
- Make p to be u's parent; u replaces v, so if v was p's right (respectively, left) child, then u is now p's right (respectively, left) child.
- Make v to be u's left child and ℓ to be u's right child. This way, ℓ the rightmost leaf node at the lowest level.

Deletion of a leaf-node ℓ works as follows. Let v be the rightmost leaf node at the lowest level, p be v's parent, and v' be v's sibling. Replace ℓ's contents by v's contents and replace p's contents by the contents of v'. Delete v and v'.

3.2 Reducing the Number of Nodes

The tree can be modified to have each leaf node correspond to a set of $\Theta(s')$ points to reduce the additional space used for maintaining this tree (pointers and such). Recall that s' is the threshold for computing a coreset. To reduce the number of nodes in the tree this way, we maintain the invariant that each leaf node, except possibly one, contains a set of size s_ℓ with $s'/2 \leqslant s_\ell \leqslant s'$. To maintain this invariant, we use a pointer p_s that points to a leaf node with less than $s'/2$ elements if such a leaf node exists.

Whenever a point is inserted, we add it to the leaf node, say ℓ_e pointed to by p_s. If ℓ_e now contains at least $s'/2$ points, then we make p_s a null pointer. If p_s was a null pointer already, then we create a new leaf node, say ℓ_n, insert the new point in ℓ_n, and make p_s point to ℓ_n. The new leaf node ℓ_n is inserted in the tree as described in Section 3.1.

Whenever a point is deleted, we check if the leaf node, say ℓ_d that contains it now contains less than $s'/2$ points. If ℓ_d contains less than $s'/2$ points, and p_s points to some leaf node, say ℓ_e, then we move points in ℓ_d into ℓ_e and delete ℓ_d. (Deletion of a leaf node is handled as described in Section 3.1.) If p_s does not point to any leaf node, then we make it point to ℓ_d.

As usual, we recompute all nodes on the affected leaf-to-root path.

4 Lower Bounds

In this section, we show lower bounds. We first see a space lower bound and then a conditional lower bound on the time per operation.

4.1 Space Lower Bound

We show a simple and very general space lower bound. Consider any problem that on input X has to output a feasible solution that is a subset of X. Moreover, if X non-empty, then all feasible solutions are also non-empty. Call such a problem *compliant*. Clearly, computing any bounded approximation for k-median and k-means and the problem of constructing any coreset with bounded quality are compliant. To get a linear space lower bound for fully-dynamic algorithms that solve a compliant problem, we use the communication problem of INDEX. In INDEX$_N$, Alice's input is an N-bit string and Bob's input is an index $I \in \{1, 2, \ldots, N\}$. Alice sends one message to Bob, and he needs to correctly output the bit at position I. By a well-known communication complexity lower bound, Alice must send a message of size $(1 - H_2(3/4))N \geqslant 2N/11$ bits so that Bob can correctly output with a success probability of $3/4$; here H_2 is the binary entropy function.

▶ **Theorem 19.** *A fully-dynamic algorithm for a compliant problem that works in the presence of an adaptive adversary and has success probability $1 - 1/(8n^2)$ must use space $\Omega(n)$, where n is the current input size.*

Proof. We describe the reduction for any compliant problem in a metric space, such as 1-median or 1-means, but it can be naturally generalized to any compliant problem. Alice defines

$$X = \{j : j\text{th bit in her string } = 1\},$$

and distance between any two points of X to be 1. She runs the fully-dynamic algorithm on X and sends the memory snapshot to Bob. Bob queries for a solution and if X is nonempty, a nonempty solution S_1 would be returned. He deletes the points in S_1 and queries again to get S_2, and so on until $\emptyset$ is returned. There would be at most N such queries. Note that

this works because the algorithm works under an adaptive adversary. If one of the S_ℓs in this process contains I, which is Bob's input for the index problem, then Bob outputs 1, else he outputs 0. In the worst case, Bob makes N queries, where query number i would have failure probability at most $1/(8(N - i + 1)^2)$. So overall failure probability by the union bound is at most

$$\sum_{i=1}^{N} \frac{1}{8(N - i + 1)^2} \leqslant \frac{1}{8} \sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{1}{8} \frac{\pi^2}{6} \leqslant \frac{1}{4} .$$

Alice communicated as many bits as the space usage of the dynamic algorithm. Then, by the INDEX$_N$ lower bound, the space usage of the algorithm is at least $2N/11 \geqslant 2n/11$ bits.   ◄

4.2   Conditional Lower Bounds on the Time Per Operation

Now, we show conditional lower bounds on the time per update and query for fully-dynamic k-means algorithms. They are based on the *OMv-conjecture* [18]: You are given an $N \times N$ Boolean matrix M that can be preprocessed in polynomial time. Then, an online sequence of N-dimensional Boolean vectors $v^1, \ldots, v^N$ is presented and the task is to compute each Mv^i (using Boolean matrix-vector multiplication) before seeing the next vector v^{i+1}. The conjecture is that finding all the N answers takes time $\Omega(N^{3-\gamma})$ for any constant $\gamma > 0$. In [18] also the following *OuMv problem* was presented: You are given an $N \times N$ Boolean matrix M that can be preprocessed in polynomial time and an online sequence of Boolean vector pairs $(u^1, v^1), \ldots, (u^N, v^N)$ with the goal to compute each $(u^i)^T M v^i$ (using Boolean matrix-vector multiplication) before seeing the next vector pair (u^{i+1}, v^{i+1}). Under the OMv conjecture, finding N answers for the OuMv problem such that the error probability is at most $1/3$ takes time $\Omega(N^{3-\gamma})$ for any constant $\gamma > 0$. We will show a reduction from the latter problem to prove the following result.

▶ **Theorem 20.** *Let $\gamma > 0$ be a constant. Under the OMv conjecture, for any $\delta > 0$, there does not exist a fully-dynamic algorithm that maintains a $(4 - \delta)$-approximation for k-means with amortized update time $O(k^{1-\gamma})$ and query time $O(k^{2-\gamma})$ such that over a polynomial number of updates the error probability is at most $1/3$.*

Proof. For the ease of presentation, we assume that k is even; if k is odd, the construction can be easily adapted. We set $N = k/2$. Given an OuMv instance with $N \times N$ matrix M, we construct the following metric space with distance function d from it:

The metric space U consists of $4N$ points numbered from 1 to $4N$. For any $1 \leqslant i < j \leqslant N$ and $N + 1 \leqslant i < j \leqslant 2N$, the distance $d(i, j) = 2$. Furthermore, for $1 \leqslant i \leqslant N$ and $N + 1 \leqslant j \leqslant 2N$, the distance $d(i, j) = 1$ if $M_{i,j-N} = 1$, and $d(i, j) = 2$ otherwise. Additionally, all $2N$ points $2N + 1, \ldots, 4N$ are at distance 100 from each other and from all the other points.

We use a k-means data structure to solve a $u^T M v$ computation as follows: Initially the set X is empty. When given a vector pair (u, v), let p be the number of ones in v and in u. Note that $p \leqslant 2N$. We insert the points i such that $u_i = 1$ and the points j such that $v_{j-N} = 1$ into X and additionally $2N + 1 - p$ of the points ℓ with $\ell > 2N$. Thus $|X| = 2N + 1 = k + 1$. Then we ask a k-means query. Afterwards, we delete the inserted points.

If $u^T M v = 1$, then there exist indices i and j such that $u_i = 1, M_{i,j} = 1$, and $v_j = 1$. Consider the optimal solution that consists of all points in X except for point i. Note that the cost of this solution for the k-means problem is 1.

If $u^T M v = 0$, then any optimal solution must also consist of $2N + 1 - p$ of the points ℓ with $\ell > 2N$, and all but one of the other points in X. But as none of the points in X has distance smaller than 2 to any other point in X, the cost of the solution is at least 4 for k-means. Thus, any $(4 - \delta)$-approximation for k-means can distinguish between the cases $u^T M v = 1$ and $u^T M v = 0$. Hence, the OMv conjecture implies that it takes at least time $\Omega(N^{2-\gamma})$ time to execute the above $2N$ update operations and 1 query operation. This implies the claimed lower bound. ◄

── **References** ──

1 Sara Ahmadian, Ashkan Norouzi-Fard, Ola Svensson, and Justin Ward. Better guarantees for k-means and euclidean k-median by primal-dual algorithms. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 61–72. IEEE Computer Society, 2017.

2 Vijay Arya, Naveen Garg, Rohit Khandekar, Adam Meyerson, Kamesh Munagala, and Vinayaka Pandit. Local search heuristics for k-median and facility location problems. *SIAM J. Comput.*, 33(3):544–562, 2004.

3 Jon Louis Bentley and James B. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *J. Algorithms*, pages 301–358, 1980.

4 Vladimir Braverman, Dan Feldman, and Harry Lang. New Frameworks for Offline and Streaming Coreset Constructions. *arXiv e-prints*, 2016. `arXiv:1612.00889`.

5 Vladimir Braverman, Gereon Frahling, Harry Lang, Christian Sohler, and Lin F. Yang. Clustering high dimensional dynamic data streams. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 576–585, International Convention Centre, Sydney, Australia, August 06–11 2017. PMLR.

6 Jaroslaw Byrka, Thomas W. Pensyl, Bartosz Rybicki, Aravind Srinivasan, and Khoa Trinh. An improved approximation for *k*-median and positive correlation in budgeted optimization. *ACM Trans. Algorithms*, 13(2):23:1–23:31, 2017.

7 Moses Charikar and Sudipto Guha. Improved combinatorial algorithms for facility location problems. *SIAM J. Comput.*, 34(4):803–824, 2005.

8 Moses Charikar, Sudipto Guha, Éva Tardos, and David B. Shmoys. A constant-factor approximation algorithm for the k-median problem. *J. Comput. Syst. Sci.*, 65(1):129–149, 2002.

9 Ke Chen. On coresets for k-median and k-means clustering in metric and euclidean spaces and their applications. *SIAM Journal on Computing*, 39(3):923–947, 2009. `doi:10.1137/070699007`.

10 Vincent Cohen-Addad, Niklas Oskar D Hjuler, Nikos Parotsidis, David Saulpic, and Chris Schwiegelshohn. Fully dynamic consistent facility location. In *Advances in Neural Information Processing Systems 32*, pages 3255–3265. Curran Associates, Inc., 2019. URL: `http://papers.nips.cc/paper/8588-fully-dynamic-consistent-facility-location.pdf`.

11 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997.

12 Dan Feldman and Michael Langberg. A unified framework for approximating and clustering data. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 569–578, 2011.

13 Dan Feldman, Melanie Schmidt, and Christian Sohler. Turning big data into tiny data: Constant-size coresets for *k*-means, PCA and projective clustering. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1434–1453. SIAM, 2013.

14   Gereon Frahling and Christian Sohler. Coresets in dynamic geometric data streams. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 209–217. ACM, 2005.

15   Anupam Gupta and Kanat Tangwongsan. Simpler analyses of local search algorithms for facility location, 2008. `arXiv:0809.2554`.

16   Sariel Har-Peled and Akash Kushal. Smaller coresets for k-median and k-means clustering. *Discret. Comput. Geom.*, 37(1):3–19, 2007.

17   Sariel Har-Peled and Soham Mazumdar. On coresets for k-means and k-median clustering. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '04, page 291–300. Association for Computing Machinery, 2004.

18   Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Symposium on Theory of Computing (STOC)*, pages 21–30, 2015. `doi:10.1145/2746539.2746609`.

19   Kamal Jain, Mohammad Mahdian, and Amin Saberi. A new greedy approach for facility location problems. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 731–740. ACM, 2002.

20   Kamal Jain and Vijay V. Vazirani. Approximation algorithms for metric facility location and *k*-median problems using the primal-dual schema and lagrangian relaxation. *J. ACM*, 48(2):274–296, 2001.

21   Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. A local search approximation algorithm for k-means clustering. *Comput. Geom.*, 28(2-3):89–112, 2004.

22   Shi Li and Ola Svensson. Approximating k-median via pseudo-approximation. *SIAM J. Comput.*, 45(2):530–547, 2016.

23   Ramgopal R. Mettu and C. Greg Plaxton. Optimal time bounds for approximate clustering. *Mach. Learn.*, 56(1-3):35–60, 2004.

24   Christian Sohler and David P. Woodruff. Strong coresets for k-median and subspace approximation: Goodbye dimension. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 802–813. IEEE Computer Society, 2018.

A   Proof of Lemma 8

▶ **Lemma 8.** *For any positive integer ℓ and $\alpha \in \mathbb{R}_+$, we have*

$$\alpha + (1 + \alpha) \sum_{i=1}^{\ell-1} \binom{\ell-1}{i} \alpha^i = \sum_{i=1}^{\ell} \binom{\ell}{i} \alpha^i \, .$$

Proof.

$$\alpha + (1 + \alpha) \sum_{i=1}^{\ell-1} \binom{\ell-1}{i} \alpha^i = \alpha + \sum_{i=1}^{\ell-1} \binom{\ell-1}{i} \alpha^i + \sum_{i=1}^{\ell-1} \binom{\ell-1}{i} \alpha^{i+1}$$

$$= \binom{\ell-1}{0} \alpha + \sum_{i=1}^{\ell-1} \binom{\ell-1}{i} \alpha^i + \sum_{i=1}^{\ell-1} \binom{\ell-1}{i} \alpha^{i+1}$$

$$\text{using the fact } \binom{\ell-1}{0} = 1$$

$$= \binom{\ell-1}{0} \alpha + \sum_{i=1}^{\ell-1} \binom{\ell-1}{i} \alpha^i + \sum_{i=2}^{\ell} \binom{\ell-1}{i-1} \alpha^i$$

$$\text{change of index in the second summation}$$

$$= \sum_{i=1}^{\ell-1} \binom{\ell-1}{i} \alpha^i + \sum_{i=1}^{\ell} \binom{\ell-1}{i-1} \alpha^i$$

incorporating first term in second summation

$$= \sum_{i=1}^{\ell} \binom{\ell-1}{i} \alpha^i + \sum_{i=1}^{\ell} \binom{\ell-1}{i-1} \alpha^i$$

using the fact $\binom{\ell-1}{\ell} = 0$

$$= \sum_{i=1}^{\ell} \left(\binom{\ell-1}{i} + \binom{\ell-1}{i-1} \right) \alpha^i$$

$$= \sum_{i=1}^{\ell} \binom{\ell}{i} \alpha^i \,,$$

where we use $\binom{\ell}{i} = \binom{\ell-1}{i} + \binom{\ell-1}{i-1}$ in the last step. ◀